Natural Gradient Algorithms for Training Deep Neural Networks



Constantin Octavian Puiu

St Anne's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity Term 2023

To my grandmother, *Lazar Maria*,

You were the towering oak, the steadfast root of our family tree, safeguarding the wisdom of ages with unwavering resolve. Your kindness and care, triumphant harp tones filling our hearts, and your strength a fortification thereof.

Your spirit, now a resplendent star, enlightens our journey and carves our path through life's rocky challenges. A whisper borne onto the winds of time arrives, reminding us we necessarily venture onwards. Your legacy, woven into the very fabric of our destinies, leads us toward boundless horizons, as we march under your watchful, benevolent gaze.

To my *wife, mother, father, brother, and family.*

With love and gratitude,
Constantin

Acknowledgements

The EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in partnership with Numerical Algorithms Group Ltd (NAG) and St Annes provided vital financial support for this research. The invaluable efforts of Prof. Chris Breward and Prof. Colin Peter Please were indispensable in making this support possible. Additionally, I would like to express my gratitude for the utilization of the University of Oxford Advanced Research Computing facility (in particular the HTC cluster) in the execution of the numerical parts of this study.

I want to thank my supervisor, Prof. Coralia Cartis, for their unwavering enthusiasm, valuable guidance, unconditional support, and for motivating me to leave no stone unturned in my research. I would also like to thank my industrial supervisors from NAG, Jan Fiala and Shuanghua Bai, for their industrial insights.

I would also like to express my gratitude to my dear friend Jaroslav Fowkes for his help and insightful mathematical discussions, which significantly enriched my Ph.D. journey. His presence was a source of great encouragement, and I'm truly thankful for his support. I would further like to thank Dr. Yuji Nakatsukasa for useful discussions.

I express my heartfelt gratitude to my dear wife, for her unwavering support, which was the cornerstone of my Ph.D. journey support system. My mother, brother and father provided invaluable belief and fortification. My family and friends, your encouragement was crucial. I am truly fortunate to have such a wonderful support system, and I could not have achieved this without you all.

Abstract

In this thesis, we reduce the computation cost of a particular class of learning algorithms for Deep Neural Networks (DNNs; human-brain inspired artificial intelligence models). Natural Gradient Descent (NGD) algorithms have favorable properties as compared to Stochastic Gradient methods. K-FAC (Martens and Grosse, 2015) is a feasible implementation of NGD for DNNs which retains some desirable properties of NGD. Still, K-FAC has cubic time-complexity in layer size, due to the computation of Kronecker factors (K-factors) inverses. This results in slow run-times for large K-factors. The thesis solves the time-complexity issue in two steps.

First, we propose low-rank inverse representations of K-factors, obtainable in quadratic-time with randomized Numerical Linear Algebra (NLA). Our theory shows that at least for sufficiently large layers, we can also control the K-factors representation error, owing to the exponentially averaged (EA) construction paradigm. Numerical results show that theory is pessimistic: the error can be controlled better in practical settings. By improving the inverse-application subroutine to scale quadratically as well, we propose two quadratic time-complexity K-FAC variants: R-KFAC and SRE-KFAC.

Secondly, by using Brand's algorithm for online NLA to exploit the EA construction paradigm of K-factors, we propose to maintain online low-rank inverse representations of K-factors, which only takes linear time. The approach is limited to sufficiently large layers, which in practice is typically satisfied by fully connected layers. By further improving the inverse-application subroutine to scale linearly as well in this case, we propose a linear time-complexity K-FAC variant: B-KFAC. B-KFAC also has linear memory-complexity in layer size, unlike quadratic for K-FAC. We also propose to algorithmically control the B-KFAC representation error, at the price of having quadratic time costs on some iterations (most iterations still take linear time). This yields three other algorithms.

Our improved-complexity K-FAC variants speed up K-FAC 3 - 4 times for most investigated settings, in both single-GPU and distributed settings.

Contents

1 Introduction **1**

1.1 Deep Neural Networks . . . 3

1.1.1 Fully Connected Nets and MLE Supervised Learning . . . 5

1.2 Training DNNs . . . 6

1.2.1 Are DNN Optimizers specific to the ML task? . . . 6

1.2.2 First Order Methods and Backprop . . . 7

1.2.3 Stochastic Gradient Descent (SGD) . . . 7

1.2.4 Adaptive Momentum (ADAM) . . . 8

1.3 Training DNNs: Second Order Methods . . . 8

1.3.1 A Short Literature Survey of Second Order Approaches . . . 9

1.4 Natural Gradient Methods . . . 13

1.4.1 The Fisher Information Matrix . . . 13

1.4.2 KL Divergence Objectives . . . 14

1.4.3 The Natural Gradient Descent Method . . . 15

1.4.4 Empirical Fisher and Empirical Natural Gradient . . . 15

1.5 Kronecker Factored Approximate Curvature . . . 16

1.6 Motivation, Contributions, and Thesis Structure . . . 19

1.6.1 Research Questions and Contributions . . . 25

1.6.2 Structure of Thesis and Contributions Summary . . . 31

1.6.3 Points to Note . . . 32

2 Technical Background **33**

2.1 Numerical Linear Algebra . . . 33

2.1.1 The QR Factorization . . . 33

2.1.2 The Eigenvalue Decomposition (EVD) . . . 33

2.1.3 The Singular Value Decomposition (SVD) . . . 34

2.2 Randomized Numerical Linear Algebra . . . 35

2.2.1 Randomized SVD (RSVD) . . . 36

2.2.2 Randomized Symmetric EVD (SREVD) . . . 38

2.2.3 Randomized NLA: Projection Error Theory 38
2.3 Online Numerical Linear Algebra 40
2.3.1 Brand's Algorithm (2006) 40

3 Randomised K-FAC algorithms 43
3.1 Motivation: Exponential Eigenspectrum Decay 46
3.1.1 Deterministic Bounds for K-Factor Eigendecay 50
3.1.2 Probabilistic Bounds for K-Factor Eigendecay 55
3.1.3 Connecting Theory to Practice 64
3.1.3.1 Fixed Low-Rank M_i allows better Inversion Complexity 68
3.1.3.2 Theoretical Case-Studies: Is better Complexity theoretically expected to give Speed-ups in Practical Settings? 70
3.1.3.3 Numerical Case-Study: Do Practical Settings empirically allow for Computational Savings? 74
3.1.4 The effects of Directly Obtaining the Target Modes 80
3.2 Proposed Optimization Algorithms 81
3.2.1 RSVD K-FAC (R-KFAC) 82
3.2.2 SREVD K-FAC (SRE-KFAC) 84
3.2.3 How do we Choose the Number of Modes r? 84
3.2.4 rNLA Projection Error: Choice of $n_{\text{pwr-it}}$ and r_o 86
3.2.5 Further Considerations of Randomized KFACs 89
3.3 Numerical Performance of Proposed Optimizers 90
3.3.1 Implementation Details 91
3.3.2 Results Discussion 92
3.4 Discussion 94
3.4.1 Future Work 95
3.4.2 Looking Ahead 95

4 Brand New K-FACs 97
4.1 Proposed Optimizers 100
4.1.1 The B-KFAC Algorithm 100
4.1.2 Motivation for Resetting B-updated $\tilde{\mathcal{M}}_k^{(B)}$ with an RSVD . . . 104
4.1.3 The BR-KFAC Algorithm 108
4.1.4 The BC-KFAC Algorithm: A Cheaper Correction 109
4.1.5 The BRC-KFAC Algorithm: A mixture of BR and BC 112
4.1.6 Important Practical Considerations 113
4.2 Investigation of the K-Factor Estimation Error 114

4.2.1 Theoretical Error Comparison: R-KFAC vs BR-KFAC 114
4.2.2 Numerical Error Investigation: Experimental Set-up 117
4.2.3 Numerical Error Investigation: Results 118
4.3 Proposing a Linear-time Inverse Application 123
4.4 Overview of the Proposed Algorithms 125
4.5 Numerical Performance of Proposed Optimizers 126
4.5.1 Algorithms Optimization Performance Comparison 128
4.6 Discussion 129
4.6.1 Future Work 132

5 Distributed Implementations of our Proposed K-FAC Variants 133
5.1 Our K-factor Computation Timing Choice 134
5.1.1 Simplification: Mentally Re-Shuffling K-factor Workload . . . 135
5.2 Related Distributed Implementations of K-FAC 137
5.2.1 K-Factor Computation: Choice of Distributing Strategy 140
5.2.2 K-Factor Inversion and Preconditioning Choice 142
5.3 Our Distributed K-FACs Design 143
5.3.1 Efficient Allocation of K-Factors Workload 144
5.3.2 Adaptive Target-Rank Strategies 146
5.4 Optimizer Specific Details of Distributed Design 147
5.4.1 Benchmark Algorithms: Distributed Design Details 147
5.4.2 R-KFAC: Distributed Design Details 147
5.4.3 B-KFAC: Distributed Design Details 148
5.4.4 BR-KFAC: Distributed Design Details 151
5.4.5 BRC-KFAC: Distributed Design Details 153
5.5 Numerical Experiments: Set-up 155
5.5.1 Datasets, Neural Networks, and GPU Specifications 155
5.5.2 Hyperparameters Tuning Methodology 156
5.5.3 Hyperparameters Choices Constant for all Experiments 157
5.5.4 Dataset-Dependent Hyperparameters Choices 159
5.6 Numerical Experiments: Results 160
5.6.1 Summary of Results 160
5.6.2 Average of Convergence Plots 162
5.6.3 Scaling Efficiency with Number of GPUs 164
5.6.3.1 Per-Step Time 166
5.6.3.2 Number of Steps (and Epochs) to Target Accuracy . 171

5.6.3.3 Time to Target Test Accuracy . . . 172
5.6.4 Comparison between Different Proposed Variants . . . 174
5.7 Discussion . . . 175
5.7.1 Future Work . . . 177

6 Conclusions and Future Work **179**
Future Directions . . . 186

Bibliography **188**

A2 Appendix of Chapter 2 **197**
A2.1 First Order Methods: Technical Details . . . 197
A2.1.1 ADAM: Technical Details . . . 197
A2.2 Natural Gradient Technical Details . . . 198
A2.2.1 The Fisher: Links with other Curvature Matrices . . . 199
A2.2.2 Interpretations and Advantages of NG . . . 201
A2.2.3 Convergence Properties of (True) NGD . . . 203
A2.2.4 Discussion of Empirical NG vs True NG . . . 211
A2.2.5 Empirical NG Minimizes the Generalization Error . . . 212

A3 Appendix of Chapter 3 **214**
A3.1 Bounds avoiding the Low-rank Assumption . . . 214
A3.2 The bounded $\sigma_{\max}(M_i)$ $\forall i$ Assumption of *S3.1.1* . . . 235
A3.3 Observing Low-rank M_i Assumption . . . 236
A3.4 More Plots for Numerical Case-study in *S3.1.3.3* . . . 239
A3.5 Caveats of our *S3.1* Theory . . . 240
A3.5.1 Practically Implied N_ϵ Values are Much Smaller . . . 240
A3.5.2 Practical $N_\epsilon = f(\epsilon)$ Function: Comparison with Theory . . . 241
A3.6 Numerical Investigation of rNLA Projection Error . . . 243
A3.6.1 Numerical Results: RSVD Projection Error . . . 244
A3.6.2 Numerical Results: SREVD Projection Error . . . 245

A4 Appendix of Chapter 4 **247**
A4.1 Proof of Example 4.1 . . . 248
A4.2 Proof of Proposition 4.3 . . . 251
A4.3 Proof of Proposition 4.5 . . . 255
A4.4 Proof of Proposition 4.6 . . . 257

A5Literature Review: Practical NG Algorithms for DNNs 260
A5.1 Base Case: Non-Problem-Specific 260
A5.1.1 Practical Empirical NG Method: TONGA 261
A5.1.2 ADAM as Empirical NG-like Algorithm 265
A5.1.3 Hessian-Free (Martens 2010) 265
A5.2 Supervised Learning: Fully Connected Layers 266
A5.2.1 K-FAC (for FC layers) 266
A5.2.2 PRONG 271
A5.3 Supervised Learning: Convolutional Layers 273
A5.3.1 Convolutional K-FAC: KFC 273
A5.3.2 SENG (Sketchy Empirical Natural Gradient) 276
A5.3.3 Scalable Practical NGD (SP-NGD) 280

List of Figures

1.1 The dependencies of an Optimizer Implementation (for a fixed idea). 7

3.1 Dissecting behavior of rank(M_i) vs d_M, #Modes to keep vs d_M, and showing Potential Inversion Scaling Improvements. 69
3.2 #Modes required for 3.3% relative truncation error: Numerical Results 76
3.3 CIFAR10 classification with VGG16_bn: Test Loss and Accuracy results. 93

4.1 B-KFAC: Excess Error Factor and Associated Probability for our Particular (Heavy-tailed M_j distribution) 2D Example. 106
4.2 Error in inverse K-Factors for Different Inverse Representations . . . 119
4.3 Error in Step for Different K-factors Inverse Representations. 120

5.1 Mentally Re-shuffling K-factor work:. An example with 2 GPUs and 2 layer networks. 136
5.2 Summary of possible distributed implementation choices for K-FAC: Decentralized Architecture only. 138
5.3 EVD, RSVD and B-update: Scaling comparison. 145
5.4 Distributed B-KFAC work allocation. 149
5.5 Distributed B-KFAC algorithm workings at different iteration indices k. 150

5.6 Distributed BR-KFAC work allocation. 152
5.7 Distributed BR-KFAC algorithm workings at different iteration indices k. 153
5.8 Numerical Results: Average of Convergence Plots when using 1 GPU. 162
5.9 Numerical Results: Average of Convergence Plots when using 2 GPUs. 163
5.10 Numerical Results: Average of Convergence Plots when using 3 GPUs. 164

A3.1 Numerical Case-study: Evolution of $\sigma_{\max}(M_i)$ with iterations (k) for all K-factors. 235
A3.2 The K-Factors eigenspectrum decay: Numerical Assumption testing for FC layers 237
A3.3 The K-Factors eigenspectrum decay: Numerical Assumption testing for Conv layers 238
A3.4 #Modes required for 0.33% relative truncation error: Numerical Results 239
A3.5 #Modes required for 1% relative truncation error: Numerical Results 239
A3.6 #Modes required for 5% relative truncation error: Numerical Results 240
A3.7 Empirically Implied N_ϵ as a function of $\ln \epsilon$ 242
A3.8 CIFAR10 with VGG11_bn Case-study: Projection error of RSVD for $\bar{\mathcal{A}}_k^{(l)}$ K-factors. 244
A3.9 CIFAR10 with VGG11_bn Case-study: Projection error of RSVD for $\bar{\Gamma}_k^{(l)}$ K-factors. 245
A3.10 CIFAR10 with VGG11_bn Case-study: Projection error of SREVD for $\bar{\Gamma}_k^{(l)}$ K-factors. 245
A3.11 CIFAR10 with VGG11_bn Case-study: Projection error of SREVD for $\bar{\mathcal{A}}_k^{(l)}$ K-factors. 246

List of Tables

A Table of Notation: Covers the most important variables. viii

3.1 CIFAR10 with VGG11: K-factors shapes and $N_\epsilon r_M$ values predicted by theory 71
3.2 CIFAR10 with VGG16: K-factors shapes and $N_\epsilon r_M$ values predicted by theory 71

3.3 Imagenet (input randomly cropped to 224×224 pixels) with resnet34: K-factors shapes and $N_\epsilon r_M$ values predicted by theory 72
3.4 Empirically Implied N_ϵ for $\epsilon = 0.033$ for CIFAR10 with VGG11-bn . 75
3.5 CIFAR10 classification with VGG16_bn: results summary of optimizers performance 92

4.1 Average Error Metrics of Proposed Algorithms and Benchmark 122
4.2 Summary of Proposed Algorithms and Benchmark 126
4.3 Optimizer Performance Results Summary. 128

5.1 Distributed Experiments Learning Rate Parameters. 159
5.2 Numerical Performance of Proposed Distributed Algorithms and Comparison with Benchmarks: Summary 161
5.3 Scaling efficiency for CIFAR10 dataset with VGG16_bn_lmxp Neural Network 167
5.4 Scaling efficiency for CIFAR100 dataset with VGG16_bn_lmxp Neural Network 168
5.5 Scaling efficiency for SVHN dataset with VGG16_bn_lmxp Neural Network 169
5.6 Scaling efficiency for Imagenette (full-size, v2) dataset with resnet50 Neural Network 170
5.7 Distributed Results Summary: Counting Wins 175

Table A. Table of Notation: Covers the most important variables.

N_D	Number of points in the dataset.
$\{(x_i, y_i)\}_i^N$	(Train) Dataset at hand.
$\mathcal{D}_{\text{train}}$	Alias for $\{(x_i, y_i)\}_i^N$.
x, y	Input to Neural Net, and Target of Neural Net respectively.
$\mathcal{X}$, $\mathcal{Y}$	Input space and Target space respectively.
θ, Θ	Neural Net Parameters and Parameter Space respectively.
$h(x;\theta)$	Neural Net output.
l	Indexes layers.
k	Indexes optimization iterations.
i	More generic index, typically indexes over data-points but may sometimes be an alternative index for iterations when k is already used, a further iteration index is required, and there is clearly no reference to different data-points in the context.
j	Generic index, context dependent.
a_l	Activation at layer l.
$\bar{a}_l$	Augmented activation at layer l ($\bar{a}_l = [a_l^T, 1]^T$).
z_l	Pre-activation at layer l.
W_l	Weight Matrix of layer l.
w_l	Width of layer l (applies to fully-connected layers).
n_L	Number of Layers.
$\mathcal{I}_{\text{B}}$	A subset of $\{1, 2, ..., N_D\}$. Represents chosen data-points index set in mini-batch.
n_{B}	Batch size used to estimate the gradient: $n_{\text{B}} = \lvert\mathcal{I}_{\text{B}}\rvert$.
$n_{\text{B,K}}$	Batch size used to estimate the K-factors, typically $n_{\text{B,K}} \leq n_{\text{B}}$ holds.
$\ell(z, y)$	Loss function.
q, $\hat{q}$	True Data Distribution and Empirical Data Distribution respectively. Sometimes q is also used as an index (eg. in *Section 3.1.2*), but notably there shall be no confusion in those rare cases, as the data distribution is not involved in any way in that context.
$p(x, y;\theta)$	The generative model distribution for x, y, induced by the DNN of parameters θ.
$p(y\|x;\theta)$	Conditional distribution of model output (of DNN with parameters θ).
$\curlywedge(y\|h(x;\theta))$	A redefinition of $p(y\|x;\theta)$ which more clearly highlights how θ contributes to the $p(y\|x;\theta)$ conditional: $p(y\|x;\theta) = \curlywedge(y\|h(x;\theta))$.
$p(x;\theta)$	The generative marginal model distribution of x, for DNN parameters θ.
$L(x, y;\theta)$	Redefinition of the loss for convenience $L(x, y;\theta) := \ell(h(x;\theta), y)$.
$\mathcal{R}(\theta)$	(True) Risk $\mathcal{R}(\theta) := \mathbb{E}_{(x,y)\sim q}[L(x, y;\theta)]$.
$\tilde{\mathcal{R}}(\theta)$	Empirical Risk $\tilde{\mathcal{R}}(\theta) := \mathbb{E}_{(x,y)\sim\hat{q}}[L(x, y;\theta)] = \frac{1}{N}\sum_{i=1}^N L(x_i, y_i;\theta)$.
δ_l	Gradient of Loss w.r.t. preactivation z_l: $\delta_l := \nabla_{z_l} L$.
$g(\theta)$	Gradient of the true risk w.r.t. θ: $g(\theta) := \nabla_\theta \mathcal{R}(\theta)$.
$g_i(\theta)$	Per-example gradient (1-datapoint gradient estimator): $g_i(\theta) := \nabla_\theta L(x_i, y_i;\theta)$.
$g_{\text{B}}(\theta)$	Mini-batch gradient estimator: $g_B(\theta) := \frac{1}{\lvert\mathcal{I}_B\rvert}\sum_{i\in\mathcal{I}_{\text{B}}} \nabla_\theta L(x_i, y_i;\theta)$.
$\tilde{g}_k$	Shorthand for $g_{\text{B}}(\theta_k)$, the batch-gradient estimate at θ_k.
α_k, α	α_k is the learning rate at iteration k. Using α if fixed over iterations.
$F(\theta)$	The Fisher information matrix: $F(\theta) := \mathbb{E}_{(x,y)\sim p}\left[\nabla_\theta \log p(x, y;\theta) \nabla_\theta \log p(x, y;\theta)^T\right]$.
F_k	Shorthand for $F(\theta_k)$.

$F^{(E)}$	A superscript of (E) to the Fisher denotes the empirical Fisher, which is defined as $F^{(E)}(\theta) := \mathbb{E}_{(x,y)\sim q}\big[\nabla_\theta \log p(x,y;\theta) \nabla_\theta \log p(x,y;\theta)^T\big]$. Here empirical refers to taking the expectation w.r.t. q rather than p (and not to data-estimation as typical).
$\tilde{F}^{(E)}$	The data-estimated $F^{(E)}$.
$\mathscr{C}$	Covariance Matrices.
C, $\mathcal{C}$	Refer to constants.
Σ	Diagonal matrix of singular values (of matrix specific to context).
σ_i	The i^{th} singular value, typically sorted descendingly, but not always.
Λ or D	Diagonal matrix of eigenvalues (of matrix specific to context).
λ	λ is mainly used for eigenvalues. However, we also use λ to denote *shifts* in eigenvalues: as is the case when we use λ to denote the regularization (damping) parameter (of the standard $+\lambda I$ regularization approach of second order methods).
λ_i	The i^{th} eigenvalue, sorted descendingly.
$\lambda_{\max}$, $\lambda_{\min}$	$\lambda_{\max}(X)$ and $\lambda_{\min}(X)$ are then λ_1 and λ_d respectively, of (generic) matrix $X \in \mathbb{R}^{d\times d}$.
$\mathscr{D}$	Refers to a probability distribution.
$\mathsf{D}(\cdot)$	Refers to the derivative of the loss w.r.t. $(\cdot)$. Notation only appears in the appendix.
Q, U, V	Orthogonal matrices typically as part of SVD, EVD, or QR decomposition.
I, O, H, J	Identity, the Zero-matrix, Hessian, and Jacobian respectively.
X	Generic matrix.
R	The R matrix in some QR decomposition.
$\tilde{F}^{(\text{KFAC})}$	K-FAC approximation to the Fisher: $F_k^{(\text{KFAC})} := \text{blockdiag}\big(\{\mathcal{A}_k^{(l)} \otimes \Gamma_k^{(l)}\}_{l=1,\dots,n_L}$.
$\mathcal{A}_k^{(l)}$	*Forward K-factor* of layer l at iteration k: $\mathcal{A}_k^{(l)} := A_k^{(l)}[A_k^{(l)}]^T \in \mathbb{R}^{d_{\mathcal{A}}^{(l)} \times d_{\mathcal{A}}^{(l)}}$.
$\Gamma_k^{(l)}$	*Backward K-factor* of layer l at iteration k: $\Gamma_k^{(l)} := G_k^{(l)}[G_k^{(l)}]^T \in \mathbb{R}^{d_{\Gamma}^{(l)} \times d_{\Gamma}^{(l)}}$.
$A_k^{(l)}, G_k^{(l)}$	Matrices out of which K-factors are constructed directly. The former is based on quantities in the forward pass, the latter to quantities in the backward pass ($A_k^{(l)} \in \mathbb{R}^{d_{\mathcal{A}}^{(l)} \times n_{\mathcal{A}}^{(l)}}$ and $G_k^{(l)} \in \mathbb{R}^{d_{\Gamma}^{(l)} \times n_{\Gamma}^{(l)}}$).
$\bar{\mathcal{A}}_k^{(l)}$, $\bar{\Gamma}_k^{(l)}$	Exponentially Averaged (EA) versions of $\mathcal{A}_k^{(l)}$ and $\Gamma_k^{(l)}$ respectively.
$d_{\mathcal{A}}^{(l)}$, $d_{\Gamma}^{(l)}$	First dimension of *forward* and *backward* K-factors respectively.
$n_{\mathcal{A}}^{(l)}$, $n_{\Gamma}^{(l)}$	Second dimension of *forward* and *backward* K-factors respectively.
$\eta^{(l)}$, $d_w^{(l)}$, $d_h^{(l)}$	The #output-channels, width, and height (respectively) of the data tensor at layer l. Specific to Convolutional layers (Conv-layers).
$\kappa_w^{(l)}$, $\kappa_h^{(l)}$	The convolution kernel width and height at layer l. Specific to Conv-layers.
ρ	Exponential Average parameter ($\rho \in (0,1)$, higher means slower decay).
$\bar{\mathcal{M}}_k$	Generic Exponentially Averaged (EA) K-factor (could be any one in the set $\{\bar{\mathcal{A}}_k^{(l)}, \bar{\Gamma}_k^{(l)}\}_l$). $\bar{\mathcal{M}}_k$ is of generic size $d_M \times d_M$
M_k	Generic matrix in the set $\{A_k^{(l)}, G_k^{(l)}\}_l$. We typically use this as M_i where i now indexes over iterations because k is taken as an iteration index in the context M_i is used (a rare case of using i to index iterations). M_i is of generic size $d_M \times n_M$.
d_M	The first dimension of our generic EA K-factor $\bar{\mathcal{M}}_k$. Note the relation to $d_{\mathcal{A}}^{(l)}$, $d_{\Gamma}^{(l)}$.
n_M	The second dimension of our generic M_k. Note the relation to $n_{\mathcal{A}}^{(l)}$, $n_{\Gamma}^{(l)}$.
d	Generic matrix dimension (mainly used for linear algebra arguments in *Chapter 2*). Typically, d becomes d_M (or the more specific $d_{\mathcal{A}}^{(l)}$, $d_{\Gamma}^{(l)}$) when the generic matrix is pinned down to be some K-FAC related quantity.
d_θ	Dimension of parameter space. Notably $d_\theta \approx \sum_{l=0}^{n_L-1} d_{\mathcal{A}}^{(l)} d_{\mathcal{A}}^{(l+1)} + \sum_{l=1}^{n_L} d_{\Gamma}^{(l-1)} d_{\Gamma}^{(l)}$.

m	Generic matrix dimension, context dependent.
s, s_k	Optimization step (generic). A k subscript indicates a step at iteration k.
$s^{(\text{NG})}$	Natural Gradient Step.
r	Rank (most of the times is some target rank of an algorithm). Also used as an index related to rank in some way.
r_o	Rank over-sampling parameter of randomized NLA algorithms.
$n_{\text{pwr-it}}$	Number of power iterations in a randomized NLA algorithm.
r_M	Used in *Chapter 3* as a fixed upper-bound for the rank of M_i's.
ϵ	Highlighting a (ideally) small quantity or an error. In our theory of *Sections 3.1.1, 3.1.2, A3.1*, and in the related explanations elsewhere, ϵ is used to refer to our relative truncation error (obtained by truncating the K-factors).
N	Used in our theory as a positive integer parameter of our eigenspectrum decay bounds of the form $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_1(\bar{\mathcal{M}}_k)} \leq A\rho^N + B$ for some constants A and B.
N_ϵ	The N (as above) for which $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_1(\bar{\mathcal{M}}_k)} \leq A\rho^N + B \leq \epsilon$. In our theory of *Sections 3.1.1 - 3.1.2* we thus have that $N_\epsilon r_M$ is the number of mods we need to keep s.t. our theory guarantees a relative truncation error below ϵ.
$\mathcal{E}$ or E	A matrix which represents an error-matrix.
$\tilde{\mathcal{M}}_k^{(B)}$	The rank-$(r + n_M)$ B-KFAC estimate of $\bar{\mathcal{M}}_k$ (where r is the target-rank in our B-update). The estimate is updated in an online fashion (see B-process definition (4.3) for details).
$\mathcal{B}_k$	The rank-r B-KFAC estimate of $\bar{\mathcal{M}}_k$ (where r is the target-rank in our B-update). The estimate is updated in an online fashion as the (leading modes) rank-r truncation of $\tilde{\mathcal{M}}_{B,k}$ (see B-process definition (4.3) for details).
$\tilde{\mathcal{M}}_k^{(\text{RSVD},r)}$	The RSVD low-rank estimate of $\bar{\mathcal{M}}_k$ when target rank is r (and rank-oversampling is r_o: omitted as parameter in $\tilde{\mathcal{M}}_k^{(\text{RSVD},r)}$ for convenience): $\tilde{\mathcal{M}}_k^{(\text{RSVD},r)} := \tilde{U}_{\bar{\mathcal{M}},k}^{(l)} \tilde{D}_{\bar{\mathcal{M}},k}^{(l)} (\tilde{U}_{\bar{\mathcal{M}},k}^{(l)})^T = \text{RSVD}(\bar{\mathcal{M}}_k, r, r_o)$.
$\tilde{\mathcal{M}}_k$	An arbitrary rank-r ($r < d_M$) representation of $\bar{\mathcal{M}}_k$. For example it could be (but is not limited to) $\tilde{\mathcal{M}}_k^{(\text{RSVD},r)}$ or $\mathcal{B}_k$. Used exclusively to present the result of *Proposition 4.3*.
$\tilde{\mathcal{M}}_k^{(c)}$	The rank-r representation obtained by applying the correction in *Algorithm 4.3* (the C-update) to $\tilde{\mathcal{M}}_k$. Used exclusively to present an important property of the C-update through *Proposition 4.3* and *Corollary 4.4*.
n_{crc}	The number of modes we correct with a C-update proposed in *Section 4.1.4* .
ϕ_{crc}	The fraction of modes in our low-rank representation that we correct in a C-update: $\phi_{\text{crc}} := n_{\text{crc}}/r$, where r is the target-rank of the B-/R-update.
T_{Cov}	Period of K-factors updates in K-FAC and proposed K-FAC improvements (update when $k\%T_{\text{Cov}} == 0$).
T_{inv}	Period of K-factor inverse recomputation in K-FAC (recompute when $k\%T_{\text{inv}} == 0$). In the case of R-KFAC this represents how often the RSVD-low-rank inverse K-factor representation is recomputed.
T_{BR}	Period of how often the RSVD-low-rank inverse K-factor representation is recomputed in BR-KFAC or BRC-KFAC: recompute when $k\%T_{\text{BR}} == 0$.
T_{Brand}	Period of B-update in B-KFAC and children algorithms (BR-KFAC, BRC-KFAC): B-update when $k\%T_{\text{Brand}} == 0$.
T_{corct}	Period of how often the C-update (proposed in *Section 4.1.4*) is performed: C-update when $k\%T_{\text{corct}} == 0$.

$\mathcal{G}^{(l)}$	Used in *Chapter 5* to represent *the distributed* K-FAC *preconditioned gradient for layer* l, when describing distributed implementations of K-FAC (and of our K-FAC improvements). In *Appendix A5.3.2*, when describing SENG ([20]) $\mathcal{G}$ is used with a different meaning: for plain gradients rather than preconditioned gradients. There is no risk of misunderstanding given clear definition at each location and very different contexts.
#EtTA	**N**umber of **E**pochs **t**o **T**arget **A**ccuracy.
TpE	**T**ime **p**er **E**poch metric (averaged within a run; measured).
TpS	**T**ime **p**er **S**tep metric (averaged within a run; measured).
TtTA	(Measured) **T**ime **t**o **T**arget **A**ccuracy metric (only used for test accuracy).
SE_n([Metric])	(Measured) **S**caling **E**fficiency of (arbitrary metric) [Metric] at n GPUs.
$n_{\text{B/GPU}}$	Batch-size per GPU.

Chapter 1

Introduction

In this thesis, we are concerned with optimization algorithms for Machine Learning systems. In particular, we focus on Natural Gradient Methods for Deep Learning.

Machine Learning (ML) can be loosely described as obtaining computer programs which can mimic (super-)human behaviour in a given task. This is obtained by training on some data, with the hope of generalization on new, unseen data. The process of training involves searching for a "best" function approximator within a function space that we design, based on available resources and knowledge of the problem (task). Often, the function is parameterized, and thus we search in a parameter space.

Deep Learning (DL) is a branch of Machine Learning which uses Deep Neural Networks (DNNs) in place of the required functions approximators. The field of research is rapidly expanding owing to its (very big) success stories: superhuman AI players trained with no expert knowledge for games like Go [2] and even Starcraft [3], having computers model (understand) natural language [4, 5], detecting objects from images [6], and most recently, generating almost perfect new images from text [7].

Where does Optimization come into play? Informally, optimization is the language of training Neural Networks. The training part of any Machine Learning (and thus Deep Learning) task is essentially minimizing an objective function (where further generalization considerations have to be accounted for). Training some of the very successful DL examples required using hundreds of GPUs for hundreds of hours each, giving total costs in the orders of 10^5-10^6 dollars [8]. Thus, developing training algorithms that can obtain the same training result in less computation time can save substantial cost, and also allow researchers to progress faster. Furthermore, obtaining training algorithms which manage to get a better solution to the training problem is also of high value (as this might compensate for using a smaller model/dataset).

The simplest way to train DNNs is to take a small step (iteration) in the parameter space down the gradient of the objective function. This is called (stochastic) **gradient**

descent (GD) and is a derivative-based optimization method. Virtually all methods for optimizing (training) DNNs are derivative-based, leveraging the available automatic-differentiation packages [9].

However, there can be objective-functions for which the geometry is so adverse that GD gives very little progress per iteration. This case of "adverse geometry" is matheatically referred to as ill-conditioning [10]. For this reason, more involved optimization methods attempt to first map the problem in a locally more "convenient" space, and take a gradient step there. This process of mapping is called **preconditioning**, and depending on the preconditioner - it might provide very good theoretical guarantees (at least in simple cases) and substantial practical progress. Nevertheless, in DNNs settings, manipulating (computing, storing, inverting) preconditioners from "classic optimization" is impossible [1, 9].

Thus, a prevalent research direction in optimization for DNNs has been obtaining ways to get the benefit of preconditioning while making computations feasible. By settling for crude approximations of "classical preconditioners" (sometimes very crude), the optimization for DNNs community has made preconditioning possible. There are many such "approximately preconditioned" algorithms in the literature, each with its own advantages/disadvantages profile [1, 11, 12, 13, 15, 16, 17, 18, 19, 20].

In this thesis, we will focus on optimization algorithms which (approximately) use an information-geometry informed preconditioner (the Fisher information matrix). This class of algorithms is called **Natural Gradient (NG) algorithms**. In particular, we will focus on Kronecker-Factored approximations to the Fisher (originally introduced as "K-FAC" in [1]). For challenging DL problems, these algorithms typically attain better training results than GD-based algorithms in practice. However, they can be significantly slower when the network width is large. In this case, K-FAC based algorithms only make sense for the practitioner wanting to squeeze even the last bit of performance from their model, and if they are ready to (computationally) pay for it.

In this thesis, we exploit the K-FAC preconditioner's properties arising from its (exponential averaging) construction-paradigm to propose two different ways of alleviating its unfavourable time-scaling with the network width, at negligible accuracy loss. One way is simple (directly exploiting the observed property), but very efficient and generally applicable (*Chapter 3*). The other way is more involved, applicable only in some situations, but typically provides slightly more speed-up (*Chapter 4*). Overall, these **proposed improvements** result in substantial speed-ups (2-3×) for typical problems and even improvements in training outcomes (see *Sections 3.3, 4.5* and *5.6*).

We now continue by slowly progressing into the mathematics of optimizing DNNs.

Formulation of Machine Learning Tasks. We can loosely think of a machine learning (ML) task (problem) as being made of 2 characteristics: the ML paradigm, and the underlying data-type. Understanding this split is important for realising in which cases our proposed algorithms may be used directly, in which cases they might be used with extra work, and in which cases they might not be used at all. We give a non-exhaustive list of most popular ML data-types and paradigms. Details about each will be given as we require them.

The most popular **ML paradigms** are: (1) Supervised Learning - Maximum Likelihood Estimation (MLE) Approach; (2) Supervised Learning - Bayesian Inference Approach; (3) Unsupervised Learning; (4) Reinforcement Learning. Supervised Learning can be also spit into classification and regression, each of which can be MLE- or Bayesian-based. The most popular **ML data-types** are *(1) natural language data, (2) image data, (3) time series.* Graphs are an increasingly popular datatype used lately, which inspired completely new Deep Learning architectures (Graph Neural Nets [21]). We will not touch on this newly formed branch here.

1.1 Deep Neural Networks

Deep neural nets (DNNs) are compositions of *"layer functions"* (or just layers). Each layer is itself a composition of a linear function with a non-linear activation. There are essentially three classes of DNN layers: *fully-connected (FC layers, or dense layers), convolutional (Conv layers),* and *recurrent layers.* A fourth class of "special" layers also exists: containing for example batch-normalization layers [22], pooling layers [23] etc. While these layers are very important for training stability and performance [22], they typically have no trainable parameters - we thus do not focus on them since our proposed algorithms' designs are agnostic of them.

DNNs might be made exclusively of one class of layers, but most state of art DNNs combine layers from different classes. The way in which layers are composed to form a DNN is referred to as the architecture.

Does the DNN architecture depend on ML Paradigm or Data type? Before we describe DNNs layers, it is useful to ask whether our choice of layers depends on ML Paradigm or data type. This will be useful later, when understanding what DNN optimizers' structure is specific to (dependent on). The short answer is that the architecture dependence on the ML paradigm can be ignored, while the dependence on data-type is very strong. We make the following notes about architectural dependence on the most popular ML data types:

1. Language Data (Natural Language Processing): State of art uses *Transformers* architecture ([24]). The trainable parameters come exclusively from FC layers.
2. Image Data (Computer Vision): State of art uses ResNets ([25]) or VGG ([26]). All nets used for this data-type are typically called "Conv Nets" (or CNNs), owing to the fact that most layers (5-20) are convolutional and only the final 1-2 layers are FC. However, due to the sheer width of FC layers required for large models, the parameters of the FC layers account for $20\% - 70\%$ of the total parameters. So even for *Conv nets*, a large proportion of computational effort comes from FC layers.
3. Time Series Data: uses Recurrent nets (RNNs; almost entirely recurrent layers).
4. Data with no obvious structure: FC layers are used exclusively (FC nets).

We thus note that while being very simplistic, FC layers are ubiquitous.

Some Notation and Related Remarks. We begin by introducing our notation. In this thesis, we work with data, and thus require to think of distributions. With a slight abuse of notation, we will address both probability distributions and their densities through the symbol of their density. As an example, we will have the density of the probability distribution p be $p(x)$ and we say that x is drawn from p as: $x \sim p$. We let q be the *true data distribution*. In practice, we will only have samples from this distribution. We can think of these samples as constituting an *empirical data distribution*, denoted by $\hat{q}$. In our reasoning, we will encounter a generative model, and we will use p for its density. We will also have a conditional distribution r, which will appear for particular choices of our generative model.

The main thesis thread only requires the simplest setup. *FC layers:* The notation for *CNNs* and *RNNs* can get involved, distracting from the main idea. Thus, we will develop most of our discussion based on *FC* layers. While we will talk about, and develop, algorithms for Conv layers as well, we will be able to abstract from their technical complexity (without loss of understanding) by drawing a simple parallel with FC layers - at the right knowledge depth level. That is, we only fully look "under the hood" for FC nets. When talking about Conv layers we will only reveal the details strictly relevant to our algorithms, and rely on our understanding of FC layers.

MLE Supervised Learning: We will focus on the MLE supervised learning (SL) case. However, our improvement ideas will be directly applicable to the Reinforcement Learning and Bayesian DNNs settings as well (because the algorithm we improve has been already adapted for those situations).

1.1.1 Fully Connected Nets and MLE Supervised Learning

In this section, we look at a gentle introduction into fully-connected nets in the case of MLE supervised learning, with the aim of defining quantities for future use. W.l.o.g. we put the bias in the weight matrix, and append a 1 to each post-activation. The last column of the weight matrix then represents the bias vector. We have

1. $\bar{a}_0 := [x, 1]^T$ (x is the input to the net),
2. The *pre-activation* at layer l: $z_l = W_l \bar{a}_{l-1}$ for $l \in \{1, 2, ..., n_L\}$, $z_l \in \mathbb{R}^{w_l}$,
3. The *post-activation* at layer l: $a_l = \phi_l(z_l)$ for $l \in \{1, 2, ..., n_L\}$, $a_l \in \mathbb{R}^{w_l}$,
4. The *augmented post-activation* at layer l: $\bar{a}_l = [a_l^T, 1]^T$ for $l \in \{1, 2, ..., n_L - 1\}$ (s.t. we can incorporate the bias into the weight matrix W_{l+1}),

where $\phi_l(\cdot)$ are non-linear activation functions and we have n_L layers. We collect all parameters, $\{W_l\}_{l=1}^{n_L}$, in a single vector $\theta \in \mathbb{R}^{d_\theta}$. The net output is $h(x; \theta) := a_{n_L} \in \mathbb{R}^{N_o}$. Note that $W_l \in \mathbb{R}^{w_l \times (w_{l-1}+1)}$ and w_l is the width (number of neurons) of layer l.

Supervised Learning Loss. We consider having N_D pairs of labeled data with input x and target y, namely $\{(x_i, y_i)\}_{i=1}^{N_D}$. The loss is defined as

$$L(x, y; \theta) := \ell(h(x; \theta), y), \tag{1.1}$$

where $\ell(z, y)$ is the loss function (typically convex). For future reference, we define

$$\delta_l := \nabla_{z_l} L, \quad \forall \quad l \in \{1, 2, ..., n_L\}, \tag{1.2}$$

where $\nabla_b a$ is the gradient of a w.r.t. b. Our learning goal is to minimize the *risk*[1]

$$\min_\theta \mathcal{R}(\theta) := \mathbb{E}_{(x,y)\sim q}[L(x, y; \theta)]. \tag{1.3}$$

In practice, we do not have q, but only $\hat{q}$, and thus relax our risk minimization problem to an *empirical risk minimization* problem

$$\min_\theta \tilde{\mathcal{R}}(\theta) := \mathbb{E}_{(x,y)\sim \hat{q}}[L(x, y; \theta)] = \frac{1}{N_D} \sum_{i=1}^{N_D} L(x_i, y_i; \theta). \tag{1.4}$$

To approach this problem, we will need to think of the gradient of the *true* and *empirical risk*. We denote

$$g(\theta) := \nabla_\theta \mathcal{R}(\theta) = \mathbb{E}_{(x,y)\sim q}[\nabla_\theta L(x, y; \theta)]. \tag{1.5}$$

We will make use of the g notation when we encounter this gradient many times in the same equation for convenience. However, we will also use the $\nabla_\theta \mathcal{R}(\theta)$ when we have

[1] We will refer to $R(\theta)$ as the *true risk* to clearly distinguish it from the *empirical risk*.

many quantities that we need to think of, and we need to understand the subtleties between using $\nabla_\theta \mathcal{R}(\theta)$ and $\nabla_\theta \tilde{\mathcal{R}}(\theta)$. Moreover, we define the per-example gradient as

$$g_i(\theta) := \nabla_\theta L(x_i, y_i; \theta), \tag{1.6}$$

and thus the batch-gradient estimate as (using sampling without replacement throughout the thesis)

$$g_B(\theta) := \frac{1}{|\mathcal{I}_\mathrm{B}|} \sum_{i \in \mathcal{I}_\mathrm{B}} \nabla_\theta L(x_i, y_i; \theta) = \frac{1}{|\mathcal{I}_\mathrm{B}|} \sum_{i \in \mathcal{I}_\mathrm{B}} g_i(\theta), \tag{1.7}$$

where $\mathcal{I}_\mathrm{B} \subseteq \{1, 2, ..., N_D\}$. We will use $\mathcal{I}_\mathrm{B}$ to refer to a subset of $\{1, 2, ..., N_D\}$, and let $n_\mathrm{B} := |\mathcal{I}_\mathrm{B}|$ for convenience. We will denote the trace of a matrix by $\mathrm{tr}(\cdot)$. In this thesis, we always think of using DNNs whenever a function approximation is required.

1.2 Training DNNs

Solving (1.4) is done iteratively with derivative based algorithms, with the typical aim of finding a local minimum. This modest goal is typically satisfactory in practice.

1.2.1 Are DNN Optimizers specific to the ML task?

Before we progress further in our discussion, it is important to ask ourselves whether DNN optimizers are inherently specific to the ML task. This will help us realise how our proposed algorithms might generalize to different settings. We break this question down in two subquestions as follows:

1. **Are DNN Optimizers specific to the ML Paradigm?** The short answer is yes, at least the practical implementation is (though the general idea might be the same). Even the simplest idea of gradient descent has critically different implementations in the supervised-learning case ([27]) and reinforcement learning case ([28]). Put simply, the specificity to ML paradigms is due to different ML paradigms having vastly different objective functions. Thus, the ways in which the objective gradients are practically implemented (and thought of) is vastly different (and so will be gradient-based pre-conditioners).

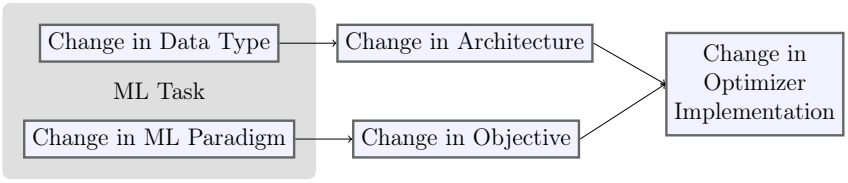
2. **Are DNN Optimizers specific to the Data Type?** The short answer is: the more involved ones are (through the network architecture). Recall we have seen that the data type only influences the architecture. While very simple optimizers are architecture-agnostic[2] ([11, 27]), the more involved optimizers exploit the

[2]As long as automatic-differentiation is available (or Finite-differences can accurately be used).

specific architecture to gain computational advantages ([1, 20]). These are thus specific to the architecture and ultimately to the data-type (which is the main driver behind the architecture).

Figure 1.1 summarizes how our optimizer would (in general) need to change if the ML task is changed. Of course, as mentioned earlier some might disobey the rules implied therein, but most optimizers with complicated implementations would typically satisfy them. Note that these dependencies are in place despite having automatic differentiation routines that can give us gradients through the net with minimal effort.

Figure 1.1: The dependencies of an Optimizer Implementation (for a fixed idea).



Optimizers can be adjusted to apply the same ideas across different paradigms and architectures, though not trivially. An example is K-FAC [1]: originally designed for MLE SL with FC nets, the idea was later used for Conv layers ([29]), recurrent layers ([30]) and Reinforcement Learning (policy-based: [31], value-function based: [32]).

1.2.2 First Order Methods and Backprop

First-order methods are derivative-based optimization methods which rely on gradient information only. Computing the required gradients in DNNs was made possible trough automatic-differentiation (AD) techniques [14]. Without AD, DL would not exist [27]. In the case of computing DNN gradients, AD works by constructing a computation graph of elementary operations as the network function is evaluated at one data-point (during the "forward pass"), and then using chain-rule on the computation graph to obtain desired gradients [14]. Because chain-rule is more efficiently performed from the final-most layers towards the starting layers in this case, this AD way of computing the gradients is called "*backprop*" (and working out the chain-rule computations once the computation graph is in place is called the "backward pass").

1.2.3 Stochastic Gradient Descent (SGD)

Let us denote $\tilde{g}_k := g_{\mathrm{B}}(\theta_k)$ for convenience. Due to the prohibitive cost of computing the gradient, first order methods solve the minimisation problem (1.4) by using the gradient estimator $g_{\mathrm{B}}(\theta)$ for a small batch size, $|\mathcal{I}_{\mathrm{B}}|$. The simplest form of update is

$$\theta_{k+1} = \theta_k - \alpha_k \tilde{g}_k, \tag{1.8}$$

where $\alpha_k > 0$ is the learning rate. Since $\tilde{g}_k$ is a stochastic gradient estimator, $-\tilde{g}_k$ is not guaranteed to be a descent direction. Line-search is very expensive in this case, and also noisy [9]. Thus, we could spend a lot of computation trying to take a descent step that does not exist. As a consequence, the learning rate is typically fixed or set to follow some pre-defined schedule [9]. SGD was initially defined to take iterations of the form (1.8) with $n_{\mathrm{B}} = 1$. The idea was subsequently extended to mini-batch GD, where $1 < n_{\mathrm{B}} \ll N_D$ (which is awkwardly also referred to as SGD at present) [9].

1.2.4 Adaptive Momentum (ADAM)

Instead of discarding the previous information, ADAM ingeniously and efficiently (both computationally and storage-wise) uses past information to perform more informed parameter updates. At each step, ADAM computes biased estimates of the first moment m_k, and of the second moment (uncentered variance) v_k, of the gradient (1.5) as [11]

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\tilde{g}_k, \quad v_k = \beta_2 v_{k-1} + (1-\beta_2)\tilde{g}_k \odot \tilde{g}_k, \tag{1.9}$$

where $\beta_1, \beta_2 \in (0,1)$ are hyperparameters (typically $\beta_1 := 0.9$ and $\beta_2 := 0.999$), and "$\odot$" denotes element-wise multiplication [11]. We typically set $m_0 = 0$ and $v_0 = 0$.

The biased estimators are then used to obtain the unbiased ones as [11]

$$\hat{m}_k = m_k/(1-\beta_1^k) \quad \text{and} \quad \hat{v}_k = v_k/(1-\beta_2^k). \tag{1.10}$$

Finally, the steps are taken as $\theta_{k+1} = \theta_k - \alpha_k \hat{m}_k/(\sqrt{\hat{v}} + \epsilon)$, where $\epsilon > 0$ (typically $\epsilon \in [10^{-8}, 10^{-6}]$) ensures we do not divide by zero. The division is element-wise [11].

ADAM has three desirable properties: Invariance to gradient rescaling, Trust-region-like behavior, and self-adjusting step-length based on signal to noise ratio (see *Section A2.1* for details) [11]. However, the assumptions under which ADAM converges do not hold in practice, and one must use AMSGrad instead, which is a correction to ADAM (see *Section A2.1* for details) [13]. We note that ADAM can be viewed as a second order method with a diagonal preconditioner. This diagonal preconditioner has deep connections with the empirical Fisher, which we defer to *Appendix A5.1.2*.

1.3 Training DNNs: Second Order Methods

Second-order methods are advantageous mainly because they help with ill-conditioning [15]. However, they may also avoid converging to "false" local minima or saddles

[15]. Still, even computing the Hessian is infeasible in the large-scale setting of DNNs. Thus, approaches in the literature focus on (crudely) approximating the Hessian (or a related matrix: the Fisher Information Matrix or the Generalized Gauss-Newton Matrix) and/or the application of its inverse to the gradient.

1.3.1 A Short Literature Survey of Second Order Approaches

In this section, we very briefly review general second-order approaches used in ML, with a focus on DL. We will then concentrate on *natural gradient (NG) methods*.

First Order Information to Approximate Second Order. One possible approach is to cheaply incorporate second-order information by trying to approximate the Hessian (or an approximation thereof) as a diagonal matrix from gradient information only. This gave rise to many successful methods which are considered to be first-order (eg. [11, 13, 12, 33]). In particular, the well-known ADAM (see *Sections 1.2.4* and *A2.1.1*) method can be viewed as using a diagonal approximation of the empirical square-rooted-Fisher to precondition the momentum step [11].

Cheap and Exact Hessian-vector Products. Another set of approaches is based on using the *Pearlmutter trick* (1994, [34]), which allows to compute *Hessian-vector (H-v) products* efficiently and exactly. This gives rise to methods like *Hessian-free* (see *Appendix A5.1.3* for further details; [15, 35]) and *Krylov Subspace Descent* ([36]). These methods use *H-v* products and *Krylov subspace methods* (eg. Conjugate Gradients: CG [37]) to approximately invert the Hessian (or an approximation thereof) over a gradient estimate, thus obtaining an estimate of the *(stochastic) Newton step* (or perhaps the Natural Gradient Descent step, or the Generalized Gauss-Newton step). While the *Pearlmutter trick* indeed allows for *efficient computation* of H-v products, we have to note that even this efficient implementation is still twice as expensive as a backward pass. Furthermore, CG requires a relatively large number of steps (order of hundreds) to achieve reasonable accuracy in estimating the *Newton step* [15, 35]. Thus, we expect *a priori* that these methods perform poorly as compared to SGD, since it is unlikely that the progress provided by the Newton step is hundreds of times better than the one of SGD. We can confirm our intuition by checking the numerical results in the literature [15, 36]. In addition, these methods will have stronger overfitting issues, since they effectively take many steps (think each step of CG is an optimization step) with the same batch, thus parsing through a much smaller amount of data per step [1]. Also detrimental to *H-v*-based methods, there exist examples of quadratic functions (see [38]), with very spread out Hessian eigenvalue spectrum, on which using

CG has no advantage over *GD with momentum*. If our curvature matrix is of that type, we cannot hope that a *Hessian-free* method will be competitive to SGD.

Quasi-Newton. A possibly more promising idea is to use Quasi-Newton methods, with the basic principle being algorithms of the L-BFGS type [9]. However, these methods are typically targeted at *(strongly) convex objectives* [39, 40]. In particular, the approach proposed by *Schraudolph and Yu* (2007, [16]) has been relatively successful for *convex objectives*[3]. However, we want to approach the type of general objectives encountered in (empirical) risk minimization problems for DNN tasks. An interesting approach for this general case is the one of *Berahas et al.* (2021, [17]). They proposed to sample points randomly around the current iterate to produce the BFGS approximation (rather than using previous iterates and gradients), in an attempt to get curvature matrices that are not stale (like the classical BFGS ones are). While the goal is indeed achieved by their proposed algorithms *S-LBFGS* and *S-LSR1*, and nice theoretical properties hold, these methods underperform SGD on DNN tasks, since each sample requires both a forward and a backward pass. Perhaps the only Quasi-Newton approach which has been successful (to date) for DNNs is the one proposed by *Golfarb et al.* (2021, [41]). This uses the Kronecker-factorization idea introduced by Martens [1].

Gauss-Newton. Gauss-Newton (GN) is a well-known way of obtaining a Hessian estimate by using only first-order information. It can be used when we have a non-linear least-squares objective. The idea can be generalized to any objective, with the note that we will then require to compute some second-order information to obtain the Generalized GN (GGN) matrix. Despite its prevalence in classical optimization, GN's success has been quite limited in deep learning [42, 43]. Perhaps the most interesting idea proposed so far is to replace the GN matrix with the Gram matrix produced by the Jacobian, yielding the so-called *Gram GN* algorithm [18]. However, the main problem with this idea is that it does not generalize in an obvious fashion to nets with multi-dimensional outputs, or to any other losses than the squared loss.

Natural Gradient Methods. Natural Gradient methods (NG; see *Section 1.4*) rely on preconditioning the gradient with the Fisher Information Matrix (FIM).

TONGA ([19]; detailed mathematical review in *Appendix A5.1.1*) is an early attempt to make NG feasible for DNNs. It does so by maintaining a low-rank approximation of the *Emirical Fisher Information Matrix* (EFIM), then casts both problems of finding the low-rank approximation and of computing the NG in a lower-dimensional space, thus attaining much better time-complexity than a *naive NG implementation* [19].

[3] In particular: parameter estimation for conditional random fields in natural language processing.

TONGA has quadratic time-complexity in layer size (linear in d_θ), which is relatively cheap (compare to K-FAC). However, the preconditioning performance is relatively poor, and thus there is no real benefit TONGA-preconditioning offers over SGD [19]. The performance can be improved slightly by pretending the EFIM is block-diagonal, and applying TONGA layer-wise (which retains its quadratic time-complexity). This works possibly because it allows for a higher total-rank of the EFIM, coupled with the fact that the true EFIM is close to being block-diagonal [19]. In an attempt to improve TONGA ([44]), the authors combine information from EFIM with Hessian information and show improved results (over block-TONGA) for some data-sets [44]. Still, an SGD variant sometimes performs better for the studied cases [44].

PRONG ([45], detailed mathematical review in *Appendix A5.2.2*) is an interesting approach obtain NG in a feasible way. Instead of trying to compute the *Fisher* cheaply, PRONG reparameterizes the network such that the diagonal blocks of the *Fisher* are closer to being the identity. In this case, performing SGD would be (almost) equivalent to *block-diagonal NGD*. Numerical results in the PRONG paper ([45]) for CIFAR10 and ImageNet-1k datasets show PRONG outperforms SGD. However, comparing PRONG to plain SGD might be unfair (since *PRONG* does something akin to *batch-normalization*). The performance of PRONG and *batch-normalized SGD* is similar [45].

K-FAC ([1, 29, 30]) is a feasible practical NG implementation which relies on the following ideas: (1) approximating the FIM as block-diagonal; (2) re-writing each diagonal block as a Kronecker product ("$\otimes$") of two smaller matrices (the *"K-factors"*); and (3) using the approximation $\mathbb{E}[A \otimes B] \approx \mathbb{E}[A] \otimes \mathbb{E}[B]$. Importantly, it retains the desirable *reparameterization invariance property* of NG (though in weaker forms, as we argue in *Section 1.6*). The main problem of K-FAC is the cubic time-complexity in layer size (compare to quadratic for SGD, TONGA, SENG etc). For this reason, numerical results of standard K-FAC algorithms are mixed ([1, 29, 30, 20]). The per-step performance significantly exceeds the one of SGD, but the steps get (cubically more) expensive for larger layers. Thus, the total time to target accuracy of K-FAC is worse than for SGD (or other more recent practical NG implementations for DNNs) for relatively wide DNNs. Improving K-FAC's scaling (time- and memory-complexity) is our thesis' main focus (see *Section 1.6.1*). We provide detailed mathematical description of K-FAC for FC and Conv layers in *Appendices A5.2.1* and *A5.3.1* respectively, and also a simpler higher-level mathematical description in *Section 1.5*.

SENG ([20]; see *Appendix A5.3.2* for mathematical details) is another approach which makes NG feasible for DNNs in 3 steps (it uses EFIM rather than true FIM): (1) it approximates the EFIM as block-diagonal; (2) by exploiting the "$F_{i,i} = G_i G_i^T$ structure"

of the EFIM diagonal blocks, it re-writes the block-diagonal inverse-application problem (with regularized block-Fisher) as a regularized least-squares (LS) problem; (3) from here, sketching methods are used to efficiently solve the regularized LS problem, and thus obtain the SENG-preconditioned step [20]. The sketching can be improved by being performed "factor-wise" (see *Appendix A5.3.2* for details). Global convergence is proved (under mild assumptions). SENG has quadratic time-complexity in layer size. SENG significantly outperforms *SGD with momentum*, ADAM, and K-FAC, with a stronger advantage in the large-batch large-layersize distributed setting [20].

Despite having worse time-complexity in layer size than SENG, K-FAC has been implemented for a much wider spectrum of Layer types and ML paradigms: FC and Conv nets [1, 29] recurrent nets [30], Reinforcement Learning: algorithm ACKTR [46], Variational Inference for Bayesian Neural Nets [47, 48, 49]). More recently, K-FAC has been used in the context of *Deep Kernel Shaping* to achieve training of skip-free batch-normalization-free DNNs at the same speed as a standard training of ResNet on Imagenet[4] [50]. For this reason, improving K-FAC has maximal practical impact at the time of writing. Notably, however, K-FAC works very well for (relatively) low-width nets (eg. resnets [25]), with per-epoch time roughly the same as SGD (see numerical results in [51, 52, 53]).

NG+ ([54]) is an approach where the authors define a *Generalized Fisher Information Matrix* (GFIM, in a layer-wise fashion). Then, they draw a connection with Empirical Fisher (EFIM), namely that: $\mathbb{E}\big[\text{EFIM}\big] = \mathbb{E}\big[I \otimes \text{GFIM}\big]$ when columns of Gradient matrix are i.i.d. (which does not happen for DNNs, nevertheless). Having defined the (block-wise) GFIM, the "generalized NG direction" is attained by solving a linear system where GFIM is the matrix to be inverted (instead of FIM as for NG). This part is done as in SENG [20], by writing the inverse-application problem as a least-square problem, then sketching (thus having the same complexity). Numerical results show NG+ slightly outperforms K-FAC on ImageNet-1k with resnet50 [54].

Finally, tough not a NG algorithm, **SHAMPOO** ([55, 56]) is worth mentioning, as it uses a similar preconditioner format to K-FAC and NG+. Notably, SHAMPOO applies in a general stochastic optimization setting and does not require a FIM to exist, like NG algorithms do [55]. It is thus oblivious to the particular DNN model structures, and only depends on gradient information (unlike K-FAC, SENG, NG+) [55]. However, it requires obtaining the $(-1/4)^{\text{th}}$ power of matrices of the same shape as the K-factors, making the algorithm slower than K-FAC. For a direct mathematical comparison of the K-FAC, SHAMPOO, and NG+ preconditioning (and step), see equation (13) in [54].

[4] Which is the first time this has been demonstrated in the literature [50].

Exponentially Averaged Curvature Matrices. Notably, in DNN settings, curvature matrices are typically estimated as an exponential average (EA) of the previously encountered local estimates ([1, 11, 57, 58, 59]). We will see the importance of this aspect for our thesis in *Chapter 3*.

1.4 Natural Gradient Methods

In this section, we mathematically describe the basics of Natural Gradient (NG) Methods. We first introduce the Fisher Information Matrix, and then formulate our SL problem as an MLE problem (s.t. we have a Fisher). We then present the (true) Natural Gradient Descent (NGD) method, as well as the empirical NGD method, briefly highlighting the differences. See *Appendix A2.2* for technical details about NG.

1.4.1 The Fisher Information Matrix

The Fisher Information Matrix (from now the *"Fisher"* or *"FIM"*) is defined as

$$F(\theta) := \mathbb{E}_{(x,y)\sim p}\big[\nabla_\theta \log p(x,y;\theta)\nabla_\theta \log p(x,y;\theta)^T\big] \tag{1.11}$$

where $p(x,y;\theta)$ is a generative model for x,y, parameterized by θ. Note that by definition $F(\theta) \succcurlyeq 0$. We will use the notation $F_k := F(\theta_k)$.

Viewing Feed-Forward Nets from an MLE Perspective. Our loss-based supervised learning problem (1.4) does not give us a Fisher. To get that, we must rewrite the learning problem as a maximum likelihood estimation. We begin by viewing the output of the net as being the parameters of a generative distribution for y.

In particular, we take our conditional $y|x$ to have the density $p(y|x;\theta) = r(y|h(x;\theta))$. That is, $p(y|x)$ depends on x and θ only through the output of the net. We can now define (choose) $p(x;\theta) := q(x)$ to get the generative model $p(x,y;\theta)$ as

$$p(x,y;\theta) = p(y|x;\theta)p(x;\theta) = r(y|h(x;\theta))q(x). \tag{1.12}$$

In principle, we could make any choice for $p(x;\theta)$, and thus pick any generative model we would like. However, we want to make a choice which gives

$$-\log(p(x,y;\theta)) = -\log p(y|x;\theta) - \log p(x;\theta) = L(x,y;\theta) + C, \tag{1.13}$$

where C is a constant w.r.t. θ, such that we have

$$\begin{aligned} F(\theta) =& \mathbb{E}_{(x,y)\sim p}\Big[\big(-\nabla_\theta \log p(x,y;\theta)\big)\big(-\nabla_\theta \log p(x,y;\theta)\big)^T\Big] \\ =& \mathbb{E}_{(x,y)\sim p}\big[\nabla L(x,y;\theta)\nabla L(x,y;\theta)^T\big]. \end{aligned} \tag{1.14}$$

This allows us to define the Fisher in terms of our loss. Note that when (1.13) holds our Risk-minimization problem (1.3) becomes a MLE problem.

It is apparent that equation (1.13) holds when choosing $p(x;\theta) = C$ and

$$L(x, y; \theta) = -\log p(y|x; \theta) \tag{1.15}$$

Thus, if the loss can be written in this specific form, setting $p(x;\theta)$ to be any distribution constant w.r.t. θ would work. However, using $p(x;\theta) = q(x)$ is helpful when drawing samples from p, as we can just use the x samples from the data distribution. Another way of thinking about this is that we do not try to model the joint $p(x, y; \theta)$, but only the conditional $p(y|x;\theta)$. We can write all typically used losses as in (1.15), and thus have a Fisher in these cases. For example, for the Cross-entropy loss, we take a multinomial generative model where the softmax-logits are the parameters of r. For the squared loss, we take our generative model to be the Gaussian with the mean being the output of the net and the covariance matrix being an arbitrary positive multiple of the identity[5]. From now on, we will assume we can find a generative model such that (1.13) holds.

Finally, note the subtlety in equation (1.14), where the expectation is taken w.r.t. the model distribution p, rather than the data distribution q (which is what we may hope). It is interesting to note that for most practically chosen losses, solving our *true risk minimization*[6] problem would lead to $p(x, y; \theta) = q(x, y)$, provided q can be written in this parameterized form. Thus, drawing $(x, y) \sim q$ is only reasonable when we are close to convergence. We will discuss this further in the context of Empirical Fisher. Note that the assertion about p being the same as q at convergence holds because, for most practically chosen losses, the true risk minimization problem can be thought of as an MLE, which in turn is equivalent to minimizing the *KL divergence* between our model p and our target q [64]. We briefly discuss this next.

1.4.2 KL Divergence Objectives

The natural gradient can be applied to any objective which measures the performance of some statistical model [61]. However, it enjoys richer theoretical properties when the objective is a KL divergence [64]. To attain these properties, we define our learning goal to be minimizing the *KL divergence* from q to p. That is

$$\min_{\theta} \mathrm{KL}(q(x, y) \,||\, p(x, y; \theta)) = \int_{\mathcal{X},\mathcal{Y}} q(x, y) \log \frac{q(x, y)}{p(x, y; \theta)} dx dy = \mathbb{E}_{(x,y)\sim q}\left[\log \frac{q(x, y)}{p(x, y; \theta)}\right]. \tag{1.16}$$

[5] In general, for (positive definite) covariance matrix $\mathscr{C}$, we have a *"squared loss"* in the norm $\mathscr{C}^{-1}$.

[6] Solving the empirical risk minimization problem we are used to would lead to $p(x, y; \theta) = \hat{q}(x)$.

Note that the KL divergence is zero if and only if $p \equiv q$. Thus, if there exists such a $\theta \in \Theta$, our global solution will have the property that the model distribution is actually the data distribution. This is not surprising, since we want our generative model to converge towards our data distribution. We can see that most losses can be converted to the form shown in (1.16) by expanding

$$\mathbb{E}_{(x,y)\sim q}\left[\log \frac{q(x,y)}{p(x,y;\theta)}\right] = \mathbb{E}_{(x,y)\sim q}\big[\log q(x,y)\big] + \mathbb{E}_{(x,y)\sim q}\big[-\log p(x,y;\theta)\big] \tag{1.17}$$

and realizing that $\mathbb{E}_{(x,y)\sim q}\big[\log q(x,y)\big]$ is constant while $\mathbb{E}_{(x,y)\sim q}\big[-\log p(x,y;\theta)\big]$ is the term we discussed about in the previous subsection.

1.4.3 The Natural Gradient Descent Method

Now that we have looked at the Fisher matrix in depth, we are ready to define the natural gradient descent (NGD). Under our assumption about the loss structure, the natural gradient of the *true risk* is defined as (recalling definition of $R(\theta)$ in (1.3))

$$\nabla_N \mathcal{R}(\theta) := F^{-1}(\theta)\nabla_\theta \mathcal{R}(\theta), \tag{1.18}$$

assuming F is non-singular (which may not always be true). In practice, we pretend our data comes from $\hat{q}$ (instead of q) for simplicity, and thus use $\tilde{\mathcal{R}}$ (recall definition (1.4)) instead of $\mathcal{R}$. NGD then takes a step in the parameter space of the form

$$s^{(NG)} := -\alpha F^{-1}(\theta)\nabla_\theta \tilde{\mathcal{R}}(\theta), \tag{1.19}$$

where α is some learning rate. NGD has several advantages over SGD: reparameterization invariance (albeit for infinitesimally small step-sizes), equivalent to generalized Gauss-Newton under some conditions which often hold in practice, better convergence properties, and Fisher Efficiency [64]. These follow from its interpretations that we (optionally) look at in *Appendix A2.2.2* and from the results in *Appendix A2.2.3*.

1.4.4 Empirical Fisher and Empirical Natural Gradient

The *empirical Fisher*[7] (EFIM) is obtained by replacing the expectation w.r.t. the model distribution p in the *true Fisher* definition (1.11) with an expectation w.r.t. the data distribution q:

$$F^{(E)}(\theta) := \mathbb{E}_{(x,y)\sim q}\big[\nabla_\theta \log p(x,y;\theta)\nabla_\theta \log p(x,y;\theta)^T\big]. \tag{1.20}$$

[7]It is empirical because we do not need to sample from the model to approximate its expectation, but only from the data distribution q.

In practice, we further approximate this as

$$\tilde{F}^{(E)}(\theta) := \frac{1}{n_{\mathrm{B}}} \sum_{i \in \mathcal{I}_{\mathrm{B}}} \nabla_\theta \log p(x_i, y_i; \theta) \nabla_\theta \log p(x_i, y_i; \theta)^T. \tag{1.21}$$

The *empirical Fisher* is also *positive semi-definite* (PSD). We can get the *EFIM* for free if we also compute the gradient. That is because, instead of drawing $y \sim p(y|x;\theta)$ and then performing another backprop as we require with the *true Fisher*, we can just use the already computed gradients. Overall, using the Empirical NG is cheaper than using the True NG. However, which one performs better and why is an ongoing research discussion (aspects of this discussion can be found in *Appendix A2.2.4*).

We now move on to look at K-FAC, a practical NG implementation for DNNs which is essential to our work. A survey of successful practical NG implementations for DNNs, across different ML tasks, can be found in *Appendix Chapter A5*.

1.5 Kronecker Factored Approximate Curvature

Although K-FAC (**K**ronecker **F**actored **A**pproximate **C**urvature) is central to our work, in this section we provide only the details that are strictly required for our understanding. While K-FAC has substantially different implementations for different layer types ([1, 29, 30]), and we will require it for both Conv ([29]) and FC layers ([1]), we can easily abstract from intricate technical details by only caring about certain matrix sizes. Deep technical details about K-FAC for different types of layers can be found in *Appendices A5.2.1* and *A5.3.1*.

In DL, the dimension of F_k is very large, and F_k can neither be stored nor used to complete a linear-solve. K-FAC ([1]) is a practical implementation of the NGD algorithm which bypasses this problem by approximating F_k as

$$F_k^{(\mathrm{KFAC})} := \mathrm{blockdiag}\big(\{\mathcal{A}_k^{(l)} \otimes \Gamma_k^{(l)}\}_{l=1,\ldots,n_L}\big), \tag{1.22}$$

where $\mathcal{A}_k^{(l)} := A_k^{(l)}[A_k^{(l)}]^T \in \mathbb{R}^{d_{\mathcal{A}}^{(l)} \times d_{\mathcal{A}}^{(l)}}$ and $\Gamma_k^{(l)} := G_k^{(l)}[G_k^{(l)}]^T \in \mathbb{R}^{d_{\Gamma}^{(l)} \times d_{\Gamma}^{(l)}}$ are the *forward K-factor* and *backward K-factor* respectively (of layer l at iteration k) [1]. Each block corresponds to a layer and $\otimes$ denotes the Kronecker product. The exact K-Factors definition depends on the layer type (see [1] or *Appendix A5.2.1* for FC layers, [29] or *Appendix A5.3.1* for Conv layers). For our purpose, it is sufficient to state the size of $A_k^{(l)} \in \mathbb{R}^{d_{\mathcal{A}}^{(l)} \times n_{\mathcal{A}}^{(l)}}$ and $G_k^{(l)} \in \mathbb{R}^{d_{\Gamma}^{(l)} \times n_{\Gamma}^{(l)}}$. We have: *(1) for FC layers:* $d_{\mathcal{A}}^{(l)} = w_{l-1}$, $d_{\Gamma}^{(l)} = w_l$, $n_{\mathcal{A}}^{(l)} = n_{\Gamma}^{(l)} = n_{\mathrm{B}}$; *(2) for Conv Layers:* , $d_{\mathcal{A}}^{(l)} = \kappa_w^{(l)} \kappa_h^{(l)} \eta^{(l)} + 1$, $d_{\Gamma}^{(l)} = \eta^{(l+1)}$, $n_{\mathcal{A}}^{(l)} = n_{\Gamma}^{(l)} = n_{\mathrm{B}} d_w^{(l-1)} d_h^{(l-1)}$, where w_l is the width of layer l (the number of neurons in

layer l), n_{B} is the batch size, $\eta^{(l)}$, $d_w^{(l)}$, $d_h^{(l)}$ are the number of output channels, width, and height of the data tensor at layer l. $\kappa_w^{(l)}$, $\kappa_h^{(l)}$ are the convolution kernel width and height at layer l.

Computing $(F_k^{(\text{KFAC})})^{-1}g_k$ can be done relatively efficiently in a block-wise fashion, since we have $(\mathcal{A}_k^{(l)} \otimes \Gamma_k^{(l)})^{-1}g_k^{(l)} = \text{vec}\big([\Gamma_k^{(l)}]^{-1}\text{Mat}(g_k^{(l)})[\mathcal{A}_k^{(l)}]^{-1}\big)$, where $\text{vec}(\cdot)$ is the matrix vectorization operation and $\text{Mat}(\cdot)$ is its inverse. Note that since $\text{Mat}(g_k^{(l)})$ is a matrix, we need to *compute* the inverses of $\bar{\mathcal{A}}_k^{(l)}$ and $\bar{\Gamma}_k^{(l)}$ (eg. through an eigendecomposition[8] - and not just linear-solve with them). This is point is essential.

K-FAC pseudo-code is shown in *Algorithm 1.1*. Note that in practice, instead of assembling $F_k^{(\text{KFAC})}$ as in equation (1.22), with the K-factors local to θ_k ($\mathcal{A}_k^{(l)}$ and $\Gamma_k^{(l)}$), we use an exponential average (EA) ($\bar{\mathcal{A}}_k^{(l)}$ and $\bar{\Gamma}_k^{(l)}$; see *lines 4* and *8* in *Algorithm 1.1*). This aspect is important for our discussion in *Chapters 3, 4* and *5*. In *Algorithm 1.1* we initialize[9] $\bar{\mathcal{A}}_{-1}^{(l)} := I$ and $\bar{\Gamma}_{-1}^{(l)} := I$. θ_0 is initialized as typical [63].

Algorithm 1.1: K-FAC [1]

1 **for** $k = 0, 1, 2, ...,$ *with sampled batch indices* $[\mathcal{I}_B]_k \subset \{1, 2, ..., N_D\}$ **do**
2 **for** $l = 0, 1, ..., N_L$ **do** `// Perform forward pass`
3 Get $a_k^{(l)}$ and $A_k^{(l)}$
4 $\bar{\mathcal{A}}_k^{(l)} \leftarrow \rho\bar{\mathcal{A}}_{k-1}^{(l)} + (1-\rho)A_k^{(l)}[A_k^{(l)}]^T$ `// Update fwd. EA K-factors`
5 Get $\tilde{f}(\theta_k)$; `// The batch-estimate of` $f(\theta_k)$`, from` $a_k^{(l)}$
6 **for** $l = N_L, N_{L-1}, ..., 1$ **do** `// Perform backward pass`
7 Get $g_k^{(l)}$ and $G_k^{(l)}$
8 $\bar{\Gamma}_k^{(l)} \leftarrow \rho\bar{\Gamma}_{k-1}^{(l)} + (1-\rho)G_k^{(l)}[G_k^{(l)}]^T$ `// Update bwd. EA K-factors`
9 Get gradient $\tilde{g}_k = \big[\big(\tilde{g}_k^{(1)}\big)^T, ...\big(\tilde{g}_k^{(N_L)}\big)^T\big]^T$
10 **for** $l = 0, 1, ..., N_L$ **do** `// Compute K-FAC step:`
11 `// Get EVD of` $\bar{\mathcal{A}}_k^{(l)}$ `and` $\bar{\Gamma}_k^{(l)}$ `for inverse application`
12 $U_{A,k}^{(l)}D_{A,k}^{(l)}\big(U_{A,k}^{(l)}\big)^T = \text{eig}(\bar{\mathcal{A}}_k^{(l)})$; $U_{\Gamma,k}^{(l)}D_{\Gamma,k}^{(l)}\big(U_{\Gamma,k}^{(l)}\big)^T = \text{eig}(\bar{\Gamma}_k^{(l)})$
13 `// Use Eigs to apply K-FAC EA matrices inverses to` $g_k^{(l)}$
14 $M_k^{(l)} = \text{Mat}(\tilde{g}_k^{(l)})U_{A,k}^{(l)}(D_{A,k}^{(l)} + \lambda I)^{-1}\big(U_{A,k}^{(l)}\big)^T$
15 $S_k^{(l)} = U_{\Gamma,k}^{(l)}(D_{\Gamma,k}^{(l)} + \lambda I)^{-1}\big(U_{\Gamma,k}^{(l)}\big)^T M_k^{(l)}$; $s_k^{(l)} = \text{vec}(S_k^{(l)})$
16 $\theta_{k+1} = \theta_k - \alpha_k[(s_k^{(1)})^T, ..., (s_k^{(N_L)})^T]^T$ `// Take K-FAC step`

[8] Tough direct inverse computation may also be used, and this is slightly cheaper than EVD (still of the same over-all complexity), the EVD approach allows for a certain type of regularization that produces empirically more stable optimization solution (see *Section 2.1.3* in [53] and/or *Section III.A* in [62]). Stability in this context refers to better performance on the test-set as the batch-size increases [62].

[9] While this was the initialization recommended in the original paper ([1, 29]), latest practical numerical implementations use different initializations. Notably, the theory we develop in *Section 3.1* would still hold for sufficiently large k (number of iterations) with any initialization. Also, initializing from $\bar{\mathcal{A}}_{-1}^{(l)} := A_0^{(l)}[A_0^{(l)}]^T$ and $\bar{\Gamma}_{-1}^{(l)} := G_0^{(l)}[G_0^{(l)}]^T$would actually work directly with our theory.

Key Notes on Practical Considerations (and definitions of T_{Cov} and T_{inv}). In practice, we update the K-factors and recompute their eigendecompositions ("inverses") only every few tens/hundreds of steps [1]. We denote the K-factor update period as T_{Cov}, and the K-factor inverse computation period as T_{inv}. Typically, we have $T_{\text{inv}} = \zeta T_{\text{Cov}}$ with $\zeta \in \mathbb{Z}^+ \setminus \{1\}$. As we began in *Algorithm 1.1*, we formulate our discussion for the case when $T_{\text{inv}} = T_{\text{Cov}} = 1$. We do this purely for simplicity of exposition[10]. Extending our simpler discussion to the case when these operations happen at a smaller frequency is *trivial*, and does not require an explicit exposition. Our practical implementations use the standard practical procedures.

Definition of Layer Size. We now provide a useful definition of layer-size. *Definition 1.1* is meant to help us aggregate the discussion between FC and Conv layers, such that we can make general statements about layer size. For FC layers, w_l is the only size parameter, so using layer width and size interchangeably in this case should not be a surprise. For Conv layers, we have three size parameters: $\kappa_w^{(l)}$, $\kappa_h^{(l)}$ and $\eta^{(l)}$. However, the kernel sizes $\kappa_w^{(l)}$, $\kappa_w^{(l)}$ are almost universally set to $\kappa_w^{(l)} = \kappa_h^{(l)} = 3$ in practice to ensure computational efficiency. The typical parameter that is increased if more layer expressivity is needed is the number of output channels $\eta^{(l)}$ (which is also the number of filters). As such, in our language, for Conv layers we will use *"size of layer l"* and $\eta^{(l)}$ interchangeably. This should feel natural, since typically the only concept of layer size which ever changes values is $\eta^{(l)}$.

Definition 1.1 (**layer size for FC and Conv layers**). *We define the layer size to be: (1) w_l (its width) for FC layers; (2) $\eta^{(l)}$ (number of out-channels) for Conv layers.*

K-FAC preconditioning Time Cost. For layer l, the time-cost of preconditioning with K-FAC is of complexity $\mathcal{O}([d_{\mathcal{A}}^{(l)}]^3 + [d_{\Gamma}^{(l)}]^3)$. In particular, the cost of obtaining the inverse is $\mathcal{O}([d_{\mathcal{A}}^{(l)}]^3 + [d_{\Gamma}^{(l)}]^3)$, owing to the eigendecompositions in *line 12 of Algorithm 1.1*. The cost of applying the inverses is also $\mathcal{O}([d_{\mathcal{A}}^{(l)}]^3 + [d_{\Gamma}^{(l)}]^3)$, owing to the inverse applications in *lines 13 - 15 of Algorithm 1.1*. That is, the over-all time-cost of preconditioning with K-FAC is cubic in layer size (see *Definition 1.1* and review the paragraph after equation (1.22)).

K-FAC preconditioning Memory Cost. K-FAC preconditioning requires storing the EA K-factors $\mathcal{A}_k^{(l)} := A_k^{(l)}[A_k^{(l)}]^T \in \mathbb{R}^{d_{\mathcal{A}}^{(l)} \times d_{\mathcal{A}}^{(l)}}$ and $\Gamma_k^{(l)} := G_k^{(l)}[G_k^{(l)}]^T \in \mathbb{R}^{d_{\Gamma}^{(l)} \times d_{\Gamma}^{(l)}}$, and their corresponding eigenvectors matrices: $U_{A,k}^{(l)}$ and $U_{\Gamma,k}^{(l)}$. Thus, for layer l, the memory

[10] To avoid *if* statements in the presented algorithm.

requirement is of complexity $\mathcal{O}([d_{\mathcal{A}}^{(l)}]^2 + [d_{\Gamma}^{(l)}]^2)$. That is, the over-all memory-cost of preconditioning with K-FAC is quadratic in layer size (see *Definition 1.1*).

1.6 Motivation, Contributions, and Thesis Structure

Our thesis is concerned with improving K-FAC ([1, 29, 30]), and this section provides the motivation behind our choice.

To understand our motivation well, we need to first draw a brief but important comparison between second order methods and first order methods. This helps us better understand when we might want to consider using one over the other (and thus what could be improved to get better optimizers in general). Our discussion will have a slight focus on K-FAC as the second order method of choice (and this choice of focus will be explained later).

Comparison of Second Order Methods with (Polyak Averaged) SGD. Due to stochasticity, SGD with Polyak averaging (PA-SGD) cannot be beaten asymptotically[11] (in terms of convergence rate / progress per step) [60, 64]. This implies the only advantage that second order methods can ever get over SGD is in the pre-asymptotic regime [64]. Theoretical results for simplified quadratic models (which are arguably sufficiently close to DNNs in the basis of some numerical investigations) show that second order methods may indeed get this type of pre-asymptotic advantage (more progress per step) [64].

Thus, whether we should choose a second order method or not then depends on the trade-off between the *preconditioning over-head* and the *size of the pre-asymptotic regime*. Lower overhead[12] and/or late onset[13] of asymptotic regime encourages using second order methods. Focusing on K-FAC, since the preconditioning overhead is cubic in time (thus scaling worse than the gradient part - which is quadratic), a deep understanding of when we shall expect this pre-asymptotic regime behavior to matter

[11] Polyak averaging (PA) SGD iterates is sufficient to obtain a Fisher-efficient optimization. From this point of view PA brings SGD on the same level as NGD.

[12] Because little does it matter applying the preconditioner may give (almost) no extra benefit when the overhead for doing this is itself small (or negligible). This is one crucial benefit of the improved complexity K-FACs we propose. Because the preconditioning overhead is of the same time-complexity (*Chapter 3*), or of lower time-complexity (*Chapter 4*) than the gradient computation, we can choose to use these algorithms without even trying to guess how significant the pre-asymptotic regime is (where second order methods get more progress per step). The idea is that by reducing the overhead, we reduce the down-side of using these preconditioners in the case where they give no practical per-step progress improvement.

[13] Because there are more steps whose per-step progress would improve if we used a preconditioner.

more is required for the practitioner to correctly "guess" (a priori) whether using K-FAC or PA-SGD is better (in terms of time to target accuracy).

To this end, theoretical and numerical investigations (sometimes for simplified settings) have been conducted in the literature [50, 64, 65]. Notably, instead of attempting to understand the size and impact of the preasymptotic regime (as would relate more to the theory in [64]), some of these results directly focus on the number of steps to target (training) loss - which is a better practical measure[14]. Importantly, the two main factors that contribute to how many steps we can save by adding in a preconditioner[15] are: *1) noise level (larger gradient batch-size means lower noise levels)*; and *2) conditioning of the objective (the trainability[16] of the net)* [65]. Results in [64] and [65] show that:

1. Theoretically, for noisy quadratic models (NQM), and in the fixed learning rate setting, increasing the batch-size reduces the steady state risk (SSR; i.e. "final" train-loss), while preconditioning increases convergence rate but also increases SSR [65]. This suggests that for large batch-size[17] applying the preconditioner would result in fewer steps to target loss/accuracy than otherwise. Notably, this result is for the expected train-loss (with no complementary measure concentration result), in the case of the NQM setting (a simplification of the true DNN setting), and when the preconditioner is a power of the Hessian[18] (not the Fisher or the K-FAC matrix).

    - Details[19] of statement *"preconditioning increases convergence rate"*: The optimal convergence rate for the NQM at hand is $2/\kappa$ (where κ is the condition number of $P^{-0.5}HP^{-0.5}$, H is the true NQM Hessian, and P is our preconditioner) [65]. This suggests that using a good preconditioner (i.e.

[14]Since the reason we cared about how long the pre-asymptotic regime is was because second order methods *may* make more progress in this regime, but surely not in the other (when compared to PA SGD).

[15]This is to be compared with the over-head traded to obtain the #steps reductions - which would give whether K-FAC or PA-SGD have a better over-all time to target accuracy.

[16] For a discussion about what makes a DNN easier to optimize (i.e. more *trainable*), see [50] (in particular *Section 12* therein), and [66, 67, 68].

[17]Eg. large enough that SSR of the plain optimizer is so low that the increase from using preconditioner still does not surpass what the "best attainable" test-loss is.

[18]Arguably, the Fisher might be "close enough" to the Hessian in certain cases, but notably there's also a difference between the Fisher and the effective K-FAC matrix. See [64] for details.

[19]It is important to look at the details of the statement *"preconditioning increases convergence rate"* (because we need to know by how much in different cases to understand whether this will indeed give a substantial reduction in the #steps to target accuracy that would dominate the time over-head we pay to precondition).

"close to the true Hessian") results in faster convergence rates (thus more progress per step). Notably however, if the conditioning of the objective landscape is already good (think $H \approx I$), then applying even the best preconditioner would not improve the convergence rate ($2/\kappa$) by much. In this case, using a preconditioner will not lead to a substantial reduction in the total #steps to target accuracy - but the time over-head we pay to precondition will still be the there.

2. Similar theoretical conclusions to point 1 above can also be drawn in the annealed learning rate setting (with schedule $\alpha_k = 1/(1+k)$ for our discussion, in particular), again for the NQM. Notably, when Polyak averaging iterates (can also apply PA to the second order methods iterates), we get that the excess expected loss (in relation to optimal loss) is bounded[20] as [64]

$$\mathbb{E}[R(\bar{\theta}_k)] - R(\theta^*) \leq \frac{\mathrm{tr}\big([H^*]^{-1}\mathscr{C}_g\big)}{2(k+1)|\mathcal{I}_B|} + \frac{\mathcal{C}(H^*, \theta_0, \theta^*)}{(k+1)^2}, \tag{1.23}$$

where $\mathcal{C}(H^*, \theta_0, \theta^*) = h(\theta_0)$ if our preconditioner if chosen to be the true Hessian (H^*) and $\mathcal{C}(H^*, \theta_0, \theta^*) = \frac{[\lambda_{\max}(H^*)]^2}{\lambda_{\min}(H^*)} \frac{\|\theta_0 - \theta^*\|^2}{2}$ when no preconditioner is used (i.e. we perform SGD). Note that θ_0 is the initial guess for our parameter solution and θ^* is the unique optimal solution (we assume $H \succ 0$), $\mathscr{C}_g$ is the covariance matrix of our 1-datapoint estimate of the gradient $g_i(\theta)$ (recall (1.6)), and H^* is the true Hessian of our NQM (see *Theorem 6* in *Section 14.3* of [64], and its consequences in *Section 14.3.1*, for more details). In the basis of equation (1.23), we have the following observations

   (a) There is an $\mathcal{O}(1/k)$ term which represents the asymptotic convergence regime, and an $\mathcal{O}(1/k^2)$ term which captures the pre-asymptotic behavior.

   (b) Increasing the batch-size $|\mathcal{I}_{\mathrm{B}}|$ increases the preasymptotic regime size (all else fixed), because it reduces the $\mathcal{O}(1/k)$ term, and thus it takes longer for the pre-asymptotic error term $\frac{\mathcal{C}(H^*, \theta_0, \theta^*)}{(k+1)^2}$ to decay down to orders of magnitude below the ones of the $\mathcal{O}(1/k)$ term.

   (c) The pre-asymptotic error term improves by a factor of $\frac{[\lambda_{\max}(H^*)]^2}{\lambda_{\min}(H^*)} \frac{\|\theta_0 - \theta^*\|^2}{2h(\theta_0)}$ when preconditioning (with the true hessian H^* as the curvature matrix) as opposed to performing no preconditioning. This improvement is higher when

[20] For simplicity of exposition we used the norm inequality $\|AB\| \leq \|A\| \, \|B\|$ to simplify the results in [64]. We also expressed everything in terms of the covariance matrix of the 1-datapoint gradient estimator, to make the batch-size $|\mathcal{I}_{\mathrm{B}}|$ show in an obvious way.

the conditioning of the objective landscape is sufficiently poor ($\frac{\lambda_{\max}(H^*)}{\lambda_{\min}(H^*)}$ is sufficiently large). If, on the other hand the conditioning is relatively good (better conditioning means $\frac{\lambda_{\max}(H^*)}{\lambda_{\min}(H^*)}$ is closer to 1) the reduction in #steps to reach the asymptotic regime by introducing the preconditioner can be insignificant. In this latter case, the asymptotic regime kicks-in relatively rapidly even when no preconditioner is applied.

(d) By looking at the generic formulation of the bound (1.23), which in particular allows for an arbitrary preconditioner B, we can see how "better preconditioners" ("closer to the true Hessian") results in better preasymptotic behavior (but a more formal discussion rapidly complicates).

3. Preconditioned SGD empirically achieves higher critical batch-size[21] than EA[22]-SGD [65]. This suggests that at certain large batch-size values, we would see more progress per-step from the preconditioned optimizer. However, this result is using two (arguably reasonable [65]) simplifications: the NQM setting (a simplification of standard DNN setting), and the preconditioner is the true Hessian to some power.

4. Using the K-FAC preconditioner empirically results in fewer steps to target-accuracy for all batch-sizes (most of the times but not always) [65]. Notably the comparison is with momentum or plain SGD but not with PA-SGD.

These results suggest[23] that in general, the more poorly conditioned objective landscape (lower DNN trainability) and/or the larger the batch-size (lower noise), the more reduction we expect to see in the #steps to target loss by applying a preconditioner (however, it is hard to estimate how much in real settings). This raises scope for reducing the complexity of the preconditioning cost: if we could attain such improvements (and we shall see we can in *Chapters 3 - 4*), then we would likely get optimizers which perform better than any of the previously available ones in cases where preconditioning reduces the number of training steps significantly (nets with worse trainability and/or large batch-size), but the preconditioning over-head is large (i.e. for large layer size).

[21] Broadly speaking, the batch-size past which further increasing n_{B} gives diminishing returns. See [65] for mathematical definition and details.

[22] Exponentially weighted moving averaging the SGD iterates, which is almost as Polyak averaging (Polyak averaging is standard average, not exponential [69]).

[23] While these results give a good gist/intuition about when we might expect K-FAC to save a significant number of steps (and thus when it may be preferable over PA-SGD), note that none of them can be used directly to compare K-FAC with PA-SGD in standard DNN settings and draw certain conclusions (due to one or more of the following reasons: are empirical results, simplify the objective landscape, require assuming the K-FAC preconditioning matrix is "close to" the Hessian).

We now motivate why we chose to focus on improving K-FAC rather than other second order optimizers for DNNs. This is essentially down to the favorable theoretical properties of K-FAC, as well as related promising empirical evidence, and available room for improvement.

K-FAC step has desirable theoretical properties. The natural gradient (NG) step is *invariant to any smooth reparameterization* for infinitely small step-size[24] (details in *Appendix A2.2.2*, or see [64]). The reparameterization invariance is desirable because it solves the problem of ill-conditioning - which leads to first-order methods taking many steps to solution, when encountered [1, 15, 64].

However, because NG algorithms operate in the parameter space (rather than in the distribution space) and take finite steps, *practical NG implementations* are provably *not*[25] invariant to reparameterization (for a complete discussion of this see [64] and in particular *Section 12* therein). Furthermore, since K-FAC introduces approximations to the Fisher matrix (and also takes finite steps), we *cannot* expect K-FAC to be invariant to arbitrary smooth reparameterizations. Nevertheless, theoretical results ([1][26], [29][27], [30][28]) show that the K-FAC step (with finite step-size) is still invariant to a *"reasonably broad"* class of reparameterizations [1, 29, 30, 64]. In particular, this *"reasonably broad"* class includes most reparameterizations that have been practically used in Deep Learning to *improve performance* (see [1, 29, 30] for details and in particular *Section 4.1* in [29] for an overview). Thus, by being invariant to these (performance enhancing) reparameterizations, we expect the K-FAC step to get all their benefits automatically (without explicitly applying them) [29, 64].

In addition to the desired *reparameterization invariance*, NG can also be proved to have better asymptotic convergence rate, at least in some particular cases and under some simplifying assumptions (see *Appendix A2.2.3* for a detailed discussion with references). This also motivates the use of K-FAC, though in a similar way

[24]Updates are equivalent under reparameterization up to first-order, for finite step-sizes [29, 64].

[25]Though they are approximately invariant, see [64] for details.

[26]Shows that K-FAC (for FC layers) is invariant to a broad class of reparameterizations: this includes affine transformations of the activations (considered as a group), both before and after the nonlinearity, and whitening of activation to have zero mean [1].

[27]Shows that the KFC (K-FAC for Conv layers, described in *Appendix A5.3.1*) updates are invariant to homogeneous, pointwise affine transformations of the activations, both before and after the nonlinearity. This result is weaker than for FC layers, since for convolutional layers, we cannot expect an algorithm to be invariant to arbitrary affine transformations of a given layer's activations (see Section 4.1 in [29] for and explanation as to why). While more limited than in the case of FC layers, KFC is still invariant to a broad set of transformations, such as normalizing activations to be zero mean and unit variance either before or after the non-linearity [29].

[28]Similar statements as in the case of FC and Conv layers, but for Recurrent Neural Nets.

to the situation for *reparameterization invariance property*, there is the caveat of a discrepancy appearing due to the K-FAC approximation to the Fisher Matrix.

Empirical evidence suggests K-FAC takes fewer steps to converge (though sometimes more time)[29]. In line with what theory about the K-FAC and NG steps suggests, empirical studies ([51, 52, 53, 65, 70, 71, 72, 73]) have shown that K-FAC typically accelerates training by enabling convergence with fewer iterations than SGD (or other first order methods)[30]. However, when thinking in terms of over-all run-time, numerical comparisons of SGD with K-FAC in the literature sometimes feature[31] papers where the over-all time-to-target-loss of K-FAC is better (eg. [29, 53, 52, 73]), and papers where the converse holds (eg. [1, 20, 29, 51]). Notably, this holds because in some instances the time over-head we pay to precondition pays off in terms of reducing the number of steps[32], but in other scenarios it does not[33].

Our discussion in this section so far motivates the improvement of the preconditioning overhead time-complexity in K-FAC. The benefits of doing this (in relation to our discussion so far) are threefold:

- First, by getting the preconditioning over-head to be of the same time-complexity as the gradient part (in *Chapter 3*), or better (in *Chapter 4*), we increase the set of problems where K-FAC has better time to target loss/accuracy than PA-SGD (and the complementary set reduces). We shall see that in practice, this

[29]Speaking about standard K-FAC ([1, 29, 30]), at the time of writing.

[30]As a concrete example, take the KAISA case, where baseline validation accuracy is achieved in 40% fewer epochs than SGD on ResNet-32 with the CIFAR-10 dataset (see Figure 1 in [53]). Many other such example exist ([51, 52, 53, 71, 70, 72, 73]). We shall see that our numerical results in *Chapters 3, 4, 5* are consistent with these literature findings (or better).

[31]The split we give here is not perfect, nor are the results in the basis of which we did this split. Notably, Some of the test-cases are pretty bad, and are either effectively measuring noise (when measuring the improvement of K-FAC over SGD), or being corrupted significantly by noise. In the basis of our discussion before, this is the case with resnets (particularly for low gradient batch-size) where the asymptotic regime is short enough s.t. the benefit of preconditioning is minimal, while the layer-width is small enough s.t. the preconditioning time-overhead is negligible (for more details see [65]). Thus whether K-FAC is better than SGD in terms of over-all time is almost a matter of noise in these cases and neither of the two main drivers behind K-FAC's over-all time performance explained above dominates sufficiently much to get a clear difference in over-all time between K-FAC and SGD (and thus time-measurement noise due to initialization, parameter tuning, not running sufficiently many runs, etc., matters a lot in those cases). This effect gets mitigated at large resnet batch-sizes (as can be deduced from [65]).

[32]This tends to be the case for relatively low-width nets - like resnets and other custom low-size nets.

[33]When the layer size is large, the preconditioning overhead dominates due to it scaling cubically in time - as opposed to the gradient computation which is quadratic in time. In this case the reduction in #steps to target accuracy that is required of K-FAC (relative to SGD) to be competitive is very high - and unliely to be the case.

complexity improvement results in significant reduction in the actual K-FAC preconditioning over-head for large layer sizes[34] - and thus a significant set of testcases where the reduction in #steps to target accuracy were not worth the overhead are now worth the (much lower) overhead (in particular this held true for all that we investigated, with a notable caveat[35]).

- Secondly, by having preconditioning overhead of the same order (or less) than the gradient part, one needs not worry so much about whether to pick K-FAC or not. In cases where the proposed (complexity) improvements are applicable, the downside of picking (our improved versions of) K-FAC (over SGD) is limited as the preconditioning overhead is minimal (with our proposed improvements), whereas the benefit of the preconditioner is typically non-negative, and sometimes substantially beneficial [1, 65].

- Thirdly, by reducing the over-all complexity of K-FAC to coincide with the one of SGD, we effectively remove any feasibility problem of K-FAC that may arise in cases when SGD can be applied (of course, only when the proposed improvements can be applied/implemented).

In this thesis, we propose mathematically principled ways to reduce K-FAC time and memory complexities[36] (such as to close the cost-gap between SGD and K-FAC). In particular, in *Chapter 3* we propose a quadratic-time approximation to K-FAC where the error (relative to K-FAC) can typically be tightly controlled. Furthermore, in *Chapter 4* we propose a linear-time and linear-memory approximation. To our knowledge, our proposed algorithms are the first K-FAC improvements which achieve scalings (of the preconditioning part of the algorithm; in layer size) of: (1) quadratic time, (2) linear time, (3) linear memory.

1.6.1 Research Questions and Contributions

To achieve our outlined aims, we pose and answer the following research questions.

Research Question 1: Can we improve K-FAC's time-complexity bottleneck? The short answer is yes. *Research Questions 1.1 - 1.2* give details.

[34] Where it mattered the most.

[35] But the comparison was with momentum SGD not PA-SGD - which is a potential future improvement of the experiments.

[36] Other ways to reduce cost have been proposed (see for example [51, 53, 73]), but these tend to give heuristic solutions, and do not improve complexities, and/or introduce relatively large errors relative to standard K-FAC.

Research Question 1.1: Can we make *the preconditioning part of the K–FAC algorithm* have quadratic time complexity in layer size? We answer this affirmatively in *Chapter 3*. Furthermore, we show that the K-factor representation error (introduced when attempting to reduce complexity) can be tightly controlled while worsening this quadratic complexity only by a log multiplicative factor. In particular, theoretical results (in *Sections 3.1* and *A3.1*) show that as the layer size increases (for Conv and FC layers), we only need to keep a constant number of singular[37]-modes (for sufficiently large layer sizes) when attempting to represent the K-factors up to a relative truncation error of at most ϵ. This statement holds for any $\epsilon \in (0, 1)$, though with different thresholds of "sufficiently large" for different ϵ values.

By directly obtaining the required number of modes (chosen to ensure the desired error) with *randomized Numerical Linear Algebra* (rNLA) techniques, we can thus reduce the complexity of obtaining the K-factors inverse representations, while controlling the error as specified (see *Sections 3.1.3 - 3.1.4* and *3.2.4* for details). We call these algorithms *"Randomized K-FACs"*. Intuitively, the complexity improvement comes from having to obtain only a *constant number of eigenmodes* as the layer size increases, rather than linearly many in layer size (as EVD in standard K-FAC does).

Notably, a slight complication arises because directly obtaining the modes introduces a further error, which we shall call *"projection error"* (see *Section 2.2.1*). However, theory in literature (described in *Section 2.2.3*, used in *Section 3.2.4*) shows that the total error of rNLA-obtained representations can be bounded above by a (small) constant multiple ($c > 1$) of SVD-based truncation error (the optimal error used in our theoretical discussion). To enforce this bound, we must choose to perform $\mathcal{O}(\log(d_M))$ power iterations[38], where d_M is (loosely speaking) the layer width. Since each of these power-iterations is of $\mathcal{O}(d_M^2)$ time-complexity, the total time-complexity we require is thus[39] $\tilde{\mathcal{O}}(d_M^2) := \mathcal{O}(\log(d_M) d_M^2)$ when attempting to control the error. Note that when we do not want to control the error, we can choose a fixed number of power iterations, and get away with $\mathcal{O}(d_M^2)$ time-complexity. Also note that the $\log(d_M)$ term is almost negligible, even theoretically. Numerically, our case-studies are in line with this idea. The total error of rNLA-obtained representations can be bounded above by as little as $1.5\times$ of SVD-based truncation error with high probability (see *Appendix A3.6*) when using as little as $n_{\text{pwt-it}} = 4$ power iterations.

[37]Also eigenmodes, due to the matrices being symmetric positive semi-definite.

[38]In our subspace-finding subroutine, see *Algorithm 2.1* and *Section 2.2.3* .

[39]In general, the $\tilde{\mathcal{O}}(f(x))$ notation means a $\mathcal{O}\big([\log(x)]^k f(x)\big)$ complexity (for some $k > 0$). However, in our thesis, the $\tilde{\mathcal{O}}(f(x))$ notation will *always* mean $\mathcal{O}(f(x)\log(x))$ (so $k = 1$).

Of course, c's numerical value depends on the hyperparameters we set for our rNLA algorithms, but these specified values are obtainable for well chosen such hyperparameters - see *Section 3.2.4*. Thus, we can still have almost the same statement as we would have if we were to directly obtain[40] the leading modes with no projection error: we can reduce the time-complexity (in layer size) of obtaining our K-factors inverse-representations by (almost[41]) a power, while controlling the error below a fixed relative truncation error $c\epsilon$, with high probability (and this statement holds for any $\epsilon \in (0, 1)$, and some small $c > 1$). We only have to be careful to enforce $c\epsilon$ (rather than ϵ) to be lower than the relative truncation error we can tolerate in practice.

Finally, in our discussion above we meant *"complexity"* in *its strictest definition as an asymptotic concept*. That is, our *improved complexity* would only translate to *practical computational savings* for sufficiently large layer size (see *Section 3.1.3.1* and *Figure 3.1* therein). In our case, sufficiently large means past a certain value which scales proportionally to $|\log(\epsilon)|$ (see *Section 3.1*). Based on our developed theory, the "sufficiently large" region does not include typical DNNs settings (see *Section 3.1.3.2*), due to our bounds being relatively loose. However, numerical case-studies (in *Section 3.1.3.3*) show that most K-factors are practically in this "sufficiently large layer-size" regime (and that the error can be thus tightly controlled while obtaining practical computational gains in standard practical settings).

Importantly, the preconditioning part involves: *(1) an inverse computation*, and *(2) an inverse application*. The answer to the question so far was purely for the inverse computation part, and reducing that complexity to (almost[42]) quadratic (in layer size). Still, naively applying the inverse (as most practical implementations do[43] [51, 53, 52, 71]) is of cubic time-complexity in standard K-FAC (notably though, a quadratic improvement is also proposed in *Section 8* of the original K-FAC paper, which is not really used in practical implementations). Thus, carelessly inheriting practically used inverse-application subroutines would give over-all cubic time-scaling for our proposed *Randomized K-FACs*, even if the inverse estimation part is (almost) quadratic. Nevertheless, by taking care when applying the low-rank *Randomized K-FACs* inverse representations, we can easily make the complexity quadratic for that part as well (see equation (3.56) and the discussion around it). Notably, we do not

[40] An idealistic and not achievable scenario.

[41] There is an extra multiplicative $\log(d)$ term.

[42] There is an extra multiplicative $\log(d)$ term when trying to control the error. This extra $\log(d)$ multiplicative term disappears if we are happy not to control the error.

[43] In fact, most practical implementations do not even explicitly say anything about the inverse application subroutine, though this is a computational bottleneck (alongside the inverse computation - having the same complexity but better multiplicative constant).

use the approach proposed in *Section 8* of the original K-FAC paper ([1]), but instead design an approach that better fits our available low-rank inverse representations. We thus obtain algorithms for which the whole preconditioning part has (almost) quadratic time complexity (in layer size).

Research Question 1.2: Can we make *the preconditioning part of the K–FAC algorithm* have linear time complexity in layer size? We answer affirmatively to this in *Chapter 4*. Therein, we use Brand's update for[44] online Numerical Linear Algebra (oNLA; [74]) to propose an online update to the inverse K-factor representation which has linear time-complexity (in layer size). We call this the *"B-update"* and combined with a low-rank truncation this gives the B-KFAC algorithm (see *Section 4.1.1*). The B-update is only applicable in some cases (see *"A Caveat for B-update Usability"* below *Algorithm 4.1*), but notably it becomes applicable for any FC or Conv layer for "sufficiently large" layer size.

As in the answer to the previous research question, we also need to have a *same-order inverse application time-complexity* to obtain a better over-all time-complexity for the preconditioning part. Thus, we also propose[45] an inverse application scheme which takes linear time, and is applicable whenever the B-update is applicable (see *Section 4.3*). Thus, the over-all preconditioning time-complexity of our proposed B-KFAC is linear in layer size, whenever applicable. Compare this to cubic for standard K-FAC and to quadratic scaling of gradient computation part (i.e. of SGD).

Unlike for *Randomized K-FACs*, for B-KFAC we do not have a theoretical way to control the error[46]. However, by using some limited theory, and some intuition we are able to propose improvements to B-KFAC which (conceptually) appear promising in *practically controlling the* B-KFAC *error*. These B-KFAC improvements now take *"linear time for most iterations"* but *"quadratic time for some iterations"* (see *Section 4.4*). Numerically, we see that though agnostic to the error, these improvements to B-KFAC indeed control the B-KFAC K-factor representation error (see *Section 4.2.3*).

All the algorithms proposed in *Chapter 4* are part of the *"Brand New K-FACs"* family of algorithms, as we call them, owing to the novel use of Brand's update. There are three improvements to our proposed B-KFAC that we give:

[44]Other Brand's updates exist.

[45]This is obtained by applying the idea of *Section 8* in the original K-FAC paper with our low-rank inverse representations - a combination which leads to great computational savings.

[46]Though error theory is possible through investigating the B-KFAC stochastic process, we left this as future work.

1. **BR-KFAC:** In the context of simple 2D examples (see *Section 4.1.2*), we show that *at least under some measure on incoming K-factor information* (M_j, see equation (4.1)), high error events can occur for B-KFAC (viewed as a stochastic process) with *low but non-negligible probability*. These high-error events are characterized by the total K-factor representation error of B-KFAC significantly exceeding the error that R-KFAC would achieve at worst, and for many consecutive iterations. This motivates periodically using an RSVD update (R-update) to the B-updated K-factors representations, as a control mechanism to ensure that, should such an event occur, its span over iterations would be lower (and thus negative effects be mitigated). This gives the BR-KFAC algorithm (see *Section 4.1.3*).

2. **BC-KFAC:** We also propose a correction (called the C-update) of the same time-complexity as RSVD (quadratic), but of cheaper total cost. This correction, when applied to any current K-factor representation (including a B-KFAC one) does not increase the Frobenius-norm error of the K-factor representation (see *Proposition 4.3* and *Corollary 4.4*). In fact, we expect the correction to typically reduce the error. This motivates periodically (but rarely) using a C-update in a B-KFAC algorithm to control the B-KFAC K-factor representation error, giving rise to the BC-KFAC algorithm (see *Section 4.1.4*).

3. **BRC-KFAC:** An improvement to B-KFAC which introduces both (typically very low frequency) R-update and (typically moderate frequency) C-updates to the B-updated K-factors. See *Section 4.1.5*.

Research Question 2: Do these complexity improvements translate to practical speed-ups for standard DNN settings? We have the following answers:

1. The numerical performance of our proposed quadratic complexity *Randomized K-FACs* is investigated in *Section 3.3*. Therein, we see that for practical settings our complexity improvement (with controlled error) translates to speed-up factors of approximately 3× (relative to standard K-FAC) for CIFAR10 classification VGG16_bn (see VGG architecture in paper [26]).

2. The numerical performance of our linear time-complexity B-KFAC (and other hybrids between B-KFAC and *Randomized K-FACs*) is investigated in *Section 4.5*. Therein, we see that our proposed *Brand New K-FACs* variants provide further speed-ups (though modest) relative to our previously proposed R-KFAC in settings where the FC layers are relatively large.

3. The numerical performance of our proposed algorithms is also tested in distributed settings (for 1 - 4 GPUs), in *Chapter 5* (*Sections 5.5 - 5.6*). Therein, our speed-ups were improved. Details are discussed in *Research Question 3*.

Research Question 3: Can we obtain *distributed implementations* of our proposed algorithms which outperform the SGD and K-FAC benchmarks? We answer this question affirmatively, at least for the studied numerical cases. In *Chapter 5* we consider providing versions of our previously proposed algorithms for distributed settings. We more carefully design practical aspects of the algorithms to squeeze further speed-ups (see *Sections 5.1 - 5.3*). We also test our algorithms on a wider set of problems than before: (1) CIFAR10 with VGG16_bn_lmxp[47], (2) CIFAR100 with VGG11_bn_lmxp, (3) SVHN with VGG11_bn_lmxp, and (4) Imagenette full-size v2 with resnet50 (all problems are classification). We also compare our algorithms with distributed versions of two benchmarks: SGD and K-FAC.

Our numerical results (for 1 - 4 GPUs) show that our proposed algorithms obtain speed-ups over the benchmarks of 3× - 4× when the K-factors were relatively large (for VGG16_bn_lmxp: problems (1) - (3)), and of 1.1× when the K-factors were smaller (for resnet50: problem (4)). This kind of behavior is expected since our proposed algorithms improve scaling (complexity). These speed-ups are somewhat larger than the ones obtained in the numerical experiments of *Chapter 3 - 4* due to further perfecting the algorithms.

The numerical experiments in the distributed setting of *Chapter 5* also act as a better comparison ground between our proposed algorithms. When comparing across all the *Chapter 5* problems and proposed algorithms (using a simple metric) we conclude BR-KFAC is the best algorithm for reaching the target test accuracy in minimal time. Also, B-KFAC is the best proposed algorithm when resources limitations are relatively significant w.r.t. the learning problem at hand (see *Section 5.6.4*).

Research Question 4: Can we propose a low-memory version of K-FAC? K-FAC requires storing square matrices of dimensions proportional to layer size, thus having a quadratic storage (memory) cost in layer size. Our B-KFAC algorithm (proposed in *Chapter 4*) is low-memory (as well as linear time complexity) when it can be applied (see *Section 4.1.6*). In particular, B-KFAC takes memory which scales linearly (in layer size) for the preconditioning quantities (at least for sufficiently large layers; see *Table 4.2*). Thus, B-KFAC reduces the memory-complexity of K-FAC from quadratic down to linear.

[47]See *Section 4.5* for definition of the "VGG16_bn_lmxp' DNN.

1.6.2 Structure of Thesis and Contributions Summary

Our thesis is structured in *six Chapters*, as follows:

1. *Chapter 1* contains introductory material. It describes the problem, poses the research questions, and briefly explains how we answered them. Together with *Chapter 2*, it constitutes the background one needs for understanding our work.

2. *Chapter 2* contains the technical background. In *Sections 2.1 - 2.3* we present Numerical Linear Algebra methods which we will use. We also present technical details of Algorithms of interest (ADAM and NG) in *Appendices A2.1* and *A2.2*.

3. *Chapter 3*: We propose quadratic time-complexity (in layer size) improvements to K-FAC (of cubic time-complexity). The error introduced to achieve this improvement can be theoretically controlled by only worsening the stated complexity through a multiplicative log factor. Numerical results show improved complexity gives practical speed-up factors of $2-3\times$ for typical test-cases. See *Research Questions 1.1* and *2* for a detailed exposition of *Chapter 3* contribution.

4. *Chapter 4*: We propose linear time-complexity (in layer size) improvements to K-FAC (of cubic time-complexity). The method can only be applied in certain cases, and we do not have theoretical error guarantees (unlike *Chapter 3*). By combining with ideas from *Chapter 3*, we propose a total of 3 new algorithms (*"The Brand-New K-FACs"*), and then discuss numerical results showing further speed-ups over *Chapter 3*. See *Research Questions 1.1, 2* and *4* for a detailed exposition of *Chapter 4* contribution.

5. *Chapter 5* provides distributed implementations of our algorithms proposed in *Chapters 3* and *4*. This work (and the related *Research Question 3*) was requested by our industrial partners, *Numerical Algorithms Group Ltd.* A detailed description of the distributed design (along with a short literature review) is provided. Numerical results show we strongly outperform standard K-FAC and SGD in this setting (at least for the 1-4 GPUs regime which we explored). See *Research Questions 2 - 3* for details about *Chapter 5* contributions.

6. *Chapter 6* contains discussions and conclusions.

The following strands of work are also worth mentioning:

1. *Chapter A5 (appendix).* Contains a literature survey of practical NG algorithms across different ML tasks (recall we defined a task as a ML paradigm "+" a data-type). This appendix chapter provides more detail of some algorithms important to our thesis as well as provide some further context.

2. We also explored a different strand of work during the DPhil. This work is not present in the thesis (see [75]), and gives a principled optimization formulation (point of view) for the ubiquitous heuristic of exponentially-averaging curvature matrices. Generalizing from this formulation (when the curvature matrix is the Fisher) we propose a new class of algorithms *KL-Divergence Wake-regularized models*. We then provide three practical instantiations (out of which 2 have analytic solution for the step) and discuss mixed numerical results [75].

1.6.3 Points to Note

Notation: Algorithms vs Their Outputs. We will use SMALL CAPS FONT to denote algorithms in the text. The only exceptions to this rule will be in titles, where we will use large caps font even for algorithms. We will use LARGE CAPS FONT to denote decompositions or other abbreviations. For example, we shall use EVD to refer to the algorithm which provides us the eigenvalue decomposition (EVD). Though the algorithm to obtain the EVD has a different name (in principle, see *Section 2.1.2*) than "EVD", we will use this kind of notation for simplicity of exposition. We shall also use RSVD to refer to the randomized singular value decomposition algorithm, and RSVD to refer to its output (the low-rank representation of the SVD returned by RSVD). We use an analogous notation for SVD and SVD, and also for SREVD and SREVD (see *Section 2.2.2*), and anything else.

Hyper-references: We acknowledge there will be many abbreviations, titles, sections, definitions, propositions, and concepts that will be referred to throughout the thesis. To facilitate reading, we have added click-able hyper-references from most such references, even when it may appear there is none (for example try clicking: LP, DNNs, B-update).

Chapter 2

Technical Background

2.1 Numerical Linear Algebra

In this section, we briefly describe the QR factorization, the eigenvalue decomposition (EVD) and the singular-value decomposition (SVD). These are a prerequisites for other linear-algebra algorithms we will use, as well as for K-FAC (directly). For the SVD, we also look at its low-rank representation optimality property (in unitary invariant norms). This will be used in *Chapters 3-4*. We will only care about real matrix case.

2.1.1 The QR Factorization

Let $X \in \mathbb{R}^{d\times m}$ for $d \geq m$ (if $d < m$ we transpose). The **thin** QR factorization of X is

$$X \overset{\text{QR}}{=} QR, \tag{2.1}$$

where $Q \in \mathbb{R}^{d\times m}$ has orthonormal columns and $R \in \mathbb{R}^{m\times m}$ is upper-triangular.

The **full** QR of X is

$$X \overset{\text{FULL-QR}}{=} \hat{Q}\hat{R}, \tag{2.2}$$

where $\hat{Q} \in \mathbb{R}^{d\times d}$ is orthonormal and $\hat{R} \in \mathbb{R}^{d\times m}$ is upper-triangular. The relationship between the full and thin versions of QR is simple. We have $\hat{Q} = [Q; S]$ and $\hat{R} = [R^T; O_{m\times(d-m)}]^T$ where $O_{m\times(d-m)}$ is a matrix full of zeros and $S \in \mathbb{R}^{d\times(d-m)}$ satisfies $S^TS = I$ and $S^TQ = 0$. Note that the columns of Q and S represent orthonormal bases for range(X) and range$(X)^\perp$ respectively. The time complexity of thin QR is $\mathcal{O}(dm^2)$ and of full QR is $\mathcal{O}(d^2m)$ [76]. See numerical implementations of QR in [76].

2.1.2 The Eigenvalue Decomposition (EVD)

The eigenvalue decomposition of a square matrix $X \in \mathbb{R}^{d\times d}$ is

$$X \overset{\text{EVD}}{=} PDP^{-1}, \tag{2.3}$$

where $P \in \mathbb{R}^{d\times d}$ is invertible and $D \in \mathbb{R}^{d\times d}$ is a diagonal matrix. We will use "$\overset{\text{EVD}}{=}$" to highlight an EVD. Note that this decomposition may not exist for some X, in which case we call X "defective" (or non-diagonalizable). For us, the EVD will be useful as a tool in obtaining a matrix inverse: $X^{-1} = PD^{-1}P^{-1}$. As we have seen, the EVD is directly used in K-FAC for obtaining the inverse of the K-factors (line 12 of *Algorithm 1.1*). Numerically, the EVD is computed by reduction to Hessenberg form with Householder matrices (see [76] for details) and has time-complexity $\mathcal{O}(d^3)$ [76]. However, we shall refer to the algorithm providing us with the EVD as "the EVD" for simplicity (as explained in *Section 1.6.3*).

2.1.3 The Singular Value Decomposition (SVD)

The **thin** (or reduced) singular-value decomposition (SVD) of a matrix $X \in \mathbb{R}^{d\times m}$ with $d \geq m$ (if $d < m$, transpose and use X^T) is given by

$$X \overset{\text{SVD}}{=} U\Sigma V^T, \tag{2.4}$$

where $\Sigma \in \mathbb{R}^{m\times m}$ contains the singular-values (which are non-negative) on the diagonal and zero otherwise, and $U \in \mathbb{R}^{d\times m}$ and $V \in \mathbb{R}^{m\times m}$ have orthonormal columns. Being square, V is also orthogonal. We will use the "$\overset{\text{SVD}}{=}$" sign to highlight a thin SVD. Most of our reasoning will be based on the thin SVD. However, when looking at the randomized SVD (in *Section 2.2.1*) we will also require a different way of expressing the SVD, called the *"full SVD"*. From now on, we use "SVD" to refer to the thin SVD.

The **full** SVD of a matrix $X \in \mathbb{R}^{d\times m}$ (where d needs not be larger than m) is

$$X \overset{\text{FULL-SVD}}{=} \hat{U}\hat{\Sigma}\hat{V}^T, \tag{2.5}$$

with $\hat{\Sigma} \in \mathbb{R}^{d\times m}$ containing the singular-values on the diagonal (and zero otherwise), and $\hat{U} \in \mathbb{R}^{d\times d}$, $\hat{V} \in \mathbb{R}^{m\times m}$ are now both orthogonal matrices. If[1] $d \geq m$, the full SVD essentially takes the matrix U in the thin SVD and pads it with $d - m$ orthonormal columns to get $\hat{U}$ as orthogonal. It also pads $\hat{\Sigma}$ with zeros accordingly. That is when $d \geq m$ we have $\hat{U} = [U; U_\perp]$ and $\hat{\Sigma} = [\Sigma^T; O_{m\times(d-m)}]^T$, and $\hat{V} = V$, where $O_{m\times(d-m)}$ is a matrix full of zeros, and $U_\perp$ satisfies $U_\perp^T U_\perp = I_{(d-m)\times(d-m)}$ and $U_\perp^T U = 0$.

Importantly, when X is square ($m = d$), and *symmetric positive semi-definite* (SPSD) the thin SVD, the full SVD, and the EVD coincide (see [76]). That is $U = V = P$ and $\Sigma = D$, and since the SVD always exists ([76]), then for SPSD matrices, the EVD also exists. Indeed, the Fisher and the Kronecker-factors will be SPSD, so we will have these properties when working with them.

[1] If not, think about transposing.

Algorithms for numerically obtaining the SVD are somewhat similar to the ones for EVD (see [76]), and have $\mathcal{O}(dm^2)$ flops-complexity for thin SVD and $\mathcal{O}(d^2m)$ for full SVD (for $d \geq m$). In practice, we use standard pytorch implementations.

Note that in practice, any $\mathcal{O}(d^3)$ time-complexity can be sped up to $\mathcal{O}(d^\omega)$ for $\omega \in (2, 2.376)$ [77]. However, this sort of speed-up would apply to all the complexities we speak about, so it will not change computational effort hierarchy. For simplicity, we will discuss in terms of the complexities in the absence of this speed-up.

We will always consider the SVD as having the singular values in Σ as sorted descendingly, such that $\Sigma[1,1] \geq \Sigma[2,2] \geq ...\Sigma[d,d]$.

The rank-r SVD Truncation is the Optimal Rank-r Representation

Let us consider the singular values of X (sorted descendingly), and denote those as $\{\sigma_i(X)\}_{i=1}^m$. The $(p,k)-$norm of $X \in \mathbb{R}^{d\times m}$ can be expressed as $\|X\|_{(p,k)} = (\sum_{i=1}^k \sigma_i^p(X))^{1/p}$ and is defined for any $p > 0$ and any $k \in [1, m] \cap \mathbb{Z}$. Note that the Frobenius and the 2-norm are particular cases as: $\|\cdot\|_F = \|\cdot\|_{(p=2,k=m)}$ and $\|\cdot\|_2 = \|\cdot\|_{(p=2,k=1)}$.

Proposition 2.1 tells us that representing any matrix by its first r singular modes is the optimal rank r representation (in the sense of minimal (p,k)-norm of error).

Proposition 2.1 (**Optimality of SVD rank-r representation in the $\|\cdot\|_{(p,k)}$**). *Let $X \in \mathbb{R}^{d\times m}$, let $r \leq rank(X)$. We have in any (p,k)-norm that*[2]

$$\arg \inf_{\substack{B\in\mathbb{R}^{d\times m} \\ rank(B)\leq r}} \|X - B\|_{(p,k)} = U_X[:,:r]\Sigma_X[:r,:r](V_X[:,:r])^T, \tag{2.6}$$

where $U_X\Sigma_X V_X^T \overset{\text{SVD}}{=} X$ is the SVD of X and standard matrix slicing notation is used.

Proof. See [78] for the general case (or [76] for a particular 2-norm result).

2.2 Randomized Numerical Linear Algebra

Randomized NLA (rNLA) is one of our tools for cheaply obtaining the first few singular/eigen-modes of a matrix. It is thus a prerequisite for *Chapters 3* and *4*. Instead of computing the full decomposition and then truncating, rNLA gives us methods to cheaply obtain an approximation for the first few modes (directly). This approximation is very accurate when there is substantial singular/eigen-value decay.

[2]To clarify, the python slicing notation $X[:,:r]$ refers to taking all rows (first argument in $[\cdot,\cdot]$) and first r columns (second argument in $[\cdot,\cdot]$) of matrix X. The $[:r,:r]$ slicing returns the first r elements on the diagonal.

In our context, rNLA can be used with minimal accuracy loss. As we shall see, this arises from the relatively strong eigenvalue decay in the K-factors (induced by the exponentially-averaged construction paradigm) coupled with the fact that regularization "washes away" small eigenvalues.

2.2.1 Randomized SVD (RSVD)

Before we begin diving into rNLA, we note that whenever we say SVD, or QR, we always refer to the thin versions unless otherwise specified. Let $X \in \mathbb{R}^{d\times m}$ be an arbitrary matrix. For convenience, assume for this section that $d \geq m$ (else we can transpose X). Consider the SVD of X

$$X \overset{\text{SVD}}{=} U_X \Sigma_X V_X^T, \tag{2.7}$$

and assume Σ_X is sorted decreasingly. From *Section 2.1.2*, we know that the best[3] rank-r approximation of X is given by $U_X[:,:r]\Sigma_X[:r,:r]V_X[:,:r]^T$ [79]. The idea behind randomized SVD is to obtain these first r singular modes without computing the entire (thin) SVD of X, which is $\mathcal{O}(dm^2)$ time complexity. *Algorithm 2.1* shows the RSVD algorithm alongside with associated time complexities. We omit the derivation and error analysis for brevity (see [79] for details).

Algorithm 2.1: Randomized SVD (RSVD) [79]

Input: $X \in \mathbb{R}^{d\times m}$, target rank $r < \min(d,m)$, oversampling parameter $r_o \leq \min(m,d) - r$, number of power iterations $n_{\text{pwr-it}} \geq 0$
Output: Approximation of the first r singular modes of X
Sample Gaussian Matrix $\Omega \in \mathbb{R}^{m\times(r+r_o)}$ // $\mathcal{O}(m(r+r_o))$ `flops`
Compute $X\Omega$ // $\mathcal{O}(md(r+r_o))$ `flops`
$QR = \text{QR_decomp}(X\Omega)$ // $\mathcal{O}(d(r+r_o)^2)$ `flops`
for $i = 1, ..., n_{pwr\text{-}it}$ **do** // Perform $n_{\text{pwr-it}}$ power iterations to better approximate leading eigen-subspace
    // Use orthogonalized version of Ω in the power iteration
    Compute $XX^TQ = X(X^TQ)$ // $\mathcal{O}(md(r+r_o))$ `flops`
    $QR = \text{QR_decomp}(XX^TQ)$ // $\mathcal{O}(d(r+r_o)^2)$ `flops`
$B := Q^TX \in \mathbb{R}^{(r+r_o)\times m}$ // $\mathcal{O}(dm(r+r_o))$ `flops`
Compute Full SVD (i.e. not the thin one) of B^T, and transpose it to recover $B \overset{\text{FULL-SVD}}{=} U_B\Sigma_B V_B^T$ // $\mathcal{O}(m^2(r+r_o))$ `flops`
$\tilde{U}_X = (QU_B)[:,:r]$; $\tilde{\Sigma}_X = \Sigma_B[:r,:r]$; $\tilde{V}_X = V_B[:,:r]$ // $\mathcal{O}(d(r+r_o)^2)$ `flops`
Return $\tilde{U}_X \in \mathbb{R}^{d\times r}$, $\tilde{\Sigma}_X \in \mathbb{R}^{r\times r}$, $\tilde{V}_X \in \mathbb{R}^{m\times r}$

The returned quantities, $\tilde{U}_X \in \mathbb{R}^{d\times r}$, $\tilde{\Sigma}_X \in \mathbb{R}^{r\times r}$, $\tilde{V}_X \in \mathbb{R}^{m\times r}$ are approximations for $U_X[:,:r]$, $\Sigma_X[:r,:r]$ and $V_X[:,:r]$ respectively - which is what we were after.

[3] As defined by closeness in the "(p,k)-norm" (see for example [78]).

These approximations are relatively good with high probability, particularly when the singular values spectrum is rapidly decaying [79]. The total complexity of RSVD is $\mathcal{O}(m^2(r+r_o) + n_{\text{pwer-it}}md(r+r_o))$ - significantly better than the complexity of SVD, $\mathcal{O}(dm^2)$, when $r + r_o \ll \min\{d, m\}$ and $n_{\text{pwr-it}} \ll \min\{d, m\}$ (which is the case even when controlling the error, as we shall see in *Section 2.2.3*).

We will see how we can use this complexity improvement to speed up K-FAC in *Chapter 3*. Note the presence of the over-sampling parameter r_o, which helps with accuracy at minimal cost. We will see this r_o appear in many places. Finally, we note that Q is meant to be a skinny-tall orthonormal matrix s.t. $\left\|X - QQ^TX\right\|_F$ is "small". There are many ways to obtain Q, but in *lines 3 - 9* of *Algorithm 2.1* we presented a simple one for brevity (see [79] for details).

Power Iteration: In practice the singular spectrum of X may decay slowly and thus the range finding technique which resumes to a single product $X\Omega$ might not give too good an estimate for the leading singular-subspace [79]. In practice, we thus perform a few power iterations, as in *lines 6 - 9* of *Algorithm 2.1*. Based on the theory we look at in *Section 2.2.3*, we will see that we need to choose $n_{\text{pwr-it}} = \mathcal{O}(\log(d))$ to control the projection error (as d increases). This fact will carry over in the complexity of our *Chapter 3* algorithms when the algorithms therein ensure the error is controlled.

RSVD Error Components. Note that there are two error components when using the returned quantities of an RSVD to approximate a matrix. The first component is the *truncation error* - which is the error we would have if we computed the SVD and then truncated. The second error is what we will call *projection error*: the error between the rank-r SVD-truncated X and the RSVD reconstruction of X (it appears due to only approximately hitting the leading mode subspace through power iteration).

RSVD for Square Symmetric PSD matrices. When our matrix X is square-symmetric PSD (the case we will fall into) we have $m = d$, $U_X = V_X$, and the SVD (2.7) is also the eigen-value decomposition. As RSVD brings in significant errors[4], *Algorithm 2.1* will return $\tilde{U}_X \neq \tilde{V}_X$ even in this case. Thus, we have to choose between using $\tilde{V}_X$ and $\tilde{U}_X$ (or any combination of these). A key point to note is that $\tilde{V}_X$ approximates $V_X[:, :r]$ better than $\tilde{U}_X$ approximates $U_X[:, :r]$ [80]. Thus, using $\tilde{V}_X\tilde{\Sigma}_X\tilde{V}_X^T$ as the rank-r approximation to X is preferable. This is what we do in practice, and it gives us virtually zero *projection error*.

[4]Relatively small, but higher than machine precision - as SVD would have.

2.2.2 Randomized Symmetric EVD (SREVD)

When X is square-symmetric PSD we have $m = d$, $U_X = V_X$, and the SVD (2.7) is also the EVD. In that case, we can exploit the symmetry to reduce the computation cost of obtaining the first r modes. SREVD is shown in *Algorithm 2.2.*

The returned quantities, $\tilde{U}_X \in \mathbb{R}^{d\times r}$ and $\tilde{\Sigma}_X \in \mathbb{R}^{r\times r}$ are approximations for $U_X[:, :r]$ and $\Sigma_X[:r, :r]$ respectively - which is what we were after. The same observations about obtaining Q that we made in *Section 2.2.1* also apply here.

The complexity is still[5] $\mathcal{O}(n_{\text{pwr-it}} d^2(r + r_o))$ as with RSVD[6], but the full-SVD of $(Q^T X)^T$ ($\mathcal{O}(d^2(r + r_o))$ complexity) is now replaced by a matrix-matrix multiplication of $\mathcal{O}(d(r + r_o)^2)$ and a virtually free eigenvalue decomposition. However, note that by projecting both the columnspace and the rowspace of X onto Q, we are losing accuracy because we are essentially not able to obtain the more accurate $\tilde{V}_X$ as we did with RSVD. That is because we have $P_C = Q^T U_X$, and thus we can only obtain $\tilde{U}_X = QQ^T U_X$ but *not* $\tilde{V}_X$. Consequently, our *projection error* is larger when using SREVD than when using RSVD, even though the truncation error is the same.

Algorithm 2.2: Symmetric Randomized EVD (SREVD) [79]

1 **Input:** Square-symmetric PSD matrix $X \in \mathbb{R}^{d\times d}$, target rank $r < d$, oversampling parameter $r_o \leq d - r$, number of power iterations $n_{\text{pwr-it}} \geq 0$
2 **Output:** Approximation of the first r eigenmodes of X
3 Sample Gaussian Matrix $\Omega \in \mathbb{R}^{d\times(r+r_o)}$ // $\mathcal{O}(d(r + r_o))$ `flops`
4 Compute $X\Omega$ // $\mathcal{O}(d^2(r + r_o))$ `flops`
5 $QR = \text{QR_decomp}(X\Omega)$ // $\mathcal{O}(d(r + r_o)^2)$ `flops`
6 Do $n_{\text{pwr-it}}$ iterations as in *lines 6 - 9* of *Algorithm 2.1* // $\mathcal{O}\big(n_{\texttt{pwr-it}} d^2(r + r_o)\big)$
7 Compute $C = Q^T X Q$ // $\mathcal{O}(d^2(r + r_o))$ `flops`
8 $P_C D_C P_C^T = \text{Eigen_decomp}(C)$ // $\mathcal{O}((r + r_o)^3)$ `flops`
9 $\tilde{U}_X = QP_C$; $\tilde{\Sigma}_X \in \mathbb{R}^{r\times r}$ // $\mathcal{O}(d(r + r_o)^2)$ `flops`
10 **Return** $\tilde{U}_X[:, :r] \in \mathbb{R}^{d\times r}$, $\tilde{\Sigma}_X[:r, :r] \in \mathbb{R}^{r\times r}$

2.2.3 Randomized NLA: Projection Error Theory

The most important error to understand and characterize in rNLA algorithms is the projection error. For a fixed target-rank r, this error represents the discrepancy between the optimal rank-r truncation obtainable with SVD, and the rNLA provided rank-r representation. Note that if our sample matrix Ω covers the (left singular) subspace of the r leading modes entirely, our previously discussed algorithms will

[5] Dropping $+(r + r_o)^3$ as $r + r_o \ll d$ will hold for us.
[6] Set $m = d$ in RSVD complexity.

obtain the optimal rank-r representation[7], thus having zero projection error for both RSVD and SREVD (which is the ideal case). In other words, the main concern of rNLA algorithms is finding the leading r-modes subspace: if we can do this with high accuracy our projection error will be low. The power iteration we discussed about in *Section 2.2.1* and the oversampling rank (parameter r_o) help with this goal. Conversely, the truncation error is only related to our choice of r, and is not a direct concern of rNLA algorithms[8].

Thus, rNLA error theory focuses around bounding the error $\left\|X - QQ^TX\right\|_2$ (note $Q \in \mathbb{R}^{d\times(r+r_o)}$). When using Gaussian Ω we have (see the *Corollary 10.10* and the simplifying version thereafter in [79])

$$\mathbb{E}\left\|X - QQ^TX\right\|_2 \leq \left(1+\sqrt{\frac{r}{r_o-1}}+\frac{e\sqrt{r+r_o}}{r_o}\sqrt{\min\{m,d\}-r}\right)^{\frac{1}{2n_{\text{pwr-it}}+1}}\sigma_{r+1}. \quad (2.8)$$

The expectation[9] results are meaningful (and practically useful), as measure provably[10] concentrates around the expectation [79]. Notably, as $n_{\text{pwr-it}} \to \infty$, the r.h.s of (2.8) converges to σ_{r+1} (the $(r+1)^{\text{th}}$ singular value of X) exponentially fast. Having bounded $\left\|X - QQ^TX\right\|_2$, one can now devise bounds on the total error of different rNLA algorithms: these can be used to understand the projection error.

For RSVD, by using *Theorem 9.3* in [79] (also see *Remark 5.1* and *Section 9.4* in [79]), one can easily get that the rank-r rsvd representation (with oversampling r_o) has total error bounded in expectation as follows

$$\left\|X - \tilde{U}_B\tilde{\Sigma}_B\tilde{V}_B^T\right\|_2 \leq \left\|X - QQ^TX\right\|_2 + \sigma_{r+1}, \quad (2.9)$$

$$\mathbb{E}\left\|X - \tilde{U}_B\tilde{\Sigma}_B\tilde{V}_B^T\right\|_2 \leq \left(1+\sqrt{\frac{r}{r_o-1}}+\frac{e\sqrt{r+r_o}}{r_o}\sqrt{\min(m,d)-r}\right)^{\frac{1}{2n_{\text{pwr-it}}+1}}\sigma_{r+1}+\sigma_{r+1}. \quad (2.10)$$

Since the optimal rank-r representation has error σ_{r+1} (which we call truncation error), we see that the bound on $\mathbb{E}\left\|X - QQ^TX\right\|_2$ can be thought of as a bound on the projection error, at least in the case of RSVD.

[7]This can be verified by thinking about the SVD of our target matrix X, and how the rNLA algorithms act on it.

[8]We only want to be able to say how close obtaining the r modes directly with rNLA is to obtaining them using the more expensive SVD (then truncate) route. How many modes we decide to keep is a different matter.

[9]Expectation is over the Gaussian matrix, which is the only source of randomness.

[10]Though the authors prove this for a slightly different case rather than the case we are interested in here. However they state theorems can be combined to achieve this, without proof.

In practice, the bounds in (2.8) and (2.10) can be significantly improved to be much tighter when there is a decay in the omitted (tail) modes [79]. For our purpose however, the exhibited results are enough. Notably, the bounds in (2.10) tell us that by picking $n_{\text{pwr-it}}$ sufficiently large we can get the total error arbitrarily close to $2\sigma_{r+1}$ (twice the optimal error). That is, we can get the projection error arbitrarily close to the truncation error σ_{r+1} (by increasing $n_{\text{pwr-it}}$).

Importantly, note that as $\min\{d, m\}$ increases (think d increases for simplicity as our matrices will have $m = d$), we can maintain the r.h.s. of the error upper-bound (2.10) constant by choosing $n_{\text{pwr-it}} = \mathcal{O}\big(\log(\min\{d, m\})\big)$. We will use this observation when obtaining the complexity of our proposed *Chapter 3* K-FAC improvements - in the case we are trying to control the error (the observation is not required when we are happy not to control the error). More details about choosing $n_{\text{pwr-it}}$ shall follow in *Chapter 3*, but we have just seen the general idea.

We do not give any bounds for SREVD, nor does the author ([79]), which does not explicitly propose the SREVD algorithm. While we know SREVD will have a higher error in general (since we also project form the right), we will choose $n_{\text{pwr-it}}$ to be the same value as we choose for RSVD. Improving this aspect is future work.

2.3 Online Numerical Linear Algebra

Online Numerical Linear Algebra (oNLA) refers to performing efficient updates to relevant linear-algebra quantities which depend on a matrix that gets modified as new information comes on the fly. Here, we only need to focus on efficiently computing additive modifications of the SVD. Alongside *Section 2.2*, this is a prerequisite for our proposed algorithms in *Chapter 4*. As an alternative to rNLA, we will use oNLA to exploit the K-factors eigenspectrum decay. We will see that for FC layers, we can leverage oNLA to further reduce costs at the expense of losing accuracy.

2.3.1 Brand's Algorithm (2006)

We now review an algorithm which allows us to cheaply update the thin-SVD of a low-rank matrix when the original matrix is updated through a low-rank addition. We will refer to this as the Brand algorithm[11] (proposed in [74], 2006). Now, consider the **low-rank** matrix $X \in \mathbb{R}^{d\times m}$, with rank $r < \min(m, d)$ and its thin SVD

$$X \overset{\text{SVD}}{=} UDV^T, \tag{2.11}$$

[11] Word of warning: there exist other algorithms by Brand M (that also go by the name of "Brand's algorithm"). In fact, the one we use is probably the least popular.

where $U \in \mathbb{R}^{d\times r}$, $V \in \mathbb{R}^{m\times r}$ are orthonormal matrices and $D \in \mathbb{R}^{r\times r}$ is a diagonal matrix. Now, suppose we want to compute the SVD of $\hat{X} := X + AB^T$, where $A \in \mathbb{R}^{d\times n}$ and $B \in \mathbb{R}^{m\times n}$ with n s.t. $r + n < \min(m, d)$. Brand's Algorithm (exactly) computes this SVD cheaper than performing the SVD of $\hat{X}$ from scratch, by exploiting the available SVD of X [74]. To do so, it uses the identity [74]

$$\hat{X} = \begin{bmatrix} U & Q_A \end{bmatrix} M_S \begin{bmatrix} V & Q_B \end{bmatrix}^T \text{ with } M_S := \begin{bmatrix} I & U^TA \\ 0 & R_A \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & V^TB \\ 0 & R_B \end{bmatrix}^T, \quad (2.12)$$

where $Q_AR_A \overset{\text{QR}}{=} (I - UU^T)A$ and $Q_BR_B \overset{\text{QR}}{=} (I - VV^T)B$ are the QR decompositions[12] of matrices $(A - UU^TA)$ and $(B - VV^TB)$ respectively. Now, we only need to perform the SVD of the *small* matrix $M_S \in \mathbb{R}^{(r+n)\times(r+n)}$. We can then use the SVD of M_S to obtain the SVD of $\hat{X}$ (since U, V, Q_A and Q_B are orthonormal and $U^TQ_A = V^TQ_B = 0$)[13]. The complete algorithm is shown in *Algorithm 2.3*.

The time complexity of *Algorithm 2.3* is $\mathcal{O}((r+n)^3 + (m+d)(r+n)^2)$. This is better than performing RSVD ([79]) on $\hat{X}$ with target rank $r + n$, which is $\mathcal{O}(md(r + n + r_o))$ (typically $r_o \approx 10$). Note that Brand's algorithm gives the exact SVD. The RSVD would also be (almost) exact when the target rank is $r + n$.

Algorithm 2.3: Brand's algorithm [74]

Input: $U \in \mathbb{R}^{d\times r}$, $V \in \mathbb{R}^{m\times r}$, $D \in \mathbb{R}^{r\times r}$, $A \in \mathbb{R}^{d\times n}$ and $B \in \mathbb{R}^{m\times n}$ `// with` $(r + n) < \min\{m, d\}$
Output: SVD of $\hat{X} := X + AB^T = UDV^T + AB^T$
Compute U^TA and V^TB ; `//` $\mathcal{O}\big((m + d)rn\big)$ `flops`
$A_\perp := A - UU^TA$; $B_\perp := B - VV^TB$; `//` $\mathcal{O}\big((m + d)rn\big)$ `flops`
$Q_AR_A = \text{QRdec}(A_\perp)$; $Q_BR_B = \text{QRdec}(B_\perp)$; `//` $\mathcal{O}\big((m + d)n^2\big)$ `flops`
Assemble M_S as in (2.12) ; `//` $\mathcal{O}\big((r + n)^3\big)$ `flops`
$U_MD_MV_M^T = \text{SVD}(M_S)$; `//` $\mathcal{O}((r + n)^3)$ `flops`
Compute $U_{\hat{X}} = \begin{bmatrix} U & Q_A \end{bmatrix} U_M$; `//` $\mathcal{O}\big(d(r + n)^2\big)$ `flops`
Compute $V_{\hat{X}} = \begin{bmatrix} V & Q_B \end{bmatrix} V_M$; `//` $\mathcal{O}\big(m(r + n)^2\big)$ `flops`
Set $D_{\hat{X}} = D_M$
Return SVD of $\hat{X}$: $U_{\hat{X}}D_{\hat{X}}V_{\hat{X}}^T \overset{\text{SVD}}{=} \hat{X}$

Brand's Algorithm for Symmetric X with Symmetric Update

In our case, we only care about the case when $X \in \mathbb{R}^{d\times d}$ is square, symmetric and positive semi-definite: with SVD $X \overset{\text{SVD}}{=} UDU^T$ and $A = B$. In this case, the SVD and EVD of $\hat{X}$ will be the same (and also for X), and we can also spare some computation.

[12] Any decomposition where Q_A and Q_B are orthonormal matrices would work, but we pin it down to QR for simplicity. See Brand's paper [74].

[13] The reader is referred to the original paper for details [74].

The Symmetric Brand's algorithm is shown in *Algorithm 2.4* (this is our own trivial adaptation after Brand's Algorithm to use the symmetry).

Note that M_S will be symmetric in this case. Furthermore the eigenvalues of M_S will be the same as the eigenvalues of $\hat{X}$, which are nonnegative. Thus, M_S's SVD and EVD are the same. Thus, we can compute U_M and D_M in practice by using a *symmetric eigenvalue decomposition* algorithm of the small matrix M_S.

Algorithm 2.4: Symmetric Brand's Algorithm

Input: $U \in \mathbb{R}^{d\times r}$, $D \in \mathbb{R}^{r\times r}$, $A \in \mathbb{R}^{d\times n}$ `// with` $(r+n) < d$
Output: SVD of $\hat{X} := X + AA^T = UDU^T + AA^T$
Compute U^TA; Compute $A_\perp := A - UU^TA$; `//` $\mathcal{O}(drn)$ `flops`
$Q_AR_A = \text{QR_decomp}(A_\perp)$; `//` $\mathcal{O}(dn^2)$ `flops`
Assemble M_S as in (2.12) with $B \leftarrow A$, and $V \leftarrow U$; `//` $\mathcal{O}\big((r+n)^3\big)$ `flops`
$U_MD_MU_M^T = \text{EVD}(M_S)$; `//` $\mathcal{O}((r+n)^3)$ `flops`
Compute $U_{\hat{X}} = \begin{bmatrix} U & Q_A \end{bmatrix} U_M$; Set $D_{\hat{X}} = D_M$; `//` $\mathcal{O}\big(d(r+n)^2\big)$ `flops`
Return (Exact) SVD of $\hat{X}$: $U_{\hat{X}}D_{\hat{X}}U_{\hat{X}}^T \overset{\text{SVD}}{=} \hat{X}$

The total complexity of *Algorithm 2.4* is $\mathcal{O}((r+n)^3 + d(r+n)^2)$. This is better than the complexity of directly performing SREVD[14] on $\hat{X}$, which is $\mathcal{O}(d^2(r+n+r_o))$ (for $r_l \approx 10$), especially when $r+n \ll d$ (the case we will fall into). However, note the computational saving w.r.t. non-symmetric Brand's Algorithm is modest.

Practical Considerations

We have seen that using symmetric Brand's algorithm to adjust for a low-rank update is faster than performing the SREVD, while giving virtually the same result[15]. However, to use Brand's algorithm we had to have the EVD of X - which in principle requires further computation. We will see that in our case, because we work in an "online" setting, we can actually use previous information to obtain an approximate EVD of X for free. Thus, we can use Brand's algorithm to obtain further speed-ups (better time-complexity) when compared to merely using SREVD (or RSVD), but at the expense of some accuracy.[16]

[14] Symmetric variant of RSVD, see [81].

[15] SREVD will (almost) give the exact EVD since the target rank is the true rank. We will need to use an "oversampling" parameter $r_o \approx 10$ say for this to happen though - which will not modify the complexity substantially. Brand's Algorithm is exact.

[16] To keep computational effort fixed with iterations, we will only use an approximate EVD of "X" rather than an exact one - by truncating to a fixed rank to avoid continual rank-growth - see *Chapter 4*. For extra clarity, the truncation will be performed outside *Algorithm 2.4* such that the input to it is always of the same size (rank r).

Chapter 3

Randomised K-FAC algorithms

Research in optimization for DL has lately focused on Natural Gradient (NG) methods, owing to its desirable properties when compared to standard gradient [64, 61]. K-FAC (**K**ronecker **F**actored **A**pproximate **C**urvature, [1]) is a *computationally feasible* implementation of NG, which nevertheless suffers from the drawback of requiring the actual inverses of the Kronecker Factors (not just a linear solve). This computation scales *cubically*[1] in layer size[2], and is typically performed using an eigenvalue decomposition[3] (EVD) [51, 62, 53]. From a practical point of view, this cubic scaling results in relatively large computation time even for modest layer sizes [20].

In this chapter, we address this scalability issue and propose a way to reduce the total time complexity of K-FAC from cubic (in layer size) to quadratic[4]. As we have seen in the literature review section, there exist other natural gradient (SENG [20]) and second order algorithms for DNNs (K-FAC BFGS [41]) that scale quadratically in layer size (but have their own difficulties, see *Section A5*). Our work here brings K-FAC on the same level with these algorithms from a time complexity point of view.

We achieve the time complexity reduction by using randomized NLA techniques (see *Section 2.2*) to compute only a (small) constant number of eigenmodes[5], rather than all the modes through a full EVD (as standard [29, 62]).

[1] See *line 12* of *Algorithm 1.1*.

[2] Recall *Definition 1.1* in *Section 1.5*.

[3] Tough direct inverse computation (or a Cholesky decomposition $LL^T = M$ followed by a forward-backward solve on the identity) may also be used. While these approaches are slightly cheaper than EVD (still of the same over-all complexity), the EVD approach allows for a certain type of regularization that produces empirically more stable K-FAC steps (see *Section 2.1.3* in [53] and/or *Section III.A* in [62]). Stability in this context refers to better performance on the testset as the batch-size increases [62].

[4] An extra multiplicative log factor appears if we also attempt to control the error (see *Section 3.2.4* and/or our short discussion on the next page).

[5] Constant w.r.t. the K-factor size (which is proportional to layer size).

Importantly, we show theoretically (under some simplifying assumptions; in *Sections 3.1.1 - 3.1.2*) that for a fixed eigenvalue cutoff threshold (as a percentage, ϵ, of the maximum eigenvalue), the number of modes we need to keep remains constant w.r.t. K-factor size d_M, as d_M increases past a certain threshold (i.e. is d_M is sufficiently large). This implies that for any $\epsilon > 0$, we can obtain inverse representations of K-factors with relative[6] total error of $c\epsilon$ in $\mathcal{O}(|\log \epsilon| n_{\text{B}} \log(d_M) d_M^2)$ time complexity through rNLA. The constant $c > 1$ appears due to directly obtaining the leading modes, which introduces a further error. Attempting to control this further error requires performing $\log(d_M)$ power iterations of $\mathcal{O}(d_M^2)$ complexity, as we briefly discussed in *Section 2.2.3* (we shall expand more upon this in *Section 3.2.4*). The value of c is approximately 3-4 in practice if we choose our rNLA hyperparameters well. This approach is preferable to obtaining full accuracy in $\mathcal{O}(d_M^3)$ as with the standard EVD approach, particularly because the K-FAC algorithm performs many other approximations. Our approach becomes increasingly relevant for large K-factor size (d_M).

Our theory guarantees we can reduce the inversion time complexity while controlling the error[7] as desired. However, this improved complexity only translates to computational benefits if the K-factor size d_M is sufficiently large[8]. When considering some standard practical settings, our theory predicted that this sufficiently large d_M regime is not reached for any K-factor (as we shall see in *Section 3.1.3.2*). However, numerical evidence for a particular case-study (in *Section 3.1.3.3*) showed that most K-factors empirically behaved as though they were in this desired sufficiently large d_M regime (in *Section 3.1.3.3*): allowing for substantial computational savings. This empirical evidence suggested the situation may be much better in practice than implied by our theory, possibly due to having loose bounds or omitting useful structural assumptions on our K-factors which hold in practice.

It is important to note that instead of controlling the error (relative to K-FAC), one may choose to control computation cost. By taking this approach, we typically lose theoretical error guarantees in standard practical settings: to obtain any computational savings, we typically have to retain a much smaller number of modes than implied by the theory. However, our numerical evidence suggests that even when retaining very few modes the error is typically controlled below a reasonable threshold. Notably, when choosing to use fewer modes than theoretical guarantees required, it might be tempting for the practitioner to also discard performing $\mathcal{O}(\log(d_M))$ power iterations

[6]Total error divided by largest eigenvalue of the K-factor in discussion.

[7]Under some simplifying assumptions, our most applicable (but loosest bounds) theory relies only on a simplifying assumption which is very mild, and evidence shows this holds well in practice.

[8]Complexity is an asymptotic concept.

and perform a fixed number of iterations instead - for further computational savings (since we lose theoretical error bounds). While this may be beneficial in some cases (as it reduces complexity slightly), our numerical findings we have just mentioned suggest it may be worth considering doing $\mathcal{O}(\log(d_M))$ power-iteration - as our low #Modes choice may still give a controlled optimal truncation error.

Finally, even if controlling the cost may sometimes lead to a large inverse representation error (relative to K-FAC), it could still be the case that our step is as good or better than the K-FAC step. This is because the K-FAC step, while practically good, is not provably the best step one can take. Thus, our most relevant test is ultimately the numerical performance of the proposed algorithms. For CIFAR10 classification with VGG16_bn, our algorithms attained the target test accuracy $3\times$ faster.

Chapter Structure. In *Section 3.1* we look at the motivation behind our proposed algorithms: the decaying eigenspectrum of EA K-factors. In *Section 3.1.1* we devise deterministic bounds which directly apply to EA K-factors, while relying only on a very mild simplifying assumption (which the evidence suggests holds in practice). These bounds are relatively loose, so under further simplifying assumptions (which may not hold in practice), we prove tighter bounds in *Section 3.1.2*. As a complement, in *Appendix A3.1* we provide theory where we remove the key low-rank M_j assumption[9] to gain better intuition about the eigenspectrum behavior in the low d_M regime.

In *Section 3.1.3* we connect the theory to practice. Applying the devised bounds, we first argue that for sufficiently large d_M, we can control the K-factor truncation error by only keeping a constant number of modes (w.r.t. d_M). In *Section 3.1.3.1*, we then argue this can be exploited to reduce inversion time complexity by a power of d_M (ignoring $\log(d_M)$ terms in complexity). In *Section 3.1.3.2*, we look at whether the theory predicts this complexity improvement would give any computational benefits in standard practical settings or not. We conclude that for standard practical settings, the value of d_M is not large enough for our theory to ensure that computational benefits can be attained while controlling the error. Nevertheless, by investigating the situation numerically in *Section 3.1.3.3*, we obtain empirical evidence on a particular case-study which suggests our "sufficiently large d_M" regime starts for much lower values in practical situations than implied by theory. We thus have hope that substantial computational savings can be practically attained (for the K-factor inverse computation) while controlling the truncation error.

[9]Which does hold in practice.

In *Section 3.2*, we propose two optimizers (R-KFAC in *Section 3.2.1* and SRE-KFAC in *Section 3.2.2*) which use rNLA to exploit the eigenspectrum decay, as described above. Drawing upon our discussion and results in *Section 3.1*, we then examine how we may choose the target number of modes for our rNLA algorithms, in *Section 3.2.3*. We state our choosing strategy for this chapter. In *Section 3.2.4*, we focus on the fact that our rNLA algorithms, by directly obtaining (an approximation for) the leading modes, introduce a further error (which we call projection error). We discuss how to control this error (thus ensuring the total error is no larger than desired) by correctly choosing the rNLA hyperparameters. Importantly, the number of power iterations has to be of $\mathcal{O}(\log(d_M))$ to control the projection error. This means the complexity of our error-controlled *Randomized KFACs* will be $\mathcal{O}(\log(d_M)d_M^2)$. We end *Section 3.2* by highlighting some important considerations relating to *Randomized KFACs*.

In *Section 3.3* we test the two proposed optimizers on CIFAR10 classification with VGG16_bn, and observe that in this particular case, the reduced time-complexity practically translates to speeding up K-FAC 2.5× and 3.3× in terms of *time per epoch* and *time to target accuracy* respectively. Our algorithms outperform SENG [20] (w.r.t. time to target accuracy) for moderate and high target test accuracy, but slightly underperform for very high test accuracy.

We provide closing discussions of the chapter in *Section 3.4*.

3.1 Motivation: Exponential Eigenspectrum Decay

Recall from *Section 1.5* that the K-factors are not used directly. Instead, an exponential average is held (see *lines 4* and *8* of *Algorithm 1.1*). Thus, in this section we study the eigenspectrum of matrices of the form

$$\bar{\mathcal{M}}_k := \rho\bar{\mathcal{M}}_{k-1} + (1-\rho)M_kM_k^T = \rho^k\bar{\mathcal{M}}_0 + (1-\rho)\sum_{i=1}^{k}\rho^{k-i}M_iM_i^T,\ \ \forall k \in \mathbb{Z}^+, \quad (3.1)$$

where $\bar{\mathcal{M}}_0$ represents the initialization of the K-factor, and $M_iM_i^T$ represents the new information coming at the i^{th} update (with $M_i \in \mathbb{R}^{d_M \times n_M}$).

In *Sections 3.1.1 - 3.1.2* (and also *Appendix A3.1*), we prove eigenspectrum bounds which all fall under the generic form:

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_1(\bar{\mathcal{M}}_k)} \leq A\rho^N + B, \quad (3.2)$$

holding with probability at least p_D individually for each $N \in \mathbb{Z}^+$, and for some $r \in \mathbb{Z}^+$. We also have $A > 0, B \geq 0$, and $\rho \in (0,1)$ as is standard for exponential averages. The

assumptions[10] we make on M_i determine A, B, p_D, r, and the relationship between each other (sometimes further bound hyperparameters, which we can choose, are also involved). The assumptions we make are different in the three subsections (*3.1.1*, *3.1.2*, *A3.1*), and are in the order of increased complexity of results.

In all cases, we can use the bound (3.2) to obtain how many modes we need to retain to ensure any relative truncation error[11] $\epsilon > B$ with probability p_D. By enforcing that the r.h.s. of (3.2) is less than or equal to ϵ, and choosing N_ϵ to be the smallest N which satisfies this desire, we get that:

- At most $N_\epsilon r$ modes are above $\epsilon\lambda_1(\bar{\mathcal{M}}_k)$ with probability at least p_D, where $N_\epsilon = \left\lceil [\log(\epsilon - B) - \log(A)] / \log(\rho) \right\rceil$, $\forall \epsilon > B$.

From here, we would like to be able to say how the number of modes we need to keep ($N_\epsilon r$) scales with d_M, while keeping a high and constant p_D (is $N_\epsilon r$ constant w.r.t. d_M? or a constant fraction of d_M? or something else?). This is important because by directly obtaining only the first $N_\epsilon r$ modes (instead of performing a full EVD), we can reduce the time-complexity from $\mathcal{O}(d_M^3)$ to $\mathcal{O}(N_\epsilon r \log(d_M) d_M^2)$, provided $N_\epsilon r < d_M$, while maintaining a relative truncation[12] error controlled by ϵ. How $N_\epsilon r$ scales with d_M would tell us whether we can reduce the complexity by a power of d_M (ignoring the negligible $\log(d_M)$ factor), by a multiplicative factor, or not at all (with error controlled by ϵ). Different scaling of $N_\epsilon r$ with d_M are achieved under different classes of assumptions on M_i.

The main goal of our theory in *Sections 3.1.1 - 3.1.2* (and optionally *Appendix A3.1*) is to understand which assumptions imply the two important behaviours of $N_\epsilon r$ as d_M increases (for constant fixed p_D): $N_\epsilon r \propto 1$ and $N_\epsilon r \propto d_M$. Then, we want to understand which assumptions hold for FC layers and which hold for Conv layers, and thus which results we can "reasonably" expect to hold (we treat this aspect in *Section 3.1.3*).

[10] We will relate these assumptions to the practical situation for the FC layer and the Conv layer in *Section 3.1.3*. Therein, we also critically discuss the implied stationarity of M_i's distribution (in the basis of our assumptions) - relevant for our probabilistic bounds in *Sections 3.1.2* and *A3.1*.

[11] Note $\epsilon \leq B$ is not attainable.

[12] As we shall see in *Sections 3.1.4* and *3.2.4*, directly obtaining the modes introduces a further *projection error* (recall *Section 2.2*). Still, we shall see the total error can be controlled within a factor of 3× the (optimal) truncation error. From an asymptotic cost point of view, this requires $\mathcal{O}(\log(d_M))$ power iterations in our rNLA algorithm, each of which costs $\mathcal{O}(d_M^2)$ - so controlling the projection error requires worsening the time-complexity by a $\log(d_M)$ multiplicative factor.

We refer to the *optimal truncation error* as simply the *"truncation error"*. That is because the optimal truncation error is the only error component inherent to low-rank truncation. The remaining part of the error can be associated to having a poor low-rank estimate. The 2-norm is always the one used for errors unless otherwise specified.

The theory in *Sections 3.1.1, 3.1.2*, and the optional A3.1 makes assumptions and gets results as follows

1. In *Section 3.1.1*, we assume $\text{rank}(M_i) \leq r_M \ \forall i$, with $r_M < d_M$ (low-rank M_i), but initially leave the relationship between r_M and d_M as arbitrary (*Proposition 3.1* and *Corollary 3.2*). We also assume that $\sigma_{\max}(M_i)$ is bounded from above and below. In this case, we get that $N_\epsilon r = N_\epsilon r_M$, with constant N_ϵ w.r.t. d_M. We also get $B = 0$ (so we can attain arbitrarily low ϵ). The results presented are deterministic. Importantly, there are two cases that we are interested in:

   (a) **Assume r_M is fixed w.r.t. d_M.** In this case, $N_\epsilon r = N_\epsilon r_M$ is fixed as d_M increases, and we can reduce complexity by a power of d_M (for fixed ϵ; ignoring $\log(d_M)$ terms).

   (b) **Assume $r_M = \lfloor a d_M \rfloor$, with some $a \in (0,1)$ fixed w.r.t. d_M.** In this case, $N_\epsilon r = N_\epsilon r_M = N_\epsilon \lfloor a d_M \rfloor$. Thus, we may be able to reduce the complexity by a constant multiplicative factor (ignoring $\log(d_M)$ terms), if a is sufficiently low (for fixed ϵ).

2. In *Section 3.1.2*, we improve the constant N_ϵ that we get in *Section 3.1.1* by devising probabilistic bounds. We maintain the same low-rank assumption, but remove the assumed lower-bound on $\sigma_{\max}(M_i)$ and instead assume the r_M non-zero singular values of M_i are i.i.d. drawn form the same distribution D_σ with p.d.f. f_σ (of bounded support). There are also some further simplifying assumptions on the structure of M_i's SVD. The cases (a) and (b) of point 1 above also hold here, but this time we have better N_ϵ and the result holds with high probability. We only discuss the case when r_M is fixed w.r.t. d_M in this section for a more succinct exposition. We omit discussing the other case, as this is better represented in practice by the analogous 3(b).

3. In *Appendix A3.1* we remove the low-rank assumption on M_i, and assume that all d_M singular values of M_i are i.i.d. draws from the same distribution $\mathscr{D}_\sigma$ with c.d.f. $F_\sigma(\cdot)$ of bounded support. Initially, we leave F_σ as general (*Proposition A3.1*, *Corollary A3.4*). We then use these bounds to treat two particular cases which are very similar to the two cases in 1 above, and arrive to the same qualitative conclusion in both cases:

(a) **Assume F_σ concentrates mass at a sufficiently high rate below a sufficiently low threshold, as d_M increases.** This closely corresponds[13] to the case when r_M is fixed under the low-rank assumption (case 1(a) above), and we get that $N_\epsilon r \propto 1$ as d_M increases with high constant p_D. In this case we can reduce complexity by a power of d_M (for fixed ϵ; ignoring $\log(d_M)$ terms).

(b) **Assume F_σ is fixed with d_M.** This closely corresponds to the case when $r_M \propto d_M$ (case 1(b) above). We get that $N_\epsilon r = N_\epsilon \lfloor a d_M \rfloor$ for any fixed $a \in (0, 1)$ (which we choose this time[14]) as d_M increases, while keeping a constant p_D. A choice of a which ensures (i) $\lfloor a d_M \rfloor N_\epsilon < d_M$, (ii) $B \approx 0$, and (iii) p_D is high, all hold (s.t. the result is practically useful) can only be made if our fixed F_σ has "sufficiently much mass" around zero. This assumption on F_σ closely corresponds to directly assuming that a is "low enough" in 1(b). Under this further assumption on F_σ, we can reduce the complexity by a multiplicative factor of $1/(aN_\epsilon)$ (ignoring $\log(d_M)$ terms; for some $a < 1/N_\epsilon$).

In *Section 3.1.3* we shall see that for sufficiently large d_M, the practical situation best relates to the assumptions of 1(a) and 2(a) and 3(a). This holds for both FC and Conv layers. However, in practice, we may not be in these sufficiently large d_M regimes (as we shall see in *Section 3.1.3*). In this case, one may choose to give up the fixed low-rank assumption (or equivalent: 3(a)) and try to make sense of 3(b).

The numbers that theory predicts for $N_\epsilon r$ are relatively large in practice (both in the case when $N_\epsilon r \propto 1$ and $N_\epsilon r \propto d_M$) for realistic numerical values of assumed quantities. However, the numerical results in *Section 3.1.3.3* show that these numbers are better in practice (and we observe the same qualitative behavior).

A Small Side Note on the Theory. Notably, our theory views the error[15] from the lens of the change in the K-factors - using norm of difference in the K-Factors space to measure deviation from "standard K-FAC". However, we are really interested in the change in K-FAC step[16] - for which a better metric would be the change in the inverse K-Factor. At first sight, it may appear as the choice of our metric is a problem, because the K-factor error metric and the K-factor inverse error metric can

[13]Here by *"corresponds"* we mean they are very similar both in terms of the implied structure on M_i and in terms of how $N_\epsilon r$ scales with d_M for a fixed high p_D.

[14]Rather than having it by assumption as in 1(b).

[15]Introduced by truncating the K-factors.

[16]This is the quantity we want to be low.

diverge arbitrarily. For example when K-factors exhibit rapid eigenspectrum decay, truncating the lowest modes results in very low error in the K-factor space, but very high (infinite) error in the Inverse K-factors space (because the very low eigenvalue eigenmodes have very high value of inverse eigenvalues). However, because we are using regularized K-factors to obtain the step in practice, this divergence problem does not occur in practice (the situation gets practically better for relatively high regularization parameters). For regularized K-factors, the eigenvalues are bounded from above (irrespective of how low the original K-factor eigenvalues are). In this case, we can prove[17] that having the error in the K-factor space (due to truncating low modes) converging to zero leads to the error in the inverse regularized K-factor space to also go to zero. The intuition for this is that the very low eigenvalues of the K-factors give eigenmodes of the inverse regularized K-factors which are of eigenvalue of $1/(\lambda+\epsilon)$: this removes the problem of being able to inflict arbitrarily[18] high error in the inverse Matrix space with an arbitrarily low error in the Matrix space. To see this, note that a change of δ (by truncating[19]) in an eigenvalue of the K-factor results in a change of the corresponding eigenvalue of the inverse regularized K-factor of magnitude bounded as $|(\lambda+\lambda_i+\delta)^{-1}-(\lambda+\lambda_i)^{-1}| = |\delta|/|(\lambda+\lambda_i)(\lambda+\lambda_i+\delta)| \leq |\delta|/\lambda^2$, where $\{\lambda_i\}_i$ are the eigenvalues of the K-factor, λ is the regularization parameter, and we used the fact that we can only truncate eigenvalues[20] (so $\lambda_i+\delta=0$), and also the fact that $\lambda_i \geq 0$ (by the fact that K-factors are PSD). Notably, having higher regularization parameter will make the error in the inverse regularized K-factor space smaller (for the same error δ in the K-factor space). Thus, our theory - though can be improved to work with the more practically useful error in the inverse regularized K-factor space - gives us results which are still useful.

3.1.1 Deterministic Bounds for K-Factor Eigendecay

Let us consider

$$\bar{\mathcal{M}}_k := \rho^k \bar{\mathcal{M}}_0 + (1-\rho)\sum_{i=1}^{k} \rho^{k-i} M_i M_i^T, \;\; \forall k \in \mathbb{Z}^+, \tag{3.3}$$

where $\bar{\mathcal{M}}_0$ is the initialization we choose in our algorithm, and $M_i \in \mathbb{R}^{d_M \times n_M}$. The matrix $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$ represents an arbitrary[21] EA K-Factor after k updates to the

[17] Proof omitted here for brevity.
[18] By choosing K-Factor matrices with ever-lower values for the smallest eigenvalue.
[19] Notably, we can only choose $\delta \in \{-\lambda_1, -\lambda_2, ... -\lambda_i, ...\}$ when truncating.
[20] A similar proof works if we allow to change any λ_i by any $\delta \in \mathbb{R}^+$.
[21] As it has the same structure as any of the EA K-factors.

K-factor from the algorithm start, as produced by *lines 4 and 8 of Algorithm 1.1*. Note that the choice of $\bar{\mathcal{M}}_0$ can be such that we simplify our bounds results.

In practice, the initialization is made as $\bar{\mathcal{M}}_0 = \gamma I$ for some $\gamma > 0$. However, without loss of generality (under the case of identity-multiple initialization), we can take our initialization to be $\bar{\mathcal{M}}_0 = 0 \in \mathbb{R}^{d_M \times d_M}$, because we can merely place the $\rho^k \gamma I$ term that would result from the standard initialization into the regularization term: we would have a regularization term of the form $+(\lambda + \rho^k \gamma)I$, rather than the standard $+\lambda I$. For the remainder of the theoretical part, we shall use the convention that $\bar{\mathcal{M}}_{-j} = 0 \in \mathbb{R}^{d_M \times d_M}$ for any $j \in \mathbb{Z}^+$.

Proposition 3.1 provides deterministic bounds which describe the eigen-decay of $\bar{\mathcal{M}}_k$ in (3.3) under a low-rank (r_M) assumption on M_i, and a bounded assumption on $\sigma_{\max}(M_i)$. The eigenvalues are sorted in descending order.

Proposition 3.1 (**Deterministic bounds describing eigenvalue decay of $\bar{\mathcal{M}}_k$**). *Consider the matrix $\bar{\mathcal{M}}_k$ in (3.3). Assume (1) $\sigma_{\max}(M_i) \in (\beta\sigma_{\max}, \sigma_{\max})$ $\forall i \in \mathbb{Z}^+$ for some fixed $\beta \in (0,1)$, $\sigma_{\max} > 0$, and (2) that rank$(M_i) \leq r_M$ $\forall i \in \mathbb{Z}^+$. Then, we have for any $N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}$*

1. *For $N \leq k-1$ (holds throughout the whole spectrum for large enough k):*

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{1}{(1-\rho)\beta^2}. \tag{3.4}$$

2. *For $N \geq k$ (transient case in the eigenspectrum tail):*

$$\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) = 0. \tag{3.5}$$

Proof. As a convention, the eigenvalues and singular values of $\bar{\mathcal{M}}_k$ are sorted in descending order, but the singular values of M_i are not sorted. We will use this convention throughout the whole *Section 3.1* for simplicity of exposition.

Weyl's monotonicity theorem states that $\sigma_{i+j-1}(A+B) \leq \sigma_i(A) + \sigma_j(B)$ $\forall$ i and j such that the indices (i, j, $i+j-1$) are well defined. Applying it to our case with $j \leftarrow r_M + 1$, and $i + j - 1 \leftarrow Nr_M + 1$, $A + B \leftarrow \bar{\mathcal{M}}_k$ and $B \leftarrow (1-\rho)M_k M_k^T$ we get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho\sigma_{Nr_M - r_M + 1}(\bar{\mathcal{M}}_{k-1}) + \frac{\sigma_{r_M+1}(M_k M_k^T)}{(1-\rho)^{-1}} = \rho\sigma_{(N-1)r_M+1}(\bar{\mathcal{M}}_{k-1}) \tag{3.6}$$

where the equality holds because rank$(M_i) \leq r_M$ ($\forall i \in \mathbb{Z}$) by assumption.

Part 1 ($N \leq k-1$): Applying (3.6) recursively $N \leq k-1$ times, we get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N \sigma_1(\bar{\mathcal{M}}_{k-N}). \tag{3.7}$$

Since $\bar{\mathcal{M}}_k$ are all SPSD their singular values are the same as their eigenvalues, so we have

$$\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}). \tag{3.8}$$

Dividing (3.8) by $\lambda_{\max}(\bar{\mathcal{M}}_k) > 0$ (as M_k's are SPSD with $\sigma_{\max}(M_k) > 0$), we get

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\lambda_{\max}(\bar{\mathcal{M}}_{k-N})}{\lambda_{\max}(\bar{\mathcal{M}}_k)}, \tag{3.9}$$

where $N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}$.

Now, we have the following upper-bound

$$\begin{aligned} \lambda_{\max}\big(\bar{\mathcal{M}}_{k-N}\big) &\leq (1-\rho) \sum_{i=1}^{k-N} \rho^{k-N-i} \lambda_{\max}\big(M_i M_i^T\big) \\ &\leq (1-\rho)\sigma_{\max}^2 \sum_{i=0}^{k-N-1} \rho^i = (1-\rho^{k-N})\sigma_{\max}^2, \end{aligned} \tag{3.10}$$

where the second inequality comes from using the assumed upper-bound on M_i's singular values (by assumption (1)). By assumption, we also have the following lowerbound

$$\lambda_{\max}(\bar{\mathcal{M}}_k) \geq (1-\rho)\lambda_{\max}(M_k M_k^T) \geq (1-\rho)\beta^2 \sigma_{\max}^2. \tag{3.11}$$

Substituting the bounds (3.10) and (3.11) into (3.9) gives

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{1}{\beta^2} \frac{1-\rho^{k-N}}{1-\rho} \leq \rho^N \frac{1}{(1-\rho)\beta^2}, \tag{3.12}$$

which is the desired inequality. This completes the proof for *Part 1.*

Part 2 ($N \geq k$): Applying (3.6) k times gives

$$\sigma_{kr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^k \sigma_1(\bar{\mathcal{M}}_0) = 0. \tag{3.13}$$

Since $N \geq k$, we thus have

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \sigma_{kr_M+1}(\bar{\mathcal{M}}_k) \leq 0. \tag{3.14}$$

Since $\bar{\mathcal{M}}_k$ is SPSD, its singular values (which are also non-negative) are also its eigenvalues. Thus, we have that $\forall N \geq k$, $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) = 0$, which completes the proof. □

Corollary 3.2 (**Deterministic bounds on the number of modes needed for a fixed accuracy, under the low-rank assumption**). *Under the assumptions in Proposition 3.1 we have that* $\forall k \in \mathbb{Z}^+$, *at most* $\min(N_\epsilon r_M, d_M)$ *eigenvalues of* $\bar{\mathcal{M}}_k$ *are above* $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$, *with*

$$N_\epsilon = \left\lceil \log\big([1-\rho]\beta^2\epsilon\big)/\log(\rho) \right\rceil. \tag{3.15}$$

Proof. First, note that we can combine *Parts 1* and *2* of *Proposition 3.1* to get

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{1}{(1-\rho)\beta^2}, \ \ \forall N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}. \tag{3.16}$$

Thus, using *Proposition 3.1*, we can ensure that at most Nr_M eigenvalues are above $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$ by enforcing the condition

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \frac{1}{(1-\rho)\beta^2}\rho^N \leq \epsilon. \tag{3.17}$$

Taking $\log(\cdot)$ of (3.17), and noting that $\rho \in (0, 1)$, we get that we need N to satisfy

$$N \geq \log([1-\rho]\beta^2\epsilon)/\log(\rho) \tag{3.18}$$

to ensure that at most Nr_M eigenvalues are above $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$. Choosing $N_\epsilon = \lceil \log([1-\rho]\beta^2\epsilon)/\log(\rho) \rceil$ (the smallest $N \in \mathbb{Z}$ satisfying (3.18)) yields our result. □

Note that the bounds in *Proposition 3.1* and *Corollary 3.2* only give useful (non-trivial[22]) bounds if $r_M \ll d_M$.

Corollary 3.2 gives the notable result that for sufficiently large d_M (compared to r_M), the number of modes to keep for a target relative truncation error ϵ is of the form $N_\epsilon r_M$, with constant N_ϵ. Thus, by computing only the first $N_\epsilon r_M$ modes directly, we can reduce our work from $\mathcal{O}(d_M^3)$ (for computing all the eigenmodes, as is standard with K-FAC through an EVD) to[23] $\mathcal{O}(N_\epsilon r_M \log(d_M) d_M^2)$ (assuming $N_\epsilon r_M < d_M$), if we are willing to accept a relative truncation error of ϵ (in the 2-norm). Importantly, there are two cases that we are interested in:

1. r_M **is fixed w.r.t.** d_M**.** In this case, $N_\epsilon r_M$ is fixed as d_M increases. Thus, the number of modes we need to save for a target relative truncation error ϵ is fixed w.r.t. d_M for sufficiently large d_M. In this case, we can obtain a (low-rank) EVD representation satisfying this desired error criterion in[24] $\tilde{\mathcal{O}}(d_M^2)$ for $d_M > N_\epsilon r_M$. Compared to using the EVD as is standard, in this case we can reduce the time-complexity by a power of d_M (once d_M exceeds the fixed threshold $N_\epsilon r_M$).

[22]Note that if the bounds become trivial, we need to keep all the modes, which is what the standard (EVD approach does. In that case, we would have no computational saving. The point is that we cannot ever require more computation than the standard approach, because we can at worst require to retain all the modes.

[23]The $\log(d_M)$ terms appears (perhaps somewhat unexpectedly at this stage) because of the need to perform $\mathcal{O}(\log(d_M))$ power iterations to control the projection error. See *Section 3.2.4* for details.

[24]Recall *Section 1.6.1* where we said $\tilde{\mathcal{O}}(f(d_M)) := \mathcal{O}\big(\log(d_M) f(d_M)\big)$ will be our notation for ignoring the $\log(d_M)$ term in our complexity discussions.

2. $r_M = \lfloor ad_M \rfloor$**, with some** $a \in (0,1)$ **fixed w.r.t.** d_M**.** In this case, the number of modes ($N_\epsilon r_M$) we need to keep for a target relative truncation error ϵ is (ignoring the flooring operator) proportional to d_M with proportionality constant $N_\epsilon a$. Thus, if a is sufficiently small s.t. $N_\epsilon a < 1$, by only computing the first $N_\epsilon \lfloor ad_M \rfloor$ modes directly, we can reduce the time-complexity by a factor of $1/(N_\epsilon a)$ (from $\mathcal{O}(d_M^3)$ to $\mathcal{O}(N_\epsilon a d_M^3)$) while maintaining a relative truncation error below ϵ. This is while ignoring the $\mathcal{O}(\log(d_M))$ multiplicative overhead to control the projection error of obtaining the leading modes (see *Section 3.2.4*). Depending on the actual values of a and $n_{\text{pwr-it}}$ (number of power iterations to control projection error), we may or may not get an actual cost-reduction in practice.

Through these two cases, *Proposition 3.1* captures all the cases that occur in practice (and that we are interested in). *Section 3.1.2* (and *Appendix A3.1*) present improved eigenspectrum decay bounds (as we described at the beginning of *Section 3.1*), but these will cover the same two cases.

Let us now look at an example where we plug realistic numbers into *Proposition 3.1* to asses the practical usefulness of the bound (for fixed r_M).

Example 3.3. *Plugging realistic values of* $\epsilon = 0.033$, $\beta = 0.5$ *and* $\rho = 0.95$, $r_M = n_B = 256$ *(r_M fixed w.r.t. d_M[25]) in Proposition 3.1 we get* $N_\epsilon = 152$*. This tells us we have to retain at least* $r_M N_\epsilon = 38912$ *eigenmodes to ensure we only ignore eigenvalues satisfying* $\lambda_i(\bar{\mathcal{M}}_k) \leq 0.033\lambda_{\max}(\bar{\mathcal{M}}_k)$.

Clearly, 38912 is very large[26], and *Proposition 3.1* would only be useful in practice for the case in *Example 3.3* if $d_M > 38912$. However, *Proposition 3.1* ensures the desirable qualitative behavior as $d_M \uparrow$ (described just before *Example 3.3*) under the assumption of $\text{rank}(M_i) \leq r_M$ (for r_M fixed w.r.t. d_M). We now look at the improved bounds under the same low-rank assumption in *Section 3.1.2*.

[25]Holds for sufficiently large d_M in both FC and Conv layers: see *Section 3.1.3*.

[26]Note that computing as many as 38,000 eigenmodes may be practically challenging from a numerical stability (accuracy) point of view. This may be because eigenvalue (or singular-value) computation algorithms involve iterative methods or numerical approximations, which can accumulate errors over many iterations. Thus, based on the theory in *Section 3.1.1*, whenever we are guaranteed to save computation cost, we could start having practical accuracy challenges. However we shall see in *Section 3.1.3.3* that in practice (for the considered case-study) the situation is better than implied by theory - we need to retain much fewer modes than implied by theory for $\epsilon = 0.033$. Thus, these accuracy problems are not a practical issue.

3.1.2 Probabilistic Bounds for K-Factor Eigendecay

Let us reiterate the form of $\bar{\mathcal{M}}_k$ for convenience

$$\bar{\mathcal{M}}_k := \rho^k \bar{\mathcal{M}}_0 + (1-\rho)\sum_{i=1}^{k} \rho^{k-i} M_i M_i^T, \ \ \forall k \in \mathbb{Z}^+. \tag{3.19}$$

where M_i are as in *Section 3.1.1.* As discussed in *Section 3.1.1*, we can choose $\bar{\mathcal{M}}_0 = 0$, for simplicity, and still encapsulate the standard initialization $\bar{\mathcal{M}}_0 = \gamma I$ ($\gamma > 0$).

Proposition 3.4 provides probabilistic bounds which describe the eigen-decay of $\bar{\mathcal{M}}_k$ in (3.3) under a low-rank (r_M) assumption on M_i. These are tighter than the ones in *Proposition 3.1* (compare *Remark 3.6* with *Proposition 3.1*).

Proposition 3.4 (**Probabilistic bounds describing eigenvalue decay of $\bar{\mathcal{M}}_k$**). *For the SPSD $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$ in (3.19) consider its eigenvalues sorted in descending order, and assume rank$(M_i) = r_M$. Further assume that M_i has the structure $M_i \overset{\text{SVD}}{=} U_i \Sigma_i V_i^T$ for all $i \in \mathbb{Z}^+$, where*

- *$\Sigma_i = diag(\sigma_i^{(1)}, ..., \sigma_i^{(r_M)})$ with $\sigma_i^{(j)} \in [\sigma_{\min}, \sigma_{\max}]$ i.i.d. $\forall i, j$, bounded $0 \leq \sigma_{\min}$ and $\sigma_{\max} < \infty$.*
- *$U_q \in \mathbb{R}^{d_M \times r_M}$, $U_q R \overset{\text{QR}}{=} X_q$, with $X_q \in \mathbb{R}^{d_M \times r_M}$ and $[X_q]_{i,j} \sim \mathcal{N}(0,1)$ i.i.d. $\forall i, j, q$. And $[X_q]_{i,j}$ are independent from $\sigma_i^{(j)}$.*
- *No assumption on V_i.*

In addition, assume that our initialization choice is the determinitic value $\bar{\mathcal{M}}_0 = 0$.

Then, for any $N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}$, we have the following bounds showing eigendecay, depending on the relationship between N and k:

1. *For $N \leq k-1$ (holds through out the whole spectrum for large enough k):*

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{(1-\rho^{k-N}) + c\Delta}{1-\rho} \frac{r_M}{d_M} \frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2} \tag{3.20}$$

holds with probability at least

$$p := \left[1 - (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M}\right] \left(1 - 2d_M \left[\frac{1}{(1+c\delta_{\min})}\right]^{1/\delta_{\min}} \left[\frac{e}{(1+c\delta_{\min})}\right]^{c}\right), \tag{3.21}$$

for any $c > 1$, $b \in [\sigma_{\min}, \sigma_{\max}]$, and

$$\Delta := \frac{(1-\rho)d_M}{r_M} \frac{\sigma_{\max}^2}{\mathbb{E}[(\sigma_i^{(j)})^2]}, \quad \delta_{\min} = \Delta/(1-\rho^{\max\{k-N,1\}}). \tag{3.22}$$

2. *For $N \geq k$ (transient case in the eigenspectrum tail):*

$$\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) = 0, \tag{3.23}$$

holds deterministically.

Proof. Importantly, throughout the proof we will consider the eigenvalues (which are also the singular values) of $\bar{\mathcal{M}}_k$ to be sorted in descending order ($\forall k$), such that $\lambda_1(\bar{\mathcal{M}}_k) = \sigma_1(\bar{\mathcal{M}}_k) \geq \lambda_2(\bar{\mathcal{M}}_k) = \sigma_2(\bar{\mathcal{M}}_k) \geq ....$

Note however that the singular values of M_i are not sorted, for simplicity.

Proof for Statement 1 (when $N \leq k-1$). Proof outline:

1. Use Weyl's monotonicity inequality to deduce $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N})$
2. Show that $\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq (1+c\delta_{\min})(1-\rho^{k-N})\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]$ with high probability $\forall c > 1$ and some $\delta_{\min}$;
3. Argue that $\lambda_{\max}(\bar{\mathcal{M}}_k) \geq (1-\rho)b^2$ holds with probability (at least) $1-\left[\mathbb{P}((\sigma_i^{(j)})^2 < b^2)\right]^{r_M}$ when the event in point 2 holds.
4. Combine 1, 2, and 3 to conclude upper bound on $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)/\lambda_1(\bar{\mathcal{M}}_k)$ with high probability ($\forall N \in \{1, 2, ...\lfloor (d_M-1)/r_M \rfloor\}$) .

Part 1: Weyl's monotonicity theorem states that $\sigma_{i+j-1}(A+B) \leq \sigma_i(A) + \sigma_j(B)\ \forall\ i$ and j such that the indices make sense. Applying it to our case with $j \leftarrow r_M + 1$, and $i + j - 1 \leftarrow Nr_M + 1$, $A + B \leftarrow \bar{\mathcal{M}}_k$ and $B \leftarrow (1-\rho)M_k M_k^T$ we get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \sigma_{Nr_M-r_M+1}(\bar{\mathcal{M}}_{k-1}) + \frac{\sigma_{r_M+1}(M_k M_k^T)}{(1-\rho)^{-1}} = \rho\sigma_{Nr_M-r_M+1}(\bar{\mathcal{M}}_{k-1}) \tag{3.24}$$

where the equality holds because $\text{rank}(M_i) = r_M$ by assumption. Applying this idea N times, we get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N \sigma_1(\bar{\mathcal{M}}_{k-N}). \tag{3.25}$$

Since $\bar{\mathcal{M}}_k$ are all SPSD their sigular values are the same as their eigenvalues, so we have

$$\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}). \tag{3.26}$$

Note that we require $N \in \{1, 2, ...\lfloor (d_M-1)/r_M \rfloor\}$ for the above equations to make sense.

Part 2: We will use Matrix Chernoff with Intrinsic Dimension. For that, we require $\mathbb{E}[\bar{\mathcal{M}}_{k-N}]$. We have

$$\mathbb{E}[\bar{\mathcal{M}}_{k-N}] = (1-\rho)\sum_{i=1}^{k-N}\rho^{(k-N-i)}\mathbb{E}[U_i\Sigma_i^2U_i^T] \tag{3.27}$$

Now,

$$\mathbb{E}[U_i\Sigma_i^2U_i^T] = \sum_{j=1}^{r_M}\mathbb{E}[(\sigma_i^{(j)})^2]\mathbb{E}[u_i^{(j)}(u_i^{(j)})^T] = \sum_{j=1}^{r_M}\mathbb{E}[(\sigma_i^{(j)})^2]\frac{1}{d_M}I = \frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]I_{d_M\times d_M}, \tag{3.28}$$

where in the first equality we have used the independence of $\sigma_i^{(j)}$ and $u_i^{(j)}$, in the second equality we have used $E[u_i^{(j)}(u_i^{(j)})^T] = (1/d_M)I_{d_M\times d_M}$, coming from the fact that the marginal of every $u_i^{(j)}$ is the uniform distribution over the sphere of radius 1: $\mathbb{S}^{d_M-1}$. This follows directly from the distribution of U_i, noting the symmetry (details omitted for brevity). Finally, the third equality holds because $\sigma_i^{(j)}$ are i.i.d.

Substituting (3.28) into (3.27) and again using the fact that $\sigma_i^{(j)}$ are i.i.d., we get

$$\mathbb{E}[\bar{\mathcal{M}}_{k-N}] = (1-\rho)\sum_{i=1}^{k-N}\rho^{k-N-i}\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]I_{d_M\times d_M} = (1-\rho^{k-N})\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]I_{d_M\times d_M}. \tag{3.29}$$

To apply Matrix Chernoff with Intrinsic Dimension, we need to know the trace and largest eigenvalue of any M s.t. $M \succcurlyeq \mathbb{E}[\bar{\mathcal{M}}_{k-N}]$. First, note that knowing $\mathbb{E}[\bar{\mathcal{M}}_{k-N}]$ allows us to choose $M := \mathbb{E}[\bar{\mathcal{M}}_{k-N}]$. Thus, we get

$$D := \frac{\mathrm{tr}(\mathbb{E}[\bar{\mathcal{M}}_{k-N}])}{\lambda_{\max}(\mathbb{E}[\bar{\mathcal{M}}_{k-N}])} = d_M, \quad \lambda_{\max}(M) = (1-\rho^{k-N})\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2] \tag{3.30}$$

We also need to know an L s.t. $\lambda_{\max}(Y_i) \le L\ \forall i \in \{1,...,k-N\}$, with

$$Y_i := (1-\rho)\rho^{k-N-i}M_iM_i^T, \tag{3.31}$$

clearly, $\lambda_{\max}(Y_i) \le \lambda_{\max}((1-\rho)M_kM_k^T) \le (1-\rho)\sigma_{\max}^2$, so we can choose $L := (1-\rho)\sigma_{\max}^2$. Applying Matrix Chernoff with intrinsic dimension (see for example *Theorem 7.2.1* in [116]) using thse values of M and L, we get

$$\mathbb{P}\left(\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \le (1+\delta)(1-\rho^{k-N})\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]\right) \ge 1 - 2d_M\left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^{1/\delta_{\min}} \tag{3.32}$$

for all $\delta \ge \delta_{\min}$ with

$$\delta_{\min} := \frac{(1-\rho)d_M}{(1-\rho^{k-N})r_M}\frac{\sigma_{\max}^2}{\mathbb{E}[(\sigma_i^{(j)})^2]} = \frac{(1-\rho)d_M}{(1-\rho^{\max\{k-N,1\}})r_M}\frac{\sigma_{\max}^2}{\mathbb{E}[(\sigma_i^{(j)})^2]} \tag{3.33}$$

Note that the second equality in (3.33) holds because under the case $N \leq k-1$ we have that $\max\{k-N,1\} = k-N$ always holds[27].

Let us choose a $\delta := c\delta_{\min}$ for arbitrary $c > 1$ to simplify the form. We have

$$d_M\left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{1/\delta_{\min}} = d_M\left[\frac{1}{(1+\delta)}\frac{e^{\delta}}{(1+\delta)^{\delta}}\right]^{c/\delta} = d_M\left[\frac{1}{(1+\delta)}\right]^{1/\delta_{\min}}\left[\frac{e}{(1+\delta)}\right]^{c}$$
$$= d_M\left[\frac{1}{(1+c\delta_{\min})}\right]^{1/\delta_{\min}}\left[\frac{e}{(1+c\delta_{\min})}\right]^{c}. \tag{3.34}$$

Thus, we have

$$\mathbb{P}\left(\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq (1+c\delta_{\min})(1-\rho^{k-N})\frac{r_M}{d_M}E[(\sigma_i^{(j)})^2]\right) \geq$$
$$\geq 1 - 2d_M\left[\frac{1}{(1+c\delta_{\min})}\right]^{1/\delta_{\min}}\left[\frac{e}{(1+c\delta_{\min})}\right]^{c} \tag{3.35}$$

$\forall c > 1$, and with $\delta_{\min}$ as in (3.33).

Part 3: We have

$$\mathbb{P}[\lambda_{\max}(M_k M_k^T) < b^2] = \prod_{j=1}^{r_M}\mathbb{P}[\sigma_k^{(j)} < b] = (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M}, \tag{3.36}$$

where the last equality holds because $\sigma_k^{(j)}$ are i.i.d.

Part 4: First, note that the assumptions on our random variables imply $\{M_i M_i^T\}_i$ are independent, and so $M_k M_k^T$ is independent from $\bar{\mathcal{M}}_{k-N}$. Thus, the events "$\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq (1+c\delta_{\min})\frac{r_M}{d_M}\mathbb{E}[(\sigma_i^{(j)})^2]$" (in part 2) and "$\lambda_{\max}(M_k M_k^T) \geq b^2$" (in part 3) are independent events. Thus, by using (3.35) and (3.36) the probability that both these events occur is *at least*

$$p := \left[1 - (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M}\right]\left(1 - 2d_M\left[\frac{1}{(1+c\delta_{\min})}\right]^{1/\delta_{\min}}\left[\frac{e}{(1+c\delta_{\min})}\right]^{c}\right) \tag{3.37}$$

When both events occur, we have the following:

1. First, we combine (3.26) and (3.35) to get

$$\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^N(1-\rho^{k-N})(1+c\delta_{\min})\frac{r_M}{d_M}E[(\sigma_i^{(j)})^2]. \tag{3.38}$$

[27]We use the $\max\{k-N,1\}$ formulation to more easily combine the two cases later (it is merely an artifice for avoiding excessive case-by-case treatment in *Corollary 3.5*).

2. Next, we have that

$$\lambda_{\max}(\bar{\mathcal{M}}_k) = \lambda_{\max}\left((1-\rho)\sum_{i=1}^{k}\rho^{k-i}M_iM_i^T,\right) \geq (1-\rho)\lambda_{\max}(M_kM_k^T), \quad (3.39)$$

deterministically. Now under the specified events, we have $\lambda_{\max}(M_kM_k^T) \geq b^2$ and thus

$$\lambda_{\max}(\bar{\mathcal{M}}_k) \geq (1-\rho)b^2. \quad (3.40)$$

3. Finally, combining (3.38) with (3.40), we get that

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N(1-\rho^{k-N})\frac{1+c\delta_{\min}}{1-\rho}\frac{r_M}{d_M}\frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2}. \quad (3.41)$$

Since the two events occur together with probability at least p (see (3.37)), we conclude that

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N(1-\rho^{k-N})\frac{1+c\delta_{\min}}{1-\rho}\frac{r_M}{d_M}\frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2} \quad (3.42)$$

holds with probability at least p as shown in (3.37).

Now, choosing

$$\Delta := (1-\rho^{k-N})\delta_{\min} = \frac{(1-\rho)d_M}{r_M}\frac{\sigma_{\max}^2}{\mathbb{E}[(\sigma_i^{(j)})^2]}, \quad (3.43)$$

and making this substitution in (3.42) we get the desired result for *Statement 1* of *Proposition 3.4*. Note that in the second equality of (3.43) we used (3.33) to substitute in for $\delta_{\min}$.

Proof for Statement 2 (when $N \geq k$): Similarly to *Part 1* of proving *Statement 1 of Proposition 3.4*, we use Weyl's monotonicity Theorem to get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \sigma_{Nr_M-r_M+1}(\bar{\mathcal{M}}_{k-1}) + \sigma_{r_M+1}(M_kM_k^T) = \rho\sigma_{Nr_M-r_M+1}(\bar{\mathcal{M}}_{k-1}). \quad (3.44)$$

This time however, we have $N \geq k$, so we can only apply the inequality recursively k times to get

$$\sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \rho^k\sigma_{(N-k)r_M+1}(\bar{\mathcal{M}}_0) = 0, \quad (3.45)$$

where the equality holds because by our initialization choice we have $\bar{\mathcal{M}}_0 = 0$. Thus, since $\bar{\mathcal{M}}_k$ is SPSD, we have $\lambda_{Nr_M+1} = \sigma_{Nr_M+1} = 0\ \forall N \geq k$

This finishes proving Statement 2 of *Proposition 3.4*. □

Corollary 3.5 (**Bounding the number of modes needed for a fixed accuracy; Low-rank M_i, probabilistic setting**). *Under the assumptions of Proposition 3.4, we have that at most* $\min(N_\epsilon r_M, d_M)$ *eigenvalues of* $\bar{\mathcal{M}}_k$ *are above* $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$*, where*

$$N_\epsilon = \left\lceil \left(\log(\epsilon) - \log\left(\frac{r_M \mathbb{E}[(\sigma_i^{(j)})^2]}{d_M(1-\rho)b^2} + c\frac{\sigma_{\max}^2}{b^2} \right) \right) / \log(\rho) \right\rceil . \tag{3.46}$$

with probability at least

$$p = \left[1 - (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M} \right] \left(1 - 2d_M \left[\frac{1}{(1+c\delta_{\min})} \right]^{1/\delta_{\min}} \left[\frac{e}{(1+c\delta_{\min})} \right]^c \right), \tag{3.47}$$

for any $c > 1$*,* $b \in [\sigma_{\min}, \sigma_{\max}]$*, with* $\delta_{\min}$ *as in (3.22).*

Proof. By *Proposition 3.4*, combining the two cases together, we get that

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{1+c\Delta}{1-\rho} \frac{r_M}{d_M} \frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2} \tag{3.48}$$

holds with probability at least p as in (3.21). To see this, note that

1. $1 - \rho^{k-N} \leq 1$ holds within the scope of *Statement 1 of Proposition 3.4* (for $N \leq k-1$), so we can replace $[1-\rho^{k-N}] \leftarrow 1$ in inequality (3.20) to get that (3.48) holds with probability at least p when $N \leq k-1$.

2. Then, note that for $N \geq k$, (3.48) trivially holds with probability 1, since $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) = 0$ for any $N \geq k$ (by *Statement 2 of Proposition 3.4*). So we can say that for $N \geq k$, (3.48) holds with probability at least p for any $p \in (0,1)$. In particular, we can choose the p in (3.21), as long as it is a legit probability (and it is, this is why we used $\max\{k-N, 1\}$ instead of simply $k-N$ in *Statement 1 of Proposition 3.4*: we have $\max\{k-N,1\} = k-N$ under the scope of *Statement 1*, but using the "max" formulation also ensures p is a probability under the scope of *Statement 2* - so we can stick to referencing the same functional form for p, rather than providing branches!).

Now, this means that for a given $\epsilon \in (0,1)$ we want to choose N s.t.

$$\frac{\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{1+c\Delta}{1-\rho} \frac{r_M}{d_M} \frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2} \leq \epsilon. \tag{3.49}$$

This choice will ensure $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) \leq \epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$ holds with probability at least p as in (3.21). Taking logs of (3.49), noting $\rho \in (0,1)$, and rearranging, we get that we want N s.t.

$$N \geq \left(\log(\epsilon) - \log\left(\frac{1+c\Delta}{1-\rho} \frac{r_M}{d_M} \frac{\mathbb{E}[(\sigma_i^{(j)})^2]}{b^2} \right) \right) / \log(\rho). \tag{3.50}$$

Plugging in Δ from (3.22) and choosing N_ϵ to be the smallest integer N that satisfies (3.50), we get the N_ϵ stated in (3.46). Also note that if $N_\epsilon r_M > d_M$ we need to take all the modes, and there's d_M of them in total. $\square$

Importantly, note that if we could make a choice for $c = c(d_M)$ which would give that

- N_ϵ is (bounded by) a constant w.r.t. d_M (at least for sufficiently large d_M),
- and that p in (3.47) is lower-bounded by a high constant value for all sufficiently large d_M,

then how the number of modes we need to keep[28] ($N_\epsilon r_M$) scales with d_M would only be determined by how r_M scales with d_M. From here we could investigate the cases $r_M \propto 1$ and $r_M \propto d_M$, as we have done in *Section 3.1.1*, and we would have the same discussion with the same conclusions as we did before (but better constants and probabilistic statements).

In *Remark 3.6* we make a choice for $c = c(d_M) > 1$ which attains the two conditions above (others choices for $c(d_M)$ are possible). We present what the limits $\lim_{d_M \to \infty} N_\epsilon$ and $\lim_{d_M \to \infty} p$ are (for simplicity), and draw conclusions from there.

Remark 3.6 (**Behavior of the Probabilistic Bounds as** $d_M \to \infty$)**.** *Let us take* $c := 1 + 1/\sqrt{\ln(d_M)}$ *and take the limit as* $d_M \to \infty$ *in Proposition 3.4 and Corollary 3.5. In this asymptotic case, we get the following 2 statements:*

1. *The eigenspectrum decay of* $\bar{\mathcal{M}}_k$ *can be described by the bound*

$$\frac{\lambda_{Nr_M+1,\infty}(\bar{\mathcal{M}}_k)}{\lambda_{\max,\infty}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\sigma_{\max}^2}{b^2}, \tag{3.51}$$

   for any $N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}$.

2. *At most* $\min(N_\epsilon r_M, d_M)$ *eigenvalues of* $\bar{\mathcal{M}}_k$ *are above* $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$*, where*

$$N_{\epsilon,\infty} = \left\lceil \left(\log(\epsilon) - \log\left(\frac{\sigma_{\max}^2}{b^2} \right) \right) / \log(\rho) \right\rceil. \tag{3.52}$$

holding with probability at least

$$p_\infty := 1 - (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M}, \tag{3.53}$$

when r_M *is fixed w.r.t.* d_M *for any fixed* $b \in [\sigma_{\min}, \sigma_{\max})$.

[28] To ensure a relative truncation error of at most ϵ, with probability lower-bounded by a high constant.

Proof. take $\lim_{d_M \to \infty}(\cdot)$ with $c := 1 + 1/\sqrt{\ln(d_M)}$ in (3.20), (3.46), and (3.22) to get (3.51), (3.52) and (3.53) respectively. For obtaining (3.51) note that if $N \geq k$ the bound (3.20) trivially holds since $\lambda_{Nr_M+1}(\bar{\mathcal{M}}_k) = 0$ deterministically, and in particular we can say it holds with probability at least p as in (3.21) (so as in the proof of *Corollary 3.5*, we can focus on *Statement 1*, and say it holds $\forall N \in \{1, 2, ...\lfloor (d_M - 1)/r_M \rfloor\}$). □

Note that the limiting value of N_ϵ, namely $N_{\epsilon,\infty}$ in (3.52), is reached for *finite* d_M due to the $\lceil \cdot \rceil$ operator. Also note that when $r_M \propto d_M$ we have $p_\infty = 1$, $\forall$ fixed $b \in [\sigma_{\min}, \sigma_{\max})$. When r_M is fixed w.r.t. d_M, the value of $1 - (\mathbb{P}[\sigma_k^{(1)} < b])^{r_M}$ can be made 1 by choosing $b = \sigma_{\min}$ (but is typically very high even for larger values of b, for reasonable distributions of $\sigma_i^{(j)}$ and[29] $r_M \geq 100$).

Thus, for both cases ($r_M \propto 1$ and $r_M \propto d_M$), we can say that N_ϵ is a constant for sufficiently large d_M, and that p is lower-bounded by a high constant w.r.t. d_M for sufficiently large d_M. From here we could then draw the same qualitative conclusions as we had in *Section 3.1.1*, but with better constants and probabilistic statements: when aiming for the relative truncation error to be controlled by ϵ with high (fixed) probability, we have

1. if $r_M \propto 1$, then we need to keep a fixed number of modes as d_M increases, in which case we can reduce the complexity by a power of d_M for d_M sufficiently large (as before, ignoring the $\log(d_M)$ term in the complexity; recall *page 53*);

2. if $r_M = \lfloor a d_M \rfloor$ for some $a \in (0, 1)$, then we need to keep a fixed fraction of modes, in which case we can reduce the complexity by a multiplicative factor of $1/(N_\epsilon a)$ if a is sufficiently small ($a < 1/N_\epsilon$ as before, ignoring the $\log(d_M)$ term in the complexity; recall *case 2* on *page 54*).

We now look at two particular choices of distribution for $\sigma_i^{(j)}$ to get a sense of how much tighter the probabilistic bounds in *Proposition 4.3* and *Corollary 4.4* are than the deterministic ones in *Propositon 4.1*. We only consider the case of $r_M \propto 1$, for conciseness.

Example 3.7. *Let* $\sigma_i^{(j)} \sim \mathcal{U}([\sigma_{\min}, \sigma_{\max}])$ *with* $\sigma_{\min} = \sigma_{\max} - s$ *for some* $s \in (0, \sigma_{\max})$. *Choose* $b = \zeta\sigma_{\max} + (1 - \zeta)\sigma_{\min}$, *for some fixed* $\zeta \in (0, 1)$. *Under* $d_M \to \infty$ *with the choice of* c *as in Remark 3.6, we have:* $\lambda_{Nr_M+1,\infty}(\bar{\mathcal{M}}_k) \leq$

[29] $r_M \geq 100$ holds in practice as for FC layers r_M is the batch size.

$(1-[1-\zeta][s/\sigma_{\max}])^{-2}\rho^N\lambda_{\max,\infty}(\bar{\mathcal{M}}_k)$, $N_{\epsilon,\infty}=\lceil\log(\epsilon[1-(1-\zeta)(s/\sigma_{\max})]^2)/\log(\rho)\rceil$ *and* $p_\infty=1-\zeta^{r_M}$.

Thus, we see that in the limit[30] *as* $s/\sigma_{\max}\to 0$*, with the choice of* $\zeta:=s/\sigma_{\max}$*, we have* $\lim_{s/\sigma_{\max}\to 0}(p_\infty)=1-\lim_{s/\sigma_{\max}\to 0}(s/\sigma_{\max})^{r_M}=1$, $\lim_{s/\sigma_{\max}\to 0}(N_{\epsilon,\infty})=\lceil\log(\epsilon)/\log(\rho)\rceil$ *and* $\lim_{s/\sigma_{\max}\to 0}\lambda_{Nr_M+1,\infty}(\bar{\mathcal{M}}_k)\leq\rho^N\lim_{s/\sigma_{\max}\to 0}\lambda_{\max,\infty}(\bar{\mathcal{M}}_k)$*. This is better than in the deterministic bounds as the sub-unitary quantity multiplying* ϵ *is now vanished into a 1.*

For numerical comparison, using the same values as in the *Example 3.3* (based on the deterministic bounds in *Proposition 3.1* and *Corollary 3.2*) with the probabilistic bounds we get $N_\epsilon=67$ (and a probability p_∞ of 1) rather than 152 as we had it there.

Example 3.8. *Let* $\sigma_i^{(j)}\sim\mathcal{U}([\sigma_{\min},\sigma_{\max}])$ *with* $\sigma_{\min}=\sigma_{\max}-s$ *with* $s=0.97\sigma_{\max}$*. Take the practical values* $\rho=0.95$, $\epsilon=0.033$, $r_M=256$*. Choose* $b=0.97\sigma_{\max}+0.03\sigma_{\min}$*. Under* $d_M\to\infty$ *with the choice of* c *as in Remark 3.6, we have* $\lambda_{Nr_M+1,\infty}(\bar{\mathcal{M}}_k)\leq(1/(1-0.97\cdot 0.03)^2)\rho^N\lambda_{\max,\infty}(\bar{\mathcal{M}}_k)\leq(1.061)\rho^N\lambda_{\max,\infty}$, $N_\epsilon=\lceil\log(0.033[1-0.97\cdot 0.03]^2)/\log(0.95)\rceil=68$ *and* $p_\infty\approx 1-4.1\cdot 10^{-4}$*. So we see that, at least for this particular case, we rapidly enter the asymptotic regime (even when* $0<s/\sigma_{\max}\approx 1$*). We also see that in this case we need to retain* $N_\epsilon r_M=256\cdot 68=17408$ *modes to have a high probability of ensuring our relative truncation error is below* $\epsilon=0.033$*.*

Example 3.8 says we need to retain $256\cdot 68=17408$ modes to have a probability of approximately 0.9995 of only truncating eigenvalues below 3.3% of the maximum eigenvalue when $d_M\gg r_M$. This is better than the numbers obtained based on the deterministic bounds in *Proposition 3.1* (see *Example 3.3*), but the implied eigenvalue decay is still more modest than we see in practice, as we shall see in *Section 3.1.3.3.*

Nevertheless, as stated before, the theory outlines the important idea that for a fixed target relative truncation error[31] ϵ, we *only need to keep a number of modes that is of the form* [Constant]$\cdot r_M$. From here, we can conclude how different scalings of r_M with d_M can be translated to different types of reduction in the complexity of obtaining a (low-rank) eigenvalue representation (while maintaining a relative truncation error controlled by ϵ) - as discussed just after *Remark 3.6.*

As optional material, in (*Appendix A3.1*), we give up on the idea of a low-rank M_i ($\text{rank}(M_i)=r_M<d_M$) assumption. Instead, we assume all the d_M singular values

[30]Practically when the probability measure is concentrated around $\sigma_{\max}$ relative to how far away it is from the origin (so when the distribution of $\sigma_i^{(j)}$ approaches a Direc-delta distribution (thus deterministic).

[31]In the 2-norm, relative to the largest eigenvalue.

of M_i are i.i.d. drawn from the same (arbitrary) distribution $\mathscr{D}_\sigma$ with p.d.f. f_σ of bounded support. We then investigate two cases for $\mathscr{D}_\sigma$ which are analogous to the cases $r_M \propto 1$ and $r_M \propto d_M$ in terms of conclusions (and also similar in terms of M_i spectrum behavior). In *Section 3.1.3*, we shall see that for d_M sufficiently large, we have r_M fixed with d_M (i.e. $r_M \propto 1$) for both FC and Conv layers. However, eliminating the low-rank assumption is helpful for investigating the non-asymptotic situation. This practically arises for *some* (although comparatively few) Conv layers (in standard practical settings). The material in *Appendix A3.1* is however not strictly required for the main thread of the thesis.

We now look at how the practical situation for FC and Conv layer connects to our theory developed in *Sections 3.1.1 - 3.1.2*.

3.1.3 Connecting Theory to Practice

The theoretical discussion in *Sections 3.1.1 - 3.1.2* concluded different qualitative behaviors of the EA K-factor eigenspectrum, each based on different assumptions about the "raw" K-Factors $M_i M_i^T$. For practical reasons, we now want to deduce the type of behavior we get for FC layers and Conv layers. While some of the assumptions can be fully justified (directly relating to the layer type), others were made as simplifications (and thus do not necessarily hold in practice). The simplifying assumptions of *Section 3.1.1* were very mild, and one should expect they hold well in practice (see *Appendix A3.2*). Conversely, the simplifying assumptions of *Section 3.1.2* (and *Appendix A3.1*) may not hold in practice, implying the theoretical results in those sections do not hold "perfectly" in practice.

The aim of this section (all *Section 3.1.3*) is to say, for FC and Conv layers, which one out of the multiple theoretical situations discussed in *Sections 3.1.1 - 3.1.2* would best resemble the practical situation at hand. We begin by discussing the assumptions.

The Simplifying Assumptions. Our key assumption is $\text{rank}(M_i) \leq r_M$, for some fixed r_M as d_M increases (and $M_i \in \mathbb{R}^{d_M \times n_M}$). Any other assumption is considered simplifying. The simplifying assumptions made in *Sections 3.1.2* and *Appendix A3.1* (where we develop tighter bounds) might be harder to accept as holding in practice, especially the fact that M_i are independent and identically distributed (which is an implication of the structural assumptions on M_i). However, the looser bounds of *Section 3.1.1* yield the same qualitative conclusion about the number of modes we need to keep, with the only simplifying assumption that $\forall i \in \mathbb{Z}^+$, $\sigma_{\max}(M_i)$ is bounded from above and from below, strictly away from zero. This assumption is very mild

and one should expect it to hold well in practice[32] (see *Appendix A3.2*). Note we can even enforce this assumption to hold in our practical algorithm by re-scaling[33] $M_i M_i^T$ if $\left\|M_i M_i^T\right\|_2^2 \notin (\beta^2\sigma_M^2, \sigma_M^2)$ (for some fixed β, σ_M of our choice). However, this is not required in practice.

Notably, the i.i.d.-ness of M_i implies a strong stationarity of M_i's distribution as optimization progresses (affects only *Sections 3.1.2* and *A3.1*). Since M_i generally depend on the parameters values θ, it is unlikely for this strong-stationarity to hold in practice. Importantly however, we could prove qualitatively similar results (though with slightly worse constants) and draw similar conclusions even in the absence of our assumptions on M_i structure (assumed in *Proposition 3.4*), and the i.i.d.-ness implied by them. In fact, no form of stationarity needs to be assumed (or implied by assumptions) provided we get certain bounds to hold. Strictly speaking we could prove a similar result (to *Proposition 3.4*) by only assuming:

1. $\mathbb{E}[\sigma_{\max}(M_i)^2] \leq \hat{s}^2$ for some fixed $\hat{s} > 0$ and all $i \in \mathbb{Z}^+$. This would ensure we can still perform *Step 2* of *Proposition 3.4's proof* (see proof outline at page 56) - as this is the most relaxed assumption on the singular values which enables applying the Matrix Chernoff Bounds. When doing this, the final result (3.20) would have the value $\mathbb{E}[(\sigma_j^{(i)})^2]$ replaced by $\hat{s}^2$. This change would propagate into all other implications, and would in principle loosen the bound, as the worst-case singular value second moment is not equal tot the typical one anymore.

2. Any assumption which implies $\lambda_{\max}(\bar{\mathcal{M}}_k) > \tilde{b}^2$ with high probability, for some $\tilde{b} > 0$. This would ensure *Step 3* of *Proposition 3.4*'s proof can still be done (see proof outline at page 56). We will not give details about what better assumptions one could make. Our assumed structure reduced proving the lower-bound required in *Step 3* of *Proposition 3.4*'s proof to a single line of proof. One could make more realistic assumptions at the expense of lengthening this part of the proof. The more realistic assumptions may further loosen the bound.

[32]Our M_i cannot be infinite in practice. Infinite M_i can only come from infinite values in the forward or backward pass, in which case the training cannot work - a much bigger problem than our theory not holding. Technically, an $M_i = 0$ (thus a $\sigma_{\max}(M_i) = 0$) may occur in practice, but this is extremely unlikely. Even if this happens, we can still get our theory to hold by skipping the EA K-factor update if $M_i = 0$. Thus, our only concern can be that there exists a sub-sequence for which $\sigma_{\max}(M_k) \to 0$, and/or one for which $\sigma_{\max}(M_k) \to \infty$. However, even in these unfavorable cases, some bounds still hold for any finite training period (finite k): so we can still apply our theory even if we had these problematic sub-sequences. The only problem would arise at $k \to \infty$, but we never use this limit for any results in *Section 3.1.1*. In *Appendix A3.2*, we show numerical evidence for a particular case-study where we have more than our assumption just holding: it holds with very high value of β for most K-factors (i.e. $\sigma_{\max}(M_i)$ is almost constant across i).

[33]Or adding some constant to M_i, if $M_i = 0$.

These relaxed assumptions could be used instead of the given ones and still draw qualitatively similar conclusions. However, we preferred the simpler (more restrictive) assumptions for better clarity. We can similarly relax the assumptions made in the more complex theory of *Section A3.1* while still preserving the conclusions qualitatively.

The Key Assumption. Our key assumption to check is whether we have a fixed upper-bound on M_i's rank as d_M increases. Whether this holds or not will change how the number of modes required scales with d_M. In particular, *Sections 3.1.1 - 3.1.2* (and *Appendix A3.1*) tell us that if this assumption holds (and under some further simplifying assumptions), we only need to keep a fixed number of modes as d_M increases, while controlling the relative truncation error below a fixed threshold (i.e. while controlling ϵ). We want this behavior because it allows us to replace our K-factors EVD with a low-rank representation based on a fixed number of modes (w.r.t. d_M), while controlling ϵ. Under this case, by directly computing[34] the required number of leading modes (to ensure our tolerance), we can improve the time complexity of our inversion scheme by almost[35] a power (from $\mathcal{O}(d_M^3)$ to $\tilde{\mathcal{O}}(d_M^2)$), as we shall see in *Section 3.2*.

Our key assumption holds for both FC and Conv layers, namely: $\exists r_M$ (fixed w.r.t. d_M) s.t. $\forall d_M \geq r_M$ we have $\text{rank}(M_i) = r_M$ ($M_i \in \mathbb{R}^{d_M \times n_M}$). This is purely based on K-factors shape: we have n_M being constant w.r.t. d_M, so we have $r_M = n_M$. To see this, we need to look at the K-factors size in terms of basic layer parameters. Recall that our M_k can be either one of $\{A_k^{(l)}\}_{l=1}^{n_L}$, or one of $\{G_k^{(l)}\}_{l=1}^{n_L}$. We have different situations for FC and Conv layers (recall *Section 1.5*):

1. **For FC Layers:** We have $A_k^{(l)} \in \mathbb{R}^{w_{l-1} \times n_{\text{B,K}}}$ and $G_k^{(l)} \in \mathbb{R}^{w_l \times n_{\text{B,K}}}$, where w_l is the width of layer l (the number of neurons in layer l), $n_{\text{B,K}}$ is the batch size used for K-factors estimation. Thus, we see that $d_M = w_l$ when $M_k = A_k^{(l)}$ and $d_M = w_{l-1}$ when $M_k = G_k^{(l)}$. In both cases, d_M is just a layer width. Typically, $n_{\text{B,K}}$ is chosen independently[36] of the architecture: a larger architecture does

[34]Directly computing the leading modes introduces a further projection error on top of the truncation error (see *Section 2.2*). In the remainder of *Section 3.1* we tacitly pretend this projection error is zero. This simplification helps us focus on the important aspects. Indeed, this extra error can be controlled as a multiple of the (optimal) truncation error by performing $\mathcal{O}(\log(d_M))$ power iterations, thus worsening the ideal quadratic complexity by a $\log(d_M)$ multiplicative factor (will see in *Section 3.2.4*). Thus, the most important part of controlling the total error is controlling the (optimal) truncation error.

[35]The final *Randomized K-FACs* complexity when controlling the error is $\mathcal{O}(\log(d_M)d_M^2)$ (see *Section 3.2.4* and recall our $\tilde{\mathcal{O}}(\cdot)$ notation on *page 26* of *Section 1.6.1*).

[36]Firstly, we typically have $n_{\text{B,K}} \leq n_{\text{B}}$, where n_{B} is the batch size used to estimate the gradient, otherwise the computations would be very inefficient. If $n_{\text{B,K}} > n_{\text{B}}$ we have: (i) for true NG an

not prompt one to increase $n_{\text{B,K}}$. Thus, when thinking of increasingly large architectures (larger d_M), the fixed low-rank assumption, $\text{rank}(M_i) = r_M$, holds with $r_M = n_{\text{B,K}}$ $(= n_M)$ fixed w.r.t. d_M, for all $d_M > n_{\text{B,K}}$.

2. **For Conv Layers:** We have $A_k^{(l)} \in \mathbb{R}^{(\kappa_w^{(l)}\kappa_h^{(l)}\eta^{(l)}+1)\times n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}}$ and $G_k^{(l)} \in \mathbb{R}^{\eta^{(l+1)}\times n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}}$, where $n_{\text{B,K}}$ is as for FC layers, $\eta^{(l)}$, $d_w^{(l)}$, $d_h^{(l)}$ are the number of channels, width, and height of the data tensor at layer l; $\kappa_w^{(l)}$, $\kappa_h^{(l)}$ are the convolution kernel width and height at layer l. Note that the second dimension of both $A_k^{(l)}$ and $G_k^{(l)}$ is $n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$. This is a multiple of what the second dimension was in the FC case. The factor $d_w^{(l-1)} d_h^{(l-1)}$ is constant w.r.t. $\kappa_w^{(l)}$, $\kappa_h^{(l)}$ and $\eta^{(l)}$, and only depends on the input image size, number and parameters of pooling layers before layer l. Thus, increasing any of the layer size parameters ($\kappa_w^{(l)}$, $\kappa_h^{(l)}$ and $\eta^{(l)}$) does not change the second dimension $n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$ (again thinking of a fixed batch size). Consequently, increasing d_M ($d_M = \kappa_w^{(l)}\kappa_h^{(l)}\eta^{(l)} + 1$ when $M_k = A_k^{(l)}$; $d_M = \eta^{(l)}$ when $M_k = G_k^{(l)}$), will not increase the second dimension of the K-factors.

   This means that when increasing d_M past the fixed value of $n_M = n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$, the matrix M_k will be of fixed low-rank (rank $r_M = n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$) due to its shape, similarly to the FC layer case. That is to say that our desired assumption holds for Conv layers too: we have $\text{rank}(M_i) \leq r_M$, $\forall d_M$ and with $r_M = n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$ $(= n_M)$ fixed w.r.t. d_M.

   Notably, $\kappa_w^{(l)}$, $\kappa_h^{(l)}$ are almost always set to 3 in practice, and are never increased, for computational efficiency reasons. Thus, only $\eta^{(l)}$ is ever increased when increasing the "size" of a Conv layer (when seeking more expresivity): so $\eta^{(l)}$ can be thought of as the only notion of size for Conv layers.

Note that in both cases, d_M is proportional to the underlying notion of layer size. Thus, we have the desirable claim that the rank of M_i is upper-bounded (by a fixed value) as the layer gets larger, once the layer is sufficiently large.

---

additional of $n_{\text{B,K}} - n_{\text{B}}$ forward passes are required, (ii) for empirical NG an additional of $n_{\text{B,K}} - n_{\text{B}}$ forward and backward passes are required. The batch size n_{B} stays fixed in practice even if we increase the architecture if our computational resources are fixed. Of course, larger architectures arise for large data-sets, and larger data-sets might prompt the usage of more GPUs. The batch size n_{B} typically scales linearly with the number of GPUs (the batch size per GPU is fixed). This ensures resources are utilized in the best way. However, we can still think of $n_{\text{B,K}}$ as being fixed even for increasingly many resources. We argue that it is best for us to choose $n_{\text{B,K}}$ to be the total K-factor batch size: this is the way in which we choose to deal with increasingly many GPUs in our distributed implementations presented in *Chapter 5*.

In both cases, we are ensured by the shape of M_i that $\text{rank}(M_i)$ is upper-bounded by a constant w.r.t. d_M. In particular, for any d_M "sufficiently large", our M_i is of fixed low-rank (we have $\text{rank}(M_i) \leq n_{\text{B}}$ whenever $d_M \geq n_{\text{B}}$ for FC layers; and $\text{rank}(M_i) \leq n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$ whenever $d_M \geq n_{\text{B,K}} d_w^{(l-1)} d_h^{(l-1)}$ for Conv layers). We have previously stated that this means we can improve the time complexity of our inversion scheme by almost a power (from $\mathcal{O}(d_M^3)$ to $\tilde{\mathcal{O}}(d_M^2)$).

Notably, complexity is an asymptotic concept so we have to understand whether this will translate to measurable improvements in practice. We now discuss in detail how our desired assumption holding implies we can improve complexity, and devise a better understanding of whether or not this complexity improvement is useful in practice.

3.1.3.1 Fixed Low-Rank M_i allows better Inversion Complexity

The plots in *Figure* 3.1 illustrate conceptually the behaviors we are interested in for a single but arbitrary K-factor. The values of ϵ, r_M are fixed, but are left as symbolic.

The gray line of the top plot (in *Figure* 3.1) shows what happens to $\text{rank}(M_i)$ as d_M is increased (all else kept fixed, including the data-set) - as described by the upper-bounds implied from the shape of M_i (see discussion in previous section). First, we get the rank upper-bound increasing linearly with d_M, but then it remains fixed once d_M exceeds r_M (r_M is fixed and depends on the corresponding layer, preceding pooling layers, and input image size, but not on the layer size parameters, as discussed).

Applying our theory in *Sections 3.1.1 - 3.1.2* with this $\text{rank}(M_i)$ behavior, we can obtain how many eigenmodes (of the EA K-factors) we need to keep while controlling the relative truncation error below a fixed (arbitrary) $\epsilon > 0$. The red curve represents this quantity. It shows that, $\forall d_M \geq N_\epsilon r_M$, the #Modes we need to keep is constant with d_M and equal to $N_\epsilon r_M$.

The situation expressed in the first plot tells us that in region III, we can obtain a better time scaling (than EVD) when computing K-factors inverse representation. To get this reduction, we must be willing to accept introducing a non-zero truncation error[37], whose value we can nevertheless control. Recall that the relative truncation error is guaranteed by theory to be below ϵ if we retain $N_\epsilon r_M$ modes, with $N_\epsilon \propto |\log(C\epsilon)|$ (for some constant $C > 0$). We can choose arbitrary $\epsilon > 0$, but smaller values will increase N_ϵ. In region III, by directly obtaining the first $N_\epsilon r_M$ modes only (rather than all d_M of them), we can reduce the leading order inversion time from

[37] And also a projection error which we pretend to be zero for now, for simplicity.

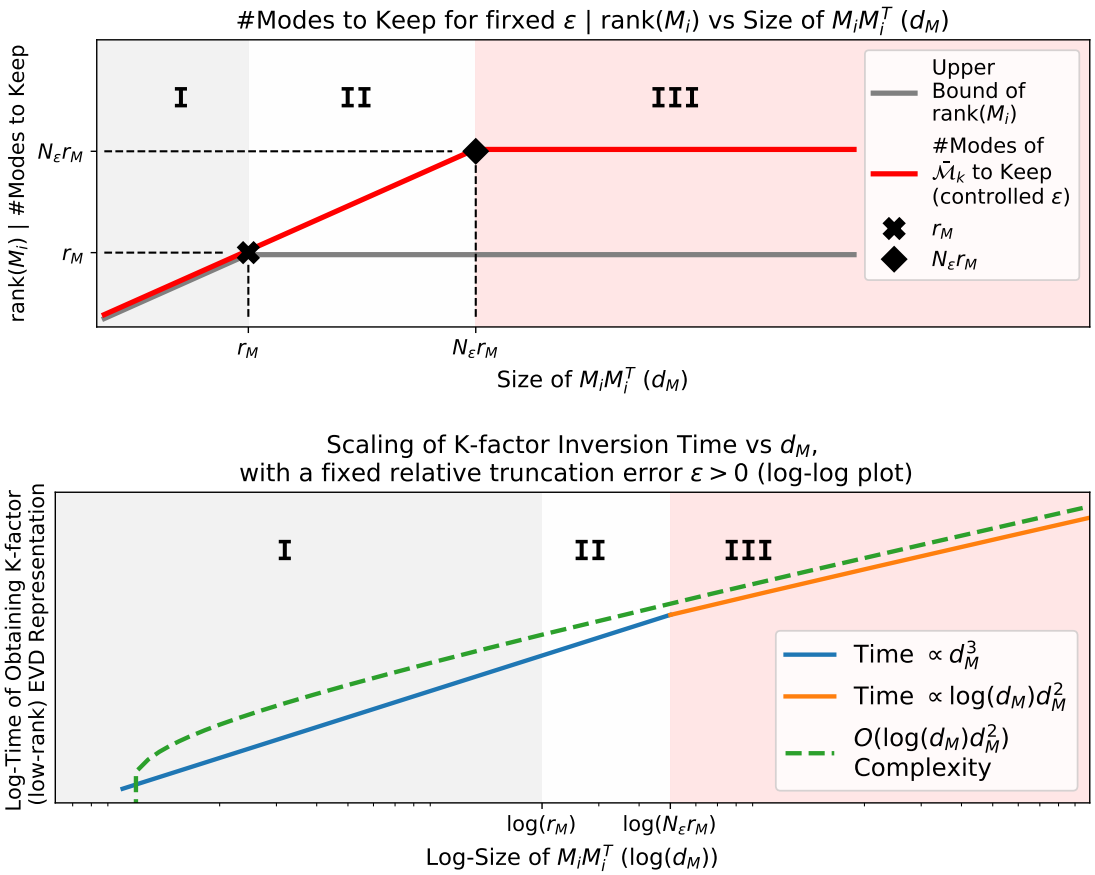


Figure 3.1: Conceptual Plots dissecting behavior of rank(M_i) vs d_M, #Modes to keep vs d_M, and showing Potential Inversion Scaling Improvements. The areas of the first plot and second plot (I, II, III) correspond to each other. In region II M_i is low-rank, but the #Modes we need to keep as predicted by theory ($N_\epsilon r_M$) is above d_M. In region III M_i is low-rank and $N_\epsilon r_M < d_M$ ($N_\epsilon r_M$ fixed).

scaling like d_M^3 to scaling[38] like $[N_\epsilon r_M] \log(d_M) d_M^2$ ($[N_\epsilon r_M]$ is constant). An exact way of how to obtain this will be discussed in *Section 3.2*. In regions I and II, our theory predicts we should save all the modes for our target ϵ, so we cannot make any savings.

In this way, we can reduce the time complexity of obtaining inverse representations of *arbitrary* tolerance $\epsilon > 0$ from $\mathcal{O}(d_M^3)$ to $\tilde{\mathcal{O}}(d_M^2)$. Note that reducing ϵ will increase N_ϵ, thus increasing region II and shrinking region III (region I is fixed w.r.t. ϵ). However, $N_\epsilon r_M$ is finite[39] for all $\epsilon > 0$, so our region III exists $\forall \epsilon > 0$. Therefore, since complexity is concerned with behavior at infinite d_M, we indeed reduce the time

[38]The $\log(d_M)$ term appears due to us having to also control the projection error when directly obtaining the modes, see *Section 3.2.4*.

[39]Recall theory in *Sections 3.1.1 - 3.1.2 (and optionally Appendix A3.1)*, for example *Corollary 3.2*.

complexity by (almost) a power for any $\epsilon > 0$. However, note that no computational improvement is obtained in regions I and II. These facts are shown in more detail in the second plot of *Figure* 3.1.

Our discussion so far raises a practical question: will our d_M's, of all, or at least some K-factors, fall into region III for practical net architectures and practically useful tolerances ϵ? Reformulating the question: will we see any benefit from this complexity reduction in practice? The short answer is that we should typically attain this: FC layers should fall in region III for all layer widths (sizes) with significant EVD cost, while Conv layers sometimes do and sometimes do not in practice. However, arriving at this answer is neither a hard truth nor straightforward. When applying theory alone for predicting which regions K-factors fall into for practical settings, we never get region III, due to our bounds being relatively loose (in *Section 3.1.3.2*). We draw our desired conclusion based on numerical case-study investigations (in *Section 3.1.3.3*). We discuss this next in more details, using theoretical and numerical case-studies of real, benchmark problem set-ups.

3.1.3.2 Theoretical Case-Studies: Is better Complexity theoretically expected to give Speed-ups in Practical Settings?

To better understand when our improved complexity may or may not help in practice (i.e. if d_M falls into region III or not), we consider the following thought experiment (no optimization solver to be ran). By considering the exact architecture and dataset, and using the known formulae for $A_k^{(l)}$ and $G_k^{(l)}$ sizes, we can theoretically investigate which layers are in the desirable asymptotic regime (of "sufficiently large d_M": region III) for practical training tasks. We consider three different tasks:

1. CIFAR10 classification with VGG11 (in *Table 3.1*). To have an investigation which better relates to subsequent numerical experiments, for VGG11 we replaced the (standard) final 512-in-10-out FC layer with two FC layers: (1) 512-in-256 out, (2) 256-in-10-out;

2. CIFAR10 classification with VGG16 (in *Table 3.2*);

3. Imagenet classification with resnet34 (in *Table 3.3*).

For each EA K-factor (2 per layer) we give all underlying size parameters, the K-factor size d_M, and maximum attainable rank r_M based on the shapes of $A^{(l)}$ and $G^{(l)}$. Based on our theory in *Section 3.1.2*, we give the number of modes we need to keep $(N_\epsilon r_M)$ to ensure the relative truncation error[40] is bounded by $\epsilon = 0.033$.

[40] A discussion about why this value of ϵ is practically useful is given in *Section 3.1.4*.

Table 3.1: CIFAR10 with VGG11: K-factors shapes and $N_\epsilon r_M$ values predicted by theory. The column R refers to which region of Figure 3.1 the K-factor falls into.

Layer	k_w	k_h	η_l	d_w	d_h	n_B	$A^{(l)}$				$G^{(l)}$			
							Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R	Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R
Conv1	3	3	64	32	32	128	577	131k	8.9mln	I	64	131k	8.9mln	I
Conv2	3	3	128	16	16	128	1153	32768	2.2mln	I	128	32768	2.2mln	I
Conv3	3	3	256	8	8	128	2305	8192	557056	I	256	8192	557056	I
Conv4	3	3	256	8	8	128	2305	8192	557056	I	256	8192	557056	I
Conv5	3	3	512	4	4	128	4609	2048	139264	II	512	2048	139264	I
Conv6	3	3	512	4	4	128	4609	2048	139264	II	512	2048	139264	I
Conv7	3	3	512	2	2	128	4609	512	34816	II	512	512	34816	I
Conv8	3	3	512	2	2	128	4609	512	34816	II	512	512	34816	I
			w_{l-1}	w_l										
FC9	-	-	512	256	-	128	512	128	8704	II	256	128	8704	II
FC10	-	-	256	10	-	128	256	128	8704	II	10	128	8704	I

We do this for each K-factor. With $\epsilon = 0.033$ and realistic values for other bounds parameters, we get $N_\epsilon = 68$ when using the more restrictive simplifying assumptions of *Section 3.1.2* (review *Example 3.8* in *Section 3.1.2*). Under the more acceptable simplifying assumptions of *Section 3.1.1*, N_ϵ would have been about 2 times that (recall *Section 3.1.1* for details). We used a batch size value of 128. The results are shown in *Tables 3.1 - 3.3*.

Table 3.2: CIFAR10 with VGG16: K-factors shapes and $N_\epsilon r_M$ values predicted by theory. The column R refers to which region of Figure 3.1 the K-factor falls into.

Layer	k_w	k_h	η_l	d_w	d_h	n_B	$A^{(l)}$				$G^{(l)}$			
							Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R	Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R
Conv1	3	3	64	32	32	128	577	131k	8.9mln	I	64	131k	8.9mln	I
Conv2	3	3	64	32	32	128	577	131k	8.9mln	I	64	131k	8.9mln	I
Conv3	3	3	128	16	16	128	1153	32768	2.2mln	I	128	32768	2.2mln	I
Conv4	3	3	128	16	16	128	1153	32768	2.2mln	I	128	32768	2.2mln	I
Conv5	3	3	256	8	8	128	2305	8192	557056	I	256	8192	557056	I
Conv6	3	3	256	8	8	128	2305	8192	557056	I	256	8192	557056	I
Conv7	3	3	256	8	8	128	2305	8192	557056	I	256	8192	557056	I
Conv8	3	3	512	4	4	128	4609	2048	139264	II	512	2048	139264	I
Conv9	3	3	512	4	4	128	4609	2048	139264	II	512	2048	139264	I
Conv10	3	3	512	4	4	128	4609	2048	139264	II	512	2048	139264	I
Conv11	3	3	512	2	2	128	4609	512	34816	II	512	512	34816	I
Conv12	3	3	512	2	2	128	4609	512	34816	II	512	512	34816	I
Conv13	3	3	512	2	2	128	4609	512	34816	II	512	512	34816	I
			w_{l-1}	w_l										
FC	-	-	10	512	-	128	512	128	8704	II	10	128	8704	I

Firstly, note that our relatively large value of $N_\epsilon = 68$ implied by theory gives very large $N_\epsilon r_M$ values (# Modes to keep) for all considered K-factors.

Secondly, we observe that all FC layers K-factors are in region II, with the exception of the K-factors of size 10. Note that computing the EVD of a size 10 matrix is a trivial task. Thus, we see that in practice for all K-factors sizes which have notable EVD cost, we have our K-factors sitting outside region I (in region II in particular).

Table 3.3: Imagenet (input randomly cropped to 224×224 pixels) with resnet34: K-factors shapes and $N_\epsilon r_M$ values predicted by theory. The column R refers to which region of Figure 3.1 the K-factor falls into.

Layer	$k_w = k_h$	η_l	d_w	d_h	n_{B}	$A^{(l)}$				$G^{(l)}$			
						Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R	Size 1 (d_M)	Size 2 (r_M)	$N_\epsilon r_M$	R
Conv1	7	64	224	224	128	3137	6.4mln	436mln	I	64	6.4mln	436mln	I
Conv2	3	64	112	112	128	577	1.6mln	109mln	I	64	1.6mln	109mln	I
5× Conv3	3	64	56	56	128	577	401k	27.3mln	I	64	401k	27.3mln	I
Conv8	3	128	56	56	128	1153	401k	27.3mln	I	128	401k	27.3mln	I
7× Conv9	3	128	28	28	128	1153	100k	6.8mln	I	128	100k	6823936	I
Conv16	3	256	28	28	128	2305	100k	6.8mln	I	256	100k	6.8mln	I
11× Conv17	3	256	14	14	128	2305	25k	1.7mln	I	256	25k	1.7mln	I
Conv28	3	512	14	14	128	4609	25k	1.7mln	I	512	25k	1.7mln	I
5× Conv29	3	512	7	7	128	4609	25k	426k	I	512	6272	426k	I
		w_{l-1}	w_l										
FC	-	512	1000	-	128	512	128	8704	II	1000	128	8704	II

Thirdly, the Conv layers K-factors are sometimes in region I and sometimes in region II. For a relatively low image size (32×32 pixels) as in the case of the CIFAR10 dataset, the $\bar{\mathcal{A}}_k^{(l)}$ K-factors of the later Conv layers are in region II. However, all other Conv layers K-factors fall into region I. Note that r_M values are particularly large for early Conv layers in the case of CIFAR10 dataset (*Tables 3.1 - 3.2*) and for all Conv layers in the case of Imagenet (image size 224×224, see *Table 3.3*).

Overall, note that no K-factor falls in region III for the investigated practical settings (according to theory). So far, our situation is as follows:

1. We theoretically found that due to the K-factor construction paradigm and matrix sizes involved, we can reduce the complexity of obtaining K-factor inverse representation by a power (while controlling the relative truncation error below a desired amount).

2. Since complexity is an asymptotic concept, we then asked ourselves whether we would indeed see any computational savings in practice. We saw that theory guarantees we can obtain this savings if our K-factors fall into region III.

3. We saw just now that for practical situations, when plugging in numbers in our theory, no K-factor falls into region III. At best, we had some K-factors falling into region II. Thus, we cannot deploy our theory to claim any computational savings can be made in these practical settings, though asymptotically (as

$d_M \to \infty$, all else fixed) we are guaranteed to reduce computational effort by (almost) a power of d_M (while controlling truncation error).

4. However, our theoretical bounds are relatively loose. In particular, our bounds based on the fixed low-rank M_i assumption never exploited the fact that M_i's eigenspectrum might have a significant decay itself[41]. We also used some bounds that were not tight for all cases. As such, one can still hope that in practice our N_ϵ is much smaller than inferred by our theory, and that most K-factors theoretically falling into region II might in practice find themselves in region III. Note our theory can only get[42] $N_\epsilon \geq 1$, which implies only K-factors which theoretically fall into region II (and not region I) stand a chance for hitting region III in practice. However, a more sophisticated theory (accounting for both low-rank M_i structure, and for the eigenspectrum decay within M_i through further M_i structural assumptions) may[43] allow for equivalent values of $N_\epsilon r_M$ below r_M (equivalent values of $N_\epsilon < 1$). So in practice, we might see region III behavior even for K-factors that fall into region I based on our theory. This is of course more unlikely.

5. Furthermore, even K-factors not practically falling into region III can get complexity improvements, though by less than a power of d_M, for example by a multiplicative factor. To see this, note that in *Appendix A3.1* we removed the low-rank assumption on M_i, and under some favorable assumptions[44] we still managed to conclude that we can attain our error constraint while only saving a

[41] One practical (and approximate) quick fix to get bounds which incorporate substantial singular-value decay of M_i (should it exist) is to apply the developed theory with an r less than the true rank, chosen to include only the modes with "significantly large" singular. That is, apply theory by pretending all singular values below a threshold are zero. Of course this would only be an approximation, but is also a starting point for thinking about how one may proceed doing an exact theory. The approximation would be good when all the columns of M_i would *approximately* lie in subspace of smaller dimension than the true rank. A slight complication may arise if this approximate low-rank varies with iterations - in which case our theory would need to be generalized a little.

[42] $N_\epsilon \geq 1$ in our theory: due to the form $N_\epsilon = \lceil N_N \rceil$, where $N_N > 0$.

[43] There exist some structural assumptions on M_i that allow us to say we need less than r_M modes for some ϵ values. Our theory does not allow for that. The simplest (but extreme) example is assuming all M_i's are identical, of rank $r_M < d_M$, and that M_i singular values are very rapidly decaying. In that case we can easily get that the eigenspectrum of the EA K-factor $\bar{\mathcal{M}}_k$ is the same as the singular spectrum squared of M_i. Knowing the latter allows us to give exactly how many modes we need to keep (which is certainly less than r_M since rank M_i is r_M). Of course, we have to choose the right structural assumptions - this is just an example showing how incorporating structural assumptions on M_i's singular spectrum may allow us to theoretically say we need to only retain less than r_M modes in some instances..

[44] Both simplifying assumptions and structural assumptions about the eigenspectrum of M_i exhibiting "relatively much decay".

constant fraction (=: $\phi < 1$) *of total modes* as d_M increases[45] (rather than all the modes). We argued that this would allow us to reduce inversion complexity by multiplicative factor (ignoring the multiplicative $\log(d_M)$ factor), rather than a power. While asymptotically smaller, this type of computational saving might still be large in practice if this constant fraction we need to keep (ϕ) is low.

6. Indeed, we shall now look at a numerical case study (experiment) where the d_M of all K-factors is increased with all else fixed. In this setting, we can empirically study which region the K-factor falls into, based on how the #Modes we need to keep scales with d_M. Our numerical results show that a substantial number of K-factor falls into region III, and out of the remaining K-factors, most of them have a behavior that still allows for complexity improvements (while controlling the error). Of course, this is just a particular practical setting, and situations may differ in general. Nevertheless, the results do show that our theory tends to overestimate N_ϵ, and thus that the situation tends to be better in practice than implied by theory.

3.1.3.3 Numerical Case-Study: Do Practical Settings empirically allow for Computational Savings?

As an experimental case-study, to better understand when we may or may not obtain computational time improvements in practice[46] we consider the particular task of CIFAR10 image classification. We put ourselves in a situation where we consider increasing d_M. This will help us empirically reveal which region the K-factor falls into. If we see that the required number of modes for our target accuracy of $\epsilon = 0.033$ is fixed as d_M increases, we conclude our K-factor practically falls into region III (for this value of ϵ).

To investigate under d_M increase, we pick an off the shelf neural network (refer to it as *"base-net"*) for the task, and consider scaling it upwards multiple times as follows. For each value (referred to as *"network scale-factors"*) in the list [1, 1.33, 1.67, 2, 2.33, 2.67, 3, 3.33, 3.67], we produce a new neural network by: (1) taking the *base-net* and (2) for each *base-net* layer multiplying the layer's size hyperparameter (w_l for FC layer and $\eta^{(l)}$ for Conv layer) by the scale-factor. This procedure produces 9 different DNNs we consider, out of which the smallest one is our base-net. VGG11_bn (modified as described in *Section 3.1.3.2*) was our base-net. We train each neural net with

[45] Recall *Corollary A3.9* for this claim.

[46] I.e. investigate if d_M practically falls into region III or not. And if it does not fall into region III, investigate if we can still obtain computation benefits.

K-FAC for 30 epochs with[47] $T_{\text{Cov}} = 20$ and $T_{\text{inv}} = 100$, and tuned the hyperparameters until we reached at least 82% test accuracy (to ensure we resemble typical practical situations). We saved the eigenspetrum of each EA K-factors ($\bar{\mathcal{A}}_k^{(l)}$ and $\bar{\Gamma}_k^{(l)}$) for each layer, and at every iteration when their EVD changed (once in $T_{\text{inv}} = 100$ iterations). Then, for each saved eigenspectrum we computed the minimal number of modes (and fraction of total) to keep while attaining $\epsilon = 0.033$ relative truncation error.

Our numerical results are presented in *Figure 3.2*. Because different K-factors have very different d_M's, our results are easier to present if we plot against the network scale-factor (the factor by which d_M was enlarged from its base-case[48] value). We show both #Modes vs scale-factor, and fraction of modes vs scale-factor, for a more thorough understanding. Each point on the plot is the average of values over an entire run. The #Modes vs scale-factor plot also shows error-bars of the size of an empirical standard deviation to help gauge within-run variability.

For each K-factor which appears to practically require a constant number of modes (based on *Figure 3.2*), we also compute the practically implied N_ϵ (the smallest N_ϵ for which $N_\epsilon r_M$ is above #Modes for all scale-factors and all iterations). We present these results in *Table 3.4*.

Table 3.4: CIFAR10 classification with VGG11_bn (modified as in previous section): Empirically Implied N_ϵ for $\epsilon = 0.033$. Only K-factors which appear to have a constant #Modes required (i.e. be in region III) are considered. See *Figure 3.2* for details.

K-factor	$\bar{\mathcal{A}}^{(1)}$ (Conv)	$\bar{\mathcal{A}}^{(2)}$ (Conv)	$\bar{\mathcal{A}}^{(3)}$ (Conv)	$\bar{\mathcal{A}}^{(4)}$ (Conv)	$\bar{\mathcal{A}}^{(5)}$ (Conv)	$\bar{\mathcal{A}}^{(6)}$ (Conv)
Empirically Implied N_ϵ	3.80e-05	3.96e-04	1.20e-03	1.80e-03	3.41e-03	5.86e-03

K-factor	$\bar{\mathcal{A}}^{(7)}$ (Conv)	$\bar{\mathcal{A}}^{(8)}$ (Conv)	$\bar{\mathcal{A}}^{(9)}$ (FC)	$\bar{\mathcal{A}}^{(10)}$ (FC)	$\bar{\Gamma}^{(7)}$ (Conv)	$\bar{\Gamma}^{(8)}$ (Conv)	$\bar{\Gamma}^{(9)}$ (FC)
Empirically Implied N_ϵ	9.77e-03	2.15e-02	5.47e-02	1.09e-01	1.41	1.17	1.07

Before we begin commenting results, we make two important notes. First, the number of classes is 10 irrespective of the network scale-factor (we cannot increase the number of classes for a fixed dataset). Thus, the $\Gamma_k^{(10)}$ K-factor will always be of size 10, and as such does not participate in our increasing d_M experiment. The curves will also be plotted for $\Gamma_k^{(10)}$ for completeness, but these should be ignored when thinking about increasing d_M values ($\Gamma_k^{(10)}$'s d_M is fixed for all net scale-factors). Secondly, note that if the required #Modes is fixed with d_M, then the required fraction of modes scales like d_M^{-1}. For this reason, our fraction of modes vs net scale-factor plots are on a log-log plot: to better reveal polynomial curves.

[47] Recall the definition of T_{inv} and T_{Cov} from *Section 1.5* (just after *Algorithm 1.1*), if required.
[48] The base-case value of d_M is denoted as $d_M|_{sf=1}$ in the plots.

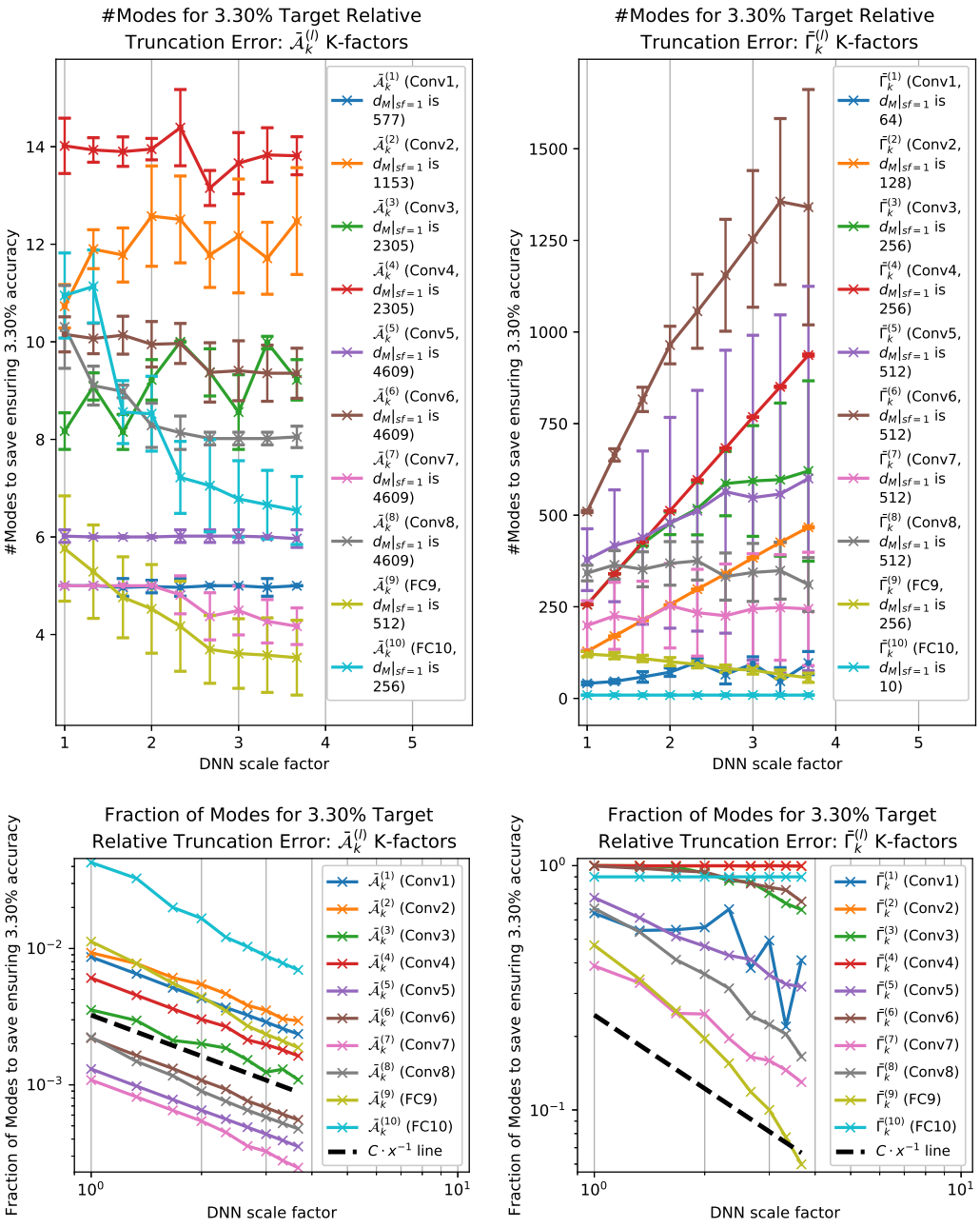


Figure 3.2: #Modes required for 3.3% relative truncation error: Numerical Results for CIFAR10 with VGG11_bn (modified as shown in *Table 3.1*). The #Modes required for 3.3% relative truncation error is obtained as follows: (1) Obtain the *EVD* of each K-factor (eigenmodes sorted descendingly); (2) compute the index of the smallest eigenvalue above $0.033\lambda_{\max}$ for each K-factor and at each iteration when the inverse changes; (3) For each K-factor we then have a distribution of #Modes required, and we can show mean±s.t.d. (as in top-plots) or just the mean (bottom plots).

We make the following observations:

- **Most K-factors practically fall in Region III of our theory.** We see that K-factors $\{\bar{\mathcal{A}}^{(l)}\}_{\forall l}$ and $\{\bar{\Gamma}^{(l)}\}_{l\in\{7,8,9\}}$ require a constant[49] number of modes as d_M (the net scale-factor) increases. This behavior is consistent with our K-factors falling into region III. When we look at the practically implied values of N_ϵ in *Table 3.4*, we see that most of these are significantly below 1 (and thus much smaller than the theoretically implied value[50] of 68). These findings together suggest that our theory captures the qualitative behavior[51] well, but overestimates the number of modes we need to keep. Of course, this is just empirical evidence and not a hard proof.

- **Out of the K-factors not practically falling into region III, we can still obtain some complexity improvement for most of them.** The fraction plots in *Figure 3.2* show that $\Gamma_k^{(1)}$, $\Gamma_k^{(3)}$, $\Gamma_k^{(5)}$, $\Gamma_k^{(6)}$ all have a decreasing fraction of modes to keep, though at a smaller rate than d_M^{-1} (since they have an increasing #Modes with d_M). Thus, for these K-factors we can reduce the inversion time scaling by a factor[52] of d_M^a with $a \in (0, 1)$ (when targeting $\epsilon = 0.033$ relative truncation error).

- **Only 2 K-factors in 19 require a constant fraction of modes equal to 1**. For the investigated scale-factor window, only $\Gamma_k^{(2)}$ and $\Gamma_k^{(4)}$ require a constant fraction of modes equal to 1 (no savings possible for all considered net scale-factors and $\epsilon = 0.033$).

- **Only 4 in 19 K-factors give no computation saving for $\epsilon = 0.033$ at base-net.** Excluding $\Gamma^{(10)}$ which is of trivial size 10, only four K-factors ($\{\Gamma_k^{(l)}\}_{l\in\{2,3,4,6\}}$) require to save all the modes at scale-factor value of 1 (when using the base-net).

- **Most K-factors require saving a tiny fraction of modes.** The values for fraction of modes we need to keep are low and very low ($5 \cdot 10^{-1}$ to 10^{-4}) for most K-factors even at net scale-factor 1. This suggests very large computational savings can be attained in practice, irrespective of how this saving scales with d_M.

[49] Or surprisingly, even decreasing number of modes.

[50] Review *Example 3.8* or *Section 3.1.3.2*.

[51] For a more critical review of how our theory reconciles with practical findings, as well as future directions for our theory, see *Appendix A3.5*.

[52] Less that a power but still more savings as d_M increases.

- **Generally, K-factors which fall deeper into region I according to *Table 3.1* required a larger fraction of modes to be kept.** The K-factors $\{\bar{\Gamma}_k^{(l)}\}_{l\in\{1,2,3,4\}}$ were most deeply into region I according to the theoretical approach *Table 3.1*. Out of these, all with the exception of $\bar{\Gamma}^{(1)}$ required a very large fraction of modes (in particular all modes at scale-factor 1). There are some exceptions: (1) $\bar{\Gamma}^{(1)}$ required a slightly smaller fraction of modes, about the same as $\bar{\Gamma}^{(5)}$ and $\bar{\Gamma}^{(8)}$ which were in region II according to *Table 3.1*. Also, $\bar{\Gamma}^{(6)}$ which was in region II according to *Table 3.1* required all modes to be kept at scale-factor 1. However, there is generally a trend. This finding again suggests our theory has the correct qualitative behavior but tends to overestimate N_ϵ.

On a closer inspection of the implied N_ϵ values in *Table 3.4*, we see that these fluctuate by orders of magnitude, but our theory can only describe cases when N_ϵ is constant. While this appears to contradict the theory, it does not. Instead, it shows that our bounds are relatively loose in practical settings: there would only be a contradiction if we had implied N_ϵ above the theoretically implied value (68 for *Section 3.1.2* results). For a discussion about how we may improve this in the future to address the looseness aspect observed here see *Appending A3.5.1.*
We have two main conclusions to draw based on our numerical case-study here:

1. **We can significantly improve K-FAC's scalability:** Our theory together with our numerical case-study here suggests[53] that we can improve the complexity of obtaining K-factor inverse representations (ignoring the $\log(d_M)$ multiplicative factor) by a power of d_M for most K-factors, and by (at least) a multiplicative factor for some others (while controlling relative truncation error).

2. **For practical DNN training situations, we can reduce K-FAC's wall time significantly:** Our numerical evidence suggests[54] that while controlling the relative truncation error at $\epsilon = 0.033$, we should be able to make substantial computational savings for practical DNN training situations. This is ultimately our main practical interest, and is more important than the attainable algorithm complexity (which is an asymptotic concept).

[53]This was true for our particular case-study. Of course, we only have some numerical evidence and not a hard proof. Our theory is sufficient to say that we can reduce the complexity by a power of d_M. However we use numerical evidence to argue that we are in the desired asymptotic regime in practice.

[54]This was true for our particular case-study. Of course, we only have some numerical evidence and not a hard proof. However the ultimate test of whether this holds or not will be the optimization performance of our proposed algorithms relative to standard K-FAC.

In *Appendix A3.4*, we have plots equivalent to the ones in *Figure 3.2* for different values of ϵ: based on these, we can draw the same qualitative conclusions for different ϵ values. We used $\epsilon = 0.033$ here as this is what was the practical value we used later in numerical experiments.

Our theoretical conclusions and numerical investigation give a strong indication that devising a K-FAC inversion scheme which only seeks to directly obtain (an approximation for) the leading modes can substantially reduce computational burden while only introducing a controlled low-rank truncation error (whose value we can adjust and ensure). In *Section 3.2* we start looking at how to practically achieve this complexity reduction, and propose practical algorithms (K-FAC improvements).

Aside: A possible Attempt in choosing the Target Rank (#Modes to keep in Practice). As a revealing aside, note that our discussion in this subsection may suggest a way to choose the target rank. In particular, the final bullet-point observation above (*page 78*) suggests that in general, it might be a good idea to keep a relatively larger fraction of modes when the K-factor falls deeper into region I. An example of how this could be done concretely is the following: (1) use n_M/d_M as a measure for how deeply the K-factor falls into region I (or a strictly increasing function thereof); (2) gather data about fraction of modes kept vs. n_M/d_M on different {data-sets, DNNs} pairs (for practically useful values of ϵ); then (3) fit a regression curve using n_M/d_M (and possibly other data-set and DNN characteristics as regression features) as predictors for fraction of modes to keep. However, we do not recommend using this approach for the following reasons: (a) though the cost happens before actual optimization run-time (we could gather the data beforehand and keep it as "library knowledge" to be accessed at runtime), the cost of gathering this data is relatively high; (b) The method would be noisy, and not fine-tuned to the case at hand.

For this reason, a much better automatic (and adaptive) approach is dynamically adjusting the target-rank at run-time based on a measurable error proxy, which is what we propose in *Section 5.3.2*. The approach therein (or improvements thereof) has a negligible cost, and the target-rank is adjusted on the fly (can account for non-stationarity of this target-rank), and is designed to works specifically for the problem at hand. Perhaps the best use of the attempt suggested here would be as an initialization for the adaptive scheme in *Section 5.3.2*. However, we never use this in our experiments, due to its sheer pre-runtime cost, but leave it as a possible avenue for the interested practitioner, and for future work.

3.1.4 The effects of Directly Obtaining the Target Modes

So far, we ignored the fact that obtaining the leading modes directly (rather than obtaining the whole EVD, and the truncating) introduces a further error. In this short section, we discuss important aspects relating to this issue.

Directly obtaining the Leading Modes introduces a further Error. Our discussion so far was about how many modes (r) we should keep, s.t. our optimal truncation error is controlled by ϵ. In practice, obtaining this optimal truncation error is only[55] possible if we perform the EVD to compute all the modes[56], and then only retain the leading ones. However, doing this does not bring any computational savings when obtaining the inverse representation. Thus, we do not want to perform an EVD. In contrast, we want to use an algorithm which can directly obtain the desired leading modes - thus obtaining computational savings. We shall see that we do this with rNLA algorithms. However, when using rNLA we introduce a further error. We call this further error the *projection error*, and refer to the optimal truncation error for any given r as simply the truncation error[57] (review *Section 2.2*). The excess error to the truncation error will be the projection error, attributed to having to project our matrix down to a subspace (which only approximately contains the leading modes) when obtaining our desired computational savings.

Choosing ϵ in practice: accounting for projection error. We shall see in *Section 3.2.4* that if we choose r s.t. our optimal truncation error is controlled below ϵ, the total representation error when using RSVD with the same target-rank r is theoretically bounded (in expectation) below 2.88ϵ in practical settings (if our rNLA parameters are chosen well). To attain this, we need to choose the number of power iterations to scale as $n_{\text{pwr-it}} = \mathcal{O}(\log(d_M))$ with increasing d_M, as we discussed in *Section 2.2.3*. Values around $n_{\text{pwr-it}} \approx 4$ are enough for investigated standard practical

[55] Of course, there are methods to directly obtain the leading modes and be within a "small error" of the true leading modes representation (for eg. see the one we discuss next). Furthermore, even computing the full *EVD* and then truncating has an error relative to the true leading-modes representation, because the computation is done numerically. In certain cases, it may well be that directly obtaining the leading modes has an error of the same order as the numerical error of the full EVD, in which case our statement that "obtaining the optimal truncation error is only possible with a full EVD" is somewhat misleading (as we can get the same order of error in representing the leading modes through direct-computation methods). However, this edge-case is extremely unlikely and also very favourable for us in practice, so is not a cause for any concern - but an important statement qualification to highlight. Our discussion was made with the most typical case in mind, which is also the most unfavorable one.

[56] Note that in our case the eigendecomposition and the singular value decomposition are the same.

[57] This is because we think about the optimal truncation error as being the only true truncation error. Anything in excess of that can be attributed to performing approximations.

settings. This choice of $n_{\text{pwr-it}} = \mathcal{O}(\log(d_M))$, imposed by having controlled projection error, results in our true complexity being $\mathcal{O}(\log(d_M)d_M^2)$ (denoted as $\tilde{\mathcal{O}}(d_M^2)$) rather than the hoped-for complexity of $\mathcal{O}(d_M^2)$. The $\log d_M$ term factor is however a minor overhead, as we shall see in *Section 3.2.4*. Regarding how concentrated the measure is around the expectation, numerical experiments for a particular case-study show that even the worst outlier typically fits within 3.2ϵ (see *Appendix A3.6*). Considering that the EA K-factors get regularized by adding $\lambda_{\text{reg}}I$, having a total error of our low-rank representation of the same order should not change the K-FAC step by much. In practice we observe that setting $\lambda_{\text{reg}} = 0.1\lambda_{\max}(\bar{\mathcal{M}}_k)$ works best. Thus, we can choose to maintain our relative truncation error below $\epsilon = 0.033$. This gives a total relative error of about 0.0952 (total error of $(0.0952)\lambda_{\max}(\bar{\mathcal{M}}_k)$) when obtaining the leading modes directly with RSVD.

For this reason, most of our case-studies and discussions give examples where the relative truncation error is controlled below $\epsilon = 0.033$. Again, this means we aim to save a number of modes r, which would give us a relative error below $\epsilon = 0.033$ when truncating the SVD down to the leading r modes. However, when attempting to directly approximate these modes, the relative error goes up, since we have to find the subspace of these leading modes to get them directly in the first place (review *Section 2.2*). The extra error occurs because our subspace finding algorithm is not perfect, though the projection error can be brought arbitrarily close to being of the same order as the truncation error by expending more computation (see *Section 2.2*).

3.2 Proposed Optimization Algorithms

Our theory of *Section 3.1* suggested that for sufficiently large K-factor size (d_M), we could save only a constant number of (leading) modes (w.r.t. d_M) while controlling the relative truncation error (ϵ). In the particular numerical case-study of *Section 3.1.3.3*, we saw that the number of modes we need to save for practical net architectures and ϵ values are much smaller than the K-factor size (and also much smaller than implied by theory), for most K-factors.

To reap the benefits of such a situation (should it exist), we need to be able to directly compute a desired number (r) of leading modes. We do this using randomized numerical algebra (rNLA). Using rNLA, we can compute the leading r modes in $\mathcal{O}((r + r_o)d_M^2)$ (for some $r_o \ll r$). This direct computation introduces a further error (the projection error), as previously discussed. We shall see in *Section 3.2.4* how to control this projection error. Notably, controlling this error requires $\mathcal{O}(\log(d_M))$

power iterations, which means that ensuring the total error is controlled takes $\mathcal{O}\big((r + r_o)\log(d_M)d_M^2\big)$ complexity (worsening by a multiplicative $\log(d_M)$ factor as compared to the case when we do not attempt to control the projection error).

We now present two approaches (in *Sections 3.2.1* and *3.2.2*) for speeding up K-FAC, which replace the typically used EVD of the K-factors with rNLA algorithms. These approaches obtain approximations to a low-rank truncation of the eigenvalue decomposition, in an attempt to reap the benefits of the favourable eigenspectrum situations outlined in *Section 3.1.* The ideas are similar in spirit and presented in the order of increasing computational saving (and reducing accuracy).

3.2.1 RSVD K-FAC (R-KFAC)

Instead of computing the eigendecompositions of the EA-matrices (K-Factors) $\bar{\mathcal{A}}$ and $\bar{\Gamma}$ (in *line 12* of *Algorithm 1.1*; of time complexity $\mathcal{O}(d_{\mathcal{A}}^3)$ and $\mathcal{O}(d_{\Gamma}^3)$), we could settle for using a rank-r RSVD approximation (dropping the l's and k's for simplicity):

$$\bar{\mathcal{A}} \overset{\text{RSVD}}{\approx} \tilde{U}_A \tilde{D}_A \tilde{U}_A^T, \quad \text{and} \quad \bar{\Gamma} \overset{\text{RSVD}}{\approx} \tilde{U}_\Gamma \tilde{D}_\Gamma \tilde{U}_\Gamma^T, \tag{3.54}$$

where $\tilde{U}_A \in \mathbb{R}^{d_{\mathcal{A}} \times r}$, $\tilde{U}_\Gamma \in \mathbb{R}^{d_\Gamma \times r}$, and $\tilde{D}_A, \tilde{D}_\Gamma \in \mathbb{R}^{r \times r}$.

Using this trick, we reduce the computation cost of *line 12* in *Algorithm 1.1* from $\mathcal{O}(d_{\mathcal{A}}^3 + d_\Gamma^3)$ to $\mathcal{O}((d_{\mathcal{A}}^2 + d_\Gamma^2) n_{\text{pwr-it}}(r + r_o))$ when using an oversampling parameter r_o and $n_{\text{pwr-it}}$ power iterations (recall *Algorithm 2.1*). This is a dramatic reduction since we can choose $(r + r_o) \ll \min(d_{\mathcal{A}}, d_\Gamma)$ in most practical cases, while maintaining minimal truncation error, as we have seen in *Section 3.1.3.* The RSVD hyper-parameter $n_{\text{pwr-it}}$ can be chosen in two ways: (1) either control the projection error (and thus the total error), in which case we have to set $n_{\text{pwr-it}} = \mathcal{O}(\log(d_M))$, worsening the complexity slightly[58] (see *Section 3.2.4*); or (2) to minimize time-complexity (cost) and losing any error guarantee, by choosing $n_{\text{PWR-IT}}$ to be constant w.r.t. d_M.

Once we have the approximate low-rank truncations, we estimate

$$(\bar{\Gamma} + \lambda I)^{-1} M \approx (\tilde{U}_{\Gamma,r} \tilde{D}_{\Gamma,r} \tilde{U}_{\Gamma,r}^T + \lambda I)^{-1} M, \tag{3.55}$$

where λ is the regularization parameter (applied to K-factors), and then compute

$$(\tilde{U}_{\Gamma,r} \tilde{D}_{\Gamma,r} \tilde{U}_{\Gamma,r}^T + \lambda I)^{-1} M = \tilde{U}_{\Gamma,r} \left[(\tilde{D}_{\Gamma,r} + \lambda I)^{-1} - \frac{1}{\lambda} I \right] \tilde{U}_{\Gamma,r}^T M + \frac{1}{\lambda} M. \tag{3.56}$$

[58] In the practical scenarios we investigated, the numerical values of $n_{\text{pwr-it}}$ required to control the projection error is very low (see *Sections 3.2.4 and A3.6.1*) and thus the slight increase in complexity to change the error is almost entirely a theoretical concern, rather than a practical one.

We use the r.h.s. of (3.56) to apply the Γ K-factor inverse: i.e. to compute $(\tilde{U}_{\Gamma,r}\tilde{D}_{\Gamma,r}\tilde{U}_{\Gamma,r}^T + \lambda I)^{-1}M$. Performing the naive computation (using the l.h.s. of (3.56)) is of cubic complexity in layer size: $\mathcal{O}(d_\Gamma^2 d_\mathcal{A})$, but computing the r.h.s. of (3.56) has complexity $\mathcal{O}(rd_\Gamma d_\mathcal{A})$. We take a perfectly analogous approach for $V(\bar{\mathcal{A}}+\lambda I)^{-1}$. Thus, we see that our low-rank inverse representation also allows us to have an inverse application complexity of $\mathcal{O}(rd_\Gamma d_\mathcal{A})$, which is better than the one of standard[59] K-FAC with complexity $\mathcal{O}([d_\Gamma^2 + d_\mathcal{A}^2]d_\Gamma d_\mathcal{A})$ (sees *lines 14 - 15* of *Algorithm 1.1*).

The R-KFAC algorithm is obtained by replacing *lines 10 - 15* in *Algorithm 1.1* with the for loop shown in *Algorithm 3.1*. Note that in practice we may have different values for the target rank r for each K-factor, thus the occurrence of super-and-sub-scripted r versions: $r_\mathcal{A}^{(l)}$, $r_\Gamma^{(l)}$. Over-all, R-KFAC's time cost scales like $\mathcal{O}(n_{\text{pwr-it}}d_M^2(r+r_o))$ (when setting $d_M = d_\mathcal{A}^{(l)} = d_\Gamma^{(l)}$ and $r = r_\mathcal{A}^{(l)} = r_\Gamma^{(l)}$, $\forall l$, for simplicity of exposition).

Note that the RSVD subroutine in *lines 4 - 6* of *Algorithm 3.1* may be executed using the RSVD in *Algorithm 2.1* (of *Section 2.2.1*). Using different RSVD implementations would not significantly change our discussion. As discussed in *Section 2.2.1*, even though $\tilde{U}_{A,k}^{(l)}$ should equal $\tilde{V}_{A,k}^{(l)}$ since $\bar{\mathcal{A}}_k^{(l)}$ is square SPSD, the RSVD algorithm returns two different matrices, of which the more accurate one is the "V-matrix" (which is the one we use; see *Algorithm 3.1*). The same observation applies to Γ-related quantities.

Algorithm 3.1: R-KFAC (our first proposed algorithm)

Replace *lines 10 - 15* in *Algorithm 1.1* with:

Choose $\{r_\mathcal{A}^{(l)}\}_{l=1}^{N_L}$, $\{r_\Gamma^{(l)}\}_{l=1}^{N_L}$ hyperparameters according to some heuristic[a];

for $l = 0, 1, ..., N_L$ **do**

`// Get RSVD of` $\bar{\mathcal{A}}_k^{(l)}$ `and` $\bar{\Gamma}_k^{(l)}$ `for inverse application`

$\tilde{U}_{A,k}^{(l)}\tilde{D}_{A,k}^{(l)}(\tilde{V}_{A,k}^{(l)})^T = \text{RSVD}(\bar{\mathcal{A}}_k^{(l)}, n_{\text{pwr-it}}, \text{target_rank} = r_\mathcal{A}^{(l)}, \text{oversample} = r_o)$

$\tilde{U}_{\Gamma,k}^{(l)}\tilde{D}_{\Gamma,k}^{(l)}(\tilde{V}_{\Gamma,k}^{(l)})^T = \text{RSVD}(\bar{\Gamma}_k^{(l)}, n_{\text{pwr-it}}, \text{target_rank} = r_\Gamma^{(l)}, \text{oversample} = r_o)$

`// Use RSVD factors to approximately apply K-factors inverses`

$J_k^{(l)} = \text{Mat}(\tilde{g}_k^{(l)})$

$M_k^{(l)} = J_k^{(l)}\tilde{V}_{A,k}^{(l)}\big[(\tilde{D}_{A,k}^{(l)} + \lambda I)^{-1} - \frac{1}{\lambda}I\big](\tilde{V}_{A,k}^{(l)})^T + \frac{1}{\lambda}J_k^{(l)}$

$S_k^{(l)} = \tilde{V}_{\Gamma,k}^{(l)}\Big[(\tilde{D}_{\Gamma,k}^{(l)} + \lambda I)^{-1} - \frac{1}{\lambda}I\Big](\tilde{V}_{\Gamma,k}^{(l)})^T M_k^{(l)} + \frac{1}{\lambda}M_k^{(l)}$

$s_k^{(l)} = \text{vec}(S_k^{(l)})$

[a] See *Section 3.2.3* for details.

Definition of the R-Update. To facilitate our discussion in the next chapters, we will refer to performing *lines 4 - 6* of *Algorithm 3.1* as the *"R-update"*.

[59] Though notably, the original K-FAC paper proposes a (different) way to apply the inverse in quadratic time (see *Section 8* in [1]). This is however not always used in practice.

3.2.2 SREVD K-FAC (SRE-KFAC)

Instead of using RSVD in *line 4* of *Algorithm 3.1*, we can exploit the symmetry and use SREVD (e.g. with *Algorithm 2.2* in *Section 2.2.2*). This would reduce the computation cost of that line by a constant factor, although the computation complexity would be the same: $\mathcal{O}\big((d_{\mathcal{A}}^2 + d_{\mathcal{A}}d_{\Gamma} + d_{\Gamma}^2)n_{\text{pwr-it}}(r + r_o)\big)$. However, this cost reduction comes at the expense of reduced accuracy, because SREVD has significant *projection error* (unlike RSVD; recall *Section 2.2*).

We refer to this algorithm as SRE-KFAC and briefly present it in *Algorithm 3.2*. Note that in *line 5* of *Algorithm 3.2* we are assigning $\tilde{V} \leftarrow \tilde{U}$ to avoid rewriting *lines 9 - 10* of *Algorithm 3.1* with $\tilde{V}$'s replaced by $\tilde{U}'s$. Over-all SRE-KFAC's time cost scales like $\mathcal{O}(d_M^2 n_{\text{pwr-it}}(r + r_o))$ (when setting $d_M = d_{\mathcal{A}}^{(l)} = d_{\Gamma}^{(l)}$ and $r = r_{\mathcal{A}}^{(l)} = r_{\Gamma}^{(l)}$, $\forall l$, for simplicity of exposition).

Algorithm 3.2: SRE-KFAC (our second proposed algorithm)

1 Replace lines *lines 5 - 6 in Algorithm 3.1* with:
2 `// Get SREVD of` $\bar{\mathcal{A}}_k^{(l)}$ `and` $\bar{\Gamma}_k^{(l)}$ `for inverse application`
3 $\tilde{U}_{A,k}^{(l)}\tilde{D}_{A,k}^{(l)}(\tilde{U}_{A,k}^{(l)})^T = \text{SREVD}(\bar{\mathcal{A}}_k^{(l)}, n_{\text{pwr-it}}, \text{target_rank} = r_{\Gamma}^{(l)}, \text{oversample} = r_o)$
4 $\tilde{U}_{\Gamma,k}^{(l)}\tilde{D}_{\Gamma,k}^{(l)}(\tilde{U}_{\Gamma,k}^{(l)})^T = \text{SREVD}(\bar{\Gamma}_k^{(l)}, n_{\text{pwr-it}}, \text{target_rank} = r_{\Gamma}^{(l)}, \text{oversample} = r_o)$
5 $\tilde{V}_{A,k}^{(l)} = \tilde{U}_{A,k}^{(l)}$; $\tilde{V}_{\Gamma,k}^{(l)} = \tilde{U}_{\Gamma,k}^{(l)}$

3.2.3 How do we Choose the Number of Modes r?

Having provided the R-KFAC and SRE-KFAC algorithms, we now need to specify how we choose our number of modes r. Based on our discussion in *Section 3.1*, this should not be a surprise, as it directly follows from the discussion therein.

Choosing r based on theory: Not practically useful. In *Section 3.1*, we saw theory that told us we can keep only the leading $r = \min(N_\epsilon r_M, d_M)$ modes (for fixed r_M and N_ϵ with d_M) while ensuring our relative truncation error is below ϵ (for both FC and Conv layers). We had concrete ways of obtaining N_ϵ values from ϵ values (we choose ϵ values), and we saw that r_M differs for different K-factors, but can be computed. As such, we can compute the $N_\epsilon r_M$ values for any given ϵ, and for all K-factors. We even did this for our particular case-studies in *Tables 3.1 - 3.3*. Thus, choosing $r = \min(N_\epsilon r_M, d_M)$ is a theoretically justified and practically feasible way to pick r. In the limit as d_M goes to infinity, this is also a sound way, as we would unlock significant computation savings: the complexity of obtaining our ϵ-error-controlled inverse representation is now $\mathcal{O}\big(d_M^2 n_{\text{pwr-it}}(N_\epsilon r_M + r_o)\big)$ for fixed

$N_\epsilon r_M$. However, we saw that the theoretical values of $N_\epsilon r_M$ are much larger than d_M in our theoretical case-studies of *Section 3.1.3.2*. Thus, using $r = \min(N_\epsilon r_M, d_M)$ for any practical setting would always result in $r = d_M$, which does not yield any computational benefit (reverts back to performing EVD). Nevertheless, the empirical evidence of our numerical case-study of *Section 3.1.3.3* suggested our N_ϵ values are highly overestimated by our theory, and that we can choose relatively small r while controlling the error.

Choosing r based on practical truncation error proxy: targeting ϵ error. *The best* way to choose r in practice is to have a proxy metric for relative truncation error, and each time we recompute the inverse, we adjust r to get closer to the desired value. We do this separately for each K-factor. If our proxy and adjusting strategy are perfect, then we would always have the $r_{\mathcal{A}}^{(l)}$, $r_{\Gamma}^{(l)}$ values which give us a relative truncation error of exactly (or just below) our chosen ϵ. This would reap all the available computational benefits while ensuring our truncation error is controlled as desired for each K-factor. According to our numerical case-study in *Section 3.1.3.3*, there would be large computational savings to obtain (with $r \ll d_M$ for most K-factors when $\epsilon = 0.033$). Of course, our truncation error proxy and adjusting strategy cannot be perfect, and these would complicate things. To avoid these complications for now, we do not use this way of choosing r in this chapter. We will do so later, in *Chapter 5*.

The Approach in this Chapter: Guessing an r. *The simplest* way to choose r is to guess it a priori, before the algorithm starts, based on user knowledge. We take this approach in this chapter, to keep everything as simple as possible, for now. To simplify even further, we choose all the $r_{\mathcal{A}}^{(l)}$ and $r_{\Gamma}^{(l)}$ to be the same, namely we set $r_{\mathcal{A}}^{(l)} = r_{\Gamma}^{(l)} = r$. With this approach, we are effectively treating r as a hyperparameter of the optimizer, instead of treating ϵ as such (as we did with the other two approaches). By doing this, we lose any guarantee on the relative truncation error, in general, but know exactly what our cost[60] will be ahead of time (unlike the proxy-based approach above). However, that is not to say the truncation error is necessarily large. Of course, if we guess $r > N_{\epsilon_1} r_M$ for some $\epsilon_1 < 1$, then our theory says the relative truncation error is controlled below ϵ_1. We might even unknowingly choose r larger than any of the values provided by the adjusting strategy above, and thus control the error even tighter (without knowing).

[60] We can use this approach whenever we prefer controlling cost to controlling error. Ultimately, it is a matter of exploring which approach works best in practice.

Ultimately, we believe this approach is sensible only if we have an indication that we can pick some r to get reasonably low errors for all K-factors, or an indication that controlling computation time gives better performance than controlling error.

Based on our numerical case-study in *Section 3.1.3.3*, we see that that at 3.3% relative error level, most K-factors only needed to keep a number of modes significantly below the batch size, though some needed more than that. Inspired from this case-study, we believe that when constraining ourselves to $r_{\mathcal{A}}^{(l)} = r_{\Gamma}^{(l)} = r$, a good guess for r is a value around n_{B}. We used $r = 0.9n_{\mathrm{B}}$ for epoch number above 15, but slightly less than that for earlier epochs to reap further savings.

3.2.4 rNLA Projection Error: Choice of $n_{\text{pwr-it}}$ and r_o

All our discussion in *Section 3.1* focused on statements about the optimal truncation error. However, we can only obtain that error if we use SVD, then truncate to rank r. But doing this would mean there are no computational benefits to be had when saving the first r modes only. To unlock these computational benefits, we need to instead directly obtain the leading modes. When doing so with rNLA techniques, we introduce a further error on top of this optimal truncation error (which we shall call simply "truncation error" from now): we introduce what we called the projection error (review *Section 2.2*). In short, this arose from the fact that directly obtaining the leading modes (rather than computing all the modes, then truncating) effectively required us to guess (find) the subspace of those leading modes. This leading subspace finding was done by power-iterating over a Gaussian sampled matrix (recall *Section 2.2*). The fact that we do not exactly hit this leading subspace with our subspace finding method is where the projection error comes from.

So far, we saw how many modes we need to keep to control our truncation error. However, because we use rNLA, we still need to control our projection error to ensure a controlled total error. We use RSVD error theory (see [79] and *Section 2.2.3*) to obtain hyperparameter values which ensure we control our total error as desired for R-KFAC. For SRE-KFAC, which relies on a less accurate rNLA algorithm, *we use the same hyperparameters*. Though the SREVD error is not theoretically guaranteed to be controlled in the same way as the RSVD error, a numerical case-study shows the SREVD projection error is still below a reasonable threshold (see *Appendix A3.6.2*). The remaining discussion in this section applies to RSVD and R-KFAC only.

The theory is favorable: it shows our total RSVD representation error can get arbitrarily close to twice the optimal truncation error (by increasing $n_{\textbf{pwr-it}}$). Recall the takeaway message from RSVD error bounds in *Section 2.2.3*: that we can get the projection error arbitrarily close to the truncation error (though in expectation, but the result is meaningful due to measure concentration effects [79]). In particular, for an arbitrary EA K-factor $\bar{\mathcal{M}}_k$ (applying *Section 2.2.3* to a generic K-factor case), our ***expected total error is bounded above*** by

$$\epsilon_T := \left[\left(1 + \sqrt{\frac{r}{r_o - 1}} + \frac{e\sqrt{r + r_o}}{r_o}\sqrt{d_M - r}\right)^{\frac{1}{2n_{\text{pwr-it}}+1}} + 1\right]\sigma_{r+1}(\bar{\mathcal{M}}_k), \tag{3.57}$$

where r_o is the oversampling parameter, r is the number of modes we choose to save, $n_{\text{pwr-it}}$ is the number of power iterations (see *Section 2.2*), and $\sigma_{r+1}(\bar{\mathcal{M}}_k)$ is the $(r+1)^{\text{th}}$ largest singular value of $\bar{\mathcal{M}}_k$ (which is also the truncation error, in the 2-norm). By controlling the factor in front of our truncation error $\sigma_{r+1}(\bar{\mathcal{M}}_k)$, we control the total error. We must choose $n_{\text{pwr-it}}$ and r_o to attain our desired error.

Choosing r_o. In practice, we simply choose $r_o \geq 10$. The role of r_o is to ensure the measure is concentrated enough around the expectation s.t. we get equivalent bounds holding with high probability (rather than expectation bounds). The authors of RSVD ([79]) provide a theoretical discussion in the particular case of $n_{\text{pwr-it}} = 1$, which shows that $r_o > 5$ is sufficient to have measure concentrated enough for practical purposes. While the discussion for the particular case does not trivially generalize, it provides intuition about how the measure concentrates in practice and the kind of values to use for r_o. We choose r_o slightly larger than 5, as the extra cost was negligible in practice.

Choosing $n_{\textbf{pwr-it}}$. Note that the factor by which the total error is larger than the truncation error (in 3.57) grows with the K-factor size d_M (all else fixed). Theoretically, we have seen in *Section 2.2.3* that we have to choose $n_{\text{pwr-it}} = \mathcal{O}\big(\log(d_M)\big)$ to have that factor fixed w.r.t. d_M. Practically, we want to choose an $n_{\text{pwr-it}}$ value for which the factor is small enough for all K-factors sizes involved. Our largest K-factor size typically encountered in practice is $d_M = 4609$ (see the case-studies in *Section 3.1.3.2*, *Tables 3.1 - 3.3*). We know that we will choose $r \leq n_{\text{B}}$, as argued in the previous section. Our batch size will be 256 or lower, as is standard. Using these values, we can see that choosing $n_{\text{pwr-it}} = 4$ gives a total error bounded in expectation below 2.88 times the optimal truncation error. This factor is satisfactory in practice: it is practically close enough to the value of 2, which equation (3.57) tells us is the best attainable value, in the limit as $n_{\text{pwr-it}} \to \infty$ (with all else fixed).

Thus, we see why we used the 2.88 factor in *Section 3.1.4* when arguing that a target rank r for which the (optimal) relative truncation error is below 3.33% gives a total expected[61] representation error no greater than 9.52%, when directly obtaining these r modes with RSVD (hyperparameters $r_o = 10$, $n_{\text{pwr-it}} = 4$).

Note that choosing $n_{\text{pwr-it}}$ values above 0 increases the cost of obtaining our inverse representations slightly. In particular, choosing $n_{\text{pwr-it}} = \mathcal{O}(\log(d_M))$ to control the projection error (and thus the total error) increases the total time-complexity by a multiplicative $\log(d_M)$ factor. That is, our complexity in obtaining the K-factors inverse representations is in fact $\mathcal{O}\big(d_M^2 \log(d_M)\big)$ rather than $\mathcal{O}(d_M^2)$. Note that if a practitioner is not interested in controlling the error, they can fix $n_{\text{pwr-it}}$ w.r.t. d_M and get a $\mathcal{O}(d_M^2)$ complexity of R-KFAC or SRE-KFAC. However, the $\log(d_M)$ factor is practically negligible - both from an asymptotic theory (complexity) point of view[62], as well as because it translates to very low values in practice (as we have just seen $n_{\text{pwr-it}} = 4$ was enough for the practical case-study we have just considered).

To complement the theory (or the absence of it in the case of SREVD), we numerically investigate the projection error of RSVD and SREVD in *Appendix A3.6*, for a particular case-study (CIFAR10 classif. with VGG11_bn). We observe that the theory of RSVD projection error is pessimistic for our case, and that the RSVD projection error is typically negligible in the case-study results (see *Appendix A3.6.1*). For SREVD, we observe that the error is actually controlled below the values theoretically predicted for RSVD (though we do not have these theoretical guarantees for SREVD). While the SREVD total error is somewhat higher than the RSVD one, we still obtain acceptable values (most of the time the total error is below 3× the optimal truncation error).

Notably, we did not numerically experiment with choosing $n_{\text{pwr-it}}$ and simply stuck with the value of 4 in the basis of our theory-driven discussion here. This statement holds for all numerical experiments in the thesis. We did not investigate different values for $n_{\text{pwr-it}}$ numerically because we believe the theory (see *Section 2.2.3*) was sufficient to pick a good-enough value for our test-cases, and thus that finding "better values" through numerical experimentation could only bring little, if any, improvement. The reasons for this are that: 1) a higher $n_{\text{pwr-it}}$ would significantly increase the cost while not doing much for the error in our case (because the error at $n_{\text{pwr-it}} = 4$ is

[61] Recall we argued the measure is concentrated around the expectation in practice (or see [79]). In *Appendix A3.6* we show a numerical case-study with DNN K-factor matrices where we observe: (1) The measure indeed concentrates tightly around the empirical expectation; (2) The situation is much better in practice: the empirical expectation is much lower than the theoretical bound, and the worst outlier barely exceeds the theoretical expected value bound (3.57).

[62] We may omit explicitly writing this $\log(d_M)$ factor by using $\tilde{\mathcal{O}}(d_M^2) := \mathcal{O}(\log(d_M) d_M^2)$.

already low enough - for our testcases); and 2) since the only lower values we can have than what the theory tells us are $n_{\text{pwr-it}} \in \{1, 2, 3\}$ - there is not much potential benefit left by trying to further reduce some computation through further reducing $n_{\text{pwr-it}}$ (even if the error would not increase substantially). Notably, the situation we encounter here may not hold in other cases. In general, the way we suggest picking $n_{\text{pwr-it}}$ is to first use the theoretical bounds (as we do here) to get a value for $n_{\text{pwr-it}}$ as above (for a level of error[63] we are targeting), and if $n_{\text{pwr-it}}$ is small (say < 5), we can skip numerical experimentation since the potential benefit of that is minimal, else, if $n_{\text{pwr-it}}$ is relatively large (and sufficient resources are available), doing a grid-search[64] for (some/all) values of $n_{\text{pwr-it}}$ less than or equal to what theory suggests could probably yield some benefit. As a consequence of never numerically experimenting with $n_{\text{pwr-it}}$, we cannot know how much different values affect the optimizer performance (this is left as future work).

3.2.5 Further Considerations of Randomized KFACs

Direct Idea Transfer to Other Applications. We can apply the method directly to EK-FAC (a K-FAC improvement; [82]) as well. Being layer-agnostic, our proposed improvement is directly applicable to any layer type K-FAC is implemented for. However, note that our theoretical and numerical error discussions in *Section 3.1* were only for Conv and FC layers. We can also directly apply our improvements to any other K-FAC-based algorithms (for eg.: ACKTR [46] in RL, algorithms where K-FAC is used as a practical implementation "platform" for NGVI in BNNs [47, 48, 49, 83] etc.).

We Closed the Complexity Gap between K-FAC and SENG. It is important to realise that this chapter gives us more than a way of significantly speeding-up K-FAC for large net widths (with controlled error). It tells us that (based on the discussion in *Section 3.1*) the scaling of $\mathcal{O}(d_M^3)$ with layer size is *not* inherent to K-FAC (at least not when $d_M \gg n_{\text{B}}$), and that we can obtain scaling of $\tilde{\mathcal{O}}(d_M^2)$ for K-FAC at practically no accuracy loss (or $\mathcal{O}(d_M^2)$ while losing error control guarantees). This opportunity conceptually arises in a simple way. Roughly speaking, we have much less information available (scales with n_{B}; and we cannot take too large batch-sizes) than would be

[63] Choosing this is a different discussion and has to be picked in relation to the level of regularization that works best / based on available computational budget.

[64] To save some computation we may look at the error-level (and time to compute) in the K-factors rather than the over-all optimizer performance - for multiple values of $n_{\text{pwr-it}}$. Doing this would only require a single R-KFAC run which saves all encountered K-factor (that are then RSVD-decomposed for different $n_{\text{pwr-it}}$ with different random seeds). This approach is substantially cheaper - but may result in poorer $n_{\text{pwr-it}}$ choice than a standard optimizer-performance based grid-search.

required to get high accuracy K-factor estimates, when $d_M \gg n_{\text{B}}$. Thus, whether the *true K-factor* has strong eigenspectrum decay or not does not matter, our *EA K-factors* (estimates) are bound to exhibit it.

So what causes a problem actually solves another: we cannot accurately estimate the K-factors for large d_M given our batch-size limitation - but this puts us in a place where our (proposed) approximate decomposition/inversion computations which scale like $\tilde{\mathcal{O}}(d_M^2)$ are virtually as good as the exact methods which scale like $\mathcal{O}(d_M^3)$. By leveraging this, we close the time-complexity gap (in layer size) between K-FAC and SENG (currently the more successful practical NG implementation, see *Appendix A5.3.2* or the SENG paper [20] for details). To see this, note that we have $\mathcal{O}(d_M^3)$ for K-FAC, and $\tilde{\mathcal{O}}(d_M^2)$ for both *Randomized K-FACs* and SENG. Conceptually, SENG has better scaling than (vanilla) K-FAC because it exploits this lack of information to speed-up computation by removing unnecessary ones. In this chapter, we implicitly showed that we can do the same for K-FAC and obtain the same scaling with d_M.

3.3 Numerical Performance of Proposed Optimizers

We now numerically compare R-KFAC and SRE-KFAC with K-FAC (the baseline we improve upon) and SENG (another NG implementation which typically outperforms K-FAC; see *Appendix A5.3.2* or [20]). We did not test SGD, as this underpeforms SENG for our considered test-case (see *Table 5* in [20]). The accuracy we refer to is always *test*[65] *accuracy*.

A Small Caveat of our Numerical Results. Before we proceed, it is worth noting that a small caveat of our numerical results is only considering to test against the standard K-FAC implementation which uses the EVD (and the associated regularization technique) to get the inverse. While this version is typically preferred in practice (because though slightly more expensive, it was empirically shown to offer better test-set performance [62]), our results would've been more thorough if we also considered the direct inverse computation. Notably, however, both approaches: 1) direct inverse

[65]Some authors argue that test-set performance (error) is not the best metric for benchmarking optimizers, (because there are other factors that affect it in addition to the optimizer), and they advocate for the use of train-set performance (error) as a better metric for comparing optimizers [1, 50]. Despite this, we only focus on test-set metrics, as these are the ones of interest to practitioners - and ultimately what matters when choosing an optimizer (though also looking at the train-set error might give slightly more insight). We omit the train-set results due to space limitation. While using test-set error might influence some conclusions slightly (as opposed to using the train-set), we believe this difference is not substantial. This is because complexity improvement is the main contribution of the thesis: and not any over-fit related improvement.

computation and 2) EVD - are of the same time-complexity, and since our proposed improvements have better complexity (improvements which result in practically better cost-scaling for standard test-cases), the conclusion implied by the numerical results (and claimed practical speed-up factors) would've been almost the same[66]. This caveat applies to all numerical results in the thesis, not just the ones of this chapter (i.e. we never considered the direct inverse computation (standard) K-FAC variant).

3.3.1 Implementation Details

We consider the CIFAR10 dataset with a modified[67] version of batch-normalized VGG16 (VGG16_bn). We used a single *NVIDIA Tesla V100-SXM2-16GB* GPU.

For SENG, we used the implementation from the *official github repo* with the hyper-parameters[68] given by the authors via email for the problem at hand. K-FAC was slightly adapted from *alecwangcq's github* (**Repo:** https://github.com/alecwangcq/KFAC-Pytorch). Our proposed solvers were built on that code.

For K-FAC, R-KFAC and SRE-KFAC we performed manual tuning[69]. We found that no

[66]Notably, the direct inverse computation K-FAC also tends to perform worse on the test-set, and that is also a component that goes into the results - not just the computation time [62].

[67]We add a 512-in 512-out FC layer with dropout ($p = 0.5$) before the final FC layer.

[68]**Repo:** https://github.com/yangorwell/SENG. **Hyperparameters:** *label_smoothing = 0, fim_col_sample_size = 128, lr_scheme = 'exp', lr = 0.05, lr_decay_rate = 6, lr_decay_epoch = 75, damping = 2, weight_decay = 1e-2, momentum = 0.9, curvature_update_freq = 200. Omitted parameters are left as default value.*

[69]Where possible we started from hyperparameters known to have worked well in the literature before ([1, 20]), or if the schedules types were different - from schedules hyperparameters which gave similar schedule. We then tried different values close to the ones we started from, and we based our choices on 1 run (seed not set) per solver (to minimize cost of tuning). Note that the seed was not set to avoid overfitting our hyperparameters to that particular seed. In the case of hyperparameters specific to *Randomized K-FACs* (number of power iterations $n_{\text{pwr-it}}$, oversampling parameter r_o, and target-rank r), we could not start our hyperparameter tuning from any form of "previous values in literature". For $n_{\text{pwr-it}}$ and r_o, values were selected as described in *Section 3.2.4*. For the target-rank, the initial schedule was chosen to be constant, with a value chosen as mentioned in *Section 3.2.3* - we then slightly tuned the schedule around this initial schedule by trial and error.

For the learning rate, we started from a "staircase" schedule which was close to the exponential decay schedule in [20] that was most successful (for K-FAC training CIFAR10 with VGG16_bn), and then adjusted from there by slightly perturbing the schedule and accepting the perturbation if the results were significantly better on one run (no seed set). We then stopped and accepted schedule as final if we rejected all considered perturbations of the given schedule. The set of proposed perturbations contained 1) changes of transition epochs (by ± 1, and by ± 2 units) of the current schedule (i.e. when the learning rate jump happens) - one at a time; and 2) increasing/decreasing the value of a single learning-rate plateau by $2\times$ - $3\times$ - one at a time. The first perturbation to cause a significant positive change in #epochs to target test accuracy was accepted before exhausting all perturbation set for the current schedule. Notably, our (greedy) tuning methodology could be improved into a proper grid-search and this should be done in future work (however, since our proposed improvements make a fundamental cost-saving due to better cost scaling - it is unlikely the results will qualitatively change in any notable aspect purely due to better learning rate schedules).

Table 3.5: CIFAR10 classification with VGG16_bn: results summary of optimizers performance. All solvers reached 91.5% accuracy within the allocated 50 epochs for 10 out of 10 runs. Some solvers did not reach 92% accuracy on all of their 10 runs, and this is shown in the sixth column of the table. Columns 2-4 show the time to get to a specific test accuracy. The fifth column shows time per epoch. All times are in seconds and presented in the form: mean ± standard deviation. For time per epoch, statistics are obtained across 500 samples (50 epochs × 10 runs). For times to a specific accuracy, statistics are obtained based only on the runs where the solver indeed reached the target accuracy (eg. for 92%, the 5 successful runs are used for K-FAC). The last column of the table shows number of epochs to get to 92% accuracy. R and SRE refer to R-KFAC and SRE-KFAC respectively.

	$t_{acc\geq 90\%}$	$t_{acc\geq 91.5\%}$	$t_{acc\geq 92\%}$	t_{epoch}	# Runs hitting 92%	$\mathcal{N}_{\text{acc}\geq 92\%}$
SENG	673.6 ± 34.4	693.2 ± 28.2	718.1 ± 26.0	16.6 ± 0.4	10/10	43.3 ± 0.9
K-FAC	1449 ± 8.7	1971 ± 225	2680 ± 636	75.5 ± 3.4	5/10	35.4 ± 8.3
R	445.8 ± 10.9	600.7 ± 4.9	732.6 ± 153	32.6 ± 0.9	10/10	23.0 ± 4.7
SRE	439.4 ± 28.5	582.2 ± 24.1	785.3 ± 156	30.0 ± 0.4	7/10	26.3 ± 5.1

momentum, weight_decay = 7e-04, $T_{\text{Cov}} = 10$, and $\rho = 0.95$, alongside with the schedules $T_{\text{inv}}(n_{\text{ce}}) = 50 - 20\mathbb{I}_{n_{\text{ce}}\geq 20}$, $\lambda_{\text{reg},k}(n_{\text{ce}}) = 0.1 - 0.05\mathbb{I}_{n_{\text{ce}}\geq 25} - 0.04\mathbb{I}_{n_{\text{ce}}\geq 35}$, $\alpha_k(n_{ce}) = 0.3 - 0.1\mathbb{I}_{n_{ce}\geq 2} - 0.1\mathbb{I}_{n_{ce}\geq 3} - 0.07\mathbb{I}_{n_{ce}\geq 13} - 0.02\mathbb{I}_{n_{ce}\geq 18} - 0.007\mathbb{I}_{n_{ce}\geq 27} - 0.002\mathbb{I}_{n_{ce}\geq 40}$ (where n_{ce} is the number of the current epoch) worked best for all three K-FAC based solvers. The hyperparameters specific to R-KFAC and SRE-KFAC were set to $n_{\text{pwr-it}} = 4$, $r(n_{\text{ce}}) = 220 + 10\mathbb{I}_{n_{\text{ce}}\geq 15}$, $r_o(n_{\text{ce}}) = 10 + \mathbb{I}_{n_{ce}\geq 22} + \mathbb{I}_{n_{\text{ce}}\geq 30}$. We set $n_{\text{B}} = 256$ throughout. We implemented all our K-FAC-based algorithms in the empirical NG paradigm (using y from the given labels when computing the backward K-factors[70]). We performed 10 runs of 50 epochs for each {solver, batch size} pair[71].

3.3.2 Results Discussion

Table 3.5 shows important summary statistics while *Figure 3.3* shows convergence plots. We see that the time per epoch is $\approx 2.4\times$ lower for our solvers than for K-FAC.

This was expected given we reduce time scaling from cubic to quadratic in layer size! In accordance with our discussion in *Section 3.2*, we see that SRE-KFAC is slightly faster per epoch than R-KFAC. Surprisingly, we see that the number of epochs to a target accuracy (at least for 92%) is also smaller for R-KFAC and SRE-KFAC than for K-FAC. This indicates that dropping the low-eigenvalue modes does not seem to hinder optimization progress, but provide a further benefit instead (for our case here).

[70]Rather than drawing $y \sim p(y|h_\theta(x))$; see [64] for details.

[71]**Our codes repo:** https://github.com/ConstantinPuiu/Randomized-KFACs

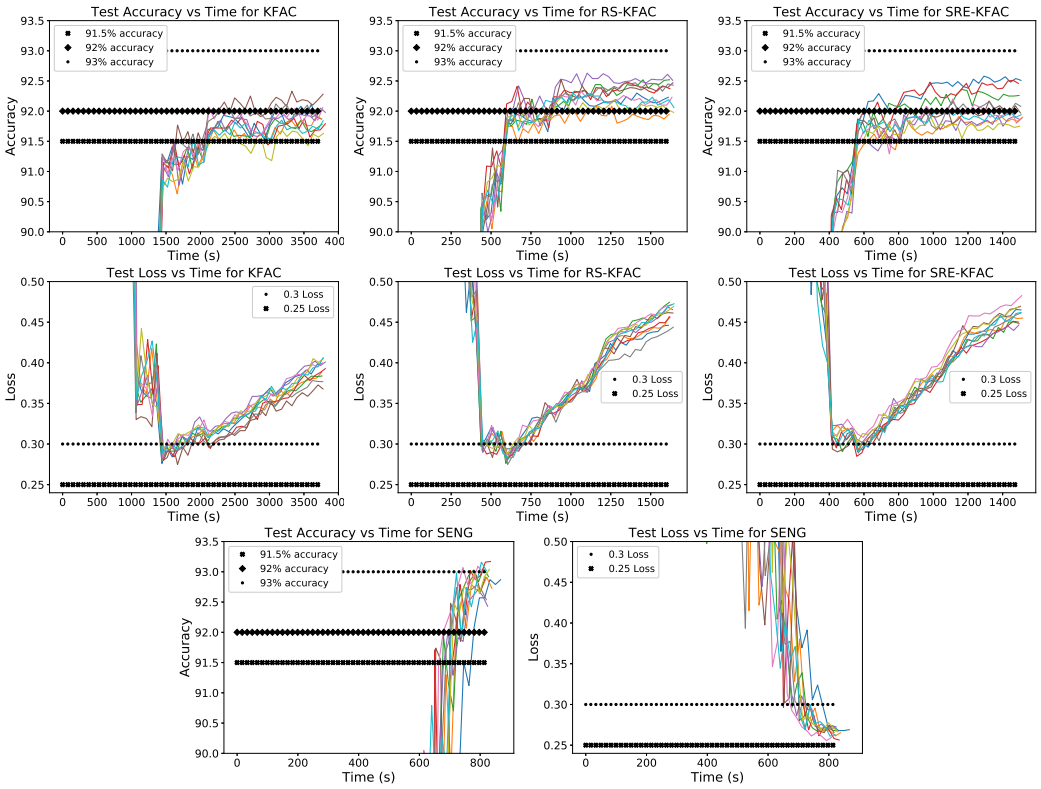


Figure 3.3: CIFAR10 classification with VGG16_bn: Test Loss and Accuracy results. Each plot shows 10 runs for the same [solver, test metric] tuple.

As a result of both much smaller per-epoch time as well as taking fewer epochs to target accuracy, the time to a specific target accuracy is improved by a factor of 3 - 4× when using R-KFAC or SRE-FAC as opposed to K-FAC.

Note that SRE-KFAC takes more epochs to reach a target accuracy than R-KFAC. This is most likely due SRE-KFAC having a larger *projection error* compared to R-KFAC (see *Section 3.2*). For the same reason, R-KFAC always achieves 92% test accuracy while SRE-KFAC only does so 7 out of 10 times. However, SRE-KFAC reaches relatively lower test accuracy targets (90% and 91.5%) in slightly less wall-time than by R-KFAC.

This shows that saving computational effort by allowing for a larger projection error may be beneficial in practice for: (1) relatively low target test accuracy training; or (2) the early stages of the training. Surprisingly, K-FAC reached 92% even fewer times than SRE-KFAC. We believe the fact that some runs of K-FAC based solvers did not reach 92% was due to a tendency to overfit, as can be seen in *Figure 3.3*.

When comparing to SENG, we see that our proposed K-FAC improvements perform slightly better for 90% and 91.5% target test accuracy, but slightly worse for 92%. We believe this problem will vanish if we can fix the overfit of our K-FAC based solvers.

Overall, the numerical results show that our proposed improvements give substantially better implementations of K-FAC, with time-to-accuracy speed-up factors of $\approx 3.3\times$.

3.4 Discussion

We theoretically observed that the eigenspectrum of the K-Factors must decay, owing to the associated EA construction paradigm. We saw that this decay implies that as the layer size increases past a certain threshold (all else fixed), we only need do keep a constant number of modes (w.r.t. layer size) while ensuring our relative (optimal) truncation error is controlled below a desired threshold ϵ. This led to the idea that by directly obtaining the leading modes (with rNLA), we could improve the inversion time-cost scaling by a power of d_M, at least for sufficiently large d_M.

Using rNLA to directly obtain the modes introduced a further error. But we saw that even so, the total error can be bounded as a constant factor of the (optimal) truncation error - and achieving this worsens the complexity only through a multiplicative $\log(d_M)$ factor (getting $\mathcal{O}(\log(d_M)d_M^2)$ overall). Practically, we saw that this constant can be made small (≈ 3) when choosing our rNLA subroutine parameters well, at least when using RSVD. Thus, our ability to control the total error amounts to controlling the (optimal) truncation error. We also saw that we can decide not to control the projection error in order to obtain a (clean) quadratic time-complexity (removing the $\log(d_M)$ term). However, we said that the benefits of controlling the projection error may be far superior to the minor computation savings attained while not attempting to control it (though future investigation is required).

While our theory allowed us to control the truncation error, it only ensured computational savings can be reached in sufficiently large layer size regime. We saw with some theoretical case studies that our theory was never able to guarantee the K-factors fall in this regime for standard practical settings. However, a numerical case-study showed that most K-factors practically fall in this desired regime, and even the few ones that do not still allow for computational savings. This left us with hope that our theory can be improved, and that in practice we can obtain great computational savings while still controlling the error (at least for most K-factors). Ultimately, we pointed out we can enforce a $\mathcal{O}(d_M^2(r+r_o))$ inversion time-scaling for $r + r_o \ll d_M$, though our theory is not strong enough to ensure the error is controlled as desired. Compared to the $\mathcal{O}(d_M^3)$ scaling of EVD, this would bring substantial savings. We have seen that this complexity reduction from $\mathcal{O}(d_M^3)$ to $\tilde{\mathcal{O}}(d_M^2)$ (when controlling the error, or $\mathcal{O}(d_M^2)$ otherwise) closes the gap between K-FAC and SENG.

We discussed theoretically that R-KFAC is more expensive but also more accurate than SRE-KFAC, and the numerical performance of the corresponding optimizers confirmed this. Numerical results for VGG16_bn classification with (slightly modified) CIFAR10 show we speed up K-FAC by a factor of 2.3× in terms of time per epoch, and even had a gain in per-epoch progress. Consequently, target test accuracies were reached about 3.3× faster in terms of wall time. Our proposed K-FAC speed-ups also outperformed the state of art SENG (on a problem where it is much faster than K-FAC; [20]) for 90% and 91.5% target test accuracy in terms of both epochs and wall time. For 92.0% our proposed algorithms only mildly underperformed SENG. We argued it may be possible to resolve this.

3.4.1 Future Work

Future work related to this chapter includes: (1) Obtaining error control theory for SRE-KFAC (of *Section 3.2.2*), by exploring SREVD error theoretically as we did for R-KFAC in *Section 3.2.4*; and (2) Improve the eigenspectrum theory of *Section 3.1* (account for within-M_i eigenspectrum decay which may substantially tighten bounds); (3) Improve numerical results thorough: better hyper-parameter search, considering direct-inverse K-FAC as well.

In the *Future Directions Section of the Conclusions Chapter*, we discuss how future work relating to this chapter integrates with the general future directions of the thesis.

3.4.2 Looking Ahead

By closing the gap between the scaling in DNN layer size of K-FAC ($\mathcal{O}(d_M^2)$ / $\mathcal{O}(\log(d_M)d_M^2)$ with our proposed solution) and the one of SENG ($\mathcal{O}(d_M^2)$) we suggest a possible answer to an unspoken-of conceptual/theoretical mystery: why would SENG scale better than K-FAC when it uses the exact same information to solve the same problem (taking the NG step)? By closing the complexity gap between K-FAC and SENG we implicitly suggest a simple hypothesis: K-FAC does not inherently scale worse, it is just that SENG chooses an approximation route which can more obviously expose (and then cut down on) wasteful computations arising from the absence of information ($n_{\text{B}} \ll d_M$). Finding and removing these wasteful computations in K-FAC makes it scale like SENG (and this is essentially what we have done in this chapter).

But can we do better? Can we make the preconditioning part of K-FAC scale linearly (computing the gradient itself will always be quadratic) in layer size?

In the next chapter (*Chapter 4*), we answer partly affirmative with hopes of ultimately fully reaching this goal. We propose an update (the B-update) for the K-factor inverse representation which only takes linear time in d_M. Notably, however, the update is only applicable to some K-factors[72]. We also propose a way to make the inverse application[73] linear in d_M without introducing any error[74]. Over-all, for all K-factors where the B-update is applicable, the step preconditioning computation becomes linear in layer size. This is a dramatic improvement over standard K-FAC (which is cubic in layer size) and even over Randomized K-FACs (quadratic).

[72]Based on our discussion in *Section 3.1.3.1*, the B-update is only applicable to K-factors satisfying a condition slightly stricter than being in region II of *Figure 3.1*.

[73]Which is applicable whenever the B-update is applicable.

[74]Though we do not implement this efficient inverse application for our numerical experiments: we used the one in *Chapter 3* to obtain the numerical results of *Chapter 4*.

Chapter 4

Brand New K-FACs

Motivated by the goal[1] we set in *Chapter 3* (*Section 3.4.2*), we consider taking the computational efficiency of R-KFAC algorithm[2] (*Algorithm 3.1*) one step further. The key inefficiency of R-KFAC, is that each time we compute an RSVD, we do so "from scratch", not making use of any of the previous RSVDs[3] (low rank representations).

Since we are always interested in the RSVD of a matrix which differs from the previous one (that we have the RSVD of) only through an update, further speed-ups may be obtained here. In particular, it is sometimes the case that this incoming update is low-rank. When K-factors fall in this regime, further computation can be saved by directly amending the available low-rank representation to incorporate the incoming low-rank update (rather than amending the K-factor, and then re-using RSVD). This direct amending is done with *Brand's Algorithm* (presented in *Section 2.3*). Applying Brand's algorithm to adjust available low-rank representations of K-factors will be referred to as the *"B-update"*. Whenever applicable, the B-update, if used exclusively, will reduce the cost of obtaining the inverse to linear time-complexity in layer size[4].

Using Brand's algorithm is exact. However, in order to control computation cost, we will perform a rank-r truncation after each B-update (for some fixed r). The interplay between the B-update and the rank-r truncation irreversibly deletes information[5], thus introducing an error which is in general larger than the optimal one achievable for rank-r representation. Still, the error may be improved as described in *Sections 4.1.2 - 4.1.5*. Notably, the B-update cost would grow if no truncation is performed, ultimately reaching the cost of an EVD - a situation we wish to avoid.

[1] To attain linear width scaling for the preconditioning part of the NGD step (see *Section 3.4.2*).
[2] The discussion in this paragraph also applies to SRE-KFAC.
[3] Note: "RSVD" refers to the low-rank representation obtained with RSVD. Recall *Section 1.6.3*.
[4] Compare to quadratic for our proposed algorithms of *Chapter 3* and cubic for K-FAC.
[5] Unlike re-obtaining a rank-r truncation from the a recomputed SVD of the EA K-factor.

To obtain an over-all linear complexity of the preconditioning part (whenever B-update is usable), we also propose a way to apply the low-rank inverse representations in linear time (in *Section 4.3*). However, note that our numerical results did not have this feature implemented (implementing is future work).

Our **contributions** in this chapter are as follows:

1. **Linear Time Inverse Computation (the B-update to K-factors inverse representation).** Proposing a new, cheaper way of performing the "inverse computation" in K-FAC ([1]), by exploiting the online construction paradigm of exponentially-averaged K-Factors (and the eigenspectrum decay thereof). Notably, the proposed inverse computation can be applied only for some K-factors. W.r.t. the randomized approach of R-KFAC (proposed in *Chapter 3*, or [81]), our proposed update is faster, but provides lower "inversion" accuracy[6]. This computation is linear in layer size (compare to quadratic for *Randomized K-FACs* [81] and cubic for standard K-FAC [1, 29]). See *Section 4.3*.

2. **Linear Time Inverse Application.** Proposing a way to apply the (proposed, low-rank) inverse representation of K-factors onto the gradient in linear time w.r.t. layer size (compare to quadratic for proposed algorithms of *Chapter 3* [81], and cubic for standard K-FAC [1, 29]). This approach leverages the idea to use matrix-multiplication associativity to improve K-factors inverse application cost (first proposed in *Section 8* of [1]), which in our context of low-rank inverse representations yields much better speed-ups (and obviously different final formulae in the code). We only propose this, but the numerical results herein do not use it yet. Implementing it is future work.

3. **Linear Memory-Complexity K-FAC Variant (in layer size):** Notably, the B-update, if used exclusively (as in B-KFAC, see *Section 4.1.1*), is also ***low-memory***: we never need to store the $d_M \times d_M$ EA K-factor (as K-FAC does). Instead, we can only save the $d_M \times (r + n_B)$ low-rank representation, where $r + n_B < d_M$, whenever we can apply the B-update (and do so exclusively). Thus, we propose a K-FAC variant (B-KFAC) with linear memory-complexity[7] in layer size (linear in d_M). Compare to quadratic memory-complexity for K-FAC.

4. **Proposing a correction to low-rank inverse K-factor representations.** We propose a correction to low-rank inverse K-factor representations which is

[6]However, we can trade-off speed to gain accuracy by increasing update frequency, or by even combining with the RSVD update (as we do in *Section 4.1.3*).

[7]Speaking about the preconditioning part of the algorithm only - thus excluding the gradient part (i.e. the "SGD" part), which is always of quadratic memory-complexity.

cheaper than RSVD (but still of quadratic complexity in layer size). We shall call this the *"C-update"*. We propose using this in the context of B-updated K-factors. By rarely using this correction, we can more tightly control the error of a purely B-updated inverse representation. This gives rise to a new algorithm: the BC-KFAC algorithm (see *Section 4.1.4*). Importantly, the correction provably (see *Proposition 4.3* and *Corollary 4.4*) does not worsen the Frobenius-norm error when applied, and intuition tells us it will typically lower the error.

5. **Simple Theoretical motivation for Combining B- and R- updates.** Simple theory to motivate the periodic (but rare) reset of the B-updated K-factors from an RSVD of the EA K-factors (in *Section 4.1.2*). As a result we propose another algorithm (BR-KFAC, see *Section 4.1.3*) which combines the proposed B-update of this chapter with the R-update proposed in *Chapter 3*.

6. **Theoretical comparison of Worst-Case Scenario Error: BR-KFAC vs R-KFAC.** In *Section 4.2.1*, we argue theoretically that the worst-case scenario error of BR-KFAC is better than the one of R-KFAC. This acts as a secondary motivation for BR-KFAC (as an R-KFAC improvement), but is also useful in itself.

7. **Eigenspectrum Continuation.** A simple but effective *"spectrum continuation" technique* which helps better control the truncation[8] error (see *Section 4.1.6*).

8. **4 practical algorithms (*"Brand New K-FACs"*)** using our proposed B-update, possibly combined with R-updates (proposed in *Chapter 3*), and the C-update (the correction) we propose in *Section 4.1.4*. These four algorithms are: **B-KFAC** (in *Section 4.1.1*), **BR-KFAC** (in *Section 4.1.3*), **BC-KFAC** (in *Section 4.1.4*), **BRC-KFAC** (in *Section 4.1.5*).

9. **Numerical results showing that (in a particular setting):**

   (a) our proposed error-controlled B-KFAC versions (BR-KFAC and BC-KFAC) indeed have smaller practical error metrics than B-KFAC (in *Section 4.2.3*).

   (b) BR-KFAC has better practical error metrics than R-KFAC (in *Section 4.2.3*).

   (c) B-KFAC and BC-KFAC outperform R-KFAC (see *Table 4.3* and *Section 4.5*) for all the considered target test accuracies, while the other one (BR-KFAC) does so (only) for high target test accuracy.

   (d) BR-KFAC is better than K-FAC for 3/4 error metrics, while being almost on par for the 4^{th} metric (see *Table 4.1*) and $3\times$ cheaper (see *Table 4.3*).

[8] In the strict sense of targeting only the part of the error arising from truncation.

(e) Our proposed *Brand New K-FACs* outperform the K-FAC ([1]) benchmark for all target test accuracies and outperform SENG for 91% target test accuracy. See *Section 4.5* and *Table 4.3*.

4.1 Proposed Optimizers

In this section, we propose three different algorithms based on the ideas outlined in the introduction (one of which is also a "low memory" algorithm). We also draw some parallels with the algorithms in *Chapter 3* for better understanding.

4.1.1 The B-KFAC Algorithm

The B-update. The idea behind our *B-update* is simple. Instead of performing an RSVD of $\bar{\mathcal{A}}_k^{(l)}$ and $\bar{\Gamma}_k^{(l)}$ at each step (as in R-KFAC), we use Brand's algorithm (see *Section 2.3*) to update the previously held low-rank representation of the K-Factors (the $\tilde{U}$'s and $\tilde{D}'s$) based on the incoming low-rank updates $(1-\rho)A_k^{(l)}(A_k^{(l)})^T$ and $(1-\rho)G_k^{(l)}(G_k^{(l)})^T$. Thus, we *directly apply* Symmetric Brand's algorithm (*Algorithm 2.4* of *Section 2.3*) to estimate a low-rank SVD representation of $\bar{\mathcal{A}}_{k+1}$ by replacing U, D and A in *Algorithm 2.4* by $\tilde{U}_{A,k}^{(l)}$, $\rho\tilde{D}_{A,k}^{(l)}$ and $\sqrt{1-\rho}A_k^{(l)}$ respectively. We also perform an analogous replacement for $\bar{\Gamma}$. Importantly, since we can send the identity initialization in the regularization (as we saw in *Chapter 3*), we have $\bar{\mathcal{A}}_0^{(l)} = 0$ and $\bar{\Gamma}_0^{(l)} = 0$ for all layer indices l. Thus, we start our B-update sequence from our $\tilde{U}_{\cdot,0} = [I_{r\times r}, O_{r\times(d_M-r)}]^T$ and $\tilde{D}_{\cdot,0} = O_{r\times r}$ (at $k=0$). Note that $\tilde{U}_{\cdot,0}$ could be any orthogonal matrix in $\mathbb{R}^{d_M\times r}$ since the singular values ($\tilde{D}_{\cdot,0}$) are all set to zero.

Controlling the size of $\tilde{U}_{(\cdot),k}^{(l)}$'s and $\tilde{D}_{(\cdot),k}^{(l)}$'s: The Truncation[9]. Each application of Brand's algorithm increases the size of carried matrices. To avoid indefinite size increase, we truncate $\tilde{U}_{A,k-1}^{(l)}\tilde{D}_{A,k-1}^{(l)}[\tilde{U}_{A,k-1}^{(l)}]^T$ to rank r once per B-update (and similarly for Γ-related quantities). In other words, we enforce $\tilde{U}_{A,k-1}^{(l)} \in \mathbb{R}^{d_{\mathcal{A}}^{(l)}\times r}$, $\tilde{U}_{\Gamma,k-1}^{(l)} \in \mathbb{R}^{d_{\Gamma}^{(l)}\times r}$, $\tilde{D}_{A,k-1}^{(l)}, \tilde{D}_{\Gamma,k-1}^{(l)} \in \mathbb{R}^{r\times r}$ by retaining only the first r modes (see *lines 2 - 4* of *Algorithm 4.1*). There are two places where we could apply the truncation:

1. Just before the use of Brand's algorithm (after applying the K-factor inverse). This is what we do in this chapter. By doing this, the inverse application is made with a rank $r+n_M$ representation (where n_M is the rank of the incoming matrices, as in *Chapter 3*). This slightly increases the cost of applying the inverse but should give us a more accurate preconditioning.

[9]The truncation works together with the B-update to achieve low, constant computation cost.

2. Just after the use of Brand's algorithm (before applying the K-factor inverse). This saves computation cost because the inverse is now applied with a rank r representation (and the inverse application is linear in this dimension, recall *Chapter 3*). However, the preconditioning is expected to be less accurate with this approach (because at least the best possible obtainable representation is - due to having lower rank). We will use this approach in *Chapter 5*, where we devise distributed versions of our algorithms.

The B-KFAC Algorithm: Putting it all Together. By putting together the B-update and the truncation (with choice 1 above here, in particular), we obtain the B-KFAC algorithm, which is shown in *Algorithm 4.1*. The inverse computation now scales linearly in layer size. In fact, as part of the B-KFAC algorithm, we also propose a more efficient inverse application technique which scales linearly in layer size as well (and can be used whenever the B-update can). However, we shall present that idea separately in *Section 4.3* for a better exposition.

Algorithm 4.1: Brand K-FAC (B-KFAC)

1 **Choice:** Target rank r
2 Replace *lines 5 - 6 in Algorithm 3.1* with:
3 `// Truncate to rank` r`:  maintain matrices sizes`
4 $\tilde{U}^{(l)}_{A,k-1} \leftarrow \tilde{U}^{(l)}_{A,k-1}[:, :r]$, $\tilde{D}^{(l)}_{A,k-1} \leftarrow \tilde{D}^{(l)}_{A,k-1}[:r, :r]$
5 $U^{(l)}_{\Gamma,k-1} \leftarrow \tilde{U}^{(l)}_{\Gamma,k-1}[:, :r]$, $\tilde{D}^{(l)}_{\Gamma,k-1} \leftarrow \tilde{D}^{(l)}_{\Gamma,k-1}[:r, :r]$
6 `// Use Symmetric Brand's low-rank update (`*`Algorithm 2.4`*`)`
7 $\tilde{U}^{(l)}_{A,k}, \tilde{D}^{(l)}_{A,k} = \text{Symmetric_Brand}(\tilde{U}^{(l)}_{A,k-1}, \rho\tilde{D}^{(l)}_{A,k-1}, \sqrt{1-\rho}\, A^{(l)}_k)$
8 $\tilde{U}^{(l)}_{\Gamma,k}, \tilde{D}^{(l)}_{\Gamma,k} = \text{Symmetric_Brand}(\tilde{U}^{(l)}_{\Gamma,k-1}, \rho\tilde{D}^{(l)}_{\Gamma,k-1}, \sqrt{1-\rho}\, G^{(l)}_k)$

Note (T_{Brand} definition). In practice we do *lines 3 - 8* only once in T_{Brand} steps.

Error Comments: Brand's algorithm is exact, but the truncations introduce an error in each of our low-rank K-Factors representations, at each iteration. We will refer to this error as the B-error. We cannot refer to it as the truncation error. The truncation error is still what we defined it: the optimal truncation error for the given rank (the truncation error of keeping the first r modes of the true SVD).

Complexity Comments: Obtaining our K-factors inverse representation now takes[10] $\mathcal{O}(d^{(l)}_\Gamma(r+n^{(l)}_\Gamma)^2 + d^{(l)}_\mathcal{A}(r+n^{(l)}_\mathcal{A})^2)$ time (for layer l) rather than $\mathcal{O}([d^{(l)}_\mathcal{A}]^2(r+r_o) + [d^{(l)}_\Gamma]^2(r+r_o))$ as for our previously proposed *Randomized K-FACs* (in *Chapter 3*), or $\mathcal{O}([d^{(l)}_\mathcal{A}]^3 + [d^{(l)}_\Gamma]^3)$ for vanilla K-FAC [1, 29]. Thus, we can get ***linear complexity***

[10] See *Section 1.5* to recall definitions of $d^{(l)}_\mathcal{A}$, $n^{(l)}_\mathcal{A}$, $d^{(l)}_\Gamma$, and $n^{(l)}_\Gamma$.

in layer size (in $d_\Gamma^{(l)}$ and $d_\mathcal{A}^{(l)}$[11]) ***for obtaining the preconditioner (K-factors) inverese representations***. However there is a caveat to this.

A Caveat for B-update Usability. To apply the B-update, we need our Γ K-factors to satisfy $(r + n_\Gamma^{(l)}) < d_\Gamma^{(l)}$, and $\mathcal{A}$ K-factors to satisfy $(r + n_\mathcal{A}^{(l)}) < d_\mathcal{A}^{(l)}$. Otherwise the actual cost we get becomes worse than both *Randomized K-FACs* and vanilla K-FAC. In practice, this means we can successfully use the B-KFAC approach for most FC layers[12] (as $n_\Gamma^{(l)} = n_\mathcal{A}^{(l)} = n_\text{B} \ll \min(d_\mathcal{A}^{(l)}, d_\Gamma^{(l)})$ holds), and some Conv layers. The case-studies in *Section 3.1.3.2* (*Tables 3.1 - 3.3*) can be used to see this.

Notably, the condition we have is very similar to the condition we had for our K-factors to fall into region II or region III of *Figure 3.1* of *Section 3.1.3.1*. To see this, consider our results in *Chapter 3* about how to chose r to control the truncation error. We saw that[13] we should choose $r \propto n_\Gamma^{(l)}$ for Γ K-factors, and $r \propto n_\mathcal{A}^{(l)}$ for $\mathcal{A}$ K-factors. In that case, we see the condition for us to be able to apply the B-update to an arbitrary K-factor $\bar{\mathcal{M}}_k$ is $(N_\epsilon + 1)n_M < d_M$ (recall notation in *Section 3.1*). This is stricter than the condition $N_\epsilon n_M < d_M$ which allowed us to be in region III of *Figure 3.1* (when we can make computational savings while controlling the optimal relative truncation error below our desired threshold ϵ).

We saw that though the theory predicts N_ϵ is relatively large, in practice the values are most of the time below 1. This means we can typically use the B-update whenever $2n_M < d_M$ and still ensure the (*optimal*) truncation error is controlled. However the B-error, introduced by repeated application of the B-update preceded by truncation, is not guaranteed to be controlled. Thus, in practice we give up on choosing r to have error guarantees, and simply set $r \approx n_M$ inspired from *Chapter 3*. Obtaining error analysis and using that to better our algorithms is future work.

Using the theoretical case-studies in *Tables 3.1 - 3.3* of *Section 3.1.3.1*, we see that our practical condition to have B-update applicable: $2n_M < d_M$, though stricter a condition, typically holds for most K-factors that were in region II in those case-studies.

We only apply B-update to FC layers in this Chapter. For this chapter, we only applied the B-update to FC layers in our numerical experiments. This is because for Conv layers we tend to have $2n_M \approx d_M$ even when $2n_M < d_M$. In this case, we prefer sticking to the higher accuracy of R-KFAC updates, rather than save some computation using the B-update but introducing a relatively[14] large error (see *Section*

[11] We previously generically called these d_M for an arbitrary Kronecker factor.
[12] We have seen in *Section 1.1* how prevalent and important FC layers are.
[13] Theoretically for sufficiently large d's only numerically that it holds for practical d's as well.
[14] Relative to the RSVD projection error.

4.2.3). This choice is purely based on author's intuition[15]. While we numerically gauged both the error difference (in *Section 4.2.3*) and the cost difference (as we just did), it is difficult to say how these differences combine to give different per-time progress. The only useful test being trying different settings. We leave the exploration of where our cut-off threshold (for applying B-update) should be as future work.

B-KFAC as a Stochastic Process: Mathematical Definition. Before we proceed with our reasoning, it is useful to reiterate the EA K-factor mathematical definition and provide the mathematical definition of our B-KFAC representation (process).

In *Chapter 3*, we saw that an arbitrary EA K-Factor $\bar{\mathcal{M}}_k$ (may be either $\bar{A}_k^{(l)}$ or $\bar{\Gamma}_k^{(l)}$ for any l) where we have incoming (random) updates $M_k M_k^T$ with $M_k \in \mathbb{R}^{d_M \times n_M}$ at iteration k (M_k may be either $A_k^{(l)}$ or $G_k^{(l)}$) follows the process[16]

$$\bar{\mathcal{M}}_0 = 0, \;\; \bar{\mathcal{M}}_j = \rho \bar{\mathcal{M}}_{j-1} + (1-\rho) M_j M_j^T \quad \forall j \geq 1, \tag{4.1}$$

and can alternatively be written as

$$\bar{\mathcal{M}}_k = \rho^k \bar{\mathcal{M}}_0 + (1-\rho) \sum_{j=1}^{k} \rho^{k-j} M_j M_j^T = \sum_{i=0}^{k} \kappa(i) \rho^{k-i} M_i M_i^T \quad \forall k \geq 0, \tag{4.2}$$

where $\kappa(i) := 1 - \rho \mathbb{I}_{i>0}$.

The B-KFAC algorithm implicitly estimates $\bar{\mathcal{M}}_k$ as $\tilde{\mathcal{M}}_k^{(B)}$ if we choose to truncate just before applying the B-update, or as $\mathcal{B}_k$ if we choose to truncate just after applying the B-update (recall *Section 4.1.1*). The matrices $\tilde{\mathcal{M}}_k^{(B)}$ and $\mathcal{B}_k$ are given by the repeated applications of the B-update and rank-r truncation, as follows:

$$\begin{aligned} \tilde{\mathcal{M}}_{i+1}^{(B)} &:= \rho \mathcal{B}_i + (1-\rho) M_{i+1} M_{i+1}^T \, \forall i \geq 0, \\ \text{with } \tilde{\mathcal{M}}_0^{(B)} = 0, \quad & U_{\tilde{\mathcal{M}}_i^{(B)}} D_{\tilde{\mathcal{M}}_i^{(B)}} U_{\tilde{\mathcal{M}}_i^{(B)}}^T \overset{\text{SVD}}{=} \tilde{\mathcal{M}}_i^{(B)}, \;\; U_{\tilde{\mathcal{M}}_i^{(B)},r} := U_{\tilde{\mathcal{M}}_i^{(B)}}[:, :r]; \\ \mathcal{B}_i :=& U_{\tilde{\mathcal{M}}_i^{(B)},r} U_{\tilde{\mathcal{M}}_i^{(B)},r}^T \tilde{\mathcal{M}}_i^{(B)} U_{\tilde{\mathcal{M}}_i^{(B)},r} U_{\tilde{\mathcal{M}}_i^{(B)},r}^T \;\; \forall i \geq 0. \end{aligned} \tag{4.3}$$

We refer to (4.3) as *"the B-process"* (or B-KFAC process). Notably, the SVD of $\tilde{M}_i^{(B)}$ is never computed with the (expensive) SVD algorithm, but rather is obtained in linear-time with the cheap and exact B-update. The B-process form presented in (4.3) is a theoretically more convenient mathematical description of the B-process (compared to the practical one in *Algorithm 4.1*).

[15]Though the two underlying effects are investigated, not knowing how these combine to give the quantity of interest makes the choice somewhat anecdotal.

[16]Recall that in practice we initialize our EA K-factor from identity, I. But in *Chapter 3* we saw this is equivalent to having an $\bar{\mathcal{M}}_0 = 0$ initialization with an extra $+\rho^k I$ term in the regularization. We could thus set $\bar{\mathcal{M}}_0 = 0$ and relate our theory to that without modifying the standard (EA) K-FAC algorithm.

4.1.2 Motivation for Resetting B-updated $\tilde{\mathcal{M}}_k^{(B)}$ with an RSVD

We now argue that sometimes performing an RSVD update into a B-KFAC algorithm (for a B-updated K-factor) may be helpful. We shall do so by showing that at least under some distributions of M_j: (1) there are events where the B-KFAC (total) error significantly exceeds the optimal truncation error for a substantial number of consecutive iterations, and (2) the probability of these events occurring, though small, is not negligible. We do so for a very particular 2D case: the reason being that to get the best insight, we need to have explicit formulae for the eigenvalues of the B-KFAC representation $\tilde{\mathcal{M}}_k^{(B)}$, and also for the ones of the EA K-factor $\bar{\mathcal{M}}_k$ as the algorithm evolves. The absence of an explicit formula for the EVD in the general case makes us turn to our 2D example. The (naive) requirement for the usage of the explicit EVD formula may be bypassed to prove similar claims in general multi-D situations. However, this would be much more complex while not adding much understanding. For this reason, we only focus on the 2D case only. This is sufficient to gain some intuition. The intuition gained here acts as a motivation for periodically resetting our B-updated sequence $\{\tilde{\mathcal{M}}_k^{(B)}\}_k$ with an RSVD of the EA K-factor $\bar{\mathcal{M}}_k$.

Example 4.1 gives us a result about extreme error events in a 2D B-KFAC algorithm (with $r = n_M = 1$) for a distribution of M_j where extreme subspaces and large (extreme) eigenvalues are relatively likely (heavy-tailed). The result lower-bounds the probability of having an extreme B-KFAC error event. The extreme error event is characterized by large values of $\text{EEF}_k^{(B)}$ - which is the factor[17] by which B-KFAC error exceeds the optimal truncation error (in the 2-norm). By comparing these extreme factor values with the ones we can expect for RSVD (recall *Section 3.2.3*), we can draw conclusions about how the error of B-KFAC and R-KFAC may compare (at least under certain low, but not negligible, probability events). From here, we can conclude how periodically (but seldomly) using an RSVD in a B-updated K-factor may help.

Example 4.1 (**Particular 2D case: Excess Error of B-KFAC may be large, with small but non-negligible probability**). *Let us consider applying* B-KFAC *with a target-rank of $r = 1$, $d_M = 2$ and $n_M = 1$. We use zero initialization for both the EA K-factor and the B-KFAC representation*[18]*. Let us further assume that the matrices $M_j M_j^T \in \mathbb{R}^{2\times 2}$ are i.i.d. and of the form*

$$M_j M_j^T \overset{\text{SVD}}{=} \psi_j u_j u_j^T, \tag{4.4}$$

[17] $\text{EEF}_k^{(B)}$ stands for **E**xcess **E**rror **F**actor of (B)-KFAC at iteration k.

[18] By sending the Identity initialization into the reg term, as before (see *Chapter 3*)

with $u_j \in \mathbb{R}^{2\times 1}$ *i.i.d. and distributed according to*

$$\mathbb{P}(u_j = u^{(1)}) = p_1, \quad \textit{and} \quad \mathbb{P}(u_j = u^{(2)}) = 1 - p_1 \tag{4.5}$$

for some[19] $p_1 \in (0,1)$ *and with the matrix* $[u^{(1)}, u^{(2)}] \in \mathbb{R}^{2\times 2}$ *being orthonormal. Our* ψ_j*'s are assumed to be i.i.d. and independent of* u_j*, with positive measure for every point on* $\mathbb{R}^+$*.*

Now, let us choose some $c_k \in \mathbb{Z}^+$*. Let us then further choose some* ψ_m*,* $\psi_M > 0$*,* $\psi_{1,m}$*,* $\psi_{1,M} > 0$ *s.t.* $0 < \psi_m < \psi_M < \psi_{1,m} < \psi_{1,M}$ *and* $\psi_M < \rho^{c_K}\psi_{1,m}$*.*

We then have that the ratio between the (2-norm) error of B-KFAC *and the (2-norm) optimal truncation error for* $r = 1$ *is lower-bounded as follows*

$$EEF_k^{(B)} := \frac{\left\|\bar{\mathcal{M}}_k - \mathcal{B}_k\right\|_2}{\min_{v\in\mathbb{R}^{2\times 1}}\left\|\bar{\mathcal{M}}_k - vv^T\right\|_2} \geq \frac{\rho^{-(k-1)} - 1}{1-\rho}\frac{\psi_m}{\psi_{1,M}} \quad \forall k \in \{k^*, k^*+1, ..., c_K\}, \tag{4.6}$$

with probability at least

$$p_k = p_1(1-p_1)^{k-1}\mathbb{P}\big[\psi_j \in (\psi_{1,m}, \psi_{1,M})\big]\left(\mathbb{P}\big[\psi_j \in (\psi_m, \psi_M)\big]\right)^{k-1}, \tag{4.7}$$

where

$$k^* := \left\lceil 1 + \frac{\ln\left(1 + (1-\rho)\frac{\psi_{1,M}}{\psi_m}\right)}{\ln\left(\frac{1}{\rho}\right)}\right\rceil. \tag{4.8}$$

Proof. See *Appendix A4.1.* □

Example 4.1 gives a result which can be used to capture the occurrence of (low probability) events where the B-KFAC error exceeds the optimal truncation error by a large factor. Of course, this is just for a 2D case and a very simple measure on u_j (to avoid the problem of not having an explicit formula for the SVD of $\bar{\mathcal{M}}_k$; our choice allows for an explicit formula and thus detailed investigation).

Depending on the actual distribution of ψ_j, the excess error factor ($\text{EEF}_k^{(B)}$) factor may or may not be large, and the probability of these large excess error occurrences may or may not be significant. We now give specific numerical values for which our B-KFAC error can exceed the optimal truncation error by a factor of more than $3\times$ for more than 10 iterations. In our example, this event has relatively low probability, but still not negligible. The results are formalized in *Example 4.2*, which is a direct number plug-in onto *Example 4.1*.

[19] Think p_1 small to aid understanding but results hold anyway.

Example 4.2 (**Particular Number Plug-in for *Example 4.1*; Heavy tailed distribution of M_j**). *Under the conditions of Example 4.1, let us make the following further choices:* $\rho = 0.95$ *(standard practical value),* $c_K = 25$, $\psi_m = 0.15$, $\psi_M = 0.33$, $\psi_{1,m} = 1.2$, $\psi_{1,M} = 1.5$ *(note that condition* $\psi_M < \rho^{c_K}\psi_{1,m}$ *is satisfied).*

Let us assume that $\ln(\psi_j)$ *are i.i.d. Cauchy shifted to have the mode at* $C_M = 0.24$ *and a scale-factor of* $C_S = 0.02$. *That is,* $\ln(\psi_j)$ *are i.i.d. with a p.d.f. of* $f(\ln(\psi_j); S_M, C_S) = \frac{1}{\pi}\left[\frac{C_S}{(\ln(\psi_j)-C_M)^2+C_S^2}\right]$. *Let us also assume* $p_1 = 0.04$.

By using (4.8), we then have $k^* = 9$. *By using (4.6) and (4.7)* $\forall k \in \{k^*, k^*+1, ...c_K\}$, *we get the* $EEF_k^{(B)}$ *and* p_k *shown in Figure 4.1.*

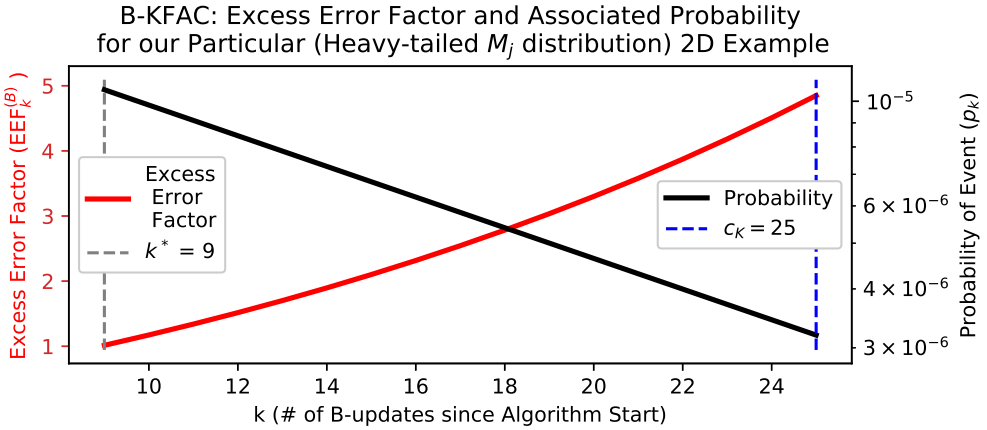


Figure 4.1: B-KFAC: Excess Error Factor and Associated Probability for our Particular (Heavy-tailed M_j distribution) 2D Example.

Proof. Merely plug-in numbers as stated. Proof not given. □

Example 4.2 illustrates[20] that when the extreme draws of $M_j M_j^T$ (in terms of both "rare" subspace[21] choice and large singular value) are relatively likely (we have a heavy-tailed distribution), then the probability of having a very large B-KFAC error (compared to optimal truncation error) is not negligible (though small). Intuitively, this happens because if we draw such an extreme $M_1 M_1^T$, with a very large singular value and a subspace that is not hit too often again, then for many subsequent B-updates, chances are the new, incoming $M_j M_j^T$ will be ignored (by the workings of B-KFAC, since the newly incoming $M_j M_j^T$ at later iterations most likely won't be from the "tail" subspace again, and their singular value will be much below the singular value of the current B-KFAC representation[22]). Thus, for all these iterations of ignored incoming

[20] Though trough a particular example, and is thus not a definite proof.

[21] Note that a $p_1 = 0.04$ is meant to characterize a case where some subspace is "rare" (in this case span$\{u^{(1)}\}$) but the distribution is heavy-tailed.

[22] To see this better, first recall the B-KFAC process definition (4.3) and the B-KFAC algorithm: (*Algorithm 4.1*). Then go through the proof of *Example 4.1*.

information, the B-KFAC error will increase substantially relative to the optimal truncation error. This happens up to some point[23] ($k = c_K$ in our exposition) until the influence of our extreme $M_1 M_1^T$ draw dies out due to sufficient decay eventually getting applied to it through the exponential average paradigm[24]. Since our extreme $M_1 M_1^T$ draw does not have such a low probability under a heavy-tailed distribution, the probability of such an unfortunate event to occur is not negligible[25].

Since, as we saw at least in this example, our B-KFAC error may exceed $3\times$ optimal error for a significant number of iterations (though with relatively low probability lower-bound), we propose an addition which would diminish the negative impact of such events. In particular, since the RSVD total error is guaranteed by the theory to be below $3\times$ the optimal error for good hyperparameter choices (and our numerics suggests the factor is typically $1.5\times$; recall *Section 3.2.3* and see *Appendix A3.6.2*), we propose periodically (but rarely) re-starting our B-update sequence from an RSVD of the EA K-factor. We call this algorithm the BR-KFAC algorithm and look at it in *Section 4.1.3*. The idea is that, should such a negative event occur in a B-update sequence, by resetting the sequence to an RSVD representation with high enough frequency (though agnostic to whether the event has occured or not) we can at least prevent long periods of high errors.

Our arguments here all relied on a very simple case of $d_M = 2$ and $r = n_{\text{B}} = 1$. However, similar examples can be constructed for higher dimensions, and we expect to see similar behavior in those cases. In particular, when $r \ll d_M$ (with a heavy-tailed distribution for singular values of M_j, and uniform measure for the singular direction matrix of M_j) we expect the large error events occurrences to be of high frequency. The reason for this is that under these conditions, the newly incoming r-dimensional information ($M_j M_j^T$) is relatively unlikely to "significantly" hit the r-dimensional subspace of the current B-KFAC representation $\mathcal{B}_{j-1}$, since the whole subspace is of dimension $d_M \gg r$. Thus, after an extreme (large) singular-value M_j is drawn, subsequent draws of incoming information will typically be completely neglected in a B-KFAC algorithm for a long time, similarly to our 2D example. Of course, this argument is all based on extending our intuition gained in the 2D case, and is not a

[23] After this point the B-KFAC error would drop under typical outcomes, as new information would start getting accounted for in our B-KFAC representation.

[24] Note that in the B-update, the previous K-Factor representation also gets multiplied by $\rho = 0.95$, so any influence dies out in time, exponentially in B-KFAC too. Specifically, note that in *lines 7 - 8* of *Algorithm 4.1* we pass "ρD_{k-1}" as the diagonal matrix of eigenvalues at k, where D_{k-1} is the diagonal matrix of eigenvalues of our B-KFAC representation at iteration $k-1$).

[25] One can try constructing similar example for lighter tailed distributions and see the negative event probability is much lower.

rigorous proof. Nevertheless, we believe this is enough to motivate the periodic (but seldom) use of an RSVD in a B-KFAC algorithm (giving the BR-KFAC algorithm).

Notably, we have a lower-bound on errors with "at least" the stated probability. Thus, the error might in practice be slightly worse, and the probability of occurrence higher. Furthermore, note that we concerned ourselves only with the case when the extreme draw happens at the very first step, for simplicity. However, such an extreme draw may also occur later in the process. In this case, the computations would be complicated by having to add the extreme draw to a non-zero previous B-KFAC representation (which is why we focused on the case we did). Still, the fact that such an extreme draw may occur at any later time shows that the probability to have *at least one such negative* B-KFAC *error event* (as the one exemplified in *Example 4.2*) is larger than the computed one when considering a full run of the algorithm. That is, resetting our B-updated sequence with an RSVD one may be more useful in practice than strictly implied by our example (at least in the 2D case).

Motivated by our discussion here, we now look at the BR-KFAC algorithm.

4.1.3 The BR-KFAC Algorithm

Based on our motivation in *Section 4.1.2*, we now provide the BR-KFAC algorithm (*Algorithm 4.2*). BR-KFAC introduces periodic (seldom) RSVD updates to B-updated K-factors. Note that BR-KFAC mostly performs B-updates, so is cheaper than R-KFAC.

Algorithm 4.2: Brand RSVD K-FAC (BR-KFAC)

Choose: over-sampling parameter for RSVD: r_o; Target ranks: $\{r_{\mathcal{A}}^{(L)}\}$, $\{r_{\Gamma}^{(L)}\}$.
Replace *lines 5 - 6 in Algorithm 3.1* with:
if $k \,\%\, T_{BR} == 0$ **then** `// Time to over-write` "$\mathcal{B}_{k-1}$"
  $\tilde{U}_{A,k-1}^{(l)} \tilde{D}_{A,k-1}^{(l)} (\tilde{U}_{A,k-1}^{(l)})^T = \text{RSVD}(\bar{\mathcal{A}}_{k-1}^{(l)}, r_{\mathcal{A}}^{(l)}, r_o)$
  $\tilde{U}_{\Gamma,k-1}^{(l)} \tilde{D}_{\Gamma,k-1}^{(l)} (\tilde{U}_{\Gamma,k-1}^{(l)})^T = \text{RSVD}(\bar{\Gamma}_{k-1}^{(l)}, r_{\Gamma}^{(l)}, r_o)$
else `// Use standard B-KFAC truncation to get` "$\mathcal{B}_k$"
  Truncate as in *lines 3 - 5* of *Algorithm 4.1*
Do *lines 6 - 8* of *Algorithm 4.1* ; `// (Perform B-update)`

A different view on BR-KFAC. In *Section 4.1.2*, we motivated BR-KFAC by saying we introduce seldom RSVD updates to better control B-updated K-factors error (this view allows us to compare BR-KFAC and B-KFAC). In contrast, BR-KFAC can also be viewed as an R-KFAC algorithm to which we introduce further B-updates[26] in between two consecutive RSVD updates (this view allows us to compare BR-KFAC and R-KFAC).

[26]For K-factors for which the B-update can be applied ($r < d_M - n_M$), as discussed under "A Caveat for B-update Usability" subtitle in *Section 4.1.1*.

Error of BR-KFAC vs error of B-KFAC and of R-KFAC. By using the former view (and the related motivation), we see that we should typically expect[27] BR-KFAC to have a better K-factor representation error than B-KFAC, in general. When taking the latter view, comparing the BR-KFAC and R-KFAC we shall argue (in *Section 4.2.1*) that the worst-case scenario error of BR-KFAC is better than the worst-case scenario error of R-KFAC. Our numerical results in *Section 4.2.3* will show this also holds for average error, at least in the particular numerical case studied therein.

Notation. To facilitate discussions based on *lines 4 - 5* of *Algorithm 4.2*, we denote

$$\tilde{\mathcal{M}}_k^{(\text{RSVD},r)} := \tilde{U}_{\bar{\mathcal{M}},k}^{(l)} \tilde{D}_{\bar{\mathcal{M}},k}^{(l)} (\tilde{U}_{\bar{\mathcal{M}},k}^{(l)})^T = \text{RSVD}(\bar{\mathcal{M}}_k, r, r_o), \tag{4.9}$$

for our (arbitrary) EA K-factor $\bar{\mathcal{M}}_k$, where $\text{RSVD}(\bar{\mathcal{M}}_k, r, r_o)$ performs an RSVD on $\bar{\mathcal{M}}_k$ with target rank r and oversampling r_o (recall *Section 2.2.1*). With this notation, a BR-KFAC algorithm periodically over-writes $\mathcal{B}_i = \tilde{\mathcal{M}}_i^{(\text{RSVD},r)}$, with $\mathcal{B}_i$ as in (4.3).

4.1.4 The BC-KFAC Algorithm: A Cheaper Correction

Periodically overwriting $\mathcal{B}_i = \tilde{\mathcal{M}}_i^{(\text{RSVD},r)}$ may impove B-KFAC, but the over-writing operation is expensive, since it employs an RSVD (of target rank r). A cheaper alternative, is to perform a *correction* where we improve the accuracy in *only* $n_{\text{crc}} < r$ modes of our current B-KFAC representation as shown in *Algorithm 4.3*.

Algorithm 4.3: Light Correction to B-KFAC Representations (the *C-update*)

Input: $\tilde{U}_{\cdot,k}^{(l)}$, $\tilde{D}_{\cdot,k}^{(l)}$ (the B-KFAC representation), EA K-Factor $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$, number of corrected modes n_{crc} `// we have`[a] $n_{\texttt{crc}} < r < d_M - n_M$

col_idx = random_choice(r, n_{crc}) `// choose` $n_{\texttt{crc}}$ `columns out of the` r `without replcement`

`// Now project` $\bar{\mathcal{M}}_k$ `on chosen random subspace`

$\mathcal{M}_{S,k} = \big(\tilde{U}_{\cdot,k}^{(l)}[:, \text{col_idx}]\big)^T \bar{\mathcal{M}}_k \big(\tilde{U}_{\cdot,k}^{(l)}[:, \text{col_idx}]\big)$ `//` $\in \mathbb{R}^{n_{\text{crc}} \times n_{\text{crc}}}$

$V D V^T = \text{symmetric_EVD}(\mathcal{M}_{S,k})$

`// Correct` $\tilde{U}_{\cdot,k}^{(l)}$, $\tilde{D}_{\cdot,k}^{(l)}$ `in the subspace described by col_idx`

$\tilde{U}_{\cdot,k}^{(l)}[:, \text{col_idx}] \leftarrow \big(\tilde{U}_{\cdot,k}^{(l)}[:, \text{col_idx}]\big) V$; $\tilde{D}_{\cdot,k}^{(l)}[:, \text{col_idx}] = D$

Output: $\tilde{U}_{\cdot,k}^{(l)}$, $\tilde{D}_{\cdot,k}^{(l)}$ (a more accurate B-RSVD representation)

[a] **Note ($\phi_{\texttt{crc}}$ definition).** In practice (eg. in *Section 4.2.2*) we will use the $\phi_{\text{crc}} \in (0,1)$ hyperparameter, defined as $\phi_{\text{crc}} := n_{\text{crc}}/r$, to specify n_{crc}, when convenient. We have $n_{\text{crc}} = \lfloor \phi_{\text{crc}} r \rfloor$.

[27] We do not have a proof for this. We only have intuition that the B-KFAC error may significantly exceed the R-KFAC error, at least in some particular case, as argued in *Section 4.1.2* based on *Examples 4.1 - 4.2*. Still, our numerical experiments in *Section 4.2* are consistent with this claim.

The correction enforces that the projection of our new B-RSVD representation (in *line 8*) on our randomly chosen n_{crc}-dimensional subspace of $\tilde{U}^{(l)}_{\cdot,k}$ (chosen in *line 2*) be the same as the one of the true EA K-factor $\bar{\mathcal{M}}_k$. We shall refer to this correction as the *"C-update"*, in an analogous way to how we named the B-update when performing Brand's algorithm to amend the current EVD representation of K-factors, or the R-update when using RSVD for the same purpose.

The correction cannot worsen error. Higher n_{crc} is preferable error-wise. We now briefly discuss the following important facts which back-up the use of our correction: (1) Applying the correction in *Algorithm 4.3* cannot increase the Frobenius-norm error of any rank-r ($\forall r < d_M$) representation; and (2) choosing a higher n_{crc} is always better than a smaller one from a Frobenius-norm of error perspective.

Proposition 4.3 (**Form of errors for $\tilde{\mathcal{M}}_k$ and its corrected version $\tilde{\mathcal{M}}^{(c)}_k$; correction using *Algorithm 4.3*).** *Let*[28] $n_{crc} < r < d_M - n_M < d_M$. *Let an arbitrary rank-r ($r < d_M - n_M$) representation of $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$ be $\tilde{\mathcal{M}}_k$, having an SVD of the form*

$$\tilde{\mathcal{M}}_k \overset{\text{SVD}}{=} \begin{bmatrix} U & U_{\perp,1} \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & D_{\perp,1} \end{bmatrix} \begin{bmatrix} U^T \\ U^T_{\perp,1} \end{bmatrix}, \tag{4.10}$$

where $U \in \mathbb{R}^{d_M \times n_{crc}}$ and $U_{\perp,1} \in \mathbb{R}^{d_M \times (r - n_{crc})}$ are orthonormal matrices which satisfy $U^T U_{\perp,1} = 0$, and D, $D_{\perp,1}$ are diagonal matrices of positive entries not (necessarily) sorted in any order[29].

Let $\tilde{\mathcal{M}}^{(c)}_k$ be the rank-r representation obtained by applying the correction in Algorithm 4.3 to $\tilde{\mathcal{M}}_k$ with a number of columns to correct being n_{crc}, and U being[30] *the singular-vectors of the subspace we choose to correct (i.e. $U = \tilde{U}^{(l)}_{\cdot,k}[:, col_idx]$).*

We then have

$$\left\| \bar{\mathcal{M}}_k - \tilde{\mathcal{M}}^{(c)}_k \right\|_F^2 = \left\| \bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k \right\|_F^2 - \left\| U^T (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k) U \right\|_F^2. \tag{4.11}$$

Proof. See *Appendix A4.2.* □

Corollary 4.4 (**The Proposed Correction either reduces the Frobenius norm of error, or keeps it the same**). *Using the notation and set-up of Proposition 4.3, we have that*

$$\left\| \bar{\mathcal{M}}_k - \tilde{\mathcal{M}}^{(c)}_k \right\|_F^2 \leq \left\| \bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k \right\|_F^2. \tag{4.12}$$

[28] To be consistent with being able to apply the B-update.

[29] This is consistent with carrying a rank-r representation of $\bar{\mathcal{M}}_k$.

[30] Note that so far in our choices we only fixed the fact that $U \in \mathbb{R}^{d_M \times n_{\text{crc}}}$ is orthonormal and perpendicular to $U_{\perp,1} \in \mathbb{R}^{d_M \times (r - n_{\text{crc}})}$, but otherwise any such U could be chosen. We now say (for convenience) that the chosen U is the same as the singular-vectors of the subspace we apply the correction in.

Proof. Trivial by noting that $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2 \geq 0$ in (4.11). $\square$

Corollary 4.4 gives us the results that the representation error either decreases or stays the same as we apply the correction in *Algorithm 4.3* to any rank-r representation ($r < d_M - n_M$), with some $n_{\text{crc}} < r$. This is sufficient to motivate periodic (seldom) use of the proposed correction in conjunction with a series of (more frequent) B-updates: the correction surely does not increase the error at the time it is performed, and may sometimes even reduce it significantly.

However, *Proposition 4.3* gives us more insight than *Corollary 4.4*. Result (4.11) tells us that[31] a higher n_{crc} tends to give a larger error reduction upon correction, as $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2$ tends to increase with n_{crc}. To see this, think of extracting a column from $U_{\perp,1}$ and appending it to U, then note the term $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2$ cannot decrease by doing this, and this is exactly the part of the error which is set to zero by our correction (see (4.11) and the proof of *Proposition 4.3*). This fits our intuition: the more modes we attempt to correct, the more the error shall reduce.

Effect of applying the Correction on Future Iterations. While applying the correction has an immediate beneficial effect on the error, as discussed so far, we cannot claim the effect on all future iterations is beneficial with the available theory. However, we expect this would hold, and our numerical results in *Section 4.2.3* show this is indeed the case on a particular numerical case-study. The situation is similar as we had when using the RSVD-based overwriting of BR-KFAC (of *Section 4.1.3*).

Other considerations. Note that we apply the correction to $\mathcal{M}_k^{(B)}$ and not[32] to $\mathcal{B}_{k-1}$, due to our choice to truncate after the inverse application (just before the B-update). Similarly to RSVD over-writing (of BR-KFAC), we apply the correction with a smaller frequency than the one of the B-KFAC update.

We prefer selecting columns of $\tilde{U}_{:,k}^{(l)}$ at random rather than picking its largest modes for 2 reasons. First, after multiple consecutive Brand updates (and no correction / overwriting) it is possible that the largest singular modes of $\bar{\mathcal{M}}_k$ are along directions of relatively low singular values of the B-KFAC representation. Second, always picking the largest singular modes of the B-KFAC representation would tend to give us scenarios where we always correct the same modes. This is because both the EA

[31] Note that performing the correction effectively sets the $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2$ component of the error to zero. See the proof of *Proposition 4.3* for more details (in *Appendix A4.2*).

[32] But we could apply this correction to $\mathcal{B}_{k-1}$ if we preferred. A discussion, about the advantages and disadvantages of choosing one over the other is omitted here, and an in-dept investigation is subject for future work.

K-factor and the incoming update are positive semi-definite, and thus the B-KFAC representation can only underestimate[33] singular-values, but not overestimate them.

The BC-KFAC algorithm. The BC-KFAC algorithm is shown in *Algorithm 4.4*.

Algorithm 4.4: Corrected Brand K-FAC (BC-KFAC)

Insert the following 2 lines after *line 8* of *Algorithm 4.1*:
if $k \,\%\, T_{corct} == 0$ **then** `// Time to correct our representation of` $\bar{\mathcal{M}}_k$
  Perform *Algorithm 4.3* to $\{(\tilde{U}^{(l)}_{A,k}, \tilde{D}^{(l)}_{A,k})\}_l$ and $\{(\tilde{U}^{(l)}_{\Gamma,k}, \tilde{D}^{(l)}_{\Gamma,k})\}_l$

Cost reduction w.r.t. an RSVD over-writing (correction). By using the lighter correction instead of the more expensive RSVD over-writing of $\mathcal{B}_k$ (as BR-KFAC does), we can reduce our computational cost from $\mathcal{O}\big(d_M^2(r+r_o)\big)$ to $\mathcal{O}\big(d_M^2 n_{\text{crc}} + n_{\text{crc}}^3\big)$. This is substantially better if we choose $n_{\text{crc}} \leq 0.5r$.

4.1.5 The BRC-KFAC Algorithm: A mixture of BR and BC

Having introduced the R-update in *Section 3.2.1*, the B-update in *Section 4.1.1*, the C-update in *Section 4.1.4*, we can now propose an algorithm which uses all these updates together, with different frequencies. This gives us the BRC-KFAC algorithm, which requires no new update to the K-factor inverse representation to be introduced, but merely recombines the already proposed updates. The idea is that by performing B-updates with high frequency, C-updates with moderate frequency and R-updates with low frequency we can get a trade-off between the BC-KFAC and BR-KFAC algorithms.

The BRC-KFAC algorithm performs the B-update once in T_{Brand} steps, the C-update with once in T_{corct} steps, and the R-update with once in T_{BR} steps[34]. Note that while we recommend using the R-update with the lowest frequency, this is not required. We can obtain any of the previously three proposed algorithms: B-KFAC, BR-KFAC, BC-KFAC as follows. Setting T_{corct} to be very large gives a BRC-KFAC algorithm which

[33] We mean underestimate in relation to our EA K-Factors, which themselves are only estimates of the ground-truth covariance matrices. Notably, in the finite-sample estimation of covariance matrices, it is well known that the spectrum of a sample covariance computed from i.i.d. samples overestimates the large eigenvalues of the ground-truth spectrum and underestimates the low eigenvalues [84, 85]. Intuitively, this is because having few samples means the sampled subspace will be over-represented (as proportions) in the average. Given that the B-update targeting such a finite sample estimate would underestimate the leading modes, it is unclear how the B-representation of the leading modes would relate to the ground-truth in a *standard* finite-sample case. In our case, the situation is further complicated because we also have the exponential-average and the target-covariance non-stationarity aspects. However, we take the EA K-factors as a reference in our discussion - because our aim is to mimic K-FAC. We make this choice of reference both for simplicity of discussion, as well as because K-FAC has been quite successful in practice [1, 29, 30].

[34] All these T's are multiple of T_{Cov} (the period with which new K-fac information arrives - i.e. the period with which $\bar{\mathcal{M}}_k$ is updated in a true EA K-factor algorithm).

is effectively a BR-KFAC algorithm. Setting T_{BR} to be very large gives a BRC-KFAC algorithm which is effectively a BC-KFAC algorithm. Setting both $T_{BR} \to \infty$ and $T_{\text{corct}} \to \infty$ gives a BRC-KFAC algorithm which is effectively a B-KFAC algorithm.

In this chapter, BRC-KFAC is the only algorithm which we propose and do not investigate numerically. This is because BRC-KFAC is a mere combination of two algorithms which we investigate, and in this chapter we are focusing on understanding how the proposed updates (the B-, R-, and C- updates) perform, rather than finding the best possible combination of updates. We shall explore the numerical performance of BRC-KFAC in *Chapter 5*, where we implement distributed versions of our proposed algorithms, with a clear focus on getting the best possible numerical performance.

Can combine the R-, C-, B- updates differently. To get an exhaustive list of algorithms based on the 3 proposed updates, we are only missing an RC-KFAC algorithm (ignoring the SRE update of SRE-KFAC, which we found inferior to R-KFAC in *Chapter 3*). However, we omit RC-KFAC in our thesis (it remains future work).

4.1.6 Important Practical Considerations

We now discuss important considerations for implementing our proposed algorithms.

A mixture of *Randomized* and *Brand New* K-FACs. Recall our discussion in *Section 1.5* about K-Factors dimensions. In practice, we have $n_{\mathcal{A}}^{(l)} > d_{\mathcal{A}}^{(l)}$ and $n_{\Gamma}^{(l)} > d_{\Gamma}^{(l)}$ for Conv layers, but $n_{\mathcal{A}}^{(l)} = n_{\Gamma}^{(l)} = n_{\text{B}} < \min(d_{\mathcal{A}}^{(l)}, d_{\Gamma}^{(l)})$ (or even $n_{\text{B}} \ll \min(d_{\mathcal{A}}^{(l)}, d_{\Gamma}^{(l)})$) for FC layers. This means B-KFAC will only save computation time (relative to R-KFAC or SRE-KFAC) for the FC layers, and will be slower for Conv layers[35]. This issue is solved by using B-KFAC, BR-KFAC, or BC-KFAC for the FC layers only, and R-KFAC (see *Chapter 3*) for the Conv layers. When the FC layers are very wide, becoming the computational bottle-neck, speeding up the FC layers computation can give substantial improvement.

Spectrum Continuation. Both randomized K-FAC algorithms (eg. R-KFAC) and the B-KFAC variants we propose here are effectively setting $d-r$ (where r is the rank of the K-Factor estimate) eigenvalues to zero[36]. In reality, we know that the EA K-Factors eigenspectrum typically decays gradually, rather than have an abrupt jump towards zero for modes of indices larger than $r+1$ (recall *Section 3.1*), and we also know all eigen-values are non-negative by K-factor construction. Using this information one may try to correct the missing eigen-tails (missing due to low-rank truncation).

[35] We also highlighted this aspect as one of the caveats in *Section 4.1.1*.

[36] We talk about the K-factor representations we have before regularization with "$+\lambda I$".

A quick fix is to say all the missing eigenvalues are equal to the minimal one available. Using this trick, we observed slightly better performance for all algorithms (R-KFAC and all B-KFAC variants). This is probably because over-estimating the eigenspectrum is better than underestimating it, since it gives more conservative steps. This spectrum continuation trick is implementable by applying the following changes in *lines 9 - 10* of *Algorithm 3.1*: $\lambda \leftarrow \lambda + \min_i D_k[i,i]$ and $D_k \leftarrow D_k - (\min_i D_k[i,i])I$. The replacements also affect all proposed algorithms, as these amend *lines 5 - 6* of *Algorithm 3.1* which are inherited by all subsequently proposed algorithms. We use this trick for all layers and all algorithms (including R-KFAC) from here onward.

May adaptively choose: T_{corct}, T_{Brand}, T_{BR} (not implemented). As future work, we may consider adaptive schemes for choosing T_{corct}, T_{Brand}, T_{BR}. Multiple approaches are possible, but the simple ones to start from would be to have an error proxy specific for each update (taken as the norm of the change in the inverse representation, over the update). Then we would increase frequency if the error proxy value is above our desired threshold, and decrease if the converse holds.

4.2 Investigation of the K-Factor Estimation Error

4.2.1 Theoretical Error Comparison: R-KFAC vs BR-KFAC

Based on *Examples 4.1 - 4.2* we argued that, given a B-KFAC algorithm, one might expect that periodically over-writing $\mathcal{B}_k$ with the rank-r R-KFAC estimate $\tilde{\mathcal{M}}_k^{(\text{RSVD},r)}$ (by performing an RSVD on $\bar{\mathcal{M}}_k$) will help mitigating events where the B-KFAC error gets very high (above $3\times$ the optimal truncation error for the chosen target-rank r). This previous comparison was between B-KFAC and BR-KFAC, and it represented our motivation behind BR-KFAC.

In this segment, we change our point of view and think about what happens if we take a given R-KFAC algorithm with[37] $[T_{\text{Inv}}/T_{\text{Cov}}] =: \tau \in \mathbb{Z}^+ \setminus \{1\}$, and introduce B-updates (to the inverse estimates) each time the K-Factors are updated, but the RSVD inverse is not recomputed. Since we are only interested in the K-factors and not the optimization steps, we can take $T_{\text{Cov}} = 1$ w.l.o.g., so $T_{\text{inv}} = \tau$. This point of view amounts to comparing R-KFAC ($T_{\text{inv}} = \tau$) with BR-KFAC ($T_{\text{BR}} = \tau$), where new K-factor information comes in every iteration. Thus, we have to compare the error of *performing no update* versus the error of *performing B-updates*, starting from an RSVD update at $k = k_{RS}$ (for some arbitrary $k_{RS} \in \mathbb{Z}^+$). *Proposition 4.5* gives us a useful

[37] Recall the definition of T_{inv} and T_{Cov} from *Section 1.5* (just after *Algorithm 1.1*), if required.

form of the error[38] for *B-update*, as well as for *no update*. We will use the results to make claims comparing errors worst-case scenarios.

Proposition 4.5 (**Error of Doing nothing vs Error of B-updates**). *Let $\tilde{\mathcal{M}}_k$ be an approximation of the EA K-factor $\bar{\mathcal{M}}_k$ which is obtained by performing an* R-KFAC *update at $k = k_{RS}$ (for some $k_{RS} \in \mathbb{Z}^+$), and either no other update thereafter, or B-updates (every step) thereafter. The error in representing $\bar{\mathcal{M}}_k$ when using either one of these approximation schemes is of the form*

$$\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k = \rho^{k-k_{RS}} E_{k_{RS}} + (1-\rho) \sum_{j=k_{RS}+1}^{k} \rho^{k-i} E_j, \;\; \forall k \geq k_{RS}+1. \tag{4.13}$$

When performing RSVD *initially (at[39] $k = k_{RS}$), and no update thereafter we have*

$$\begin{aligned} E_{k_{RS}} &= \bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}, \;\; \textit{and} \\ E_j &= M_j M_j^T - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)} \;\; \forall j \in \{k_{RS}+1, ..., k\}. \end{aligned} \tag{4.14}$$

When performing RSVD *initially (at $k = k_{RS}$), and B-updates thereafter we have*

$$\begin{aligned} E_{k_{RS}} &= \bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}, \;\; \textit{and} \\ E_j &= \frac{1}{1-\rho}\big(\tilde{\mathcal{M}}_j^{(B)} - \mathcal{B}_j\big) \;\; \forall j \in \{k_{RS}+1, ..., k\}, \end{aligned} \tag{4.15}$$

where the B-process quantities $\tilde{\mathcal{M}}_j^{(B)}$ and $\mathcal{B}_j$ are for a B-process as in (4.3) but with "initialization" of $\mathcal{B}_{k_{RS}} = \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}$ at $k_{RS} \geq 1$, rather than of $\tilde{\mathcal{M}}_0^{(B)} = 0$ at $k = 0$.

Proof. See *Appendix A4.3*. □

Firstly, by the form of $\{E_j\}_{j=k_{RS}+1}^{k}$ for the "no update thereafter" result (4.14), we get the obvious interpretation that performing no update thereafter is equivalent to performing an update where we pretend $M_j M_j^T = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ for all $j \geq k_{RS}$. This is apparent as $E_j = 0$ if our pretended $M_j M_j^T = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ would indeed hold.

Secondly, by showing that the worst-case E_j of the no-update is higher than the worst-case E_j of B-updated $\tilde{\mathcal{M}}_k$ sequence, *Proposition 4.5* can be used to claim that at least the worst-case scenario is improved when doing B-updates (as opposed to doing no updates). *Proposition 4.6* formalizes these worst-case error claims for E_j based on the error forms (4.14) and (4.15) in *Proposition 4.5*.

Proposition 4.6 ($\|E_j\|_F \;\forall j \in \{k_{RS}+1, ..., k\}$**: Comparison for No-update vs for B-update**). *For the E_j errors as defined in Proposition 4.5, we have the following statements holding:*

[38] Measured as the difference between the true EA-Kfactor and our approximate representation.
[39] For arbitrary $k_{RS} \in \mathbb{Z}^+$.

1. *When performing* RSVD *initially (at $k = k_{RS}$), and no updates thereafter $\|E_j\|_F$ can get as high as*

$$\|E_j\|_F = \sqrt{\left\|M_j M_j^T\right\|_F^2 + \left\|\tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}\right\|_F^2}, \ \ \forall j \in \{k_{RS}+1, ..., k\}. \tag{4.16}$$

2. *When performing* RSVD *initially, and B-updates thereafter $\|E_j\|_F$ is bounded as:*

$$\|E_j\|_F \leq \left\|M_j M_j^T\right\|_F, \ \ \forall j \in \{k_{RS}+1, ..., k\}, \ \ \textit{and} \ \ E_k = 0. \tag{4.17}$$

Proof. See *Appendix A4.4*. □

The intuition behind *Proposition 4.6* (and its proof) is simple. Equation (4.15) tells us that, the error E_j is proportional to the optimal error encountered when truncating $\tilde{\mathcal{M}}_j^{(B)}$. By using the $\tilde{\mathcal{M}}_j^{(B)} = \rho\mathcal{B}_{j-1} + (1-\rho)M_j M_j^T$ line of the B-process (4.3), and (4.15), we see that we can choose our rank-r $\mathcal{B}_j$ at least[40] as good as $\mathcal{B}_j = \rho\mathcal{B}_{j-1}$. By making this choice, we get our error to be of norm $\|E_j\|_F = \left\|M_j M_j^T\right\|_F$, but our B-process chooses $\mathcal{B}_j$ to minimize $\|E_j\|_F$. That is how the bound (4.17) arises.

Conversely, by performing no update, we are effectively performing an update where we are saying that $M_j M_j^T = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ (s.t. the current representation does not change by applying the fictitious EA update to the EA K-factor). By performing this (no-update) update, if $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ and $M_j M_j^T$ belong to orthogonal subspaces[41], the squared Frobenius norm of the error introduced at iteration j (E_j) is the sum of such errors for two terms: (1) the first term $\left\|M_j M_j^T\right\|_F^2$ is due to ignoring the incoming information (since we perform no update), and (2) the second term $\left\|\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\right\|_F^2$ is due to not decaying the $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ as should happen in an EA K-factor construction if the update was performed (since $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ is old information at iteration $k \geq k_{RS}+1$). Following the proof (in *Appendix A4.4*) will provide further insight.

Note that $\left\|\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\right\|_F^2$ is always positive. Using (4.13) and triangle inequality, we see that the overall EA K-factor error norm has the upper bound

$$\left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k\right\|_F \leq \rho^k \left\|\bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\right\|_F + (1-\rho)\sum_{j=k_{RS}+1}^{k} \rho^{k-j}\|E_j\|_F. \tag{4.18}$$

in both cases (no-update and B-update thereafter). The upper-bound (4.18) is also tight (attainable) in practical cases when all E_j's are proportional[42] to each other.

[40] This is just a possible rank-r choice, and the optimal choice is at least as good as any arbitrary choice.

[41] Which is possible because $M_j \in \mathbb{R}^{d_M \times n_M}$ and $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} \in \mathbb{R}^{d_M \times r}$ and to have B-KFAC applicable we need $r < d_M + n_M$.

[42] $\upsilon_{k_{RS}} E_{k_{RS}} = \upsilon_{k_{RS}+1} E_{k_{RS}+1} = ... = \upsilon_k E_k$, for some scalars $\{\upsilon_j\}_j$. Though this is unlikely under reasonable distributions on M_j

With this in mind, using *Proposition 4.6*, we see that the overall error is better *under the worst case scenario* for B-updates than under the worst-case scenario for no-updates.

Our results here are comparing the worst-case occurrences based on deterministic arguments. While insightful, these are relatively weak, as they do not tell us much about typical occurrences (the worst-case scenarios may be very unlikely, depending on the probability measure on M_j). Obtaining probabilistic typical-case results (or w.h.p. result) is future work. Still, based on *Propositions 4.5 - 4.6*, it is clear we shall expect that for sufficiently many iterations[43], we will in practice see better worst-case occurrence for BR-KFAC than for R-KFAC (where the R-update frequency is the same for both algorithms). Thus, in this respect, introducing B-updates in between R-updates also improves the worst-case scenario[44] (in the same way as introducing the R-update do B-KFAC did).

4.2.2 Numerical Error Investigation: Experimental Set-up

We now look at the error of B-updates numerically as a way of complementing the theoretical results. To do so, we consider the following setup. We fix the frequency at which the updates to K-Factors are incoming (i.e. fix T_{Cov}; here we set $T_{\text{Cov}} = 10$). For our fixed T_{Cov}, a K-FAC algorithm with $T_{\text{inv}} = T_{\text{Cov}} = 10$ always maintains the inverse K-factors at their exact values. Thus, we take this to be the benchmark in our numerical error measurements. We then ask: what is the error between an algorithm which does not hold the exact value of inverse K-factors (for example an R-KFAC, a B-KFAC, or a even K-FAC with $T_{\text{inv}} > T_{\text{Cov}}$) and the benchmark? In principle, we could compute this error at each and every step. However, doing so is very expensive. We thus choose to only compute the error for two sequences of 300 consecutive steps - one starting at epoch 15, and one starting at epoch 30. This is sufficient to draw conclusions. To clarify, the experiment is set-up such that we have the same "carrier" steps (which are the highest considered frequency K-FAC steps), and from those locations we also compute the required quantities for the other algorithms. The reason for doing this is to have the same path and thus get a more fair comparison.

Many **error metrics** could be used. We consider four which we believe are most relevant: *(1) Norm Error in* $\mathcal{A}^{-1}$: $\left\|\tilde{\mathcal{A}}_k^{-1} - \mathcal{A}_{k;(\text{ref})}^{-1}\right\|_F$ / $\left\|\mathcal{A}_{k;(\text{ref})}^{-1}\right\|_F$, *(2) Norm*

[43] Which of course may be practically too large to get near to, in which case we will not see what the theory implies here in our experiments.

[44] Which is of course not a guarantee that the typical scenario (or most scenarios) will be improved s.t. we see an over-all better error. But we would nevertheless intuitively expect so.

Error in Γ^{-1}*:* $\left\|\tilde{\Gamma}_k^{-1} - \Gamma_{k;(\text{ref})}^{-1}\right\|_F$ / $\left\|\Gamma_{k;(\text{ref})}^{-1}\right\|_F$, *(3) Norm Error in Subspace Step:* $\left\|\tilde{s}_k - s_{k;(\text{ref})}\right\|_F$ / $\left\|s_{k;(\text{ref})}\right\|_F$, *(4) Angle Error in Subspace Step:* $1 - \cos(\angle[\tilde{s}_k, s_{k;(\text{ref})}])$. Here, quantities marked with *tilde* represent the ones of our approximate algorithms, while quantities marked with *"ref"* represent the ones of the reference (benchmark) algorithm (K-FAC with $T_{\text{inv}} = T_{\text{Cov}} = 10$).

Recall that, our proposed algorithm requires K-factors with $r + n_M < d_M$. To satisfy this constraint, we will focus on the FC layers only. The network architecture[45] we use is the one given in *Section 4.5*. This only has two FC layers, out of which it only makes sense to perform B-updates for the first FC layer. Thus, *our error metrics will only relate to this first FC layer* (marked as *"FC layer 0"* in figures). Note that the steps s_k considered in the above paragraph are the subspace steps[46] (in the *FC layer 0* parameters subspace), thus slightly overwriting the notation, just for this section.

The **algorithms we consider** are the ones introduced in *Section 4.1*: (1) B-KFAC with $T_{\text{Brand}} = 10$; (2) BR-KFAC with $T_{\text{Brand}} = 10$, $T_{\text{BR}} = 50$, (3) BC-KFAC with $T_{\text{Brand}} = 10$; $T_{\text{corct}} = 50$, $\phi_{\text{crc}} = 0.5$; (4) R-KFAC with $T_{\text{inv}} = 50$; (5) R-KFAC with $T_{\text{inv}} = 10$; (6) R-KFAC with $T_{\text{inv}} = 300$; (7) K-FAC with $T_{\text{inv}} = 50$. For all these algorithms, new K-factor data is incoming with period $T_{\text{Cov}} = 10$. All unspecified algorithms hyperparameters are as described in *Section 4.5*. Note that R-KFAC with $T_{\text{inv}} = 300$ is meant to show how the error would increase with k if no update is performed to the inverse K-factors - which are initially estimated in the R-KFAC style. Comparing R-KFAC $T_{\text{inv}} = 300$ and BR-KFAC directly relates to the theoretical result in *Section 4.2.1*. Other comparisons also give useful insights, as we shall see.

Note that for all algorithms based on low-rank truncation, the eigenspectrum is continued as explained in *Section 4.1.6*. We always start our sequence of 300 steps (over which the error is measured) exactly when the heaviest update of the algorithm at hand is performed. For this reason, the error measuring of R-KFAC with $T_{\text{inv}} = 300$ starts slightly later in the epoch. However since we practically found that the eigenspectrum profile varies slowly with k, this difference is immaterial.

4.2.3 Numerical Error Investigation: Results

Figure 4.2 shows the error metrics (1) and (2). *Figure 4.3* shows the error metrics (3) and (4). Error metric and epoch vary across columns and rows respectively.

[45]The learning problem is CIFAR10 classification with slightly amended VGG16_bn.

[46]Not the full-space steps, which are vectors containing step-entries for all layers (not just the FC 0 layer).

Error Periodicity and "Reset Times". Note that aside from B-KFAC and R-KFAC with $T_{\text{inv}} = 300$, all other algorithms have a period of 50 steps. This arises because these algorithms have a heavier update every 50 steps (either an RSVD-overwriting of "$\mathcal{B}_k$", or a correction), and a lighter (or no) update every 10 steps. The numerical results show that performing an over-writing of "$\mathcal{B}_k$" (see BR-KFAC) or an EVD re-computation of inverse K-factor (see K-FAC $T_{\text{inv}} = 50$) always resets the error back to roughly the same level, while performing a correction reduces the error down to more variable levels (see BC-KFAC). This is intuitive: unlike the other two heavier updates, the correction's output depends on the approximate K-factor inverse representation to be corrected.

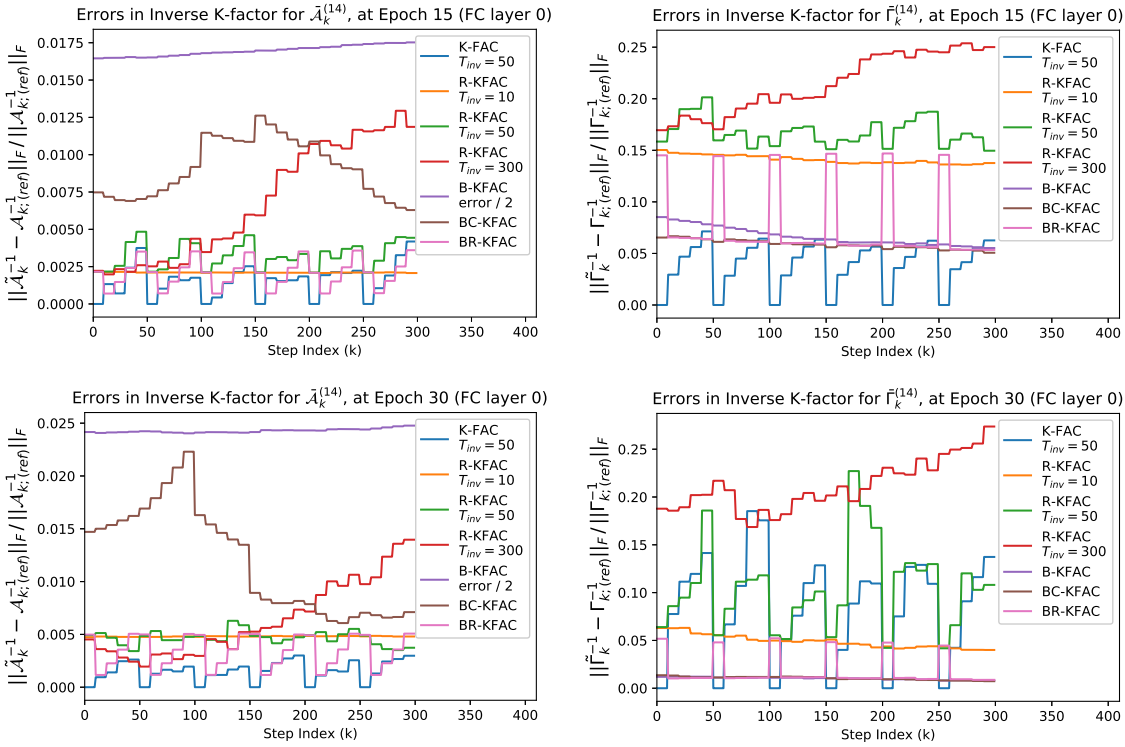


Figure 4.2: Error in inverse K-Factors. Error metric (1) is on the first column of plots. Error metric (2) is on the second columns of plots. Each curve is averaged over 5 runs. K-factor inverses used in this section were computed with "$+\lambda I$" (regularization).

R-KFAC $T_{\text{inv}} = 50$ vs K-FAC $T_{\text{inv}} = 50$. We see that the error patterns of these two algorithms are very similar. The error of R-KFAC is only mildly larger than the one of K-FAC, showing there is significant eigenspectrum decay. This reconciles well with our findings in the previous chapter[48].

[48] In *Section 3.2.4* we saw theoretically that the RSVD total error is typically no larger than 3× the

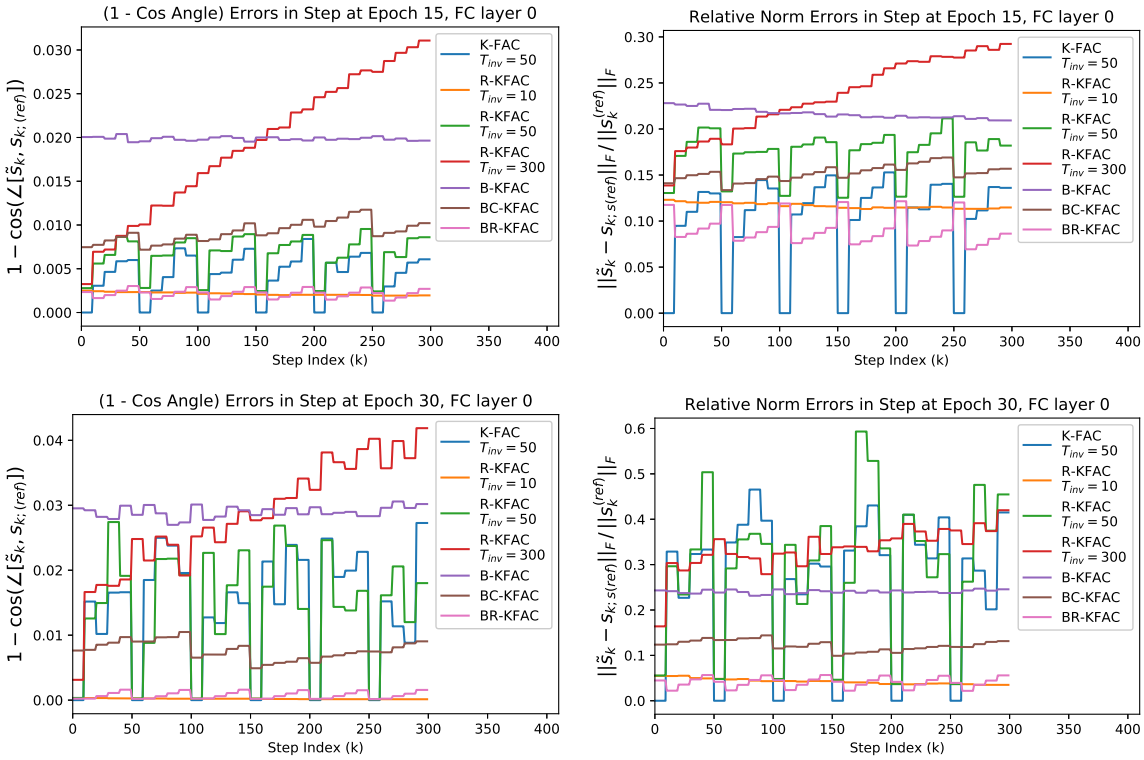


Figure 4.3: Error in Step. Error metric (3) is on the first column of plots. Error metric (4) is on the second columns of plots. Since the gradient changes at each location, the error in step typically changes on every iteration[47], even though the error in K-Factor can change only once every 10 iterations. Thus, we effectively have 10 different samples from the distribution of step-error for each given {approximate K-Factor representation, Benchmark K-Factor} pair. We report the average over these 10 samples only (giving us a constant error over iterations 200-209 for instance). As before, we further average over 5 runs. Inverses used in this section were computed with "$+\lambda I$" (regularization).

BR-KFAC vs R-KFAC: Relationship to *Propositions 4.5* and *4.6*. There are two important comparisons to note here. First, comparing BR-KFAC and R-KFAC with $T_{\text{inv}} = 50$, we see that performing a B-update is almost always better than performing no update in terms of all error metrics and for all considered epochs (at least for the considered set-up). This result for the first two error metrics strongly relates[49] to our conclusion drawn in *Section 4.2.1* based on *Propositions 4.5 - 4.6*, but further to the

optimal truncation error, while in *Appendix A3.6* we numerically saw this factor is more like 1.2× in practice. We also saw in *Section 3.1.3.3* (and even in *Appendices A3.4 - A3.5*) that the optimal truncation error is expected to be low for a target-rank of $r \approx n_{\text{B}}$ (here we chose $r = 220$, $n_{\text{B}} = 256$). Putting these pieces of information together, the total error of our RSVD representation ought to be low.

[49]The relation is not perfect because the error metrics (1) and (2) consider the more practical error based on inverses, and uses spectrum continuation (see *Section 4.1.6*).

weaker result that the theory predicts, it shows that the error in K-Factors is almost always better when a B-update is performed than when no update is performed[50].

Secondly, comparing the results of B-KFAC and R-KFAC $T_{\text{inv}} = 300$ is effectively comparing the steady-state[51] error of B-KFAC with the error of "no reset" R-KFAC (which performs one R-KFAC update at the beginning of the examined period, and no updates thereafter). We see that for all error metrics, the error of B-KFAC is fairly constant while the error of R-KFAC $T_{\text{inv}} = 300$ grows fast. The error of R-KFAC $T_{\text{inv}} = 300$ very rapidly exceeds the one of B-KFAC for the error metric (2). In relation to *Proposition 4.5*, this suggests that $\|E_j\|_F$ is much larger for no-update than for B-updates when $\bar{\mathcal{M}}_k = \bar{\Gamma}_k$. While the same phenomenon did not occur on the 300 steps interval considered for error metric (1), this would have occurred about 900 steps later (numerical results obtained but not shown here).

From a more practical point of view, the step-related error metrics (in *Figure 4.3*) of "no reset" R-KFAC very rapidly increase past the steady-state error of B-KFAC, despite the behaviour in error metric (1). Thus, the errors we are most interested in are much more favorable when using B-updates than when using no updates. As an example, at epoch 15, it took only 10-15 skipped inverse-updates after an R-KFAC update for the step-error to exceed the steady-state level of B-KFAC error.

B-KFAC vs BR-KFAC: Relationship to *Examples 4.1 - 4.2*. *Figures 4.2* and *4.3* show that when one adds periodic RSVD overwritings to a B-KFAC algorithm, the error is in general better for all iterations (compare plots of B-KFAC with plots of BR-KFAC). This aligns with the intuition we have developed around the particular 2D case in *Examples 4.1 - 4.2*. There is a slight exception for the $\left\|\tilde{\Gamma}_k^{-1} - \Gamma_{k;(\text{ref})}^{-1}\right\|_F / \left\|\Gamma_{k;(\text{ref})}^{-1}\right\|_F$ error metric: at the time RSVD is performed we get a significant spike in the BR-KFAC error, which then drops below the B-KFAC error again, as soon as the B-update is performed. This occurred because the RSVD projection error happened to be higher than the B-KFAC excess[52] error (compare the R-KFAC error curve with the B-KFAC error curve). However, this exception does not also occur on the more practical step-related metrics of *Figure 4.3*, and it is ultimately these error metrics we care most about.

BC-KFAC vs B-KFAC: Relationship to *Proposition 4.3* and *Corollary 4.4*. In *Section 4.1.4* based on *Proposition 4.3* and *Corollary 4.4*, we saw that applying

[50] Compare to our theoretical discussion in *Section 4.2.1* based on *Propositions 4.5 - 4.6* which was based on worst-case scenario only.

[51] We did not prove theoretically that the error of our B-KFAC reaches a steady state. However, we can see here numerically that it appears to do so.

[52] Relative to optimal truncation error.

Table 4.1: Average Error Metrics of Proposed Algorithms and Benchmark. Average is for each error curve at epoch 30. We also show t_{epoch} in the format mean ± SD.

Optimizer	Avg. Err. metric 1	Avg. Err. metric 2	Avg. Err. metric 3	Avg. Err. metric 4	t_{epoch} (s)
K-FAC $T_{\text{inv}} = 50$	1.51e-03	8.42e-02	2.69e-01	14.4e-02	169 ± 5.6
R-KFAC $T_{\text{INV}} = 50$	4.65e-03	10.4e-02	2.96e-01	14.6e-02	58.5 ± 0.6
R-KFAC $T_{\text{INV}} = 10$	4.8e-03	4.9e-02	0.43e-01	0.02e-02	71.2 ± 1.0
B-KFAC	48.6e-03	1.02e-02	2.40e-01	29.0e-02	59.2 ± 0.6
BC-KFAC	11.9e-03	1.04e-02	1.22e-01	0.77e-02	59.4 ± 0.6
BR-KFAC	3.40e-03	1.81e-02	0.40e-01	0.74e-02	59.5 ± 0.6

a correction (as in *Algorithm 4.4*) is better than not applying a correction from the point of view of the *current* K-factor estimate (and thus current step). However, the theory we developed did not allow us to say future iterations following the correction are also better (than if no correction was performed). Intuitively, we might believe this to be the case, though had no proof. Based on numerical results in *Figures 4.2 - 4.3*, by comparing the BC-KFAC and B-KFAC plots we see this is indeed the case, at least for the studied numerical example.

B-KFAC vs BR-KFAC vs BC-KFAC. The error of BC-KFAC lies in between the one of (the more expensive) BR-KFAC and (the cheaper) B-KFAC. Thus, BC-KFAC allows us to trade between CPU time and error by tuning ϕ_{crc}.

Average Error in Relation to Time-per-Epoch (t_{epoch}). It is instructive to consider the relationship between average error and t_{epoch}. *Table 4.1* summarizes the results in *Figures 4.2* and *4.3*, but also shows t_{epoch} measurements. The average-error order obviously carries on from the one observable in *Figures 4.2* and *4.3*, while the t_{epoch} ordering is what one would expect based on our discussions in *Section 4.1*. Note that when considering t_{epoch}, the fair comparison between B-KFAC and R-KFAC is when the latter has $T_{\text{inv}} = 10$ (both perform an inverse-update every 10 steps). In this case, we see that B-KFAC is much cheaper than R-KFAC. However, increasing T_{inv} to $T_{\text{inv}} = 50$ gets R-KFAC slightly cheaper per epoch than B-KFAC.

No algorithm has all metrics better than any other. Thus, we cannot say which one will give better optimization performance purely based on this table. There is a trade-off to be made between time and error, and we cannot say which trade-off results in better time to target accuracy a priori. However, we can see that R-KFAC has significantly smaller t_{epoch} than K-FAC with only marginally larger error, suggesting R-KFAC will most likely perform better in practice (reconciling with numerical findings in *Chapter 3*). We can also see we dramatically reduce the error of R-KFAC ($T_{\text{inv}} = 50$)

by adding in B-updates (getting BR-KFAC), while the t_{epoch} overhead is minimal (reconciles with our discussion in *Sections 4.1.2 - 4.1.3*). Finally, we see some of BR-KFAC's accuracy can be given away in exchange for slightly smaller t_{epoch} by turning the RSVD-overwriting either into a correction (getting BC-KFAC), or into a B-update (getting B-KFAC). Note that changes in t_{epoch} are relatively small when taking out/putting in B-updates, because these updates are applied only to the first FC layer, but t_{epoch} measures computations across all the layers.

4.3 Proposing a Linear-time Inverse Application

So far, we have seen that for certain layers we can use the B-update to obtain low-rank inverse representations of K-factors in linear time[53] in d_M (though with no error guarantee). To reap the most benefits, we would like the inverse application itself to also scale no worse than linear in d_M. With the inverse application procedure proposed in *Chapter 3*, we saw we could make it quadratic. In this section, we argue that, for the layers where applying the B-update makes sense, we can make our inverse application linear in d_M as well. However, in our numerical experiments we did not implement this feature, and used the quadratic-time inverse application of *Chapter 3*. This is because implementing the linear inverse application required "opening up the hood" of the practical back-prop pytorch algorithms. We regarded this as a future software development work (outside the scope of our thesis) with strong motivation behind it, based on our discussion in this chapter.

We now look at our proposed linear-time inverse application. The gradient of each layer (in matrix form, $\text{Mat}(g^{(l)})$) is of the form (at iteration k)

$$\text{Mat}(g_k^{(l)}) = G_k^{(l)}[A_k^{(l)}]^T, \tag{4.19}$$

where $G_k^{(l)} \in \mathbb{R}^{d_\Gamma^{(l)} \times n_M^{(l)}}$ and $G_k^{(l)} \in \mathbb{R}^{d_\mathcal{A}^{(l)} \times n_M^{(l)}}$ are the same matrices as the ones used to generate the EA K-factors as (review *Section 1.5*).

Thus, we see that whenever the B-update is applicable ($n_M^{(l)} + n_{\text{B}} < d_\mathcal{A}^{(l)}$ for the "A" K-factors, and $n_M^{(l)} + n_{\text{B}} < d_\Gamma^{(l)}$ for the "G" K-factors), we can make further computational savings by avoiding the multiplication of $G_k^{(l)}[A_k^{(l)}]^T$ in the backward pass, and instead applying the inverse representation by first taking a product with $G_k^{(l)}$, and then with $A_k^{(l)}$. That is, we compute the product $[\bar{\Gamma}^{(l)}]^{-1}\text{Mat}(g_k^{(l)})[\bar{\mathcal{A}}_k^{(l)}]^{-1}$ as

$$[\bar{\Gamma}_k^{(l)}]^{-1}\text{Mat}(g_k^{(l)})[\bar{\mathcal{A}}_k^{(l)}]^{-1} = \left([\bar{\Gamma}_k^{(l)}]^{-1}G_k^{(l)}\right)\left([A_k^{(l)}]^T[\bar{\mathcal{A}}_k^{(l)}]^{-1}\right), \tag{4.20}$$

[53] Compare to cubic time in d_M as is typically done in standard K-FAC, or with quadratic time in d_M as we did in *Chapter 3*.

but of course we use our low-rank inverse representations for $[\bar{\Gamma}_k^{(l)}]^{-1}$ and $[\bar{\mathcal{A}}_k^{(l)}]^{-1}$ rather than the standard EVD inverses. *Algorithm 4.5* shows how this works in practice.

Algorithm 4.5: Linear Inverse Application (applicable when B-UPDATE is)

1 `// Required: low-rank inverse representation of` $[\bar{\Gamma}^{(l)}]^{-1}$, $[\bar{\mathcal{A}}_k^{(l)}]^{-1}$

2 Modify the back-prop[a] to return $G_k^{(l)}$, $A_k^{(l)}$ rather than $\text{Mat}(g_k^{(l)})$

3 Replace the *lines 8 - 10* of *Algorithm 3.1* with: `// Apply the low-rank inverse representation with linear time in` d_Γ `and` $d_\mathcal{A}$

4 `// Estimate` $[A_k^{(l)}]^T[\bar{\mathcal{A}}_k^{(l)}]^{-1}$`, use the low-rank representation of` $\bar{\mathcal{A}}_k^{(l)}$`:`

5 $[\mathbf{A}_k^{(l)}]^T := [A_k^{(l)}]^T\tilde{V}_{A,k}^{(l)}\big[(\tilde{D}_{A,k}^{(l)} + \lambda I)^{-1} - \frac{1}{\lambda}I\big](\tilde{V}_{A,k}^{(l)})^T + \frac{1}{\lambda}[A_k^{(l)}]^T$; `//` $\mathcal{O}(rd_\mathcal{A}^{(l)}n_M^{(l)})$

6 `// Estimate` $[\bar{\Gamma}_k^{(l)}]^{-1}G_k^{(l)}$`, use the low-rank representation of` $\bar{\Gamma}^{(l)}$`:`

7 $\mathbf{G}_k^{(l)} := \tilde{V}_{\Gamma,k}^{(l)}\Big[(\tilde{D}_{\Gamma,k}^{(l)} + \lambda I)^{-1} - \frac{1}{\lambda}I\Big](\tilde{V}_{\Gamma,k}^{(l)})^T G_k^{(l)} + \frac{1}{\lambda}G_k^{(l)}$; `//` $\mathcal{O}(rd_\Gamma^{(l)}n_M^{(l)})$ `flops`

8 `// Construct the Preconditioned Step from Partial Quantities`[b]

9 $S_k^{(l)} = \mathbf{G}_k^{(l)}[\mathbf{A}_k^{(l)}]^T$ `//` $S_k^{(l)}$ `is the step for layer` l `in matrix form`

[a] While we do have access to the matrices $G_k^{(l)}$ and $A_k^{(l)}$ in a standard K-FAC algorithm with typical back-prop, this is done through a backward and forward-pass hook. By using only using the hooks, and not modifying the back-prop, we would still compute the unnecessary quantities $\{G_k^{(l)}[A_k^{(l)}]^T\}_{l,k}$. We thus need an actual modification of the back-prop to also save on this wasteful operation - and if we do not save on this operation our inverse-application is actually not of linear time-cost. Notably, the reason we can claim that *Algorithm 4.5* has over-all linear cost in layer width (despite the fact that *line 9* is of quadratic complexity in layer width) is because by not performing the product $G_k^{(l)}[A_k^{(l)}]^T$ in the backward-pass, we can then consider the $S_k^{(l)} = \mathbf{G}_k^{(l)}[\mathbf{A}_k^{(l)}]^T$ product (of the same cost as the one we skip) as being performed as part of the backward pass. So effectively we can think as though the inverse is applied "in the middle of the backward pass", and the extra cost for doing that is linear. If on the other hand we do not remove the wasteful computation $G_k^{(l)}[A_k^{(l)}]^T$ in the backward-pass, we cannot "absorb" the cost of *line 9* into the backward pass anymore - and our inverse-application cost becomes quadratic. As a simpler implementation solution, we could use JAX which would "self-optimize away" the computation of unnecessary quantities mentioned above.

[b] Note that the matrices $[\mathbf{A}_k^{(l)}]^T \approx [A_k^{(l)}]^T[\bar{\mathcal{A}}_k^{(l)}]^{-1}$ and $\mathbf{G}_k^{(l)} \approx [\bar{\Gamma}_k^{(l)}]^{-1}G_k^{(l)}$ are the preconditioned $[A_k^{(l)}]^T$ and $G_k^{(l)}$, and they are first defined in *Algorithm 4.5*.

We thus see that whenever the B-update is applicable ($n_M^{(l)} + n_\text{B} < d_\mathcal{A}^{(l)}$ for the "A" K-factors, and $n_M^{(l)} + n_\text{B} < d_\Gamma^{(l)}$ for the "G" K-factors), we reduce the time scaling of our inverse application from $\mathcal{O}(r([d_\Gamma^{(l)}]^2 + [d_\mathcal{A}^{(l)}]^2))$ (as we had in R-KFAC, *Algorithm 3.1*), to $\mathcal{O}(r(d_\Gamma^{(l)} + d_\mathcal{A}^{(l)})n_M^{(l)})$. This inverse application methodology offers:

1. An improved inverse application complexity down to linear in layer size for all K-factors (we saw in *Section 3.1.3* that n_M is fixed with layer size while d_M is proportional to layer size for both FC and Conv layers);

2. A concrete, practical computational saving whenever the B-update is applicable[54] (since $n_M < d_M$ when this happens).

Notably, the idea of separating the inverse application as in equation (4.20) was first introduced in the original K-FAC paper (see top of *page 31* in *Section 8* of [1]). While this idea of separating the product (using associativity) is the main driver behind the approach proposed here (and this is indeed as in [1]), it is worth noting that there are 2 practical differences between the inverse-application method proposed here and the one in the K-FAC original paper ([1]). These differences arise because we have a low-rank representation of the inverses $[\bar{\Gamma}_k^{(l)}]^{-1}$ and $[\bar{\mathcal{A}}_k^{(l)}]^{-1}$ rather than the actual inverses (as in standard K-FAC), and are: 1) The time-complexity is linear in K-factor size for us ($\mathcal{O}(r(d_\Gamma^{(l)} + d_\mathcal{A}^{(l)})n_M^{(l)})$ per layer) but quadratic in the original K-FAC paper ($\mathcal{O}(([d_\Gamma^{(l)}]^2 + [d_\mathcal{A}^{(l)}]^2)n_M^{(l)})$ per layer); 2) The final formulae are different (see lines 5 and 7 in *Algorithm 12* and compare those with the top equation on *page 31* of [1] - which is the same as (4.20)).

4.4 Overview of the Proposed Algorithms

We now present a summary of our proposed algorithms in *Table 4.2*, and compare them against R-KFAC (porposed in *Chapter 3*) and the starting point, K-FAC. Note that $\mathcal{O}(d_M)$ memory requirement means the algorithm is low-memory. We only give error rankings in *Table 4.2* (based on numerical experiments in *Section 4.2.3*). We have just seen a more detailed summary[55] of the errors in *Table 4.1*, and the reader can look therein (or even at *Figures 4.2 - 4.3*) for more details.

Note that K-FAC's error is set to zero merely as a convention: we choose K-FAC as the absolute benchmark. But we should realize that all K-FAC variants try to estimate the NG step. That is, each algorithm has an inverse representation procedure which ultimately aims to approximate the Fisher application. Since the K-FAC Fisher representation is itself relatively crude[56] we cannot be certain that departing from it will give worse steps. Thus, it could be that there are algorithms with relatively high error w.r.t. the K-FAC benchmark and still get better per-step progress than K-FAC.

[54] In fact, the condition is looser: we have practical computational savings whenever the K-factor falls into *"region II"* of *Figure 3.1* in *Section 3.1.3.1.*

[55] Note that in *Table 4.2*, $\tilde{\mathcal{O}}(d_M^2)$ suggests the actual complexity may or may not have the $\log(d_M)$ term, depending on whether we choose to control the projection error or not.

[56] Using $\mathbb{E}[A \otimes B] \approx \mathbb{E}[A] \otimes \mathbb{E}[B]$. The K-factor itself is estimated with little information (batch) and rarely, as well as uses an EA.

Table 4.2: Summary of Proposed Algorithms and Benchmark.

Optimizer	Time complexity[A] of getting the "inverse"	Time complexity[A] of applying the "inverse"	Overall time complexity[A]	Memory Requirement (Complexity)	K-Factor Representation Error Ranking[B]
K-FAC	$\mathcal{O}(d_M^3)$	$\mathcal{O}(d_M^3)$	$\mathcal{O}(d_M^3)$	$\mathcal{O}(d_M^2)$	I (zero error by definition)
R-KFAC	$\mathcal{O}(d_M^2)$ or[C] $\mathcal{O}(\log(d_M)d_M^2)$	$\mathcal{O}(d_M^2)$	$\mathcal{O}(d_M^2)$ or[C] $\mathcal{O}(\log(d_M)d_M^2)$	$\mathcal{O}(d_M^2)$	IV
B-KFAC[D]	$\mathcal{O}(d_M)$	$\mathcal{O}(d_M)$	$\mathcal{O}(d_M)$	$\mathcal{O}(d_M)$	V (the worst; still reasonable)
BC-KFAC	$\mathcal{O}(d_M)$ mostly $\tilde{\mathcal{O}}(d_M^2)$ rarely	$\mathcal{O}(d_M^2)$	$\tilde{\mathcal{O}}(d_M^2)$	$\mathcal{O}(d_M^2)$	III
BR-KFAC	$\mathcal{O}(d_M)$ mostly $\tilde{\mathcal{O}}(d_M^2)$ rarely	$\mathcal{O}(d_M^2)$	$\tilde{\mathcal{O}}(d_M^2)$	$\mathcal{O}(d_M^2)$	II

[A] We drop all non-d_M terms in our complexities here, for convenience. See complete complexities where each algorithm is first introduced.
[B] See *Table 4.1* for numerical values. $\mathcal{O}(d_M^2)$ if we are happy not to control the error.
[C] $\mathcal{O}(\log(d_M)d_M^2)$ when we are controlling the error; $\mathcal{O}(d_M^2)$ if we are happy not to control the error.
[D] Inverse and over-all time-complexities achieved in our numerical experiments here are $\mathcal{O}(d_M^2)$, because as we discussed, we did not implement the $\mathcal{O}(d_M)$ inverse application subroutine.

Nevertheless, we take K-FAC as the benchmark because it is easy to compare with, and we know its step tends be relatively good in practice [1, 29, 51, 53].

We Proposed a Low-Memory Improvement (Version) of K-FAC. Importantly, one of our *Brand New K-FACs* is low-memory (B-KFAC). To our knowledge, this is the first K-FAC implementation where the **storing requirements** (of quantities specific to taking a NG step) **scale linearly in layer size**, as opposed to quadratic. This means we enable using K-FAC on much wider nets for a fixed memory machine/cluster. Of course, we need $r + n_M < d_M$ to be able to use B-KFAC and see an improvement. Note that we cannot use BR-KFAC and BC-KFAC as low-memory, since they require to store the EA K-Factors (i.e. store $\bar{\mathcal{M}}_k$), and thus have quadratic memory-complexity.

4.5 Numerical Performance of Proposed Optimizers

We now compare the numerical performance of our proposed algorithms: B-KFAC, BR-KFAC or BC-KFAC, with relevant benchmark algorithms: R-KFAC (*Algorithm 3.1*; [81]), K-FAC ([1]), and SENG (the state of art implementation of NG for DNNs; [20]). We consider the CIFAR10 classification problem with a modified[57] version of

[57] We reduce the five max-pooling kernels sizes from 2x2 to three 2x1 and two 1x2. We do this to increase the width of the FC layer 0 of VGG16_bn (by 32×), to put us in a position where K-Factor computations in the FC layers are not negligible. Thus, we have FC layer 0: 16384-in×2048-out with dropout ($p = 0.5$), and the final FC layer: 2048-in×10-out. We define this as VGG16_bn_lmxp.

VGG16_bn: call it VGG16_bn_lmxp. All experiments ran on a single *NVIDIA Tesla V100-SXM2-16GB* GPU. The accuracy and loss we refer to are always on the *test set*.

For SENG, we used the *official github repo* implementation with the hyperparameters directly recommended by the authors for the problem at hand. We have: **(1) Repo:** https://github.com/yangorwell/SENG; **(2) Hyperparameters:** *label_smoothing = 0, fim_col_sample_size = 128, lr_scheme = 'exp', lr = 0.05, lr_decay_rate = 6, lr_decay_epoch = 75, damping = 2, weight_decay = 1e-2, momentum = 0.9, curvature_update_freq = 200. For the omitted parameters we used default values.*

For all algorithms except SENG, we use $\rho = 0.95$, $n_{\text{B}} = 256$ and $T_{\text{Cov}} = 25$, weight decay of $7e-04$, no momentum, a clip parameter of 0.07 and a learning rate schedule[58] of $\alpha_k = 0.3 - 0.1\mathbb{I}_{n_{ce}(k)\geq 2} - 0.1\mathbb{I}_{n_{ce}(k)\geq 3} - 0.07\mathbb{I}_{n_{ce}(k)\geq 13} - 0.02\mathbb{I}_{n_{ce}(k)\geq 18} - 0.007\mathbb{I}_{n_{ce}(k)\geq 27} - 0.002\mathbb{I}_{n_{ce}(k)\geq 40}$ (where $n_{\text{ce}}(k)$ is the number of the current epoch at iteration k). For all these algorithms, we set the regularization to be depend on layer, K-factor type ($\mathcal{A}$ vs Γ), and iteration as[59] $\lambda_{k,l}^{(\mathcal{M})} = \lambda_{\text{max},k,l}^{(\mathcal{M})}\phi_{\lambda,k}$ with the schedule $\phi_{\lambda,k} = 0.1 - 0.05\mathbb{I}_{n_{\text{ce}}(k)\geq 25} - 0.04\mathbb{I}_{n_{\text{ce}}(k)\geq 35}$.

For K-FAC and R-KFAC we set $T_{\text{inv}} = 250$. We also consider an R-KFAC which uses all the previous settings but $T_{\text{inv}} = 25$ - we refer to it "R-KFAC $T_{\text{inv}} = 25$". The hyperparameters[60] specific to R-KFAC were: $n_{\text{pwr-it}} = 4$, target-rank schedule $r(k) = 220 + 10\mathbb{I}_{n_{\text{ce}(k)}\geq 15}$, and oversampling parameter schedule $r_o(k) = 10 + \mathbb{I}_{n_{ce}(k)\geq 22} + \mathbb{I}_{n_{\text{ce}(k)}\geq 30}$.

For B-KFAC, BR-KFAC and BC-KFAC we set the *truncation rank schedule* to be the same as the target-rank schedule ($r(k)$) of R-KFAC. For B-KFAC we used $T_{\text{Brand}} = 125$. BC-KFAC had $T_{\text{Brand}} = 125$, $T_{\text{corct}} = 500$, and $\phi_{\text{crc}} = 0.5$. BR-KFAC had $T_{\text{Brand}} = 25$, and $T_{\text{BR}} = 250$.

Recall from *Section 4.1.6* that the implemented B-KFAC, BR-KFAC and BC-KFAC use the corresponding proposed updates only for the first FC layer, and R-KFAC updates (*Algorithm 3.1*; see *Chapter 3*) for all other layers. The hyperparameters of the R-KFAC part of B-KFAC, BR-KFAC or BC-KFAC are the ones we described for the R-KFAC algorithm.

[58] For all hyperparameters that were also present in *Chapter 3*, we used the schedules we arrived at in our tuning done in the experiments of *Section 3.3*.

[59] $\mathcal{M}$ can be $\mathcal{A}$ or Γ - see *lines 9 - 10* of *Algorithm 3.1*. $\lambda_{\text{max},k,l}^{(\mathcal{M})}$ is the maximum eigenvalue of our possibly approximate representation of the K-factor $\mathcal{M}$ at layer l, iteration k. In other words, we regularize by adding a fraction of the maximum eigenvalue of our K-factor representation at hand (be it the EVD, the RSVD or the B-updated representation).

[60] See *Chapter 3* or [81] for details.

Table 4.3: Optimizer Performance Results Summary. We perform 10 runs of 50 epochs for each considered solver. Columns 2-4 show times to a target test accuracy. Columns 5 shows the time per epoch. Column 6 shows how many runs get to 93.5% test accuracy. All solvers get to 93% accuracy 10 out of 10 times. The final column shows the number of epochs to 93% test accuracy. All results concerning times and number of epochs are shown as empirical mean ± empirical st. dev. The results for columns 2, 3, 7 use 10 samples (since 10/10 runs get to 93% test accuracy). The results in column 4 use only the runs which got to 93.5%. The results in column 5 use 500 samples. All times are in seconds.

	$t_{acc\geq 91\%}$	$t_{acc\geq 93\%}$	$t_{acc\geq 93.5\%}$	t_{epoch}	# Runs hit 93.5%	$\mathcal{N}_{\text{acc}\geq 93\%}$
SENG	999.0 ±54.1	1098 ±37.5	1144 ± 51	25.4 ± 0.81	10/10	43.1 ± 1.5
K-FAC	2610 ±213	4021 ±433	N/A	97.8 ± 8.2	0/10	41.1 ± 4.6
R-KFAC $T_{\text{inv}} = 250$	920.3 ±24.5	1357 ±38.7	N/A	47.7 ± 0.54	0/0	28.9 ± 0.8
R-KFAC $T_{\text{inv}} = 25$	1019 ±3.05	1526 ±49.3	2108 ± 373	53.3 ± 0.69	6/10	28.5 ± 0.9
B-KFAC	894.0 ±22.9	1325 ±58.9	1913 ± 125	46.3 ± 0.62	10/10	28.6 ± 1.2
BC-KFAC	911.2 ±18.9	1352.3 ±32.54	2031 ± 312	47.4 ± 0.45	6/10	28.5 ± 0.7
BR-KFAC	945.9 ±39.6	1324.8 ±58.9	1975 ± 323	48.4 ± 0.53	8/10	28.3 ± 0.5

4.5.1 Algorithms Optimization Performance Comparison

Table 4.3 shows a summary of results. We make the following observations:

Benchmark 0: SENG. Relatively large number of epochs to target accuracy but very low t_{epoch}, giving the best performance across all benchmarks (and all algorithms in general) for all target test accuracies apart from 91%.

Benchmark 1: K-FAC. The weakest of all considered algorithms with very large t_{epoch}, and surprisingly (unknown reason[61]), much larger number of epochs to target accuracy than any of its sped-up versions (all of which approximate the K-factors).

Benchmark[62] 2: R-KFAC. Moderate number of epochs to certain accuracy and moderate t_{epoch}, giving a performance always better than K-FAC (benchmark 1).

[61]There are many possible reasons. It *may* be due to slightly sub-optimal tuning, or due to the fact that using the same learning-rate for different algorithms may favor one over the other when the algorithms' raw step-sizes are substantially different (though this is probably not the case with our algorithms due to K-factor error control).

[62]An algorithm proposed in *Chapter 3*, but a benchmark for the algorithms proposed in *Chapter 4*

Nevertheless, it only outperforms SENG (benchmark 0) for 91% test accuracy.

B-KFAC: Has the lowest computational cost per epoch (t_{epoch}) across all KFAC-based algorithms, while taking nearly the same amount of epochs to a target test accuracy as the other KFAC-based algorithms. It outperforms K-FAC and R-KFAC (benchmarks 1 and 2) for all considered target accuracies, being *the best performing* B-$(\cdot)$ *variant*. It outperforms SENG only for 91% test accuracy.

BR-KFAC: essentially upgrades R-KFAC (with $T_{\text{inv}} = 250$) to also perform a B-update every time new K-factor information arrives. This improves $\mathcal{N}_{\text{acc}\geq 93\%}$ but makes t_{epoch} worse. Over-all it seems that the extra accuracy gained through introducing B-updates is favourable for large target test accuracy: BR-KFAC reaches 93.5% test accuracy 8 times, while R-KFAC ($T_{\text{inv}} = 250$) never does so.

BC-KFAC: Lies in between B-KFAC and BR-KFAC in terms of both t_{epoch} and $\mathcal{N}_{\text{acc}\geq 93\%}$, but provids a worse cost-accuracy trade-off than either of these in this case. Nevertheless, it outperforms R-KFAC and K-FAC benchmarks for all considered target test accuracies, and outperforms SENG for low target accuracy.

4.6 Discussion

By exploiting the EA construction paradigm of the K-factors, we proposed an online inverse-update to speed-up K-FAC ([1]. This update relied on Brand's algorithm ([74]), and we called it the "B-update" (of K-factors inverses). We saw it could be applied for K-factors which satisfy[63] $r + n_M < d_M$. If we use the update exclusively, we obtain the K-factor inverse representation in linear time (w.r.t. layer size d_M), as opposed to quadratic (or "almost quadratic" when controlling the error) for *Randomized* K-FACs in *Chapter 3* (or see [81]), and cubic for standard K-FAC [1, 29].

Furthermore, we also proposed an inverse application technique which also takes linear time in layer size (in *Section 4.3*). This worked whenever the B-update could be applied, and is a substantial improvement over the quadratic one we proposed in *Chapter 3*, or the cubic one which is for standard K-FAC (see *Algorithm 1.1* or [1, 29]). Thus, we proposed schemes which ensure that the all preconditioning computations can be performed in linear time (w.r.t. layer size) for all K-factors which satisfy $r + n_M < d_M$ (for our chosen/desired representation rank r). This is a substantial improvement considering the gradient computation is quadratic, the

[63] See *"A Caveat for B-update Usability"* just below *Algorithm 4.1* in *Section 4.1.1* to recall this.

R-KFAC preconditioning computations are also quadratic, and the ones of standard K-FAC are cubic.

An important consideration was that repeated B-updates applications, with no other operations, induce a rank growth in our K-factor representation. This rank growth would ultimately increase the cost of our computations all the way to a full EVD cost. To practically prevent this, we proposed performing an SVD (optimal) rank-r truncation of our B-updated K-factors representation just before each B-update. This ensured we get our desired linear time scaling of our preconditioning part.

Notably, introducing such a truncation comes, nevertheless, at the cost of errors. Thus, though the B-update (in isolation) is exact (recall *Section 2.3*), our B-KFAC algorithm has some error in representing the K-factors, due to repeated loss of information through specified truncation.

Unlike for the methods proposed in *Chapter 3*, we had no theory to inform us how to choose our target-rank r to control the error. While we know our rank-r B-KFAC representation surely has a worse error than the optimal rank-r truncation of $\bar{\mathcal{M}}_k$, which has been extensively studied in *Chapter 3*, we do not have bounds on the excess error (as we did for RSVD, see *Section 3.2.4*). Thus, we practically use the target-rank (r) values implied by our discussion in *Chapter 3*, and accept having an (a priori[64]) unknown (but in our hope low) excess error.

To aid with the control of this unknown excess error, we proposed a few algorithmic solutions. While these solutions were agnostic of the actual excess error in any way, numerical results in *Section 4.2.3* showed they work well in practice (at least for the case-study therein). In particular, our solutions were to change using B-KFAC (of *Section 4.1.1*) with using BR-KFAC (of *Section 4.1.3*), BC-KFAC (of *Section 4.1.4*), or BRC-KFAC (of *Section 4.1.5*). These improved B-KFAC versions helped with controlling the excess error as we have just mentioned, but at the expense of losing the low-memory property of B-KFAC (see *Section 4.4*) and getting an over-all (almost) quadratic time-complexity.

In addition to numerically seeing that BR-KFAC and BC-KFAC control the excess error better than B-KFAC, we also had some theoretical results which provided useful intuition in this respect. In *Section 4.1.2*, we looked at a 2D particular case in which the total error of the B-KFAC representation was above $4\times$ the optimal truncation error (for the same r) for multiple iterations, and the event happened with low but non-negligible probability. This acted as motivation for periodically resetting the B-KFAC representation from an RSVD, since for RSVD we know from *Section 3.2.3* that

[64]before the algorithm is run and errors are measured.

we should expect this factor to be no larger than 3×, while numerical results show it is more like 1.5× (see *Appendix A3.6.1*). For BC-KFAC, we had even better theoretical results. We showed that the proposed correction (the *C-update*[65]) is preferable to not performing the correction (in *Proposition 4.3* and *Corollary 4.4*) from the point of view of the iteration at which the correction it is performed. While the effect of the correction on subsequent iterations is yet to be investigated, our stated theory and numerical results in *Section 4.2.3* are enough to strongly recommend using the *C-update* to control the B-KFAC error.

Furthermore, we saw that BR-KFAC can also be seen as an R-KFAC algorithm to which we introduce B-updates at times when no RSVD would have been performed, rather than a corrected (or improved) B-KFAC. In *Section 4.2.1* we took this point of view on BR-KFAC and argued that (at least) when comparing the worst-case scenarios, using a B-update is better than using no update. This suggested that our observed worst-case representation error of BR-KFAC should be better than the one of R-KFAC (at least for sufficient run-time). Our numerical results in *Section 4.2.3* were indeed consistent with this observation (and stronger).

Numerical results concerning K-Factors errors (see *Section 4.2.3*) show that all our proposed algorithms (B-KFAC, BR-KFAC, and BC-KFAC) had errors comparable to K-FAC ([1]) while offering an $\approx 3\times$ speed-up per epoch (recall *Table 4.1*). W.r.t. the more competitive R-KFAC ([81]), our proposed algorithms offered similar metrics but more trade-offs to choose from. Notably, BR-KFAC was significantly better than R-KFAC - across all investigated error metrics at minimal computational overhead.

Finally, we tested the optimization performance of B-KFAC, BR-KFAC and BC-KFAC and compared it to R-KFAC (proposed in *Chapter 3*, now considered a benchmark), standard K-FAC (see [1, 29]), and SENG (the state of art; [20]).

Numerical results concerning optimization performance show B-KFAC and BC-KFAC consistently outperform R-KFAC (the best K-FAC benchmark; [81]) by a *moderate amount*, while BR-KFAC only does so for relatively large target test accuracy. All our B-(·) algorithms outperform SENG (see [20]) for low target test accuracy. We believe that the performance of our B-(·) variants would notably (but not excessively) improve by implementing the linear inverse-application technique proposed in *Section 4.3* - however observe that R-KFAC could also benefit from this implementation (while vanilla K-FAC could not).

[65]The "C" in BC-KFAC, see *Algorithm 4.3*.

4.6.1 Future Work

Future work involves:

1. Implementing the proposed inverse application methodology (of *Section 4.3*) and re-running numerical experiments;

2. Exploring where our cut-off threshold (in terms of d_M/n_M ratio) for applying B-update should be;

3. Obtaining theoretical error guarantees as a function of r in a B-KFAC algorithm, and use that to improve the choice of r for our B-updated K-factors - this would also aid the selection of which K-factors to be B-updated (helps point (2));

4. Improve the theoretical results in *Examples 4.1 - 4.2* (by generalizing the context) to better understand BR-KFAC and its error's relationship to B-KFAC error;

5. Improve on theoretical results of *Section 4.2.1* (*Propositions 4.5 - 4.6*) by obtaining a probabilistic (theoretical) comparison of R-KFAC and BR-KFAC error, thus improving our understanding about when to use one or the other;

6. Develop, implement, and theoretically investigate an RC-KFAC algorithm (using the R-updates and C-updates), as discussed at the end of *Section 4.1.5*;

7. Numerically investigate the performance of this to be proposed RC-KFAC.

In the *Future Directions Section of the Conclusions Chapter*, we discuss how future work relating to this chapter integrates with the general future directions of the thesis.

Chapter 5

Distributed Implementations of our Proposed K-FAC Variants

In this chapter, we propose and examine distributed implementations of our K-FAC improvements described in *Chapters 3* and *4*. The motivation behind this is two-fold.

Firstly, training Deep Learning models on very large datasets can take weeks when using a single GPU (as we did in the experiments of *Chapters 3* and *4*) [86]. Thus, we need distributed implementations of our proposed algorithms if these are ever to become used in practical, large-scale, Deep Learning training.

Secondly, it has been empirically observed in the literature that distributed K-FAC converges in fewer iterations[1] (epochs) than distributed SGD ([51, 71, 52, 53, 70, 73]). Since our proposed K-FAC improvements reduce the time complexity of K-FAC preconditioning overhead down to (or below) gradient computation complexity, we believe we can find at least one distributed implementation of our proposed K-FAC variants which can significantly outperform[2] SGD in large-scale distributed settings. This chapter represents a first attempt towards obtaining such a distributed implementation. The contributions of this chapters are as follows:

1. We provide distributed implementations of our proposed KFAC improvements (R-KFAC, B-KFAC, BR-KFAC, BRC-KFAC). Codes are available on github.
    - We use a novel mixture of previously proposed choices for distributed K-FAC implementation.
    - We provide a thorough discussion about the workloads specific to each proposed K-FAC improvement (and about their efficient allocation), in the context of distributed computing.

[1] An effect attributed to alleviating pathological curvature effect through preconditioning [87].
[2] And also perform better than current distributed implementations of standard K-FAC.

2. We test our distributed implementations on 1, 2 and 4 GPUs against two synchronous distributed benchmarks (SGD, and standard K-FAC[3]), on four datasets CIFAR10, CIFAR100, SVHN and Imagenette[4].

    - Relative to the benchmarks, we obtained speed-ups of 3× - 4× when using a slight modification of the VGG16_bn DNN (used for CIFAR10, CIFAR100, SVHN), and a 1.1× speed-up when using resnet50 (used for Imagenette).

3. We examine the scaling efficiency of our distributed K-FAC improvements[5].

In *Section 5.1* we pin down a particular choice of our distributed design before discussing any related work, or other choices. We do this because it substantially simplifies our discussion. We also describe the simplified (and approximate) way we will use to think about our K-factor computation workload[6]. In *Section 5.2* we give sufficient background to understand our remaining choices in the context of distributed computing. In *Section 5.3* we describe our distributed algorithms design in more detail, with a focus on the specifics of each proposed K-FAC improvement. In *Section 5.5* we describe the numerical experiments methodology. In *Section 5.6* we show summaries of the numerical results and comment on these. In *Section 5.7* we conclude and highlight future directions. In our discussion, the per-GPU gradient batch size will always be fixed (as the number of GPUs increases), as was found most efficient ([53]).

5.1 Our K-factor Computation Timing Choice

The choice of our K-factor computation timing is important for our future exposition. Different choices provide different workload profiles and require us to think differently when distributing our work. To simplify our exposition, we present this choice first, and only then present related work and the other relevant choice points through the lens of this fixed choice. We have two choices when it comes to computing K-factors:

1. **Compute K-factors during the forward and backward passes** ([88, 73]). That is, as soon as the relevant quantities become available.

2. **Cache the required quantities during the forward and backward passes, and compute K-factors later** ([52, 53]). This approach avoids computing

[3]Using the same distributed design as our proposed K-FAC improvements.

[4]Used Imagenette full-size v2 (a small subset of the large-scale data set benchmark ImageNet-1k).

[5]Though in the limited setting of 4 GPUs, which were the best resources available.

[6]This arises as a necessity to simplify the complexity induced by the interaction of our first choice (about K-factor computation timing) with the (most efficient) gradient communication design.

the K-factors during the forward and backward passes stage. Using the cached quantities, we can then perform the K-factor computation at any later time before inversion is required. The K-factor computation is usually done just after the gradient all-reduce is performed (as in [52, 53]; see *Figure 3* in [53]).

In this chapter, we chose to compute the K-factors during the forward and backward pass directly (*point 1* above). The only advantage[7] of our choice is having simpler implementation and a slightly lower memory footprint, while the other choice (point 2) allows for more flexibility in the workload profile, which may enable slightly better work allocation (thus potentially yielding better wall-time). Implementing our algorithms using the other choice is subject for future work.

5.1.1 Simplification: Mentally Re-Shuffling K-factor Workload

Because we focus on K-FAC preconditioning, we would like to mentally disentangle all the K-FAC related costs from the forward pass, backward pass and gradient communication costs.

In the naive gradient communication approach (the one in Figure 5.1(T.i), we can view it as if all the K-factor computation workload happens after the gradient computation for all layers, but before gradient communication (starting at the same time on all processes) - without changing the total time span for any workload distribution. Figure 5.1(T.ii)-(T.iii) illustrates this. This happens because all forward and backward pass workloads are identical on all processes (due to same batch size), start at the same time, and there is also no communication between processes until all the K-factors have been computed. This situation allows us to mentally[8] re-shuffle work as shown in Figure 5.1(T.iii) without changing the makespan[9]. We like this mental re-shuffling because it helps us understand how to efficiently distribute workload in a simpler manner, without ever having to think about gradient computation or communication. With this re-shuffling, we can leave the gradient related work outside our discussion, and only add a synchronization point at the end of the K-factor computation stage (due to gradient all-reduce).

[7] If we chose the *HP K-factor computation strategy* in *Section 5.2.1*, we could have additional communications costs saving with this choice (by using the pipelining approaches in [106] or [88]; see for example *Figure 1(b)* in [88]). However, we will not make the HP choice in *Section 5.2.1*, so we do not have this benefit.

[8] We pretend the workloads have a different order, but do not actually force this to happen.

[9] *Makespan* refers to the time taken to complete all tasks when they are scheduled in parallel across multiple resources. It is effectively the total time taken by the algorihtm to complete execution, as observed when putting a black-box to cover-up the internal workings.

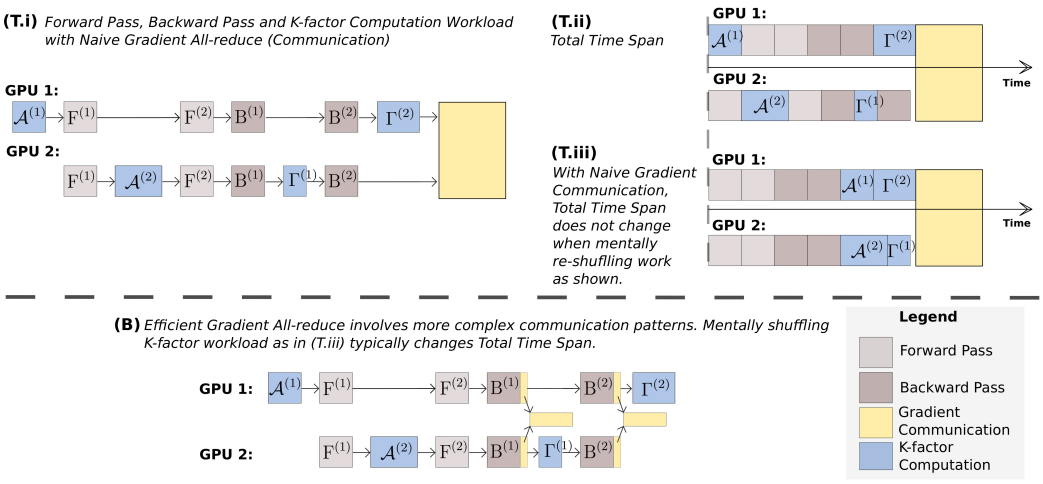


Figure 5.1: Mentally Re-shuffling K-factor work. An example with 2 GPUs and 2 layer networks is shown. Only layer indices are shown (as superscripts). Note that no K-factor computation work is duplicated as we draw the diagram under the LP K-factor computation choice, which is a choice we make later, in *Section 5.2.1*. In *(T.i)*, we are representing the computation flowchart in the performed order, for the naive gradient communication. Notably, the allocation of K-factors is arbitrary and meant to show that we are not forced to put $\Gamma^{(1)}$ and $\mathcal{A}^{(1)}$ workloads on the same GPU, for instance. Each GPU performs the forward and backward pass with a different batch, and computes the K-factors immediately once the required information becomes available. *(T.ii)* shows what the time-cost of workload looks like for each GPU, in the naive gradient communication implementation. Notably, any gap (see GPU 2) is an idle time. *(T.iii)* shows how reshuffling our workloads preserves the same over-all time-cost for each GPU (under the naive gradient communication paradigm). Finally, (B) shows the flowchart for efficient gradient-communication routines, and having the yellow workloads partly along the GPU1 and GPU2 lines and partly in between is meant to illustrate that part of the communication cost is parallelizable, but not all of it. With this communication paradigm, the mental reshuffling of workloads may change total expended time (and is thus a simplifying approximation).

However the naive implementation has long been superseeded in practice. There are many current proposals to perform all-reduce in ways that (partly) parallelize the communication and computation (see [86, 88, 90, 92, 93, 94, 97]).

With these efficient approaches (a simplistic illustration in Figure 5.1(B)), performing our mental shuffling presented here would change the time span[10], so it is only an approximate way of thinking. Designing an efficient work-load allocation for our proposed K-FAC improvements needs to account for the interaction between

[10]Under our choice to compute the K-factors during the forward and backward pass (review the beginning of *Section 5.1* and our choice of LP K-factor computation described later in Section 5.2.1).

the K-factors computation and gradient computation and communication (under our choice to compute the K-factors during the backprop phase). Since this depends on the gradient communication strategy used, and there are many ([86, 88, 90, 92, 93, 94, 97]), the simplest solution for future work might be to avoid this interaction by choosing to avoid the K-factors computation during the forward and backward passes and cache the required data instead (see *point 2* of *Section 5.1*).

For this chapter, we use the simplified way of thinking (as though we used naive gradient communication). However, this is not a big cause for concern for our work allocation strategies. We shall see that most allocation mechanisms we discuss will inherit the K-factor work allocation from an efficient K-factor inverse allocation, rather than being able to choose it - in which case our approximate simplification is irrelevant.

5.2 Related Distributed Implementations of K-FAC

Multiple ways to implement K-FAC in a distributed settings have been proposed in the literature (eg. [51, 52, 53, 70, 71, 73]). In this section, we condense the most successful ones into a single framework where two choices are made: (1) choices relating to K-Factors computation, and (2) choices relating to K-Factor inversion and step preconditioning. We present this framework through the lens of the K-factor computation timing choice specified in *Section 5.1*. Though most algorithms encapsulated by the framework were proposed in papers using *the other choice*, this does not impede the understanding of key ideas (but may modify workload structure).

The framework encapsulating these multiple algorithms is represented in *Figure 5.2* in its simplest form. It shows how K-FAC distributed computation may work. In particular, it illustrates where the two important choices are made, and presents two (simple) options for each point of choice. The gradient-related work is always done in a data-parallel fashion, as is standard ([53, 95, 96]). Other features of the distributed K-FAC algorithms also exist, but they are of secondary importance.

Different choices lead to different algorithms: some combination of choices may have not been proposed explicitly yet, though each choice in isolation has been proposed. Notably, some proposed algorithms make these choices implicitly (without acknowledging there is a choice to be made: eg. [51, 52, 71]).

The aim of this presentation is to justify the distributed implementation choices we will use for our proposed K-FAC improvements, as well as provide sufficient context to understand the workings of our distributed R-KFAC, B-KFAC, BR-KFAC, BRC-KFAC.

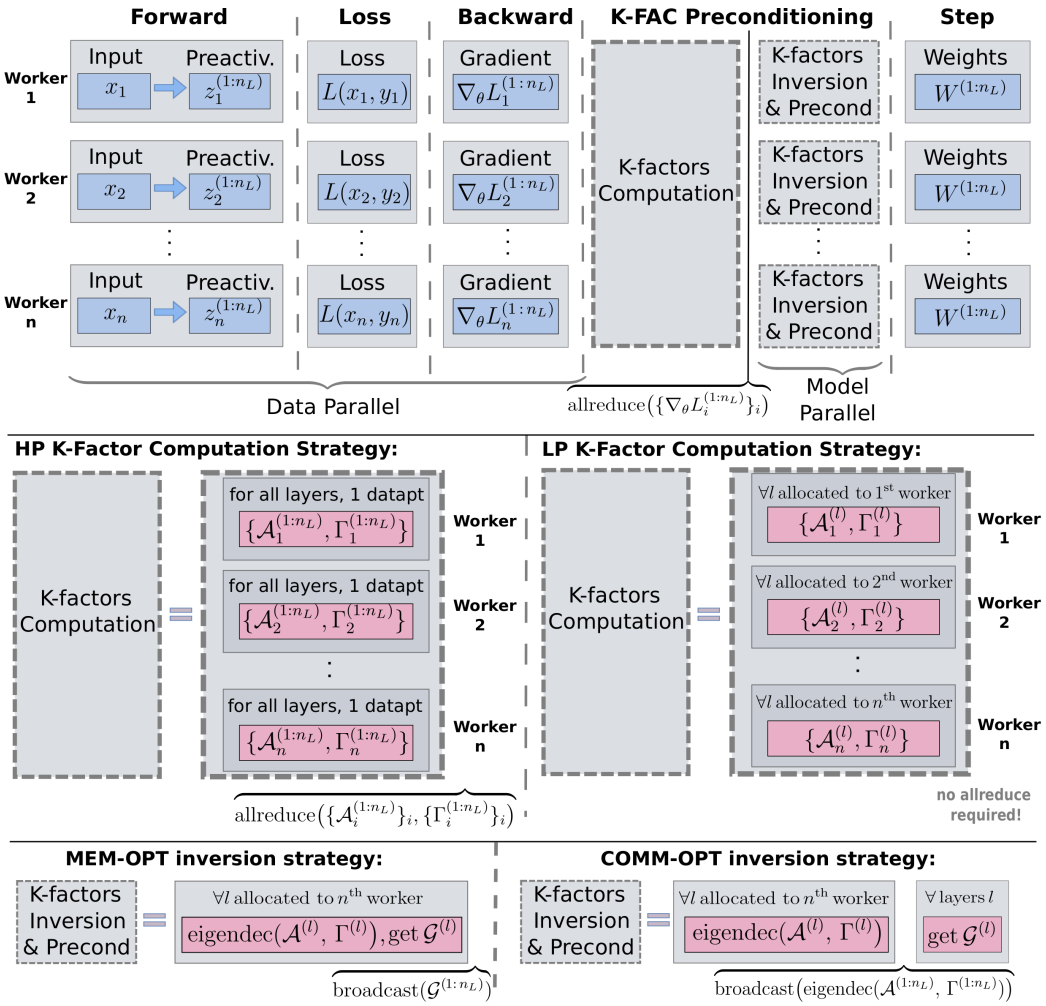


Figure 5.2: Summary of possible distributed implementation choices for K-FAC. Only the Decentralized Architecture is considered in this diagram. Subscripts represent indexing over different data points. Superscripts represent indexing over different layers. No step index is added. $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$ represent the all-reduced quantities over all data when HP is chosen for K-Factor computation strategy, but represent the corresponding $\mathcal{A}_i^{(l)}$ and $\Gamma_i^{(l)}$ when LP is chosen. $\mathcal{G}^{(l)}$ is the preconditioned gradient for layer l. Presentation idea inspired from [53].

The diagram is constructed based on K-FAC, which uses an eigendecomposition to obtain K-factors inverse representations. Our proposed K-FAC variants use different inverse-estimation techniques. However, we can simply think in terms of replacing the eigenvalue-decomposition (EVD) with a different inverse-estimation technique, for now.

Importantly, Figure 5.2 only encapsulates a decentralized architecture parallelization. To put this into context, we have the following two possibilities:

- **Centralized Architecture with Parameter Server (PS): reference paper [72], also see [73].** In a centralized parameter server (PS) architecture

(with no preconditioner), a PS is required to gather all gradients, update parameters, then push parameters back to the gradient workers [91, 98]. When K-FAC is distributed within this framework, additional dedicated K-Factors workers are added for the computation and eigendecomposition of K-factors. This work is performed independently of the gradient computation and preconditioning that happens on the gradient workers. The dedicated K-factor workers receive parameter updates from PS and push back eigenfactorizations (EVDs), which the PS uses to precondition the gradient it receives from the gradient workers to take the optimization step [72, 73].

- **Decentralized Architecture: reference papers [51, 52, 53, 70, 89], also see [73].** In a decentralized architecture, each worker computes the gradient (in a data-parallel fashion) and also does (at least some) work related to a K-Factor. Each worker holds a copy of the DNN, and all copies take the same optimization step. A revealing difference from the centralized architecture approach is that the data used to compute the gradient is also used on the same machine to assemble (at least some) K-Factors, and (some of) the eigendecompositions also happen on the machine.

In practice, decentralized architecture is much more widely used ([51, 52, 53, 73]), possibly because the centralized architecture requires more communication[11], and also has more stale[12] eigenfactors due to its decoupled gradient / preconditioner computation (asynchronous in nature). In general, having dedicated K-factor workers may only be worth these drawbacks if the eigen-factor computation is significantly slower than a gradient computation - that is when asynchronously assembling and inverting the K-Factors reduces the time of the main-thread significantly.

While this might be the case for K-FAC in most instances, it will not be the case for our proposed K-FAC improvements. That is because our proposed K-FAC improvements have brought the time-complexity of the preconditioner part to be of the same order or better than the complexity of obtaining the gradient (see *Chapters 3 - 4*). With such a low eigenfactors computation cost, paying the price of having

[11]For each gradient all-reduce there is also a reciprocal communication of parameters, which does not happen in a decentralized architecture. Whenever receiving EVDs of K-Factors, the PS also communicates back the parameters (which again do not happen on a decentralized architecture).

[12]By the time the K-Factor dedicated workers have finished computing the K-Factors and inverting them at parameter θ_k, the parameter server might be at $\theta_{k+\kappa}$, and then this preconditioner will be used for another T_{inv} iterations. Thus, our preconditioner is always between $[\kappa, \kappa + T_{\text{inv}}]$ iterations old in a centralized architecture. With a decentralized approach, the preconditioner would only be $[0, T_{\text{inv}}]$ iterations old. And κ might be fairly large when the K-Factors are large.

a more stale preconditioner and additional communication cost is not justified. For this reason, our proposed K-FAC improvements should be used with a decentralized architecture to best utilize their advantages over plain K-FAC. While our proposed K-FAC improvements could also be used with a centralized architecture parallelization paradigm, and it would likely outperform K-FAC within that architecture as well, we believe the improvement would be severely diminished.

In the next two subsections we discuss the two decisions that are necessarily made (sometimes implicitly) when implementing a decentralized architecture parallelization of K-FAC. We state our choice and the motivation behind it.

5.2.1 K-Factor Computation: Choice of Distributing Strategy

For the K-Factor inversion and preconditioning, we have 2 possible choices (see *Figure 5.2*). The first choice presented is the standard one in the literature, while the second choice is not well established. Using our own naming convention, the two choices are:

- **Heavy Preconditioning (HP): reference papers [51, 52, 53], also see [73].** For the HP approach, each worker (GPU) computes all the K-Factors for the data that it has. The multiple instances of the K-Factors are then aggregated across different data (processes) through an all-reduce operation. With this approach, adding more GPUs: (1) increases the total batch size of estimating the K-Factor (thus improving the K-Factors quality), (2) maintains the same per-GPU K-Factor computation cost[13], (3) increases the all-reduce K-Factors communication cost[14]. Overall, with HP, adding more GPUs results in increasing K-Factor estimate quality (due to more data being used), but at the expense of higher wall time of obtaining these K-Factors [73].
- **Lean Preconditioning (LP): reference paper [73].** In the LP approach each K-Factor is computed on exactly 1 process (GPU), unlike in the HP approach. The GPU on which each K-Factor is computed is the same as the GPU on which the eigendecomposition of that K-Factor is performed. This ensures no communication requirement exists due to mere computation of K-Factors (unlike in HP). With this set-up, increasing the number of GPUs: (1) maintains the quality[15] of the K-Factors, (2) the per-GPU K-Factor computation

[13]The same number of K-Factors with the same per-GPU batch size are computed at each GPU, irrespective of how many total GPUs there are.

[14]All-reduce takes linear or logarithmic time in #GPUs (depending on algorithm choice [90]).

[15]The total batch size per K-Factor is equal to the per-GPU gradient batch size, which is fixed.

cost reduces, until work cannot be further divided[16], (3) the communication cost associated with K-factor computation is always non-existent. Over-all, with an LP approach, adding more GPUs results in a constant K-Factor quality while reducing the associated time cost (until work cannot be further divided) [73].

Implementing distributed versions of our proposed K-FAC improvements (R, B, BR, and BRC) can be done with either of these choices. We choose the *Lean Preconditioning (LP)* approach, due to it being much more efficient (lower memory footprint[17], communication cost, and K-Factor computation overhead: see Figure 3 in [73] and discussion above). The drawback of having a lower total batch size estimate of K-Factors [73] does not seem to be a problem in practice (see [73]).

Furthermore, recall from *Section 3.1* that a lower K-factor (total) batch size allows us to have a lower target-rank in our proposed K-FAC improvements (for the same target truncation error). This means lower (total) K-factor batch size translates into faster estimation of K-Factor inverse representation and lower memory footprint for our proposed K-FAC improvements. Thus, choosing the LP approach unlocks further benefits for our proposed K-FAC improvements (which K-FAC does not benefit from).

Conversely, if we chose the HP approach, and insist on maintaining the target truncation error below a certain threshold (as we did so far), for *sufficiently many GPUs*, our target rank will reach[18] its maximal value (equal to K-Factor size, see *Chapter 3*). In this limit, we would need to compute all the modes, and our proposed K-FAC improvements would give no computational savings. Thus, our speed-up factor would converge to 1 from below as more GPUs are added - which is highly undesirable.

The best way to use the HP approach would be with a fixed target-rank (rather than a fixed target truncation error). From a cost scaling perspective, this would put us back in the same scaling scenario as with the LP approach and fixed target truncation error. However, the K-factor truncation error might get uncontrollably large, particularly because a larger total-batchsize generally creates a relatively weaker eigen-decay (aside from requiring many modes to accurately truncate). Thus, not only our truncation error would be large (due to targeting relatively few modes), but also the projection-error of RSVD (in an attempt to approximate these desired leading modes) would be large (review *Section 3.2.4*). Whether this error would be a practical

[16]That is because adding in more GPUs splits the same number of K-Factor computations ($2n_L$ K-Factors) to more GPUs.

[17]Only a subset of K-Factors are computed on each process, unlike in HP

[18]In general, though there may be particular cases when it would not. This would happen because a batch size greater or equal to the K-Factor size would in general not imply any decay in our EA-K-Factors: Review *Chapter 3* for details.

issue remains to be investigated as future work. In our thesis, we only investigate the LP approach, which appears from the outset as being the most convenient for our proposed K-FAC improvements.

5.2.2 K-Factor Inversion and Preconditioning Choice

For the K-Factor inversion and preconditioning, we have 3 possible choices[19] (only the first two are presented in Figure 5.2 for simplicity):

- **MEM-OPT: reference paper [71], also see [53].** MEM-OPT bundles the inversion (eigendecompositions) and preconditioning for each layer as a single chunk of work. It then distributes these chunks over the available workers [71]. The key feature of MEM-OPT (in comparison to COMM-OPT) is that the EVD of the K-factors remains only on the GPU (process) it was computed on - and it is not broadcasted to all processes as in COMM-OPT. Thus, each process can only compute a portion of the preconditioned step: $\mathcal{G}^{(l)}$ only for l allocated to the local process. To recover the full preconditioned step on all processes ($\{\mathcal{G}^{(l)}\}_{\forall l}$), we then need to broadcast the locally computed $\mathcal{G}^{(l)}$ to all processes. This approach minimizes the memory footprint, because no duplicate EVDs are ever stored. However, it obtains that by having a high communication cost (relative to COMM-OPT). Since K-Factors inverses are computed much more rarely than the preconditioned step, choosing to broadcast $\mathcal{G}^{(l)}$ rather than eigendec$(\mathcal{A}^{(l)}, \Gamma^{(l)})$ requires communication to happen much more frequently: at each step, rather than only when inverses are updated, as in COMM-OPT.

- **COMM-OPT: reference paper [52], also see [53].** In COMM-OPT, the eigendecomposition of each K-Factor forms a chunk of work, which is then distributed to available workers [52]. After each process (GPU) computes its allocated eigendecompositions, these are broadcast to all other processes, such that all processes end up storing all the EVDs. The step $\{\mathcal{G}^{(l)}\}_{\forall l}$ is now computed locally in full on each worker, as they all have the required eigendecompositions stored. In relation to MEM-OPT, COMM-OPT minimizes communication cost because it only needs to broadcast EVDs, not preconditioned steps. Since the eigendecompositions only need to be updated rarely (about once in 100 steps [1, 53]), much less communication is required. However, since all processes store all the eigen-factorizations, COMM-OPT utilizes much more memory. COMM-OPT is superior to MEM-OPT whenever sufficient memory is available.

[19] Choice names first coined in paper [53], but methodologies existed before under different names.

- **HYBRID-OPT: reference paper [53].** HYBRID-OPT is a parameter-based trade-off between MEM-OPT and COMM-OPT (see [53] for details). We do not present it here to avoid adding unnecessary complexity for an otherwise simple intuition. It is sufficient for us to mention that MEM-OPT and COMM-OPT are the two extreme cases of HYBRID-OPT [53].

Implementing distributed versions of our proposed K-FAC improvements (R, B, BR, and BRC) can be done with either of these three choices. We choose COMM-OPT because we want to minimize wall-time expenditure (through minimizing communication cost). The motivation for this is two-fold. First, our proposed algorithms have relatively low memory footprint (our approximate <low-rank> K-Factor inverse representations require less memory than K-FAC's[20] full eigendecompositions). Secondly, COMM-OPT should always be the preferred choice assuming our GPUs have enough memory[21].

At this point, it is important to notice how our proposed K-FAC improvements may provide further speed-ups[22] in limited memory situations: by using less memory, our K-FAC improvements are able to use COMM-OPT (or a HYBRID-OPT parameter closer to COMM-OPT) when standard K-FAC may be forced to use MEM-OPT (or a HYBRID-OPT parameter closer to MEM-OPT). The B-KFAC improvement which has the smallest memory footprint is of particular relevance here.

5.3 Our Distributed K-FACs Design

The distributed framework we use is mainly based on choosing from work that has already been done before. Based on our discussion in *Section 5.2* we choose to use a **decentralized architecture** with **LP** K-Factor computation (see [73]) and **COMM-OPT** K-Factor inversion and preconditioning (see [52, 53]). Our **K-factor computation timing** is chosen to be **during the forward and backward passes**, as specified in *Section 5.1* (immediately after the required quantities become available,

[20] All our proposed algorithms reduce the Eigenfactors storage from $\mathcal{O}(d_M^2)$ to $\mathcal{O}(d_M r)$ where r is the target rank (see Chapters 3 and 4). Furthermore, the B-KFAC approximations require substantially lower memory footprint since there is also no need for K-Factors storage at all - see Chapter 4).

[21] With an ever-increasing GPU memory, lately reaching up to 80GB for NVIDIA A-100, the memory should not be an issue in most cases, unless enforced through budget constraints.

[22] Further speed-ups on top of the ones we have seen so far, and the ones we will see. All our presented experiments ensured there was sufficient memory for K-FAC to be implemented with COMM-OPT. When there is a memory limitation and K-FAC cannot be implemented with COMM-OPT (but rather is implemented with HYBRID-OPT or MEM-OPT), our proposed algorithms, using much less memory, will be able to use a HYBRID-OPT parameter which is much closer to COMM-OPT, thus obtaining further speed-up from this. An edge-case of this is when having only 1 GPU with insufficient memory to run K-FAC, but with sufficient memory to run R-KFAC or B-KFAC.

no caching). We use these distributed design choices for all K-FAC based algorithms (both benchmark and proposed improvements).

Strictly speaking, our proposed combination of choices is novel, because the paper which proposes the LP K-Factor computation approach ([73]) uses it with MEM-OPT, and the paper which proposes COMM-OPT uses it with HP K-factor computation approach. Still, while the mix of ideas is novel, each idea was proposed before.

To obtain distributed versions of our K-FAC improvements, one has to further consider some algorithm-specific details. We also have to specify a work-allocation mechanism. The remainder of *Section 5.3* together with *Section 5.4* provide these details. *Sections 5.5* and *5.6* present numerical results. We then conclude this chapter in *Section 5.7*.

5.3.1 Efficient Allocation of K-Factors Workload

Similarly to KAISA ([53]), we aim to distribute our K-Factor work in a fashion that ensures we minimize the makespan T (the time it takes for all parallel processes to finish computation) of K-factor inverse update. To obtain this, we use the longest processing time algorithm (as in KAISA [53]), which ensures our obtained makespan is $\hat{T} \leq (3/2)T^*$ (where T^* is the optimal makespan) [99]. This algorithm is a greedy approach which sorts the required time for each chunk of work descendingly, and then always allocates the largest remaining chunk to the worker with smallest current workload, until full allocation [53]. Note that because we chose *LP Distributed Strategy*, we can compute the EVDs of $\mathcal{A}^{(i)}$ and $\Gamma^{(i)}$ on separate processes. Thus, we have $2n_L$ individual chunks of work to be split across our GPUs.

In order to practically perform this work allocation, we first need to estimate the time required to finish an inverse update. We know the complexity of each inverse-update we propose (also of standard K-FAC's EVD) and the size of the K-factors. So we could be tempted to estimate the computation time based on these two pieces of information. However, some simple numerical experiments show that for the size of the K-Factors that are typically encountered in practice (up to 20,000 $\times$ 20,000), merely using the asymptotic behavior complexity is not accurate enough. This asymptotic regime is not reached, and so, assuming it does produces highly suboptimal allocations. Furthermore, different target-ranks reach this asymptotic regime for different sizes, and also have different scaling behavior in the non-asymptotic regime. Figure 5.3 illustrates these points. Similarly, changing the GPU type will alter the curves.

For this reason, we believe it is best to take a measurement-based approach. We could either measure the times during the optimization run, or measure beforehand

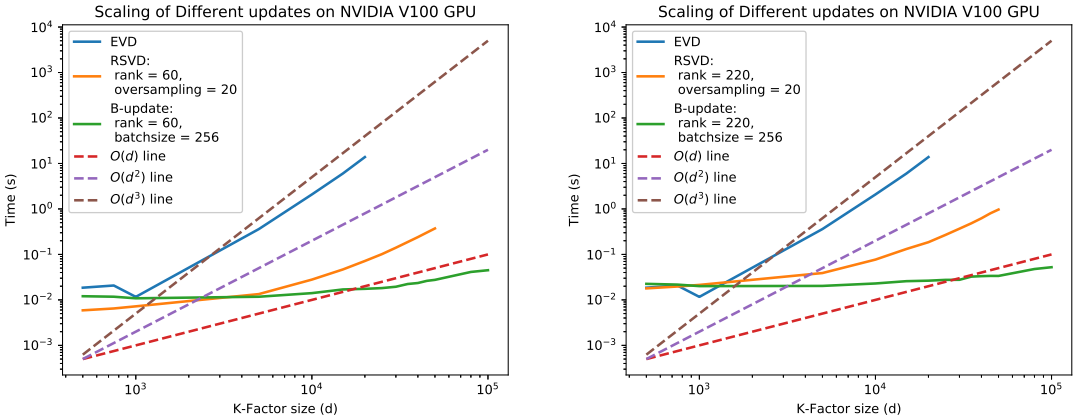


Figure 5.3: EVD, RSVD and B-update: Scaling comparison. The figure shows that the asymptotic regimes of $\mathcal{O}(d)$, $\mathcal{O}(d^2)$ and $\mathcal{O}(d^3)$ for B-update, RSVD and EVD respectively are not reached until a large K-Factor size. This suggests using asymptotic scaling behavior to predict inverse-update time yields sub-optimal work allocations. Each data curve is an average over 7 different measurements.

(possibly many more times to denoise), then fit some curves of known shape (implied by complexity) to the data, and then using these parameterized curves at run-time.

We prefer measuring before optimization run-time to avoid overheads and reduce estimation error. For each proposed inverse update and our given GPU type, we fix a target-rank, and then measure the computation time for increasing K-Factor size. We use the known complexity of our update to choose the polynomial order we should be fitting to this data. We regress on the data to obtain the time vs K-Factor size curve for a specific inverse-update and a specific target-rank. We repeat the experiment for multiple target-ranks. Thus, at run-time we can estimate the inverse-update computation time for any target-rank, using interpolation for unseen target-ranks.

Using these time estimates, we then allocate work when the algorithm starts. For our proposed algorithms, we have three inverse-updates: RSVD, B-update and the correction introduced in *Algorithm 4.3* of *Chapter 4*. We perform the experiments to get the curves for RSVD and B-update, but ignore the correction in *Algorithm 4.3* (setting its estimated times to zero, for simplicity). We also use this allocation strategy (fitting curves to predict EVD time based on size) for standard K-FAC for a fair algorithm comparison.

5.3.2 Adaptive Target-Rank Strategies

Unlike in the previous chapters, all our distributed solvers now use an adaptive target-rank strategy[23]. The idea is to obtain an almost fixed relative truncation error for each K-Factor, rather than setting a fixed target-rank.

The adaptive strategy computes the ratio[24] between the minimum and maximum available eigenvalues (in our low-rank K-factor representation) each time the inverse is updated. It then increases the target-rank if our ratio is (significantly) above our error threshold, or reduces it if our error is (significantly) below the error threshold. The scheme applies to each K-factor individually, so in principle each K-Factor could have a different target-rank at a given time. Note that with the adaptive heuristic, we have to set a maximum rank that cannot be exceeded. We need this to ensure our K-FAC improvements do not revert back to doing standard K-FAC by computing all the modes for all the K-factors (and thus get no computation savings). In practice we used a maximum allowable target rank of 700 (and this rank was never used).

Our simple heuristic[25] described here is shown in *Algorithm 5.1* with further details. We use it for the results in this chapter. More complicated heuristics can be designed (subject for future work), but the one given here provided significant improvement.

Algorithm 5.1: Simple Adaptive Target-Rank Strategy

Parameters: target error proxy $\epsilon \in (0,1)$; maximum and minimum allowable ranks: r_M, r_m; rank-increase tolerance ϵ_I; rank-decreasing tolerance ϵ_D; rank initial guess r_0 to initialize current_target_rank$_{\mathcal{K}}$ to $(\forall \mathcal{K})$

for *For each K-factor* $\bar{\mathcal{K}} \in \{\bar{\mathcal{A}}_k^{(l)}\}_{l=1}^{n_L} \cup \{\bar{\Gamma}_k^{(l)}\}_{l=1}^{n_L}$ **do**

  let its low-rank representation be $\tilde{U}_{\mathcal{K}}\tilde{D}_{\mathcal{K}}\tilde{U}_{\mathcal{K}}^T \approx \bar{\mathcal{K}}$ `// Obtained with either R-update, B-update, BR, or BRC update schemes.`

  if $\tilde{U}_{\mathcal{K}}\tilde{D}_{\mathcal{K}}\tilde{U}_{\mathcal{K}}^T$ *was updated at iter.* k **then** `// Adjust target rank for` $\bar{\mathcal{K}}$

    $\rho = \min(\tilde{D}_{\mathcal{K}}) / \max(\tilde{D}_{\mathcal{K}})$

    current_target_rank$_{\mathcal{K}}$ += $+10\mathbb{I}_{\{\rho > \epsilon + \epsilon_I\}} - 10\mathbb{I}_{\{\rho < \epsilon - \epsilon_D\}}$

    current_target_rank$_{\mathcal{K}}$ = $\max(\min(\text{current_target_rank}_{\mathcal{K}}, r_M), r_m)$

[23] This feature is not relevant to standard K-FAC as EVD has no target-rank.

[24] Which is a proxy for our relative truncation error(in the 2-norm).

[25] For simplicity, we chose additive increments to target rank (see *Algorithm 5.1*). Multiplicative ones are also possible and these would help find the order of magnitude faster, which may be more desirable. Though slower, additive increments are better for higher precision in picking the rank. Notably however, this benefit of additive updates may disappear in a situation where the target-rank for a fixed value of ϵ is non-stationary - due to slow convergence to "correct rank" of additive updates. A better approach might be a mix of multiplicative and additive updates, where we use the former when a larger change should be done as implied by our error proxy, and the latter when a smaller target-rank change should be done. Defining "large" and "small" then becomes a heuristic to explore, and we leave this as future work.

Note that by changing the target-rank as the optimization progresses, the optimal K-Factor work allocation changes (see *Section 5.3.1*), so in principle we should also reallocate work when target-ranks are adjusted. However, in practice the changes are minor if a we have a reasonable initial guess for the target-rank. In addition, using the longest processing time allocation does not exactly yield the optimal allocation to begin with (review *Section 5.3.1*), and the actual computation times are also scattered around the predicted time. With these considerations, we found it was best to avoid the overhead of reallocating work - though this might change if target-ranks change more significantly in other instances. In practice we used $r_M = 700$, $r_m = 10$, $\epsilon = 0.033$, $\epsilon_I = 0.02$, $\epsilon_D = 0.007$ for all our proposed K-FAC improvements.

5.4 Optimizer Specific Details of Distributed Design

All our proposed optimizers use the rank-continuation technique proposed in *Chapter 4*. For our K-FAC improvements, the identity initialization is dealt with in the regularization term as described in *Section 3.1.1*. This ensures a better outcome for our B and R updates. The step-clip strategies and regularization are as in the previous chapters (see *Chapters 3* and *4*). In *Sections 5.4.1 - 5.4.5*, we present some distributed design details specific to each solver. Exact numerical values of hyperparameters and hyperparameter tuning methodology will be specified in *Section 5.5*.

5.4.1 Benchmark Algorithms: Distributed Design Details

Distributed SGD (benchmark). Our distributed SGD is a decentralized, data-parallel, synchronous implementation (see for example [86, 92, 100, 101, 102, 105]). This implementation has been the most successful (more successful than asynchronous one, see [86, 98, 103]), being able to train ImageNet-1k with resnet50 to about 75% test accuracy in 1 hour ([100]), four minutes ([101]), or even 74s ([105]). Our exact data-parallel synchronous SGD implementation can be obtained by removing the K-FAC preconditioning part in *Figure 5.2*.

Distributed K-FAC (benchmark). Our distributed K-FAC implementation is fully specified through the choices and additions described so far. Note that K-FAC has no target-rank (since it performs a full EVD), so *Section 5.3.2* is irrelevant to it.

5.4.2 R-KFAC: Distributed Design Details

Distributed R-KFAC is fully specified through the choices and additions stated so far.

5.4.3 B-KFAC: Distributed Design Details

As in *Chapter 4*, the B-KFAC algorithm uses the B-update only for K-Factors of FC layers whose size is above $n_{\text{BS/GPU}} + r_{\text{target, max}}$ (where $r_{\text{target, max}}$ is the maximum allowable target-rank in our adaptive-rank heuristic, and $n_{\text{BS/GPU}}$ is the batch size per GPU). The other K-Factors use the RSVD inverse update.

Importantly, for the B-updated K-Factors, the K-FAC preconditioning stage in *Figure 5.2* is not made of two sub-stages as for the RSVD / (EVD updated layers (as was depicted in *Figure 5.2*). Instead, the K-Factor computation stage is missing: we only have a direct update to the low-rank K-Factors representation which replaces the inverse estimation. This inverse update is done with our B-update. The B-update requires the raw matrices ($A_k^{(l)}$ and $G_k^{(l)}$ in *Algorithm 1.1*) used to put together the K-Factors, which are computed through forward and backward hooks immediately after the respective passes. To avoid storing these matrices and save memory, we choose to perform our B-update immediately after computing the raw matrices $A_k^{(l)}$ and $G_k^{(l)}$. That is, if we have both B-updated and R-updated layers[26], the B-update will happen during the K-Factor computation stage of the R-updated layers[27]. The computational work performed for an R-updated layer is qualitatively the same as standard K-FAC approach presented in *Figure 5.2* (but with lower inverse-computation times). The left hand side (L) of *Figure 5.4* illustrates all the workloads and their timings for B-updated and R-updated layers.

The right part (R) of *Figure 5.4* illustrates how we would most optimally allocate work when both B-updated and R-updated K-factors are present - when constraining ourselves to use the simple greedy algorithm: longest processing time algorithm (which we do s.t. we get a good allocation solution in little time). First, we would greedily allocate all R-updated K-Factors in proportion to their RSVD cost, as we did with R-KFAC (see R.i)). This allocation would then imply some fixed work distribution for the R-updated layers during the K-Factor computation stage (see R.ii). Then, starting from this current work allocation, we would use longest processing algorithm to greedily allocate the remaining B-work (see R.iii). However, in practice the time spent during the inversion stage is much larger than the time spent in the K-Factor computation stage. In this case we can obtain very similar total makespan by ignoring the K-factor computation costs when allocating B-update work. That is, we allocate

[26]Which will always be the case unless the Neural Net is only made of wide fully connected layers.

[27]Note that we did not specify this order when parallelism was not used in *Chapter 4* because order is irrelevant for sequential operations. However, an operations order choice influences how efficiently resources are allocated in a distributed setting.

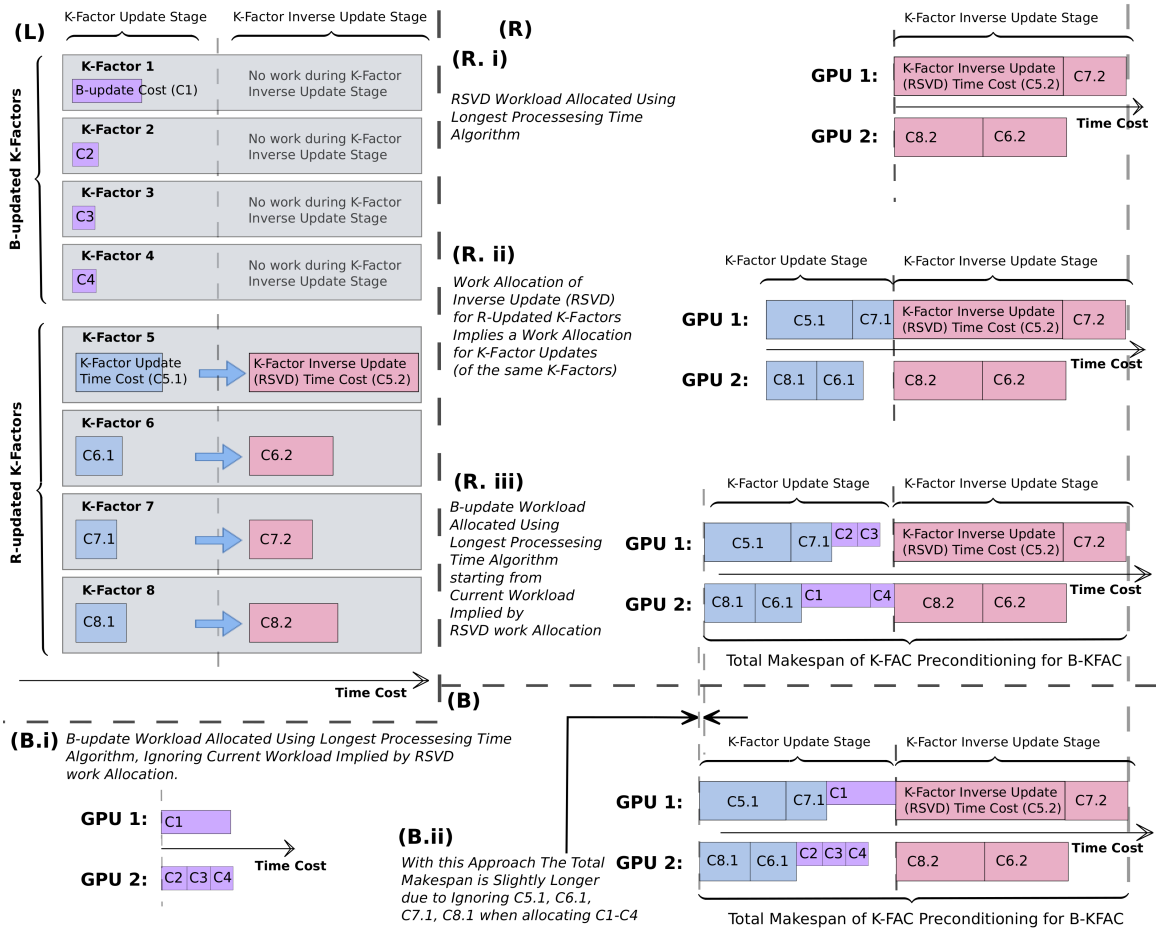


Figure 5.4: Distributed B-KFAC work allocation. The left side (L) of the figure shows the work for each K-Factor. Some K-Factors will be on the B-update track (large FC layers), while some will be on the R-update track (Conv or small FC layers). The gray boxes indicate that all the chunks of work within must go to the same process (GPU), though they happen at different stages. This constraint comes from our choice to use LP K-factor computation strategy. The right side (R) shows how we would optimally allocate the raw workload presented in (L) if we constrain ourselves to only use the longest processing time (greedy) algorithm. Two identical GPUs are assumed to be available. The dashed vertical line between the K-Factor Update Stage and K-Factor Inverse Update Stage marks a synchronization point, where all processes must wait for each other: due to all-reducing the gradient. The bottom side (B) of the figure shows what may happen if we ignore the current work allocation in the K-Factor Update Stage implied by the RSVD work allocation. With this approach, we get a worse (or the same) allocation as in R.iii) because we are unaware of any "gaps" in blue workload allocation. Still, total makespan difference is minimal in practice, as the main cost is in the K-Factor Inverse Update stage.

R-update and B-update work independently using the greedy allocation algorithm. The bottom part (B) of *Figure 5.4* illustrates this.

In *Figure 5.4*, note that the workloads of one or more color-codes might all disappear on some iterations - and this depends on our hyperparameter choice. For example we perform the purple workloads (once in T_{Brand} steps) less frequently than the blue ones (once in T_{Cov} steps) in principle, and the red ones even less frequent than that (once in T_{inv} steps). *Figure 5.5* illustrates this idea by showing us more detail about how the K-FAC preconditioning part of our distributed B-KFAC algorithm works in practice. *Figure 5.5* is constructed based on the work-loads from *Figure 5.4*.

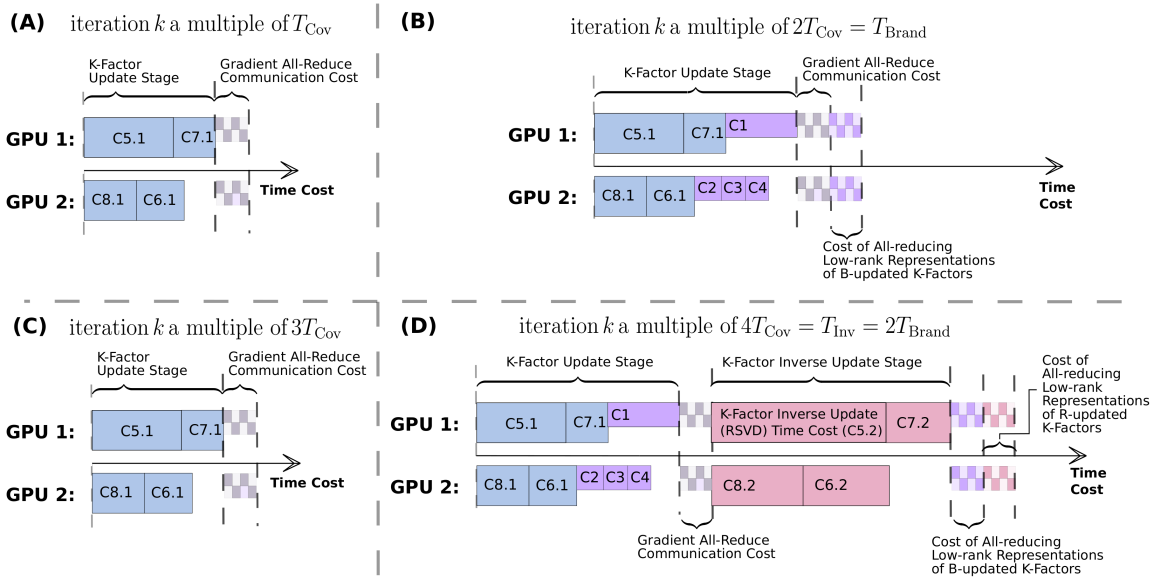


Figure 5.5: Distributed B-KFAC algorithm workings at different iteration indices k. We assumed $T_{\text{Brand}} = 2T_{\text{Cov}}$ and $T_{\text{Inv}} = 4T_{\text{Cov}}$ to illustrate this point, but any integer multiples of T_{Cov} can be chosen in practice. The Gradient computation work is not shown. The preconditioning of the gradient given the K-Factors inverse representations is also not shown. We omit these two as they happen at every iteration. Iterations which are not a multiple of any of our important $T's$ (T_{Brand}, T_{Cov}, T_{inv}) only perform this omitted work and the Gradient all-reduce communication (shown in textured in the figure).

Future Directions for B-update work allocation. Different work-allocation approaches are possible, and our proposed one here can be improved. Exploring these is subject for future work. Our chapter here is merely providing a reasonable distributed implementation, not the best. For future work, we believe that storing the matrices $A_k^{(l)}$ and $G_k^{(l)}$, choosing to perform the B-update during the R-update stage, and optimally allocating the B-update and R-update workload together (since they happen at the same stage under this paradigm) is probably the best approach in terms of minimizing average time per-step. That is because the higher complexity R-updates tend to give more imbalanced work allocations (even when optimal). The

gaps of the imbalanced work allocation can easily be filled by cheap updates with little or no makespan increase. While this approach may generally improve makespan, it will also increase memory usage. Furthermore, the makespan improvement may not be significant since B-updates are cheap, so we need further investigation as part of future work to take an informed decision.

5.4.4 BR-KFAC: Distributed Design Details

A distributed BR-KFAC algorithm is like a distributed B-KFAC algorithm where we sometimes reset our B-updated K-factors representations by using an RSVD. That is, the B-updated K-factors also get (rare) R-updates to the inverse representation (and are now BR-updated K-factors). These R-updates increase the memory and computational requirements because now we need to store and compute the BR-updated K-Factors, rather than directly carrying the inverse representation as with B-KFAC (review *Chapter 4* for all these details). For the BR-updated K-factors, we can distribute the B-update and the R-update work of the same K-factor to different processes without incurring any additional communication cost. *Figure 5.6(L)* illustrates the workloads that have to be allocated for a distributed BR-KFAC algorithm. Only the costs related to K-FAC preconditioning are shown. We omit the actual preconditioner application, gradient computation, and all communication costs.

These computation costs are shown in a raw format on the (L) side of *Figure 5.6*. To obtain a functional algorithm, we need to have a workload allocation mechanism. *Figure 5.6(R)* illustrates the mechanism we used here. To allocate the BR-KFAC workload, we first allocate all the workload that was also present in B-KFAC (B-updates of BR-updated K-factors and R-updates of R-updated K-Factors) in the same way as we did for B-KFAC (see *Figure 5.6(R.i))*. Then, we are left to allocate the K-Factor update and RSVD workloads of BR-updated K-Factors, which must go together for each K-factor. We allocate this work by using the greedy longest processing time algorithm on the RSVD workload, starting from the already allocated RSVD workload of the R-updated K-Factors. This procedure is show in *Figure 5.6(R.ii)*. The work allocation is now fully determined through the gray-box constraints: which imply a work allocation of the K-Factors updates of the BR-updated K-factors, shown in *Figure 5.6(R.iii)*.

As with distributed B-KFAC, not all iterations have all the workload shown in *Figure 5.6.(R.iii)*. In fact, with BR-KFAC performing a B-update and an R-update at the same iterations for the same K-factors is wasteful, because the R-update overwrites all previous B-updates. Thus, in practice there is no iteration containing all the

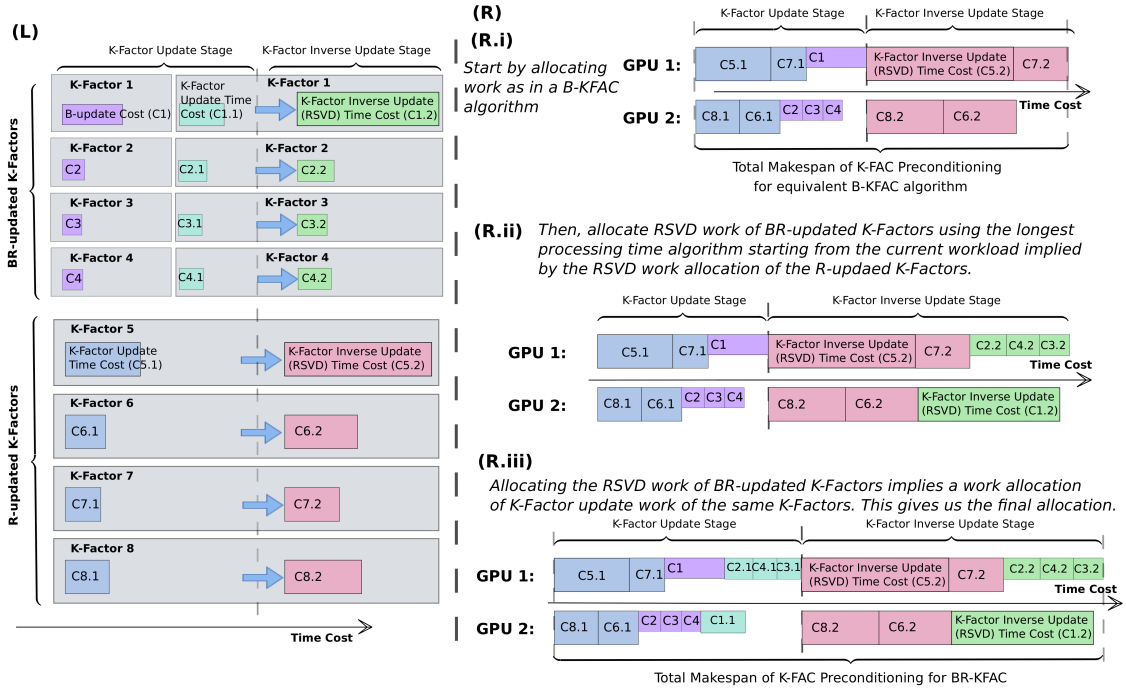


Figure 5.6: Distributed BR-KFAC work allocation. The left side (L) of the figure shows the work for each K-Factor. Some K-Factors will be on the BR-update track (large FC layers), while some will be on the R-update track (Conv or small FC layers). As before, the gray boxes indicate that all the chunks of work within must go to the same process (GPU). This constraint comes from our choice to use LP K-factor computation strategy. Note that for BR-updated K-Factors we have two distinct gray chunks of work. The right side (R) demonstrates how we allocate work in our BR algorithm. The dashed vertical line between the K-Factor Update Stage and K-Factor Inverse Update Stage marks a synchronization point, where all processes must wait for each other: due to all-reducing the gradient.

workloads shown in the allocation[28] of *Figure 5.6.(R.iii)*. *Figure 5.7* shows what a typical BR-KFAC algorithm would function as. It shows all possible workload patterns that might be incurred at an iteration (again, work present at all iterations is omitted). In practice, we can get fewer of these cases, by making specific choices of our T's. For the purpose of this chapter, we chose $T_{\text{Brand}} = T_{\text{Cov}}$ and T_{BR} an integer multiple of T_{inv} (in this case we only encounter cases (B), (E) and (F)). Having fewer cases to deal with allowed us to better tune our work allocation to minimize idling time. Our proposed work allocation discussed before was designed with this choice in mind.

As with distributed B-KFAC, our proposed work allocation can be improved, and there are more possible allocation strategies for distributed BR-KFAC due to it having more workload types. Obtaining the best possible allocation is left as future work.

[28]Our work allocation mechanism accounted for this.

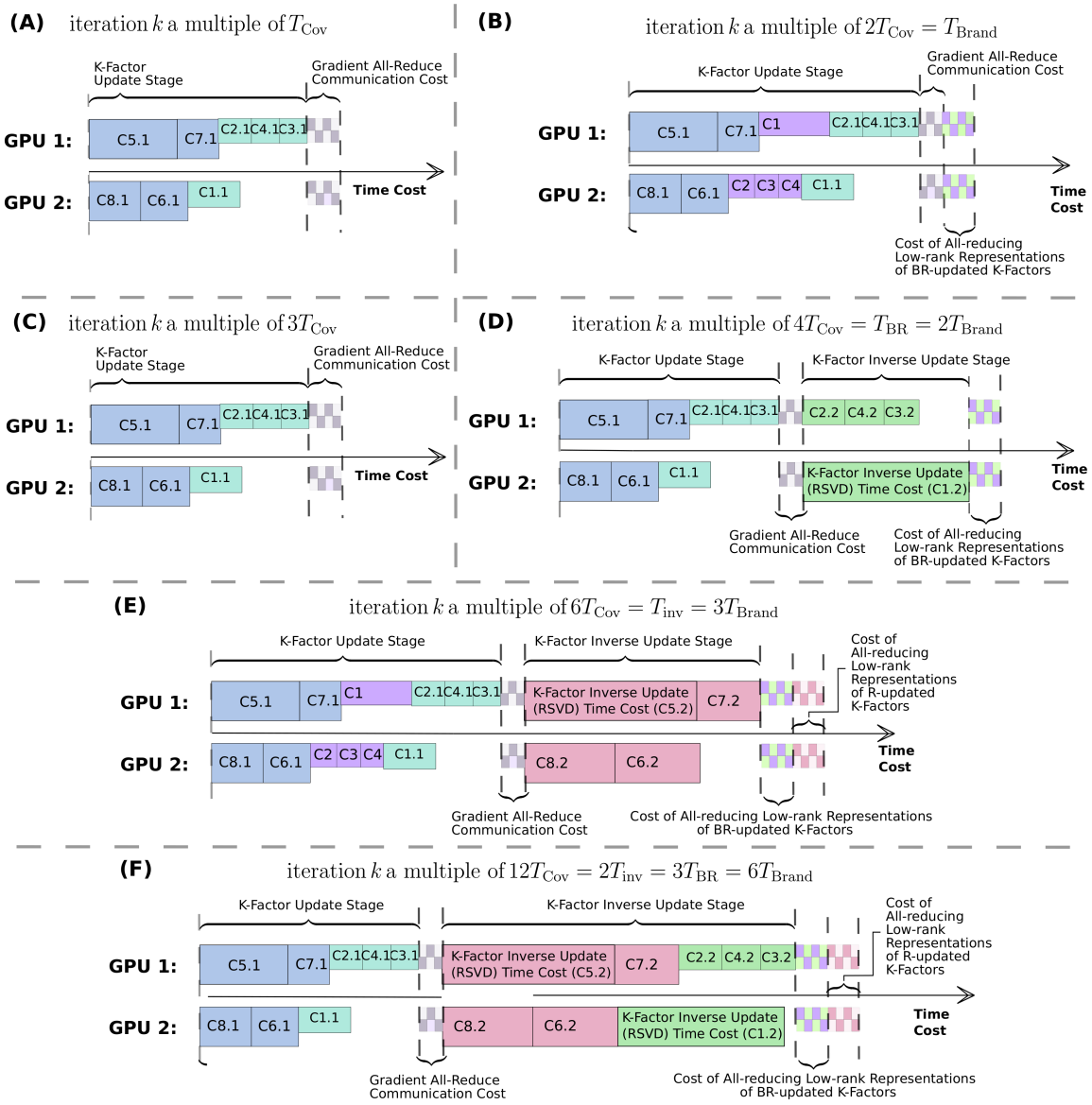


Figure 5.7: Distributed BR-KFAC algorithm workings at different iteration indices k. We assumed $T_{\text{Brand}} = 2T_{\text{Cov}}$, $T_{\text{BR}} = 4T_{\text{Cov}}$, and $T_{\text{inv}} = 6T_{\text{Cov}}$ to illustrate this point, but any integer multiples of T_{Cov} can be chosen in practice. The preconditioning of the gradient given the K-Factors inverse representations and the gradient computation are not shown, as they happen for all k. Iterations which are not a multiple of any of our important $T's$ (T_{Brand}, T_{Cov}, T_{BR}, T_{inv}) only perform this omitted work and the gradient all-reduce. Note that at k a multiple of $12T_{\text{Cov}} = 2T_{\text{inv}} = 3T_{\text{BR}} = 6T_{\text{Brand}}$, we do not have both a B-update and an R-update of the BR-updated K-factors.

5.4.5 BRC-KFAC: Distributed Design Details

Recall from *Chapter 4* (*Section 4.1.5*) that the BRC-KFAC algorithm is obtained from BR-KFAC by sometimes replacing the R-update of a BR-updated K-Factors with a

cheaper but less accurate correction proposed in Algorithm 4.3 of *Chapter 4* (the C-update). To produce the equivalent of *Figures 5.6* and *5.7* for BRC-KFAC, one has to consider that the workloads *C1.2*, *C2.2*, *C3.2* and *C4.2* therein are sometimes the R-update cost (as represented there) but sometimes the cheaper correction cost. Thus, the equivalent of *Figures 5.6* and *5.7* for BRC-KFAC are yet more complex (skipped here), and so is designing a (nearly) optimal work allocation strategy that accounts for this correction feature of BRC-KFAC. Note the correction in *Algorithm 4.3*, unlike an R-update (of the BRC-updated K-factors), does not fully over-write the previous B-updates, and so, is allowed to happen at the same iteration with a B-update.

In this chapter we completely ignored the difference between BRC-KFAC and BR-KFAC for the purpose of allocating work. That is, we allocated BRC-KFAC work pretending it is BR-KFAC. While this work allocation strategy is even more sub-optimal for BRC-KFAC than it was for BR-KFAC, the implied workload scenarios are the same in the two extreme cases when the percentage of corrected modes is either 0%[29] or 100%[30]. That is, we know that for these extreme two cases, the idle time of BRC-KFAC is not worse than the one of BR-KFAC (under the BR-KFAC allocation strategy proposed). Furthermore, one can see that an optimal BRC-KFAC work allocation strategy should converge to the optimal BR-KFAC allocation strategy under these two extreme cases (though these extreme cases of BRC-KFAC have different equivalent T_{BR}; and T_{BR} needs to be an input to the optimal allocation strategy[31]).

Proposing a better workload allocation strategy is subject for future work.

We may have different Periods for Different K-factors (not implemented). Before ending *Section 5.4*, note that each K-factor (or group thereof) could have its own, different, periods T_{Brand}, T_{Cov}, T_{BR}, T_{inv}. This may be beneficial if there are K-factors which change at a slower rate than others, and if we designed an adaptive scheme to identify and exploit this aspect. We do not consider implementing this functionality here, mainly because this would be outside the scope of the thesis: we can ask the same question about vanilla K-FAC periods (this is not specific to our

[29] In which case we have no correction, so we fall back onto doing BR-KFAC.

[30] In which case performing a correction has the same cost as doing R-update (though is less accurate because of constraining our chosen subspace to be the already existing one, rather than obtained through power iteration - review Section 4.1.4). So from a computation cost point of view we fall back onto doing a BR-KFAC with higher R-update frequency (of BR-updated K-factors). Optimal work allocation is only influenced by the computation cost of workloads, not by accuracy.

[31] To see this, note that the best work allocation is the one where the total wall-time of a full workload scenario cycle is minimized. By a full workload scenario cycle we here mean the minimal repeating unit of horizontally-stacked makespans (when considering horizontally stacking all workload scenarios like the ones in Figure 5.6 in appropriate time order).

proposed variants, though our variants complicate the matter). Also, work allocation would change significantly from the one discussed, and it would be harder to optimize. Having constant periods across all K-factors is reasonable given that standard K-FAC implementations typically have the same T_{Cov} and T_{inv} for all K-factors.

5.5 Numerical Experiments: Set-up

In this section we present numerical results comparing distributed implementations of our proposed K-FAC improvements against two benchmarks: K-FAC and SGD. Note that the distributed design of K-FAC is based on the same choices as for our proposed K-FAC improvements (as specified in *Section 5.3*).

5.5.1 Datasets, Neural Networks, and GPU Specifications

We investigated the numerical performance of our algorithms on four image classification tasks: for the CIFAR10, CIFAR100, SVHN and Imagenette[32] (full-size, v2) datasets. We compared our solvers against the SGD and K-FAC benchmarks for *1, 2 and 4 GPUs*. For CIFAR10, CIFAR100 and SVHN datasets the neural net we used was VGG16_bn_lmxp as in *Chapter 4* (a slight modificaton to VGG16_bn to ensure the final FC layer computations are not of negligible cost). For the Imagenette dataset we used the standard resnet50[33] neural net (see [25]).

[32] A 10 classes subset of the ImageNet dataset (see https://github.com/fastai/imagenette).

[33] Note that in the light of our discussion about *which factors favor choosing second order methods* (in *Section 1.6*), the low *batch-size resnet50* is a relatively poor choice of test case for second order methods (and thus for K-FAC). This is because, in this case, adding standard K-FAC preconditioning: 1) gives a low improvement in the #steps to target-loss (as is the case with low batch-size and simpler-to-optimizer DNNs, like resnet50, as discussed in *Section 1.6*), and 2) introduces low preconditioning overhead (because the resnet layer size is low). This kind of test-case is not great because having a thin margin on both important factors that drive the over-all time difference between standard K-FAC and SGD means results are more subject to noise (a situation we can improve by more experiments - at least for mean-time based conclusions). Larger batch-size resenet50 test-cases would have been more appropriate (as these would have given larger improvement in #steps to target accuracy [65]). However, we only trained with low batch-size due to computational resources limitations.

We indeed see the lower-quality of results in this sense for resnet: 1) For our low-batch Resnet50 Imagenette results (*Table 5.6*), the metric $\text{SNR}_{\text{Improve}} := (\overline{\text{TtTA}}_{\text{K-FAC}} - \overline{\text{TtTA}}_{\text{Improvement}})/\sqrt{\hat{\mathbb{V}}(\text{TtTA}_{\text{K-FAC}}) + \hat{\mathbb{V}}(\text{TtTA}_{\text{Improvement}})}$ is relatively small (compared to other experiments), typically about 2.1 and at worst gets to be ≈ 1.5 (that is, the mean-time improvement between standard K-FAC and our proposed variants is somewhat close to the experiment noise level, and dominates it by little margin); 2) for the other test-cases (*Tables 5.3 - 5.5*) this metric ($\text{SNR}_{\text{Improve}}$) has much better values, almost all being above 6, some getting as high as 20, and the worst being no-less than 3. However, we believe there is still some value in our low batch-size resnet50 results - as it shows the behavior of our proposed improvements in a certain situation relative to standard K-FAC, and there is still some reasonable level of statistical significance in them (because

For Imagenette dataset, we used NVIDIA A100-PCIe GPUs with 40Gb of memory. For all other datasets we used NVIDIA Tesla V100-SXM2 GPUs with 32Gb of memory.

For each [GPU-type, dataset, neural net, solver] quadruple we tuned the hyperparameters as described next, and then did 5 runs with the final values. We stopped our optimization as soon as the test accuracy reached the target value. We used 92%, 70%, 95%, 91% target test-accuracy for CIFAR10, CIFAR100, SVHN and Imagenette respectively. For the first three datasets, the target accuracy choice is standard in the literature (see [20]). For Imagenette we chose a threshold which was reasonably challenging to hit with K-FAC.

5.5.2 Hyperparameters Tuning Methodology

For our hyperparameter tuning, we first tuned the parameters for K-FAC. Then all our proposed K-FAC improvements inherited the parameter values for the common hyperparameters (including learning rate schedules and weight decay).

The parameters specific to our K-FAC improvements did not need much tuning. Our adaptive target-rank strategy allowed us to bypass target-rank selection by instead setting a more intuitive parameter: target relative truncation error (a value of 3.3% was deemed sensible based on our regularization parameter). Our initial target-rank guess was simply set to be slightly larger than the batch size, based on the intuition gained in Chapter 3 about eigenspectrum decay (this hyperparameter has limited influence due to adaptive rank strategy). The times between different updates (T_{Brand}, T_{BR} etc.) were chosen to achieve a similar update frequency as K-FAC would. The RSVD parameters were selected as in *Chapters 3 - 4* (to control the projection error, review *Section 3.1.4* if required). None of the parameters specific to our proposed K-FAC improvements were tuned (no trial-error was performed): we simply used our best guesses in the basis of the previous chapters' results. Even so, the improvement we got over K-FAC was substantial.

To tune the K-FAC hyperparameters for each problem, we started from the hyperparameters used in *Chapter 4*, and manually optimized them. We began by performing one 20-epochs run on CIFAR10 with VGG16_bn_lmxp on 2 GPUs. We then changed one hyperparameter at a time for a few different values, performed a 20-epochs run for each, and chose the one which gave us the best convergence curve. We repeated this

the number of runs was reasonable - any hypothesis testing argument is omitted for brevity but the numbers to get these are in *Tables 5.3 - 5.6*).

Another argument against our resnet50 experiments may be that being relatively simple to optimize (relatively good conditioning), the test-case does not truly stress-test our low-rank improvements. This is not a problem, as we can have both easy and difficult (eg. VGG16_bn) test-cases.

for different parameters until 92% target-accuracy was reached. We then used this final setting for CIFAR10 on 2 GPUs as initial guess for different datasets and different number of GPUs. When changing the number of GPUs, the base learning rate guess was scaled in proportion to the square-root of the maximum batch size (as proposed in [104]). For different data-sets we changed the target-accuracy to the values stated in *Section 5.5.1*, and possibly increased the 20-epochs run to more epochs if needed. We did not use grid-search to avoid computationally-heavy tuning.

We tuned SGD independently of K-FAC (and of any proposed K-FAC improvements), using the same strategy and initial guesses from [20].

5.5.3 Hyperparameters Choices Constant for all Experiments

In this section we present the hyperparameters which were the same for all numerical experiments. We will present hyperparameters common to all algorithms, then go into hyperparameters specific to SGD, K-FAC, R-KFAC, B-KFAC, BR-KFAC and BRC-KFAC. Note that each K-FAC-based algorithm inherits all the parameters from the ones specified before it, and on top of that has its own specific parameters. This is because each K-FAC variant adds complexity (and hyperparameters) to the one before it in the list. Note that the hyperparameters from R-KFAC that are inherited by B, BR and BRC variants are only relevant to the R-updated K-Factors (not the B-, BR- or BRC-updated K-Factors of the respective algorithms). Review *Figures 5.4(L)* and *5.6(L)* for a better understanding of this latter statement.

In our experiments, we set all the common parameters to the same value, which we specify only once, unless otherwise stated. For example, by saying $T_{\text{inv}} = 100$ under the K-FAC hyperparameters description we also mean $T_{\text{inv}} = 100$ for all algorithms listed afterwards. Similarly, an $r_{\text{rsvd}} = 220$ for R-KFAC implies $r_{\text{rsvd}} = 220$ for B-KFAC, BR-KFAC and BRC-KFAC as well (see *Chapters 3 - 4* for details about hyperparameters).

Hyperparameters Common to all Algorithms. For all algorithms we used a per-GPU batch size of 128 and weight decay parameter of 7e-04.

SGD Hyperparameters. For SGD, we used a momentum parameter of 0.9 throughout as used with success in literature (see numerical experiments in [20, 52, 105]). The learning rate tuning methodology was as follows: (1) start with hyper-paremeters values which give schedules as close as possible to what was found to be the optimal in the literature for the problems at hand ([20, 52, 105]); and (2) "locally" fine-tune these hyper-parameters with a few attempts (no grid-search).

K-FAC Hyperparameters. We set $T_{\text{inv}} = 100$ and $T_{\text{Cov}} = 20$, $\rho = 0.95$. The K-FAC clip parameter was 0.07, the regularization parameter was set to use the schedule in *Chapter 4* (see $\phi_{\lambda,k}$ in Section 4.4). Gradient momentum was set to 0.

R-KFAC Hyperparameters. Our adaptive-rank strategy (described in *Section 5.3.2*) starts from an initial target-rank guess of 180 and targets a 3.3% relative truncation error (the other parameters in *Algorithm 5.1* were $r_M = 700$, $r_m = 10$, $\epsilon = 0.033$, $\epsilon_I = 0.02$, $\epsilon_D = 0.007$). We adjust the rank every time the low-rank inverse representation is changed (once in $T_{\text{inv}} = 100$ iterations). Our RSVD used 3 power iterations and an oversampling parameter of 10.

B-KFAC Hyperparameters. As implied by our explanation at the start of *Section 5.5.2*, our R-updated K-Factors[34] use the adaptive rank strategy hyperparameters specified for R-KFAC. For our B-updated K-factors we use the same adaptive rank hyperparameters except for the maximum admissible target-rank, which we set to 500. We set $T_{\text{Brand}} := 100$ $(= T_{\text{inv}})$. Unlike in *Chapter 4*, in the B-update we choose to truncate before inversion (recall *Section 4.1.1*). This saves computation time. We found (with limited experiments) this works best in practice for our distributed setting.

BR-KFAC Hyperparameters. We change T_{Brand} from 100 to $T_{\text{Brand}} := 20$ $(= T_{\text{Cov}})$ and set $T_{\text{BR}} := 500$ $(= 5T_{\text{inv}})$.

BRC-KFAC Hyperparameters. We correct 20% of modes once in 100 iterations. That is, we set $\phi_{\text{crc}} = 0.2$ and $T_{\text{corct}} = 100$.

[34]Review Figure 5.4(L) to recall that B-KFAC has R-updated and B-updated K-Factors.

5.5.4 Dataset-Dependent Hyperparameters Choices

The learning rate schedule was different for different combinations of solver and datasets. However, all our learning rate schedules were of the 'staircase' type: $\alpha_k = \alpha_0 \cdot \phi_{lr}^{\lfloor n_e(k)/T_{lr} \rfloor}$, where $n_e(k)$ is the number of epochs fully expended till iteration k (i.e. we start from an α_0 learning rate and every T_{lr} epochs we reduce the learning rate again by a factor of $\phi_l r > 1$). *Table 5.1* shows the learning rate schedule parameters for each situation.

Table 5.1: Distributed Experiments Learning Rate Parameters.

			VGG16 - bn - lmxp					
			SGD	**K-FAC**	**R**	**B**	**BR**	**BRC**
CIFAR10	**1 GPU:**	α_0	0.07071	0.21213	0.21213	0.21213	0.21213	0.21213
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	6	6	6	6	6
	2 GPUs:	α_0	0.1	0.3	0.3	0.3	0.3	0.3
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	6	6	6	6	6
	4 GPUs:	α_0	0.14142	0.42426	0.42426	0.42426	0.42426	0.42426
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	38	6	6	6	6	6
CIFAR100	**1 GPU:**	α_0	0.07071	0.21213	0.21213	0.21213	0.21213	0.21213
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	6	6	6	6	6
	2 GPUs:	α_0	0.1	0.3	0.3	0.3	0.3	0.3
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	6	6	6	6	6
	4 GPUs:	α_0	0.14142	0.42426	0.42426	0.42426	0.42426	0.42426
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	38	6	6	6	6	6
SVHN	**1 GPU:**	α_0	0.07071	0.21213	0.21213	0.21213	0.21213	0.21213
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	4	4	4	4	4
	2 GPUs:	α_0	0.1	0.3	0.3	0.3	0.3	0.3
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	25	4	4	4	4	4
	4 GPUs:	α_0	0.14142	0.42426	0.42426	0.42426	0.42426	0.42426
		ϕ_{lr}	3.1	3.1	3.1	3.1	3.1	3.1
		T_{lr}	38	5	5	5	5	5
			ResNet50					
			SGD	**K-FAC**	**R**	**B**	**BR**	**BRC**
Imagenette full-size, v2	**1 GPU:**	α_0	0.07071	0.21213	0.21213	0.21213	0.21213	0.21213
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	29	11	11	11	11	11
	2 GPUs:	α_0	0.1	0.3	0.3	0.3	0.3	0.3
		ϕ_{lr}	3	3	3	3	3	3
		T_{lr}	35	18	18	18	18	18
	4 GPUs:	α_0	0.14142	0.424	0.424	0.424	0.424	0.424
		ϕ_{lr}	3.1	3.1	3.1	3.1	3.1	3.1
		T_{lr}	70	26	26	26	26	26

5.6 Numerical Experiments: Results

We first look at a summary of all our results which highlights the kind of speed-up we can get by using our proposed K-FAC improvements (in *Section 5.6.1*). We then look at some deeper details for a better understanding. In *Section 5.6.2* we investigate some illustrative convergence plots to gain more intuition about the algorithms. In *Section 5.6.3* we examine the scaling efficiencies of all algorithms used. When doing so, we also examine more basic quantities like per-step time and run-to-run variability.

5.6.1 Summary of Results

Table 5.2 shows a highly condensed summary of all our results for this chapter. We see that our proposed K-FAC improvements outperform both SGD and K-FAC in all settings. In general, this is attributed to our proposed improvements having about the same time per step as SGD, but a much better direction due to preconditioner (similar ot K-FAC). That is, SGD requires many more iterations than our proposed K-FAC improvements (see the #EpochsToAcc% values), and K-FAC has much more expensive steps (use TimeToAcc% / #EpochsToAcc% to see this).

For Imagenette, SGD did not reach our target accuracy: outlining the importance of having a preconditioner. In those cases, we report the time to a lower target accuracy.

For CIFAR10, CIFAR100 and SVHN (with VGG16_bn_lmxp) we get speed-ups ranging from 3× to 4×. For Imagenette, we get much lower speed-ups of only about 1.1×. This can be attributed to the neural net used: VGG16_bn (and VGG16_bn_lmxp) has much wider layers than resnets, so the K-factors are much larger. Recall from *Chapters 3* and *4* that we expected our K-FAC improvements to have a higher speed-up factor when the layer width is large. This behavior is not concerning, and actually aligns with our goal: improving K-FAC scalability with layer size. Note that for Imagenette with resnet50, the time per epoch of standard K-FAC is only 10% larger than the one of SGD (divide time to accuracy by # epochs to accuracy in *Table 5.2* to see this). This indicates that standard K-FAC is not in a layer-width regime where its inverse complexity is a major bottleneck, for this problem[35]. In this case, we do not need to improve K-FAC by much to begin with.

We also see that all proposed K-FAC improvements are much closer to each other in terms of performance than either of them is to K-FAC. This can be attributed to the fact that the over-all complexity of all our K-FAC improvements is quadratic in

[35]Compare with test-cases using VGG16_bn_lmxp, when the K-FAC time per epoch is more than double the one of SGD.

Table 5.2: Numerical Performance of Proposed Distributed Algorithms and Comparison with Benchmarks: Summary. Time is in seconds. All numbers are average across all successful runs. The percentages shown refer to test accuracy (review *Section 5.5.1*). We had a total of 5 runs per solver-dataset-net-#GPUs quadruple.

VGG16 - bn - lmxp (with NVIDIA V100-SXM2 32GB GPUs)							
		SGD	**K-FAC**	**R**	**B**	**BR**	**BRC**
CIFAR 10	**1 GPU:** TimeTo92%	3583.89	2112.38	551.39	535.84	540.94	541.22
	#EpochTo92%	75.00	19.80	18.40	17.80	18.00	18.00
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
	2 GPUs: TimeTo92%	1763.88	1643.61	541.51	540.56	549.47	550.96
	#EpochTo92%	72.2	24.00	18.60	18.40	18.00	18.00
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
	4 GPUs: TimeTo92%	1089.34	1067.05	352.04	349.56	355.75	355.68
	#EpochTo92%	88.00	25.20	24.00	24.00	24.00	24.00
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
CIFAR 100	**1 GPU:** TimeTo70%	2384.97	3172.80	1113.57	1103.05	1073.44	1087.55
	#EpochTo70%	50.00	18.00	18.00	17.80	16.00	16.20
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
	2 GPUs: TimeTo70%	1217.76	1643.61	541.51	540.56	549.47	550.96
	#EpochTo70%	50.00	18.60	18.00	18.40	18.00	18.00
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
	4 GPUs: TimeTo70%	546.37	1081.59	356.03	351.67	358.76	359.54
	#EpochTo70%	44.00	25.00	24.00	24.00	24.00	24.00
	#Successful Runs/#Runs	5/5	5/5	5/5	5/5	5/5	5/5
SVHN	**1 GPU:** TimeTo95%	4397.94	2511.39	1024.88	742.11	917.79	846.15
	#EpochTo95%	62.75	9.75	11.33	8.25	9.5	8.8
	#Successful Runs/#Runs	4/5	4/5	3/5	4/5	4/5	5/5
	2 GPUs: TimeTo95%	2112.59	2003.99	616.03	561.17	535.77	535.97
	#EpochTo95%	59.00	15.40	14.00	13.00	12.00	12.00
	#Successful Runs/#Runs	3/5	5/5	4/5	5/5	5/5	5/5
	4 GPUs: TimeTo95%	696.07	2142.65	524.79	512.63	524.88	490.34
	#EpochTo95%	38.25	34.50	24.40	24.00	24.00	22.40
	#Successful Runs/#Runs	4/5	2/5	5/5	5/5	5/5	5/5
ResNet50 (with NVIDIA A100-PCIe 40GB GPUs)							
		SGD	**K-FAC**	**R**	**B**	**BR**	**BRC**
Image-nette full-size v2	**1 GPU:** TimeTo91%	2619 (89.5%)	1137.01	1000.72	1004.88	1040.38	1003.68
	#EpochTo91%	123 (89.5%)	46.00	44.40	44.60	44.20	42.40
	#Successful Runs/#Runs	0/5	5/5	5/5	5/5	5/5	5/5
	2 GPUs: TimeTo91%	2047 (88.5%)	915.70	875.96	872.66	816.03	847.84
	#EpochTo91%	175 (88.5%)	72.20	72.00	72.00	67.20	70.00
	#Successful Runs/#Runs	0/5	5/5	5/5	5/5	5/5	5/5
	4 GPUs: TimeTo91%	1081 (87.5%)	958.92	825.51	786.57	836.76	784.51
	#EpochTo91%	173 (87.5%)	138.00	126.00	119.75	127.20	119.25
	#Successful Runs/#Runs	0/5	5/5	5/5	4/5	5/5	4/5

layer size (though B-KFAC has linear complexity inverse computation), while the one of standard K-FAC is cubic. However, some of them tend to take slightly less time than others. We investigate this further in the next sections.

When comparing our K-FAC improvements to each other, there are two competing effects: cheaper inverse updates imply lower per-step time, but a less accurate preconditioner which might give a worse per-step performance (and thus more steps to target accuracy). However, these two effects are complicated by the adaptive rank strategy we run: less accurate updates may compensate by increasing the target rank[36]. Still,

[36]At least reducing the truncation error - though this is not the only error there is.

in *Sections 5.6.2 - 5.6.3* we try to dissect the performance of each optimizer in terms of *preconditioner quality* and *time per step*. This will allow us to (simply but crudely) infer preconditioner quality (using number of steps to target accuracy as a metric).

5.6.2 Average of Convergence Plots

We now look at convergence plots. To condense our results, we plot the average across the 5 runs we performed for each solver. The SGD-SVHN pair is an exception: in this case we omitted all runs where SGD got stuck at 20% test accuracy[37], and only used the remaining number of runs, rather than the total of 5 runs when averaging. The average is performed over both x-axis and y-axis of the original curves. *Figures 5.8, 5.9, 5.10* show test accuracy average plots for 1, 2, 4 GPUs respectively.

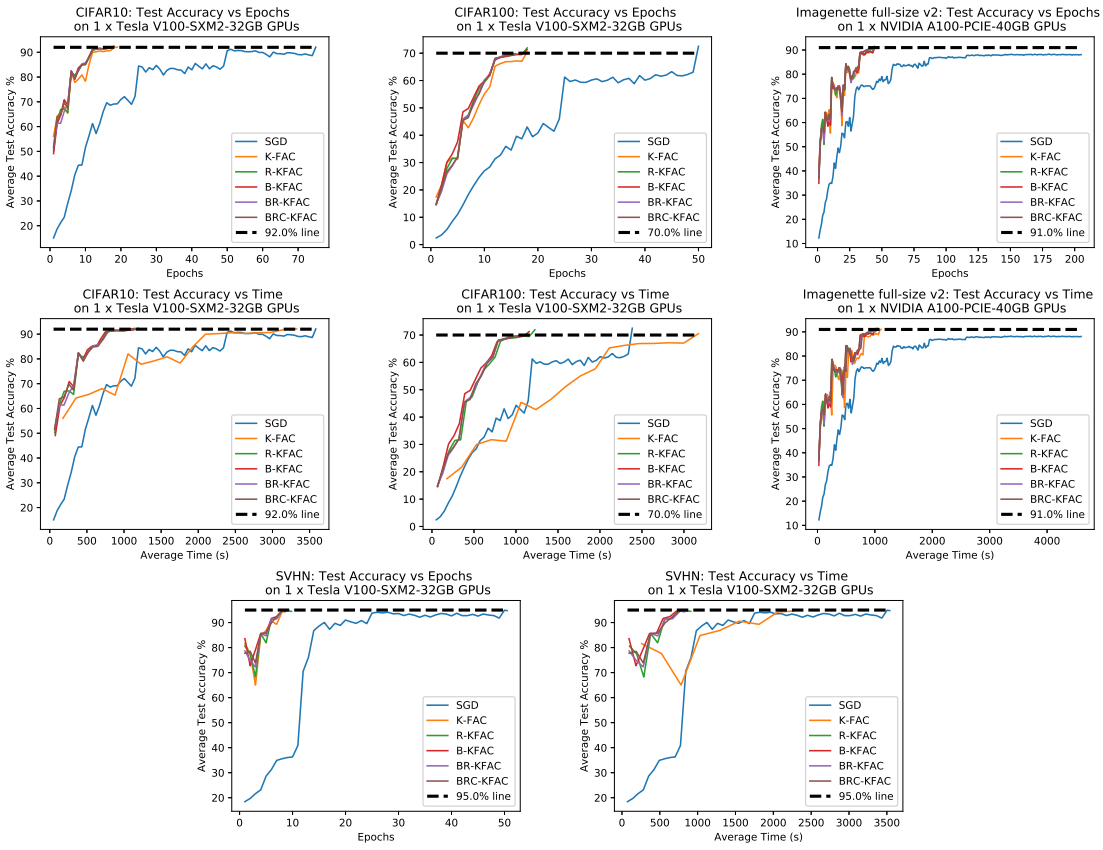


Figure 5.8: Average of Convergence Plots when using 1 GPU. One SGD on SVHN run got stuck at 20% test accuracy and was removed from averaging.

The average of convergence plots must be interpreted with care, particularly when the run-to-run variability is large. In our case, this variability was small (after the excluded SGD-SVHN runs).

[37]Possibly in a local minima or a saddle.

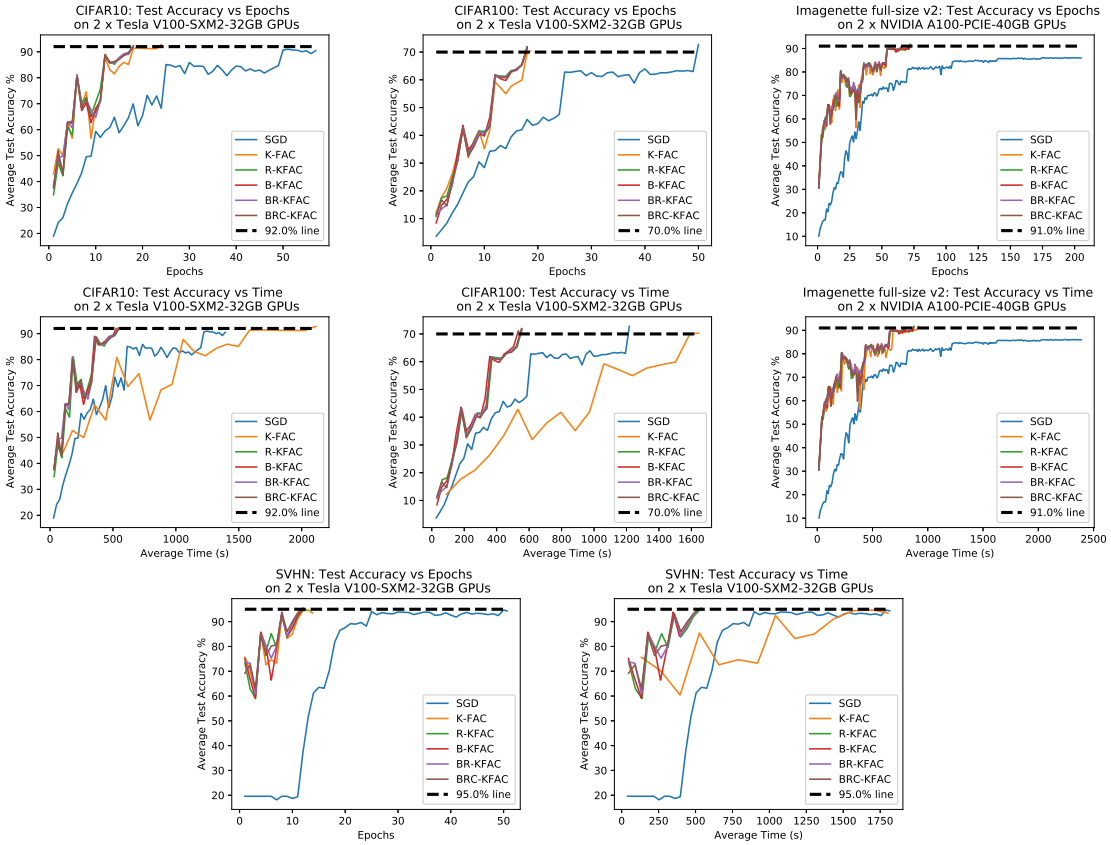


Figure 5.9: Average of Convergence Plots when using 2 GPUs. Two SGD on SVHN runs got stuck at 20% test accuracy and was removed from averaging.

The average of convergence plots were particularly revealing for gauging the quality of the preconditioner of our proposed K-FAC improvements (relative to the standard K-FAC preconditioner). On the test accuracy vs epochs plots, we see that our K-FAC improvements curves very closely resemble the K-FAC curve (for all datasets and number of GPUs). This suggests that the preconditioner (and thus the step) of our proposed K-FAC improvements are all very similar to the K-FAC preconditioner. This was expected given our target relative truncation error of 3.3% with a regularization of 10% of the maximum eigenvalue (review *Section 5.5* for details). However, when plotting test-accuracy against time we see that our proposed K-FAC improvements attain substantial computational savings (as we saw in *Table 5.2*).

The reality for SGD on SVHN is much worse than it appears from the picture: sometimes SGD got stuck at 20% test accuracy (all unsuccessful runs observed in *Table 5.2* did so). This was due to SGD gradient becoming vanishingly small (possibly due to running into a local minima, saddle, or very low gradient region). The occurrence of

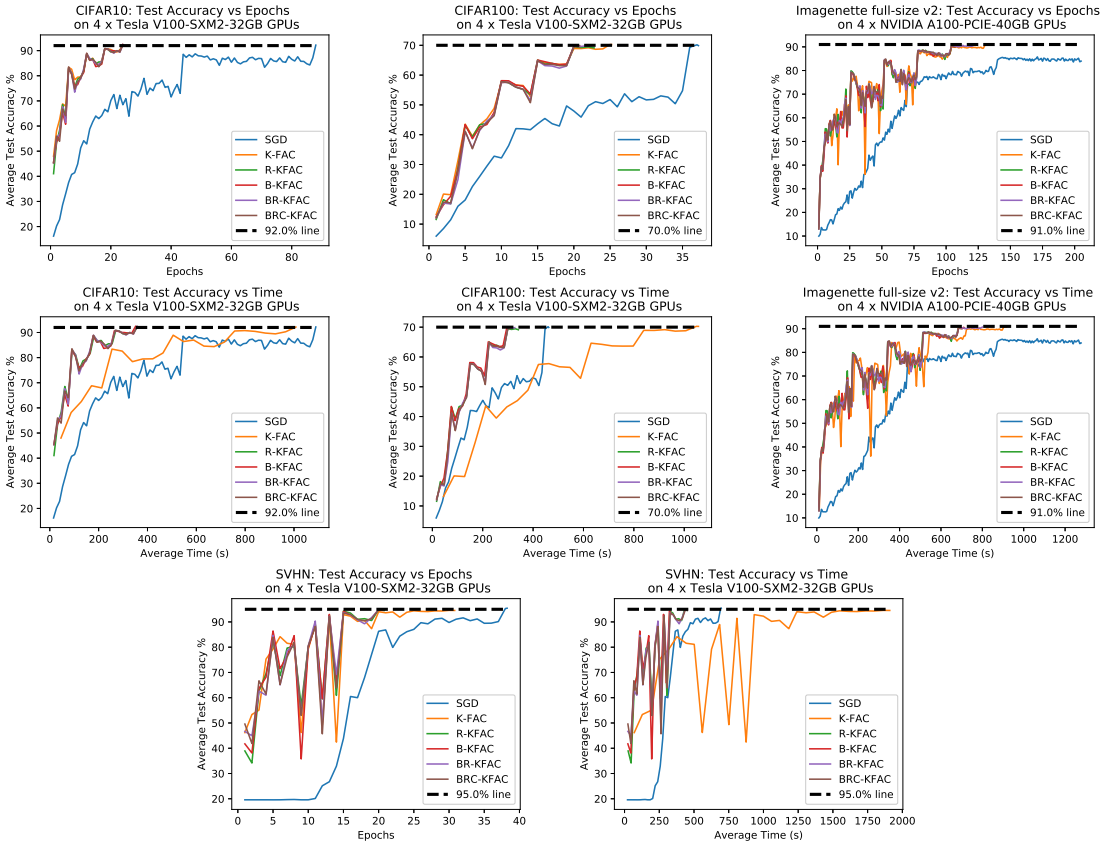


Figure 5.10: Average of Convergence Plots when using 4 GPUs. One SGD on SVHN run got stuck at 20% test accuracy and was removed from averaging.

these outcomes emphasize another advantage of using the K-FAC preconditioner[38]: we avoid getting stuck in saddles or local minima ([1, 20]), thus avoiding wasteful runs.

5.6.3 Scaling Efficiency with Number of GPUs

In this section, we look at detailed results with the aim of understanding the scaling efficiency of our proposed algorithm, and compare this to the benchmark. The results are presented in *Tables 5.3, 5.4, 5.5, 5.6* for CIFAR10, CIFAR100, SVHN and Imagenette-fs-v2 respectively.

We consider four metrics: number of epochs to target test accuracy (#EtTA), time per epoch (TpE), time per step (TpS), time to target test accuracy (TtTA) and also their respective scaling efficiencies (denoted as SE (#EtTA), SE (TpE), SE (TpS), SE (TtTA)). We advise the reader to familiarize with these acronyms well, as they will be extensively used henceforth. Typically, literature is concerned with TtTA and TpE

[38]Aside from getting: (1) higher test accuracy, and (2) smaller run times for successful runs through our improved K-FAC proposed algorithms.

and their scaling efficiency only. However, the other two metrics are also revealing. Note that the time per step is an average (total time / total number of steps).

The scaling efficiencies are defined in the literature having the following ideal scenario in mind: when doubling the number of GPUs we would ideally see the same (average) time per step, and reach our target test accuracy after parsing the same amount of data (after the same number of epochs). Since the epochs have 2 times fewer steps when there are 2 times more GPUs (double total gradient batch size), doubling the number of GPUs should halve the time to target accuracy [20].

Thus, we have the following definitions for our scaling efficiencies (for n GPUs):

$$\mathrm{SE}_n(\#\mathrm{EtTA}) = \frac{\#\mathrm{EtTA \text{ for } 1 \text{ GPU}}}{\#\mathrm{EtTA \text{ for n GPUs}}}, \tag{5.1}$$

$$\mathrm{SE}_n(\mathrm{TpE}) = \frac{1}{n}\frac{\mathrm{TpE \text{ for } 1 \text{ GPU}}}{\mathrm{TpE \text{ for n GPUs}}}, \tag{5.2}$$

$$\mathrm{SE}_n(\mathrm{TpS}) = \frac{\mathrm{TpS \text{ for } 1 \text{ GPU}}}{\mathrm{TpS \text{ for n GPUs}}}, \tag{5.3}$$

$$\mathrm{SE}_n(\mathrm{TtTA}) = \frac{1}{n}\frac{\mathrm{TtTA \text{ for } 1 \text{ GPU}}}{\mathrm{TtTA \text{ for n GPUs}}}. \tag{5.4}$$

Referencing to 1 GPU is somewhat arbitrary, we can choose any number as our reference (typically the smallest available number of GPUs is chosen). First, note that [TpE for n GPUs] = [TpS for n GPUs] $\cdot \lceil |\mathcal{D}_{\mathrm{train}}|/(n \cdot n_{\mathrm{B/GPU}}) \rceil$, where $|\mathcal{D}_{\mathrm{train}}|$ is the number of points in the training data set. Thus, substituting into (5.2) we can see that $\mathrm{SE}_n(\mathrm{TpE}) \approx \mathrm{SE}_n(\mathrm{TpS})$ holds.

Further note that $\mathrm{SE}_n(\mathrm{TtTA})$ can be decomposed as $\mathrm{SE}_n(\mathrm{TtTA}) = \mathrm{SE}_n(\mathrm{TpE}) \cdot \mathrm{SE}_n(\#\mathrm{EtTA})$ since $\#\mathrm{EtTA} \cdot \mathrm{TPE} = \mathrm{TtTA}$. So we can understand our $\mathrm{SE}_n(\mathrm{TtTA})$ better (which is the practically important metric) by looking at the two terms in its factorization. Importantly, $\mathrm{SE}_n(\mathrm{TpE})$ ($\approx \mathrm{SE}_n(\mathrm{TpS})$) is a metric for how time-efficiently we perform computations when adding in more GPUs, while $\mathrm{SE}_n(\#\mathrm{EtTA})$ is a metric for how efficiently we use the data[39] when adding in more GPUs. In general, there is a trade-off between the two: if we want to better utilize data we then have to increase computation. Our LP K-factor computation choice favors a better $\mathrm{SE}_n(\mathrm{TpS})$ since the per-GPU K-FAC related workload reduces with the number of GPUs (however the communication workload always increases, as there is an inverse representation all-reduce - but less than it would with the HP approach; review *Section 5.2*). Conversely,

[39]Through the point of view of progress per data units.

the HP K-factor computation approach (which we did not choose) favors a higher SE_n(#EtTA) as the preconditioner quality[40] is increased with the number of GPUs (review *Section 5.2*). However, a better preconditioner quality may not always directly translate to more progress per step. This argument is another way to motivate our choice of LP K-factor computation.

We now present our discussion for the results in *Tables 5.3, 5.4, 5.5*, and *5.6*.

5.6.3.1 Per-Step Time

There are two competing effects for the per-step time:

- Per-step time tends to reduce when adding more GPUs due to reducing the per-GPU K-Factors computation and inversion load (up until some #GPUs),
- Per-step time tends to increase (linearly in # GPUs) due communications costs: gradient all-reduce and K-Factor inverse representation all-reduce.

Depending on which cost dominates, we might see a reducing or increasing per-step time as the number of GPUs increases. For relatively large K-Factors, we expect to see a decrease in per-step time, while for small K-factors we expect to see the reverse.

For CIFAR10, CIFAR100 and SVHN we see that the time per step for all K-FAC based algorithms reduces, giving efficiencies above 1 (SE(TpS) $>$ 1). This happens because the reduction in per-GPU K-FAC preconditioning workload exceeds the increase in communication costs as more GPUs are added. We are in this type of regime because the neural net used (VGG16_bn_lmxp) has relatively large K-Factors, and thus the K-FAC preconditioning work dominates communication costs. To see this, compare the TpS of standard K-FAC with the ones of SGD - the difference between the two is due to the K-Factor related computation and the K-Factor inverse representation communication. Note that the TpS of SGD increases with increasing #GPUs, as there is no preconditioner work to be split, and the communication cost (gradient all-reduce) increases linearly (or logarithmically, depending on the communication algorithm choice) in the number of GPUs. Note that the difference between a K-FAC variant TpS and the SGD TpS is the average cost per step associated with the K-FAC preconditioning of the K-FAC variant. We see that for CIFAR10, CIFAR100 and SVHN the average cost per step of the standard K-FAC preconditioning part is about 3 times the cost of obtaining the gradient. Again, this happens because the K-FAC preconditioning is of cubic time-complexity in layer size, while the gradient computation is quadratic, and

[40]Lower preconditioner variance around the expected value.

Table 5.3: Scaling efficiency for CIFAR10 dataset with VGG16_bn_lmxp Neural Network. The gradient BS is #GPUs × 128, the preconditioner BS is 128 (review *Section 5.2.1*, we choose $n_{\mathrm{B/GPU}} = 128$.). Target Accuracy for this dataset is 92.00%. Times are presented as mean ± std. Only the runs that reach the target accuracy (out of a total of 5) are used. This number can be observed in the last column.

	SGD (benchmark)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	75.00 ±0.00	1.00	47.74 ±0.17	1.00	0.12211 ±0.0005	1.00	3583.89 ±7.69	1.00	5/5
2	72.20 ±7.68	1.04	24.39 ±0.11	0.98	0.12442 ±0.0006	0.98	1763.88 ±188.03	1.02	5/5
4	88.00 ±0.00	0.85	12.35 ±0.09	0.97	0.12603 ±0.0009	0.97	1089.34 ±2.29	0.82	5/5
	K-FAC (benchmark algorithm, but using our proposed distributed design)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	19.80 ±1.60	1.00	174.80 ±2.52	1.00	0.44705 ±0.0064	1.00	3463.24 ±277.48	1.00	5/5
2	24.00 ±0.00	0.83	87.53 ±1.91	1.00	0.44656 ±0.0098	1.00	2112.38 ±4.59	0.82	5/5
4	25.20 ±2.40	0.79	41.68 ±1.20	1.03	0.42528 ±0.0122	1.03	1058.04 ±99.85	0.81	5/5
	R-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	18.40 ±0.49	1.00	60.14 ±0.62	1.00	0.15382 ±0.0016	1.00	1109.90 ±30.01	1.00	5/5
2	18.60 ±0.80	0.99	29.52 ±0.29	1.02	0.15059 ±0.0015	1.02	551.39 ±23.37	1.01	5/5
4	24.00 ±0.00	0.77	14.57 ±0.14	1.03	0.14870 ±0.0015	1.03	352.04 ±0.30	0.79	5/5
	B-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	17.80 ±0.40	1.00	60.20 ±0.55	1.00	0.15396 ±0.0014	1.00	1074.27 ±22.59	1.00	5/5
2	18.40 ±0.49	0.97	29.00 ±0.19	1.04	0.14796 ±0.0010	1.04	535.84 ±13.54	1.00	5/5
4	24.00 ±0.00	0.74	14.47 ±0.13	1.04	0.14764 ±0.0013	1.04	349.56 ±0.40	0.77	5/5
	BR-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	18.00 ±0.00	1.00	65.75 ±1.80	1.00	0.16816 ±0.0046	1.00	1181.44 ±0.65	1.00	5/5
2	18.00 ±0.00	1.00	29.98 ±0.41	1.10	0.15296 ±0.0021	1.10	540.94 ±0.22	1.09	5/5
4	24.00 ±0.00	0.75	14.73 ±0.15	1.12	0.15033 ±0.0015	1.12	355.75 ±0.26	0.83	5/5
	BRC-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	18.00 ±0.00	1.00	65.75 ±1.82	1.00	0.16815 ±0.0047	1.00	1181.47 ±0.90	1.00	5/5
2	18.20 ±0.40	1.00	29.99 ±0.43	1.10	0.15301 ±0.0022	1.10	541.22 ±0.46	1.09	5/5
4	24.00 ±0.00	0.75	14.73 ±0.14	1.12	0.15030 ±0.0014	1.12	355.68 ±0.40	0.83	5/5

Table 5.4: Scaling efficiency for CIFAR100 dataset with VGG16_bn_lmxp Neural Network. The gradient BS is #GPUs × 128, the preconditioner BS is 128 (review *Section 5.2.1*, we choose $n_{\mathrm{B/GPU}} = 128$.). Target Accuracy for this dataset is 70.00%. Times are presented as mean ± std. Only the runs that reach the target accuracy (out of a total of 5) are used. This number can be observed in the last column.

	SGD (benchmark)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	50.00 ±0.00	1.00	47.64 ±0.13	1.00	0.12185 ±0.0003	1.00	2384.97 ±4.73	1.00	5/5
2	50.00 ±0.00	1.00	24.30 ±0.10	0.98	0.12396 ±0.0005	0.98	1217.76 ±3.23	0.98	5/5
4	44.00 ±14.00	1.14	12.35 ±0.09	0.96	0.12603 ±0.0009	0.96	546.37 ±172.98	1.09	5/5
	K-FAC (benchmark algorithm, but using our proposed distributed design)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	18.00 ±0.00	1.00	176.17 ±4.96	1.00	0.45056 ±0.0127	1.00	3172.80 ±76.73	1.00	5/5
2	18.60 ±0.49	0.97	88.29 ±0.23	1.00	0.45047 ±0.0012	1.00	1643.61 ±42.75	0.97	5/5
4	25.80 ±1.60	0.70	41.82 ±0.12	1.05	0.42676 ±0.0012	1.05	1081.59 ±66.71	0.73	5/5
	R-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	18.00+ ±0.00	1.00	61.76 ±0.37	1.00	0.15795 ±0.0010	1.00	1113.57 ±0.40	1.00	5/5
2	18.00 ±0.00	1.00	29.97 ±0.22	1.03	0.15291 ±0.0011	1.03	541.51 ±0.82	1.03	5/5
4	24.00 ±0.00	0.75	14.74 ±0.14	1.05	0.15043 ±0.0014	1.05	356.03 ±0.21	0.78	5/5
	B-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	17.80 ±0.40	1.00	61.89 ±0.40	1.00	0.15829 ±0.0010	1.00	1103.05 ±24.06	1.00	5/5
2	18.40 ±0.80	0.97	29.27 ±0.17	1.06	0.14935 ±0.0009	1.06	540.56 ±23.40	1.02	5/5
4	24.00 ±0.00	0.74	14.56 ±0.13	1.06	0.14856 ±0.0013	1.06	351.67 ±0.41	0.78	5/5
	BR-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	16.00 ±1.26	1.00	67.27 ±1.85	1.00	0.17204 ±0.0047	1.00	1073.44 ±88.01	1.00	5/5
2	18.00 ±0.00	0.89	30.47 ±0.46	1.11	0.15547 ±0.0024	1.11	549.47 ±0.42	1.09	5/5
4	24.00 ±0.00	0.67	14.86 ±0.17	1.13	0.15165 ±0.0018	1.13	358.76 ±0.73	0.75	5/5
	BRC-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	16.20 ±1.33	1.00	67.31 ±1.85	1.00	0.17215 ±0.0047	1.00	1087.55 ±92.30	1.00	5/5
2	18.00 ±0.00	0.89	30.46 ±0.49	1.12	0.155542 ±0.0025	1.12	550.96 ±3.25	0.99	5/5
4	24.00 ±0.00	0.67	14.90 ±0.16	1.13	0.15200 ±0.0017	1.13	359.54 ±0.71	0.76	5/5

Table 5.5: Scaling efficiency for SVHN dataset with VGG16_bn_lmxp Neural Network. The gradient BS is #GPUs × 128, the preconditioner BS is 128 (review *Section 5.2.1*, we choose $n_{\mathrm{B/GPU}} = 128$.). Target Accuracy for this dataset is 95.00%. Times are presented as mean ± std. Only the runs that reach the target accuracy (out of a total of 5) are used. This number can be observed in the last column.

	SGD (benchmark)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	62.75 ±12.75	1.00	70.00 ±0.19	1.00	0.01690 ±0.0000	1.00	4397.94 ±895.88	1.00	4/5
2	59.00 ±11.34	1.06	35.71 ±0.13	0.98	0.01724 ±0.0001	0.98	2112.59 ±403.92	1.04	3/5
4	38.25 ±0.43	1.64	18.17 ±0.12	0.96	0.01754 ±0.0001	0.96	696.07 ±7.75	1.58	4/5
	K-FAC (benchmark algorithm, but using our proposed distributed design)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	9.75 ±1.48	1.00	256.79 ±4.66	1.00	0.06201 ±0.0011	1.00	2511.39 ±372.95	1.00	4/5
2	15.40 ±1.20	0.63	129.93 ±3.40	0.99	0.06274 ±0.0016	1.00	2003.99 ±153.02	0.63	5/5
4	34.50 ±4.50	0.28	61.70 ±4.11	1.04	0.05955 ±0.0040	1.04	2142.65 ±279.71	0.29	2/5
	R-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	11.33 ±2.62	1.00	92.26 ±2.26	1.00	0.02228 ±0.0005	1.00	1024.88 ±237.83	1.00	3/5
2	14.00 ±2.00	0.81	43.45 ±0.65	1.06	0.02098 ±0.0003	1.06	616.03 ±84.46	0.83	4/5
4	24.40 ±3.14	0.46	21.32 ±0.24	1.08	0.02058 ±0.0002	1.08	524.79 ±67.35	0.49	5/5
	B-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	8.25 ±0.43	1.00	88.81 ±1.43	1.00	0.02145 ±0.0003	1.00	742.11 ±38.81	1.00	4/5
2	13.00 ±1.55	0.63	42.95 ±0.47	1.03	0.02074 ±0.0002	1.03	561.17 ±66.10	0.66	5/5
4	24.00 ±6.72	0.34	21.25 ±0.21	1.04	0.02051 ±0.0002	1.04	512.63 ±141.68	0.36	5/5
	BR-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	9.50 ±1.66	1.00	97.40 ±2.14	1.00	0.02352 ±0.0005	1.00	917.79 ±166.53	1.00	4/5
2	12.00 ±0.00	0.79	44.53 ±0.47	1.09	0.02150 ±0.0002	1.09	535.77 ±0.24	0.86	5/5
4	24.00 ±3.52	0.40	21.77 ±0.22	1.12	0.02101 ±0.0002	1.12	524.88 ±77.42	0.44	5/5
	BRC-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	8.80 ±1.17	1.00	96.40 ±2.38	1.00	0.02328 ±0.0006	1.00	846.15 ±116.99	1.00	5/5
2	12.00 ±0.00	1.00	44.53 ±0.53	1.09	0.02150 ±0.0003	1.09	535.97 ±0.32	0.79	5/5
4	22.40 ±2.33	0.39	21.78 ±0.24	1.11	0.02102 ±0.0002	1.11	490.34 ±51.10	0.43	5/5

Table 5.6: Scaling efficiency for Imagenette (full-size, v2) dataset with resnet50 Neural Network. The gradient BS is #GPUs × 128, the preconditioner BS is 128 (review *Section 5.2.1*, we choose $n_{\mathrm{B/GPU}} = 128$.). Target Accuracy for this dataset is 91.00%. Times are presented as mean ± std. Only the runs that reach the target accuracy (out of a total of 5) are used. This number can be observed in the last column.

	SGD (benchmark) - did not reach more than 89.5% test accuracy. TpS(s) for: 1 GPU was 0.30253±0.0146; 2 GPUs was 0.31411±0.0086; 4 GPUs was 0.32704±0.0093								
	K-FAC (benchmark algorithm, but using our proposed distributed design)								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	46.00 ±1.79	1.00	24.66 ±1.06	1.00	0.33327 ±0.0144	1.00	1137.01 ±59.58	1.00	5/5
2	72.20 ±0.40	0.64	12.61 ±0.85	0.98	0.34083 ±0.0229	0.98	915.70 ±15.25	0.62	5/5
4	138.00 ±6.07	0.33	6.91 ±0.78	0.89	0.36386 ±0.0411	0.92	958.92 ±42.89	0.30	5/5
	R-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	44.40 ±0.49	1.00	22.51 ±0.57	1.00	0.30415 ±0.0077	1.00	1000.72 ±10.24	1.00	3/5
2	72.00 ±0.00	0.62	12.12 ±0.37	0.93	0.32747 ±0.0100	0.93	875.96 ±4.13	0.57	4/5
4	126.00 ±9.03	0.35	6.52 ±0.25	0.86	0.34332 ±0.0130	0.89	825.51 ±57.16	0.30	5/5
	B-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	44.60 ±0.49	1.00	22.50 ±0.53	1.00	0.30404 ±0.0071	1.00	1004.88 ±12.16	1.00	5/5
2	72.00 ±0.00	0.62	12.08 ±0.49	0.93	0.32639 ±0.0133	0.93	872.66 ±6.70	0.58	5/5
4	119.75 ±11.80	0.37	6.53 ±0.21	0.86	0.34391 ±0.0111	0.88	786.57 ±78.38	0.32	4/5
	BR-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	44.20 ±0.40	1.00	23.51 ±1.01	1.00	0.31770 ±0.0136	1.00	1040.38 ±15.92	1.00	4/5
2	67.20 ±6.37	0.66	12.11 ±0.46	0.97	0.32721 ±0.0124	0.97	816.03 ±81.06	0.64	5/5
4	127.20 ±4.35	0.35	6.55 ±0.24	0.90	0.34448 ±0.0124	0.92	836.76 ±28.67	0.31	5/5
	BRC-KFAC								
#GPUs	#EtTA	SE (EtTA)	TpE (s)	SE (TpE)	TpS (s)	SE (TpS)	TtTA (s)	SE (TtTA)	$\frac{\#\text{succ. runs}}{\#\text{total runs}}$
1	42.40 ±2.06	1.00	23.65 ±1.05	1.00	0.31963 ±0.0142	1.00	1003.68 ±49.84	1.00	5/5
2	70.00 ±2.61	0.61	12.08 ±0.45	0.98	0.32647 ±0.0122	0.98	847.84 ±29.15	0.59	5/5
4	119.25 ±16.32	0.36	6.54 ±0.25	0.90	0.34437 ±0.0132	0.93	784.51 ±106.40	0.32	4/5

our DNN had relatively large layers. Note that for our proposed K-FAC improvements, the average per step cost of preconditioning is only about 30% of the SGD cost. Thus, we see that our proposed K-FAC variants reduce the preconditioning cost by a factor

of 10× in this case. This however results in an over-all time (TtTA) reduction of only $2-3\times$ (as the gradient-related cost is fixed).

For **Imagenette (full-size, v2)** with resnet50 we see that the TpS of all K-FAC variants increases with the number of GPUs. Thus, our SE(TpS) is below 1 (but fairly close to 1 nevertheless). This is because the reduction in per-GPU K-FAC preconditioning load is less than the increase in communication costs as more GPUs are added. We observe this behaviour because resnet50 has relatively small layer widths, and thus small K-factors, meaning low K-FAC preconditioning overhead. To see this, note that the TpS of K-FAC is only 10% larger than the one of SGD: meaning the K-FAC preconditioning overhead is only 10% of the gradient computation cost (compare this to the 300% figure we had for VGG16_bn_lmxp). We also see that the preconditioning overhead for our proposed K-FAC improvements is about 0.002s - 0.015s per step, while the one of standard K-FAC is 0.03s. Thus, our proposed K-FAC improvements reduce our average per-step preconditioning cost by a factor of 2×-15×, which is (sometimes) lower than the one we had for the VGG16_bn_lmxp experiments (due to the relatively lower size of K-factors). Furthermore, because the preconditioning cost of standard K-FAC was relatively low to begin with in this case, we only see TpS improvement of about $4-8\%$ when using our proposed K-FAC improvements. However, the TtTA (s) is typically about 10% better, because (somewhat surprisingly) our proposed improvements also seem to take slightly fewer steps to the solution (see the #EtTA column and discussion in *Section 5.6.3.2*).

Finally, note that all SE (TpE) and SE (TpS) are almost the same, as expected.

5.6.3.2 Number of Steps (and Epochs) to Target Accuracy

For all experiments, we see that SGD takes significantly more epochs (thus steps[41]) than any K-FAC variant. Surprisingly our proposed K-FAC improvements tend to take slightly less epochs than standard K-FAC[42]. Also, for all experiments, we see that SE (EtTA) of our proposed K-FAC improvements is slightly better than the one of K-FAC. However, the SE (EtTA) of SGD tends to be much better than the one of any K-FAC variant. Even so, our proposed K-FAC improvements always take fewer epochs than SGD, for our considered #GPUs values. This is ultimately what matters the most, as scaling efficiency is a concept relative to the performance at some #GPUs (better scaling efficiency is pointless if the performance is always worse). Of course,

[41]Each epoch has $\lceil |\mathcal{D}_{\text{train}}| / \left([\#\text{GPUs}] \cdot n_{\text{B/GPU}} \right) \rceil$ steps (and we fixed $n_{\text{B/GPU}} = 128$ throughout).

[42]Despite the fact that our proposed K-FAC improvements are hypothesized to have a worse quality preconditioner, since they only approximate the K-FAC step.

different conclusions might be drawn if we performed experiments for many more GPUs. Investigating this further with much higher #GPUs values is subject for future work. Comparing our SE (EtTA) values with values in the literature would be revealing. However, this metric is not used: we will discuss such a comparison for SE (TtTA), which is the metric[43] typically reported in the literature.

The SE (EtTA) of K-FAC variants particularly low for Imagenette and SVHN (when compared to SGD or with results for the other data sets). For Imagenette an explanation for this behaviour can be hypothesized. In short, the number of total steps to convergence was too low to begin with, due to relatively small dataset, and this may be the cause of low SE (EtTA). The number of steps per epoch at 1 GPU, 2 GPUs and 4GPUs (with $n_{\mathrm{B/GPU}} = 128$) is only 74, 37, and 19 respectively. This leads to about 3256, 2664, 2394 total steps to convergence for 1 GPU, 2 GPUs and 4 GPUs respectively for our K-FAC variants (using the #EtTA values). Doubling the #GPUs halves the number of steps per epoch while improving the gradient estimate. Intuitively, having so little number of total steps may mean the negative impact of reducing our total steps number (by halving the number of steps per epoch) is stronger than the positive impact of improving the gradient estimate. Thus, we needed to increase our number of epochs as more GPUs are added, resulting in low SE (TtTA). However, for SVHN there were 4141 steps per epoch at 1 GPU and the number of epochs to convergence was about 10 for our K-FAC variants, and so we cannot explain the low SE (#EtTA) in the same way. Further investigation into SE (#EtTA) with the aim of improving these values is future work.

5.6.3.3 Time to Target Test Accuracy

As we saw in *Section 5.6.1* (*Table 5.2*), for VGG16_bn_lmxp experiments our proposed K-FAC improvements have a TtTA about $3-4\times$ lower than standard K-FAC and SGD. We saw that this factor is only about $1.1\times$ for our resnet50 experiment. Nevertheless, for all our experiments, our proposed K-FAC improvements outperformed these two benchmarks in terms of TtTA, which is the most important fact in practice.

In *Section 5.6.3*, we have seen that $\mathrm{SE}_n(\mathrm{TtTA})$ can be decomposed as $\mathrm{SE}_n(\mathrm{TtTA}) = \mathrm{SE}_n(\mathrm{TpE}) \cdot \mathrm{SE}_n(\#\mathrm{EtTA})$. In *Section 5.6.3.1* we have seen that for our proposed K-FAC improvements $\mathrm{SE}_n(\mathrm{TpE})$ was typically about $0.89-1.12$, and $\mathrm{SE}_n(\#\mathrm{EtTA})$ was about $0.33-0.89$. Thus, we can easily infer that in general our $\mathrm{SE}_n(\mathrm{TtTA})$ was relatively low for our proposed K-FAC improvements. The same information is more accurately

[43]Though more practical, it is less revealing as it does not break down the total time based on step quality and step cost as we do.

presented in the ninth column of *Tables 5.3, 5.4, 5.5, 5.6*, where we see SE_n(TtTA) values of $0.75 - 1.02$ for 2 GPUs and $0.30 - 0.75$ for 4 GPUs. The deterioration of SE values with #GPUs is not ideal, and needs to be further investigated and improved.

Still, while the efficiency dropped, adding in further GPUs always provided a lower TtTA (for our experiments), which is ultimately the most practical metric[44]. Thus, even though adding in further computational resources might be relatively inefficient, we still reduce the total wall-time. Of course, we might obtain different behaviors at higher values of #GPUs (exploring this is future work).

Comparing SE Results to Literature: Cannot Directly Compare. Comparing our SE (TtTA) values to the ones in the literature is tempting, but needs to be thought of with care. In the literature, we find results like:

1. Distributed SENG ([20]) achieves 0.9-0.93 SE (TtTA) values for #GPUs values of 4, 8, 16, and 32 (the reference performance was taken to be the one at 4 GPUs, the problem was Imagenet-1k classification with resnet50).

2. A cutting-edge distributed K-FAC implementation (K-FAC-OPT, [52]) achieves 1, 0.92, 0.84, 0.72, and 0.48 SE (TtTA) values for #GPUs values of 16, 32, 64, 128, and 256 respectively (with reference performance at 16 GPUs, for Imagenet-1k with resnet50). Similarly to our results, the scaling efficiency degrades with higher #GPUs, while the range of SE values is also similar to ours.

However, the results in the literature are not directly comparable to ours: our results are for much smaller datasets (CIFAR10, CIFAR100, SVHN, Imagenette-fs-v2) and a much smaller #GPUs values (1, 2, and 4): and thus the important effects on scaling efficiency will be different. For much higher total batch size (and #GPUs) we tend to have deterioration mainly due to overfit ([51]) and communication cost growth ([52]). These effects are not dominant for our relatively low total batch size and #GPUs, but are more significant for the results in the literature.

Furthermore, note that the reference #GPUs has a crucial role in computing scaling efficiency. In particular, with the typical deteriorating efficiency behavior (present in both our numerical experiments and the [52] results) re-choosing the reference to be at a higher #GPUs values will increase the efficiency of all the larger #GPUs

[44]The SE (TtTA) metric must be interpreted with care. It is relative to the performance at some chosen #GPUs value, and is thus meaningless in the absence of an actual total run-time. We may have an algorithm with all SE (TtTA) values better, but which is always slower. To see this on an extreme artificial example, note that we can always construct algorithms with arbitrarily better SE (TtTA) by purposely introducing junk computations preferentially more at the lower #GPUs values.

values. For example, if we chose the reference #GPUs to be 32 (rather than 16) in the results of K-FAC-OPT ([52]), we would increase the previously found values of 0.92, 0.84, 0.72, and 0.48 to 1, 0.91, 0.78, and 0.52 respectively. Our results not only span different intervals of #GPUs values, but also have a different reference performance value (reference at 1 GPU rather than 16). The efficiency of our algorithms for the 16, 32, 64, 128 and 256 #GPUs values would get a boost from rechoosing our reference at 16 (as done in [52]). However, different scaling issues might arise at those high #GPUs values (deterioration due to overfit and growing communication costs), negatively impacting our SE (TtTA). Thus, it is hard to predict how our SE (TtTA) values would compare to the ones in the literature under the high #GPUs (high total batch size) and very large dataset regime.

We devolve *achieving a direct comparison between literature results and our algorithms for large #GPUs values* to future work.

SE Deterioration may not be due to our Proposed Improvements. Note that the SE deteriorates in a similar manner with higher #GPUs values for our distributed implementation of standard K-FAC as well (not just for the distributed K-FAC variants using our proposed improvements). Thus, the issue is probably more related to our distributed setting choices, or to K-FAC itself, rather than to our proposed inverse-update improvements. This idea is supported by two facts (but needs further investigation to be backed up concretely). First, the results in *Tables 5.3, 5.4, 5.5, 5.6* mostly show better SE (TtTA) for our proposed improvements than for standard K-FAC, owing to better SE (TpS) and better SE (EtTA). This suggests our proposed improvements may tend to increase the scaling efficiency (compared to the standard K-FAC approach). Secondly, other experiments in the literature observe a similar deterioration for different distributed implementations of (standard) K-FAC (see [51, 52]). However, using this second fact as an argument requires performing our experiments at the same scale as the ones in the literature (which is subject or future work). We have seen before that otherwise the results are not directly comparable, and thus we cannot draw conclusions.

5.6.4 Comparison between Different Proposed Variants

Our numerical results consist of 5 runs for each solver, for 12 settings: 3 #GPUs settings (1, 2 and 4) $\times$ 4 datasets (CIFAR10, CIFAR100, SVHN, Imagenette). These results are most extensively shown in *Tables 5.3, 5.4, 5.5, 5.6*. For each solver, and each of the metrics: EtTA, TpS, and TtTA, we now count the number of settings

when the average was the lowest among all 6 solvers. For TpS, we exclude SGD since it will always have the lowest TpS (but no preconditioner, and worse TtTA). We break ties based on variance, and if the variance was the same we divide the point to all winners equally. The results are shown in *Table 5.7*. We use this simple metric to infer which solver has the best EtTA, TpS and TtTA behavior. Of course, better metrics can be used[45], but we stick to the one stated for simplicity.

Table 5.7: Distributed Results Summary: Counting Wins.

		SGD	K-FAC	R	B	BR	BRC
Number of Times when the Optimizer had the Best:	**#EtTA**	0/12	0/12	0.83/12	2.5/12	**5.33/12**	3.33/12
	TpS	N/A	0/12	3/12	**9/12**	0/12	0/12
	TtTA	0/12	0/12	1/12	**6/12**	3/12	2/12

Best #EtTA. We see that BR-KFAC scores the most points for #EtTA, followed by BRC-KFAC. This was expected since the two algorithms perform more frequent updates (to K-factors of FC layers). Surprisingly, B-KFAC had a better score than R-KFAC in this metric.

Best TpS. B-KFAC scored the most TpS points, having the lowest TpS 9 out of 12 times. Again, this was expected from the algorithm design, as B-updating the large FC layers reduces significant time relative to R-updating them.

Best TtTA. Over-all, B-KFAC has the best TtTA 6 out of 12 times. We argued before that a low TtTA can be obtained either by reducing TpS or #EtTA. We saw that B-KFAC has a reasonable score in #EtTA as well, and also has the best score in TpS, so based on this, the result is justifiable.

5.7 Discussion

In this chapter, we proposed distributed implementations of our previously proposed K-FAC improvements: R-KFAC, B-KFAC, BR-KFAC and BRC-KFAC. We began by giving some context for the distributed setting choices we made, and then specified our choices (in *Sections 5.1* and *5.2*). We chose an LP K-factor computation (like in [73]) done during the forward and backward passes, and used COMM-OPT K-Factor inversion strategy (like in [52]).

We then described our distributed design in more detail (in *Section 5.3*). We proposed using an adaptive target-rank strategy which aims to maintain a desired

[45]For the metric we use, second place receives no points, even if it was very close to the winner. For example, this metric leads to few wins for R-KFAC in TtTA, though R-KFAC was many times very close to overtaking B-KFAC.

relative truncation error (rather than having a fixed truncation rank). We used a very simple practical heuristic to achieve this (in *Section 5.3.2*). We also described the general idea of how we practically achieve a good work allocation for each algorithm (in *Section 5.3.1*). We then looked at this in more depth on a case-by-case basis, placing special emphasis on understanding the workload distribution of each proposed algorithm in a distributed setting (in *Sections Sections 5.4.1 - 5.4.5*). The workload allocation we used involved certain simplifying approximations, and thus could be improved upon in the future. The understanding gained in *Section 5.4* represents the basis for such future improvements.

In *Section 5.5*, we specified all the details of our numerical experiments set-up, including hyperparameters tuning methodology, the hyperparameter values we finally settled upon, data sets, neural networks, and computational resources. Importantly, the K-FAC benchmark used the same distributed setting choices as our proposed K-FAC variants, for a fair comparison. This distributed K-FAC implementation is closest to the one in [73] (which is one of the best performing distributed implementations in the literature at present). Our maximum number of available GPUs was 4.

In *Section 5.6*, we presented some aspects of our numerical results and commented on these. We saw that our algorithms outperformed both the K-FAC and SGD benchmarks on all test cases. We looked at average convergence graphs and saw that when plotting against epoch number, the test accuracy curves of our proposed K-FAC improvements were very similar to the ones of K-FAC. This suggested our K-FAC improvements took steps very similar to the K-FAC step, owing to the relatively low target truncation error (3.3%) which the algorithms attempted to maintain. Because our proposed K-FAC improvements took less time per step, they reached the target accuracy in lower time, outperforming K-FAC. We saw that the speed-up factor was larger (3× - 4×) when the K-factors were large (for VGG16_bn_lmxp experiments), and smaller (1.1×) when the K-factors were smaller (for resnet50). This behavior was expected, as our proposed K-FAC variants improved the complexity of the preconditioning part by a power or two, which yields more speed-up for larger K-factors.

Finally, we looked at our scaling efficiency (with #GPUs), and saw that for all K-FAC algorithms (including the benchmark) this was sometimes low (also exhibiting the deteriorating SE with increasing #GPUs, similarly to literature results), and required improvement. We argued that this is not a major cause for concern as our proposed K-FAC variants always outperformed SGD. We also argued that since the K-FAC benchmark also exhibits a relatively low scaling efficiency, the issue is probably more related to our distributed setting choices, or to K-FAC itself, rather than to

our proposed improvements. We stated that experiments on more GPUs and larger datasets have to be carried out to examine, and possibly improve, scaling efficiency in the large scale distributed setting.

In conclusion, the distributed implementation of our proposed K-FAC improvements resulted in significant speed-ups over the standard K-FAC and SGD benchmarks, for every single test-case. This suggests our proposed K-FAC improvements are always preferable to standard K-FAC, but further numerical experiments are required to draw such a strong conclusion.

5.7.1 Future Work

The future work arising from this chapter's work can be summarized as follows:

1. Improve our LP K-factor computation strategy (particularly relevant for large #GPUs; recall *Section 5.2.1*). Note that when the number of GPUs is large enough that the K-factor work cannot be further divided (under our current implementation), further adding in GPUs does not improve the average time per step. In this situation, we would like to have the idling GPUs compute duplicates of K-Factors (on different data), such that new GPUs improve the K-factor estimate quality. Designing such a distributed strategy that is based on this idea (particularly relevant to large #GPUs) is a subject for future work. This would effectively be a hybrid between LP and HP K-factor computation strategies.

2. Improve our adaptive target-rank heuristic given in *Algorithm 5.1*.

3. Improve B-KFAC work allocation (recall *Section 5.4.3*): consider storing the $A_k^{(l)}$ and $G_k^{(l)}$ matrices and performing the B-update during the R-update of the R-updated K-factors phase.

4. Improve the BR-KFAC and BRC-KFAC work allocations: for each algorithm and its specific workload profile, consider an allocation algorithm which minimizes the wall-time of the whole run, rather than using the greedy and approximate approach suggested here (recall *Sections 5.4.4* and *5.4.5*). These allocation algorithms would be closer to optimal allocation by also using the important periods (T_{Cov}, T_{Inv}, T_{Brand}, T_{BR}, etc) as an input, unlike before.

5. Implement and examine distributed versions of our K-FAC variants where we *cache the required quantities during the forward and backward passes, and compute K-factors later* (see *Section 5.1*).

6. Perform a better hyperparameter search for our proposed K-FAC improvements, in particular for BR-KFAC and BRC-KFAC (recall *Sections 5.5.2 - 5.5.4*).

7. Run numerical experiments for much larger #GPUs values (recall *Section 5.5.1*), and Imagenet-1k (possibly changing algorithms accordingly).

    - Other distributed K-FAC implementations take 35-45 epochs to train ImageNet to 75.9% test accuracy (with resnets) [51, 53, 71, 73, 106]. In contrast, typical distributed SGD implementations[46] take 68 epochs with carefully tuned hyperparameters [73]. Since our proposed K-FAC improvements have negligible overhead over SGD for resnet50 (as we saw in the numerical experiments: review TpS values in *Table* 5.6), but took significantly fewer epochs in general, we are hopeful that implementing our proposed K-FAC improvements for this setting will result in significant speed-ups over SGD, and notable speed-ups over other K-FAC implementations.

In the *Future Directions Section of the Conclusions Chapter*, we discuss how future work relating to this chapter integrates with the general future directions of the thesis.

[46] Though there are tricks to reduce number of epochs for SGD, such as using a Layer-wise Adaptive Rate Scaling (LARS) learning rate ([108, 109]): which uses different learning rates for different layers. LARS has been successfully used with distributed SGD to reach 75.9% test accuracy on ImageNet-1k classification with resnet50 in 38 epochs [107].

However, there are also very large scale (with very large total batch-sizes) implementations which take about 90 epochs. For example, training Imagenet-1k with resnet50 in 74s with SGD on 2048 GPUs with 81,920 batch size takes about 90 epochs (see appendix of [105]).

Chapter 6

Conclusions and Future Work

Recent research in Optimization for Deep Neural Nets (DNNs) has lately focused on Natural Gradient algorithms. This research direction was mainly driven by two aspects. On one hand, preconditioning the steepest descent direction is provably useful in classical optimization. Being simpler to approximate than the Hessian Matrix, the Fisher Information Matrix (which only really needs first-order information) naturally arises as the first preconditioner one should try. On the other hand, Natural Gradient Descent (NGD) has provably better properties than SGD, the most notable of which is reparameterization invariance, although holding only for infinitesimally small step-sizes (see *Appendix A2.2* for a discussion with relevant references).

Although the Fisher only uses first-order information, practical NGD implementation for DNNs is problematic: off-the-shelf NGD is computationally infeasible, due to the sheer size of the Fisher (as we saw in *Sections 1.4 - 1.5*). Thus, one has to resort to (relatively crude) approximations. There exist many such approximations, and the practical implementations tend to depend on layers' type (which are directly determined by data-type) and on the ML paradigm (see our extensive literature survey in *Appendix A5* for a complete discussion with references).

The most popular and successful practical implementations of NG for DNNs are K-FAC ([1]) and SENG ([20]), as we argued in *Section 1.3.1*. Despite worse scaling in layer width, K-FAC is much more widely used than SENG (which is more recent), with variants across a much wider spectrum of layer-types and ML paradigms (eg. recurrent nets [30], RL: algorithm ACKTR [46], VI for BNNs [47, 48, 49], and more).

K-FAC works very well for low-width nets, with per-epoch time roughly the same as SGD (as we argued in *Section 1.3.1*). However, it has cubic time-complexity in layer width, which is a crucial bottleneck for large layer widths. SENG, on the other hand has quadratic time-complexity in net width, which is remarkable (recall *Section 1.3.1*). Since both K-FAC and SENG aim to take the same step, and they do so with

the same information[1], the time-complexity difference raises a conceptual question: is K-FAC inherently worse? Or is it that wasteful computations are much more obviously exposed and thus eliminated when taking the SENG approach?

Reducing K-FAC's time-complexity in layer size thus becomes of both conceptual interest (answering the question above), as well as of practical interest (get practically faster algorithms). In *Chapters 3 - 4* we proposed improvements to K-FAC which reduce the time-complexity in layer size (recall *Definition 1.1*) from cubic to: (1) quadratic[2] (in *Chapter 3*); (2) linear (*Chapter 4*). We thus answered *Research Question 1.*

In ***Chapter 3***, we began by theoretically showing that the exponentially-averaged (EA) K-Factors must exhibit a certain type of eigenspectrum decay, owing to EA construction paradigm. In particular, we saw that the decay is such that we can control the K-factor representation error below a fixed threshold ($\epsilon \in (0, 1)$), by only representing the K-factors with a constant number of (leading) eigenmodes as the layer size increases past a "sufficiently large" value. This statement holds for any $\epsilon \in (0, 1)$ (recall *Section 3.1*).

These theoretical properties of the EA K-factors eigenspectrum led to the idea that we should directly obtain a constant number of leading modes (for any layer size). If we ignored the aspect of controlling the projection error, doing this would reduce the time-complexity of obtaining our K-factors inverse representation by a power (relative to standard K-FAC which uses EVD), since we would compute a constant number of modes in layer size, rather than linearly many of them. We proposed to achieve our desire to directly obtain the leading modes with Randomized Numerical Linear Algebra (rNLA; recall *Section 2.2*). This gave rise to the *Randomized K-FACs* algorithms (see *Section 3.2*).

Notably, using rNLA to get the leading modes introduces a further error (on top of the optimal truncation error that our theory in *Section 3.1* is constructed on). However, we saw that with high probability, this total error of our rNLA-obtained rank-r representation is still within a small constant (multiplicative) factor ($c > 1$) of the optimal rank-r truncation error (see *Section 3.2.4*). This, however, happens only if we choose to perform $\mathcal{O}(\log(d_M))$ power iterations, which worsens the complexity to $\mathcal{O}(\log(d_M)d_M^2)$ (note the $\log d_M$ factor). Over-all, we could still control the total error as our theory in *Section 3.1* tells us, but we had to choose an r ensuring an optimal truncation error smaller by a factor of c than our desired error level.

[1]Though SENG relies on empirical NG, K-FAC can be empirical too.

[2]Or "almost quadratic" ($\mathcal{O}(\log(d_M)d_M^2)$) when controlling the error (recall *Section 3.2.4*).

Naively applying (as in standard K-FAC) the proposed low-rank K-factor inverse-representations would still be of cubic time-complexity in layer size (see *lines 13 - 15 of Algorithm 1.1*). Therefore, to obtain an over-all $\tilde{\mathcal{O}}(d_M^2)$ time-complexity for our *Randomized K-FACs*' preconditioning (thus answering *Research Question 1.1*), we changed the inverse-application subroutine, as showed in *lines 7 - 10 of Algorithm 3.1*.

Importantly, in *Section 3.2.3*, we outlined that we could choose to do one of two things when directly obtaining the leading r modes (with rNLA). (1) Either control the error, and accept whichever time-complexity this may give. Practically, this could be achieved by measuring error-proxies and adjusting the target-rank r in a reactive way to reach the desired error-proxy value[3]. A theoretical approach to controlling the error would be achieved by choosing r as the theory implies[4] for a desired ϵ. The practical numerical approach was preferable, since loose theoretical bounds might imply we attempt to maintain too many modes. (2) Or, give up on the idea of controlling the error, and enforce our desired quadratic complexity (by fixing target-rank r for any layer size and the number of power iterations in RSVD). Recall *"The Approach in this Chapter: Guessing an r"* of *Section 3.2.3*.

Our theory in *Section 3.1* tells us we can obtain both benefits (controlling the error and improved time-complexity) together for sufficiently large layer size. However, *a limitation of our theory* was that it cannot guarantee this sufficiently large layer regime is reached in standard practical settings, as showed in our theoretical case-studies in *Section 3.1.3.1*. Nevertheless, our numerical case-study of *Section 3.1.3.2* showed that most K-factors practically fell in this desired regime, and the few ones that did not still allowed for some computational savings. This raised hope that we could achieve both error-control and improved time-cost in practice, which would ultimately translate to much better run-times to a certain target-accuracy.

Our numerical results (in *Section 3.3*) showed this was indeed the case. Our proposed *Randomized K-FACs* obtained significant speed-ups $\approx 3\times$ for CIFAR10 classification with VGG16_bn in single-GPU settings.

In ***Chapter 4***, we noted that recomputing the K-factors inverse-representations from scratch every time (like standard K-FAC and *Randomized K-FACs* do) is wasteful, considering their EA construction paradigm. We thus leveraged *Online Numerical Linear Algebra (oNLA)*, in particular Brand's algorithm (recall *Section 2.3*; [74]), to

[3]We have stated this idea under *"Choosing r based on practical truncation error proxy: targeting ϵ error."* in *Section 3.2.3*, but only implemented it numerically in the context of the distributed versions of *Chapter 5* - recall *Section 5.3.2*.

[4]See *"Choosing r based on theory: Not practically useful"* of *Section 3.2.3*.

propose a K-factor inverse-representation construction algorithm that takes *linear time in the layer width*. We called this the Brand-update (or B-update), and the main idea was to recycle the previously held inverse representation to obtain computational savings (recall *Section 4.1.1*). We pointed out the caveat that this B-update can only be successfully applied only if the layer size is sufficiently large. For practical settings, we saw this is typically the case for FC layers (recall *"A Caveat for B-update Usability"* and *"We only apply B-update to FC layers in this Chapter"* in *Section 4.1.1*).

Having the B-update was not enough. Care had to be taken to produce a useful algorithm with linear-time complexity of the preconditioning part (to answer *Research Question 1.2*). In particular, there were two main concerns which we had to solve.

The *first concern* was that repeated B-update applications result in a rank-growth of the B-updated K-factor representation over iterations. This would increase the time-cost of the algorithm towards ultimately reaching the cost of an EVD (an undesirable behavior). To solve cost-growth issue, we proposed using a rank-r SVD (optimal) truncation just before each B-update (recall *Section 3.1.1*).

The *second concern* was that achieving a linear time-complexity for the whole preconditioning part required a linear-time inverse application as well (not just a linear-time inverse computation). The best previously proposed inverse application was of quadratic time. To solve this problem, and obtain a K-FAC improvement with an over-all linear time-complexity of the preconditioning part, we also proposed a linear time-complexity inverse application subroutine (in *Section 4.3*). This subroutine can be used whenever the B-update can be. By solving *these two concerns*, we were able to obtain the algorithm that we wanted. The B-update, together with the truncation and the linear-time inverse application subroutine gave the B-KFAC algorithm.

Importantly, we also saw that if used exclusively (i.e. without combining with an *R-* or *C-update*), the B-update has a huge advantage: it is low-memory: we never need to form/store the (EA) K-factors, but only a much smaller-matrix which approximates the first few eigenmodes (recall *Sections 4.1.1* and *4.4* for details). Thus, our proposed B-KFAC algorithm has linear memory-complexity[5] in layer size (compare to quadratic for all other proposed K-FAC variants and standard K-FAC).

Despite accomplishing our goal to *propose a linear time-complexity* K-FAC *variant*, as desired (thus affirmatively answering *Research Question 4*), there were some further issues to be faced. In particular, our solution to the *first concern* above came at the huge cost of introducing a challenging error (to deal with) in our B-updated K-factors

[5]Speaking about the preconditioning part of the algorithm only - thus excluding the gradient part (i.e. the "SGD" part), which is always of quadratic memory-complexity.

representations. In particular, though the B-update is exact, the introduction of our truncation gives only greedily optimal[6] rank-r representation. Thus in general, our B-updated rank-r representation will not be optimal: that is, *the total error* B-KFAC *will generally exceed the optimal truncation error* (recall *Section 4.1.1* for details).

From this point, two natural questions arose. *Firstly*, whether we can attain an error level we can tolerate in practice; and *secondly* if so, how to choose our target-rank r to have the desired error level. Unlike in *Chapter 3* we do not respond to these questions with theoretical results. As such, the excess error of our rank-r B-KFAC representation (over the optimal rank-r truncation error) was not theoretically characterized, and thus unknown. As a consequence to this, we had to accept having an unknown K-factor representation error (unless measured numerically) in our B-KFAC algorithm, and use r values implied by our discussion in *Chapter 3* as our best guess.

To aid with the control of this unknown excess error, we proposed a few algorithmic solutions. While these solutions were agnostic of the actual excess error, numerical results in *Section 4.2.3* showed they work well (at least for the case-study therein). In particular, our solutions were to change using B-KFAC (of *Section 4.1.1*) with using BR-KFAC (of *Section 4.1.3*), BC-KFAC (of *Section 4.1.4*), or BRC-KFAC (of *Section 4.1.5*). Notably, our algorithmic error-control solutions came at the price of losing the low-memory property of B-KFAC, as well as now having *linear time-complexity for most iterations* but *quadratic time-complexity for some iterations* (recall *Section 4.4*).

Our first proposed algorithmic error-control solution (BR-KFAC in *Section 4.1.3*) was motivated by theoretically showing in *Section 4.1.2* that extreme error events can occur in B-KFAC, at least in the limited 2D setting considered therein. In particular, under these extreme error events the total B-KFAC error would significantly exceed the R-KFAC error (for the same target-rank r) for a significant number of consecutive iterations. Though these events may occur with low probability, this is not negligible. Periodically using an R-update (recall *"Definition of the R-Update"* in *Section 3.2.1*) in a B-KFAC algorithm is thus used to mitigate the effect of such extreme error events, should these occur. This gave rise to the BR-KFAC algorithm.

Our second proposed algorithmic error-control solution (BC-KFAC in *Section 4.1.4*) replaces the periodic R-update with a periodic C-update, where the C-update is a lighter correction (to the current K-factor representation) that we proposed (recall *Algorithm 4.3*). Importantly, we theoretically showed that the proposed correction is preferable to not performing the correction (in *Proposition 4.3* and *Corollary 4.4*) from

[6]It ensures the best rank-r truncation at the current step with the available representation. But the available representation is also erroneous because of previous truncations. Recall *Section 4.1.1*.

the point of view of the iteration at which the correction it is performed. While the effect of the correction on subsequent iterations is yet to be investigated, our stated theory and numerical results in *Section 4.2.3* are enough to strongly recommend using the C-update to control the B-KFAC error.

Our third algorithmic error-control solution was BRC-KFAC (see *Section 4.1.5*). This is a hybrid between BR-KFAC and BC-KFAC, having them as extreme cases.

Having proposed a BR-KFAC algorithm, a natural desire to compare the errors of BR-KFAC and R-KFAC theoretically arises. We did this in *Section 4.2.1* by noting that BR-KFAC can also be seen as an R-KFAC algorithm to which we introduce B-updates at times when no RSVD would have been performed. This view is in stark contrast to seeing BR-KFAC as a corrected (or improved) B-KFAC. We theoretically argued that (at least) when comparing the worst-case scenarios, using a B-update is better than using no update (recall *Proposition 4.3* and *Corollary 4.4*). This suggested that our observed worst-case representation error of BR-KFAC should be better than the one of R-KFAC (at least for sufficient run-time). Our numerical results in *Section 4.2.3* were indeed consistent with this observation (but stronger).

Numerically, the speed-up from *Randomized K-FACs* to *Brand New K-FAC* is notable, but modest. However, this comes from the fact that our B-update was only applicable for FC layers, and the net we tested on only had 1 FC layer. Furthermore, the linear time-complexity inverse-application routine was not implemented (the quadratic one of *Chapter 3* was used). Implementing this would give further speed-ups.

In ***Chapter 5*** we proposed distributed implementations of previously proposed algorithms, describing our proposed distributed design in detail. We outlined all choices that need to be made for distributed designs, and stated our choices with justifications (recall *Sections 5.1 - 5.3*). We also looked at details of the distributed-setting workload distribution of each K-factor improvement proposed in *Chapters 3* and *4*, excepting for SRE-KFAC and BC-KFAC, which were not considered (recall *Section 5.4*). We leveraged this understanding to propose (relatively) efficient (but not necessarily optimal) work-allocation paradigms, in *Section 5.4*.

We also proposed a concrete adaptive target-rank strategy, which aimed to practically maintain K-factor representation error below the desired threshold (recall *Section 5.3.2*). We first outlined such an approach is possible in the *"Choosing r based on practical truncation error proxy: targeting ϵ error"* paragraph of *Section 3.2.3*, but we did not implement it in that chapter. Our proposed adaptive target-rank strategy in *Algorithm 5.1* is relatively simple (and can be improved), but worked well in practice.

Numerical results in *Section 5.6* showed speed-up factors (w.r.t. standard K-FAC, based on time to target accuracy) of 3× - 4× when the K-factors were large, and of 1.1× when the K-factors were smaller, a behavior consistent with our algorithms having a better time-complexity. The speed-up factors w.r.t. distributed SGD (with momentum) were 1.8 - 6×. We thus affirmatively answered *Research Question 3*. Our scaling efficiencies were reasonable, but their degradation rate with increasing #GPUs was slightly concerning, and has to be further explored and addressed in the future.

Having proposed five different K-FAC improvements, prompts us to give a **selection guide** for the reader (or user). We did this implicitly, in two steps. First, we offered a conceptual overview of all these proposed variants from multiple perspectives in *Section 4.4*. This was done for the single-GPU setting. Then, we offered numerical results to complement the conceptual overview. In *Section 4.5* we compared all proposed variants numerically in a single-GPU setting. A more complete discussion, based on improved and distributed algorithms versions, as well using more test-cases was presented in *Section 5.6.4*. These sections together help the reader make the desired selection for the situation at hand.

From a **conceptual point of view**, *Chapter 3* proposes a way to close the time-complexity gap between SENG (quadratic in layer width) and K-FAC (cubic), while *Chapter 4* proposes ways to take this K-FAC improvement even further with the aim of outperforming SENG. We do this by first turning K-FAC into a quadratic[7] time-complexity algorithm in *Chapter 3*, and then further reduce the complexity of the preconditioning part down to linear (in *Chapter 4*). From a **practical point of view**, our approaches translate to very strong speed-ups relative to standard K-FAC ([1, 29]).

Our quadratic time-complexity K-FAC improvements (R-KFAC, BR-KFAC, BC-KFAC, BRC-KFAC) provide speed-ups (in terms of time to target accuracy) of: (1) $\approx$ 2.5 - 3.5× for CIFAR10 with VGG16_bn (and modifications: VGG16_bn_lmxp, recall *Section 4.4*) for single GPU settings (recall *Sections 3.3, 4.5*); (2) $\approx$ 2.5 - 3.5× for CIFAR10, CIFAR100, SVHN datasets with VGG16_bn_lmxp for 1 - 4 GPUs distributed setting (recall *Section 5.6*); and (3) $\approx$ 1.1× for Imagenette-fs-v2 with resnet50 for 1 - 4 GPUs distributed setting (recall *Section 5.6*). Over-all, these numerical results provide an affirmative answer to *Research Question 2*.

Our proposed linear time- and memory-complexity K-FAC variant (B-KFAC) always obtained better per-step time and expended less memory (as it was designed to) than the other proposed variants. However due to lower accuracy in representing

[7]The complexity worsens by a multiplicative log factor when error is controlled.

the K-factors, the speed-up factor (in terms of time to target accuracy) was the same on average as the one of the quadratic (or "almost quadratic") time-complexity K-FAC improvements (with better factors in some situations and worse in others; recall *Sections 4.5* and *5.6*). Still, B-KFAC is superior in low-memory situations, as it enables using K-FAC when memory might prohibit the practitioner from doing so.

Finally, our **proposed improvements**[8] **are widely applicable** in practical situations. The ideas proposed in *Chapter 3* are directly applicable in any setting where K-FAC is implemented. We have seen in the *"Natural Gradient Literature Survey"* paragraph of *Section 1.3.1* that there are many applications for K-FAC, for example: FC layers/nets (the main trainable structure behind Natural Language Processing, [1]), Conv layers/nets (the main trainable structure behind computer vision tasks, [29]), recurrent nets ([30]), reinforcement learning ([46]), Variational inference for Bayesian Neural Networks ([47, 48, 49]). A more recent (and very successful) practical application of K-FAC is in the context of *Deep Kernel Shaping*, in which case K-FAC enables the training of skip-free batch-normalization-free DNNs at the same speed as standard training of Imagenet with Resnets (which was previously unheard of) [50].

The ideas in *Chapter 4* are directly applicable only if certain shape constraints are satisfied by the K-factors (which surely hold for sufficiently large layer size). We saw that in practice it is FC layers that typically satisfy this shape constraint (recall *"A Caveat for B-update Usability"* in *Section 4.1.1*). Though this may seem like a limiting the applicability of our proposed improvement significantly, that is not the case. Importantly, most state of art DNNs contain FC layers for both computer vision (CV) and natural language processing (NLP) tasks. In particular, FC layers are the only trainable components of state-of-art large language models for NLP, which are based on Transformers architecture ([24]), like BERT ([4]). Furthermore, any Conv net used for CV tasks always contains at least one FC (relatively wide) layer.

Future Directions

To further advance the research outlined in this thesis, there are five main future directions (strands of work) to be followed[9]. These are:

1. **Better Understand the Proposed Algorithms Properties.** For this, we propose to: *(1) Better explore (theoretically and numerically) for which K-factors sizes to use the B-update*; *(2) Generalize Examples 4.1 - 4.2 to a broader theoretical setting to better understand* BR-KFAC *error* (recall *Section 4.6.1*).

[8] Which we have seen provide substantial benefits.

[9] Obtained by grouping the future work arising in *Chapters 3 - 5* (see *Sections 3.4.1*, *4.6.1*, *5.7.1*).

2. **Improve Algorithms Error Control.** While we have good theory informing our error control for R-KFAC, we do not have similar theoretical results for the other proposed algorithms. For those we rely on error-agnostic algorithmic error control procedures (for *Brand New K-FACs*) or numerical studies (SRE-KFAC) to understand and/or control the error. To fix this, we propose as future work to: *(1) Obtain error control theory for* SRE-KFAC; and *(2) Obtain theoretical error guarantees for a rank-r B-process.* In addition, *(3) Improving tightness of bounds in Section 3.1* will also help. See *Sections 3.4.1* and *4.6.1* for details.

3. **Provide Better Selection Guidelines for Proposed Algorithms.** This can be done through *(1) further numerical studies.* Theoretical investigations, such as for example *(2) Improving theoretical error comparison between* R-KFAC *and* BR-KFAC *(of Section 4.2.1)*; or *(3) Obtaining other similar theoretical error comparisons* would help. See *Section 4.6.1* for details.

4. **Improve Practical Implementations or Develop New Algorithms.** For *Brand New K-FACs*, *(1) implementing the proposed Inverse application methodology in Section 4.4* is the starting point. Then, we would like to *(2) Improve our adaptive target-rank heuristic given in Algorithm 5.1* to obtain better practical error-cost trade-offs; *(3) Develop and implement* RC-KFAC *(see Section 4.1.5), theoretically investigate its error, and numerically investigate its optimization performance.* Finally, we want to improve our distributed algorithms by: *(4) Improving the current LP K-factor computation strategy in the many-GPUs regime*; *(5) Improving the current distributed* B-KFAC*,* BR-KFAC *and* BR-KFAC *work allocations*; *(6) Performing a more in-depth hyperparameter search for distributed* BR-KFAC *and* BRC-KFAC; (*7) Implement and examine distributed versions of our* K-FAC *variants where we cache the K-factor required quantities during the forward and backward passes, and compute K-factors later.* See *Sections 4.6.1* and *5.7.1* for details.

5. **Obtain More Numerical Results.** Our main effort here should go towards *(1) Obtaining numerical experiments for clusters with many more GPUs*; and *(2) Implementing ideas and obtaining numerical experiments for other DL settings, such as: Reinforcement Learning, Natural Language processing*[10]*, and Variational Inference for Bayesian Neural Nets.*

[10]For NLP, one issue is that updates to FC layers will never be low-rank for K-FAC (or any related method like Shampoo [55] or TNT). In this case, we would need to handle this problem. We omit specific details of how this may be approached but ensure the reader this is possible.

References

[1] Martens, J.; Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature, *In: arXiv:1503.05671* (here referencing the 8 June 2020 version), originally 2015.

[2] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; Driessche, v. d. G.; Graepel, T.; Hassabis, D. Mastering the game of Go without Human Knowledge, Nature, 2017.

[3] Wang, X.; Song, J.; Qi, P.; Peng, P.; Tang, P.; Zhang, W.; Li, W.; Pi, X.; He, J.; Gao, C.; Long, H.; Yuan, Q. SCC: an efficient deep reinforcement learning agent mastering the game of StarCraft II, arXiv:2012.13169, 2020.

[4] Devlin, J.; Chang, M.-W.; Kenton, L.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, arXiv:1810.04805, 2018.

[5] Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language models are unsupervised multitask learners, 2019.

[6] Zhao, Z.-Q.; Zheng, P.; Xu, S.; Wu, X.; Object Detection with Deep Learning: A Review, 2019.

[7] Rombach, R.; Blattmann, A.; Lorenz, D.; Esser, P.; Ommer, B. High-Resolution Image Synthesis with Latent Diffusion Models. CVPR 2022 (ORAL).

[8] Yang, Y.; Training Massive Deep Neural Networks in a Smart Contract: A New Hope, arXiv2106.14763, 2021.

[9] Bottou, L.; Curtis, F. E.; Nocedal, J. Optimization methods for large-scale machine learning, 2018.

[10] Goodfellow, I.; Bengio, Y.; Courville, A. Deep Learning, An MIT Press book.

[11] Ba, J.; Kingma, D. Adam: A method for stochastic optimization, ICLR, 2015.

[12] Zeiler, M. D. ADADELTA: An adaptive learning rate method, arXiv:1212.5701, 2013.

[13] Reddi, J.; Kale, S.; Kumar, S.; On the convergence of ADAM and beyond, *In:Google New York*, 2018.

[14] Nocedal, J.; Wright, S. J. Numerical Optimization, *Springer Series in Operations Research, Second Edition and Financial Engineering*, 2006.

[15] Martens, J. Deep learning via Hessian-free optimization. In Proceedings of the 27th International Conference on Machine Learning (ICML), 2010.

[16] Schraudolph, N. N.; Yu, J. A stochastic quasi-newton method for online convex optimization, in Proceedings of 11th International Conference on Artificial Intelligence and Statistics, 2007.

[17] Berahas, A. S.; Jahani, M.; Takac, M. Quasi-Newton Methods for Deep Learning: Forget the past, just sample, arXiv:1901.09997, 2021.

[18] Cai, T.; Gao, R.; Hou, J.; Chen, S.; Wang, D.; He, D.; Zhang, Z.; Wang, L. A Gram-Gauss-Newton method learning overparameterized deep neural networks for regression problems, 2019.

[19] Le Roux, N.; Manzagol, P.; Bengio, Y. Topmoumoute online natural gradient algorithm, Advances in Neural Information Processing Systems 20, 2007.

[20] Yang, M.; Xu, D; Wen, Z.; Chen, M.; Xu, P. Sketchy empirical natural gradient methods for deep learning, *In:arXiv:1503.05671*, 2021.

[21] Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; Monfardini, G. The Graph Neural Network Model, in IEEE Transactions on Neural Networks. 20 (1): 61-80, doi:10.1109/TNN.2008.2005605, ISSN 1941-0093, 2019.

[22] Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning, pp. 448-456, 2015.

[23] Gholamalinezhad, H.; Khosravi, H. Pooling Methods in Deep Neural Networks, a Review, arXiv:2009.07485, 2020.

[24] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; Polosukhin, I. Attention Is All You Need, arXiv:1706.03762, 2017.

[25] He, K.; Zhang, X.; Sun, J. Deep Residual Learning for Image Recognition, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[26] Zisserman, A.; Simonyan, K. Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv:1409.1556, 2014.

[27] Rumelhart, D. E.; Hinton G. E.; Williams R. J. Learning representations by back-propagating errors, in Nature, 1986.

[28] Sutton, R. S.; McAllester, D.; Singh, S.; Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In Advances in Neural Information Processing Systems 12, 2000.

[29] Grosse, R.; Martens J. A Kronecker-factored approximate Fisher matrix for convolution layers, arXiv:1602.01407, 2016.

[30] Martens, J.; Ba, J.; Johnson, M. Kronecker-factored curvature approximations for recurrent neural networks, ICLR, 2018.

[31] Wu, Y.; Mansimov, E.; Liao, S.; Grosse, R.; Ba, J. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation, in Advances in Neural Information Processing Systems, 2017, pp. 5279-5288.

[32] Beltiukov, B. Optimizing Q-learning with K-FAC Algorithm, Springer International Publishing, 2020. Code URL: https://github.com/maybe-hello-world/qfac.

[33] Tieleman, T.; Hinton, G. Lecture 6.5. RMSPROP: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[34] Pearlmutter, A., B. Fast exact multiplication by the Hessian, in Neural computation 6.1, 1994.

[35] Martens, J.; Sutskever, I. Training deep and recurrent networks with Hessian-free optimization. In Neural Networks: Tricks of the Trade, pp. 479-535. Springer, 2012.

[36] Vinyals, O.; Povey, D. Krylov subspace descent for deep learning. In AISTATS, 2012.

[37] Hestenes, M. R.; Stiefel, E. Methods of Conjugate Gradients for Solving Linear Systems, *Journal of Research of the National Bureau of Standards. 49 (6): 409. doi:10.6028/jres.049.044*, 1952.

[38] Li, R.-C. Sharpness in rates of convergence for CG and symmetric Lanczos methods. Technical Report 05-01, Department of Mathematics, University of Kentucky, 2005.

[39] Mortiz, P.; Nishihara, R.; Jordan, M. I. A Linearly-convergent stochastic L-BFGS algorithm, arXiv:1508.02087, 2016.

[40] Gower, R.; Hanzely, F.; Richtarik, P.; Stich, S. U. Accelerated stochastic matrix inversion: General theory and speeding up BFGS rules for faster second-order optimization, Advances in Neural Information Processing Systems, Curran Associates, Inc., 2018.

[41] Goldfarb, D.; Ren, Y.; Bahamou, A.;. Practical Quasi-Newton methods for training deep neural networks, arXiv:2006.08877, 2021.

[42] Botev, A.; Ritter, H.; Barber D. Practical Gauss-Newton optimisation for deep learning, arXiv:1706.03662, 2017.

[43] Henriques, J. F.; Ehrhardt, S.; Albanie, S.; Vedaldi, A. Small steps and giant leaps: Minimal Newton solvers for deep learning, *In: arXiv:1805.08095*, 2018.

[44] Le Roux, N.; Fitzgibbon, A. W. A fast natural newton method. In ICML, pp. 623-630, 2010.

[45] Desjardins, G.; Simonyan, K.; Pascanu, R.; Kavukcuoglu, K. Natural neural networks, in Advances in Neural Information Processing Systems, pp. 2071-2079, 2015.

[46] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In Advances in neural information processing systems, pages 5285-5294, 2017.

[47] Zhang, G.; Sun, S.; Duvenaud, D.; Grosse, R. Noisy Natural Gradient as Variational Inference, 2017.

[48] Khan, M. E.; Nielsen, D. Fast yet Simple Natural-Gradient Descent for Variational Inference in Complex Models, arXiv:1807.04489v2, 2018.

[49] Khan, M. E.; Lin, W. Conjugate-Computation Variational Inference : Converting Variational Inference in Non-Conjugate Models to Inferences in Conjugate Models, 2017.

[50] Martens, J.; Ballard, A.; Desjardins, G.; Swirszcz, G.; Dalibard, V.; Sohl-Dickstein; J.; Schoenholz, S. S. Rapid training of deep neural networks without skip connections or normalization layers using Deep Kernel Shaping, arXiv preprint arXiv:2110.01765, 2021.

[51] Osawa, K.; Yuichiro Ueno, T.; Naruse, A.; Foo, C.-S.; Yokota, R. Scalable and practical natural gradient for large-scale deep learning, arXiv:2002.06015, 2020.

[52] Pauloski, J. G.; Zhang, Z.; Huang, L.; Xu, W.; Foster, I. T. Convolutional neural network training with distributed K-FAC. In International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-14, 2020.

[53] Pauloski, J. G.; Huang, Q.; Huang, L.; Venkataraman, S.; Chard, K.; Foster, I. T.; Zhang, Z. KAISA: An Adaptive Second-order Optimizer Framework for Deep Neural Networks. International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21). Association for Computing Machinery, New York, NY, USA, Article 13, pp 1-14, 2021.

[54] Yang, M.; Xu, D.; Cui, Q.; Wen, Z.; Xu, P. NG+ : A Multi-Step Matrix-Product Natural Gradient Method for Deep Learning, arXiv:2106.07454, 2021.

[55] Gupta, V.; Koren, T.; Singer, Y. Shampoo: Preconditioned Stochastic Tensor Optimization, arXiv preprint arXiv:1802.09568, 2018.

[56] Anil, R.; Gupta, V.; Koren, T.; Regan, K.; Singer, Y. Scalable second order optimization for deep learning. arXiv preprint arXiv:2002.09018, 2020.

[57] LeCun, Y.; Bottou, L.; Orr, G.; Muller, K. Efficient backprop. Neural networks: Tricks of the trade, pages 546-546 (1998).

[58] Schaul, T.; Zhang, S.; LeCun, Y. No more pesky learning rates. In ICML (2013).

[59] Park, H.; Amari, S.-I.; Fukumizu, K. Adaptive natural gradient learning algorithms for various stochastic models. Neural Networks, 13(7):755-764 (2000).

[60] Polyak B. T.; Juditsky, A. B. Acceleration of stochastic approximation by averaging. SIAM J. Control Optim., 30(4), July 1992.

[61] Amari, S. I.. Natural gradient works efficiently in learning, Neural Computation, 10(20), pp. 251-276, 1998.

[62] Pauloski, J. G.; Zhang, Z.; Huang, L.; Xu, W.; Foster, I. T. Convolutional Neural Network Training with Distributed K-FAC. International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20). IEEE Press, Article 94, pp 1 -14., 2020.

[63] Murray, M.; Abrol, V.; Tanner, J. Activation function design for deep networks: linearity and effective initialisation, in arXiv:2105.07741, (2021).

[64] Martens, J. New insights and perspectives on the natural gradient method, arXiv:1412.1193, 2020.

[65] Zhang, G.; Li, L.; Nado, Z.; Martens, J. ; Sachdeva, S.; Dahl, G. E.; Shallue, C. J.; Grosse, R. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model, arXiv preprint arXiv:1907.04164, 2019.

[66] Schoenholz, S. S; Gilmer, J.; Ganguli, S.; Sohl-Dickstein, J. Deep information propagation, In International Conference on Learning Representations, 2017.

[67] Daniely, A.; Frostig, R.; Singer, Y. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. Advances In Neural Information Processing Systems, 29:2253-2261, 2016.

[68] Poole, B.; Lahiri, S.; Raghu, M.; Sohl-Dickstein, J.; Ganguli, S. Exponential expressivity in deep neural networks through transient chaos. Advances in neural information processing systems, 29:3360-3368, 2016.

[69] Polyak, B. T.; Juditsky, A. B. Acceleration of stochastic approximation by averaging, SIAM J. Control Optim., 30(4), July 1992.

[70] Y. Ueno, Y.; Osawa, K.; Tsuji, Y.; Naruse, A.; Yokota, R. Rich information is affordable: A systematic performance analysis of second-order optimization using K-FAC, in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2145-2153, 2020.

[71] Osawa, K.; Tsuji, Y.; Ueno, Y.; Naruse, A.; Yokota, R.; Matsuoka, S, Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 12 359-12 367, 2019.

[72] Ba, J.; Grosse, R.; Martens, J. Distributed second-order optimization using kronecker-factored approximations, in International Conference on Learning Representations, 2017.

[73] Zhang, L.; Shaohuai Shi, S.; Wang, W.; Li, B. Scalable K-FAC Training for Deep Neural Networks with Distributed Preconditioning, arXiv:2206.15143, 2022.

[74] Brand, M. Fast Low-Rank Modifications of the Thin Singular Value Decomposition, Linear Algebra and its Applications Volume 415, Issue 1, pp. 20-30, 2006.

[75] Puiu, C. O. Rethinking Exponential Averaging of the Fisher, arXiv:2204.04718 (2022).

[76] Trefethen, L. N.; Bau, D. Numerical linear algebra. Philadelphia, PA: Society for Industrial and Applied Mathematics, ISBN 978-0-898713-61-9, 1997.

[77] Demmel, J.; Dimitriu, I.; Holtz, O. Fast linear algebra is stable, arXiv:math/0612264, 2007.

[78] Mazeika M. The Singular Value Decomposition and Low Rank Approximation. *Available: http://math.uchicago.edu/ may/REU2016/REUPapers/Mazeika.pdf.*

[79] Halko N.; Martinsson P.G.; Tropp J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2011.

[80] Saibaba, A. K. Randomized subspace iteration: Analysis of canonical angles and unitarily invariant norms, arXiv:1804.02614 (2018).

[81] Puiu, C. O. Randomized KFACs: Speeding up K-FAC with Randomized Numerical Linear Algebra, arXiv:2206.15397, 2022.

[82] Gao, K.-X.; Liu X.-L.; Huang Z.-H.; Wang, M.; Want S.; Wang, Z.; Xu, D.; Yu, F. Eigenvalue-corrected NG Based on a New Approximation, arXiv:2011.13609 (2020).

[83] Khan, M. E.; Nielsen, D.; Tangkaratt, V.; Lin, W.; Gal, Y.; Srivastava, A. Fast and scalable Bayesian deep learning by weight-pertubation in Adam. In Proceedings of the 35th International Conference on Machine Learning, 2018.

[84] Ledoit, O; Wolf, M. A well-conditioned estimator for large-dimensional covariance matrices. Journal of Multivariate Analysis, 88(2):365-411, 2004.

[85] Ledoit, O; Wolf, M. Honey, I Shrunk the Sample Covariance Matrix, The Journal of Portfolio Management, 30(4), pp. 110-119, DOI: 10.3905/jpm.2004.110, 2004.

[86] Zhang, H.; Zheng, Z.; Xu, S.; Dai, W.; Ho, Q.; Liang, X.; Hu, Z.; Wei, J.; Xie, P.; Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters, in USENIX ATC 17, pp. 181-193 , 2017.

[87] S.-I. Amari, S.-i.; Ba, J.; Grosse, R. B.; Li, X.; Nitanda, A.; Suzuki, T.; Wu, D.; Xu, J. *When does preconditioning help or hurt generalization?*, in International Conference on Learning Representations, 2021.

[88] Shi, S.; Zhang, L.; Li, B. Accelerating distributed k-fac with smart parallelism of computing and communication tasks, in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), IEEE, 2021.

[89] Shallue, C. J.; Lee, J.; Antognini, J.; Sohl-Dickstein, J.; Frostig, R.; Dahl, G. E. Measuring the effects of data parallelism on neural network training, Journal of Machine Learning Research, vol. 20, no. 112, pp. 1-49, 2019.

[90] Shi, S.; Chu, X.; Li, B. MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms, in IEEE INFOCOM 2019-IEEE Conference on Computer Communications, pp. 172-180, 2019.

[91] Li, M.; Andersen, D. G.; Park, J. W.; Smola, A. J.; Ahmed, A.; Josifovski, V. ; Long, J. ; Shekita, E. J.; Su, B.-Y. Scaling distributed machine learning with the parameter server, in 11th USENIXOSDI, pp. 583-598, 2014.

[92] Peng, Y.; Zhu, Y.; Chen, Y.; Bao, Y.; Yi, B.; Lan, C.; Wu, C.; Guo, C. A generic communication scheduler for distributed DNN training acceleration, in Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 16-29, 2019.

[93] Bao, Y.; Peng, Y.; Chen, Y.; Wu, C. Preemptive all-reduce scheduling for expediting distributed DNN training, in IEEE INFOCOM 2020-IEEE Conference on Computer Communications, pp. 626-635, 2020.

[94] Shi, S.; Chu, X.; Li, B. MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning, IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 8, pp. 1903-1917, 2021.

[95] Li, S.; Zhao, Y.; Varma, R.; Salpekar, O.; Noordhuis, P.; Li, T.; Paszke, A.; Smith, J.; Vaughan, B.; Damania, P.; Chintala, S. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, arXiv:2006.15704, 2020.

[96] Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, M. X.; Chen, D.; Lee, H.; Ngiam, J.; Le, Q. V.; Wu, Y.; Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, arXiv:1811.06965, 2018.

[97] Zhang, L.; Shi, S.; Chu, X.; Wang, W.; Li, B.; Liu, C. DeAR: Accelerating Distributed Deep Learning with Fine-Grained All-Reduce Pipelining, arxiv 2302.12445, 2023.

[98] Cui , H.; Zhang, H.; Ganger, G. R.; Gibbons, P. B.; Xing, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems 2016, ACM, p. 4, 2016.

[99] Kleinberg, Jon.; Tardos, E. Algorithm Design. Addison-Wesley Longman, Publishing Co., Inc., USA, 2005.

[100] Goyal, P.; Dollar, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; He, K.; Accurate, large minibatch SGD: Training ImageNet in 1 hour, arXiv preprint arXiv:1706.02677, 2017.

[101] Jia, X.; Song, S.; Shi, S.; He, W.; Wang, Y.; Rong, H.; Zhou, F.; Xie, L.; Guo, Z.; Yang, Y.; Yu, L.; Chen, T.; Hu, G.; Chu, X. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes, in Proc. of Workshop on Systems for ML and Open Source Software, collocated with NeurIPS 2018, 2018.

[102] You, Y.; Li, J.; Reddi, S.; Hseu, J.; Kumar, S.; Bhojanapalli, S.; Song, X.; Demmel, J.; Keutzer, K.; Hsieh, C.-J. Large batch optimization for deep learning: Training BERT in 76 minutes, in International Conference on Learning Representations, 2020.

[103] Chen, J.; Monga , R.; Bengio, S.; Jozefowicz, R. Revisiting distributed synchronous sgd. arXiv preprint arXiv:1604.00981, 2016.

[104] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks, arXiv:1404.5997, 2014.

[105] Yamazaki,M.; Kasagi, A.; Tabuchi, A.; Honda, T.; Miwa, M.; Fukumoto, N.; Tabaru, T.; Ike, A.; Nakashima, K. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds, arXiv:1903.12650, 2019.

[106] Chen, M.; Gao, K.; Liu, X.; Wang, Z.; Ni, N.; Zhang, Q.; Chen, L.; Ding, C.; Huang, Z.; Wang, M. THOR, trace-based hardware-driven layer-oriented natural gradient descent computation, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 8, pp. 7046-7054, 2021.

[107] MLPerf v1.1 Training Results. MLPerf ID: 1.1-2040, 1.1-2042, 1.1-2044, 1.1-2045, 1.1-2065. The MLPerf name and logo are trademarks. See www.mlperf.org for more information. Also see https://towardsdatascience.com/accelerating-resnet-50-training-on-the-ipu-behind-our-mlperf-benchmark-2cefe43ab2b2 .

[108] You, Y.; Gitman, I.; Ginsburg, B. Large batch training of convolutional networks, arXiv preprint arXiv:1708.03888, 2017.

[109] Jia, X.; Song, S.; He, W.; Wang, Y.; Rong, H.; Zhou, F.; Xie, L.; Guo, Z.; Yang, Y.; Yu, L.; Chen, T.; Hu, G.; Shi, S.; Chu, X. Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes, arXiv preprint, arXiv:1807.11205, 2018.

[110] David H. A. and Nagaraja H. N. Order Statistics, Third Edition. Wiley, Hoboken NJ, 2003.

[111] Fresen, D. Simultaneous concentration of order statistics, 2011.

[112] Xia, D. Non-asymptotic bounds for percentiles of independent non-identical random variables, 2019.

[113] Schraudolph, N. N. Fast curvature matrix-vector products for second-order gradient descent, Neural Computation, 14, 2002.

[114] Pascanu, R.; Bengio, Y. Revisiting natural gradient for deep networks, arXiv:1301.3584, 2014.

[115] Tropp, J. A.; Yurtsever, A.; Udell, M.; Cevher, V. Practical Sketching Algorithms for Low-Rank Matrix approximation, arXiv:1609.00048 (2017).

[116] Tropp, J. A.; *An Introduction to Matrix Concentration Inequalities*, FnTML Draft, Revised, 24 December 2014.

[117] Park, H.; Amari, S.-I.; Fukumizu. K. Adaptive natural gradient learning algorithms for various stochastic models, Neural Networks, 13(7):755-764, 2000.

[118] Heskes, T. On "natural" learning and pruning in multilayered perceptrons, Neural Computation, 12(4):881-901, 2000.

[119] Honkela, A.; Raiko, T.; Kuusela, M.; Tornio, M.; Karhunen, J. Approximate riemannian conjugate gradient learning for fixed-form variational bayes, Journal of Machine Learning Research, 11, pp. 3235-3268, 2010.

[120] Arnold, L.; Auger, A.; Hansen, N.; Ollivier, Y. Information-geometric optimization algorithms: A unifying picture via invariance principles. 2011.

[121] Raskutti, G.; Mukherjee, S. The information geometry of mirror descent, arXiv:1310.7780, 2013.

[122] Thomas, P. S.; Dabney, W. C.; Giguere, S.; Mahadevan, S. Projected natural actor-critic, in Advances in Neural Information Processing Systems 26, 2013.

[123] Zhang, G.; Martens, J.; Grosse, R. Fast convergence of natural gradient descent for overparameterized neural networks, Advances in Neural Information Processing Systems 32 (NeurIPS 2019), 2019.

[124] Arora, S.; Du, S. S.; Hu, W.; Li, Z.; Wang, R. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks, arXiv preprint, arXiv:1901.08584, 2019.

[125] Allen-Zhu, Z.; Li, Y.; Song, Z. A convergence theory for deep learning via over-parameterization, arXiv preprint, arXiv:1811.03962, 2018.

[126] Li, Y.; Liang, Y. Learning overparameterized neural networks via stochastic gradient descent on structured data, Advances in Neural Information Processing Systems, pp. 8168-8177, 2018.

[127] Zou, D.; Cao, Y.; Zhou, D.; Gu, Q. Stochastic gradient descent optimizes over-parameterized deep ReLU networks, arXiv preprint arXiv:1811.08888, 2018.

[128] Du, S. S.; Zhai, X.; Poczos, B.; Singh, A. Gradient descent provably optimizes over-parameterized neural networks, arXiv preprint, arXiv:1810.02054, 2019.

[129] Oymak S.; Soltanolkotabi, M. Towards moderate overparameterization: global convergence guarantees for training shallow neural networks, arXiv preprint, arXiv:1902.04674, 2019.

[130] Wu, X.; Du, S. S.; Ward, R. Global convergence of adaptive gradient methods for an over-parameterized neural network, arXiv preprint, arXiv:1902.07111, 2019.

[131] Hazan, E.; Agarwal, A.; Kale, S. Logarithmic regret algorithms for online convex optimization, Machine Learning, 69(2-3):169-192, 2007.

[132] Duchi, J. C.; Hazan, E.; Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization, Journal of Machine Learning Research, 12:2121-2159, 2011.

[133] Collobert, R. Large scale machine learning, PhD thesis, Universite de Paris VI, LIP6, 2004.

[134] Muller K.-R.; Montavon, G. Deep boltzmann machines and the centering trick, in Neural Networks: Tricks of the Trade, Springer, 2013.

[135] Beck A.; Teboulle, M. Mirror descent and nonlinear projected subgradient methods for convex optimization, Operations Research Letters, 2003.

[136] Barron, A.; Markov, T.; Swafford, Z. Deep q-learning with natural gradients, 2016. URL: https://github.com/todor-markov/natural-q-learning/blob/master/writeup.pdf.

[137] Keskar, N. S.; Mudigere, D.; Nocedal, J.; Smelyanskiy, M.; Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima, arXiv preprint arXiv:1609.04836, 2016

[138] Goyal, P.; Dollar, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; He, K. Accurate, large minibatch SGD: training ImageNet in 1 hour, arXiv preprint arXiv:1706.02677, 2017.

[139] Ying, C.; Kumar, D.; Chen, D.; Wang, T.; Cheng, Y. Image classification at supercomputer scale, arXiv preprint arXiv:1811.06992, 2018.

[140] Smith, S. L. Kindermans, P.-J.; Ying, C.; Le, Q. V. Don't decay the learning rate, increase the batch size, in International Conference on Learning Representations, 2018.

[141] Akiba, T.; Suzuki, S.; Fukuda K. Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 Minutes, arXiv preprint arXiv:1711.04325, 2017.

[142] Thomas, V.; Pedregosa, F.; van MerriÃńnboer, B.; Mangazol, P.-A.; Bengio, Y.; Le Roux, N. Information matrices and generalization, arXiv preprint arXiv:1906.07774, 2019.

[143] Kunstner, F.; Balles, L.; Hennig, P. Limitations of the Empirical Fisher approximation, arXiv preprint arXiv:1905.12558, 2019.

[144] Amari, S. I.; Karakida, R.; Oizumi, M. Fisher information and natural gradient learning in random deep networks, in The 22nd International Conference on Artificial Intelligence and Statistics, pp. 694-702, 2019.

Appendix A2

Appendix of Chapter 2

A2.1 First Order Methods: Technical Details

A2.1.1 ADAM: Technical Details

We can easily prove that the bias correction in equation (1.10) is indeed the right one by considering m_k as an example. We wish to know how $\mathbb{E}[m_k]$ is related to $\mathbb{E}[\tilde{g}_k]$. Expanding m_k in terms of all previous gradients, we get

$$m_k = (1-\beta_1)\sum_{t=1}^{k}\beta_1^{k-t}\tilde{g}_k. \tag{A2.1}$$

Taking expectations gives

$$\mathbb{E}[m_k] = \mathbb{E}[\tilde{g}_k](1-\beta_1)\sum_{t=1}^{k}\beta_1^{k-t} + \xi = (1-\beta_1^k)\mathbb{E}[\tilde{g}_k] + \xi, \tag{A2.2}$$

where $\xi = 0$ if the true first-moment $E[\tilde{g}_k]$ is stationary [11]. Otherwise, ξ should be small because β_1 should be chosen such that it assigns negligible weights to gradients too far in the past [11]. Thus, the bias correction shown in (1.10) is at least approximately right. The exact same rationale can be applied to v_t, with virtually no difference, as shown in the original ADAM paper [11]. We now briefly look at the properties of the ADAM step which make it perform so well, and then at some convergence-related issues. To ease our discussion, let us assume $\epsilon = 0$ and define the effective step to be $\Delta_k := \alpha\hat{m}_k/\sqrt{v_k}$. Vector division and comparison with scalars should be thought of as entry-wise.

Properties of ADAM step:

1. **Invariance to Gradient Scaling:** Rescaling $\nabla F_k^{(n)}$ by a factor of $C > 0$ also results in a rescaling of $\hat{m}_k$ and $\hat{v}_k$ by a factor of C and C^2 respectively, which cancel out when performing $\hat{m}_k/\sqrt{\hat{v}_k}$ to compute the step [11]

2. **Trust-Region-like Behavior:** By noting that $\left(\mathbb{E}[\nabla F_k^{(n)}]\right)^2 / \mathbb{E}[\nabla F_k^{(n)} \odot \nabla F_k^{(n)}] < 1$, we get that after sufficiently many iterations we should have $\hat{m}_k / \sqrt{\hat{v}_k} < 1$ [11]. Thus, the effective step satisfies $\Delta_k < \alpha$, which can be understood as setting a box trust-region out of which our steps cannot fall. Most of the times, by knowing the radius of the ball within which the solution may lie, and by estimating a number of steps in which we may converge, we can easily work out an appropriate order of magnitude for α. Thus ADAM does not require involved tuning to achieve good performance [11].
3. **Self-Adjusting Step-Length based on Signal to Noise Ratio (SNR):** We can think of $\hat{m}_k / \sqrt{\hat{v}_k}$ as the signal to noise ratio [11]. Thus, we see that for small SNR, we take smaller steps, which is a desirable property, as a small SNR implies that there is a higher uncertainty about whether the direction $\hat{m}_k$ resembles the direction of the true gradient. As an example, SNR becomes almost zero close to an optimum, which provides an automatic annealing [11].

Convergence Issues of ADAM and the Quick Fix: The original authors of ADAM show that the method is globally convergent [11]. However, Reddi et. al. (2018, [13]) outline that the assumptions made by Kingma et. al. (2015, [11]) do not hold in practice, and show particular numerical results where ADAM converges to the worst point in the set. Furthermore, they show that for any set of parameters $\{\beta_1, \beta_2, \epsilon\}$, there exists a convex stochastic problem for which ADAM does not converge to the optimal solution [13]. They also propose a correction to ADAM which makes it globally convergent: AMSGrad. The intuitive reason behind why ADAM can achieve non-convergence is because the step size is not guaranteed to shrink with increasing k [13]. In particular, if the sequence $\{\nabla F_k^{(n)}\}_{k=1,2,\ldots}$ contains only a few large (informative) gradients and many small (uninformative) ones, then we only have a small number of informative updates. However, their influence dies out quickly due to exponential averaging leading to poor convergence [13]. To fix this problem, as suggested by intuition, we set $\hat{v}_k = \max(\hat{v}_{k-1}, v_k)$, where v_k is computed as shown in equation (1.9). Note that no bias correction is performed, and this fix leads to AMSGrad. AMSGrad has the step sizes upper bounded by the ADAM ones.

A2.2 Natural Gradient Technical Details

We begin by an in depth analysis of the *Fisher Information Matrix,* considering the *true Fisher* and its relationship to other well-known matrices. For the *(true) Natural Gradient (NG) method*, we then look at related interpretations, advantages, and theory.

Finally, we consider the *Empirical Fisher*, the *empirical NG method*, and one related interpretation. *Section A2.2* is mostly based on Martens's paper [64].

A2.2.1 The Fisher: Links with other Curvature Matrices

Connections with the Hessian. There are multiple connections between the Fisher and the Hessian of the loss. Under regularity conditions on p (related to interchanging integration and differentiation), we can show that [64]

$$F(\theta) = \mathbb{E}_{(x,y)\sim p}\big[\nabla_\theta \log p(x,y;\theta) \nabla_\theta \log p(x,y;\theta)^T\big] = \mathbb{E}_{(x,y)\sim p}\big[\nabla^2\big(-\log p(x,y;\theta)\big)\big]. \tag{A2.3}$$

Thus, equation (A2.3) tells us that under our assumption $L(x,y;\theta) = -\log p(y|x;\theta)$, the Fisher is simply the expected value of the Hessian of loss. Note the subtlety here that the expectation is taken w.r.t. the model distribution p, and not the data distribution q[1]. Furthermore, in our typical practical approach, we would replace the true risk minimization problem with an empirical risk minimization problem. Thus, our (practical) Hessian would effectively be

$$\begin{aligned}\nabla^2 \tilde{R}(\theta) = \sum_{i=1}^{N} \nabla^2\big(-\log p(x_i|y_i;\theta))\big) &= \sum_{i=1}^{N} \nabla^2\big(-\log p(x_i,y_i;\theta) - C\big) \\ &= \mathbb{E}_{(x,y)\sim\hat{q}}\big[\nabla^2\big(-\log p(x,y;\theta)\big)\big].\end{aligned} \tag{A2.4}$$

We then see that the difference between the (true) Fisher and the Hessian amounts to changing the distribution w.r.t. which we take the expected value. Another way to view the connection, is to recall[2] that $p(x,y;\theta) = q(x)\pi(y|h(x;\theta))$ and then use conditional expectations to get

$$\begin{aligned}F(\theta) &= \mathbb{E}_{x\sim q(x)}\Big[\mathbb{E}_{y\sim\pi(y|h(x;\theta))}\big[\nabla^2\big(-\log p(x,y;\theta)\big)\big]\Big] \\ &= \mathbb{E}_{x\sim q(x)}\Big[\mathbb{E}_{y\sim\pi(y|h(x;\theta))}\big[\nabla^2\big(-\log \pi(y|h(x;\theta))\big)\big]\Big].\end{aligned} \tag{A2.5}$$

The Hessian of the *true/empirical risk* would be obtained by changing $y \sim \pi(y|h(x;\theta))$ in equation (A2.5) to $y \sim q(y|x)$ and $y \sim \hat{q}(y|x)$ respectively [64].

Connections with the Generalized Gauss-Newton (GGN) Matrix. For our *empirical risk*, the Generalized Gauss-Newton (GGN) matrix is defined as [113]

$$G := \frac{1}{N}\sum_{i=1}^{N} J_{h_i}^T H_{l_i} J_{h_i}, \text{ with } H_{l_i} := \nabla_z^2 l(z,y_i)\big|_{z=h(x_i;\theta)}, \; J_{h_i} := \nabla_\theta h(x;\theta)\big|_{x=x_i} \in \mathbb{R}^{N_o \times d}, \tag{A2.6}$$

[1] Which is what we do for the Hessian of R in (1.3).

[2] Recall that when doing supervised learning with DNNs we choose $p(y|x;\theta) = \pi(y|h(x;\theta))$.

and $l(z, y)$ is an arbitrary convex loss [64]. Equation (A2.6) can be derived by writing

$$\tilde{R} = \frac{1}{N} \sum_{i=1}^{N} l(h(x_i; \theta), y_i), \tag{A2.7}$$

then differentiating twice using chain-rule (to get the Hessian), and finally dropping the term where we sum over entries in the net output. Note that in the particular case of $l(z, y) = (1/2) \|z - y\|^2$ the GGN matrix becomes the Gauss-Newton matrix. For the *true risk*, the GGN matrix is defined analogously to (A2.6) where the sum over i is replaced by an expectation over $(x, y) \sim q$.

To view the connection between GGN and Fisher, we begin by noting that [64]

$$\nabla_\theta \log p(y|x; \theta) = J_h^T \nabla_z \log r(y|z)\big|_{z=h(x;\theta)} \tag{A2.8}$$

where $J_h := \nabla_\theta h(x; \theta)$. From now on, we will write $\nabla_z \log r(y|z)\big|_{z=h(x;\theta)} = \nabla_z \log r(y|z)$ for convenience. Rearranging the Fisher definition (1.11) by using (A2.8), and dropping the obvious arguments, we get

$$F(\theta) = \mathbb{E}_{x\sim q}\left[J_h^T \mathbb{E}_{y\sim r(y|h(x;\theta))}\left[\nabla_z \log r(y|z) \nabla_z \log r(y|z)^T\right] J_h\right] = \mathbb{E}_{x\sim q}\left[J_h^T F_r J_h\right], \tag{A2.9}$$

where $F_r := \mathbb{E}_{y\sim r(y|h(x;\theta))}\left[\nabla_z \log r(y|z) \nabla_z \log r(y|z)^T\right]$ is itself a Fisher, namely the one of r (the conditional predictive distribution of y given the net output). In practice, we take our target distribution to be the *empirical data distribution*, and thus replace q with $\hat{q}$. This gives:

$$F(\theta) = \mathbb{E}_{x\sim \hat{q}}\left[J_h^T F_r J_h\right] = \frac{1}{N} \sum_{i=1}^{N} J_{h_i}^T F_{r_i} J_{h_i}, \tag{A2.10}$$

where $F_{r_i} := \mathbb{E}_{y\sim r(y|h(x_i;\theta))}\left[\nabla_z \log r(y|z)|_{z=h(x_i;\theta)} \nabla_z \log r(y|z)|_{z=h(x_i;\theta)}^T\right]$. To avoid confusion, note that the Fisher in equation (A2.10) is *not* the *Empirical Fisher*. Instead, it is the *true Fisher* when we take our target distribution to be the *empirical data distribution* (rather than the true data distribution). As we saw in *Section 1.4.4*, the *empirical Fisher* refers to pretending that $p(y|x) \equiv q(y|x)$, *not* that $q \equiv \hat{q}$.

By comparing equations (A2.6) and (A2.10), we note that the GGN matrix and the Fisher would be the same if $H_{l_i} = F_{r_i}$. While this is not true in general, we now look at a sufficient condition for this to hold. We will present the condition proposed by *Martens* (2020, [64]), as to date, this is the most general. A slightly less general condition was given by *Pascanu et. al.* (2014, [114]). More particular conditions have also been provided for the equivalence of GN (rather than GGN) and Fisher [117, 118].

Proposition A2.1 (**GGN and Fisher equivalence**). *The GGN and the Fisher are equivalent when the following conditions hold [64]:*

1. *The loss function $l(z, y)$ is convex.*
2. *The loss can be written as $L(x, y; \theta) = -\log p(y, x; \theta)$. That is, we can find a generative model and a conditional density $r(y|z)$ s.t. $l(z, y) = -\log r(y|z)$.*
3. *$r(y|z)$ corresponds to an exponential family with the natural parameters given by z. That is $\log r(y|z) = z^T T(y) - \log Z(z)$, for some function $T(y)$, with $Z(z)$ being the normalizing constant.*

Proof. The first point of *Condition 3.1* is required for the GGN to exist, while the second point is required for the Fisher to exist. When both matrices exist, we have

$$H_{l_i} = -\nabla_z^2 \log r(y|z)|_{z=h(x_i;\theta)} \quad \text{and} \quad F_{r_i} = -\mathbb{E}_{y\sim r(y|h(x_i;\theta)}\left[\nabla_z^2 \log r(y|z)|_{z=h(x_i;\theta)}\right]. \tag{A2.11}$$

To express F_{r_i} as in (A2.11), we required interchaning differentiation and integration. We thus want

$$\nabla_z^2 \log r(y|z)|_{z=h(x_i;\theta)} = \mathbb{E}_{y\sim r(y|h(x_i;\theta)}\left[\nabla_z^2 \log r(y|z)|_{z=h(x_i;\theta)}\right], \tag{A2.12}$$

which holds under point 3 in *Proposition A2.1* (see *Martens*[3]*, 2020*, [64]) $\square$.
Examples of $r(y|z)$ distributions that satisfy *Condition 3.1* include [64]

1. Multivariate normal ($\mathcal{N}(\mu, \Sigma)$) where z parameterizes only the mean.
2. Multivariate normal ($\mathcal{N}(\mu, \Sigma)$) where z is the concatenation of $\Sigma^{-1}\mu$ and the vectorization of Σ^{-1}.
3. Multinomial where the softmax of z is the vector of probabilities for each class.

We have met *examples 1* and *3* when we looked at how to pick r for typically used losses. Thus, we see that in some practical cases, GGN and Fisher are indeed equivalent.

A2.2.2 Interpretations and Advantages of NG

The Metric Tensor on Riemannian (Distribution) Manifold Interpretation. Let $p_\theta := p(\cdot, \cdot; \theta)$. Since we parameterize our model distribution by θ, we can view our optimization as searching through a space of (density) functions $\mathcal{S}_D := \cup_{\theta\in\Theta} p_\theta$. Thus, choosing $\theta \in \Theta$ corresponds to picking one element in $\mathcal{S}_D$. The *KL divergence*

[3]Clarification: The proof given is not part of our contribution (and is the only proof given in the thesis which is not a contribution; it is provided for a better understanding).

behaves as a *distance measure* asymptotically as the two considered distributions tend to one another [114, 119]. By Taylor-expanding the KL divergence to get [64, 120]

$$\mathrm{KL}\big(p(x,y;\theta)\,||\,p(x,y;\theta+d)\big) = \frac{1}{2}d^T F d + \mathcal{O}(d^3), \tag{A2.13}$$

we note that $\mathrm{KL}\big(p(x,y;\theta)\,||\,p(x,y;\theta+d)\big) \to (1/2)d^T F d$ as $d \to 0$. Thus, our Riemannian metric tensor is the *Fisher* F. That is, F locally defines an inner-product around θ (which varies smoothly in θ):

$$< u, v >_\theta = u^T F(\theta) v, \tag{A2.14}$$

and hence a local measure of *distance* [114].

The KL Divergence (or SD in Distribution Space) Interpretation. To understand this interpretation, we first consider gradient descent (GD). The negative gradient represents the direction which locally maximizes the reduction in the objective per unit of distance traveled [64]. Formally, that is

$$-\frac{\nabla R}{\|\nabla R\|} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \underset{d:\|d\| \leq \epsilon}{\arg\min} R(\theta + d). \tag{A2.15}$$

Thus, we see that the gradient direction depends on the Euclidean geometry of the parameter space, and thus is not invariant w.r.t. reparametrization [64]. Conversely, it would be more sensible to take our distance metric to be the KL divergence in the distribution space. This gives us that our Riemannian metric tensor is F, as we have just seen. Now, consider the following result, which holds for any symmetric positive semi-definite (SPSD) matrix[4] A [123]:

$$-\frac{A^{-1}\nabla R}{\|\nabla R\|_{A^{-1}}} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \underset{d:\|d\|_A \leq \epsilon}{\arg\min} R(\theta + d). \tag{A2.16}$$

Substituting $A = (1/2)F$ in (A2.16), and recognizing that $\mathcal{O}(d^3) \to 0$ as $\epsilon \to 0$ in (A2.13), we get:

$$-\sqrt{2}\frac{F^{-1}\nabla R}{\|\nabla R\|_{F^{-1}}} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \underset{d:\mathrm{KL}\left(p(x,y;\theta)\,||\,p(x,y;\theta+d)\right) \leq \epsilon^2}{\arg\min} R(\theta + d). \tag{A2.17}$$

Thus, we see that the natural gradient represents the steepest descent direction in $\mathcal{S}_D$, where the distance between two very close points is approximately the square root of the KL divergence [120]. Since the KL divergence only depends on the distributions,

[4]While $\|\nabla R\|_A$ is not technically a norm, as the matrix is PSD (not PD) we make this slight abuse of notation here for brevity.

but not on how they are parameterized, the r.h.s. of equation (A2.17) is invariant w.r.t. reparameterization. Thus, the natural gradient direction (which is a scalar multiple of the l.h.s. of (A2.17)) is also invariant w.r.t. reparameterization [64]. While this is a much-desired property (similar to the Newton step), we note that it breaks down in practice, due to using finite step sizes. To see this, note that in practice, we do not compute the natural gradient direction in $\mathcal{S}_D$, but rather its representation in Θ, namely $F^{-1}\nabla R$. Because the mapping between $\mathcal{S}_D$ and Θ is non-linear, our representation in Θ of the same direction in $\mathcal{S}_D$ changes when we move a finite distance in Θ. Geometrically this means straight lines in $\mathcal{S}_D$ are not straight lines in Θ. Practically it means that $F^{-1}\nabla R$ would need to be recomputed after each infinitesimally small step in Θ, even if the natural gradient direction remains the same in $\mathcal{S}_D$ [64]. Of course, we also have the natural gradient typically changing in $\mathcal{S}_D$.

A related interpretation of *NGD*, which we omit here for brevity is that NGD is a special case of *Mirror Descent*, where the distance generating function[5] is $\psi(\theta) = (1/2)\theta^T F\theta$ (see [121, 122]).

The Second Order Method Interpretation. The simplest but most practical interpretation comes from thinking about the natural gradient direction as the solution of a local quadratic model

$$s^{(NG)} := \arg\min_s \frac{1}{2} s^T F s + s^T \nabla R. \tag{A2.18}$$

Under some technical conditions (but not always; see *Proposition A2.1*) the Fisher is equivalent to the Generalized Gauss-Newton matrix, which can itself be thought of as an approximation for the Hessian [64]. Thus, the NGD step in (1.19) (with $\alpha = 1$) can be thought of as the optimal update based on a particular second-order model of $R(\theta)$. This view helps us understand the break-downs that may occur when F has some very small eigenvalues, as well as think about the region in which this model is accurate. It thus motivates the use of damping or trust-region.

A2.2.3 Convergence Properties of (True) NGD

Having seen all these interpretations and advantages of NGD (some of them more practical than others), a natural question to ask is whether (stochastic) NGD also achieves better convergence rates than (stochastic) GD. We discuss three of the most important results next.

[5] I.e. the Bregman divergence is $\Delta_\psi(x, y) = (1/2)(x - y)^T F(x - y)$.

Amari's Result: Natural Gradient is "Fisher Efficient" [61]. Amari (1998, [61]) proves that NGD is (asymptotically) *Fisher efficient*. He showed that taking steps of the form $s_k = -\alpha_k F^{-1}\tilde{\nabla}R$ (with $\theta_{k+1} = \theta_k + s_k$), where $\alpha_k = \mathcal{O}(1/k)$ and $\tilde{\nabla}R$ is a stochastic estimate of ∇R with just one example, will produce an estimator[6] θ_k which has the following properties [61]:

1. Is asymptotically unbiased. That is, $\lim_{k\to\infty} \mathbb{E}[\theta_k] = \theta^*$, where $\theta^* := \arg\min_\theta R(\theta)$, and the expectation is taken w.r.t. the whole stochastic process represented by the stochastic optimization algorithm.

2. Has the following asymptotic expected squared error matrix:

$$\mathscr{C}_{MSE} := \mathbb{E}[(\theta_k - \theta^*)(\theta_k - \theta^*)^T] = \frac{1}{k}F^{-1}(\theta^*) + \mathcal{O}(1/k^2). \tag{A2.19}$$

From *property 1* we deduce that the bias of θ_k (seen as an estimator of θ^*) converges to zero, and thus $\mathscr{C}_{MSE} \to \mathscr{C}$, where $\mathscr{C}$ is the asymptotic covariance matrix of θ_k. Thus, we conclude that asymptotically,

$$\mathscr{C} \sim \mathscr{C}_{MSE} = \frac{1}{k}F^{-1}(\theta^*) + \mathcal{O}(1/k^2). \tag{A2.20}$$

This shows that the sequence of iterates $\{\theta_k\}_{k=1,2,\dots}$ is asymptotically *Fisher efficient*, since $\mathscr{C}$ behaves asymptotically as the smallest[7] possible covariance matrix as given by the Cramer-Rao bound [61]. Fisher efficiency is a much-desired property because if we reach it, we know that we cannot come up with any other estimator which has a lower variance for the amount of samples we have collected (here we look at them, then discard). This result tells us that in essence, (stochastic) NGD can be thought of as a rule for constructing an estimator, based on (gradient) samples from q, which is optimal in the sense that no other construction rule can yield an estimator which has a smaller variance for the number of samples k that we have seen. Of course, this result of *Amari* (1998, [61]) holds under some assumptions which we have omitted [64]. A mild assumption is that the loss must have the structure we also assumed in *Section 1.4.1* (for the Fisher to exist). However, the other assumptions are more questionable, as outlined by *Martens* (2020, [64]).

The caveats of *Amari's* results are [64]:

- Assuming $\theta_k \to \theta^*$ at some unspecified rate (and then we find the rate is optimal in the *Fisher efficiency sense*). For convex objectives, this is sensible (provable in most cases), but raises a question for non-convex objectives.

[6] θ_k is seen as an estimator of the global solution θ^*.

[7] Here, *"smallest"* is in the sense of positive semi-definiteness, namely $A \succeq B \iff A - B$ is PSD.

- The Fisher needs to be estimated by using all samples x from $\hat{q}(x)$. This challenges the idea that NGD optimally uses the data it *"sees"*. Considering the *Fisher* construction required by *Amari*, NGD would only be optimal if we can obtain the *Fisher* for free. However, to compute it under *Amari's* assumption, we need to make a full pass through the data at each step, thus *seeing* $kN + k$ examples rather than k after k steps. This effectively says that we would be sub-optimal (in the sense of attaining the minimal covariance matrix) by a factor of $(N + 1)$. That is, asymptotically, we would need to see $N + 1$ times more examples than the best possible algorithm.

- Assuming that we have the density $q(y|x)$, and keep on sampling from it each time we encounter the same x, rather than just having one realization per x, and keep on cycling through (x_i, y_i) (which is what we do in practice). This says we have an infinite number of distinct training sample pairs, even though we have a finite number of samples for x. Clearly, we do not have $q(y|x)$ in practice (and not even sufficiently many draws from it).

- The convergence is w.r.t. θ and not R, but we can convert from one another.

- Assuming the *"realizability"* condition. In short, this tells us that the global (parametric) solution of the *KL divergence minimization problem*[8] gives $p(x, y; \theta^*) \equiv q(x, y)$. Note that this condition does not only require that our model is expressive enough s.t. we can represent the data distribution for some parameter θ^*, but also that we have[9] an infinite set of training pairs (x_i, y_i), as above. It is unclear whether a weaker asymptotic result would still hold when the *"realizability"* condition is not satisfied (which is what we would expect).

Even without the outlined caveats, there is another problem. *Polyak and Juditsky* (1992, [69]) have shown that SGD also becomes *Fisher efficient* if we use *Polyak averaging* [69]. This raises two fundamental questions. The first one is, why would we even try using (stochastic) NGD[10] when *Polyak averaged SGD* also seems to have the same asymptotic optimality property? The second one is, why do we empirically observe that, even in stochastic regimes, some second-order methods achieve better

[8] We have seen we can convert our loss to this form in most cases.

[9] The idea is that we have to keep on seeing new samples from q, rather than the same for the Cramer-Rao bound to hold. Otherwise, we are effectively fitting to a distribution having a *probability mass function positive at a finite number of points* (rather than density), in which case the Cramer-Rao bound does not hold. In this case, we can perfectly fit $\hat{q}$ after finitely many samples, but this would correspond to an overfit [64]. See *Section 14 in Martens* (2020, [64]) for details.

[10] Or any second-order method for that matter.

progress per iteration[11] than (Polyak averaged) SGD. An answer to this, based on a particular case, is given by Martens (2020, [64]), which is what we look at next.

Martens' Result: NG on quadratic objective [64]. *Martens* (2020, [64]) provides two results, both for the *particular case of convex quadratic objective*. To draw parallels with the realtively general ML risk objective, we will refer to this particular objective as $R_S(\theta)$, defined as:

$$R_S(\theta) := \frac{1}{2}(\theta - \theta_S^*)^T H^* (\theta - \theta_S^*), \tag{A2.21}$$

where H^* is a symmetric PD matrix. Note that $\theta_S^* = \arg\min_\theta R_S(\theta)$. To put ourselves in the stochastic regime, assume we only have access to noisy unbiased estimates of ∇R_S, which we denote as $g_S(\theta)$. The author further assumes that the covariance matrix of $g_S(\theta)$ is constant w.r.t. θ. Let us denote this matrix as $\mathscr{C}_{g_S}$. We consider analyzing steps of the form

$$\theta_{k+1} = \theta_k - \alpha_k B^{-1} g_S(\theta_k), \tag{A2.22}$$

for some fixed positive definite matrix B. Note that this includes both the case of second order methods for nontrivial B, as well as first-order methods for B being a multiple of the identity. We will use $\lambda_{\max}(\cdot)$ and $\lambda_{\min}(\cdot)$ to refer to the maximum and minimum eigenvalue of a matrix respectively. We are now ready to proceed. Instead of looking at the two results, and then at their implications, we will look at the implications directly, to avoid prolonged discussions. There are three main implications.

The ***first implication*** tells us that for a fixed step-size $\alpha_k = \alpha$, with $\alpha \leq 1/\lambda_{\max}(B^{-1}H^*)$, we have [64]

$$\lim_{k\to\infty} \mathbb{E}[R_S(\theta_k)] = R_S(\theta^*) + \frac{\alpha}{4}\mathrm{tr}\left(\left(B - \frac{\alpha}{2}H^*\right)^{-1}\mathscr{C}_{g_S}\right). \tag{A2.23}$$

Thus, we recover a type of result very well known in the literature for stochastic optimization with constant step size. In particular, we see that convergence happens up to a noise factor, which decreases with both the constant stepsize α, and the noise level.

The ***second important implication*** tells us that for a decreasing step-size of the form $\alpha_k = 1/(k+a+1)$, with any $a \geq 1$, and a choice of $B = H^*$, we get [64]

$$\mathbb{E}[R_S(\theta_k)] - R_S(\theta^*) \leq \frac{1}{2(k+a)}\mathrm{tr}\big((H^*)^{-1}\mathscr{C}_{g_S}\big) + R_S(\theta_0)\left(\frac{1+a}{k+a}\right)^2. \tag{A2.24}$$

[11] Even though not necessarily per unit time.

We can bound the coefficient of the asymptotically dominant term in equation (A2.24) as [64]

$$\frac{1}{2\lambda_{\max}(H^*)}\mathrm{tr}(\mathscr{C}_{g_S}) \leq \frac{1}{2}\mathrm{tr}\big((H^*)^{-1}\mathscr{C}_{g_S}\big) \leq \frac{1}{2\lambda_{\min}(H^*)}\mathrm{tr}(\mathscr{C}_{g_S}). \tag{A2.25}$$

The ***third important implication*** tells us that for a decreasing step-size of the form $\alpha_k = 1/(k+\kappa(H^*)+1)$, where $\kappa(H^*) = \lambda_{\max}(H^*)/\lambda_{\min}(H^*)$, and a choice of $B = \lambda_{\min}(H^*)I$, we get [64]

$$\mathbb{E}[R_S(\theta_k)] - R_S(\theta^*) \leq \zeta\left(\frac{1}{4(k+\kappa(H^*))} + \mathcal{O}\left(\frac{1}{(k+\kappa(H^*))^2}\right)\right) + R_S(\theta_0)\left(\frac{1+\kappa(H^*)}{k+\kappa(H^*)}\right)^2, \tag{A2.26}$$

with

$$\zeta := \frac{1}{\lambda_{\min}(H^*)}\mathrm{tr}\left(\left(I - \frac{\lambda_{\min}(H^*)}{2}(H^*)^{-1}\right)^{-1}\mathscr{C}_{g_S}\right). \tag{A2.27}$$

We can bound the coefficient of the asymptotically dominant term (ζ) in equation (A2.26) as [64]

$$\frac{1}{4\lambda_{\min}(H^*)}\mathrm{tr}(\mathscr{C}_{g_S}) \leq \zeta \leq \frac{1}{2\lambda_{\min}(H^*)}\mathrm{tr}(\mathscr{C}_{g_S}). \tag{A2.28}$$

Note that *the second implication* refers to preconditioning a stochastic gradient step with the *true* Hessian, while the *third implication* refers to taking an SGD step with a particular step-size schedule. In both cases, the convergence rate is $\mathcal{O}(1/k)$. However, by comparing equations (A2.24) and (A2.26), we see that using $B = H^*$ can lead to a faster decay rate in the $\mathcal{O}(1/k^2)$ term for well-chosen a. That is because, for $B = H^*$, we may choose any $a \geq 1$ (which includes $\kappa(H^*)$), and thus can decay the $\mathcal{O}(1/k^2)$ term at least as fast as in the case when $B = \lambda_{\min}I$. The $\mathcal{O}(1/k^2)$ term is important in the early phases of convergence, and this simple analysis suggests that second-order methods are likely to perform better than first-order ones *in the first few steps*. Furthermore, by comparing equations (A2.25) and (A2.28) we see that while the rate might be $\mathcal{O}(1/k)$ in both cases, the constants are different. In particular, the lower bound on the constant in front of the $\mathcal{O}(1/k)$ term is better in the case of $B = H^*$. While this does not mean that the constant itself is better, it suggests that this may happen at least in some particular cases. *Martens* (2020, [64]) provides some realistic particular cases when this holds true. Still, recall that the considered framework was itself very limited.

Having seen these results, we realize that they were more about using the Hessian than using the Fisher, and ask ourselves: how does this relate to Natural Gradient? The answer to this is that we have seen before that the *true Fisher* is equivalent to the

GGN matrix in certain settings, which is itself (sometimes) a reasonable approximation of the Hessian. Thus, we can hope that using the Fisher as a preconditioner should help convergence rates, at least in the very particular case outlined above. Of course, the question of how much of this advantage is left in practice when using the *Fisher* remains open, since we will not have convex quadratic objectives and the *Fisher* might as well be significantly different from the *Hessian*. We end this section by looking at a slightly more convincing result (for overparameterized nets).

NGD on Overparameterized Nonlinear NNs with Squared-error Loss [123]. *Zhang et. al.* (2019, [123]) prove that for *overparametrized* neural nets, NGD attains *efficient convergence* towards *a global minimum* (of the *empirical risk*) under two intuitive conditions. While it may be surprising that NGD manages to find *a global solution*, similar results have been provided for (S)GD, and thus are common in this setting. In particular, it has been proved that (S)GD finds a global optimum from a random initialization if the net is sufficiently wide (s.t. it is sufficiently overparametrized; see [125, 126, 127, 128, 129]). It is thus the rate at which this happens (or rather the constant of this rate) which is better for NGD, at least on two-layer ReLU nets, as we will see based on the results of *Zhang et. al.* (2019, [123]). While there is a *caveat*[12] related to attaining *zero training loss*, we can still get an intuition for the advantages of NGD over SGD.

We first provide the general result of *Zhang et. al.* (2019, [123]) for overparameterized nets. We will then give up on some generality by *studying the particular case of a two-layer (overparameterized) ReLU network*, in order to increase the interpretablility of results. We will then provide results concerning GD for the same *particular setting* ([128, 129, 130]), and thus see that NGD has better constants than GD. All results presented below consider the *squared loss* and a *scalar net output*. All results which refer to the *two-layer ReLU net* are based on mild assumptions about the data set which we omit here for brevity (see [123, 128, 129, 130]). To begin, let

[12] [126] provide convincing results where the global optimum is viewed in the wider sense of minimum *generalization error*. In essence, they do this by *using each data point exactly once*. Conversely, [123, 125, 127, 128, 129] (which are the results we compare here) provide results for which the global optimum translates to *zero error on the training set*. Arguably, these fall into the overfit case, because their global solution perfectly represents $\hat{q}$ - which may not generalize. Note that finding a global optimum of the *empirical risk* is not outstanding in the context of overparameterized nets, as in this case there would be many global solutions, each of which represents (to some extent) an overfit. What we are really after is finding the global minimum of the *true risk* (like [126] implicitly does to within some error), which does not coincide in general with the *global minimum of the empirical risk*. While generalization loss upper-bounds exist, and can be applied to the case of attaining *zero training loss* (see [124] for SGD and [123] for NGD) it is unclear how useful these are in practice, as we may have unrealistic conditions or impractically large resulting upper-bounds.

$\mathbf{h}(\theta) := [h(x_1;\theta), h(x_2;\theta), ..., h(x_N;\theta)]^T$, $\mathbf{y} := [y_1, y_2, ..., y_N]^T$, $\mathbf{X} := [x_1, x_2, ..., x_N]^T$, and $\mathbf{J}(\theta) \in \mathbb{R}^{N\times d}$ be the Jacobian of $\mathbf{h}(\theta)$ w.r.t. θ. Four of the following results will only train the first layer of a two-layer net. Thus, we will refer to the parameters of the i^{th} layer as $\theta^{(i)}$, with $\theta = [\theta^{(1)}, \theta^{(2)}, ...]$. Finally, let w be the width of the net ($\theta^{(2)} \in \mathbb{R}^w$), and $\lambda_0 := \lambda_{\min}(\mathbf{G}^\infty)$, where $\mathbf{G}^\infty$ is the *limmiting Gramm matrix*, whose definition we skip here for brevity (see *Section 4* in [123]).

The most general result of ***Zhang et. al.* (2019, [123])** states that when taking NGD steps

1. If $\mathbf{J}(\theta_0)$ is full row-rank, $N \leq d$, and $\mathbf{J}$ is *"stable"* (a technical condition; see [123]), we have
$$\|\mathbf{h}(\theta_k) - \mathbf{y}\|^2 \leq (1-\eta)^k \|\mathbf{h}(\theta_0) - \mathbf{y}\|^2, \tag{A2.29}$$
where $\eta \leq (1-2C)/(1+C)^2$, is the learning rate, for $C > 0$ a small constant related to $\mathbf{J}$'s *"stability"*. Note that the NGD steps are taken using an *approximate true Fisher* by saying $p(x, y; \theta) = q(x)p(y|x;\theta)$ and drawing samples from $q(x)$ and $p(y|x;\theta)$. While in principle this is *not* the *empirical Fisher*, note that in this case the GGN matrix, the GN matrix, the *true Fisher*, and even the *empirical Fisher* all coincide. The *true Fisher approximation* (obtained by estimating expectations from samples) is $F \approx \mathbf{J}^T\mathbf{J}$ which is low-rank in this case, and thus the generalized inverse is used. While this result is interesting, we cannot know whether the required conditions hold, particularly the *"Jacobian stability"*. The authors thus provide a particular case when these indeed hold, which we look at next.

In particular, for a *two-layer ReLU net*, where we use NGD to train the *first layer only*, the result of ***Zhang et. al.* (2019, [123])** reduces to:

2. If we initialize $\theta_0^{(1)} \sim \mathcal{N}(0, \nu^2 I)$, $\theta_0^{(2)} \sim \mathcal{U}([-1, 1])$, choose $w = \Omega\big(N^4\nu^{-2}\lambda_0^{-4}\delta^{-3}\big)$, set the learning rate $\eta = \mathcal{O}(1)$ with $\eta \in (0, 1)$, then with probability at least $1-\delta$ over the random initialization we have
$$\|\mathbf{h}(\theta_k) - \mathbf{y}\|^2 \leq (1-\eta)^k \|\mathbf{h}(\theta_0) - \mathbf{y}\|^2. \tag{A2.30}$$
Note that $\nu^2 > 0$ is arbitrary. This is almost the same as *result 1*, but with milder assumptions because we chose a particular network architecture. The analysis can be extended to include the case when both layers are trained [123].

In contrast, for a *two-layer ReLU net*, where we use GD to train the *first layer only*, the result of ***Du et. al.* (2019, [128])** states that:

3. If we initialize $\theta_0^{(1)} \sim \mathcal{N}(0, I)$, $\theta_0^{(2)} \sim \mathcal{U}([-1, 1])$, choose $w = \Omega\big(N^6 \lambda_0^{-4} \delta^{-3}\big)$, set the learning rate $\eta = \mathcal{O}(\lambda_0 / N^2)$ with $\eta \in (0, 1)$, then with probability at least $1 - \delta$ over the random initialization we have

$$\|\mathbf{h}(\theta_k) - \mathbf{y}\|^2 \leq \left(1 - \frac{\eta \lambda_0}{2}\right)^k \|\mathbf{h}(\theta_0) - \mathbf{y}\|^2 . \tag{A2.31}$$

Similarly, for a *two-layer ReLU net* where we use GD to train the *the first layer only*, the result of ***Oymak et. al.* (2019, [129])** states that

4. If we initialize $\theta_0^{(1)} \sim \mathcal{N}(0, I)$, half *weights in* $\theta_0^{(2)}$ as $-\|\mathbf{y}\| / (Nw)$ and half as $+\|\mathbf{y}\| / (Nw)$, choose $w = \Omega\big((1 + \mu)^2 N^9 \|\mathbf{X}\|^6 \iota^{-4}\big)$, set the learning rate $\eta = \bar{\eta} N / (3 \|\mathbf{y}\|^2 \|\mathbf{X}\|^2)$ with $\bar{\eta} \in (0, 1)$, then with probability at least $1 - \frac{2}{N} - \exp(-\mu^2 N / \|\mathbf{X}\|^2)$ over the random initialization we have

$$\|\mathbf{h}(\theta_k) - \mathbf{y}\| \leq \left(1 - c \frac{\bar{\eta} \iota}{N^2 \|\mathbf{X}\|^2}\right)^k \|\mathbf{h}(\theta_0) - \mathbf{y}\| . \tag{A2.32}$$

Note that ι is a constant telling us how well spaced the data is (part of omitted assumptions) and $\mu > 0$ is arbitrary. Under the crude approximation that $\mathbf{y} \sim \mathcal{N}(0, \nu_y^2 I)$ and $\mathbf{x}_i \sim \mathcal{N}(0, \nu_x^2 I)$, we have $\|\mathbf{y}\|^2 = \Theta(N)$ and $\|\mathbf{X}\|^2 = \Theta(N)$, and thus we require $\eta = \mathcal{O}(1/N)$.

Finally, for a *two-layer ReLU net* where we use GD to train the *first layer only*, the result of ***Wu et. al.* (2019, [130])** states that

5. If we initialize $\theta_0^{(1)} \sim \mathcal{N}(0, I)$, $\theta_0^{(2)} \sim \mathcal{U}([-1, 1])$, choose $w = \Omega\big(N^6 [\lambda_{\min}(\mathbf{G}^\infty)]^{-4} \delta^{-3}\big)$, set the learning rate $\eta = \Theta(1 / \|\mathbf{G}^\infty\|)$ with $\eta \in (0, 1)$, then with probability at least $1 - \delta$ over the random initialization we attain an objective function error (w.r.t. global optimum) of ϵ in

$$T = \tilde{\mathcal{O}}\left(\frac{\|\mathbf{G}^\infty\|}{\lambda_0} \log(1/\epsilon)\right) \tag{A2.33}$$

iterations. Note that $\|\mathbf{G}^\infty\| = \mathcal{O}(N)$, and thus we require $\eta = \mathcal{O}(1/N)$ [130]. While this result is in a slightly different format, for small enough $\lambda_0 / \|\mathbf{G}^\infty\|$ it is roughly the same as saying $\|\mathbf{h}(\theta_k) - \mathbf{y}\|^2 \leq \big(1 - c \frac{\lambda_0}{\|\mathbf{G}^\infty\|}\big)^k \|\mathbf{h}(\theta_0) - \mathbf{y}\|^2$ for some constant $c > 0$.

Result 2 tells us that for sufficiently many parameters, we can take η to be $\mathcal{O}(1)$, and thus converge in $\mathcal{O}(1)$ steps [123]. Furthermore, we see that in the case of NGD (*result 2*) the negative term of the convergence constant can be larger by a factor

of $N/\lambda_{\min}(\cdot)$ than the maximum allowable one for GD (attained by *result 5*). Thus, NGD has a better convergence constant (at least in this simple setting), albeit both NGD and GD converge linearly. While these results were for *exact natural gradient*, ***Zhang et. al.* (2019, [123])** provide a similar result for K-FAC (a practical NGD algorithm, which we saw in *Section 1.5* and is presented in more depth in *Sections 3.2.2* and *3.3.1*), although the level of overparameterization required is significantly higher. This suggests that there should be ways to implement NGD to outperform GD in terms of iterations (although outperforming in terms of time is not guaranteed).

In summary, we have looked at three main sets of theoretical results in this section. Firstly, we have seen that NGD is asymptotically *Fisher efficient* (SGD is not unless *Polyak averaging* is used), though under some questionable conditions [61, 64, 69]. Considering that obtaining the Fisher is challenging in practice, and that SGD with *Polyak averaging* is also Fisher efficient, we asked ourselves why to bother trying to use NGD. We partly answered this question by looking at *a second set of results*, after *Martens* (2020, [64]). He showed that in a very particular setting, *stochastic Newton's method* will achieve faster progress in the initial phase of convergence, while in the final phase, *it is mathematically possible* (but not a certainty) that the constants of the asymptotically dominant terms are better than for SGD. This still left us with a gap, since the *Fisher* is not a good approximation for the *Hessian* in all cases, and we desire generality. We partly bridged the gap by considering a *third set of theoretical results*. For an overparameterized two-layer ReLU network with squared-error loss and scalar net outputs, we compared results for NGD after *Zhang et. al.* (2019, [123]) with results for GD ([128, 129, 130]). From these, we have seen that NGD has a better convergence constant, at least in the considered setting. While the comparisons we have made were fairly particular, all the results we have seen suggest that NGD should outperform SGD w.r.t. iterations in general.

We now finish the chapter by comparing *empirical NG* with the *true NG* as well as look at an interesting interpretation of the *Empirical NGD step*.

A2.2.4 Discussion of Empirical NG vs True NG

The main disadvantage of using the $F^{(E)}$ rather than F is losing the interpretations and advantages outlined so far in *Section A2.2*. One of the biggest problems is that $F^{(E)}$ will not have any connections with the GGN matrix anymore, and thus we cannot hope for it to be a reasonable substitute for the Hessian (in a local quadratic model), as we do in the case of F [64]. The *empirical Fisher* also loses the connection with the Hessian that we established in *Section A2.2.1*. That is because the second equality

sign in equation (A2.3) only holds for $\mathbb{E}_{(x,y)\sim p}[\cdot]$ but not for $\mathbb{E}_{(x,y)\sim q}[\cdot]$. However, $F^{(E)}$ would still retain the invariance property of F.

While *Martens* (2020, [64]) gives an artificial example on which he argues that using $F^{(E)}$ is *"at least questionable"*, *Yang et. al.* (2021, [20]) provide an *empirical Fisher* algorithm (SENG) which outperforms not only K-FAC (Marten's *true Fisher* algorithm, [1]), but also SGD and ADAM on all test problems. This is the first second-order (in particular *natural gradient*) algorithm to provide such convincing results. Although SENG outperforms K-FAC due to the computation scaling linearly in net width (unlike K-FAC which scales cubically[13]), the fact that it also outperforms first order methods may suggest that $F^{(E)}$ does not have such terrible properties.

A fact reinforcing this idea is that ADAM (very successful first-order stochastic optimization algorithm, [11]) relies on diagonal estimates of the *empirical Fisher*, and not of the *true Fisher*. To allow for such good performance of *empirical Fisher* algorithms to exist in his explanation framework[14], *Martens* (2020, [64]) provides an *ad-hoc* explanation saying it matters how you approximate $F^{(E)}$. He argues that certain approximations might in fact retain desired properties, or provide other[15], but that these (see for example [131, 132]) would not translate to $F^{(E)}$ (or to an arbitrary approximation of $F^{(E)}$). In the support of the $F^{(E)}$ having reasonable properties also comes *Le Roux et. al.* (2008, [19]) which theoretically show that the *empirical natural gradient* direction represents the direction which locally minimizes the probability to increase the generalization error. We look at this next, in *Section A2.2.5*. In summary, whether to use $F^{(E)}$ or F, both from a theoretical and practical perspective remains a subject for future research [64].

A2.2.5 Empirical NG Minimizes the Generalization Error

The discussion in this section only holds for MLE supervised learning.

The g_i's (see equation (1.6)) are drawn from some true gradient distribution. If we make the rough approximation that these draws are i.i.d, we get by CLT that our batch gradient approximately satisfies (assuming large enough n_{B})

$$g_{\text{B}} \sim \mathcal{N}\left(g, \frac{1}{n_{\text{B}}}\mathscr{C}\right), \tag{A2.34}$$

[13]And one may think, based on Martens' reasoning, that True-NG-based K-FAC may outperform SENG if we can get it to scale linearly in net width.

[14]Where the *empirical Fisher* is questionable.

[15]Which is untypical of approximations. They usually make properties worse.

where $\mathscr{C}$ is the true covariance matrix of g_i, and g is the true mean (which is also the gradient of the *true risk*), as we have seen in *Section 1.4* [19]. Thus, if we take a search direction of the form $v = M^T g_{\mathrm{B}}$ (which all practical second order methods do) for some symmetric matrix M, we get [19]

$$v^T g = g_{\mathrm{B}}^T M g \sim \mathcal{N}\left(g^T M g, \frac{g^T M^T \mathscr{C} M g}{n_{\mathrm{B}}}\right). \tag{A2.35}$$

Now, the value of M which minimizes $\mathbb{P}(v^T g > 0)$ can be found as

$$M^* = \arg\min_M \frac{g^T M g}{g^T M^T \mathscr{C} M g}. \tag{A2.36}$$

Differentiating (A2.36) w.r.t M, and setting to zero gives

$$\left[g(g^T M^T \mathscr{C} M g) - 2\mathscr{C} M g (g^T M g)\right] g^T = 0. \tag{A2.37}$$

The first term in the brackets is in the span of g, while the second one is in the span of $\mathscr{C} M g$. For the brackets to be zero, we thus need $M^* = \gamma \mathscr{C}^{-1}$, for some scalar $\gamma \in \mathbb{R}^*$. This gives that $v^* = \gamma \mathscr{C}^{-T} g_{\mathrm{B}} = \gamma \mathscr{C}^{-1} g$ is the direction that minimizes the probability of locally increasing the generalization[16] error [19].

Now, note that by definition, the *empirical Fisher* is the uncentered covariance matrix, namely $F^{(E)} = \mathscr{C} + g g^T$. Thus, we can see that if $v = \mathscr{C}^{-1} g$ then $\bar{v} = [F^{(E)}]^{-1} g$ is given by $\bar{v} = (1 + g^T v)^{-1} v$. As such, given our noisy batch-gradient estimate g_{B}, the *empirical natural gradient descent direction* $-\bar{v} = -[F^{(E)}]^{-1} g_{\mathrm{B}}$ will be the direction along which the probability of locally increasing the error is minimized[17]. This may help with the overfitting issue [19, 44].

Note that we cannot have the same interpretation for the *true Fisher*, as $\mathscr{C}$ is the covariance matrix which we *"see"* when sampling our gradients by picking some data points (that is, using q). Having seen a compelling discussion about *Fisher*, *natural gradient*, and *their empirical counterparts*, we now start considering practical natural gradient methods. The main focus of these methods is to reduce the computation, storage, and inversion cost of the *Fisher*, all of which are prohibitive for large problems.

[16] We can indeed conclude about the generalization error rather than the measured training loss because we looked at $v^T g$, where g is the true gradient (here we implicitly assume that the realizations of the test and train sets come from the same distribution).

[17] Of course, the minimization happens only for a certain step-size implied by equation (A2.37) which no-one bothers to use in practice to compute the step-size!

Appendix A3

Appendix of Chapter 3

A3.1 Bounds avoiding the Low-rank Assumption

This section acts as a complement for *Sections 3.1.1 - 3.1.2* of the main body. It provides theory which, unlike the one in the main body, does not assume low-rank M_j.

Proposition A3.1 (**Probabilistic bounds describing eigenvalue decay of $\bar{\mathcal{M}}_k$ with no low-rank assumption**). *For the SPSD $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$ in (3.19) consider its eigenvalues sorted in descending order. Further assume that M_i has the structure $M_i \overset{\text{SVD}}{=} U_i \Sigma_i V_i^T \ \forall i \in \mathbb{Z}^+$ where*

- *$\Sigma_i = diag(\sigma_i^{(1)}, ..., \sigma_i^{(d_M)})$ with $\sigma_i^{(j)} \in [\sigma_{\min}, \sigma_{\max}]$, $\sigma_i^{(j)} \overset{iid}{\sim} \mathscr{D}_\sigma$, i.i.d. $\forall i, j$, bounded $0 \leq \sigma_{\min}$ and $\sigma_{\max} < \infty$.*
- *$U_q \in \mathbb{R}^{d_M \times d_M}$, $U_q R \overset{\text{QR}}{=} X_q$, with $X_q \in \mathbb{R}^{d_M \times d_M}$ and $[X_q]_{i,j} \sim \mathcal{N}(0,1)$ iid $\forall i, j, q$. And $[X_q]_{i,j}$ are independent from $\sigma_i^{(j)}$.*
- *$V_i^T \in \mathbb{R}^{d_M \times n_M}$ with $n_M \geq d_M$, but no assumption its distribution.*

In addition, assume that our initialization choice is the deterministic value $\bar{\mathcal{M}}_0 = 0$. Let $F_\sigma(\cdot)$ be the c.d.f. of (any) $\sigma_i^{(j)}$.

Then, for any $r \in \{1, 2, ...d_M - 1\}$, we have the following bounds showing eigendecay, depending on the relationship between N and k:

1. *For $N \leq k - 1$ (holds through out the whole spectrum for large enough k):*

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{(1-\rho^{k-N})\sigma_{\max}^2}{(1-\rho)b^2} + \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2} \tag{A3.1}$$

holds with probability

$$p_{OS} := \mathbb{P}\big(\sigma_{(1)} \geq b \ \&\ \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}\big(\sigma_{(r+1)} \leq \eta\big)\big]^{Q-1} \tag{A3.2}$$

for any $b, \eta \in [\sigma_{\min}, \sigma_{\max}]$ (with $b > \eta$), and any $Q \in \mathbb{Z}^+$, where $\sigma_{(1)} \geq ... \geq \sigma_{(r+1)} \geq ... \geq \sigma_{(d_M)}$ are the order statistics of the i.i.d. draws $\{\sigma^{(j)} \sim \mathscr{D}_\sigma\}_{j=1}^{d_M}$.

2. *For $N \geq k$ (transient case in the eigenspectrum tail):*

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2}, \tag{A3.3}$$

holds with probability at least p_{OS} as in (A3.2) for any $Q \in \mathbb{Z}^+$, and any $b, \eta \in [\sigma_{\min}, \sigma_{\max}]$, with $b > \eta$.

Proof. Importantly, throughout the proof we will consider the eigenvalues (which are also the singular values) of $\bar{\mathcal{M}}_k$ to be sorted descendingly ($\forall k$), such that $\lambda_1(\bar{\mathcal{M}}_k) = \sigma_1(\bar{\mathcal{M}}_k) \geq \lambda_2(\bar{\mathcal{M}}_k) = \sigma_2(\bar{\mathcal{M}}_k) \geq ....$ We will also consider the eigenvalues of $M_i M_i^T$ to be sorted descendingly as well in a similar fashion. However, note that the singular values of M_i are not sorted. We make all these choices for simplicity.

Also note that $\{M_i M_i^T\}_{\in \mathbb{Z}^+}$ is a set of i.i.d. random matrices, and this is a direct consequence of the assumed structure of M_i's. We will use this (particularly the independence) many times.

Proof for Statement 1 (when $N \leq k-1$). Proof outline:

1. Use Weyl's monotonicity inequality to deduce

$$\lambda_{Nr+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}) + (1-\rho)\sum_{i=0}^{N-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T). \tag{A3.4}$$

   Then say

$$\sum_{i=0}^{N-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T) \leq \sum_{i=0}^{\infty} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T), \tag{A3.5}$$

   which is a trivial inequality because $M_{k-i}M_{k-i}^T$ are SPSD[1] for all $i > 0$. Note that the negative indexed matrices d not exist in practice, but we can imagine they did since $M_i M_i^T$ are i.i.d.

2. Show $\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq (1-\rho^{k-N})\sigma_{\max}^2$ deterministically[2]

[1] While the inequality might seem unnecessary or negligent, we use it to simplify our result and avoid "switches" in N. The price we pay is reduced tightness of bounds, especially for low N. However we still preserve the main message of the theory, and the approach is simpler to follow, which is why we choose it.

[2] We could also use this deterministic bound in conjunction with the Chernoff bound we used in *Section 3.1.2* to say that $\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq \min\{(1-\rho^{k-N})\mathbb{E}[(\sigma_i^{(j)})^2] + c(1-\rho)\sigma_{\max}^2, (1-\rho^{k-N})\sigma_{\max}^2\}$

3. Use the fact that $\lambda_r(M_{k-i}M_{k-i}^T)$ (eigenvalues sorted non-increasingly) is the $(d_M - r)^{th}$ order statistic of i.i.d. random variables set $\{[\sigma_i^{(j)}]^2\}_{j=1}^{d_M}$ to compute the probability of the event $\{\lambda_{r+1}(M_{k-i}M_{k-i}^T) \leq \eta^2\}_{i=0}^{Q} \cup \lambda_1(M_k M_k^T) \geq b^2$, for any $Q \in \{1, ..., N-1\}$, and any $b, \eta \in [\sigma_{\min}, \sigma_{\max}]$, with $b > \eta$.

   - We use $\lambda_{r+1}(M_{k-i}M_{k-i}^T) \leq \sigma_{\max}^2 \ \forall i \geq Q$ to avoid raising probabilities to powers that may go to infinity as $N \to \infty$.
   - Use these bounds to upper-bound the term $(1-\rho)\sum_{i=0}^{N-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T)$.
   - Use the lower bound on $\lambda_1(M_k M_k^T)$ to lower-bound $\lambda_1(\bar{\mathcal{M}}_k)$.

4. Combine 1, 2, and 3 to conclude upper bound on $\lambda_{Nr+1}(\bar{\mathcal{M}}_k)/\lambda_1(\bar{\mathcal{M}}_k)$ with high probability $(\forall N \in \{1, 2, ...\lfloor (d_M - 1)/r \rfloor\})$.

Part 1: Weyl's monotonicity theorem states that $\sigma_{i+j-1}(A+B) \leq \sigma_i(A) + \sigma_j(B) \ \forall \ i$ and j such that the indices make sense. Applying it to our case with $j \leftarrow r+1$, and $i+j-1 \leftarrow Nr+1$, $A+B \leftarrow \bar{\mathcal{M}}_k$ and $B \leftarrow M_k M_k^T$ we get

$$\sigma_{Nr+1}(\bar{\mathcal{M}}_k) \leq \sigma_{(N-1)r+1}(\bar{\mathcal{M}}_{k-1}) + \sigma_{r+1}(M_k M_k^T) \tag{A3.6}$$

Applying this idea N times, we get

$$\sigma_{Nr+1}(\bar{\mathcal{M}}_k) \leq \rho^N \sigma_1(\bar{\mathcal{M}}_{k-N}) + (1-\rho)\sum_{i=0}^{N-1} \rho^i \sigma_{r+1}(M_{k-i}M_{k-i}^T). \tag{A3.7}$$

Since $\bar{\mathcal{M}}_k$ are all SPSD their singular values are the same as their eigenvalues, so we have

$$\lambda_{Nr+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}) + (1-\rho)\sum_{i=0}^{N-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T). \tag{A3.8}$$

Now, we can add terms of the form $\rho^i \lambda_{r+1}(M_{k-i}M_{k-i}) \ \forall i \in \{N, n+1, ...\infty\}$ on the r.h.s. of (A3.8) and preserve the inequality order since $M_i M_i^T$ are all SPSD. We get

$$\lambda_{Nr+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}) + (1-\rho)\sum_{i=0}^{\infty} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T). \tag{A3.9}$$

---

holds with probability at least $p_{CB} := 1 - 2d_M\left[\frac{1}{(1+c\delta_{\min})}\right]^{1/\delta_{\min}}\left[\frac{e}{(1+c\delta_{\min})}\right]^c$ for any $c > 1$, $\delta_{\min} := (1-\rho)\frac{\sigma_{\max}^2}{(1-\rho^{\max\{k-N,1\}})\mathbb{E}[(\sigma_i^{(j)})^2]}$. The min function together with "$\forall c > 1$" would act as a switch, to allow us to switch to the deterministic bound by making c very large (beneficial when the Chernoff bound is too lose). This appraoch would give us slightly tighter bounds, but since it does not add anything to the understandng of the main point presented here, we do not pursue it.

Note that the nonpositive indexed $M_j M_j^T$ ($\forall j \in \mathbb{Z}^- \cup \{0\}$) do not exist, strictly speaking, but we can think of them being just different iid draws from the same distribution as the one of $M_i M_i^T$ (for positive i). While adding these terms is unneccessary, it simplifies our computations and presentation dramatically, at the expense of making out bounds looser. However, this simplified version still preserves the main message of the chapter, and thus we chose presenting it.

Finally, note that we require $N \in \{1, 2, ...\lfloor (d_M - 1)/r \rfloor\}$ for the above equations to make sense.

Part 2: Now, $\lambda_{\max}(\cdot) = \|\cdot\|_2$ for us, since all the matrices are SPSD. We can thus use triangle inequality and the fact that $\sigma_{\max}(M_i) \leq \sigma_{\max}$ to obtain the following deterministic bound on $\lambda_{\max}(\bar{\mathcal{M}}_{k-N})$

$$\lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \leq \sum_{i=1}^{k-N} \rho^{k-N-i} \left\| M_i M_i^T \right\|_2 \leq \sum_{i=1}^{k-N} \rho^{k-N-i} \sigma_{\max}^2 = (1 - \rho^{k-N}) \sigma_{\max}^2. \quad \text{(A3.10)}$$

Part 3: We first want to bound $\lambda_{r+1}(M_{k-i} M_{k-i}^T)$ from above $\forall i \in \{0, ..., Q\}$ for some $Q \in \mathbb{Z}^+$. We focus on some arbitrary fixed i in the stated set, and then we can combine all the statements by multiplying probabilities since $M_i M_i^T$ are independent, which is a direct consequence of the assumed distributions.

We begin by noting that $\lambda_{r+1}(M_{k-i} M_{k-i}^T)$ is the order $d_M - r$ statistic of the set of i.i.d. random variables $\{(\sigma_i^{(j)})^2\}_{j=1}^{d_M}$. Note that $\sigma_i^{(j)}$'s are not sorted (in j) but $\lambda_r(M_{k-i} M_{k-i}^T)$ are sorted descendingly in r. The sorted eigenvalues of $M_{k-i} M_{k-i}^T$ are ordered statistics (sorted descendingly) because these are the ordered set of d_M i.i.d random variables coming from the distribution of $(\sigma_i^{(j)})^2$.

Now, noting that the eigenvalues of $M_i M_i^T$ are the squared singular values of M_i, and choosing $\eta \in [\sigma_{\min}, \sigma_{\max}]$, we have

$$\mathbb{P}(\lambda_{r+1}(M_{k-i} M_{k-i}^T) \leq \eta^2) = \mathbb{P}(\sigma_{(r+1)} \leq \eta), \quad \forall i \in \{0, 1, ..., k-1\} \quad \text{(A3.11)}$$

where $\sigma_{(r+1)}$ is the $(r+1)^{\text{th}}$ order statistic of the d_M i.i.d. random variables set $\{\sigma^{(j)} \overset{iid}{\sim} \mathscr{D}_\sigma\}_{j=1}^{d_M}$.

Similarly, choosing $b \in (\eta, \sigma_{\max})$, we have that $\forall i \in \{0, 1, ..., k-1\}$

$$\mathbb{P}\big(\lambda_{r+1}(M_{k-i} M_{k-i}^T) \leq \eta^2 \;\&\; \lambda_1(M_{k-i} M_{k-i}^T) \geq b^2\big) = \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big). \quad \text{(A3.12)}$$

Thus, combining (A3.11) $\forall i \in \{1, ..., Q-1\}$ with some $Q \in \mathbb{Z}^+$, and (A3.12) for $i = 0$, and using the fact that M_i are independent, we get that with probability

$$p_{OS} := \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}(\sigma_{(r+1)} \leq \eta)\big]^{Q-1}, \tag{A3.13}$$

all the following events hold simultaneously:

$$\lambda_1(\bar{\mathcal{M}}_k) \geq (1-\rho)\lambda_1(M_k M_k^T) \geq (1-\rho)b^2, \tag{A3.14}$$

$$\lambda_{r+1}(M_{k-i}M_{k-i}^T) \leq \eta^2, \quad \forall i \in \{0, 1, ..., Q-1\}. \tag{A3.15}$$

The first inequality in (A3.14) holds deterministically by the construction of $\bar{\mathcal{M}}_k$ and the fact that all eigenvalues of $M_i M_i^T$ are non-negative, and the second inequality in (A3.14) holds under the stated event which occurs with probability p_{OS}.

Part 4: The bound (A3.10) is deterministic. Thus, the bounds (A3.10), (A3.14), (A3.15) all hold together with probability p_{OS} as in (A3.13).

From *Part 1*, rearranging (A3.9) slightly, we have

$$\begin{aligned}\lambda_{Nr+1}(\bar{\mathcal{M}}_k) \leq\; & \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}) \\ & + (1-\rho)\Bigg[\sum_{i=0}^{Q-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T) + \sum_{i=Q}^{\infty} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T)\Bigg].\end{aligned} \tag{A3.16}$$

Using the assumed bounds on the distribution of $\sigma_i^{(j)}$, we get that all the eigenvalues of $M_i M_i^T$ are within $[\sigma_{\min}^2, \sigma_{\max}^2]$. Applying this deterministic bound for $i \in \{Q, ...\infty\}$, we have the deterministic bound:

$$\lambda_{Nr+1}(\bar{\mathcal{M}}_k) \leq \rho^N \lambda_{\max}(\bar{\mathcal{M}}_{k-N}) + (1-\rho)\sum_{i=0}^{Q-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T) + \rho^Q \sigma_{\max}^2. \tag{A3.17}$$

Dividing by $\lambda_1(\bar{\mathcal{M}}_k)$ we get that the following bound holds deterministically:

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_1(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\lambda_{\max}(\bar{\mathcal{M}}_{k-N})}{\lambda_1(\bar{\mathcal{M}}_k)} + \frac{(1-\rho)\sum_{i=0}^{Q-1} \rho^i \lambda_{r+1}(M_{k-i}M_{k-i}^T) + \rho^Q \sigma_{\max}^2}{\lambda_1(\bar{\mathcal{M}}_k)}. \tag{A3.18}$$

Now, using the upper-bound (A3.10) for $\lambda_{\max}(\bar{\mathcal{M}}_{k-N})$, the lowerbound (A3.14) for $\lambda_1(\bar{\mathcal{M}}_k)$, and the upper-bounds (A3.15) for $\lambda_{r+1}(M_{k-i}M_{k-i}^T)$ with $i \in \{0, 1, ..., Q-1\}$, we get that

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_1(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{(1-\rho^{k-N})\sigma_{\max}^2}{(1-\rho)b^2} + \frac{(1-\rho^Q)\eta^2 + \rho^Q \sigma_{\max}^2}{(1-\rho)b^2}. \tag{A3.19}$$

with probability p_{OS} as in (A3.13) for any $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$ with $\eta < b$. This is the desired result and thus completes the proof of *Statement 1 of Proposition A3.1.*

Proof for Statement 2 (when $N \geq k$): Similarly to *Part 1* of proving *Statement 1 of Proposition A3.1*, we use Weyl's monotonicity Theorem to get

$$\sigma_{Nr+1}(\bar{\mathcal{M}}_k) \leq \sigma_{(N-1)r+1}(\bar{\mathcal{M}}_{k-1}) + \sigma_{r+1}(M_k M_k^T). \tag{A3.20}$$

This time however, we have $N \geq k$, so we can only apply the inequality recursively k times to get

$$\begin{aligned} \sigma_{Nr_M+1}(\bar{\mathcal{M}}_k) &\leq \rho^k \sigma_{(N-k)r_M+1}(\bar{\mathcal{M}}_0) + (1-\rho)\sum_{i=0}^{k-1} \rho^i \sigma_{r+1}(M_{k-i} M_{k-i}^T) \\ &= 0 + (1-\rho)\sum_{i=0}^{k-1} \rho^i \sigma_{r+1}(M_{k-i} M_{k-i}^T) \leq \sum_{i=0}^{\infty} \rho^i \sigma_{r+1}(M_{k-i} M_{k-i}^T), \end{aligned} \tag{A3.21}$$

where we used the equality $\sigma_{(N-k)r_M+1}(\bar{\mathcal{M}}_0) = 0$, which holds because by our initialization choice we have $\bar{\mathcal{M}}_0 = 0$.

From here, we follow the exact logic of *Parts 3 and 4* of *Proof of Statement 1*, noting that *Part 2* is not required as there is no more $\lambda_1(\bar{\mathcal{M}}_{k-N})$ to be bounded. Also note that our sum is now for $i \in \{0, ..., k-1\}$ rather than $i \in \{0, ..., N-1\}$ as in the proof of *Statement 1 of Proposition A3.1.* So we will have $N \leftarrow k$ on the r.h.s of all the deduced bounds and in the interval in which Q exists. The details are omitted, but the proof is mechanically doing the exact same steps as we did when proving *Statement 1 of Proposition A3.1*, but noting these differences (which make it slightly easier). This finishes proving Statement 2 of *Proposition A3.1*, and thus the entire proof of *Proposition A3.1.*

□

Based on *Proposition A3.1*, we have the following important remarks.

Remark A3.2. *Importantly, Proposition A3.1 gives eigenspectrum decay bounds of the form:*

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq A\rho^N + B(\eta, b, Q) \tag{A3.22}$$

for some $A, B(\eta, b, Q) > 0$ and any $r \in \{1, ..., d_M\}$, which holds with some probability $p(\eta, b, Q, r, d_M)$. Note that $A\rho^N$ is an exponential decay term and $B(\eta, b, Q)$ is a "plateau term". Thus, we have the following:

1. ***Bounds with hyperparameters:*** *$r, Q \in \mathbb{Z}^+$, $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$. The bound is best understood as viewing $r, Q \in \mathbb{Z}^+$, $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$ as bound parameters, fixing them, and then looking at what happens as we increase $N \in \mathbb{Z}^+$. Of course it can also be viewed in other ways (eg. one may choose one or more of r, Q, η, b to be a function of each other or of d_M).*

2. ***Can easily get tighter bounds, but these would not change the essence of the message in this chapter, while complicating the exposition.*** *In the proof, by avoiding transitioning to inequality (A3.9) and using (A3.8) instead (also see text around those equations), one may get a bound which changes Proposition A3.1 such that (A3.1) is replaced by $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^{\min\{N,Q\}} \frac{\sigma_{\max}^2 - \eta^2}{(1-\rho)b^2} + \frac{\eta^2}{(1-\rho)b^2}$ (when $N \leq k-1$), (A3.3) is replaced by $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \frac{\eta^2}{(1-\rho)b^2}$ (when $N \geq k$), and (A3.2) is replaced by $p_{OS} \geq \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}\big(\sigma_{(r+1)} \leq \eta\big)\big]^{\min\{Q,N\}-1}$ ($\forall Q \in \mathbb{Z}^+$). One can check that these more complicated bounds are much tighter for relatively low N when Q is low and η is high. But they become very close otherwise. The simpler approach is sufficient for the transmitting the main message of this section, hence-why we chose it.*

3. ***Suggestive probability name.*** *p_{OS} is the probability with which the bounds on order statistics hold together. Note that we chose to use a looser deterministic bound for $\lambda_1(\bar{\mathcal{M}}_{k-N})$, rather than the Chernoff bound we used in Section 3.1.2. Therefore there is no probability term of the form (3.35), as we had in Section 3.1.2 (which came from using the Chernoff bounds). We made this choice as it dramatically simplifies the exposition while preserving the conclusion (only makes the constants slightly worse).*

4. ***Exponential decay, then plateau.*** *As N increases, Proposition A3.1 predicts that the eigenspectrum initially decays almost exponentially if $B \ll A$, and then it plateaus. Note however that we need to choose r s.t. p_{OS} is high and fixed as d_M increases (at least for sufficiently large d_M to make sense of the bound). whether $r \propto 1$ or $r \propto d_M$ must be chosen to attain this desire depends on F_σ, as described in point 9 below.*

5. ***Trade-off for choosing bounds hyperparameters*** *b, η* ***and*** *Q. Higher b, $1/\eta$ and Q improve the bounds (by getting lower $B(\eta, b, Q)$), but reduce the probability $p(\eta, b, Q, r)$ with which these hold.*

6. ***Higher*** r ***gives lower plateau (with the same probability), but slower exponential decay.*** *As we increase* r*, we can reduce* η *while maintaining the same* p_{OS}*, all else fixed. Nevertheless, the index* $Nr+1$ *will be larger.*

7. ***Cannot hope for arbitrarily small tolerance to be attainable, in general.*** *From Proposition A3.1, in general, we cannot hope for arbitrarily small* $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)}$ *to be attainable, as we did in Propositions 4.1 and 4.4. That is because of the presence of* $B(\eta, b, Q) \geq 0$*. Having* $B(\eta, b, Q) = 0$ *is only achievable under special cases.*

8. ***Proposition 3.4 is almost a particular case of Proposition A3.1.*** *If we choose our* $\sigma_i^{(j)}$ *distribution to yield* $\sigma_i^{(j)} = 0$ *with probability* $\frac{d_M - \tilde{r}}{d_M}$ *(for some fixed* $\tilde{r}$*). This is one* ***special case*** *under which arbitrarily small* $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)}$ ***is attainable*** *(within a fixed number of modes w.r.t.* d_M *and w.h.p), as in this case* $B(\eta, b, Q)$ *can be made to vanish for large* Q *while retaining reasonably high probability. We will see this in Example A3.7.*

9. ***Choosing*** r ***(as a function of*** d_M***) to get high fixed probability*** p_{OS} ***as*** d_M ***increases.*** *The bounds we established in Proposition A3.1 hold for any choice of* $r = r(d_M)$*. However, the important aspect is how to choose* $r = r(d_M)$ *to get our bounds holding with high and fixed probability* p_{OS}*. We can then turn these into a conclusion about how the number of modes we need to keep scales with* d_M *(with high probability and for relative truncation error controlled by* ϵ*) in Section 3.1.2. The choice of* $r(d_M)$ *to achieve this desired goal is determined by how* F_σ *behaves with increasing* d_M*. To understand our discussion note that our bounds are only useful if* $\eta < \sqrt{1-\rho}\sigma_{\max}$ *(otherwise the minimum attainable relative truncation error is above* 1*, since* $B(\rho, b, \eta, Q) > 1$*,* $\forall Q, b$ *in their respective allowed sets - in which case we need to retain all the modes for any desired* $\epsilon > 0$*). Thus, our* η *needs to be fixed as some* $\eta \ll \sqrt{1-\rho}\sigma_{\max}$ *for our bounds to be useful, and then for this* η*, we need to choose* $r = r(d_M)$ *to get constant and high* p_{OS} *(at least for all* d_M *sufficiently large). With* η *fixed as just established, have the following cases:*

   (a) F_σ ***concentrates mass below some*** $\epsilon_r \sigma_{\max}$ ***(***$\epsilon_r \ll 1$***) at a sufficiently fast rate with increasing*** d_M***.*** *Under this case we can choose a fixed* r *and still get a fixed high* p_{OS} *as* d_M *increases. This case closely corresponds to the case when* $r_M \propto 1$ *in Section 3.1.2, but the low-rankness of* M_i *is now approximate, and holds only with some high probability. The rigurous*

treatment of this aspect is done in Corollary A3.5 and Examples A3.6 and A3.7. We avoid technical details here for brevity.

(b) F_σ ***is fixed w.r.t.*** d_M***.*** *In this case, any fixed* r *leads to* $\lim_{d_M \to \infty} \mathbb{P}\big(\sigma_{(r+1)} \leq \eta\big) = 0$*, since* $\sigma_{(r+1)}$ *is the* $(r+1)^{th}$ *(highest) order statistic of the set* $\{\sigma^{(j)} \overset{iid}{\sim} F_\sigma(\cdot)\}_{j=1}^{d_M}$ *(and we know that for fixed* $F_\sigma(\cdot)$ *the top* r *order statistics, for any fixed* r*, converge almost surely*[3] *to* $\sup_{x \in [\sigma_{\min}, \sigma_{\max}]}\{x | F_\sigma(x) < 1\} = \sigma_{\max}$ *as* $d_M \to \infty$*, see [110]). In this case, we can only get a constant* $P\big(\sigma_{(r+1)} \leq \eta\big) > 0$ *if we choose* $r = \lfloor a d_M \rfloor$ *(for fixed* $a \in (0, 1]$*). We will investigate this case in Corollary A3.9 and Example A3.10 and see that they lead to a conclusion similar to the one when* $r_M \propto d_M$ *in Section 3.1.2.*

Remark A3.3. *Note that Proposition A3.1 gives us a collection of bounds (each of which holds with its very high probability) and choosing larger* r *gives slower decay in the exponential part (because* $Nr + 1$ *is larger), but a also lower plateau can be obtained for the same probability* $p(\eta, b, Q, r)$ *(see Remark A3.2(5)). Thus, by combining different bounds from Proposition A3.1, for different* r*'s and using the fact that* $\mathbb{P}(\mathscr{E}_1 \cup \mathscr{E}_2) \geq \mathbb{P}(\mathscr{E}_1) + \mathbb{P}(\mathscr{E}_2) - 1$ *we get get tighter bounds which squeeze some more decay in the "plateau region", but these would hold with slightly lower probability. We make this remark for completeness, to understand our bounds are stronger than what they might seem, and to briefly understand what happens to the eigenspectrum, but do not pursue this avenue further.*

We now turn *Proposition A3.1*, a result which gives bounds implying some form of eigenspectrum decay, into a result about how many eigenmodes we need to keep for a relative truncation error no larger than some ϵ of our choice: in *Corollary A3.4*. Because *Proposition A3.1* gives us a set of probabilistic bounds ($\forall r \in \{1, ...d_M - 1\}$), we are also free to choose r in *Corollary A3.4* to get the lowest number of modes N_ϵ possible.

Corollary A3.4 (**The number of modes we need to keep for a fixed accuracy, probabilistic setting, no low-rank assumption**)**.** *Choose any* $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$ *with* $\eta < b$*, any* $r \in \{1, 2, ...d_M - 1\}$*, and any* $Q \in \{1, ..., N - 1\}$*.*

Under the assumptions of Proposition A3.1, we have the following two statements:

[3]This convergence holds whether or not $\sup_{x \in [\sigma_{\min}, \sigma_{\max}]}\{x | F_\sigma(x) < 1\} = \sigma_{\max}$ is finite. In our case, it will be finite, but we make this note.

1. *Fix a tolerance $\epsilon > B(\eta, b, Q)$. We have that at most $\min(N_\epsilon r, d_M)$ eigenvalues of $\bar{\mathcal{M}}_k$ are above $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$, where*

$$N_\epsilon = \left\lceil \left(\log\big(\epsilon - B(\eta, b, Q)\big) - \log\left(\frac{\sigma_{\max}^2}{(1-\rho)b^2}\right)\right) / \log(\rho) \right\rceil. \tag{A3.23}$$

with probability

$$p_{OS} = \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}(\sigma_{(r+1)} \leq \eta)\big]^{Q-1}, \tag{A3.24}$$

where $\sigma_{(1)} \geq ... \geq \sigma_{(r+1)} \geq ... \geq \sigma_{(d_M)}$ are the order statistics of the i.i.d. draws $\{\sigma^{(j)} \sim \mathscr{D}_\sigma\}_{j=1}^{d_M}$, and

$$B(\eta, b, Q) := \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2}. \tag{A3.25}$$

2. *Any $\epsilon \leq B(\eta, b, Q)$ cannot be attained based on our bounds (for our chosen η, b, Q) no matter how many modes we keep. In this case, we need to keep all the modes to attain a truncation error of zero (if $\epsilon \leq B(\eta, b, Q)$ is desired).*

Proof. We unify the two statements of *Proposition A3.1* (losing some bound strength), and then reverse the relationship to bound N.

From Statement 1 of Proposition A3.1: For $N \leq k-1$, we have $1 - \rho^{k-N} \leq 1$. Plugging this inequality in *Statement 1 of Proposition A3.1*, we get that for $N \leq k-1$:

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\sigma_{\max}^2}{(1-\rho)b^2} + \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2} \tag{A3.26}$$

holds with probability

$$p_{OS} = \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}(\sigma_{(r+1)} \leq \eta)\big]^{Q-1} \tag{A3.27}$$

for any $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$ with $\eta < b$, $r \in \{1, ..., d_M - 1\}$, and any $Q \in \mathbb{Z}^+$.

From Statement 2 of Proposition A3.1: For $N \geq k$, we have from (A3.3) that

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2} \tag{A3.28}$$

holds with probability at least p_{OS} as in (A3.27), for any $\eta, b \in [\sigma_{\min}, \sigma_{\max}]$ with $\eta < b$, and any $Q \in \{1, ..., k-1\}$.

Since the first term on the r.h.s. of (A3.26) is nonegative, and the second term on the r.h.s. of (A3.26) is the same as the term on the r.h.s. of (A3.28), we can unify the statements for any $N \geq 1$. We have that

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\sigma_{\max}^2}{(1-\rho)b^2} + B(\eta, b, Q), \tag{A3.29}$$

holds with probability p_{OS} as in (A3.27), and

$$B(\eta, b, Q) := \frac{\rho^Q \sigma_{\max}^2 + (1-\rho^Q)\eta^2}{(1-\rho)b^2}. \tag{A3.30}$$

Now, we can ensure $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \epsilon$ by setting the r.h.s. of (A3.29) to be below ϵ and finding the N for which this occurs. Inspecting the r.h.s of (A3.29) we note that it is always above $B(\eta, b, Q)$ $\forall N \geq 1$. Thus, an $\epsilon < B(\eta, b, Q)$ **cannot** be attained based on our bounds. This proves *Statement 2 of Corollary A3.4.*

For any $\epsilon \geq B(r, Q, d_M)$, imposing

$$\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \rho^N \frac{\sigma_{\max}^2}{(1-\rho)b^2} + B(\eta, b, Q) \leq \epsilon, \tag{A3.31}$$

is sufficient to ensure $\frac{\lambda_{Nr+1}(\bar{\mathcal{M}}_k)}{\lambda_{\max}(\bar{\mathcal{M}}_k)} \leq \epsilon$ with probability p_{OS} as in (A3.27). Rearranging (A3.31) to obtain $N \geq ...$ and choosing N_ϵ to be the smallest N that satisfies this obtained inequality completes the proof of *Statement 1* of *Corollary A3.4*, thus completing the proof. □

Remark A3.2(9) tells us that in order to get useful bounds holding with constant and high probability p_{OS} as d_M increases, we need to choose $r = r(d_M)$ in different ways depending on the behavior of F_σ with d_M. *Remark A3.2(9)* also hinted (without proof) at the fact that this choice of $r = r(d_M)$ is: (a) $r \propto 1$ if F_σ concentrates mass sufficiently close to zero at sufficiently high rate with increasing d_M, and (b) $r \propto d_M$ if F_σ is fixed w.r.t. d_M. In the remaining portion of this section (*Section A3.1*) we rigorously treat these aspects, making *Remark A3.2(9)* precise. *Corollary A3.5* and *Examples A3.6 - A3.7* cover case (a). *Corollary A3.9* and *Example A3.10* cover case (b).

Corollary A3.5 ($F_\sigma(\cdot)$ **concentrating mass below a threshold as** d_M **increases means we only need to keep a fixed number of modes**). *In addition to the assumptions in Corollary A3.4, let us assume that* $\sigma_{\min} < \epsilon_\sigma \sigma_{\max}$ *for some* $\epsilon_\sigma \in [0, \sqrt{1-\rho})$*, and that* F_σ *concentrates mass below* $\epsilon_\sigma \sigma_{\max}$ *as* d_M *increases in such a way that we have*

$$\mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) \geq 1 - \omega, \qquad \forall d_M > r_M + 1, \tag{A3.32}$$

for some $\omega \in (0,1)$ and $r_M \in \mathbb{Z}^+$, both fixed w.r.t. d_M. Further assume that F_σ varies in such a way with d_M that it also satisfies

$$\mathbb{P}\big(\sigma_{(1)} \geq \hat{b}\sigma_{\max}\big) \geq 1 - \bar{\omega}, \qquad \forall d_M > r_M + 1 \tag{A3.33}$$

for some $\bar{\omega} \in (0,1)$, and some $\hat{b} \in (\epsilon_\sigma/\sqrt{1-\rho}, 1]$, both fixed w.r.t. d_M. Let

$$B_{\min} := \epsilon_\sigma^2/[(1-\rho)\hat{b}^2]. \tag{A3.34}$$

Then, $\forall d_M > r_M + 1$, we have the following statements

1. *Fix a tolerance $\epsilon \in (B_{\min}, 1)$. We have that at most $\min(N_\epsilon r_M, d_M)$ eigenvalues of $\bar{\mathcal{M}}_k$ are above $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$, where*

$$N_\epsilon = \left\lceil \left(\log\big(\epsilon - B(\epsilon_\sigma\sigma_{\max}, \hat{b}\sigma_{\max}, Q)\big) - \log\left(\frac{1}{(1-\rho)\hat{b}^2}\right)\right)/\log(\rho) \right\rceil, \tag{A3.35}$$

   with probability p_{OS}, where

$$p_{OS} \geq (1-\omega-\bar{\omega})(1-\omega)^{Q-1}, \tag{A3.36}$$

$$B(\epsilon_\sigma\sigma_{\max}, \hat{b}\sigma_{\max}, Q) = \frac{\rho^Q}{(1-\rho)\hat{b}^2} + (1-\rho^Q)B_{\min}, \tag{A3.37}$$

   for any $Q \in \mathbb{Z}^+$ with $Q > \hat{Q}$:

$$\hat{Q} := \left(\frac{(1-\rho)\hat{b}\epsilon - \epsilon_\sigma^2}{1-\epsilon_\sigma^2}\right)/\log(\rho) \tag{A3.38}$$

2. *If we desire $\epsilon \leq B_{\min}$, we need to retain all the d_M modes of $\bar{\mathcal{M}}_k$ and get zero truncation error with certainty.*

Proof. **Outline:** *We are just applying Corollary A3.4 with the choices choices $\eta := \epsilon_\sigma\sigma_{\max}$, $b := \hat{b}\sigma_{\max}$ and $r := r_M$, and use the assumptions (A3.32) and (A3.33). We then note that $\epsilon > B(\epsilon_\sigma\sigma_{\max}, \hat{b}\sigma_{\max}, Q)$ is equivalent to $\epsilon > B_{\min}$ and $Q > \hat{Q}$ ($Q \in \mathbb{Z}^+$) holding together, and use the latter condition instead.*

Recall from *Corollary 4.1.2* that $p_{OS} = \mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big)\big[\mathbb{P}(\sigma_{(r+1)} \leq \eta)\big]^{Q-1}$. The following inequality holds for the intersection of any two events:

$$\mathbb{P}\big(\sigma_{(1)} \geq b \;\&\; \sigma_{(r+1)} \leq \eta\big) \geq \mathbb{P}\big(\sigma_{(1)} \geq b\big) + \mathbb{P}\big(\sigma_{(r+1)} \leq \eta\big) - 1. \tag{A3.39}$$

Thus, we have

$$p_{OS} \geq \left(\mathbb{P}\big(\sigma_{(1)} \geq b\big) + \mathbb{P}\big(\sigma_{(r+1)} \leq \eta\big) - 1\right)\big[\mathbb{P}(\sigma_{(r+1)} \leq \eta)\big]^{Q-1}. \tag{A3.40}$$

Now, substituting the assumptions (A3.32) and (A3.33) in the p_{OS} bound (A3.40) with the choices $\eta := \epsilon_\sigma \sigma_{\max}$, $b := \hat{b}\sigma_{\max}$ and $r := r_M$, we get that

$$p_{OS} \geq (1 - \omega - \bar{\omega})(1 - \omega)^{Q-1}. \tag{A3.41}$$

Note that the condition $\eta > b$ required by *Corollary A3.4* is satisfied since $\eta = \epsilon_\sigma \sigma_{\max}^2 < \sqrt{1-\rho}\,\hat{b}\sigma_{\max}^2 < \hat{b}\sigma_{\max}^2 = b$, where the equalities hold by our choice of η and b, the first inequality holds by the assumed restrictions on ϵ_σ and b, and the second equality holds because $\rho \in (0, 1)$.

Substituting the assumptions (A3.32) and (A3.33) in the equation for N_ϵ of *Corollary A3.4*, (A3.23), and noting $\eta := \epsilon_\sigma \sigma_{\max}$, $b := \hat{b}\sigma_{\max}$, we get

$$N_\epsilon = \left\lceil \left(\log\left(\epsilon - B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, Q)\right) - \log\left(\frac{1}{(1-\rho)\hat{b}^2}\right)\right) / \log(\rho) \right\rceil. \tag{A3.42}$$

where

$$B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, Q) = \frac{\rho^Q + (1-\rho^Q)\epsilon_\sigma^2}{(1-\rho)\hat{b}^2} = \frac{\rho^Q}{(1-\rho)\hat{b}^2} + (1-\rho^Q)B_{\min}. \tag{A3.43}$$

with $B_{\min} := \epsilon_\sigma^2/[(1-\rho)\hat{b}^2]$. We also need to substitute $r = r_M$ everywhere in the *Corollary A3.4* statement (we make this choice above). This almost completes the proof but since we can choose Q, we want to change the condition $\epsilon > B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, Q)$ (under which statement 1 of *Corollary A3.4* holds) into a condition where Q is not present (and a restriction on Q) as this will help with our reasoning later.

Now, based on equation (A3.43), we see that as long as $\epsilon > B_{\min}$, we can always choose $Q \in \mathbb{Z}^+$ large enough such that $B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, Q) < \epsilon$ holds. Furthermore, we can choose any $Q \in \mathbb{Z}^+$ above the smallest $\hat{Q} \in \mathbb{R}$ which solves $B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, \hat{Q}) = \epsilon$. Thus, the restriction $\epsilon > B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, Q)$ is equivalent to $\epsilon > B_{\min}$ and $Q > \hat{Q}$ ($Q \in \mathbb{Z}^+$) holding together, so we can change the condition in the statement as such. The value of $\hat{Q} \in \mathbb{R}$ which solves $B(\epsilon_\sigma \sigma_{\max}, \hat{b}\sigma_{\max}, \hat{Q}) = \epsilon$ can easily be worked out to be the one in (A3.38). □

Notes about the assumptions of *Corollary A3.5*:

1. Note that under assumptions (A3.32) and (A3.33), we get that any M_i is "almost" (tolerance $\epsilon_\sigma/\hat{b}$) low-rank (rank r_M, or less), since we can combine (A3.32) and (A3.33) by using $\mathbb{P}(A \cap B) \geq \mathbb{P}(A) + \mathbb{P}(B) - 1$ to get that $\forall d_M > r_M + 1$ we have

$$\mathbb{P}\left(\frac{\sigma_{(r_M+1)}}{\sigma_{(1)}} \leq \frac{\epsilon_\sigma}{\hat{b}}\right) = \mathbb{P}\big(\{\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\} \cap \{\sigma_{(1)} \geq \hat{b}\sigma_{\max}\}\big) \geq 1 - \omega - \bar{\omega}, \tag{A3.44}$$

with $\omega + \bar{\omega} \ll 1$ and $\epsilon_\sigma / \hat{b} \ll 1$.

2. Note that the assumption (A3.32) can only hold if F_σ changes with d_M. To see this, note that F_σ is fixed, the convergence of order statistics gives that for any fixed r_M, $\sigma_{(r_M+1)} \to \sigma_{\max}$ in probability as $d_M \to \infty$ ([110]), in which case $\lim_{d_M \to \infty} \mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) = 0$ for any $\epsilon_\sigma \in (0, 1)$. In particular, using the well-known c.d.f. formula of the $(r_M + 1)^{\text{th}}$ order statistic, one can deduce that having $\mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) = 1 - \omega$ fixed $\forall d_M$ (as assumed) can only be achieved if F_σ concentrates its mass below $\epsilon_\sigma \sigma_{\max}$ at a fast enough rate as $d_M \to \infty$. While this assumption may seem artificial, from the note above we see that it is just a "noisy" (probability is not 1) and approximate ($\epsilon_\sigma \geq 0$) version of the low-rank M_i assumption (with fixed low-rank) we have made in *Section 3.1.2*.

3. The lower-bound on $\hat{b}$ is required s.t. $B_{\min} < 1$ (s.t. the interval $(B_{\min}, 1)$ in which ϵ can take useful values exists) and the upper-bound on ϵ_σ ensures the interval in which $\hat{b}$ takes values exists.

Further Notes about *Corollary A3.5* result and interpretation:

1. Note that $B_{\min}$ has no dependence on d_M, and we can fix $Q \in \mathbb{Z}^+$ to any value $Q \geq \hat{Q}$ (note $\hat{Q}$ has no dependence on d_M) in which case our p_{OS}, $B(\epsilon_\sigma \sigma_{\max}, \hat{b} \sigma_{\max}, Q)$ and N_ϵ do not depend on d_M. We also have r_M constant w.r.t. d_M by our assumption about F_σ.

2. Directly following from the above note, *Corollary A3.5* shows that under the stated assumptions on F_σ, we can ensure our relative truncation error is upper-bounded by any $\epsilon \in (B_{\min}, 1)$ of our choice, with probability p_{OS}, by only retaining a fixed number of modes $(N_\epsilon r_M + 1)$ as d_M increases past $d_M > N_\epsilon r_M$. Importantly, the probability p_{OS} is bounded below by a threshold that is fixed w.r.t. d_M.

3. Of course, the result in *Corollary A3.5* is only useful if we can find a $Q \in \mathbb{Z}^+$ $(Q > \hat{Q})$ for which $B(\epsilon_\sigma \sigma_{\max}, \hat{b} \sigma_{\max}, Q)$ is low enough to allow for practical ϵ values to be reached, and p_{OS} is sufficiently high. This can be ensured if we have $\epsilon_\sigma, \omega, \bar{\omega}, 1 - \hat{b} \ll 1$.

We have seen that the assumptions of *Corollary A3.5* give us that, loosely speaking, if $\epsilon_\sigma, \omega, \bar{\omega}, 1 - \hat{b}$ are "low enough", we can ensure with high probability, that the relative

truncation error can be controlled by ϵ (for practically useful values of ϵ) by only retaining a fixed number of modes (for d_M sufficiently large). However, a generic F_σ that satisfies the assumptions (A3.32) and (A3.33) and yields "sufficiently low" $\epsilon_\sigma, \omega, \bar{\omega}, 1-\hat{b}$ may be hard to comprehend, and the presence of choosable parameters like Q further complicates our message.

In *Example A3.6* we aid our understanding of this matter by looking at a concrete F_σ that satisfies (A3.32) and (A3.33) and yields "sufficiently low" $\epsilon_\sigma, \omega, \bar{\omega}, 1-\hat{b}$. Notably, we need $\mathscr{D}_\sigma$ to concentrate mass below $\epsilon_\sigma \sigma_{\max}$ as d_M increases to maintain a fixed low-rank r_M of M_i with high probability. Intuitively, this is because the typical realization of an i.i.d. sample of size d_M from $\mathscr{D}_\sigma$ will have around as many realizations above $\epsilon_\sigma \sigma_{\max}$ as[4] $\lfloor d_M \cdot P_{\sigma \sim \mathscr{D}_\sigma}(\sigma > \epsilon_\sigma \sigma_{\max}) \rfloor$: and we want that number to stay fixed with d_M if the (noisy and approximate) fixed low-rank assumption is to hold.

Example A3.6 (M_i **assumed to be "almost" low-rank** r_M **(fixed), with high probability for all** d_M**; Concrete Numbers**). *Let us assume that* $\sigma_{\min} = 0$ *and that for a fixed* $0 < \epsilon_\sigma \ll 1$ *and some fixed* $\tilde{r} < r_M$ *we have*

$$F_\sigma(\epsilon_\sigma \sigma_{\max}) = 1 - \frac{\tilde{r}}{d_M}. \tag{A3.45}$$

$$F_\sigma(x) = 1 - \frac{\tilde{r}}{d_M} + \frac{x - \epsilon_\sigma \sigma_{\max}}{(1-\epsilon_\sigma)\sigma_{\max}} \frac{\tilde{r}}{d_M} \quad \forall x \in (\epsilon_\sigma \sigma_{\max}, \sigma_{\max}] \tag{A3.46}$$

such that the mass of $\mathscr{D}_\sigma$ *concentrates below* $\epsilon_\sigma \sigma_{\max}$*, and the mass above* $\epsilon_\sigma \sigma_{\max}$ *is distributed uniformly. We want to compute* ω *and* $\bar{\omega}$ *in*

$$\mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) = 1 - \omega, \quad \textit{and} \quad \mathbb{P}\big(\sigma_{(1)} \geq \hat{b} \sigma_{\max}\big) = 1 - \bar{\omega}, \tag{A3.47}$$

and show ω *and* $\bar{\omega}$ *are fixed for all* d_M *sufficiently large and low. Using the well-known c.d.f. of the* $(r_M+1)^{th}$ *order statistic (see [110]), we have*

$$\mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) = \sum_{j=d_M-r_M}^{d_M} \binom{d_M}{j} [F_\sigma(\epsilon_\sigma \sigma_{\max})]^j [1 - F_\sigma(\epsilon_\sigma \sigma_{\max})]^{d_M - j}, \tag{A3.48}$$

and substituting in (A3.45), we get

$$\mathbb{P}\big(\sigma_{(r_M+1)} \leq \epsilon_\sigma \sigma_{\max}\big) = \sum_{j=d_M-r_M}^{d_M} \binom{d_M}{j} \left[1 - \frac{\tilde{r}}{d_M}\right]^j \left[\frac{\tilde{r}}{d_M}\right]^{d_M - j}. \tag{A3.49}$$

[4]Thus, keeping $\mathscr{D}_\sigma$ fixed will give us increasingly more singular values above $\epsilon_\sigma \sigma_{\max}$.

From (A3.49), one can compute $\lim_{d_M\to\infty}\mathbb{P}\big(\sigma_{(r_M+1)}\leq\epsilon_\sigma\sigma_{\max}\big)$ *to get*

$$\lim_{d_M\to\infty}\mathbb{P}\big(\sigma_{(r_M+1)}\leq\epsilon_\sigma\sigma_{\max}\big)=e^{-(\tilde{r}+1)}\sum_{i=0}^{r_M}\frac{(\tilde{r}+1)^i}{i!}. \tag{A3.50}$$

Note that the sum in (A3.50) is the incomplete taylor expansion of $e^{+(\tilde{r}+1)}$*, and thus higher* $r_M/\tilde{r}$ *gives higher probability. Also note that the series rapidly converges so for a fixed* $\tilde{r}$ *increasing* r_M *past some threshold is practically useless. Equation (A3.50) gives us that in the limit as* $d_M\to\infty$

$$\omega_\infty:=\lim_{d_M\to\infty}\omega=e^{-(\tilde{r}+1)}\sum_{i=r_M+1}^{\infty}\frac{(\tilde{r}+1)^i}{i!}, \tag{A3.51}$$

which rapidly becomes very low for fixed $\tilde{r}$ *as we increase* r_M*.*

Now, we also have

$$\mathbb{P}\big(\sigma_{(1)}\geq\hat{b}\sigma_{\max}\big)=1-[F_\sigma(\hat{b}\sigma_{\max})]^{d_M}=1-\left(1-\frac{\tilde{r}}{d_M}\cdot\frac{1-\hat{b}}{1-\epsilon_\sigma}\right)^{d_M} \tag{A3.52}$$

for any $\hat{b}\in(\epsilon_\sigma,1)$ *(which we decide to choose as a parameter of our bounds, note that the* b *in Corollary A3.4 is* $b=\hat{b}\sigma_{\max}$*). From (A3.52), one can prove compute* $\lim_{d_M\to\infty}\mathbb{P}\big(\sigma_{(1)}\geq\hat{b}\sigma_{\max}\big)$ *to get*

$$\lim_{d_M\to\infty}\mathbb{P}\big(\sigma_{(1)}\geq\hat{b}\sigma_{\max}\big)=1-\exp\left(-\tilde{r}\frac{1-\hat{b}}{1-\epsilon_\sigma}\right). \tag{A3.53}$$

Equation (A3.53) gives us that in the limit as $d_M\to\infty$

$$\bar{\omega}_\infty:=\lim_{d_M\to\infty}\bar{\omega}=\exp\left(-\tilde{r}\frac{1-\hat{b}}{1-\epsilon_\sigma}\right). \tag{A3.54}$$

So we can see that at least in the limit as $d_M\to\infty$*, our assumptions in Corollary A3.5 can be satisfied*[5] *for low* $\omega,\bar{\omega},(1-\hat{b})$ *and some* $r_M>\tilde{r}$ *fixed, if we assume low enough* ϵ_σ *and fixed* $\tilde{r}$ *(and the form of* F_σ *above).*

Now let us further pin down our assumptions to get numbers. Let us assume $\epsilon_\sigma=10^{-3}$ *and* $\tilde{r}=200$*. With these choices, we get*[6] *that* $\omega_\infty\leq5\cdot10^{-4}$ *(*$\omega_\infty\approx5\cdot10^{-4}$*) for an* $r_M:=250$ *(since* ω_∞ *is low for this* $r_M=250$*, we can choose this* r_M *to be the one which gives the approximate rank of* M_i *in Corollary A3.5).*

Let us now make our bounds-parameter choices as $\hat{b}=0.95$*,* $Q=200$*,* $r=r_M$*. And choose the standard* $\rho=0.95$*. These give* $\bar{\omega}_\infty\leq4.5\cdot10^{-5}$ *(* $\bar{\omega}_\infty\approx4.5\cdot10^{-5}$*).*

[5]It is only this limit that we care about because we ultimately want to know how many modes to keep (here argued to be fixed) as $d_M\to\infty$ as this will give us the computational complexity.

[6]Note that the infinite sum in (A3.51) can be computed exactly as $e^{(\tilde{r}+1)}-\sum_{i=0}^{r_M}\frac{(\tilde{r}+1)^i}{i!}$.

Plugging these values in (A3.41) and (A3.43) and noting that this is all in the limit as $d_M \to \infty$ we get

$$p_{OS,\infty} := \lim_{d_M\to\infty} p_{OS} \geq 0.904, \quad \lim_{d_M\to\infty} B_\infty := B(\epsilon_\sigma\sigma_{\max}, \hat{b}\sigma_{\max}, Q) \leq 8\cdot 10^{-4} \quad \text{(A3.55)}$$

(with $B(\epsilon_\sigma\sigma_{\max}, \hat{b}\sigma_{\max}, Q) \approx 8\cdot 10^{-4}$). And one can check that $p_\infty := \lim_{d_M\to\infty} p/p_{OS} = 1$ when we choose the bound-parameter $c > 1$ to be $c = 1 + 1/\sqrt{\ln(d_M)}$ (we had the same calculation in the proof of Remark 3.6).

Now, we need to upper-bound (easier than computing) $\lim_{d_M\to\infty} \mathbb{E}[(\sigma_i^{(j)})^2]$ to substitute in (A3.23). We have

$$\mathbb{E}[(\sigma_i^{(j)})^2] = \mathbb{E}_{\sigma\sim D_\sigma}[\sigma^2] \leq \epsilon_\sigma^2\sigma_{\max}^2\left(1 - \frac{\tilde{r}}{d_M}\right) + \sigma_{\max}^2\frac{\tilde{r}}{d_M}. \quad \text{(A3.56)}$$

and thus

$$\lim_{d_M\to\infty} \mathbb{E}[(\sigma_i^{(j)})^2] \leq \epsilon_\sigma^2\sigma_{\max}^2. \quad \text{(A3.57)}$$

Thus, in the limit as $d_M \to \infty$, with our assumptions in Example A3.6, Corollary A3.4 tells us that for any $\epsilon \geq 8\cdot 10^{-4}$ we have at most $250N_\epsilon$ eigenvalues of $\bar{\mathcal{M}}_k$ above $\lambda_1(\bar{\mathcal{M}}_k)$ with

$$N_\epsilon = \left\lceil \left(\log(\epsilon - 8\cdot 10^{-4}) - \log\left(\frac{\epsilon_\sigma^2 + (1-\rho)}{(1-\rho)\hat{b}^2}\right)\right) / \log(0.95) \right\rceil \approx \left\lceil \frac{\log(\epsilon - 8\cdot 10^{-4})}{\log(0.95)} + 2 \right\rceil. \quad \text{(A3.58)}$$

So for example (in this limit as $d_M \to \infty$, under our assumptions) if we wanted to truncate only eigenvalues of $\bar{\mathcal{M}}_k$ below $0.033\lambda_1(\bar{\mathcal{M}}_k)$ we would need to retain a number of modes of $250\lceil 68.98\rceil = 17250$ and have a success probability of at least 90.4%. Of course 17250 is relatively large and we would hope that this number is lower in practice. However, the example is meant to show that the number of modes we need to keep for any target tolerance (ϵ) above B_∞ is fixed with d_M as $d_M \to \infty$ - under the assumptions we've made. So for a "reasonable" tolerance we can settle for computing only a fixed number of modes (which will mean we can reduce the time complexity from $\mathcal{O}(d_M^3)$ to $\mathcal{O}(d_M^2)$) if the assumptions happen to hold.

Example A3.7. *Finally, in the computation of Example A3.6 above note what would happen if we were to concentrate the mass at of $F_\sigma(\cdot)$ at zero as $d_M \to \infty$ by setting $\epsilon_\sigma = 0$ in the example above. This would give us that $B_\infty \propto \rho^Q$ so by choosing large enough Q we can reduce the minimal attainable truncation error (ϵ) arbitrarily close to zero. Note however, that in order to compensate for the reduced p_{OS} due to higher Q power, we would have to increase the r_M for a fixed $\tilde{r}$ (thus reducing ω_∞, see (A3.51)).*

The point is that under this scenario we can again obtain arbitrarily low truncation accuracy within a fixed (w.r.t. d_M) number of modes, with fixed high probability, as was the case in Section 3.1.2. Of course, this fixed number would increase for stricter truncation accuracy (but would be fixed w.r.t. d_M). Compare this to what we get when $0 < \epsilon_\sigma \ll 1$, as in Example A3.6 above, in which case we can only attain truncation errors above a finite value (within a fixed number of modes w.r.t. d_M and with high probability).

In addition to the assumptions in *Proposition A3.1*, *Corollary A3.5* (and the examples deriving from it) made assumptions on $F_\sigma(\cdot)$ which ensured that M_i is of "approximate rank r_M" (only r_M singular vales of M_i are above the very low threshold $\epsilon_r \sigma_{\max}$) with high (fixed) probability and fixed r_M (fixed w.r.t. d_M). Under this case, we have seen that the number of modes we need to keep $(N_\epsilon r_M + 1)$ for a fixed relative truncation error ϵ (assuming it is attainable: $\epsilon > B(\eta, b, Q)$) is fixed as $d_M \to \infty$. Thus, under this assumption on F_σ, we can reduce the complexity of obtaining the eigendecomposition representation by a power of d_M (while controlling the relative truncation error by ϵ w.h.p.), as we obtained for fixed r_M w.r.t. d_M in *Section 3.1.1* and *3.1.2*.

Notably, because in *Section A3.1* we are sampling d_M i.i.d. singular values of M_i, maintaining an "almost low-rank M_i (fixed w.r.t. d_M)", required the mass of $F_\sigma(\cdot)$ to concentrate below a small value as $d_M \to \infty$. We now study (in *Corollary A3.9* and *Example A3.10*) how the number of modes we need to keep (to ensure a fixed relative truncation error ϵ with fixed high probability) scales with d_M when F_σ is fixed w.r.t. d_M. However, before we present *Corollary A3.9* we simplify some results in the order statistics literature such that they are easily and directly applicable for our case. We present these results in *Lemma A3.8*.

Lemma A3.8 (**Bounds describing convergence of central and extreme order statistics for fixed cumulative distribution function**). *Let $X_1, ..., X_n$ be i.i.d. random variables on some finite support $\mathcal{X} := [x_{\min}, x_{\max}]$, with continuous twice differentiable c.d.f. $F(\cdot)$ and p.d.f. $f(\cdot)$ (fixed with respect to n). Let $X_{(1)} \geq X_{(2)} \geq ...X_{(n)}$ be the order statistics of the set $\{X_i\}_{i=1}^n$. Let $F^{-1}(\cdot)$ be the inverse of $F(\cdot)$. Then, we have*

1. *For any $a \in (0,1)$ s.t. $an > 1$, and any $\kappa > 0$*

$$\mathbb{P}\left(X_{(\lfloor an \rfloor)} \leq F^{-1}(1-a) + \frac{\kappa}{n^{1/3}}\right) \geq 1 - \exp\left(-2\left[\kappa n^{1/3}\left[f\left(F^{-1}(1-a)\right)\right]^2 + \mathcal{O}(1)\right]\right). \tag{A3.59}$$

2. The highest order statistic $X_{(1)}$, satisfies

$$\mathbb{P}(X_{(1)} \geq x_{\max} - \xi n^{-1/2}) \geq 1 - \exp(-\sqrt{n}\xi f(x_{\max})), \tag{A3.60}$$

for any $\xi > 0$ in the limit as $n \to \infty$.

Proof. For *Part 1*, apply *Theorem 2 (at page 4)* in *Xia, 2019 ([112])* to get

$$\mathbb{P}\big(X_{(\lfloor an \rfloor)} \geq F^{-1}(1-a)+t\big) \leq \exp\left(-2n\left[\frac{\lfloor an \rfloor}{n} + F\big(F^{-1}(1-a)+t\big) - 1\right]^2\right), \tag{A3.61}$$

$\forall t > 0$. In applying *Theorem 2* of [112], we noted that our variables are also identical (and not just independently distributed) and that our F is also continuous. Now, applying Taylor's theorem around $F^{-1}(1-a)$, and then choosing $t := \kappa n^{-1/3}$ (any $\kappa > 0$ fixed w.r.t. n), we get

$$F\big(F^{-1}(1-a)+t\big) = 1 - a + f\big(F^{-1}(1-a)\big)t + \mathcal{O}(t^2) \tag{A3.62}$$

$$F\big(F^{-1}(1-a)+\kappa n^{-1/3}\big) = 1 - a + \kappa f\big(F^{-1}(1-a)\big)n^{-1/3} + \mathcal{O}(n^{-2/3}). \tag{A3.63}$$

Substituting (A3.63) into (A3.61) proves the result (A3.59).

For *Part 2*, simply note that

$$\mathbb{P}(X_{(1)} \geq b) = 1 - [F(b)]^n, \tag{A3.64}$$

and then choose $b := x_{\max} - \xi/\sqrt{n}$ for any $\xi > 0$, and apply Taylor's theorem to $F(x_{\max} - \xi/\sqrt{n})$. We get

$$\mathbb{P}(X_{(1)} \geq x_{\max} - \xi/\sqrt{n}) = 1 - \big[1 - \xi n^{-1/2} f(x_{\max}) + \mathcal{O}(n^{-1})\big]^n \asymp 1 - \exp(-\sqrt{n}\xi f(x_{\max})), \tag{A3.65}$$

where "$\asymp$" means "asymptotically behaves as when $n \to \infty$". This finishes the proof. □

When the c.d.f. of X is fixed w.r.t. n, *Part 1* of *Lemma A3.8* can give non-trivial upper-bounds for $X_{(\lfloor an \rfloor)}$ of the form $X_{(\lfloor an \rfloor)} \leq F^{-1}(1-a) + \kappa n^{-1/3} < x_{\max}$ with nonvanishing probability only if $a > 0$ is fixed w.r.t. n. When we apply this to our case[7], when F_σ is fixed w.r.t. d_M, *Part 1* of *Lemma A3.8* can get nontrivial bounds of the form $\sigma_{(r+1)} \leq \eta < \sigma_{\max}$ with nonvanishing probability only if we choose $r + 1 = \lfloor a d_M \rfloor$ for some fixed d_M. That is, *Part 1* of *Lemma A3.8* can only give us nontrivial bounds if we have r roughly proportional to d_M. From our previous discussion we know we cannot do better than this when F_σ is fixed w.r.t. d_M (see *Remark A3.2(9)(b)*).

[7] In our case $x \leftarrow \sigma$, $n \leftarrow d_M$ and $F \leftarrow F_\sigma$.

Corollary A3.9 ($F_\sigma(\cdot)$ **fixed w.r.t.** d_M **means we need to keep a fixed fraction of modes**). *Under the assumptions in Proposition A3.1, let us fix* $F_\sigma(\cdot)$ *w.r.t.* d_M*, and let us assume* F_σ *is continuous, twice differentiable on its support* $[\sigma_{\min}, \sigma_{\max}]$*, and that* $\sigma_{\min} = 0$ *and* $f_\sigma = F'_\sigma$ *is strictly positive on the support* $[\sigma_{\min}, \sigma_{\max}]$*.*

Then, we get that in the limit as $d_M \to \infty$*, the following statements hold:*

1. *Fix a relative truncation error* $\epsilon > B_\infty := \frac{[F_\sigma^{-1}(1-a)]^2}{(1-\rho)\sigma_{\max}^2}$ *and any* $a \in (0,1)$*. We have that with probability one, at most* $\min(N_{\epsilon,\infty}(\lfloor ad_M \rfloor - 1), d_M)$ *eigenvalues of* $\bar{\mathcal{M}}_k$ *are above* $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$*, where*

$$N_{\epsilon,\infty} = \left\lceil \frac{\log(\epsilon - B_\infty) + \log(1-\rho)}{\log(\rho)} \right\rceil. \tag{A3.66}$$

2. *If we desire* $\epsilon \leq B_\infty$*, we need to retain all the* d_M *modes of* $\bar{\mathcal{M}}_k$*.*

Proof. In *Corollary A3.4*, let us choose $\eta = F_\sigma^{-1}(1-a) + \kappa d_M^{-1/3}$ and $b := \sigma_{\max} - \xi d_M^{-1/2}$, for some fixed $a \in (0,1)$, $\kappa, \xi > 0$. Finally, choose $r := \lfloor ad_M \rfloor - 1$ and $Q := 1 + d_M$ (any positive power of d_M give the same result in the limit $d_M \to \infty$).

Substituting these values in *Corollary A3.4*, and taking the limit as $d_M \to \infty$ gives

$$B_\infty := \lim_{d_M \to \infty} B(F^{-1}(1-a) + \kappa d_M^{-1/3}, \sigma_{\max} - \xi d_M^{-1/2}, 1 + d_M) = \frac{[F_\sigma^{-1}(1-a)]^2}{(1-\rho)\sigma_{\max}^2} \tag{A3.67}$$

$$p_{OS,\infty} := \lim_{d_M \to \infty} p_{OS} = 1. \tag{A3.68}$$

In computing the limit (A3.68), we applied the inequality $\mathbb{P}(\mathcal{A}\&\mathcal{B}) \geq \mathbb{P}(\mathcal{A}) + \mathbb{P}(\mathcal{B}) - 1$ and then *Lemma A3.8* to get $p_{OS} \geq (1 - \bar{\omega} - \omega)(1-\omega)^{Q-1}$ with $\bar{\omega} = \exp(-\sqrt{d_M}\xi f_\sigma(\sigma_{\max}))$, $\omega = \exp\big(-2\kappa d_M^{1/3} f_\sigma(F_\sigma^{-1}(1-a)) + \mathcal{O}(1)\big)$, and $Q = 1 + d_M$. We then took the limit as $d_M \to \infty$ noting that f_σ is strictly positive on $[0, \sigma_{\max}]$.

Thus, under the assumed structure of F_σ and the assumptions in *Proposition A3.1*, we get that in the limit as $d_M \to \infty$, for any tolerance $\epsilon > B_\infty = \frac{[F_\sigma^{-1}(1-a)]^2}{(1-\rho)\sigma_{\max}^2}$ at most $\min(N_{\epsilon,\infty}(\lfloor ad_M \rfloor - 1), d_M)$ eigenvalues of $\bar{\mathcal{M}}_k$ are above $\epsilon\lambda_{\max}(\bar{\mathcal{M}}_k)$ with probability $p_{OS,\infty} = 1$, and with

$$N_{\epsilon,\infty} := \lim_{d_M \to \infty} N_\epsilon = \left\lceil \left(\log\left[\epsilon - \frac{[F_\sigma^{-1}(1-a)]^2}{(1-\rho)\sigma_{\max}^2} \right] - \log\left(\frac{1}{(1-\rho)} \right) \right) / \log(\rho) \right\rceil, \tag{A3.69}$$

for any $a \in (0,1)$. In when taking the limit of N_ϵ above we used our choices of $\eta = F^{-1}(1-a) + \kappa d_M^{-1/3}$ in (A3.23) $\square$

Importantly, $N_{\epsilon,\infty}$ in (A3.66) is constant w.r.t. d_M, so *Corollary A3.9* tells us that under the stated assumptions and in the limit as $d_M \to \infty$, we need to keep a fixed fraction of modes to attain a desired relative truncation error $\epsilon > B_\infty$.

Of course, the result in *Corollary A3.9* is only useful when we select an a for which B_∞ is small enough s.t. we can achieve practical values of ϵ, and also get an $N_{\epsilon,\infty}$ for which $a \ll 1/N_{\epsilon,\infty}$ happens to hold. Whether or not such a choice of a can be made depends on the distribution $\mathscr{D}_\sigma$ (through F_σ and $\sigma_{\max}$) and ρ. Note that if F_σ places sufficiently much mass close to zero (but is fixed with d_M, unlike in the previous discussion), then we can choose a small a for which the minimum attainable tolerance $B_\infty = \frac{[F_\sigma^{-1}(1-a)]^2}{(1-\rho)\sigma_{\max}^2}$ is also very small, and thus our small a satisfies $a \ll 1/N_{\epsilon,\infty}$ (s.t. the bound is useful).

Example A3.10 makes a further assumption on F_σ (to make it more concrete), and gives some numbers to show it is possible for F_σ to have sufficient mass around zero s.t. the results of *Corollary A3.9* are useful in practice.

Example A3.10 ($F_\sigma(\cdot)$ **fixed w.r.t.** d_M **(Continued with concrete numbers)).** *In addition to the assumptions in Corollary A3.9, let us assume that* $F_\sigma(\sigma_{\max}/30) = 0.999$*, such that* 99.9% *of the mass of* F_σ *is below* $\sigma_{\max}/30$ *and* 0.1% *of the mass lies in* $[\sigma_{\max}/30, \sigma_{\max}]$*. Let us assume* $\rho = 0.95$ *and choose* $\epsilon = 0.033$*. In this case, by choosing* $a = 10^{-3}$*,* $N_{\epsilon,\infty} = \lceil 146.72 \rceil = 147$*. Thus, Corollary A3.9 now says that with probability one, and in the limit as* $d_M \to \infty$*, at most* $147(\lfloor d_M/1000 \rfloor - 1)$ *eigenvalues of* $\bar{\mathcal{M}}_k$ *are above* $0.033\lambda_{\max}(\bar{\mathcal{M}}_k)$*. That is, we need to keep roughly* 14.7% *of the total number of eigenmodes to ensure a relative truncation error of below* 3.3% *(in the limit as* $d_M \to \infty$*). So by obtaining only these modes directly and efficiently, we can in this case only speed-up obtaining the eigen-representation by a factor of* $1/0.147 \approx 6$*. Of course, the exact factor highly depends on* F_σ *in practice, but the example was only meant to show that there exist* F_σ *which give practically useful results.*

Though *Corollary A3.9* shows a qualitative behavior for which we can reduce the complexity (of obtaining the eigendecomposition representation) by a multiplicative factor, *Example A3.10* demonstrates that under our derived bounds, we might need a very high percentage of the mass of F_σ to be close to zero for this kind of reduction to be possible. In particular *Example A3.10* demonstrates that under the stated assumptions we need 99.9% of the mass of F_σ to be placed close to zero (below $\sigma_{\max}/30$) for us to get away with only computing 14.7% of the total number of modes (thus speeding-up the EVD estimation by about $\times 6$), while controlling the relative truncation error by $\epsilon = 0.033$ with probability 1. This suggests our theory may predict

substantial speed-ups only if a very high percentage of the mass of our fixed F_σ is placed close[8] to zero. However, the numerical results in *Section 3.1.3.3* show that the situation is better in practice (a discrepancy which probably arises from bounds not being tight).

A3.2 The bounded $\sigma_{\max}(M_i)\ \forall i$ Assumption of *S3.1.1*

Our only simplifying assumption for the results in *Section 3.1.1* is that $\sigma_{\max}(M_i)$ is bounded $\forall i$: $\sigma_{\max}(M_i) \in (\beta\sigma_M, \sigma_M)$, $\forall i \in \mathbb{Z}^+$, with some fixed $\sigma_M > 0$ and $\beta \in (0, 1)$.

Our M_i cannot be infinite in practice. Infinite M_i can only come from infinite values in the forward or backward pass, in which case the training cannot work - a much bigger problem than our theory not holding. Technically, an $M_i = 0$ (giving a $\sigma_{\max}(M_i) = 0$) may occur in practice, but it is very unlikely. Even if this happens, we can still get our theory to hold by skipping the EA K-factor update if $M_i = 0$.

Thus, our only real concern can be that there exists a sub-sequence for which $\sigma_{\max}(M_k) \to 0$, and/or one for which $\sigma_{\max}(M_k) \to \infty$. However, even in these cases, some bounds still hold for any finite training period (finite k): so we can still apply our theory even if we had these problematic sub-sequences. The only problem would arise at $k \to \infty$, but we never use this limit for any results in *Section 3.1.1*.

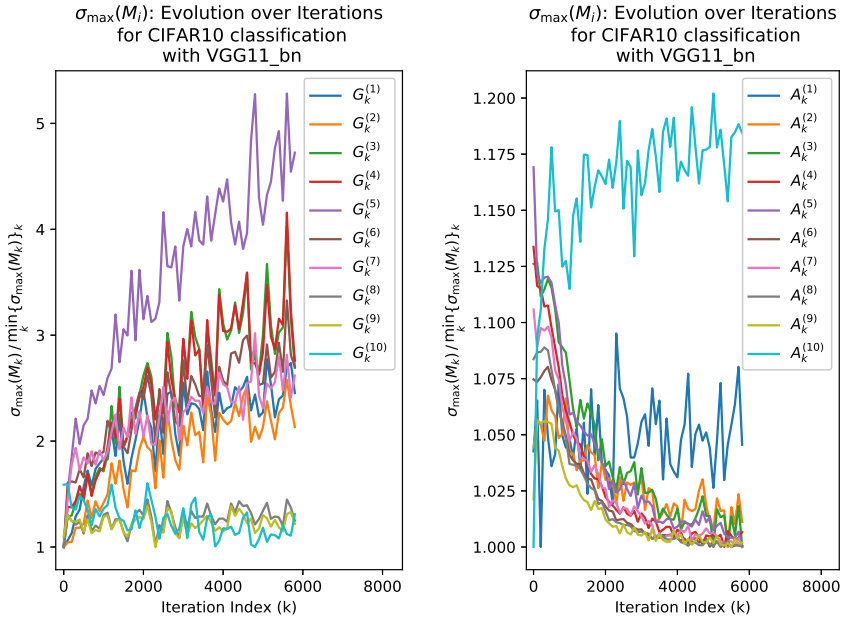


Figure A3.1: CIFAR10 classification with VGG11_bn: Numerical Case-study of $\sigma_{\max}(M_k)$'s Evolution with iterations (k), for all K-factors.

[8] "Close" is relative to $\sigma_{\max}$.

To better justify our assumption (and gauge realistic values of β), we use the numerical set-up of *Section 3.1.3.3* to obtain the maximum singular values of each "M_k" matrix in $\{A_k^{(l)}, G_k^{(l)}\}_{l,k}$ when training CIFAR10 classification with VGG11_bn using K-FAC. We saved the maximum singular values every 100 iterations. That is, for each of the $M_k \in \{A_k^{(l)}, G_k^{(l)}\}_{k\in\mathcal{I}_K, l\in\{1,2,...,10\}}$ (with $\mathcal{I}_k = \{100, 200, ...5800\}$) we have the maximum singular value $\sigma_{\max}(M_k)$. To investigate our assumption, we plot the collections of values $\{\sigma_{\max}(A_k^{(l)})\}_{k\in\mathcal{I}_k}$ and $\{\sigma_{\max}(G_k^{(l)})\}_{k\in\mathcal{I}_k}$ against $\mathcal{I}_k$ for each different layer $l \in \{1, 2, ..., 10\}$. We normalized each of the plots s.t. its lowest value is 1 (by dividing each raw curve by its minimal encountered value). We did this to obtain all plots on a more similar scale. For each K-factor, the value of β we could practically use to apply theory to this particular case is the inverse of the maximum value that each curve achieves. The results are shown in *Figure A3.1.* We see that:

1. In the case of $A_k^{(l)}$ matrices, we have $\sigma_{\max}(A_k^{(l)})$ almost constant as a function of k, not varying more than 20% for the investigated problem in the considered number of iterations.
    - Using the numerical results to imply a β for the considered iterations interval gives $\beta \geq 0.83$ for all $\sigma_{\max}(A_k^{(l)})$ matrices.
2. In the case of $G_k^{(l)}$ matrices, we have $\sigma_{\max}(G_k^{(l)})$ varying slightly more with k, some even by a multiplicative factor of 5. for the investigated problem in the considered number of iterations.
    - Using the numerical results to imply a β for the considered iterations interval gives $\beta \geq 0.18$ for all $\sigma_{\max}(G_k^{(l)})$ matrices.
3. In the case of $A_k^{(l)}$ matrices, there is a slight tendency for $\sigma_{\max}(A_k^{(l)})$ to decrease with k. In the case of $G_k^{(l)}$ matrices, there is a relatively strong tendency for $\sigma_{\max}(G_k^{(l)})$ to increase with k. While these tendencies may potentially be problematic (though not necessarily so) at $k \to \infty$ as they may give $\sigma_{\max}(M_k) \to 0$ and/or $\sigma_{\max}(M_k) \to \infty$ as $k \to \infty$, this does not affect us. We only train for finite k, and our results in *Section 3.1.1* never relied on the limit $k \to \infty$.

A3.3 Observing Low-rank M_i Assumption

For all K-factors, purely based on the shape of $M_i \in \mathbb{R}^{d_M \times r_M}$, we have seen in *Section 3.1.3* that we have $\text{rank}(M_i) \leq r_M$ for fixed constant r_M, for all d_M (in particular useful for $d_M > r_M$). No numerical evidence is required for this argument. Still, in *Figures A3.2* and *A3.3* we provide numerical evidence to observe how our claim holds

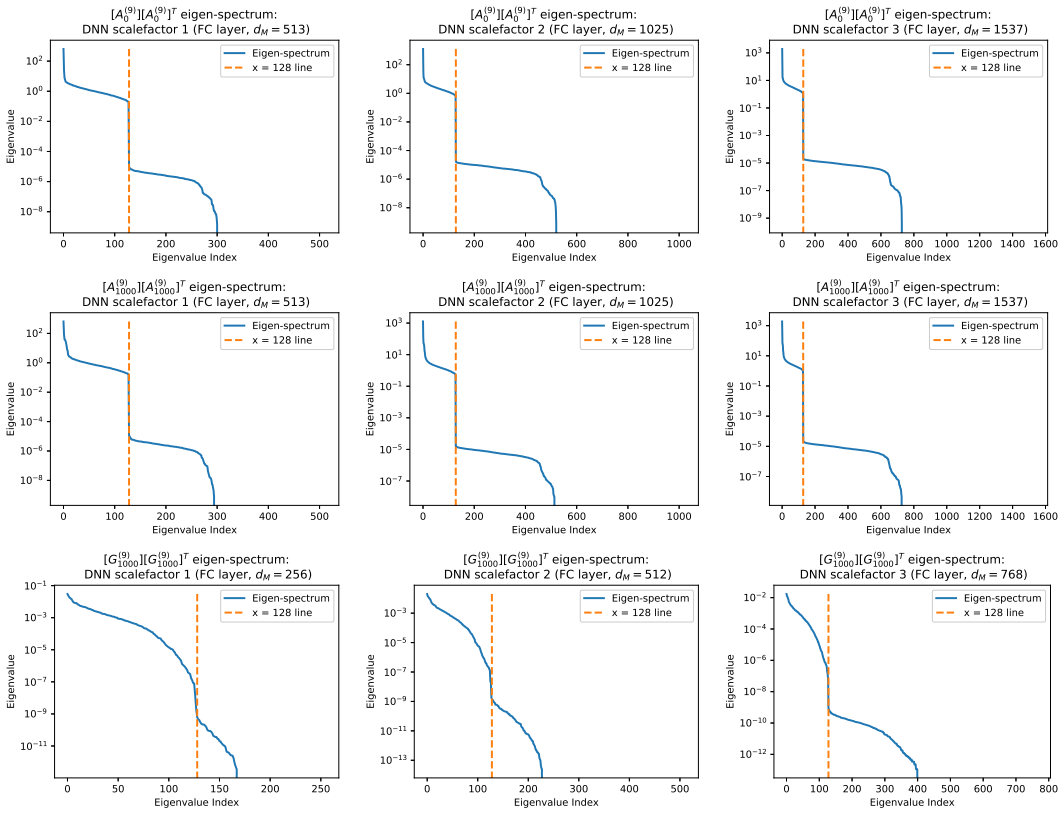


Figure A3.2: Testing Assumption for FC layers: Numerical Experiments. The superscript indicates the layer index. The subscript indicates the iteration index. The DNN scaling factors considered were 1, 2 and 3 - and this varies across columns.

in practice. We choose some particular examples of K-factors, using the numerical set-up of *Section 3.1.3.3.* for net scale-factors of 1, 2 and 3. Note that:

- We show different K-factor type (A and G) at different iterations. We cannot show all K-factors at all iterations. But the qualitative behavior is always the same as highlighted by these figures here.
- The orange vertical line is the $x = n_{\mathrm{B}}$ (=128 here) line: this is the value which upper-bounds the FC K-factors rank (based on matrix shapes).
- The green vertical line is the $x = d_w^{(l)} d_h^{(l)} n_{\mathrm{B}}$ line ($d_w^{(l)} d_h^{(l)}$ in our experiment but the values were variable at different layers). The x-value of this green line is the upper-bound for the Conv K-factors rank (based on matrix shapes).
- The eigenvalues do not become exactly zero at the orange line for FC K-factors and at the green line for Conv K-factors (as they should). This is due to numerical errors, but we can see a sharp drop at the correct index.

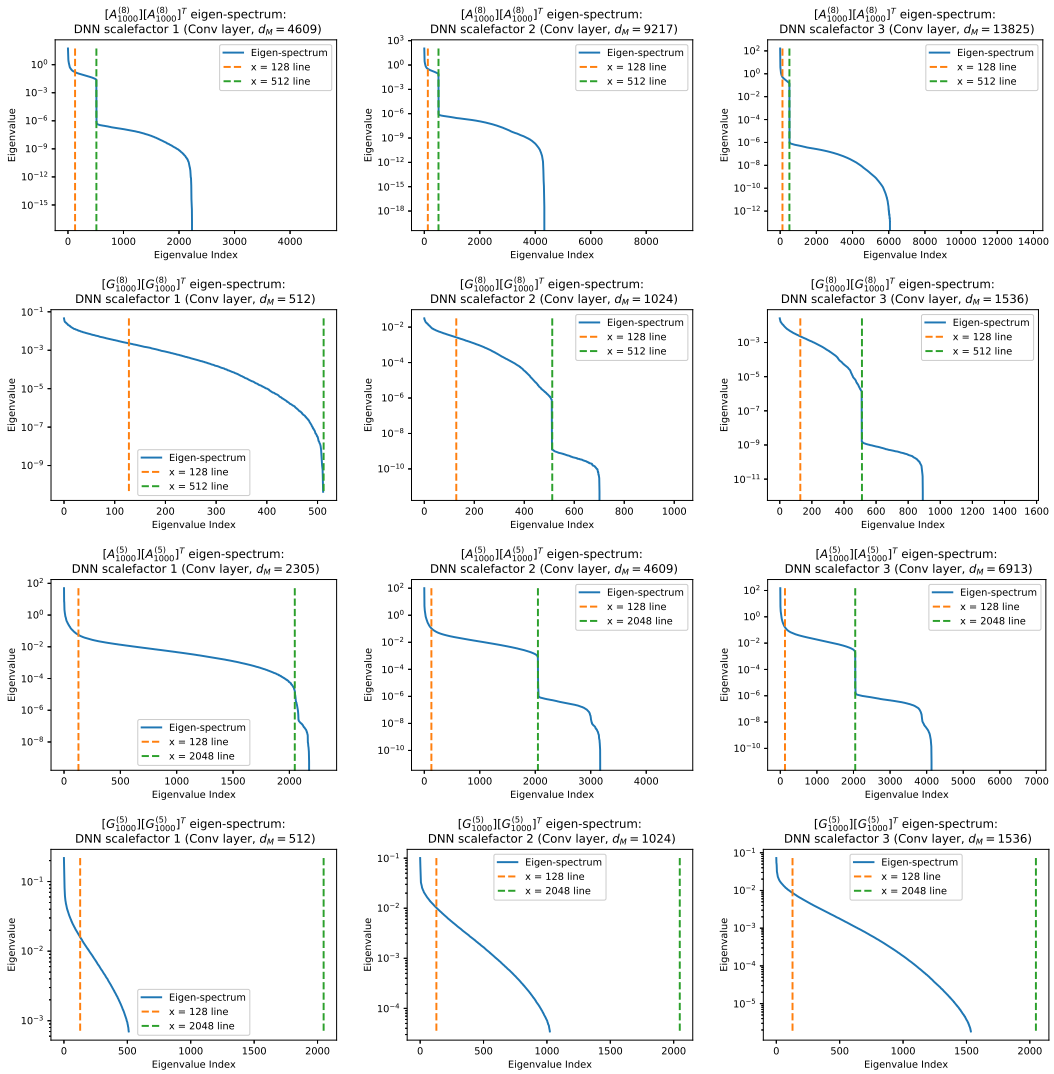


Figure A3.3: The K-Factors eigenspectrum decay: Numerical Assumption testing for Conv layers. The DNN scaling factors considered were 1, 2 and 3 (varying across columns). Only some Conv layers K-factors are practically in the regime where we can use low-rankness due to shape. However, M_i becomes low-rank for sufficiently large d_M in all cases, as provable through shape of matrices (review *Section 3.1.3*).

- We used the eigenvalues of $M_i M_i^T$ rather than the singular values of M_i for convenience.
- The size d_M of each K-factor is stated in the title of the corresponding plot.

Figures A3.2 and *A3.3* show that as expected based purely on the matrices shapes, we have $\text{rank}(M_i) \leq r_M$ for fixed constant r_M, for all d_M (useful for $d_M > r_M$). It also shows that the values of r_M are indeed $r_M = n_\text{B}$ for FC layers K-factors and $r_M = d_w^{(l)} d_h^{(l)} n_\text{B}$ for Conv layers K-factors (as provable purely based on matrix sizes).

A3.4 More Plots for Numerical Case-study in *S3.1.3.3*

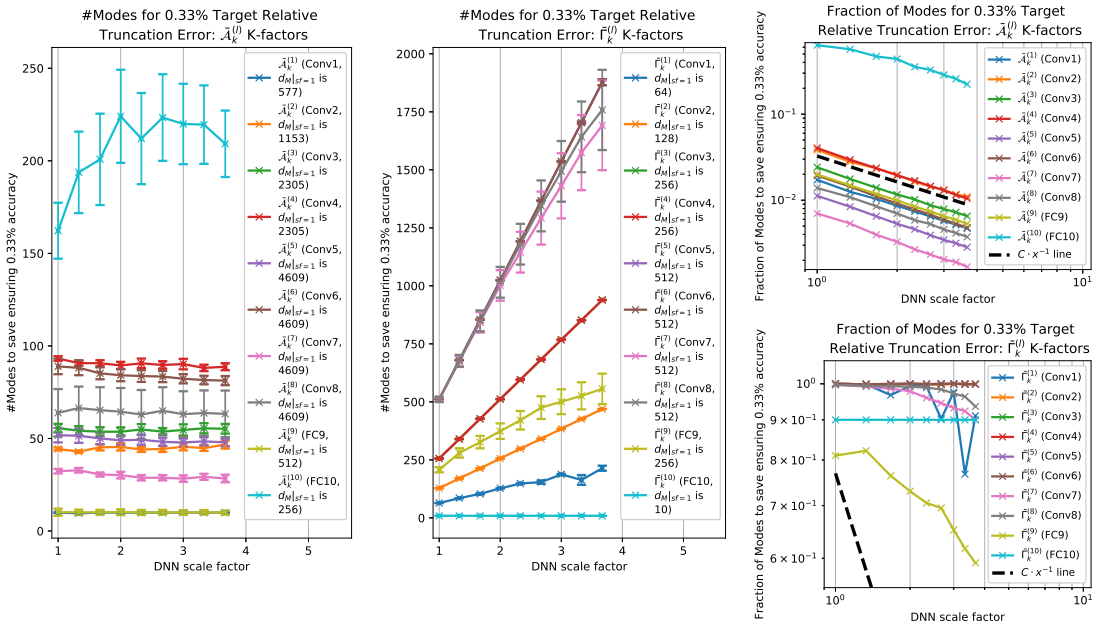


Figure A3.4: #Modes required for 0.33% relative truncation error: Numerical Results for CIFAR10 with VGG11_bn (modified as shown in *Table 3.1*).

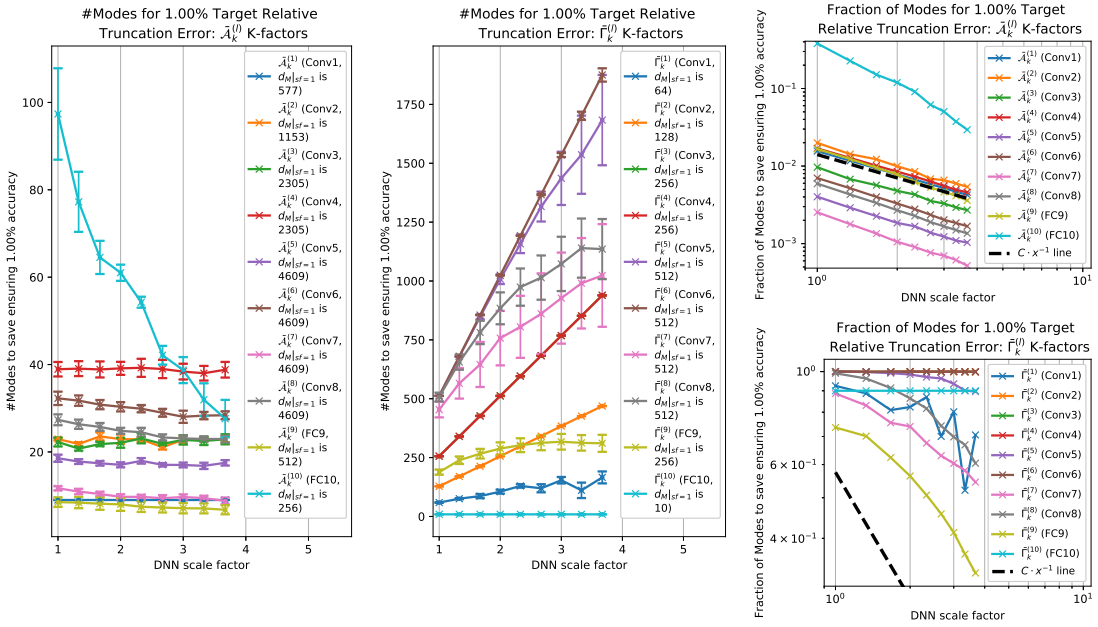


Figure A3.5: #Modes required for 1% relative truncation error: Numerical Results for CIFAR10 with VGG11_bn (modified as shown in *Table 3.1*).

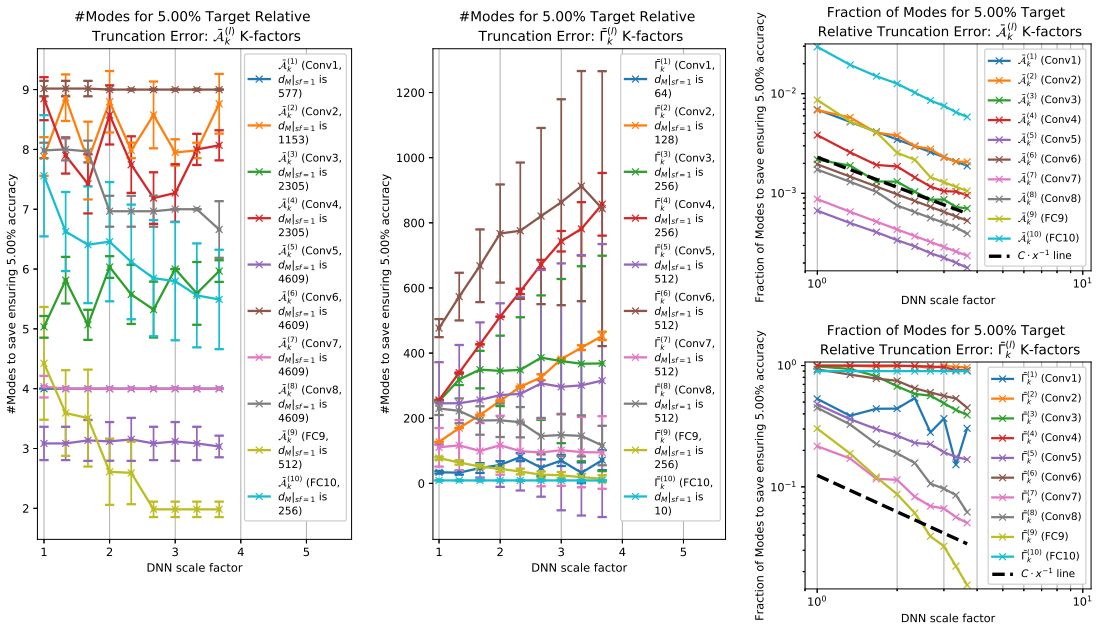


Figure A3.6: #Modes required for 5% relative truncation error: Numerical Results for CIFAR10 with VGG11_bn (modified as shown in *Table 3.1*).

A3.5 Caveats of our *S3.1* Theory

A3.5.1 Practically Implied N_ϵ Values are Much Smaller

On a closer inspection of the implied N_ϵ values in *Table 3.4*, we see that these fluctuate by orders of magnitude, but our theory can only describe cases when N_ϵ is constant. While this appears to contradict the theory, it does not.

Instead, it shows that our bounds are relatively loose in practical settings: there would only be a contradiction if we had implied N_ϵ above the theoretically implied value (68 for *Section 3.1.2* results). Our bounds become particularly loose if all the M_i's align their leading modes. This, together with the much smaller and highly variable implied values of N_ϵ show that our bounds could be improved by finding the right assumption which incorporates the idea of M_i being correlated.

In *Section 3.1.1* we were completely agnostic to this aspect, obtaining worst-case deterministic bounds which hold whether or not M_i are correlated. In *Section 3.1.2* we considered M_i to be i.i.d. (an implication of the structural assumption on M_i). By considering the correlation between M_i, we could do away with Weyl's inequality bounds which we used and were relatively loose due to their worst-case scenario approach (which are tight only when the the incoming modes $M_k M_k^T$ are aligned in the worst possible way relative to the modes we already have $\bar{\mathcal{M}}_{k-1}$).

A3.5.2 Practical $N_\epsilon = f(\epsilon)$ Function: Comparison with Theory

Our theory stated that

1. When assuming $\text{rank}(M_i) \leq r_M$ (constant w.r.t. d_M) we had N_ϵ being a decreasing linear function of $\ln(\epsilon)$ of the form

$$N_\epsilon = \frac{1}{\ln(\rho)} \ln(\epsilon) - \frac{1}{\ln(\rho)} \ln\big(C\big), \tag{A3.70}$$

   where C is a constant that was different depending on the simplifying assumptions. We had $C = [1-\rho]\beta^2$ in *Section 3.1.1* (*Corollary 3.2*), and $C = \sigma^2_{\max}/b^2$ in *Section 3.1.2* (*Corollary 4.6*). Importantly C may be different from K-factor to K-factor, and its best (largest) value we can get is 1 (which we can only get under stricter simplifying assumptions and in the limit as $d_M \to \infty$). Under the more realistic assumptions of *Section 3.1.1* values of C are around $C = 0.05$ (with $\beta \approx 1$ - which is not too far from truth after having investigated all plots as in *Section A3.2*, and the standard value of $\rho = 0.95$).

2. When removing the low-rank M_i assumption (and under some other simplifying assumptions), we had N_ϵ of a slightly different (non-linear) form:

$$N_\epsilon = \frac{1}{\ln(\rho)} \ln(\epsilon - B) - \frac{1}{\ln(\rho)} \ln\big(C\big), \tag{A3.71}$$

   with some constant $B > 0$ (recall our theory works only for $\epsilon > B$).

Importantly, point 1 above tells us that if we plot the practically implied N_ϵ against $\ln(\epsilon)$ for K-factors which are in region III, then we should observe a linear plot.

To investigate whether this holds or not, we take the results presented in *Figure 3.2*, and for each K-factor with relatively constant #Modes (thus practically deemed to live in region III[9]) we compute[10] the practically implied N_ϵ. These results are for $\epsilon = 0.033$. Now, changing the value of ϵ we can get similar plots to the ones in *Figure 3.2*, but for different ϵ values. These would give us different implied N_ϵ. Doing this, we can get a curve implied N_ϵ vs ϵ for each K-factor that appeared to be in region III based on numerical evidence. These plots are shown in *Figure A3.7*. We have:

- The scaling of practically implied N_ϵ with $\ln(\epsilon)$ seems to be relatively close to linear. This qualitatively corresponds to theoretical predictions. To see this, observe the first row of plots in *Figure A3.7*. However, different K-factors

[9] It is much harder to practically infer N_ϵ if the K-factor does not practically fall in region III.
[10] As the maximum N_ϵ ever encountered across all net scale-factors and all iterations.

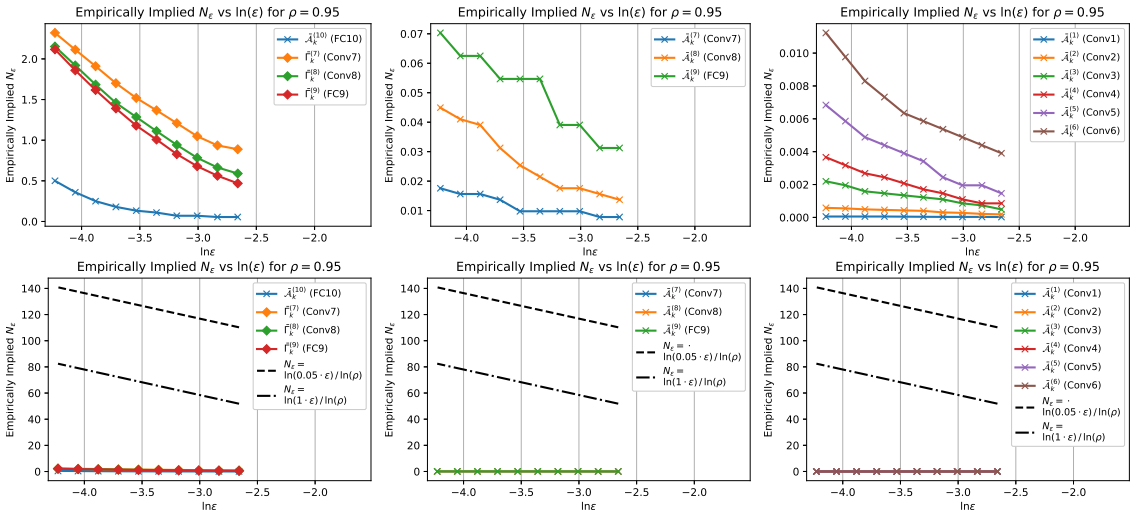


Figure A3.7: Empirically Implied N_ϵ as a function of $\ln \epsilon$.

appear to have different slopes, and based on the theory using the low-rank M_i assumption (see point 1 above), we cannot explain different slopes, but only different intercepts. This details suggests we could improve our theory, which remains as subject for future work. Note that different slope for different K-factors (when theory says there should only be one) is analogous to what we discussed in *Section A3.5.1* (and is probably because the bounds are loose).

- When plotting the theoretical $N_\epsilon = f(\epsilon)$ with realistic C values, we get N_ϵ of orders of magnitude larger than the practically implied ones. To see this, observe the second row of plots in *Figure A3.7*. Based on our discussion in *Section 3.1.3.3*, this is not surprising. This clearly shows our bounds are very loose, and that these may be improved by further adding in assumptions which closely describe structural aspects of actual K-factors that were omitted in our theory.

- Also, the slope of practically obtained curves appears to increase slightly at lower $\ln(\epsilon)$ for some K-factors. This behavior is more consistent with the theory discussed in *Section A3.1*, where we remove the low-rank M_i assumption (see point 2 above). Still, the theory which tells us N_ϵ is a linear function of $\ln(\epsilon)$ relies on relatively loose bounds[11]. Furthermore, the slope change is relatively low in most cases, and our practically implied N_ϵ is just an empirical estimate. Considering all this, we cannot expect that we get perfectly straight lines in

[11]Which only account for the singular-spectrum of M_i through its maximum singular-value, under the most realistic simplifying assumptions in *Section 3.1.1*; these type of bounds are particularly loose around the first n_B modes of the K-factor (i.e.) for $N_\epsilon < 1$ because it is essentially treating all the singular values of M_i as being equal to the largest one.

practice, even in cases when theory holds. We need further investigations to draw conclusions based on the observation in this point.

Over-all our low-rank M_i theory seems to reasonably capture the qualitative behavior, but it overestimates N_ϵ, and fails to capture some fine details (like allowing different slopes for different K-factors and higher slope for lower ϵ values). This all suggests our bounds are relatively loose, and that we should improve our theory by adding in further assumptions which closely describe structural aspects of actual K-factors that were omitted in this thesis. We leave this as future work. For the purpose of this thesis, the theory presented here reconciles with practice sufficiently well, as argued in the main body.

A3.6 Numerical Investigation of rNLA Projection Error

To study the projection error of RSVD and SREVD for our particular practical set-ups, we run numerical experiments with the same set-up we had in *Section 3.1.3.3* (CIFAR10 classification with our slightly modified VGG11_bn). We run K-FAC for 30 epochs with $T_{\text{Cov}} = 20$ and $T_{\text{Inv}} = 100$. Each time we update the EVD of our K-factors (once in 100 iterations) we also use the eigenspectrum to get the minimum number of modes r for which our truncation error is below $\epsilon = 0.033$.

We then use this r as our target rank in an RSVD with an oversampling parameter of $r_o = 10$. Then, we compute the 2-norm error between the RSVD representation and the K-factor itself. We also use the eigendecomposition to obtain the (optimal) truncation error for our r (which is just the largest omitted eigenvalue). This will be close to and just under $\epsilon = 0.033$ by our methodology. Finally, we compute the multiplicative factor (c) by which our total error is larger than our (optimal) truncation error. We know the discrepancy comes from the projection error. A projection error of zero corresponds to $c = 1$. We do the same for SREVD as we did for RSVD.

For each K-factor, we are then left with an empirical distribution of c's, with different draws coming at different optimization iterations. For both RSVD and SREVD, we plot histograms of c for each K-factors for which our $r + r_o$ was below the K-factor size. The results for RSVD are shown and commented in *Section A3.6.1*, and the ones for SREVD are in *Section A3.6.2*. We set $n_{\text{pwr-it}} = 4$, the same as for our performance experiments in *Chapter 3*.

A3.6.1 Numerical Results: RSVD Projection Error

The results for RSVD are shown in *Figures A3.8 - A3.9*. Our results show we typically had $c < 1.35$ most of the time (our total error was most of the time below 1.4 times the optimal truncation error), and we always had $c < 3.2$. The empirical expected value of c is always significantly below the theoretical bounds in *Section 3.2.4* (which was 2.88) in all cases. In fact, our worst outlier value ($c = 3$) only mildly exceeds the theoretical expected value bound. This is because our theoretical bounds used in *Section 3.2.3* virtually assumed there is no decay in the ignored eigenspectrum part (i.e. $\lambda_{r+1} = \lambda_{r+2} = ...$).

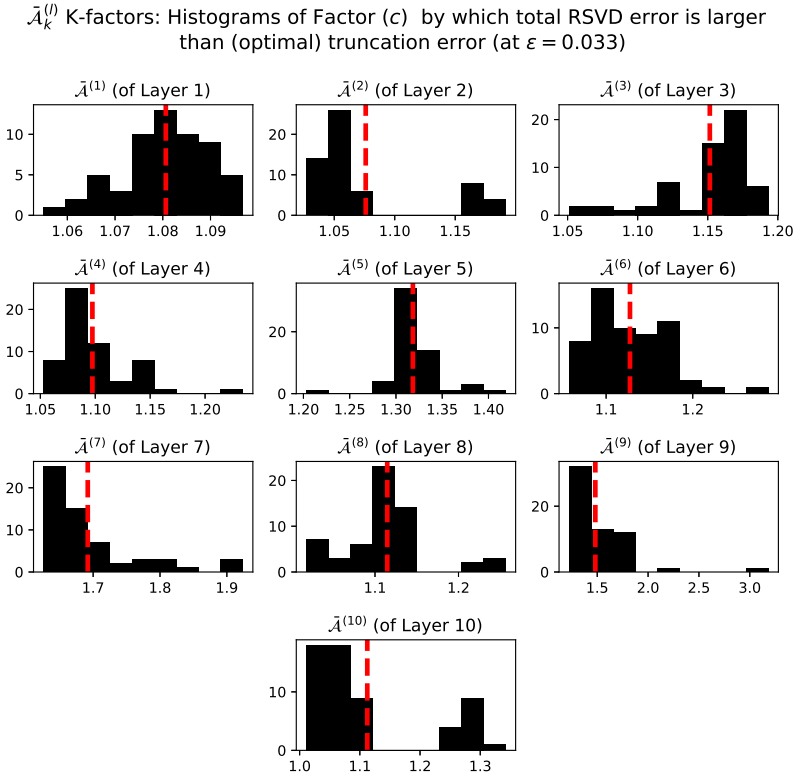


Figure A3.8: CIFAR10 with VGG11_bn: Projection error of RSVD for $\bar{\mathcal{A}}_k^{(l)}$ K-factors.

Importantly, we see the measure is concentrated around the expectation. This was stated in the main body and in the rNLA error theory but lacked a rigorous discussion.

Having $c < 1.35$ most of the time is remarkable (the projection error is at most 35% of the truncation error). That is, for a target relative (optimal) truncation error of 3.3%, we get a relative total error of at most 4.46% (with our chosen RSVD hyperparameters).

Thus, we see that with high (empirical) probability, our RSVD projection error is effectively negligible for $n_{\text{pwr-it}} = 4$ and $r_o = 10$ in this case-study (better than implied

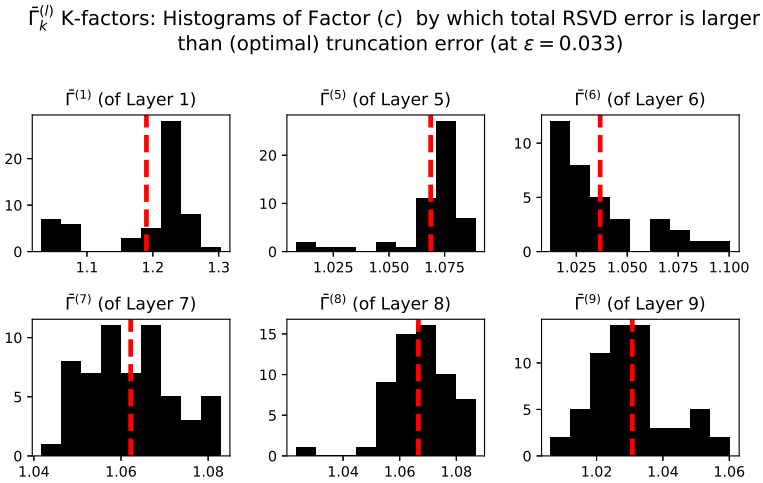


Figure A3.9: CIFAR10 with VGG11_bn: Projection error of RSVD for $\bar{\Gamma}_k^{(l)}$ K-factors.

by error theory). Over-all, combining with our theoretical guarantees of *Section 3.2.4*, we should expect to have low (or even negligible) RSVD projection errors (with the stated hyperparameters) in standard DNN K-FAC training settings.

A3.6.2 Numerical Results: SREVD Projection Error

The results for SREVD are shown in *Figures A3.10 - A3.11*.

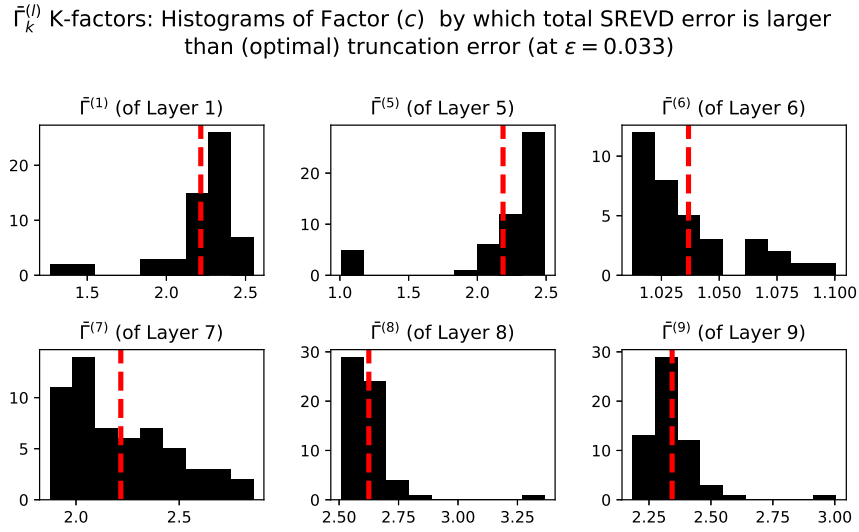


Figure A3.10: CIFAR10 with VGG11_bn: Projection error of SREVD for $\bar{\Gamma}_k^{(l)}$ K-factors. The red dashed line indicates the empirical mean.

For SREVD, we see that the factor c is typically below 3, and it never reaches values larger than 5. These values are significantly larger than the ones of RSVD - which was expected from the outset (as stated in *Section 2.2*). Note that we had no theory for controlling SREVD projection error (though this could be done). However, somewhat confusingly, the empirical expected values of constant c for SREVD are about the same as the theoretical expected values bounds for RSVD (again showing the error projection theory is pessimistic).

We saw in *Section 3.2.3* that a value of $c \leq 3$ is acceptable, and for SREVD we get this most of the times. Even in the worst-case scenarios where we get $c \approx 5$, our total relative error would still be controlled below 16.5%, when controlling our (optimal) truncation error below 3.3%. Given they rarely occur, these values of c larger than 3 should not affect us much in practice.

Finally, note that the measure concentrates around the empirical expectation fairly tightly for c values of SREVD as well.

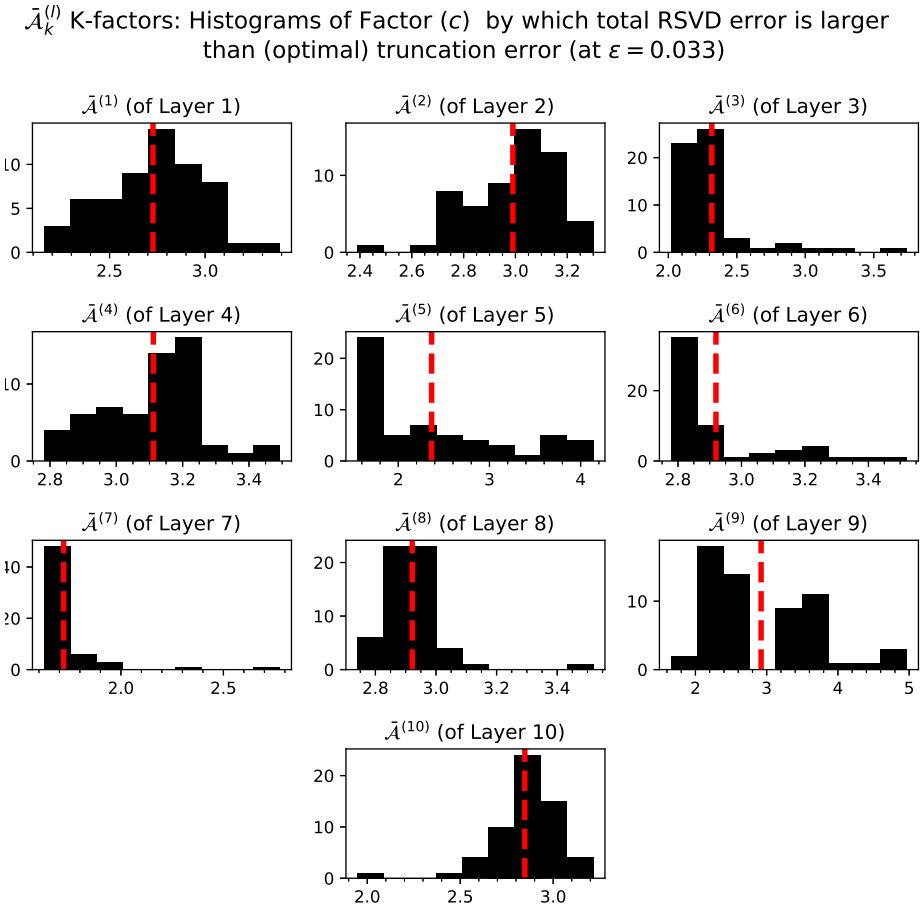


Figure A3.11: CIFAR10 with VGG11_bn: Projection error of SREVD for $\bar{\mathcal{A}}_k^{(l)}$ K-factors. The red dashed line indicates the empirical mean.

Appendix A4

Appendix of Chapter 4

This appendix contains supplementary material for *Chapter 4*, which is mainly made of proofs. We first recall useful equations (from the main body) for the proofs below.

The EA K-factor. In *Chapter 3*, we saw that an arbitrary EA K-Factor $\mathcal{M}_k$ (may be either $\bar{\mathcal{A}}_k^{(l)}$ or $\bar{\Gamma}_k^{(l)}$ for any l) where we have incoming (random) updates $M_k M_k^T$ with $M_k \in \mathbb{R}^{d_M \times n_M}$ at iteration i (M_k may be either $A_k^{(l)}$ or $G_k^{(l)}$) follows the process[1]

$$\bar{\mathcal{M}}_0 = 0, \;\; \bar{\mathcal{M}}_j = \rho \bar{\mathcal{M}}_{j-1} + (1-\rho) M_j M_j^T \;\; \forall j \geq 1, \tag{A4.1}$$

and can alternatively be written as

$$\bar{\mathcal{M}}_k = \rho^k \bar{\mathcal{M}}_0 + (1-\rho) \sum_{j=1}^{k} \rho^{k-j} M_j M_j^T = (1-\rho) \sum_{j=1}^{k} \rho^{k-j} M_j M_j^T, \tag{A4.2}$$

where again we used the fact that our initialization is zero: $\bar{\mathcal{M}}_0 = 0$ (and this incorporates the case where we initialize from an identity multiple, which can be dealt with by sending this identity multiple in the regularization term; recall *Section 3.1.1*).

The B-process. We now also reiterate the mathematical definition of the B-process (4.3) for convenience. The B-KFAC algorithm implicitly estimates $\bar{\mathcal{M}}_k$ as $\tilde{\mathcal{M}}_k^{(B)}$ if we choose to truncate just before applying the B-update or as $\mathcal{B}_k$ if we choose to truncate just after applying the B-update (recall *Section 4.1.1*). The matrices $\tilde{\mathcal{M}}_k^{(B)}$ and $\mathcal{B}_k$ are given by the repeated applications of the B-update and rank-r truncation, as follows:

$$\begin{aligned} \tilde{\mathcal{M}}_{i+1}^{(B)} &:= \rho \mathcal{B}_i + (1-\rho) M_{i+1} M_{i+1}^T \, \forall i \geq 0, \\ \text{with } \tilde{\mathcal{M}}_0^{(B)} = 0, \;\; U_{\tilde{\mathcal{M}}_i^{(B)}} D_{\tilde{\mathcal{M}}_i^{(B)}} U_{\tilde{\mathcal{M}}_i^{(B)}}^T &\overset{\text{SVD}}{=} \tilde{\mathcal{M}}_i^{(B)}, \; U_{\tilde{\mathcal{M}}_i^{(B)},r} := U_{\tilde{\mathcal{M}}_i^{(B)}}[:, :r]; \\ \mathcal{B}_i &:= U_{\tilde{\mathcal{M}}_i^{(B)},r} U_{\tilde{\mathcal{M}}_i^{(B)},r}^T \tilde{\mathcal{M}}_i^{(B)} U_{\tilde{\mathcal{M}}_i^{(B)},r} U_{\tilde{\mathcal{M}}_i^{(B)},r}^T \;\; \forall i \geq 0. \end{aligned} \tag{A4.3}$$

[1] In practice we initialize our EA K-factor from identity, I. But in *Chapter 3* we saw this is equivalent to having an $\bar{\mathcal{M}}_0 = 0$ initialization with an extra $+\rho^k I$ in the regularization. We could thus set $\bar{\mathcal{M}}_0 = 0$ and relate our theory to this without modifying the standard ()EA) K-FAC algorithm.

A4.1 Proof of Example 4.1

We reiterate *Example 4.1* here for convenience (re-stated as *Example A4.1*).

Example A4.1 (**Particular 2D case: Excess Error of B-KFAC may be large, with small but non-negligible probability**). *Let us consider applying* B-KFAC *with a target-rank of* $r = 1$, $d_M = 2$ *and* $n_M = 1$. *We use zero initialization for both the EA K-factor and the* B-KFAC *representation*[2]. *Let us further assume that the matrices* $M_j M_j^T \in \mathbb{R}^{2\times 2}$ *are i.i.d. and of the form*

$$M_j M_j^T \overset{\text{SVD}}{=} \psi_j u_j u_j^T, \tag{A4.4}$$

with $u_j \in \mathbb{R}^{2\times 1}$ *i.i.d. and distributed according to*

$$\mathbb{P}(u_j = u^{(1)}) = p_1, \quad \textit{and} \quad \mathbb{P}(u_j = u^{(2)}) = 1 - p_1 \tag{A4.5}$$

for some[3] $p_1 \in (0,1)$ *and with the matrix* $[u^{(1)}, u^{(2)}] \in \mathbb{R}^{2\times 2}$ *being orthonormal. Our* ψ_j*'s are assumed to be i.i.d. and independent of* u_j, *with positive measure for every point on* $\mathbb{R}^+$. *Now, let us choose some* $c_k \in \mathbb{Z}^+$. *Let us then further choose some* ψ_m, $\psi_M > 0$, $\psi_{1,m}$, $\psi_{1,M} > 0$ *s.t.*

$$0 < \psi_m < \psi_M < \psi_{1,m} < \psi_{1,M} \textit{ and } \psi_M < \rho^{c_K}\psi_{1,m}. \tag{A4.6}$$

We then have that the ratio between the (2-norm) error of B-KFAC *and the (2-norm) optimal truncation error for* $r = 1$ *is lower-bounded as follows*

$$EEF_k^{(B)} := \frac{\left\|\bar{\mathcal{M}}_k - \mathcal{B}_k\right\|_2}{\min_{v\in\mathbb{R}^{2\times 1}}\left\|\bar{\mathcal{M}}_k - vv^T\right\|_2} \geq \frac{\rho^{-(k-1)} - 1}{1-\rho}\frac{\psi_m}{\psi_{1,M}} \quad \forall k \in \{k^*, k^*+1, ..., c_K\}, \tag{A4.7}$$

with probability at least

$$p_k = p_1(1-p_1)^{k-1}\mathbb{P}\big[\psi_j \in (\psi_{1,m}, \psi_{1,M})\big]\left(\mathbb{P}\big[\psi_j \in (\psi_m, \psi_M)\big]\right)^{k-1}, \tag{A4.8}$$

where

$$k^* := \left\lceil 1 + \frac{\ln\left(1 + (1-\rho)\frac{\psi_{1,M}}{\psi_m}\right)}{\ln\left(\frac{1}{\rho}\right)}\right\rceil. \tag{A4.9}$$

Proof. Consider that for $j \in \{1, 2, ..., k\}$ we get the draws: $u_1 = u^{(1)}$ and $u_j = u^{(2)}$ $\forall j \in \{2, 3, ..., k\}$. We have

$$\mathbb{P}\big(u_1 = u^{(1)} \,\&\, u_j = u^{(2)}\,\forall j \in \{2, 3, ..., k\}\big) = p_1(1-p_1)^{k-1}, \tag{A4.10}$$

[2]By sending the Identity initialization into the reg term, as before (see *Chapter 3*)

[3]Think p_1 small to aid understanding but results hold anyway.

by independence u_j and our definition of p_1 in (A4.5). Further consider the draws $\psi_1 \in (\psi_{1,m}, \psi_{1,M})$ and $\psi_j \in (\psi_m, \psi_M)$ $\forall j \in \{2, ..., k\}$ with our constants ψ_m, $\psi_M > 0$, $\psi_{1,m}$, $\psi_{1,M} > 0$ satisfying the conditions (A4.6). Using the independence between ψ_j and u_j, and the fact that ψ_j are i.i.d, we get that the event when all these draws occur simultaneously, defined as

$$\Psi_k := \left\{ \omega \in \Omega \,\middle|\, u_1(\omega) = u^{(1)}, \psi_1(\omega) \in (\psi_{1,m}, \psi_{1,M}) \right. \\ \left. \& \, \forall j \in \{2, 3, ..., k\} : u_j(\omega) = u^{(2)}, \psi_j(\omega) \in (\psi_m, \psi_M) \right\} \tag{A4.11}$$

holds with probability

$$p_k = p_1(1-p_1)^{k-1} \mathbb{P}\big[\psi_j \in (\psi_{1,m}, \psi_{1,M})\big] \big(\mathbb{P}\big[\psi_j \in (\psi_m, \psi_M)\big]\big)^{k-1}. \tag{A4.12}$$

Note that this holds for any $k \geq 2$.

We now want to compute the SVD of $\bar{\mathcal{M}}_k$ and $\tilde{\mathcal{M}}_k^{(B)}$, and from there, deduce the required error ratio $\text{EEF}_k^{(B)}$ for a rank-1 representation. We do this under event Ψ_k.

Under event Ψ_k, we have that our EA K-factor is (recalling we initialize $\bar{\mathcal{M}}_0 = 0$)

$$\bar{\mathcal{M}}_k = (1-\rho)\left[\rho^{k-1}\psi_1 u^{(1)}[u^{(1)}]^T + u^{(2)}\left(\sum_{i=2}^{k} \rho^{k-i}\psi_i\right)[u^{(2)}]^T\right]. \tag{A4.13}$$

Note that (A4.13) is also the SVD (and also the EVD) of $\bar{\mathcal{M}}_k$, due to the fact that $[u^{(1)}, u^{(2)}] \in \mathbb{R}^{2\times 2}$ is orthonormal as established in *Example A4.1* statement.

Now, the eigenvalue $(1-\rho)\rho^{k-1}\psi_1$ of the first mode $u^{(1)}$ of $\bar{\mathcal{M}}_k$ decreases with k. Under Ψ_k, using the bounds on $\{\psi_j\}_{j=2}^k$, the eigenvalue $(1-\rho)\big(\sum_{i=2}^k \rho^{k-i}\psi_i\big)$ of the second mode $u^{(2)}$ of $\bar{\mathcal{M}}_k$ is lower-bounded as

$$\left(\sum_{i=2}^{k} \rho^{k-i}\psi_i\right) \geq (1-\rho)\frac{1-\rho^{k-1}}{1-\rho}\psi_m. \tag{A4.14}$$

Notably, the r.h.s. of our lower-bound (A4.14) increases with k. Thus, for all k s.t.

$$(1-\rho)\frac{1-\rho^{k-1}}{1-\rho}\psi_m \geq (1-\rho)\rho^{k-1}\psi_{1,M} \tag{A4.15}$$

holds we have that our optimal rank-1 representation of $\bar{\mathcal{M}}_k$ is choosing the second mode, since $(1-\rho)\rho^{k-1}\psi_{1,M} \geq (1-\rho)\rho^{k-1}\psi_1$ holds under Ψ_k. That is, $\forall k \geq k^*$, with

$$k^* = \left\lceil 1 + \frac{\ln\left(1 + (1-\rho)\frac{\psi_{1,M}}{\psi_m}\right)}{\ln\left(\frac{1}{\rho}\right)} \right\rceil \tag{A4.16}$$

the 2-norm optimal truncation error is the first eigenvalue of $\bar{\mathcal{M}}_k$. Using the upper-bound on ψ_1 under Ψ_k, this 2-norm optimal truncation error can be bounded as

$$\min_{v\in\mathbb{R}^{2\times 1}} \left\|\bar{\mathcal{M}}_k - vv^T\right\|_2 = (1-\rho)\rho^{k-1}\psi_1 \leq (1-\rho)\rho^{k-1}\psi_{1,M}, \quad \forall k \in \{k^*, k^*+1, ...\}, \tag{A4.17}$$

whenever Ψ_k holds.

We now go through a similar calculation for the SVD and error of the B-KFAC representation $\tilde{\mathcal{M}}_k^{(B)}$ (again under Ψ_k), then conclude. This time however, we will need to restrict ourselves to $k \leq c_K$ (note that for the EA K-factor we needed to restrict ourselves to $k \geq k^*$). This is how the $\forall k \in \{k^*, k^*+1, ..., c_K\}$ comes about in (A4.7), when combining the results. For our B-KFAC error, we need to follow the B-KFAC process step-by-step, according to (4.3) and the M_j draws under Ψ_k. We have

$$\tilde{\mathcal{M}}_1^{(B)} = (1-\rho)\psi_1 u^{(1)}[u^{(1)}]^T, \tag{A4.18}$$

which also happens to equal the true EA K-factor $\bar{\mathcal{M}}_k$. We also have $\mathcal{B}_1 = \tilde{\mathcal{M}}_1^{(B)}$ since $\tilde{\mathcal{M}}^{(B)}$ is rank-1. Going further (under Ψ_k), we have

$$\tilde{\mathcal{M}}_2^{(B)} = \rho\mathcal{B}_1 + (1-\rho)\psi_2 u^{(2)}[u^{(2)}]^T = (1-\rho)\rho\psi_1 u^{(1)}[u^{(1)}]^T + (1-\rho)\psi_2 u^{(2)}[u^{(2)}]^T, \tag{A4.19}$$

Now using conditions (A4.6) and the bounds on ψ_2 in Ψ_k, we get

$$\psi_2 \leq \psi_M \leq \rho^{c_K}\psi_{1,m} < \rho^{c_K}\psi_1 \leq \rho\psi_1, \tag{A4.20}$$

where the second inequality holds by our imposed conditions (A4.6), and the last one is because $c_k \geq 1$ (A4.19). Thus, when truncating $\tilde{\mathcal{M}}_2^{(B)}$ to rank-1, we get

$$\mathcal{B}_2 = (1-\rho)\rho\psi_1 u^{(1)}[u^{(1)}]^T. \tag{A4.21}$$

Repeating the argument $c_K - 1$ times (noting that when dealing with $\mathcal{B}_i$ we have $\rho\psi_1 \leftarrow \rho^i\psi_1$ in equations (A4.19)- (A4.20), and this only holds surely for all $i \leq c_k$), we get that

$$\mathcal{B}_k = (1-\rho)\rho^{k-1}\psi_1 u^{(1)}[u^{(1)}]^T, \ \forall k \in \{1, 2, ..., c_K\} \tag{A4.22}$$

Thus, we have that the B-KFAC 2-norm error is lower-bounded as

$$\left\|\mathcal{E}_k^{(B)}\right\|_2 := \left\|\bar{\mathcal{M}}_k - \mathcal{B}_k\right\|_2 = \left\|u^{(2)}\left(\sum_{i=2}^{k}\rho^{k-i}\psi_i\right)[u^{(2)}]^T\right\|_2 \geq (1-\rho)\frac{1-\rho^{k-1}}{1-\rho}\psi_m, \quad \forall k \in \{1, 2, ..., c_K\}. \tag{A4.23}$$

In the first equality of (A4.23) we plugged in (A4.13) for $\bar{\mathcal{M}}_k$ and (A4.22) for $\mathcal{B}_k$. For the inequality we used the lower-bound $\psi_j \geq \psi_m \ \forall j \in \{2, 3, ...k\}$ which holds in Ψ_k.

Revealing aside: As a revealing aside, note how our B-KFAC is retaining the first mode (for $k \leq c_K$) while the optimal rank-1 representation retained the second mode

(for $k \geq k^*$). Also note that $\mathcal{B}_k$ did not incorporate any of the information in $\{M_j\}_{j=2}^{k}$ (for all $k \leq c_K$) under Ψ_k. Intuitively, this is why the B-KFAC error will prove to increase (substantially) under Ψ_k: our B-KFAC representation holds onto the first mode for a long while. Though decaying, B-KFAC retains this mode because none of the $\{M_j M_j\}_{j=2}^{c_K}$ are large enough to exceed the first mode eigenvalue $(1-\rho)\rho^{k-1}\psi_1$ in isolation. However because $\{M_j M_j\}_{j=2}^{c_K}$ give contributions that stack-up in the true EA K-factor (unlike in the B-KFAC representation where these are completely deleted for not being big enough in isolation), the optimal representation switches to being the one along $u^{(2)}$ for any $k \geq k^*$. Thus, if the interval $[k^*, c_K]$ is large enough, we get the error of B-KFAC increasing to large values under Ψ_k (for $k \in [k^*, c_K]$) under Ψ_k. The probability does reduce slightly as the error increases (recall p_k definition (A4.12)). However, for heavy-tailed distributions of M_j, we shall expect this probability to be non-negligible (as we shall see in *Example 4.2*).

Back to the Proof. Combining (A4.17) and (A4.23), taking care that (A4.17) requires $k \geq k^*$ (and (A4.23) required $k \leq c_K$: thus intersecting the sets), we get that

$$\text{EEF}_k^{(B)} := \frac{\left\|\mathcal{E}_k^{(B)}\right\|_2}{\min_{v\in\mathbb{R}^{2\times 1}}\left\|\bar{\mathcal{M}}_k - vv^T\right\|_2} \geq \frac{\rho^{-(k-1)}-1}{1-\rho}\frac{\psi_m}{\psi_{1,M}} \quad \forall k \in \{k^*, k^*+1, ...c_K\}, \tag{A4.24}$$

holds whenever event Ψ_k holds. Since event Ψ_k holds with probability p_k, and (A4.24) may hold for events other than Ψ_k as well, we have that (A4.24) holds with probability at least p_k as in (A4.12). This is the desired statement and completes the proof. □

A4.2 Proof of Proposition 4.3

We reiterate *Proposition 4.3* here for convenience (re-stated as *Proposition A4.2*).

Proposition A4.2 (Form of errors for $\tilde{\mathcal{M}}_k$ and its corrected version $\tilde{\mathcal{M}}_k^{(c)}$; correction using *Algorithm 4.3*). *Let*[4] $n_{crc} < r < d_M - n_M < d_M$. *Let an arbitrary SPSD rank-r ($r < d_M - n_M$) representation of the SPSD matrix $\bar{\mathcal{M}}_k \in \mathbb{R}^{d_M \times d_M}$ be $\tilde{\mathcal{M}}_k$, having an SVD of the form*[5]

$$\tilde{\mathcal{M}}_k \overset{\text{SVD}}{=} \begin{bmatrix} U & U_{\perp,1} \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & D_{\perp,1} \end{bmatrix} \begin{bmatrix} U^T \\ U_{\perp,1}^T \end{bmatrix} = UDU^T + U_{\perp,1}D_{\perp,1}U_{\perp,1}^T, \tag{A4.25}$$

[4]To be consistent with being able to apply the B-update.

[5]This is consistent with carrying a rank-r representation of $\bar{\mathcal{M}}_k$.

where $U \in \mathbb{R}^{d_M \times n_{crc}}$ and $U_{\perp,1} \in \mathbb{R}^{d_M \times (r-n_{crc})}$ are orthonormal matrices which satisfy $U^T U_{\perp,1} = 0$, and D, $D_{\perp,1}$ are diagonal matrices of positive entries not (necessarily) sorted in any order.

Let $\tilde{\mathcal{M}}_k^{(c)}$ be the rank-r representation obtained by applying the correction in Algorithm 4.3 to $\tilde{\mathcal{M}}_k$ with a number of columns to correct being n_{crc}, and U being[6] the singular-vectors of the subspace we choose to correct (i.e. $U = \tilde{U}_{\cdot,k}^{(l)}[:, col_idx]$).

We then have

$$\left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}\right\|_F^2 = \left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k\right\|_F^2 - \left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2. \tag{A4.26}$$

Proof. In our proof, we shall use $U_\perp$, which is defined from U and $U_{\perp,1}$ as follows. We shall let $U_\perp \in \mathbb{R}^{d_M \times (d_M - n_{\text{crc}})}$ be an orthonormal[7] matrix such that $U_{\perp,1}$ is just a subset of $r - n_{\text{crc}}$ columns of $U_\perp$, and $U_\perp^T U = 0$.

We first give a proof outline. This is as follows:

1. Get a useful Frobenius-norm error decomposition that holds for any symmetric matrix $E \in \mathbb{R}^{d_M \times d_M}$ and any orthogonal matrix $[Q, Q_\perp] \in \mathbb{R}^{d_M \times d_M}$ with $Q \in \mathbb{R}^{d_M \times r}$ and $Q_\perp^T Q = 0$:

$$\|E\|_F^2 = \left\|Q^T E Q\right\|_F^2 + 2\left\|Q_\perp^T E Q\right\|_F^2 + \left\|Q_\perp^T E Q_\perp\right\|_F^2. \tag{A4.27}$$

2. Get SVD form of $\tilde{\mathcal{M}}_k^{(c)}$.
3. Prove that $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U\right\|_F^2 = 0$, and the equalities $\left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp\right\|_F^2 = \left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp\right\|_F^2$, $\left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U\right\|_F^2 = \left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2$, and also $\left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp\right\|_F^2 = \left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp\right\|_F^2$, using SVD of $\tilde{\mathcal{M}}_k^{(c)}$, $\tilde{\mathcal{M}}_k$.
4. Apply (A4.27) twice: (1) $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)$ and $Q \leftarrow U$ and (2) $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})$ and $Q \leftarrow U$ and compare the two errors to conclude.

We now begin the proof.

[6]Note that so far in our choices we only fixed the fact that $U \in \mathbb{R}^{d_M \times n_{\text{crc}}}$ is orthonormal and perpendicular to $U_{\perp,1} \in \mathbb{R}^{d_M \times (r-n_{\text{crc}})}$, but otherwise any such U could be chosen. We now say the chosen U is the same as the singular-vectors of the subspace we apply the correction in.

[7]This choice of $U_\perp$ is possible by simply choosing the first $r - n_{\text{crc}}$ columns to be $U_{\perp,1}$, and then filling in the rest of the columns to give an orthonormal matrix covering the remaining subspace of dimension $d_M - r$.

1. Get a useful Frobenius-norm error decomposition. For any matrix $E \in \mathbb{R}^{d_M \times d_M}$, and any orthogonal matrix $[Q, Q_\perp] \in \mathbb{R}^{d_M \times d_M}$ with orthonormal $Q \in \mathbb{R}^{d_M \times r}$ (and $Q^T Q_\perp = 0$), we have the following $\|\cdot\|_F$ decomposition:

$$\begin{aligned} \|E\|_F^2 &= \left\| \begin{bmatrix} Q^T \\ Q_\perp^T \end{bmatrix} E \begin{bmatrix} Q & Q_\perp \end{bmatrix} \right\|_F^2 = \left\| \begin{bmatrix} Q^T E Q & Q^T E Q_\perp \\ Q_\perp^T E Q & Q_\perp E Q_\perp \end{bmatrix} \right\|_F^2 \\ &= \left\| Q^T E Q \right\|_F^2 + \left\| Q_\perp^T E Q \right\|_F^2 + \left\| Q^T E Q_\perp \right\|_F^2 + \left\| Q_\perp^T E Q_\perp \right\|_F^2 \\ &= \left\| Q^T E Q \right\|_F^2 + 2 \left\| Q_\perp^T E Q \right\|_F^2 + \left\| Q_\perp^T E Q_\perp \right\|_F^2 \end{aligned} \tag{A4.28}$$

where in the first equality we used the invariance of Frobenius norm to unitary (orthogonal) matrices, in the third equality we used standard properties of $\|\cdot\|_F^2$, and in the last equality we used the fact that E is symmetric, and that $\|A\|_F = \left\|A^T\right\|_F$ for any matrix A.

2. Get SVD form of $\tilde{\mathcal{M}}_k^{(c)}$. By applying *Algorithm 4.3* to correct in the subspace spanned by the columns of U (as the proposition states), we have the SVD of $\tilde{\mathcal{M}}_k^{(c)}$ as

$$\tilde{\mathcal{M}}_k^{(c)} \overset{\text{SVD}}{=} \begin{bmatrix} UV & U_{\perp,1} \end{bmatrix} \begin{bmatrix} D_V & 0 \\ 0 & D_{\perp,1} \end{bmatrix} \begin{bmatrix} V^T U^T \\ U_{\perp,1}^T \end{bmatrix} = (UV) D_V (UV)^T + U_{\perp,1} D_{\perp,1} U_{\perp,1}^T, \tag{A4.29}$$

where

$$V D_V V^T \overset{\text{EVD}}{=} U^T \bar{\mathcal{M}}_k U, \tag{A4.30}$$

as *line 5* of *Algorithm 4.3* states. Note that we obtain the SVD of $\tilde{\mathcal{M}}_k^{(c)}$ simply by replacing $U \leftarrow UV$ and $D \leftarrow D_V$, as one can see happens in *Algorithm 4.3*. The form (A4.29) is indeed the SVD of $\tilde{\mathcal{M}}_k^{(c)}$ (and not just some similarly looking form which is not an SVD). To see this, note that we have $UV \perp U_{\perp,1}$ (and thus $\begin{bmatrix} UV & U_{\perp,1} \end{bmatrix}$) is orthogonal) and the matrix $\begin{bmatrix} D_V & 0 \\ 0 & D_{\perp,1} \end{bmatrix}$ is diagonal with positive entries.

3. Prove useful $\|\cdot\|_2^F$ equalities. Using (A4.29), we get

$$\left\| U^T (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}) U \right\|_F^2 = \left\| U^T \bar{\mathcal{M}}_k U - U^T (UV) D_V (V^T U^T) U - U^T U_{\perp,1} D U_{\perp,1}^T U \right\|_F^2, \tag{A4.31}$$

$$\left\| U^T (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}) U \right\|_F^2 = \left\| U^T \bar{\mathcal{M}}_k U - V D_V V^T \right\|_F^2, \tag{A4.32}$$

$$\left\| U^T (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}) U \right\|_F^2 = \left\| U^T \bar{\mathcal{M}}_k U - U^T \bar{\mathcal{M}}_k U \right\|_F^2 = 0. \tag{A4.33}$$

Note that in getting from (A4.31) to (A4.32) we used the fact that $U^T U = I \in \mathbb{R}^{r \times r}$ and $U_{\perp,1}^T U = 0$, and to get from (A4.32) to (A4.33) we used (A4.30).

Using (A4.29) again, we also have

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U = U_\perp^T\bar{\mathcal{M}}_k U - U_\perp^T U V D_V V^T U^T U - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U. \quad \text{(A4.34)}$$

and by noting that $U_\perp^T U = 0$ and $U_{\perp,1}^T U = 0$ we get

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U = U_\perp^T\bar{\mathcal{M}}_k U. \quad \text{(A4.35)}$$

Using the same reasoning but with the SVD of $\tilde{\mathcal{M}}_k$ (see (A4.25)), we similarly get

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U = U_\perp^T\bar{\mathcal{M}}_k U - U_\perp^T U D U^T U - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U = U_\perp^T\bar{\mathcal{M}}_k U. \quad \text{(A4.36)}$$

Combining (A4.35) and (A4.36) gives

$$U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp = U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp = U^T\bar{\mathcal{M}}_k U_\perp. \quad \text{(A4.37)}$$

Finally, using the SVDs of $\tilde{\mathcal{M}}_k^{(c)}$ and $\tilde{\mathcal{M}}_k$ once more, and using the fact that $U^T U_\perp = 0$ we have

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp = U_\perp^T\bar{\mathcal{M}}_k U_\perp - U_\perp^T U V D_V V^T U^T U_\perp - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U_\perp, \quad \text{(A4.38)}$$

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp = U_\perp^T\bar{\mathcal{M}}_k U_\perp - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U_\perp, \quad \text{(A4.39)}$$

and

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp = U_\perp^T\bar{\mathcal{M}}_k U_\perp - U_\perp^T U D U^T U_\perp - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U_\perp \quad \text{(A4.40)}$$

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp = U_\perp^T\bar{\mathcal{M}}_k U_\perp - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U_\perp. \quad \text{(A4.41)}$$

Combining (A4.39) and (A4.41) we get

$$U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp = U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp = U_\perp^T\bar{\mathcal{M}}_k U_\perp - U_\perp^T U_{\perp,1} D_{\perp,1} U_{\perp,1}^T U_\perp. \quad \text{(A4.42)}$$

4. Apply (A4.28) twice. We first note that $\bar{\mathcal{M}}_k$ and $\tilde{\mathcal{M}}_k$ are symmetric by assumption[8] which together with the construction mechanism of $\tilde{\mathcal{M}}_k^{(c)}$ imply $\tilde{\mathcal{M}}_k^{(c)}$ is also symmetric. We thus can apply (A4.28) with $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)$ or with $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})$ as the E's will be symmetric (as required in the proof of (A4.28)).

[8]By assumption in the proposition statement. By construction in our (EA) K-FAC and B-KFAC algorithms (so the assumption holds in practice).

Applying (A4.28) with $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)$, $Q \leftarrow U$ we get

$$\left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k\right\|_F^2 = \left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2 + 2\left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2 + \left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp\right\|_F^2. \tag{A4.43}$$

Now, we also apply (A4.28) with $E \leftarrow (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})$ and $Q \leftarrow U$ to get

$$\left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}\right\|_F^2 = \left\|U^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U\right\|_F^2 + 2\left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U\right\|_F^2 + \left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)})U_\perp\right\|_F^2. \tag{A4.44}$$

Plugging (A4.33), (A4.37) and (A4.42) into (A4.44), we get

$$\left\|\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(c)}\right\|_F^2 = 2\left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U\right\|_F^2 + \left\|U_\perp^T(\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k)U_\perp\right\|_F^2. \tag{A4.45}$$

Substituting the r.h.s. of (A4.45) into the r.h.s. of (A4.43) gives us the desired result (A4.26). □

A4.3 Proof of Proposition 4.5

We reiterate the statement of *Proposition 4.5* for convenience (as *Proposition A4.3*).

Proposition A4.3 (**Error of Doing nothing vs Error of B-updates**). *Let $\tilde{\mathcal{M}}_k$ be an approximation of the EA K-factor $\bar{\mathcal{M}}_k$ which is obtained by performing an* R-KFAC *update at $k = k_{RS}$ (for some $k_{RS} \in \mathbb{Z}^+$), and either no other update thereafter, or B-updates (every step) thereafter. The error in representing $\bar{\mathcal{M}}_k$ when using either one of these approximation schemes is of the form*

$$\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k = \rho^{k-k_{RS}} E_{k_{RS}} + (1-\rho)\sum_{j=k_{RS}+1}^{k} \rho^{k-i} E_j, \ \ \forall k \geq k_{RS}+1. \tag{A4.46}$$

When performing RSVD *initially (at*[9] *$k = k_{RS}$), and no update thereafter we have*

$$\begin{aligned} E_{k_{RS}} &= \bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}, \ \ \textit{and} \\ E_j &= M_j M_j^T - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)} \ \ \forall j \in \{k_{RS}+1, ..., k\}. \end{aligned} \tag{A4.47}$$

When performing RSVD *initially (at $k = k_{RS}$), and B-updates thereafter we have*

$$\begin{aligned} E_{k_{RS}} &= \bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}, \ \ \textit{and} \\ E_j &= \frac{1}{1-\rho}\big(\tilde{\mathcal{M}}_j^{(B)} - \mathcal{B}_j\big) \ \ \forall j \in \{k_{RS}+1, ..., k\}. \end{aligned} \tag{A4.48}$$

[9]For arbitrary $k_{RS} \in \mathbb{Z}^+$.

where the B-process quantities $\tilde{\mathcal{M}}_j^{(B)}$ and $\mathcal{B}_j$ are for a B-process as in (A4.3) but with "initialization" of $\mathcal{B}_{k_{RS}} = \tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}$ at $k_{RS} \geq 1$, rather than of $\tilde{\mathcal{M}}_0^{(B)} = 0$ at $k = 0$.

Proof. ***Part 1 (no updates thereafter):*** First, using (A4.1) $k - k_{RS}$ times recursively (backwards from $\bar{\mathcal{M}}_k$), we get

$$\bar{\mathcal{M}}_k = \rho^{k-k_{RS}} \bar{\mathcal{M}}_{k_{RS}} + (1-\rho) \sum_{i=k_{RS}+1}^{k} \rho^{k-i} M_i M_i^T \quad \forall k \geq k_{RS}+1,\ k_{RS} \geq 1. \quad \text{(A4.49)}$$

In the proof we shall always consider $k \geq k_{RS} + 1 \geq 2$.

$\tilde{\mathcal{M}}_k$ is obtained with a R-KFAC update at $k = k_{RS}$ and no update thereafter. In this case, we have $\tilde{\mathcal{M}}_k = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\ \forall k \geq 0$. We have

$$\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} = \rho^{k-k_{RS}} \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} + (1-\rho) \sum_{i=k_{RS}+1}^{k} \rho^{k-i} \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} \quad \forall k \geq k_{RS}+1,\ k_{RS} \geq 1, \quad \text{(A4.50)}$$

because all the coefficients on the r.h.s. of $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ in (A4.50) sum to 1. Subtracting (A4.50) from (A4.49) and using the fact that $\tilde{\mathcal{M}}_k = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ in the case of interest, we get

$$\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k = \rho^{k-k_{RS}} \big(\bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\big) + (1-\rho) \sum_{i=k_{RS}+1}^{k} \rho^{k-i} (M_i M_i^T - \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}), \quad \text{(A4.51)}$$

which proves the claim about errors $\{E_j\}_{j=k_{RS}}^{k}$ in (A4.47), when performing RSVD initially (at $k = k_{RS}$), and no update thereafter.

Part 2 (B-updates thereafter): *For an $\tilde{\mathcal{M}}_k$ with an* RSVD *update at $k = k_{RS}$, and B-updates at each iteration thereafter*, our B-process evolves as described by (A4.3) from k_{RS} to k, but with "initialization" of $\mathcal{B}_{k_{RS}} = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ at $k_{RS} \geq 1$, rather than of $\tilde{\mathcal{M}}_0^{(B)} = 0$ at $k = 0$. We shall bear this in mind, as the notation will not explicitly show this (for ease of notation). All the evolution equations hold normally from k_{RS} to k.

We have

$$\bar{\mathcal{M}}_k - \mathcal{B}_k = (\bar{\mathcal{M}}_k - \tilde{\mathcal{M}}_k^{(B)}) + (\tilde{\mathcal{M}}_k^{(B)} - \mathcal{B}_k). \quad \text{(A4.52)}$$

by using (A4.1) with $j \leftarrow k$ and (A4.3) with $i \leftarrow k-1$ into (A4.52), we get

$$\bar{\mathcal{M}}_k - \mathcal{B}_k = \rho(\bar{\mathcal{M}}_{k-1} - \mathcal{B}_{k-1}) + (\tilde{\mathcal{M}}_k^{(B)} - \mathcal{B}_k). \quad \text{(A4.53)}$$

Applying (A4.53) recursively onto itself a total of $k - k_{RS}$ times, we get

$$\bar{\mathcal{M}}_k - \mathcal{B}_k = \rho^{k_{RS}}(\bar{\mathcal{M}}_{k_{RS}} - \mathcal{B}_{k_{RS}}) + (1-\rho)\sum_{i=k_{RS}+1}^{k} \rho^{k-i}\frac{1}{1-\rho}\big(\tilde{\mathcal{M}}_i^{(B)} - \mathcal{B}_i\big). \quad \text{(A4.54)}$$

Now, using the fact that we performed an RSVD with target rank r at iteration k_{RS} we have $\mathcal{B}_{k_{RS}} = \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$, and thus (A4.54) turns into

$$\bar{\mathcal{M}}_k - \mathcal{B}_k = \rho^{k-k_{RS}}\big(\bar{\mathcal{M}}_{k_{RS}} - \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\big) + (1-\rho)\sum_{i=k_{RS}+1}^{k} \rho^{k-i}\frac{1}{1-\rho}\big(\tilde{\mathcal{M}}_i^{(B)} - \mathcal{B}_i\big). \quad \text{(A4.55)}$$

Equation (A4.55) proves the claim about errors $\{E_j\}_{j=k_{RS}}^{k}$ in (A4.48), when performing RSVD initially (at $k = k_{RS}$), and B-updates thereafter.

These two parts together finalize the proof. □

A4.4 Proof of Proposition 4.6

We reiterate the statement of *Proposition 4.6* for convenience, as *Proposition A4.4*.

Proposition A4.4 ($\|E_j\|_F \ \forall j \in \{k_{RS}+1, ..., k\}$**: Comparison for No-update vs for B-update**). *For the E_j errors as defined in Proposition 4.5, we have the following statements holding:*

1. *When performing* RSVD *initially (at $k = k_{RS}$), and no updates thereafter $\|E_j\|_F$ can get as high as*

$$\|E_j\|_F = \sqrt{\left\|M_j M_j^T\right\|_F^2 + \left\|\tilde{\mathcal{M}}_{k_{RS}}^{(RSVD,r)}\right\|_F^2}, \ \ \forall j \in \{k_{RS}+1, ..., k\}. \quad \text{(A4.56)}$$

2. *When performing* RSVD *initially, and B-updates thereafter $\|E_j\|_F$ is bounded as:*

$$\|E_j\|_F \leq \left\|M_j M_j^T\right\|_F, \ \ \forall j \in \{k_{RS}+1, ..., k\}, \ \ \textit{and} \ \ E_k = 0. \quad \text{(A4.57)}$$

Proof. ***Part 1 (no updates thereafter):*** *For* R-KFAC *update at $k = k_{RS}$ and no update thereafter.* Recall that our RSVD representation $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ has the form $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} = \tilde{U}_{k_{RS},r}\tilde{D}_{k_{RS},r}\tilde{U}_{k_{RS},r}^T$ (by *Algorithm 3.1*). Since $\tilde{U}_{k_{RS},r} \in \mathbb{R}^{d_M \times r}$ in an orthogonal matrix and $\tilde{D}_{k_{RS},r}$ is diagonal with non-negative entries[10] (by *Algorithm*

[10] And $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}$ is provably symmetric by construction.

3.1), we also have that this form is also its SVD: $\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} \overset{\text{SVD}}{=} \tilde{U}_{k_{RS},r}\tilde{D}_{k_{RS},r}\tilde{U}_{k_{RS},r}^T$. Recall that $M_j \in \mathbb{R}^{d_M \times n_M}$. We thus have

$$E_j = M_j M_j^T - \tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)} = M_j M_j^T - \tilde{U}_{k_{RS},r}\tilde{D}_{k_{RS},r}\tilde{U}_{k_{RS},r}^T. \tag{A4.58}$$

Let

$$V_{M_j} D_{M_j} V_{M_j}^T \overset{\text{SVD}}{=} M_j M_j^T \tag{A4.59}$$

Since $r + n_M < d_M$ (s.t. we can apply the B-update, recall *Section 4.1.1*), we can have a case when $V_{M_j}^T \tilde{U}_{k_{RS},r} = 0$. In this case, we have the SVD of E_j as

$$E_j \overset{\text{SVD}}{=} \begin{bmatrix} V_{M_j} & \tilde{U}_{k_{RS},r} \end{bmatrix} \begin{bmatrix} D_{M_j} & 0 \\ 0 & -\tilde{D}_{k_{RS},r} \end{bmatrix} \begin{bmatrix} V_{M_j}^T \\ \tilde{U}_{k_{RS},r}^T \end{bmatrix}. \tag{A4.60}$$

Thus, in this case

$$\|E_j\|_F = \sqrt{\left\|D_{M_j}\right\|_F^2 + \left\|\tilde{D}_0\right\|_F^2}, \tag{A4.61}$$

by the property of the Frobenius norm which tells us the norm is the square-root of sum of squared singular values.

Now, by using the fact that $\left\|UDU^T\right\|_F^2 = \|D\|_F^2$ for arbitrary orthonormal U of appropriate dimensions, and the fact that $V_{M_j}^T V_{M_j} = I_{n_M \times n_M}$, and $\tilde{U}_{k_{RS},r}^T \tilde{U}_{k_{RS},r} = I_{r\times r}$ we get

$$\|E_j\|_F = \sqrt{\left\|M_j M_j^T\right\|_F^2 + \left\|\tilde{\mathcal{M}}_{k_{RS}}^{(\text{RSVD},r)}\right\|_F^2}, \tag{A4.62}$$

which completes the proof[11] for statement 1 (equation (A4.56)) of *Proposition A4.4.*

Part 2 (B updates thereafter): *For* R-KFAC *with an R-update at $k = k_{RS}$, and B-updates at each k thereafter,* we have from equation (4.15) of *Proposition 4.5*

$$E_j = \frac{1}{1-\rho}\left(\tilde{\mathcal{M}}_j^{(B)} - \mathcal{B}_j\right) \ \forall j \in \{k_{RS}+1, ..., k\}. \tag{A4.63}$$

By using $\tilde{\mathcal{M}}_j^{(B)} = \rho\mathcal{B}_{j-1} + (1-\rho)M_j M_j^T$ line of (A4.3) into (A4.63) we get

$$E_j = \frac{1}{1-\rho}\left(\rho\mathcal{B}_{j-1} + (1-\rho)M_j M_j^T - \mathcal{B}_j\right) \ \forall j \in \{k_{RS}+1, ..., k\}. \tag{A4.64}$$

By using (A4.64), the fact that the B-process (A4.3) chooses $\mathcal{B}_j$ as the first r singular-modes of $\tilde{\mathcal{M}}_j^{(B)}$, and the fact that maintaining the first r modes of the SVD is the

[11] Another proof is to use the relationship between trace and Frobenius norm directly in (A4.60), and then use further trace properties. This proof is slightly longer, but gives some further insights.

optimal rank-r representation (in any p,k-norm[12], and in particular in the Frobenius norm,) we get that[13]

$$\|E_j\|_F = \min_{B\in\mathbb{R}^{d_M\times r}} \left\| \frac{1}{1-\rho}\left(\rho\mathcal{B}_{j-1} + (1-\rho)M_jM_j^T - BB^T\right)\right\|_F \quad \forall j \in \{k_{RS}+1,...,k\}. \tag{A4.65}$$

Now, we know from our B-process (A4.3) that $\mathcal{B}_{j-1}$ is of rank r. Thus, for the (generally sub-optimal) choice $BB^T \leftarrow \rho\mathcal{B}_{j-1}$ the term $\left\| \frac{1}{1-\rho}\left(\rho\mathcal{B}_{j-1} + (1-\rho)M_jM_j^T - BB^T\right)\right\|_F$ becomes equal to $\left\|M_jM_j^T\right\|_F$. Thus, we have the following bound[14]

$$\begin{aligned}\|E_j\|_F &= \min_{B\in\mathbb{R}^{d_M\times r}} \left\| \frac{1}{1-\rho}\left(\rho\mathcal{B}_{j-1} + (1-\rho)M_jM_j^T - BB^T\right)\right\|_F \\ &\leq \left\|M_jM_j^T\right\|_F \qquad \forall j \in \{k_{RS}+1,...,k\}.\end{aligned} \tag{A4.66}$$

Equation (A4.66) proves (A4.57), and together with part 1 of this proof finalizes the proof of *Proposition A4.4.*

□

[12]Recall *Section 2.1.3*

[13]And $\mathcal{B}_j$ is arg min of the same expression in (A4.65).

[14]Since we found a choice of $BB^T = \rho\mathcal{B}_{j-1}$ which gave an error of $\left\|M_jM_j^T\right\|_F$, the best attainable error (which is also the error attained by our B-process truncation) is at least that.

Appendix A5

Literature Review: Practical NG Algorithms for DNNs

In this chapter we look at practical NG implementations for DNNs. We consider: (1) non-problem specific NG implementations (in *Section A5.1*) - these are based on the empirical Fisher and connect to stochastic second order methods; (2) NG implementations in the case of supervised learning for FC and Conv layers (in *Sections A5.2* and *A5.3* respectively). In particular, we will focus on K-FAC (as this is our thesis' central interest) for FC and Conv layers, and its main competitors (like SENG). While K-FAC has implementations across a much wider spectrum of layer types and ML paradigms than considered here (recall *"Natural Gradient Methods"* paragraph of *Section 1.3.1* for details), we limit ourselves to the stated settings for conciseness.

A5.1 Base Case: Non-Problem-Specific

In this section we look at practical second order stochastic optimization methods which are generally applicable, and in some particular cases may be thought of as practical implementations of NG or NG-like algorithms. The essential point about these algorithms is that they treat DNNs as black boxes, and only use gradient information to form preconditioners. We are interested in these because the preconditioners they form may be thought of as (possibly very crude) approximations of the Fisher. Importantly, however, this view would not be true in all ML tasks.

Since they do not look "under the hood" of DNNs to exploit problem structure, they have the following strong limitation: they are either very crude approximations or rapidly become computationally infeasible for large nets (although working for small nets). Nevertheless, this provides a base-case for NG implementations, as it represents the most simple and widely applicable approach. We look at:

1. TONGA ([19]) and its sister ([44]) which precondition by precondition by $[F^{(E)}]^{-1}$ based on a block-diagonal approximation of $F^{(E)}$ (only holds for MLE SL).
2. ADAM ([11]), viewed as a NG-like algorithm where we precondition by $[F^{(E)}]^{-1/2}$ based on a very crude diagonal approximation of $F^{(E)}$ (only holds for MLE SL).
3. *Hessian-free* ([15]) which approximately solves $F_k s_k^{(NG)} = -g_k$ for the NG direction with a fixed number of Conjugate Gradient iterations.

A5.1.1 Practical Empirical NG Method: TONGA

The authors motivate TONGA (topmoumoute online NG algorithm) by suggesting that overfitting may happen at a small rate throughout the whole training, and not only in the later phases when it becomes apparent [19]. The intuition behind this is not mentioned in the paper, but is simple: we only have noisy gradient estimates, and this means that taking a step which is downhill on our objective-estimate may be uphill in the true, inaccessible, objective. Intuitively, each time we fall in this situation, there is a chance of having stepped irreversibly into an "overfit" basin, where our objective estimate appears to be low, but the true, inaccesible objective is high[1].

The authors of TONGA argue that this problem can be somewhat fixed by replacing the typically used SGD direction with the *empirical NG* direction. This would work because, as we have seen in *Section A2.2.5*, the *empirical NG* direction minimizes the probability of increasing the generalization error, given our noisy gradient estimate. Relating this to our intuition in the paragraph above, by taking the empirical NG step instead of the SGD step, we know that we have at least minimized a greedy metric[2] of how likely we are to fall in an overfit basin at the next step.

The idea behind TONGA is simple: maintain a low-rank approximation of the *emirical Fisher*, then cast both problems of finding the low-rank approximation and of computing the natural gradient in a lower-dimensional space, thus attaining much better complexity than a *naive implementation* [19].

Before we begin, it is important to bear in mind that TONGA represents the empirical Fisher as the outer-product of objective gradients. While this outer-product always exists and can always be used, it only represents the Fisher in MLE SL cases.

[1]That is, our step does not even give us a greedy dervative-based progress on the true objective, as it appears on the estimate-objective. Rather, our step gives us a greedy increase on the true objective. As a minimum, we would like a greedy-decrease on the true objective to control overfit at least in some (greedy and limited) metric.

[2]Not the best, as a local increase in the true objective does not necessarily mean overfit in the long run - but at least it minimizes the chance of a very disastrous training event happening where the estimate-objective keeps on decreasing while the true objective keeps on increasing.

The TONGA Algorithm. By exploiting the experimentally observed fact that the *empirical Fisher* varies smoothly throughout training, the author claims we can construct "sensible estimates"[3] as [19]

$$F_K^{(E)} \approx \tilde{F}_K^{(E)} := \sum_{k=0}^{K} \gamma^{K-k} g_{\mathrm{B}}(\theta_k) g_{\mathrm{B}}(\theta_k)^T, \tag{A4.1}$$

where $\gamma \in (0,1)$ is a constant. However, we will only retain low-rank (in particular rank r) approximations of $\tilde{F}_k^{(E)}$, which we define as $\tilde{\mathscr{C}}_k$. Let us denote $\tilde{g}_k := g_{\mathrm{B}}(\theta_k)$ for convenience (note $\tilde{g}_k \in \mathbb{R}^{d_\theta}$). To maintain a low-rank approximation in an online fashion, we must be able to cheaply compute the low-rank approximation ($\tilde{\mathscr{C}}_k$) of

$$\hat{\mathscr{C}}_k := \gamma \tilde{\mathscr{C}}_{k-1} + \tilde{g}_k \tilde{g}_k^T \tag{A4.2}$$

given $\tilde{\mathscr{C}}_{k-1}$. Note that we take $\tilde{\mathscr{C}}_{-1} := 0$. If we write $\tilde{\mathscr{C}}_{k-1} := G_{k-1} G_{k-1}^T$, then we have $\hat{\mathscr{C}}_k = \hat{G}_k \hat{G}_k^T$, where $\hat{G}_k := [\sqrt{\gamma} G_{k-1}, \tilde{g}_k]$. We now want a rank r approximation of $\hat{G}_k$, which will be precisely G_k. The authors suggest computing the low-rank approximation of $\hat{\mathscr{C}}_k$ by computing the eigenvalue decomposition of the Gram matrix

$$\hat{G}_k^T \hat{G}_k = V D V^T, \tag{A4.3}$$

and recognizing the relation to the SVD of $\hat{G}_k$, namely $\hat{G}_k = U D^{1/2} V^T$ [19]. Thus, if we define $U^{(r)} := U[:, 0:r]$, $D^{(r)} := D[0:r, 0:r]$, and $V^{(r)} = V[:, 0:r]$, We can then compute $U^{(r)} = \hat{G}_k V^{(r)} [D^{(r)}]^{-1/2}$, and finally get

$$\tilde{\mathscr{C}}_k = G_k G_k^T = U^{(r)} D^{(r)} [U^{(r)}]^T, \tag{A4.4}$$

where $\tilde{\mathscr{C}}_k = G_k G_k^T$ is a rank r approximation for $\hat{\mathscr{C}}_k = \hat{G}_k \hat{G}_k^T$ [19]. In particular, we set $G_k := U^{(r)} [D^{(r)}]^{1/2}$ for future use. Note that if we do this procedure, we will always have G_k and thus the procedure is well defined.

We compute an *approximate (regularized) empirical NG* direction as [19]

$$s_k^{(eNG)} = (\hat{G}_k \hat{G}_k^T + \lambda I)^{-1} \tilde{g}_k = (\hat{G}_k \hat{G}_k^T + \lambda I)^{-1} \hat{G}_k y_k, \quad \text{with} \quad y_k = [0, 0, ..., 1]^T. \tag{A4.5}$$

By using Woodbury's identity with positive definite matrices we get

$$s_k^{(eNG)} = \hat{G}_k (\hat{G}_k^T \hat{G}_k + \lambda I)^{-1} y_k, \tag{A4.6}$$

whose computation cost is $\mathcal{O}(d_\theta r^2 + r^3)$. Computing the low-rank estimate of $\hat{\mathscr{C}}_k$ is also $\mathcal{O}(d_\theta r^2 + r^3)$. We can do better if we let $\hat{\mathscr{C}}$ grow for b iterations as $\gamma^b \tilde{\mathscr{C}}_k +$

[3] We critically review this way of constructing estimates at the end of this subsection

$\sum_{i=1}^{b} \gamma^{b-i} \tilde{g}_{k+i} \tilde{g}_{k+i}^T$, and only then approximate it again as rank r. By doing this, $\tilde{\mathscr{C}}_k$ and G_k will only be defined for $k = \bar{k}$, $\bar{k} \in \{b, 2b, ...\}$. We can still compute $s_k^{(eNG)}$ for each k, as $\hat{G}_{\bar{k}+t} := [\gamma^{t/2} G_{\bar{k}}, \gamma^{(t-1)/2} g_{\bar{k}+1}, ..., \gamma^{1/2} \tilde{g}_{\bar{k}+t-1}, \tilde{g}_{k+t}]$ exists for all $k = \bar{k} + t$, $t \in \{0, 1, ..., b-1\}$.

If we do this, the cost of updating the low-rank approximation is $\mathcal{O}\big((r+b)^3 + d_\theta(r+b)r\big)$ but now happens only once every b iterations. Furthermore, by noting that $\hat{G}_k^T \hat{G}_k = [\sqrt{\gamma}\hat{G}_{k-1}, \tilde{g}_k]^T [\sqrt{\gamma}\hat{G}_{k-1}, \tilde{g}_k]$, $\forall k \in \mathbb{Z}^+ | \{b, 2b, ...\}$, expanding this into a 2×2 block-matrix, and realizing we already have $\hat{G}_{k-1}^T \hat{G}_{k-1}$ from the previous iteration, we can reduce the cost of computing $s^{(eNG)}$ to $\mathcal{O}\big(d(r+t) + (r+t)^3\big)$ (with $t = k - \lfloor k/b \rfloor$) on most iterations (when $t \neq 0$). On all others, this cost will be $\mathcal{O}\big(d_\theta r^2 + r^3\big)$. These costs stress the requirement to keep r and b low. Indeed, the author performs experiments with $d_\theta \approx 10^4$, $r \leq 5$ and $b \leq 50$ (notably $d_\theta \approx n_l w_l^2$ where w_l is the layer width, so TONGA is quadratic in layer width) [19].

Performing the Updates Layer-wise. A final idea is that instead of using a rank $r + t$ approximation for $F^{(e)}$, we can use a rank $r + t$ approximation for each of its diagonal blocks (and set all other blocks to zero). Each diagonal block would correspond to a layer. This would increase our representation rank from $r+t$ to $n_L(r+t)$ (where n_L is the number of layers), while maintaining a similar cost. Indeed, it was shown that in some particular cases, the Hessian (which *may* be approximated by F) converges to a block-diagonal matrix at the end of the training [1, 133]. Furthermore, the authors show numerical results where the block-diagonal low-rank approximation is better for $r + t \leq 100$ (while the reverse holds true for $r + t > 100$), and thus is more cost-effective for small computational budget [19].

Numerical Performance and Related Algorithms. The authors show numerical experiments for values of $r = 5$, $b = 50$, $\gamma = 0.995$ where block-diagonal TONGA outperforms SGD on *MNIST* and *Rectangles-images* classification, in terms of both time and generalization error. They note that setting $\lambda \to \infty$ reverts to SGD, and thus suggest a good strategy for choosing λ is to search for *"optimal"* λ in powers of ten, without specifying which value they used exactly. They report that plain TONGA (without the block-diagonal idea) underperforms SGD significantly, and attribute this difference to the better representation capabilities of the block-diagonal approach in the low $r + t$ regime.

In a different piece of work ([44]), the authors extend this idea by combining information from $F^{(E)}$ with Hessian information and show improved results (over block-TONGA) for *Alpha, Gamma, Delta, Epsilon, Zeta and Faces data sets.* Still, an

SGD variant sometimes performs better [44]. Finally, note that the naming of their improved TONGA, *natural Newton*, is misleading, as the algorithm does not take a Newton step in the natural space (the distribution space), but merely combines NG and Newton steps in the parameter space.

Critical Review of TONGA's Fisher Approximation. Firstly, note that the sum over k in (A4.1) is essentially maintaining an EA over $\tilde{g}_k\tilde{g}_k^T$ initialized from zero at $k=-1$. This is similar in spirit to how K-FAC holds an EA for the K-factors (as we saw in *Section 1.6*). Secondly, note that the empirical Fisher at each location k that feeds into the EA is merely estimated as the outerproduct $\tilde{g}_k\tilde{g}_k^T$ (rather than the more standard estimate $\frac{1}{n_\text{B}}\sum_{i\in\mathcal{I}_\text{B}} g_i(\theta_k)g_i(\theta_k)^T$). This creates a few problems. To see the problems, consider the expectation of a term in the sum (A4.1). We have the desirable fact

$$\begin{aligned}\mathbb{E}_{x_i,y_i\sim q}[g_\text{B}(\theta_k)g_\text{B}(\theta_k)^T] &= \mathbb{E}_{x_i,y_i\sim q}\left[\left(\frac{1}{n_\text{B}}\sum_{i\in\mathcal{I}_\text{B}} g_i(\theta_k)\right)\left(\frac{1}{n_\text{B}}\sum_{i\in\mathcal{I}_\text{B}} g_i(\theta_k)\right)^T\right] \\ &= \frac{1}{n_\text{B}}\mathbb{E}_{x_i,y_i\sim q}[g_i(\theta_k)g_i(\theta_k)^T] = \frac{1}{n_\text{B}}F^{(E)}(\theta_k)\end{aligned} \tag{A4.7}$$

holding *only under the assumptions* that $g_i(\theta_k)$ are independent. Thus, if this assumption does not hold, our "sensible estimate" $\tilde{F}_K^{(E)}$ in (A4.1) is very far from being any kind of (empirical) Fisher estimator. Furthermore, even when the assumption holds, TONGA Fisher estimate is off by a constant multiplicative-factor of n_B (which is nevertheless fixable form the learning rate choice, thus not a real problem). Finally, using $g_\text{B}(\theta_k)g_\text{B}(\theta_k)^T$ rather than $\frac{1}{n_\text{B}}\sum_{i\in\mathcal{I}_\text{B}} g_i(\theta_k)g_i(\theta_k)^T$ (as is standard) to estimate the empirical Fisher at θ_k gives us a much lower-rank estimate. While this may be beneficial in terms of computational savings (using lower r's), it is unclear whether the precondioner obtained is not too low-rank to be practically useful.

TONGA's scaling in Layer Size and the Bigger Picture. TONGA scales linearly in d_θ as we defined it here (whether performed over-all or layer-wise - which is preferable). Since[4] $d_\theta = \sum_{l=1}^{n_L} w_{l-1}w_l$ TONGA's time cost scales quadratically in layer size (i.e. scales like $\sum_l \mathcal{O}(w_l^2)$). This is better than K-FAC, which scales cubically. However, the relatively poor preconditioner performance of TONGA makes K-FAC a much better option, at least whenever the net width is not too large.

[4]Thinking of FC layers or simplicity of exposition.

A5.1.2 ADAM as Empirical NG-like Algorithm

Recall from *Section 1.2.4* that ADAM takes steps of the form [11]

$$\theta_{k+1} = \theta_k - \alpha \hat{m}_k / (\sqrt{\hat{v}} + \epsilon), \tag{A4.8}$$

where the division is understood element-wise, $\hat{m}_k$ is the momentum gradient and

$$\hat{v}_k = \frac{1}{1-\beta_2^k} \sum_{i=0}^{k} \beta_2^{k-i} (1 - \mathbb{I}_{i \neq 0} \beta_2) \tilde{g}_i \odot \tilde{g}_i. \tag{A4.9}$$

Equation (A4.9) is obtained by rearranging the equations in *Section 1.2.4*, and $\mathbb{I}$ is the indicator function. That is, $\hat{v}_k$. By ignoring ϵ (setting it to zero[5]), and incorporating the $1/(1-\beta_2^k)$ into the stepsize, $\alpha_k := \alpha/(1-\beta_2^k)$, we can write the ADAM step (A4.8) as

$$\theta_{k+1} = \theta_k - \alpha_k [\tilde{F}_k^{(E)}]^{-1/2} \hat{m}_k, \tag{A4.10}$$

where

$$\tilde{F}_k^{(E)} = \sum_{i=0}^{k} \beta_2^{k-i} (1 - \mathbb{I}_{i \neq 0} \beta_2) \text{Diag}(\tilde{g}_i \tilde{g}_i^T), \tag{A4.11}$$

can be seen as a very crude approximation of the exponentially-averaged Empirical Fisher. Note that $\text{Diag}(\cdot)$ is an operator which sets all the non-diagonal elements of a matrix to zero. The ADAM $\tilde{F}_k^{(E)}$ is essentially the same as the one of TONGA, but only saves diagonal information, rather than (layer-wise) block-diagonal information. Thus, it inherits all the problems outlined when critically reviewing the TONGA empirical Fisher approximation in *Section A5.1.1*. Importantly, the ADAM preconditioner can only be seen as a brutal approximation to the empirical Fisher for MLE Supervised Learning (same as TONGA). Finally, note that ADAM uses $-1/2$ as the preconditioner power rather than -1. This means that ADAM cannot be seen as a NG algorithm, but is obviously related when thinking in terms of the empirical fisher approximation.

A5.1.3 Hessian-Free (Martens 2010)

The idea behind the Hessian-free[6] approach is simple and a direct application of ideas already existing in the classical optimization literature. It relies on approximately solving[7] $F_k s_k^{(NG)} = -g_k$ with Conjugate Gradient (CG), where we would perform only

[5] Which is very small (i.e. 10^{-6}) there purely for numerical stability.

[6] While the name Hessian-free is confusing, it really refers to never assembling the curvature matrix and only accessing required information through Matrix-vector products.

[7] Altough the Curvature matrix presented in the original work is the Generalized Gauss-Newton matrix, this was equivalent to the Fisher under the scenario investigated therein [15].

a fixed number of CG iterations (about 100) [15]. Importantly, CG is computationally feasible because it only ever needs Fisher vector (F-v) products (and never form the full matrix F_k). The F-v products can be computed efficiently as suggested in [113].

Main Drawback: Very Slow. While computationally feasible, the method is still very slow, as each step requires hundreds of CG iterations (thus hundreds of F-v products), and each F-v product costs roughly as much as an SGD step [29]. The only advantage of this approach would be that the equation $F_k s_k^{(NG)} = -g_k$ is solved much better than other approaches[8]. However, this would only translate into a better approximation to the NGD step if the Fisher itself is accurately approximated - which requires a very large batch size, thus further slowing down the algorithm.

A5.2 Supervised Learning: Fully Connected Layers

A5.2.1 K-FAC (for FC layers)

We now expand on the minimalistic presentation of K-FAC given in *Section 1.5*, and look at K-FAC for FC layers (see *Section A5.3.1* for convolutional K-FAC).

K-FAC Fisher Approximation: Derivation. We begin by defining "$\otimes$" to be the Kronecker product operator. K-FAC (**K**ronecker **F**actored **A**pproximate **C**urvature) works by approximating F^{-1} as block (tri-)diagonal (we focus on block-diagonal here for simplicity), and taking advantage of the structure of each block to perform an approximation of the form $\mathbb{E}[A \otimes B] \approx \mathbb{E}[A] \otimes \mathbb{E}[B]$, which reduces the matrix inversion cost [1]. Recall that $\bar{a}_l$ is the augmented post-activation at the l^{th} layer and that $\delta_l = \nabla_{z_l} L$ (see *Chapter 1*). By using chain-rule, one can easily show that [1]

$$\nabla_\theta L(x, y; \theta) = [\text{vec}(\delta_1 \bar{a}_0^T),\ \text{vec}(\delta_2 \bar{a}_1^T),\ ...,\ \text{vec}(\delta_l \bar{a}_{n_L - 1}^T)]^T. \tag{A4.12}$$

Using this, we have [1]

$$F = \mathbb{E}_{x,y\sim p}[\nabla_\theta L(x, y; \theta) \nabla_\theta L(x, y; \theta)^T] = \begin{pmatrix} \cdots & \cdots & \cdots \\ \vdots & \mathbb{E}_{x,y\sim p}[\text{vec}(\delta_l \bar{a}_{l-1}^T)\text{vec}(\delta_m \bar{a}_{m-1}^T)^T] & \vdots \\ \cdots & \cdots & \ddots \end{pmatrix}, \tag{A4.13}$$

where blocks correspond to layers, and we showed only the block on the block-row l, block-column m on the r.h.s [1]. We now approximate[9]

[8] Which also solve this equation implicitly, and with many approximations.

[9] The block-diagonal approximation can be thought of as an exact result under the assumption that derivatives of the loss w.r.t. weights from different layers are uncorrelated [29].

$$F \approx \text{block-diagonal}_{l\in\{1,\dots,n_L\}}\Big(\mathbb{E}_{x,y\sim p}[\text{vec}(\delta_l \bar{a}_{l-1}^T)\text{vec}(\delta_l \bar{a}_{l-1}^T)^T]\Big), \tag{A4.14}$$

and further use the Kronecker identity $\text{vec}(\delta_l \bar{a}_{l-1}^T)\text{vec}(\delta_l \bar{a}_{l-1}^T)^T = (\bar{a}_{l-1}\bar{a}_{l-1}^T) \otimes (\delta_l \delta_l^T)$, and the approximation[10] $\mathbb{E}_{x,y\sim p}[(\bar{a}_{l-1}\bar{a}_{l-1}^T) \otimes (\delta_l \delta_l^T)] \approx \mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T] \otimes \mathbb{E}_{x,y\sim p}[\delta_l \delta_l^T]$, to obtain yet another approximation for F, namely [1]

$$\tilde{F} := \text{block-diagonal}_{l\in\{1,\dots,N_L\}}\Big(\mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T] \otimes \mathbb{E}_{x,y\sim p}[\delta_l \delta_l^T]\Big). \tag{A4.15}$$

We now denote $\mathcal{A}_{l,l} := \mathbb{E}_{x,y\sim p}[\bar{a}_l \bar{a}_l^T]$ and $\Gamma_{l,l} := \mathbb{E}_{x,y\sim p}[\delta_l \delta_l^T]$. Clearly, we also have

$$\tilde{F}^{-1} = \text{block-diagonal}_{l\in\{1,\dots,N_L\}}\big([\mathcal{A}_{l-1,l-1} \otimes \Gamma_{l,l}]^{-1}\big). \tag{A4.16}$$

We can thus get $u = \tilde{F}^{-1}v$ as

$$U_l = \Gamma_{l,l}^{-1} V_l \mathcal{A}_{l-1,l-1}^{-1}, \tag{A4.17}$$

where v maps to $\{V_1, V_2, ...V_{n_l}\}$ and u maps to $\{U_1, U_2, ...U_{n_l}\}$ in an analogous way to how our gradient $\nabla_\theta L(x, y; \theta) = [\text{vec}(\delta_1 \bar{a}_0^T), \text{vec}(\delta_2 \bar{a}_1^T), ..., \text{vec}(\delta_l \bar{a}_{l-1}^T)]^T$ maps to $\{\delta_1 \bar{a}_0^T, \delta_2 \bar{a}_1^T, ..., \delta_{n_L} \bar{a}_{n_L-1}^T\}$. Thus, we can compute our *(block-diagonal)* K-FAC *direction* if we have $\{\mathcal{A}_{l,l}^{-1}\}_{l=0}^{n_L-1}$ and $\{\Gamma_{l,l}^{-1}\}_{l=1}^{n_L}$.

Computing the K-Factors Inverses. Note that actual inverses are required (not just solving a linear system), so we compute the inverse of $2n_L$ matrices of the form $\mathcal{A}_{l,l}, \Gamma_{l,l} \in \mathbb{R}^{w_l \times w_l}$, where w_l is the width of layer l. Even though $w_l \ll d$, obtaining these inverses can get expensive for wide nets ((EVD is cubic in layer width w_l).

Estimating K-Factors with Monte Carlo. While Monte-Carlo estimates for $\mathcal{A}_{l,l}$ can be assembled cheaply because a_l does not depend on y (and thus we can use the quantities obtained in the forward pass), estimating $\Gamma_{l,l} = \mathbb{E}_{x,y\sim p}[\delta_l \delta_l^T]$ is more expensive. That is because δ_l *does* depend on y, and thus we must sample $y \sim p(y|x)$, and then compute another backward pass per sample, using the new target[11]. In practice we use a Monte-Carlo estimate with just one sample per x_i.

[10]This approximation can be thought of as an exact result under the assumption that (the random variable) δ_l is independent of (the random variable) $\bar{a}_{l-1}$. This is called the IAD assumption (independent activations and derivatives) [29].

[11]Since $y \sim \hat{q}$ when computing g_B, we cannot use the δ_l's obtained in that backward pass. If we did this, we would (approximately) take *empirical NG steps* rather than *true NG steps*. We want *true NG steps*.

This means we approximate our K-Factors as[12] $\mathcal{A}_{l-1,l-1} \approx \mathcal{A}^{(l)} := A_{(l)}[A_{(l)}]^T$ and $\Gamma_{l,l} \approx \Gamma^{(l)} := G^{(l)}[G^{(l)}]^T$ where $A_l \in \mathbb{R}^{w_{l-1} \times n_\text{B}}$ (where each column of $A^{(l)}$ is the $\bar{a}_{l-1}$ obtained for one of the x_i's in the batch of size n_B; and each column of $G^{(l)} \in \mathbb{R}^{w_l \times n_\text{B}}$ is the δ_l obtained for one of the x_i in the batch). Thus, we practically estimate the Fisher as

$$F \approx \text{block-diagonal}_{l \in \{1,\dots,N_L\}}\big(\mathcal{A}^{(l)} \otimes \Gamma^{(l)}\big), \tag{A4.18}$$

with the $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$ we have just defined. Note that the rank of $\Gamma^{(l)}$ and $\mathcal{A}^{(l)}$ is (at most) n_B when $w_l > n_\text{B}$ and $w_{l-1} > n_\text{B}$. This observation is essential for *Chapter 4* and will be reiterated there.

EA of K-Factors. Exponentially-averaged ("online") estimates of $\{\mathcal{A}_{l,l}\}_{l=0}^{n_L-1}$ and $\{\Gamma_{l,l}\}_{l=1}^{n_L}$ are maintained: where we take the current values to be the new values weighted by $1-\rho_k$, plus the old values weighted by ρ_k. A simple schedule of the form $\rho_k = \min\{1-1/k, 0.95\}$ was initially proposed ([1]) but practically we $\rho = 0.95$ is typically used.

Resorting to Infrequent Updates. Recomputing the local K-Factors (to update the EA K-Factor) requires an additional back-prop per data-point in the batch. Recomputing the inverses of Kronecker factors requires $2n_L$ matrix inversions (possibly through SVD or (EVD; discussed soon). Both these operaions are quite expensive. An obvious strategy for reducing cost is to only update the online estimates once every $b \in [5, 100]$ iterations, at the expense of sacrificing accuracy and thus progress per iteration [1]. The authors set $b = 20$ [1]. This can be taken a step further and recompute the inverses even less frequently than we update the EA K-Factors (typically done in practice).

Preconditioner Regularization (Damping). In practice, we also need to add *update damping* (i.e. regularization) to our quadratic model induced by $\tilde{F}$. We now describe the damping originally proposed by the authors. However, in practice a much simpler regularization is used (adding $+\lambda I$ to the Kronecker factors). The authors report that all simple regularization methods significantly reduced the progress per iteration. Thus, the damping is performed in two stages. In the first stage, they use a Tikhonov[13] regularization with relatively low λ to compute a proposal $s_k^{(\text{p})} := -(\tilde{F} +$

[12]We only ever use $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$ in the other chapters for simplicity (we only care about the attainable estimate rather than the conceptual quantity we are trying to estimate). Note that we define $A^{(l)}$ in terms of quantities at layer $l-1$ (stack of $\bar{a}_{l-1}$ for different x_is). We do this for convenience s.t. we can say the preconditioner for layer-l gradient requires $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$.

[13]To be precise, $\lambda = \bar{\lambda} + \eta$, where η l_2-regularizes the objective and $\bar{\lambda}$ is adapted in a Levenberg-Marquardt style.

$\lambda I)^{-1} g_{\mathrm{B}}$. Note that now, we will need to compute $u_l = (\mathcal{A}_{l-1,l-1} \otimes \Gamma_{l,l} + \lambda I \otimes I)^{-1} v_l$, and cannot apply (A4.17) directly (where $u_l = \mathrm{vec}(U_l)$ and $v_l = \mathrm{vec}(V_l)$). By using some Kronecker identities, we can rearange this as

$$\Gamma_{l,l} U_l \mathcal{A}_{l-1,l-1}^T + U_l = V_l, \tag{A4.19}$$

where we want to solve for U_l. This is a *generalized Stein equation*, for which the authors present an approach where they compute the s.v.d. of $\mathcal{A}_{l-1,l-1}$ and $\Gamma_{l,l}$, and having done that, can solve (A4.19) cheaply for any given V_l. This is useful in our context, because we maintain the same values for $\{\mathcal{A}_{l,l}^{-1}\}_{l=0}^{n_L-1}$ and $\{\Gamma_{l,l}^{-1}\}_{l=1}^{n_L}$ for many consecutive steps, thus performing the s.v.d. rarely. Note that the cost of inverting $2n_L$ matrices is now replaced by performing the s.v.d. of the same matrices.

The second step of the *damping rule* is to scale the proposal $s_k^{(\mathrm{p})}$, to minimize the regularized quadratic model induced by *the exact* F, along $s_k^{(\mathrm{p})}$. Thus, the *KFAC* step is $s_k^{(KFAC)} := \alpha_k s_k^{(\mathrm{p})}$, where

$$\alpha_k := \arg\min_{\alpha} \frac{1}{2} \alpha^2 [s_k^{(\mathrm{p})}]^T (F + \lambda I) s_k^{(\mathrm{p})} + g_{\mathrm{B}}^T s_k^{(\mathrm{p})}. \tag{A4.20}$$

We obtain α_k as

$$\alpha_k = \frac{-g_{\mathrm{B}}^T s_k^{(\mathrm{p})}}{[s_k^{(\mathrm{p})}]^T F s_k^{(\mathrm{p})} + \lambda [s_k^{(\mathrm{p})}]^T s_k^{(\mathrm{p})}}. \tag{A4.21}$$

Note that we can efficiently estimate $v^T F v$ in equation (A4.21) by recalling equation (A2.10), from which we can see that

$$v^T F v \approx \frac{1}{|\mathcal{I}_{\mathrm{B}}|} \sum_{i \in \mathcal{I}_{\mathrm{B}}}^{N_D} (J_{h_i} v)^T F_{r_i} (J_{h_i} v). \tag{A4.22}$$

Thus, estimating $v^T F v$ amounts to computing the Jacobian[14]-vector product $J_{h_i} v$ for each data point in the mini-batch, then performing the multiplications shown in (A4.22), and finally averaging over $\mathcal{I}_{\mathrm{B}}$. The $J_{h_i} v$ products can be obtained with just one forward pass (forward mode AD). Estimating F_{r_i} as well as taking products with it is cheap as this typically represents a diagonal matrix possibly plus some low-rank terms [1]. For example, $F_{r_i} = I$ for the squared loss. Thus, our *two-steps update damping* adds little overhead on top of the no-damping case [1].

Desirable K-FAC Properties. Finally, we note that the K-FAC's path in the distribution space is *invariant w.r.t. a certain class of reparameterization* when damping is absent (or negligible), momentum is not used, and the learning rate is

[14]The Jacobian of the net output w.r.t. parameters. See *Section A5.3.1* for details.

chosen independently of the network's parameterization [1]. Two practical consequences are that under the stated conditions, the solution found by K-FAC is invariant to affine transformations of x, and also to transforming $\phi_l(\cdot)$ in a certain way, which includes the particular case of replacing *tanh* with *sigmoid* [1]. Of course, we must appropriately amend our initialization as well. Another consequence that only holds in the limit of zero stepsize is that K-FAC is also invariant w.r.t whitening [1].

Momentum. Momentum can also be added, with the net effect of incorporating another degree of freedom to our *second step of damping* (problem (A4.20)). That is, the search is now made in $\text{span}\{s_k^{(\text{prop})}, s_{k-1}^{(\text{KFAC})}\}$. The optimal solution of the local quadratic model will now require 2 J-v products rather than just one, as is the case for no momentum. A version where we use a block-tridiagonal approximation for F^{-1} (note that this allows for F to be a dense matrix, as block-tridiagonal F^{-1} does not imply block-tridiagonal F) is also provided by the author [1]. This improves the approximation accuracy (as in practice F^{-1} is quite close to being block-tridiagonal) at the expense of further computational cost. In practice, we would typically use the block-diagonal version (presented here) at least for wide enough nets, where the cost of inverting the Kronecker factors is significant.

In summary, we have seen that K-FAC can approximate the *true NG direction* in large DNN models. To achieve this, KFAC requires *1 forward pass*, *1 backward pass*, and *1 forward mode AD* (or 2 if we use momentum)[15] on *each* iteration. Once in b iterations, it also requires computing the SVD of $2n_L$ matrices in $\mathbb{R}^{w_l \times w_l}$ and *1 extra backprop*. Comparing this with first-order methods, which compute only *1 forward pass* and *1 backprop* on all iterations, the extra cost of KFAC is sometimes (but not always[16]) outweighed by the benefits. Indeed, the authors show results for *CURVES, MNIST,* and *FACES deep autoencoder problems* where K-FAC outperforms SGD with momentum in terms of CPU time [1]. However, other experiments show that K-FAC can significantly underperform SGD when the net width is large [20].

The K-FAC algorithm we presented is only applicable to *FC* nets. Thus, *Grosse and Martens* (2016, [29]) and *Martens et. al.* (2018, [30]) provide ways to extend the idea to *CNNs* (details in *Section A5.3.1*) and *RNNs* respectively, and show results where their proposed K-FAC extensions outperform SGD and ADAM.

Golfarb et al. (2021, [41]) have recently used the K-FAC idea to design better *BFGS* (or *L-BFGS*) algorithms for large-scale DNNs. They approximate the Hessian as a

[15] In practice the forward-mode AD is avoided by simply regularizing the K-Factors as $+\lambda I$. Numerical evidence shows this approach works well [71, 51, 52, 53, 73].

[16] Certainly not when the net width is large.

Kronecker-factored block-diagonal matrix, and then use the *BFGS* method to hold estimates of the K-factors. However, note that this method is *not a NG method*.

A5.2.2 PRONG

Desjardins et. al. (2015, [45]) devise an interesting alternative approach to natural gradient. Instead of trying to compute the *Fisher* cheaply, they aim to reparameterize the network such that the diagonal blocks of the *Fisher* are closer to being the identity. In this case, performing SGD would be (almost) equivalent to *block-diagonal NGD*. Similarly to K-FAC, *PRONG* (*Projected Natural Gradient*) uses the approximation $\mathbb{E}_{x,y\sim p}[(\bar{a}_{l-1}\bar{a}_{l-1}^T) \otimes (\delta_l\delta_l^T)] \approx \mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T] \otimes \mathbb{E}_{x,y\sim p}[\delta_l\delta_l^T]$, thus considering

$$\tilde{F} = \text{block-diagonal}_{l\in\{1,\dots,N_L\}}\left(\mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T] \otimes \mathbb{E}_{x,y\sim p}[\delta_l\delta_l^T]\right), \tag{A4.23}$$

though the authors write equation (A4.23) in scalar form. They justify the "$\mathbb{E}[(\cdot)\otimes(\cdot)]$" approximation through assuming independence of δ_l and $\bar{a}_{l-1}$ [45]. The authors then aim to add some layers to the net such that the term $\mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T]$ (or rather its equivalent reparametrization) is the identity. We could do the same for $\mathbb{E}_{x,y\sim p}[\delta_l\delta_l^T]$, but this works worse in practice due to arising complications [45]. Thus, we settle for only setting one of the K-factors to be the identity (the forward one $\mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T]$).

The main idea is to use Whitened Neural Networks (WNN), which are typical nets with added whitening layers. These layers have the purpose of transforming the covariance matrix of the post-activations to I [45]. The authors propose a novel whitened neural layer under the assumption that the network statistics $\mu_l(\theta) := \mathbb{E}_{x\sim q}[a_l]$ and $\mathscr{C}_l(\theta) := \mathbb{E}_{x\sim q}[a_l a_l^T]$ are fixed. In essence, they add a whitening layer after each typical layer (similarly to [134]), thus getting

$$a_l = \phi_l(W_l U_{l-1}(a_{l-1} - \mu_l) + b_l) \tag{A4.24}$$

rather than

$$a_l = \phi_i(W_l a_{l-1} + b_l) = \phi_l(W_l \bar{a}_{l-1}) \tag{A4.25}$$

which is what we are normally used to. In equation (A4.24), $U_l := (\Lambda_l + \epsilon I)^{-1}\tilde{U}_l$, where $\mathscr{C}_l = \tilde{U}_l\Lambda_l\tilde{U}_l^T$ is the eigendecomposition of $\mathscr{C}_l$. Note that ϵ is a small regularization term and that W_l and b_l are typical network parameters.

Crucially, we will not view $\{U_l\}_{l=0}^{n_L-1}$ and $\{\mu_l\}_{l=0}^{n_L-1}$ as learnable parameters, as this would make the training even less efficient. Thus, our parameter vector θ only includes $\{W_l\}_{l=1}^{n_L}$ and $\{b_l\}_{l=1}^{n_L}$, and we take $\{U_l\}_{l=0}^{n_L-1}$ and $\{\mu_l\}_{l=0}^{n_L-1}$ as being fixed. Indeed,

with the added whitened layers, we now have that one of the Kronecker factors[17] of each block of $\tilde{F}$ (where the learnable parameters are only the ones in θ), namely $\mathbb{E}_{x,y\sim p}[(U_{l-1}a_{l-1})(U_{l-1}a_{l-1})^T]$, would be the identity.

Because $\{U_l\}_{l=0}^{n_L-1}$ and $\{\mu_l\}_{l=0}^{n_L-1}$ change with θ, we have to re-estimate them periodically. We do this once in T iterations using Monte-Carlo estimates for μ_l and $\mathscr{C}_l$, namely $\hat{\mu}_l$ and $\hat{\mathscr{C}}_l$ respectively. We have to ensure that we do not interfere with the convergence properties of SGD. We do this by maintaining W_lU_{l-1} constant across an U_{l-1} update, and an analogous constraint for biases [45]. The update algorithm for $\{U_l\}_{l=0}^{n_L-1}$ and $\{\mu_l\}_{l=0}^{n_L-1}$ is shown below (in *Algorithm A5.1*).

Algorithm A5.1: Update rule for $\{U_l\}_{l=0}^{n_L-1}$ and $\{\mu_l\}_{l=0}^{n_L-1}$ [45]

if *We are at the first iteration* **then**
  Perform a Monte-Carlo estimation of $\hat{\mu}_l$ and $\hat{\mathscr{C}}_l$ by using N_S draws $x \sim q$;
  Compute eigendecomposition $\hat{\mathscr{C}}_l = \tilde{U}_l\Lambda_l\tilde{U}_l^T$ and set $U_l := (\Lambda_l + \epsilon I)^{-1}\tilde{U}_l$;
else if *T iterations have passed* **then**
  Compute canonical parameters $\tilde{W}_l = W_lU_{l-1}$, $\tilde{b}_l = b_l - W_lU_{l-1}\mu_l$;
  `// These must stay fixed across an update`
  Re-estimate $\hat{\mu}_l$ and $\hat{\mathscr{C}}_l$ by using N_S draws $x \sim q$;
  Compute eigendecomposition $\hat{\mathscr{C}}_l = \tilde{U}_l\Lambda_l\tilde{U}_l^T$ and reset U_l to $U_l := (\Lambda_l + \epsilon I)^{-1}\tilde{U}_l$;
  Set $W_l = \tilde{W}_lU_{l-1}^{-1}$ and $b_l = \tilde{b}_l + W_lU_{l-1}\mu_l$;

We can indeed use samples from q in this context as we only concern ourselves with $\{a_l\}_{l=0}^{n_L-1}$ which are label-agnostic, and we chose $p(x) \equiv q(x)$ (recall *Section 1.4.1*). Note that we still need to perform an eigenvalue decomposition of $\hat{\mathscr{C}}_l$ once per T iterations, which is relatively expensive. While the authors did not propose this solution, we note here that we could dramatically save on this eigendecomposition by observing that $\hat{\mathscr{C}}_l$ is of the form $\hat{\mathscr{C}}_l = X_aX_a^T$, where X is *"skinny-tall"* since we typically have a relatively small batch size. Thus, we could get a much cheaper estimate of the first few eigenmodes of $\hat{\mathscr{C}}_l$ by performing a *randomized-SVD* (see [79]) of X_a, with a relatively low target-rank. This would be reasonable if our eigenvalues decay rapidly.

The authors tune the hyperparameters on MNIST and report $\epsilon = 0.1$, $T = 100$ to be the best values. They then show results on CIFAR10 and ImageNet datasets where PRONG outperforms SGD in terms of time. However, comparing PRONG to plain SGD might be unfair (since PRONG does something akin to *batch-normalization*). The performance of PRONG and *batch-normalized SGD* is similar PRONG. To conclude, we mention two interesting ways of viewing PRONG. The first one is that by noting there is a linear bijection between the canonical parameter space and whitened net parameter

[17] Which is equivalent to $\mathbb{E}_{x,y\sim p}[\bar{a}_{l-1}\bar{a}_{l-1}^T]$ for the plain net.

space, we can view PRONG as a *Mirror Descent* algorithm [45, 135]. The second, more practical one, is that PRONG implicitly preconditions the gradient (directly through the backward pass) with a rough approximation of the *inverse Fisher* by adding some *"appropriate"* layers to our given network. Still, obtaining these layers' parameters has a cost. PRONG has also developed for Deep Q-learning [136].

A5.3 Supervised Learning: Convolutional Layers

A5.3.1 Convolutional K-FAC: KFC

In *Section A5.2.1* we have seen K-FAC (a successful practical NGD implementation for DNNs) which was only applicable to FC layers. We now look at an analogous approach which makes K-FAC applicable to Conv layers as well. The authors use KFC ("Kronecker Factors for Convolution" [29]) to refer to convolutional K-FAC. In this section, we might also use KFC for convenience, but outside this section we will always use "Convolutional K-FAC" (to keep names obvious).

Conv Layers. To introduce convolutional K-FAC, we need to consider how a Convolutional layer works and annotate the required quantities. We do so based on the material in [29]. We only focus on one layer for simplicity (so there will be no layer index). We will use the notation $\mathsf{D}X$ for the derivative of the loss w.r.t. X.

The input to the convolutional layer is the pre-actionation tensor $\{a_{j,t}\}$ where $j \in \mathcal{J}$ indexes over different input maps and $t \in \mathcal{T}$ indexes over different spatial locations. For computer vision, there are two spatial dimensions, so t is a tuple (thus $\{a_{j,t}\}$ represents a 3D tensor. The layer is parameterized by a set of weights $\{w_{i,j,t_\delta}\}$ and biases $\{b_i\}$ where $i \in \mathcal{I}$ indexes over different output maps, j indexes over input maps and $t_\delta \in \mathcal{T}_\delta$ is a tuple which indexes over the spatial offset from the filter centre (each $w_{i,:,:}$ is a filter). We have $\mathcal{T}_\delta = \{-R, ..., R\}^2$ for some $R \in \mathbb{N}^*$. For simplicity assume the stride is 1 and padding is R. Naively, the pre-activations[18] $\{z_{i,t}\}$ are computed as

$$z_{i,t} = b_i + \sum_{j \in \mathcal{J}} \sum_{t_\delta \in \mathcal{T}_\delta} w_{i,j,t_\delta} a_{j,t+t_\delta}, \tag{A4.26}$$

where $a_{j,t+t_\delta}$ is zero when $t + t_\delta \notin \mathcal{T}$. The pre-activations are then passed through a non-linear function and the layer output is obtained. For each layer, let us stack all biases b_i in a vector $\underline{b} \in \mathbb{R}^{|\mathcal{I}|}$.

[18] Also a 3D tensor for image vision with t being a tuple. As is obvious form the convolution euqation, i indexes over output maps.

Conv Layers as Matrix-vector Products The naive implementation in (A4.26) is very inefficient, and never used in practice. Instead, the convolution operation has to be framed as matrix-vector multiplications to take advantage of the efficient parallel BLAS operators and memory locality [29]. To achieve this, we have [29]

- **The preactivations** stored as a $n_{\mathrm{B}} \times |\mathcal{T}| \times |\mathcal{J}|$ array, which can be (efficiently) interpreted as a $n_{\mathrm{B}}|\mathcal{J}| \times |\mathcal{T}|$ matrix: $\tilde{A}_{l-1} \in \mathbb{R}^{n_{\mathrm{B}}|\mathcal{J}| \times |\mathcal{T}|}$ for layer $l-1$.
- **The (Filter) Weights** stored as a $|\mathcal{I}| \times |\mathcal{T}_\delta| \times |\mathcal{J}|$ array, which can be (efficiently) interpreted as a $|\mathcal{I}| \times |\mathcal{T}_\delta||\mathcal{J}|$ matrix $W_l \in \mathbb{R}^{|\mathcal{I}| \times |\mathcal{T}_\delta||\mathcal{J}|}$
- **The post-activations** $\tilde{Z}_{l-1} \in \mathbb{R}^{n_{\mathrm{B}}|\mathcal{I}| \times |\mathcal{T}|}$ have the same shape as the pre-activations of the next layer $\tilde{A}_l$.

Note that we put the layers indices back in as we are dealing with objects with much fewer indices. To obtain efficient implementation we need use the expansion operator $\langle \cdot \rangle$ which extracts the patches surrounding each spatial location and flattens them into vectors [29]. These vectors become the rows of a matrix. For example expanding the matrix $\tilde{A}_{l-1} \in \mathbb{R}^{n_{\mathrm{B}}|\mathcal{J}| \times |\mathcal{T}|}$ gives $\langle \tilde{A}_{l-1} \rangle \in \mathbb{R}^{n_{\mathrm{B}}|\mathcal{T}| \times |\mathcal{J}||\Delta|}$, defined as

$$\big[\langle \tilde{A}_{l-1} \rangle\big]_{tn_{\mathrm{B}}+\ltimes,\, j|\mathcal{T}_\delta|+t_\delta} := [\tilde{A}_{l-1}]_{(t+t_\delta)n_{\mathrm{B}}+\ltimes,\, j} = a^{(\ltimes)}_{j,t+t_\delta}, \tag{A4.27}$$

where $\ltimes$ indexes across datapoints within a batch.

The forward pass is computed as [29]

$$\tilde{A}_l = \phi(\tilde{Z}_l) = \phi\big(\langle \tilde{A}_{l-1} \rangle W_l^T + \underline{1}\underline{\mathrm{b}}_l^T\big) = \phi\big(\langle \tilde{A}_{l-1} \rangle_H \bar{W}_l^T\big), \tag{A4.28}$$

with the new definitions[19] $\langle \tilde{A}_{l-1} \rangle_H := [\underline{1}, \langle \tilde{A}_{l-1} \rangle] \in \mathbb{R}^{n_{\mathrm{B}}|\mathcal{T}| \times (1+|\mathcal{J}||\Delta|)}$ and $\bar{W}_l := [\underline{\mathrm{b}}_l, W_l] \in \mathbb{R}^{\times}$. The reason behind writing the forward pass in terms of these newly defined quantities is analogous to the FC case: we want to incorporate the bias in the weight matrix for easy handling with K-FAC [29]. We omit the details of the backward pass as these are not strictly necessary (see [29]). Suffice to say we can get $\mathsf{D}\tilde{Z}_l$ efficiently [29].

Finally, we define A_l and Z_l through their relationship with $\tilde{A}_l$ and $\tilde{Z}_l$: the non-tilde quantities correspond to a single data point (rather than n_{B} of them as for $\tilde{A}_l$, $\tilde{Z}_l$).

The KFC Fisher Approximation. Just like with K-FAC for FC layers (see *Section A5.2.1*), we start by approximating our Fisher as block-diagonal (by assuming independence between loss derivatives at different layers). That is, we approximate the

[19] Note that $\underline{\mathrm{b}}_l$ is the bias vector for layer l

Fisher as

$$\tilde{F} := \text{block-diagonal}_{l\in\{1,\dots,N_L\}}\big(\tilde{F}_l\big), \tag{A4.29}$$

where $\tilde{F}_l$ is the diagonal block associated with layer l of the true Fisher. We have seen that this approximation is not sufficient to make computations practically feasible. For FC layers (in *Section A5.2.1*) we further Kronecker-factorized $\tilde{F}_l$ to achieve tractability, but the Kronecker-factors found there no longer hold for Conv layers [29].

For Conv layers, the Kronecker factorization which holds is

$$\tilde{F}_l = \mathcal{A}_{l,l} \otimes \Gamma_{l,l} \tag{A4.30}$$

with

$$\mathcal{A}_{l,l} = \mathbb{E}\Big[\langle A_l\rangle_H^T \langle A_l\rangle_H\Big] \quad \text{and} \quad \Gamma_{l,l} = \frac{1}{|\mathcal{T}|}\mathbb{E}\Big[\big(\mathsf{D}Z_l\big)^T \mathsf{D}Z_l\Big]. \tag{A4.31}$$

This result requires the **IAD assumption** (which we also had for FC K-FAC), but also two further assumptions which did not appear for FC layers [29]:

- **Spatial homogeneity (SH).** The first-order statistics of the activations are independent of spatial location. The second-order statistics of the activations and pre-activation derivatives at any two spatial locations t and t' depend only on $t - t'$. This implies there are functions j, ω and γ such that:

$$\mathbb{E}[a_{j,t}] = m(j), \quad \mathbb{E}[a_{j,t}a_{j',t'}] = \omega(j, j', t - t'), \quad \mathbb{E}[\mathsf{D}z_{i,t}\mathsf{D}z_{i',t'}] = \gamma(j, j', t - t'). \tag{A4.32}$$

- **Spatially uncorrelated derivatives (SUD).** The pre-activation derivatives at any two distinct spatial locations are uncorrelated: $\gamma(i, i', t' - t) = 0$ if $t' \neq t$.

The author ([29]) provides numerical evidence showing these approximations hold reasonably well in practice. Proving that the appropriate Kronecker factorization under the stated assumptions is indeed (A4.30)-(A4.31) becomes very technical fast [29]. However, the final result is remarkably easy to use[20] in practice. By employing Monte-Carlo techniques, we can obtain the approximations [29]

$$\mathcal{A}_{l,l} \approx \mathcal{A}^{(l)} := \frac{1}{n_{\text{B}}}\langle \tilde{A}_l\rangle_H^T \langle \tilde{A}_l\rangle_H \quad \text{and} \quad \Gamma_{l,l} \approx \Gamma^{(l)} := \frac{1}{n_{\text{B}}|\mathcal{T}|}\big(\mathsf{D}\tilde{Z}_l\big)^T \mathsf{D}\tilde{Z}_l, \tag{A4.33}$$

where the labels $\{y_i\}$ are drawn from the model distribution (but of course the dataset label[21] can also be used to save CPU time and perform "Empirical K-FAC"). We can then estimate our block-diagonal Fisher as

$$F \approx \text{block-diagonal}_{l\in\{1,\dots,N_L\}}\big(\mathcal{A}^{(l)} \otimes \Gamma^{(l)}\big) \tag{A4.34}$$

[20] And also an intuitive generalization from the FC case.

[21] Draw from q.

with $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$ as in (A4.33). Notably, the approximation (A4.34) has exactly the same format as we obtain in the case of FC layers[22] - with the only difference being the definition of $\mathcal{A}^{(l)}$ and $\Gamma^{(l)}$.

Desirable Properties. KFC step is invariant to oft-used reparameterizations, such as whitening and centering of the activations [29]. The matrices $\langle \tilde{A}_l \rangle$ and $\mathsf{D}\tilde{Z}_l$ are the same ones used to implement forward and backward passes through the conv layers - thus computing the (Conv) K-factors leverages the same efficient memory locality as the two passes do [29].

Convolutional K-FAC Numerical Performance. Based on the numerics in the original paper [29], convolutional K-FAC (KFC) outperformed SGD significantly both in terms of number of iterations and total CPU time, as long as KFC does not start overfitting. However, out of two presented datasets, KFC clearly exhibited strong overfit on one of them (CIFAR10), never exceeding 80% test-accuracy. This shows that one significant problem of K-FAC is overfitting. Other K-FAC (and Conv K-FAC) implementations (not from the original paper) show much better results with highly mitigated overfit issue, which is encouraging. However, when implementing our proposed K-FAC improvements, we still get a very gentle overfit (around 92-93% test-accuracy). While the issue is minuscule, it is sufficient to prevent our proposed K-FAC improvements from outperforming SENG[23] above 92-92.5% test-accuracy, though they outperform SENG for any accuracy before the overfit starts.

Furthermore, it is important to realize that the datasets used imply net architectures with relatively small widths. From *Chapter 1*, we know that K-FAC's computational complexity is cubic in width while SGD's one is quadratic. This tells us that SGD will always outperform K-FAC (and KFC) whenever the net width is sufficiently large (which practically occurs on ImageNet). This is where our proposed update steps in to rescue K-FAC-based solvers. In *Chapter 3* we propose a way to reduce the cubic complexity of K-FAC down to quadratic, while in *Chapter 4* we go a step further and even make it *"partly linear"*.

A5.3.2 SENG (Sketchy Empirical Natural Gradient)

Contrary to K-FAC, SENG (**S**ketchy **E**mpirical **N**atural **G**radient) is an *empirical NG method*, rather than a *true NG method* [20]. SENG uses *matrix sketching* ([115]) to

[22]Compare with (A4.18) and the discussion in *Section A5.2.1.*

[23]The state of art practical NGD implementation, which outperforms first order methods.

reduce cost. It can be implemented for both FC and Conv layers. We study from a Conv-layer perspective, and then think of the FC layer scenario as a degenerate case.

Derivation of Basic SENG step. SENG estimates the *empirical Fisher* as *block-diagonal* using only a mini-batch. Let us define $\mathcal{G}_{\mathcal{I}_\text{B}} = \sqrt{|\mathcal{I}_\text{B}|}[..., \nabla_\theta L(x_i, y_i; \theta), ...] \in \mathbb{R}^{d \times |\mathcal{I}_\text{B}|}$ with $i \in \mathcal{I}_\text{B}$, and drop the $\mathcal{I}_\text{B}$ index for convenience (knowing that $\mathcal{G}$ makes reference to a chosen batch). We have seen when describing K-FAC (*Sections 3.2.2* and *3.3.1*) that $\nabla_\theta L(x, y; \theta)$ has particular structure (see (A4.12)). In particular, it is a vertical stack of vectorized matrices (each of which corresponds to a layer), which are computed with quantities obtained in the forward and backward passes. Let us then define $\mathcal{G}_l \in \mathbb{R}^{w_l(w_{l-1}+1) \times |\mathcal{I}_\text{B}|}$ to be the horizontal stack of gradients w.r.t. parameters of layer l only, with all data points from $\mathcal{I}_\text{B}$ (one per column). This gives

$$\mathcal{G} = \mathcal{G}_{\mathcal{I}_\text{B}} = \begin{pmatrix} \mathcal{G}1 \\ \vdots \\ \mathcal{G}n_L \end{pmatrix}, \tag{A4.35}$$

which is another way to write $\mathcal{G}$. Recall that n_L is the number of layers.

For the remaining of *Section A5.3.2* we will use[24] the subscript to index the layers and the superscrip to index the datapoint (no index for the optimizations step).

Given this structure of $\mathcal{G}$, and recalling equation (1.21), we can see that a natural approximation for $\tilde{F}^{(E)} = \mathcal{G}\mathcal{G}^T$ is to use

$$\tilde{F}^{(E)} \approx \hat{F}^{(E)} := \text{block-diagonal}_{l \in \{1,...,n_L\}}\Big(\mathcal{G}_l \mathcal{G}_l^T\Big), \tag{A4.36}$$

Indeed, we have seen when investigating TONGA that when $\mathcal{G}$ has few columns, the block-diagonal approximation is a better choice not only from a computational point of view, but also from an accuracy point of view. However, note the subtle difference here that TONGA was using batch-gradients ($\tilde{g}$) at different locations to estimate blocks of $F^{(E)}$, while *SENG* is using different per-example gradients *at the same location*[25] [20]. Thus, the problem of computing the *regularized (approx.) empirical block-diagonal NG step*, namely $s^{(SENG)} = -(\hat{F}^{(E)} + \lambda I)^{-1}\tilde{g}$, reduces to solving n_L problems of the form

$$s_l^{(SENG)} = -(\mathcal{G}_l \mathcal{G}_l^T + \lambda I)^{-1}\tilde{g}_l, \tag{A4.37}$$

where $\tilde{g}_l \in \mathbb{R}^{w_L \times (w_{l-1}+1)}$ is the batch-gradient estimate w.r.t. the parameters in the l^{th} layer only [20]. By using the Sherman-Morrison-Woodbury (SMW) formula, we have

$$s_l^{(SENG)} = -\frac{1}{\lambda} g + \frac{1}{\lambda}\mathcal{G}_l b_l, \quad \text{with} \quad b_l := (\lambda I + \mathcal{G}_l^T \mathcal{G}_l)^{-1}(\mathcal{G}_l^T \tilde{g}_l). \tag{A4.38}$$

[24] This convention will break once the section ends.

[25] Given the definition of $F^{(E)}$, SENG is making the mathematically justifiable choice.

Rewrite the Step as Least-squares, then Sketch. Equation (A4.38) tells us that we would like to efficiently compute b_l. Our main problem is that while the height of $\mathcal{G}_l$ is smaller than d, it is still large enough for the computation of $\mathcal{G}_l^T\mathcal{G}_l$ (and even $\mathcal{G}_l^T\tilde{g}_l$) to be expensive. To reduce this cost and make the method competitive with first-order methods, we proceed by noting that we can rewrite b_l as the solution to regularized least-squares problem [20]

$$b_l = \arg\min_{b\in\mathbb{R}^{|\mathcal{I}_\mathrm{B}|}} \left\|\mathcal{G}_l b - \tilde{g}_l\right\|^2 + \lambda \left\|b\right\|^2. \tag{A4.39}$$

We can now reduce the dimension which was problematic by sketching with a matrix $\Omega_l \in \mathbb{R}^{\rho\times w_l(w_{l-1}+1)}$ to obtain

$$\tilde{b}_l := \arg\min_{b\in\mathbb{R}^{|\mathcal{I}_\mathrm{B}|}} \left\|(\Omega_l\mathcal{G}_l)b - \Omega_l\tilde{g}_l\right\|^2 + \lambda \left\|b\right\|^2, \tag{A4.40}$$

which can be solved as [20]

$$\tilde{b}_l = \left[\lambda I + (\Omega_l\mathcal{G}_l)^T(\Omega_l\mathcal{G}_l)\right]^{-1}[(\Omega_l\mathcal{G}_l)^T(\Omega_l\tilde{g}_l)]. \tag{A4.41}$$

We can now approximate our step as

$$\hat{s}_l^{(SENG)} = -\frac{1}{\lambda}g + \frac{1}{\lambda}\mathcal{G}_l\tilde{b}_l. \tag{A4.42}$$

Note that by performing the sketching, we have reduced our cost from computing $s^{(SENG)}$ in $\mathcal{O}\big(w_l^2(w_{l-1}+1)^2|\mathcal{I}_\mathrm{B}| + |\mathcal{I}_\mathrm{B}|^3\big)$ to computing $\hat{s}^{(SENG)}$ in $\mathcal{O}(\rho^2|\mathcal{I}_\mathrm{B}| + |\mathcal{I}_\mathrm{B}|^3 + w_l(w_{l-1}+1)|\mathcal{I}_\mathrm{B}|\big)$, provided we can sketch cheaply. Our bottle-neck now becomes assembling $\mathcal{G}_l$ (cost $\mathcal{O}(n_c w_l(w_{l-1}+1)|\mathcal{I}_\mathrm{B}|\big)$) and performing the multiplication $\mathcal{G}_l\hat{b}_l$ (cost $\mathcal{O}(w_l(w_{l-1}+1)|\mathcal{I}_\mathrm{B}|\big)$). Thus, reducing this cost would be particularly useful for wide nets (but may not be necessary otherwise). To see how we can achieve this for a relatively general case, let us consider the case of *CNNs*, and then view *FC nets* as a degenerate case of *CNNs*. In particular, let us consider the structure of $\mathcal{G}_l$ for *CNNs*:

$$\mathcal{G}_l = [\psi_l^{(1)}, ..., \psi_l^{(i)}, ..., \psi_l^{(|\mathcal{I}_\mathrm{B}|)}] \;\text{ with }\; \psi_l^{(i)} := \mathrm{vec}\big(\Gamma_l^{(i)}[\bar{A}_l^{(i)}]^T\big) \in \mathbb{R}^{w_l(w_{l-1}+1)}, \tag{A4.43}$$

where $\Gamma_l^{(i)} \in \mathbb{R}^{w_l\times n_c}$ and $\mathcal{A}_l^{(i)} \in \mathbb{R}^{(w_{l-1}+1)\times n_c}$ are *matrices*[26] which are analogous to the *vectors* (for data point x_i, y_i) δ_l and $\bar{a}_{l-1}$ that appear in the *FC* case (as we have seen when describing K-FAC). Note that i is indexing over different samples $\{x_i, y_i\}$ and l is indexing over different layers. Further note that for *FC nets* $n_c = 1$. The multiplication $\Gamma_l^{(i)}[\bar{A}_l^{(i)}]^T$, which is performed automatically in the backward pass is $\mathcal{O}(w_l(w_{l-1}+1)n_c)$. The naive approach is to compute $|\mathcal{I}_\mathrm{B}|$ such multiplications, which we then vectorize to assemble $\mathcal{G}_l$, and finally perform the multiplication $\mathcal{G}_l\hat{b}_l$

[26]This is specific to *Conv* nets, and the matrices degenerate to vectors in the *fully-connected* case.

(which is $\mathcal{O}(w_l(w_{l-1}+1)|\mathcal{I}_{\mathrm{B}}|)$). However, we can reduce the number of times we compute matrix-matrix multiplications of the cost $\mathcal{O}(w_l(w_{l-1}+1)n_c)$ by denoting $\tilde{b}_l = [\tilde{b}_l^{(1)}, ..., \tilde{b}_l^{(i)}, ..., \tilde{b}_l^{(|\mathcal{I}_{\mathrm{B}}|)}]$ (with $\tilde{b}_l^{(i)} \in \mathbb{R}$), and approximating [20]

$$\mathcal{G}_l\tilde{b}_l = \sum_{i\in\mathcal{I}_{\mathrm{B}}} \mathrm{vec}\big(\Gamma_l^{(i)}[\bar{A}_l^{(i)}]^T\big)\tilde{b}_l^{(i)} \approx \mathrm{vec}\Bigg[\bigg(\sum_{i\in\mathcal{I}_{\mathrm{B}}} \sqrt{|\tilde{b}_l^{(i)}|}\Gamma_l^{(i)}\bigg)\bigg(\sum_{i\in\mathcal{I}_{\mathrm{B}}} \frac{\tilde{b}_l^{(i)}}{\sum_{i\in\mathcal{I}_{\mathrm{B}}}\sqrt{|\tilde{b}_l^{(i)}|}}\bar{A}_l^{(i)}\bigg)^T\Bigg]. \tag{A4.44}$$

This approximation is based on the *IAD approximation* initially proposed by *Grosse and Martens* (2016, [29]) to extend the idea behind K-FAC to *CNNs*. Equation (A4.44) tells us that instead of computing $|\mathcal{I}_{\mathrm{B}}|$ matrix-matrix products of cost $\mathcal{O}(w_l(w_{l-1}+1)n_c)$ and one matrix-vector multiplication of cost $\mathcal{O}(w_l(w_{l-1}+1)|\mathcal{I}_{\mathrm{B}}|)$ to obtain $\mathcal{G}_l\tilde{b}_l$ *exactly*, we can compute just one matrix-matrix multiplication of cost $\mathcal{O}(w_l(w_{l-1}+1)n_c)$ and $2|\mathcal{I}_{\mathrm{B}}|$ scalar-matrix multiplications to get an *approximate* $\mathcal{G}_l\hat{b}_l$. That is, we can compute an approximate $\mathcal{G}_l\tilde{b}_l$ in $\mathcal{O}\big(w_l(w_{l-1}+1)n_c\big)$ instead of $\mathcal{O}\big(w_l(w_{l-1}+1)|\mathcal{I}_B|(n_c+1)\big)$. Indeed, this approach is preferred when w_l are large. Note that the same reasoning applies for *FC* with $n_c = 1$ (for *CNNs* $n_c > 1$).

Factor-wise Sketching. We can further reduce cost by noting that instead of forming $\mathcal{G}_l$, and then sketching, we can sketch $\Gamma_l^{(i)}$ and $\bar{A}_l^{(i)}$ directly. This corresponds to writing our sketch-matrix Ω as a kronecker product, $\Omega_l = \Omega_l^{(A)} \otimes \Omega_l^{(\Gamma)}$, computing the sketched matrices $\Omega_l^{(A)}\bar{A}_l^{(i)}$ and $\Omega_l^{(\Gamma)}\Gamma_l^{(i)}$, and then assembling $\Omega_l\mathcal{G}_l$ as [20]

$$\Omega_l\mathcal{G}_l = \Big[\mathrm{vec}\big([\Omega_l^{(\Gamma)}\Gamma_l^{(1)}][\Omega_l^{(A)}\bar{A}_l^{(1)}]^T\big), ..., \mathrm{vec}\big([\Omega_l^{(\Gamma)}\Gamma_l^{(|\mathcal{I}_{\mathrm{B}}|)}][\Omega_l^{(A)}\bar{A}_l^{(|\mathcal{I}_{\mathrm{B}}|)}]^T\big)\Big] \in \mathbb{R}^{\rho\times|\mathcal{I}_{\mathrm{B}}|}. \tag{A4.45}$$

The same trick can also be used for *FC nets*, noting that even if in that case the matrices $\Gamma^{(i)}$ and $\bar{A}_l^{(i)}$ degenerate to vectors, the same idea applies. Note that implementing the approaches proposed in equations (A4.44) and (A4.45) require modifying the backward pass, rather than just using the default algorithm. In practice, $\{\Omega_l\mathcal{G}_l\}_{l=1}^{n_L}$ are computed only once every few tens of iterations, and online estimates are maintained, in a similar way to K-FAC [20]. Thus, we incur the cost of constructing $\Omega_l\mathcal{G}_l$ (using (A4.45)) only rarely, but we encounter the cost of solving for $\tilde{b}_l$ as in (A4.41) and of obtaining $\mathcal{G}_l\tilde{b}_l$ (as in (A4.44)) on *each* iteration. Note that in *SENG*, $\mathcal{G}_l$ is *never* constructed. Only the related approximate/sketched quantities are. The authors propose that sketching be performed by sampling the rows of *the sketched matrix*, either with *uniform sampling* or based on *leverage scores*. They report that both approaches give the same progress per iteration, thus recommend *uniform sampling* (picking rows uniformly at random), since it is cheaper [20].

Numerical Performance of SENG. The authors show numerical results for training *FC nets* (*VGG16bn*, *ResNet18* and *ResNet50*) where SENG significantly outperforms *SGD with momentum*, ADAM, and K-FAC, with a particularly notable advantage in the large-batch distributed setting. The practical performance of SENG for *FC nets* remains yet to be clarified. Global convergence of SENG is guaranteed under typical *sketching* and *stochastic quasi-Newton* assumptions. Finally, the authors show that the results of *Zhang et. al.* (2019, [123]) which we investigated in *Section A2.2.4* (and told us that the convergence constant of exact NG is better than that of GD in overparameterized nets) also hold for SENG [20].

Time Complexity: Comaprison with K-FAC. A central theme in our thesis is that K-FAC has cubic time-complexity in layer size while SENG has quadratic time-complexity. We provide improvements to K-FAC for closing this gap and make K-FAC scale quadratically (*Chapter 3*) and "partially linearly (*Chapter 4*).

A5.3.3 Scalable Practical NGD (SP-NGD)

For very large training problems, feasible training times are typically achieved using *distributed parallel computing*. In this setting, the total batch size (typically) scales linearly in the number of workers, as the per-GPU batch size is fixed. For most practical distributed settings, the (overall) batch size becomes very large [51]. Large batch size helps convergence dues to reducing the variance of the gradient estimator, may lead to overfit[27] beyond a certain threshold (as empirically observed) [51, 137].

SP-NGD is a K-FAC improvement which partly alleviates K-FACs problem for large layer sizes[28]. It was used to train DNNs with very large batch-sizes (as large as 131,072 in only 873 steps and still obtain 74.9% (top-1) test accuracy (76.3% is the best to date, attained by [138, 139]). The authors of SP-NGD argue that their algorithm exhibits a smaller accuracy reduction with batch size (at very large batch-sizes), and that this is done in a more mathematically principled way (due to the K-FAC preconditioning), and not heuristically, like previous (successful) approaches: [138, 140, 141, 108, 109]. The main SP-NGD proposed changes to K-FAC are:

1. **Empirical K-FAC.** Use a K-FAC approximation of the *empirical Fisher* (not of the *true Fisher*). This removes the necessity of an extra backward pass at

[27]For a practical paper whose results slightly contradict this idea, but which otherwise provide similar insight see [89]. This may be because the batch-sizes used in that work were too small to enter the overfit regime.

[28]Though the time-complexity remains the same, the bottle-neck cost is reduced.

the expense of arguably worse properties [1, 64, 142, 143]. Still, the *empirical Fisher* approach does not degrade performance in their experiments [51].

2. **Adaptive Refresh of Statistics.** Refresh the statistics $\{\bar{A}_{l,l}\}_{l=0}^{n_L-1}$ and $\{\Delta_{l,l}\}_{l=1}^{n_L}$ in an *adaptive fashion*. The required statistics vary rapidly in the first phase of learning [71]. Thus, the naive strategy originally proposed by *Martens and Rogers* (2015, [1]), where we refreshed $\{\bar{A}_{l,l}\}_{l=0}^{n_L-1}$ and $\{\Delta_{l,l}\}_{l=1}^{n_L}$ every 20 steps, can be improved [71]. The proposed adaptive heuristic tries to determine the number of steps, $n_{\text{skip}}(X)$, for which it is *"acceptable"* to skip refreshing K-factor $X \in \{\bar{A}_{l,l}\}_{l=0}^{n_L-1} \cup \{\Delta_{l,l}\}_{l=1}^{n_L}$. By computing the dissimilarity between the current K-factor and previously held one, n_{skip} is heuristically adjusted on the fly [51]. This approach significantly reduces the cost of refreshing the statistics [51].

3. **Practical FIM Estimation for BatchNorm Layers.** *Batch normalization layers* are typically used in CNNs. These have as many parameters as the activations (a 3D tensor) at that layer, which are many more than the *"convolutional"* parameters. This means that the diagonal blocks of $\tilde{F}^{(E)}$ which correspond to *batch normalization* layers will be very large, and thus the bottleneck in computing our block-diagonal approximation of $\tilde{F}^{(E)}$ for CNNs. By noting that these diagonal blocks are themselves *Empirical Fishers*, the authors propose to further approximate them as block-diagonal, and then use the K-FAC approximations for those smaller blocks, rather than using the K-FAC approximation directly, as *Grosse and Martens* (2016, [29]) do. These smaller diagonal blocks are chosen to correspond to channels, similarly to what *Amari et. al.* (2019, [144]) proposed [51]. Roughly speaking, the proposed approach can be seen as reducing the maximum net *"width"*, thus cutting on the inversion cost dramatically (the authors report a speed-up of $3\times$ just from this approximation). However, note that while SP-NGD cuts down on the width of the bottle-neck layers (batch-norm), the other layers will remain with the same width and thus computational complexity as in K-FAC.

Based on the proposed changes, the authors provide a distributed parallelization scheme among multiple GPUs for which the overhead of *SP-NGD* (compared to SGD) decreases with the number of GPUs. This allowed them to train with *extremely large batches* (result previously mentioned), as well as training *CNNs* with *batch normalization* faster[29] than (the original) K-FAC [51].

[29]Results for this are not shown but implied through their results in *Figure 5*, and K-FAC results obtained by [20].