A Taylor polynomial expansion line search for large-scale optimization

by

Michael B. Hynes

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in

Applied Mathematics

University of Waterloo
Waterloo, ON, Canada, 2016
© Michael Hynes 2016

Author's Declaration

This thesis consists of material all of which I authored or co-authored; please refer to the enclosed Statement of Contributions for details. I certify that this document is a true copy of my thesis, including any required final revisions as accepted by my examiners. Furthermore, I bear all responsibility for any inconsistencies with previous work or errors otherwise appearing in this document, and understand that this thesis may be made electronically available to the public.

Statement of Contributions

SOME material herein has been published prior with joint authorship. A substantial portion of §2 and the entirety of §3 are reproduced almost verbatim from the paper *A polynomial expansion line search for large-scale unconstrained minimization of smooth L_2-regularized loss functions, with implementation in Apache Spark* [58], which appeared in the proceedings of the SIAM international conference on Data Mining in 2016 having coauthorship with Hans De Sterck. While I authored this paper, it received numerous improvements and modifications in consultation with Hans.[1]

[1]Related note: the use of the *royal we* throughout this thesis is a stylistic choice stemming from most of its chapters being facsimiled from aforesaid co-authored published works in which original forms the plural first-person was less disingenuous and had no associated deictic quandary. However, I am aware that some readers may experience a twinge of unease from a solely-authored document with a first-person plural narrative voice. If that's you, please consider the practical situation I faced of either maintaining or laboriously altering the voice and grammatical conjugations of the *circa* 10,000 words of existing academic work available *sic erant scriptum* before this document's exigent deadline. My personal ambivalence about having maintained the *royal we* stems mostly from its overt side-effect of casting this work in the smoke and mirrors of the academic rhetorical style in which the plural or third-

§4 contains work from the paper *Algorithmic Acceleration of Parallel ALS for Collaborative Filtering: Speeding up Distributed Big Data Recommendation in Spark* [108], which appeared in the proceedings of the 2015 International Conference on Parallel and Distributed Systems (ICPADS) and was co-authored with Manda Winlaw, Anthony Caterini, and Hans De Sterck. The results presented in §4 are my contributions from the latter half of that paper concerning the parallel implementation of the ALS-NCG algorithm. However, be aware Manda wrote the original introduction and mathematical formulation in the first half of the ICPADS manuscript, and hence much of mathematical typesetting and most of the citations are *excerpted* from her work (cf. also her thesis [107]).

person objective voices are used to confer a [possibly unwarranted] sense of authorativeness to statements that are [possibly] categorically editorial or downright biased. This subject is complex and intriguing in its own right, and I can envision a Nature letter for anyone who conducts a large-scale meta-analysis of academic rhetoric to probe its lairs. Something along the lines of the lexical analysis of voice in astrophysical papers by Tarone et al. [100], but on the grander scale of the recent sentiment analysis of Google's corpus of 20th century English books [1]. But digression aside, to counterbalance some of my qualms, there are numerous footnotes included throughout this document that, despite their plural voice, offer both further historical context for and commentary and critique on various aspects of this work. You are free to ignore them, of course—but if you are of a fastidious or persnickety persuasion, they may occasionally answer the question you just posed.

Abstract

IN trying to cope with the Big Data deluge, the landscape of distributed computing has changed. Large commodity hardware clusters, typically operating in some form of MapReduce framework, are becoming prevalent for organizations that require both tremendous storage capacity and fault tolerance. However, the high cost of communication can dominate the computation time in large-scale optimization routines in these frameworks. This thesis considers the problem of how to efficiently conduct univariate line searches in commodity clusters in the context of gradient-based batch optimization algorithms, like the staple limited-memory BFGS (LBFGS) method. In it, a new line search technique is proposed for cases where the underlying objective function is analytic, as in logistic regression and low rank matrix factorization. The technique approximates the objective function by a truncated Taylor polynomial along a fixed search direction. The coefficients of this polynomial may be computed efficiently in parallel with far less communication than needed to transmit the high-dimensional gradient vector, after which the polynomial may be minimized with high accuracy in a neighbourhood of the expansion point without distributed operations. This Polynomial Expansion Line Search (PELS) may be invoked iteratively until the expansion point and minimum are sufficiently accurate, and can provide substantial savings in time and communication costs when multiple

iterations in the line search procedure are required.

Three applications of the PELS technique are presented herein for important classes of analytic functions: (i) logistic regression (LR), (ii) low-rank matrix factorization (MF) models, and (iii) the feedforward multilayer perceptron (MLP). In addition, for LR and MF, implementations of PELS in the Apache Spark framework for fault-tolerant cluster computing are provided. These implementations conferred significant convergence enhancements to their respective algorithms, and will be of interest to Spark and Hadoop practitioners. For instance, the Spark PELS technique reduced the number of iterations and time required by LBFGS to reach terminal training accuracies for LR models by factors of 1.8–2. Substantial acceleration was also observed for the Nonlinear Conjugate Gradient algorithm for MLP models, which is an interesting case for future study in optimization for neural networks. The PELS technique is applicable to a broad class of models for Big Data processing and large-scale optimization, and can be a useful component of batch optimization routines.

Acknowledgements

First and foremost, I would like to recognize my supervisor Hans as being the only reason you're able to read this thesis. Not only did he take a chance on me as a student from a different discipline, he has always been patient, helpful, insightful—and also a good friend.

This thesis would also not have seen the light of day without considerable help from my love, Mona, who helped me cope whenever the going got hard.

Finally, I wish to thank above all my family: my brother, my Mom, and my Dad. They have sacrificed for me and helped me throughout my entire life. You mean the world to me, and I love you.

For Dean.

Contents

List of Figures xv

List of Tables xvii

List of Algorithms xix

1 Introduction 1
1.1 Line Search Algorithms 6

2 Apache Spark 13

3 Polynomial Expansion Line Search 23
3.1 Wolfe Approximate Line Searches 24
3.2 Polynomial Expansion Line Search 26
3.3 PELS Implementation In Spark 31
3.4 PELS Performance Tests 34
3.5 Results & Discussion 37
3.6 Conclusion 45

4 When ALS Met NCG 47
4.1 Low Rank Matrix Factorization 50
4.2 Alternating Least Squares 50
4.3 The ALS-NCG Algorithm 56

4.4 Parallel Performance of ALS-NCG 62
4.5 Conclusion 67

5 PELS for the Multilayer Perceptron **69**
5.1 Sigmoidal Multilayer Perceptron 72
5.2 PELS Formulation for the MLP 77
5.3 PELS Feasibility Tests 88
5.4 Results & Discussion 91
5.5 Conclusion 95

6 Conclusion **97**

References **99**

Appendices **111**

A Himrod Cluster Architecture **113**

List of Figures

2.1 Partitioning an RDD across a cluster. 16
2.2 Overview of a Spark map operation. 17
2.3 Spark communication patterns for reductions. . . . 19
2.4 Computing $\nabla\mathcal{L}$ with map/reduce in Spark. 21

3.1 Illustration of cubic interpolation. 25
3.2 Comparison of PELS vs. WA cubic interp. iterates. 31
3.3 Convergence traces for the Epsilon dataset. 39
3.4 Convergence traces for the RCV1 dataset. 39
3.5 Convergence traces for the URL dataset. 40
3.6 Convergence traces for the KDD-A dataset. 40
3.7 Convergence traces for the KDD-B dataset. 41
3.8 Speedup factors of LBFGS-P over LBFGS. 41

4.1 Illustration of the relationship $\mathbf{R} = \mathbf{U}^T\mathbf{M}$. 49
4.2 Hash partitioning of the columns of $\mathbf{M}$ 54
4.3 Schematic of the ALS routing table. 55
4.4 ALS-NCG convergence trace for MovieLens 20M . . 65
4.5 Linear scaling of ALS-NCG. 66
4.6 Speedup factors of ALS-NCG over ALS 67

5.1 Schematic of a feedforward MLP. 73

5.2 Example PELS models overlaid with an MLP loss. . 84
5.3 Convergence traces for the IRIS dataset. 93
5.4 Convergence traces for the PARKINSON'S dataset. . . 94
5.5 Convergence traces for the MNIST dataset. 94
5.6 Speedup factors of LBFGS-P and NCG-P. 95

List of Tables

3.1 Properties of LIBSVM classification datasets. 36
3.2 Default settings for Spark configuration parameters. 37
3.3 Timing measurements for PELS and WA algorithms. 43

5.1 Summary of datasets used for MLP classification. . . 91

A.1 Processor information for the `himrod` cluster. 114
A.2 Memory information for the `himrod` cluster 114

List of Algorithms

1 Limited-memory BFGS 9
2 Nonlinear Conjugate Gradient 11

3 Cubic Interpolating WA Line Search 26
4 Polynomial Expansion Line Search 29

5 Alternating Least Squares (ALS) 52
6 Nonlinearly Preconditioned NCG 58
7 RDD Block Vector `BLAS.axpby` 59

8 PELS for the MLP 89

1

Introduction

Everything under the sun (and the sun) is the result of an optimization problem. This is not hyperbole—or at least, not entirely: the second law of thermodynamics describes how particles and systems transition towards low energy states and achieve minimal potential energy at equilibrium. Take your existence, as a topical example. Your DNA is none other than an ordered sequence of three billion nucleotides being iterated upon each generation over millions of years such that the current material is hardy enough to survive its environment and be replicated in turn.[1] Furthermore, said replication often happens with a mate chosen as the solution to the formally-termed *stable matching* problem, which is the colourful optimization of pairing off three billion men with

[1]In fact, the field of *DNA Computing* is predicated on the idea of DNA being used as a mathematical *vector* in computations; beginning in 1994 with Leonard Adleman solving a Hamiltonian path problem [2], researchers are developing ways to exploit molecular interactions in DNA equivalently to register operations in conventional computers. Fans of Douglas Adams hitherto unaware of Adleman's work may now be experiencing an eerie feeling of foreboding—don't panic.

three billion women with minimal ensuing conflict. People seem particularly and almost invariably fixated on the matching problem, to the general mercantile benefit of others optimizing their revenues by offering cosmetic enhancements and geolocating dating services and [seemingly] unironic reality TV.

This thesis is written for that latter group: the good capitalists providing material support to consumers solving daily optimization problems of their own. With the omnipresence of the e-commerce, internet search, and social media industries in North American daily life, internet companies are inundated with data about online users' behaviours and interactions. While social media companies like Facebook and InterActiveCorp[2] are obvious progenitors of the data deluge, the ripples permeate far and wide via tracking capabilities provided to other sites and advertisers through the purchase of cookies from e.g. Google's subsidiary DoubleClick or Facebook's subsidiary Atlas Solutions.[3] The marriage of rampant

[2]IAC is an American conglomerate that doesn't get much mention unless you read the fine print, but owns all the big household dating subsidiaries like Tinder, OkCupid, Match.com, and PlentyOfFish—which basically makes them capable of studying and experimenting with North America's [explicit] electronic sexual selection habits, some of which are revealed in the book Dataclysm [92] by the co-founder of OkCupid, Chrisitan Rudder. Gratuitous example [according to Rudder]: while women target partners whose ages are approximately linearly correlated with their own, men of any age unanimously seek women in the youngest cohort on OkCupid. Admittedly, this little factoid merely reaffirms a phenomenon some of us are already uncomfortably aware of, like *teen* being the American porn industry's most frequent titular landmark [78].

[3]Facebook's transition from social media platform to universal advertising infrastructure via social promotion is a notable one (for a condensed history, see Gerlitz & Helmond [43]). Your authors posit that strong parallels abound between Facebook's incremental and ongoing transformation and how the credit card industry repurposed itself into the fundamental payment infrastructure for daily commercial purchases [22], even those for which drawing credit is financially *unnecessary*. In purview of how [both] credit card companies quietly became virtual tollbooths that affect the pricing of almost all consumer goods via merchant transaction fees and are (almost) without exception the *sole* payment method for online transactions, Facebook may be poised to wield similar powers—let's hope they have some scruples.

targeted behavioural advertising and cheap computer storage has culminated as one instance of the *Big Data* problem trumpeted by tech tabloids [and this thesis], whereby traditional computational methods for data analysis and pattern extraction cannot scale up to the sheer *amount* of data and still churn out results quickly enough. For instance, imagine trying to compute the singular values of a matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ when the problem dimension m can be greater than 10^7, and the number of rows (datapoints) is growing into the *billions*—fat chance.

The response to the Big Data problem over the past fifteen years has been twofold as both the systems and the algorithms have changed drastically. First of all, traditional workstations or high performance computing (HPC) clusters have been replaced by clusters of thousands of *commodity* hardware machines [6, 106],[4] such that data is distributed across the commodity cluster and network communication costs and faults (i.e. machine failures) are crucial factors in the completion times of jobs submitted to the cluster [87, 98]. Because standard parallel HPC techniques like the Message Passing Interface (MPI) [46] are ill-suited for this system architecture, the highly fault-tolerant MapReduce [32] processing paradigm has been a *de facto* standard since its publication by Google in 2004. The second trend has been the rise of *stochastic* algorithms for large-scale optimization [14] to become standard techniques for companies like Google [31] and Twitter [67], particularly over the past five years with the popularity of artifical neural network models for classification problems [64, 63, 97]. This ascent has been driven partly by the increasing cost of iterating through the massive datasets and concomittant communication costs of old-school deterministic algorithms, but also by the effectiveness of stochastic methods in the face of the redundancy and noise currently inherent to data dredged from clickbaited internet users, since these methods only calculate *approximate* function and gradient values by

[4]Commodity hardware is important jargon referring specifically to inexpensive, run-of-the-mill computer servers that can be purchased in bulk indiscriminately from any manufacturer [106]. Their purpose and usage are more apparent from their specs: these machines often have slow yet power-efficient CPUs with clock frequencies less than 2.0 GHz, compared to deluxe workstations that now reach 4.0 GHz.

sampling. Without editorializing the stochastic versus deterministic debate,[5] the underlying message is that for algorithms and their implementations to be useful in modern machine learning applications, the use of commodity hardware and the high cost of communication between nodes for prevalent MapReduce architectures are vital design criteria.

And that's where this thesis comes in: accelerating algorithms in the context of Big Data and distributed parallel systems. In particular we are concerned with efficiently solving the prototypical problem in machine learning—the minimization of a loss function averaged over a large dataset of many records—via gradient-based batch optimization algorithms that conduct univariate line searches in each iteration to minimize the loss function along the search direction generated by the algorithm. Accelerating the function and gradient evaluations in the line search is initially straightforward within a commodity cluster: multiple compute nodes can parallelize the function and gradient computations and reduce the CPU-bound computational time. However, this approach isn't a free lunch; the associated increased communication costs from the increasing number of compute nodes in the cluster eventually and ouroborically negate the savings, since network traffic remains orders of magnitude slower than the memory accesses required to retrieve data in the local computations [40]. Thus, the reduction in the size of data transferred between the compute nodes can be a determining factor in the efficiency of implementations of parallel batch algorithms, which communicate gradient vectors between compute nodes potentially multiple times per univariate line search.

The main contribution of this thesis is a general technique for efficiently performing parallel line searches in batch optimization routines. Our method is here termed the *Polynomial Expansion Line Search* (PELS), and is applicable whenever the loss function is analytic as in logistic regression, low rank matrix factorization, or

[5] Although it may interest the reader to know that Jorge Nocedal, one of the foremost researchers in the field of [traditional] numerical optimization, and discoverer of the staple LBFGS algorithm [84, 70], has for several years been working on stochastic variants of quasi-Newton methods [18, 19, 20]. Said Pliny the Elder: *Ruinis inminentibus musculi praemigrant.*

feedforward artificial neural network models. In this technique, we approximate the loss function by a truncated Taylor polynomial, whose coefficients may be computed efficiently in parallel with less communication than evaluating the gradient, since the Taylor coefficients of a low degree (i.e. 4–6) form a *smaller* vector than the gradient in large-scale problems. Once the coefficients are computed, the polynomial may be minimized with high accuracy in a neighbourhood of the expansion point, with the process repeated iteratively in a line search invocation until the expansion point and minimum are sufficiently accurate. The secondary contribution of this thesis is that we present the PELS method in the Apache Spark distributed data processing environment [114], which is an evolutionary descendent of the MapReduce framework and is designed to narrow the performance gap between HPC and MapReduce systems. Since Spark is designed to supersede Hadoop [the open-source MapReduce framework [12]], and is quickly being adopted by tech giants such as Amazon, eBay, and Baidu [according to Spark's developers], we hope that the results herein will be relevant for both industry and academia solving large-scale optimization problems both now and in the future.

Three applications of PELS are presented in the subsequent chapters, two of which appear in the context of the Apache Spark distributed data processing environment described in §2. The first application concerns the use of PELS in the convex L_2-regularized logistic regression optimization problem, and is presented in §3. There we provide a comparison of the common gradient-based Nonlinear Conjugate Gradient (NCG) and limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithms using standard line search techniques versus PELS for large-scale datasets in Spark with millions of records and model parameters. The second application is matrix factorization, for which §4 details an effective nonlinearly preconditioned version of the NCG algorithm in which the polynomial structure of the loss function allows the optimal step length in the line search to be computed *exactly* after the PELS coefficients are known. Finally, in §5 we derive expressions for the PELS coefficients for deep feedforward artifical neural network functions, and present initial serial comparison tests for the NCG and LBFGS algorithms.

The remainder of this introduction describes the mathematical

formulation of the optimization problems and LBFGS and NCG algorithms considered in later chapters. The notation attempts to follow the conventions of the fields of machine learning as presented by Hastie et al. [52] and numerical optimization as presented by Nocedal & Wright [85]. While most symbols can co-exist harmoniously, n.b. that we have adopted the convention from machine learning that the *parameter* vector be denoted $\mathbf{w}$ and a given datum *instance* be denoted by $\mathbf{x}$—don't let that get you down.

1.1 Line Search Algorithms

In large-scale optimization and machine learning, the problem often considered is the minimization of a scalar loss function $\mathcal{L}(\mathbf{w})$ with a parameter vector $\mathbf{w} \in \mathbb{R}^m$ [13, 52], where the value of $\mathcal{L}$ depends on a dataset of n observations, $\mathcal{R}$. Though the form of the data depends on the particular problem, we consider supervised learning data of input/response tuples, $\mathcal{R} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^p$ is the independent vector variable, and $y_i \in \mathbb{R}$ is the dependent scalar response variable. Using image classification as an example, the $\{\mathbf{x}_i\}$ may be a set of vectorized images with their accompanying enumerated labels $\{y_i\}$, whereas in linear regression each y_i would instead be a continuous value. Given a problem-dependent function f, the loss for each $(\mathbf{x}_i, y_i)$ is evaluated as $f(\mathbf{w}; \mathbf{x}_i, y_i)$, with the full loss $\mathcal{L}$ computed as the mean of f over the dataset,

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{w}; \mathbf{x}_i, y_i) + \lambda R(\mathbf{w}) \tag{1.1}$$

where $R(\mathbf{w})$ is an additional *regularization* term with coefficient $\lambda \geq 0$ designed to prevent *overfitting* to the data, or specify conditions on $\mathbf{w}$. In general the form of $R(\mathbf{w})$ considered here is an L_2 penalty, $R(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$, which is commonly used for many practical problems [112, 111], though other forms of regularization exist and are burgeoning field of study in their own right.[6]

[6] An optimization subfield garnering a lot of attention in the literature is that of *sparse* optimization. Here the regularization term is a *nonsmooth* L_1-norm penalty, which induces sparsity (i.e. few nonzero parameters, reducing storage requirements and model complexity). The lasso

The unconstrained optimization problem for (3.2) is to determine the optimal parameters $\mathbf{w}^*$ as

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \tag{1.2}$$

for which any unconstrained optimization algorithm may be applied. Gradient-based optimization algorithms such as LBFGS and NCG that solve (1.2) through iterative updates with *line searches* work by computing a search direction $\mathbf{p}_k \in \mathbb{R}^m$ in each kth iteration, and determining the next iteration's approximate solution $\mathbf{w}_{k+1}$ through the recurrence

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k,$$

where the scalar α_k is the *step size*. This step size is computed as the approximate or exact solution to the univariate line search problem for a given ray $\mathbf{w} + \alpha\mathbf{p}$ as

$$\alpha^* = \arg\min_{\alpha>0} \mathcal{L}(\mathbf{w} + \alpha\mathbf{p}) = \arg\min_{\alpha>0} \phi(\alpha). \tag{1.3}$$

As such, gradient-based optimization methods are inherently two-stage processes: first the direction is fixed, and afterwards the function is minimized along the fixed direction. While the outer algorithm prescribes how the $\{\mathbf{p}_k\}$ are chosen, the line search plays a large role in the *efficiency* of the entire minimization routine due to the number of evaluations of $\phi(\alpha)$ and $\phi'(\alpha)$ required. As will

method of Tibshirani [102] for sparse linear regression is a seminal and renowned example in statistics, and the QUIC method for sparse inverse covariance matrix estimation [55] is another recent and well-received application. These techniques are arguably another trend driven by the Big Data problem, or at least the *curse of dimensionality* interwoven with it [in which modern models used in prediction and regression have so many parameters that only a sparse solution is feasible or desirable]. To our knowledge, the PELS method is less immediately amenable to L_1-regularized models due to their non-differentiability. However, this may be an interesting direction of future research, since a general analysis technique in sparse optimization is to separate the loss function into its smooth and non-smooth components, and then conduct coordinate descent on the smooth local model with bound constraints derived from the non-smooth components [104].

be discussed in §3, in practice one often only seeks approximate solutions to (1.3) that satisfy the *Wolfe conditions* of sufficient decrease in magnitude and curvature of $\phi(\alpha)$, since these conditions are sufficient for convergence of the algorithms in the standard toolbox. However, note that an inexact line search cannot in general be expected to perform better than an exact line search [85].

1.1.1 Limited Memory BFGS

The (full) BFGS algorithm is a gradient-based quasi-Newton optimization algorithm developed simultaneously and independently by all of its eponymous discoverers in an annus mirabilis between 1969–1970 [34], and is widely credited as being the quasi-Newton update formula of choice [70, 34, 85]. The algorithm determines the search direction $\mathbf{p}_k$ in each iteration by forming an approximation to the inverse Hessian $\mathbf{H}_k^{-1} \approx [\nabla^2 \mathcal{L}]^{-1}$ from previous parameters $\{\mathbf{w}_j\}_{j \leq k}$ and gradient estimates $\{\mathbf{g}_j\}_{j \leq k}$, where $\mathbf{g}_k = \boldsymbol{\nabla} \mathcal{L}(\mathbf{w}_k)$.[7] Once formed, a search direction is computed as an approximate Newton step, $\mathbf{p}_k = -\mathbf{H}_k^{-1} \mathbf{g}_k$, after which a line search is performed. Furthermore, due to the quasi-Newton property, the direction $\mathbf{p}_k$ is often well-scaled such that the Newton step size $\alpha_k = 1$ often satisfies the Wolfe conditions.

However, note that the above BFGS formula for approximating $\mathbf{H}_k$ depends on *all* previously computed gradients $\{\mathbf{g}_j\}_{j \leq k}$ and parameters $\{\mathbf{w}_j\}_{j \leq k}$, which has a large storage cost for difficult or high dimensional problems. To rememdy this, the limited-memory BFGS [84, 70] algorithm is a modification to the original algorithm that stores only n_c previous gradients, such that the approximate Hessian is constructed from $\{\mathbf{w}_j\}_{j=k-n_c}^{k}$ and $\{\mathbf{g}_j\}_{j=k-n_c}^{k}$. This method is summarized in Alg. 1. With $n_c \leq 5$ in practice since further corrections do not substantially improve the approximation for large problems [70], the LBFGS algorithm has a *fixed*

[7] The actual theory and procedure for this is beyond this introduction's scope; see Nocedal & Wright [85] for the gritty details. In brief, the secant rule is enforced for each $(\mathbf{w}_{k+1} - \mathbf{w}_k)$ and $(\mathbf{g}_{k+1} - \mathbf{g}_k)$, and the Sherman-Woodbury formula allows $\mathbf{H}_k^{-1} \mathbf{g}_k$ to be computed using rank-one updates that need only inner products, multiplications, and additions. The whole idea is really quite nifty.

Algorithm 1: Limited-memory BFGS

Input: $\mathbf{w}_0 \in \mathbb{R}^m$
Output: $\mathbf{w}_k \approx \mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$
1 $\mathbf{p}_0 \leftarrow -\mathbf{g}_0$
2 $k \leftarrow 0$
3 **while** *not converged* **do**
4 $\quad \alpha_k \leftarrow \arg\min_{\alpha} \mathcal{L}(\mathbf{w}_k + \alpha \mathbf{p}_k)$
5 $\quad \mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \alpha_k \mathbf{p}_k$
6 $\quad k \leftarrow k + 1$
7 $\quad$ Compute $\mathbf{H}_k$ from $\{\mathbf{w}_j\}_{j=k-n_c}^{k}$ and $\{\mathbf{g}_j\}_{j=k-n_c}^{k}$
8 $\quad \mathbf{p}_k \leftarrow \mathbf{H}_k^{-1} \mathbf{g}_k$

and reasonable storage requirement. Coupled with its very fast linear convergence [85], it's no wonder that LBFGS has been a staple method in the optimization community [117, 27, 65].

1.1.2 Nonlinear Conjugate Gradient

The NCG algorithm was first proposed in 1964 for nonlinear optimization by Fletcher & Reeves [39], and has since received considerable use and refinement (the survey and review by Hager & Zhang [49] is highly recommended). As summarized in Alg. 2, the search direction $\mathbf{p}_{k+1}$ for the next line search is chosen for parameters $\mathbf{w}_{k+1}$ through a simple linear combination of the previous search direction and current gradient as

$$\mathbf{p}_{k+1} = \beta_k \mathbf{p}_k - \mathbf{g}_{k+1}, \tag{1.4}$$

where $\mathbf{p}_0 = -\mathbf{g}_0$, and various forms for the update parameter β_k exist. The motivation for such an update is to recombine directions in an intelligent manner such that the directions are *conjugate* and have the property that $\mathbf{p}_{k+1}^T(\mathbf{g}_{k+1} - \mathbf{g}_k) = 0$, which hopefully avoids the situation where $\mathbf{p}_{k+1}$ has considerable similarity to previous search directions. While the performance depends on both the problem and the form for β_k, one formula that consistently performs well numerically (and the only one considered here) is

the positive Polak-Ribière formula given by

$$\beta_k = \max\left(\frac{\mathbf{g}_{k+1}^T(\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T\mathbf{g}_k}, 0\right). \tag{1.5}$$

In particular, Alg. 2 with β_k as in (1.5) is globally convergent on non-convex functions with an inexact Strong Wolfe line search [44]. With minimal storage requirements and simple code, the NCG algorithm can be a very effective tool for certain problems [82], however there are some practical considerations to be bourne in mind for its effective use. To begin, the search directions $\{\mathbf{p}_k\}$ can be poorly scaled (in contrast to LBFGS), and several iterations with corresponding function/gradient evaluations are needed to find a step size satisfying the Wolfe conditions in a univariate line search. Because of this, it's necessary to estimate the initial step size α_k^0 in a univariate line search procedure, for which a common and simple technique is to take [85]

$$\alpha_k^0 = \alpha_{k-1}\frac{\mathbf{g}_{k-1}^T\mathbf{p}_{k-1}}{\mathbf{g}_k^T\mathbf{p}_k}, \tag{1.6}$$

where α_{k-1} is the accepted step size from the previous iteration as in (1.1), and $\alpha_k^0 = 1/\|\mathbf{g}_0\|$ for $k = 0$. Finally, to cope with the eventuality in which the search directions lose conjugacy, periodic restarts of the NCG search direction are often used. A popular restart method was given by Powell [89] that performs a gradient cosine check in each iteration; for a restart threshold $\gamma \approx 0.2$, β_k is set to zero if

$$\left|\frac{\mathbf{g}_{k+1}^T\mathbf{g}_k}{\mathbf{g}_{k+1}^T\mathbf{g}_{k+1}}\right| \geq \gamma. \tag{1.7}$$

Note also that when $\beta_k = 0$ in (1.4), a restart is implied in the NCG algorithm. While the restarts provide convergence guarantees by taking gradient descent steps by default, restarting too frequently is undesirable: while the Gradient Descent algorithm is globally convergent, it is quixotically and painstakingly slow [85].

Algorithm 2: Nonlinear Conjugate Gradient

Input: $\mathbf{w}_0 \in \mathbb{R}^m$

Output: $\mathbf{w}_k \approx \mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$

1 $\mathbf{p}_0 \leftarrow -\mathbf{g}_0$

2 $k \leftarrow 0$

3 **while** *not converged* **do**

4 $\quad \alpha_k \leftarrow \arg\min_\alpha \mathcal{L}(\mathbf{w}_k + \alpha \mathbf{p}_k)$

5 $\quad \mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \alpha_k \mathbf{p}_k$

6 $\quad$ Compute β_k

7 $\quad \mathbf{p}_{k+1} \leftarrow \beta_k \mathbf{p}_k - \mathbf{g}_{k+1}$

8 $\quad k \leftarrow k + 1$

2

Apache Spark

Apache Spark is a fault-tolerant, distributed cluster computing framework designed to supersede MapReduce by maintaining program data in memory as much as possible between distributed operations. It was first conceived in 2009 by Matei Zaharia [114], a then PhD candidate at UC Berkeley, and now operates as an open-source software project under the Apache Foundation umbrella. The system differs fundamentally from Hadoop and other MapReduce models by treating all its compute nodes identically; by contrast, the MapReduce framework explicitly allocates the available nodes in the cluster as either mappers or reducers with disjoint functions, such that the two sets must communicate an unnecessarily large amount of program data between themselves in each phase of computation. By using homogeneous compute nodes and synchronizing computational tasks, Spark's goal is to reduce unnecessary communication costs by maintaining program data in local RAM and to provide an expressive programming model better suited to iterative optimization algorithms in commodity clusters than MapReduce.

The Spark environment is built upon two components: a data abstraction, termed a resilient distributed dataset (RDD), and the

task scheduler, which uses a *delay scheduling* algorithm [113]. A Spark cluster is composed of a set of slave executor programs running on homogeneous compute nodes, and a driver program running on a master node that is responsible for managing on which nodes data is stored, and then scheduling and allocating tasks to the compute nodes based on data locality. Each executor program manages the local memory and processors of the machine on which it's instantiated, and executes machine instructions received from the master on its locally stored data. In this chapter we describe the fundamental aspects of RDDs and the basic operations of the scheduler, with an emphasis on conveying the general execution of the system as it pertains to the gradient-based optimization routines considered in §1.1.

RDDs are immutable, distributed datasets that are evaluated lazily via their provenance information—that is, their functional relation to other RDDs or datasets in stable storage. To formally describe an RDD, consider an immutable distributed dataset D of n records with homogeneous type: $D = \bigcup_i^n d_i$ with $d_i \in \mathcal{D}$. The distribution of D across a computer network of n_p nodes $\{v_t\}$, such that d_i is stored in memory or on disk on node v_t, is termed its *partitioning* according to a partition function $P(d_i) = v_t$. If D is expressible as a finite sequence of deterministic operations on other datasets $D_1, \ldots, D_l$ that are either RDDs or persistent records, then its lineage may be written as a directed acyclic graph L formed with the parent datasets $\{D_l\}$ as the vertices, and the operations along the edges. Thus, an RDD of type $\mathcal{D}$ (written $\texttt{RDD}\,[\mathcal{D}]$) is the tuple (D, P, L).[1] An illustration of an RDD partitioned across

[1] If you have ever heard of the functional programming paradigm, then you may be experiencing déjà vu from this paragraph, especially at the words *immutable* and *lazy*. The idea of specifying quantities as immutable, categorically-*typed* objects with functional provenance relationships has been kicking around in the theoretical computer science world since at least the late 1920s when λ-calculus was proposed by Alonzo Church as a formal system of logic [25]. The history is actually quite fascinating to read retrospectively (see Cardone & Hindley's account [21]), since it revolves around logicians attempting to devising axiomatic systems of computability just as the wrench of Kurt Gödel's incompleteness theorems was thrown at them. But be they inconsistent or not, the 1960s brought about renewed research into functional paradigms for computer

programming with work like McCarthy's on the LISP programming language [75], amongst others, to be followed by later formulations such as the ML language by Milner [79]. A large motivation for these λ-calculus-based systems is for a computer program to be written using mathematical operations and then *proven* correct using a series of irreducible algebraic steps (without side-effects) and the structure of the objects. While this all sounds rather rosey, the systems lose their shine [in your authors' opinion] the moment you have to understand whatever monads are to read or write a file. But that aside, a proper treatment of the RDD abstraction should in truth devolve into a formal specification of its list operations according to an appropriate axiomatic, functional specification (pick whichever one strikes your fancy—our favourites are those by Meertens [76, 77] and Bird [10, 9]). Now, don't strap yourself in yet, because we shan't go *that* far; but if you finish this extended footnote to get a feel for the basic building blocks of a Turing-complete functional programming language specification, hopefully the RDD abstraction will feel more intuitive.

Consider the ordered list $[x_1, \ldots, x_n]$ where all elements are drawn from the same space, $x_i \in \mathbb{X}$. The fundamental operation on list is the constructor or `cons` operator that constructs a list recursively from a scalar and an existing (possibly empty) list. The specification is `cons` : $\mathbb{X} \times [\mathbb{X}] \to [\mathbb{X}]$, where the set of all lists formable from the elements of $\mathbb{X}$ is denoted $[\mathbb{X}]$. . Denoting `cons` by ":", any list may be recursively constructed by applying `cons` to its elements

$$[x_1, \ldots, x_n] = x_1 : (x_2 : (\ldots (x_n : [\,]) \ldots)),$$

noting that $x : [\,] = [x]$ for any x and the empty list $[\,]$. The `concatenation` of two lists $[x_i], [\hat{x}_i] \in [\mathbb{X}]$ of respective lengths n and $\hat{n}$ is the operator $+\!\!+ : [\mathbb{X}] \times [\mathbb{X}] \to [\mathbb{X}]$. This operator can be expressed recursively in terms of `cons`, and gives the convenient shorthand

$$[x_i] +\!\!+ [\hat{x}_i] = [x_1, \ldots, x_n, \hat{x}_1, \ldots, \hat{x}_{\hat{n}}].$$

A `map` operation on $[x_i] \in [\mathbb{X}]$ is the element-wise application of a function $f : \mathbb{X} \to \mathbb{Y}$ on $[x_i]$. Formally written $* : f \times [\mathbb{X}] \to [\mathbb{Y}]$, we have that

$$f * [x_1, \ldots, x_n] = [Tx_1, \ldots, Tx_n] = [y_i]$$

for $[y_i] \in [\mathbb{Y}]$. A `reduce` operation on $[x_i]$ produces a scalar value in $\mathbb{X}$. For an associative infix operator $\oplus : \mathbb{X} \times \mathbb{X} \to \mathbb{X}$, the reduction of $\oplus$ on $[x_i]$ is $(/) : \oplus \times [\mathbb{X}] \to \mathbb{X}$:

$$\oplus / [x_1, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n = \bigoplus_i^n x_i.$$

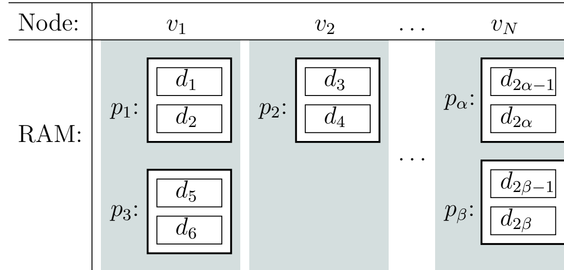


Figure 2.1: *Distribution of RDD partitions* $\{p_1, p_2, \ldots\}$ *of the dataset* $\{d_i\}$ *in the RAM of* N *compute nodes. Note that the enumeration of the partitions does not correspond to either that of the RDD elements or the compute nodes, which is representative of the physical distribution of data in a Spark cluster. Furthermore, the number of partitions stored per node is depicted as unequal in this example, which is possible for RDDs.*

several compute nodes is shown in Fig. 2.1, where the partitions of elements $\{d_i\}$ have been denoted as $\{p_1, p_2, \ldots\}$. Note that the enumeration of the partitions does not correspond to either that of the RDD elements or the compute nodes.

Physically computing the records $\{d_i\}$ of an RDD is termed its *materialization*, and is managed by the Spark scheduler program. To allocate computational tasks to the compute nodes, the scheduler traverses an RDD's lineage graph L and divides the required operations into *stages* of local computations on parent RDD partitions. Suppose that $R_0 = (\bigcup_i x_i, P_0, L_0)$ is an RDD of numeric type $\texttt{RDD}[\mathbb{R}]$, and let $R_1 = (\bigcup_i y_i, P_1, L_1)$ be the RDD resulting from the application of a function $f : \mathbb{R} \to \mathbb{R}$ to each record of

(continued) Finally, the `filter` operation applies a logical predicate φ to each member of $[x_i]$, and returns only those elements for which $\varphi(x_i)$ is true. For $\varphi : \mathbb{X} \to \mathbb{B}$ where $\mathbb{B}$ is the boolean set the `filter` operator is denoted $\triangleleft : \varphi \times [X] \to [X]$, where

$$\varphi \triangleleft [x_i] = [\hat{x}_i], \{\hat{x}_i \in [x_i] : \varphi(\hat{x}_i)\}.$$

These operations can be recursively composed to form the *higher-order* functions of MapReduce frameworks, with the details here omitted in the interest of ever concluding this chapter.

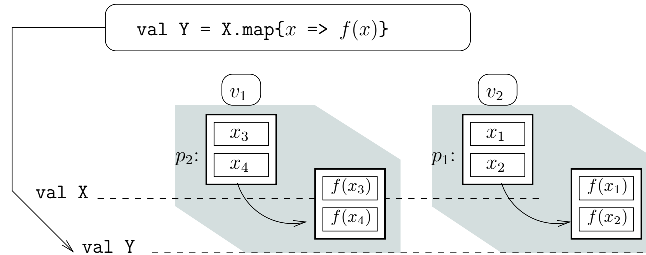


Figure 2.2: *Overview of a Spark map operation of a function $f(x)$ on an RDD `X`. The `Scala` code snippet encircled at the top expresses the application of $f(x)$ to all elements of `X`. The two compute nodes apply the function to the elements of `X` in their local partitions, creating partitions of the RDD `Y`, which has an identical partitioning to `X` due to the narrow dependency.*

R_0. To compute $\{y_i\}$, R_1 has only a single parent in the graph L_1, and hence the set of tasks to perform is $\{f(x_i)\}$. This type of operation is termed a *map* operation, and has a *narrow* lineage dependency: $P_1 = P_0$, and the scheduler would allocate the task $f(x_i)$ to a node that stores x_i since each y_i may be computed locally from x_i. Fig. 2.2 depicts a map operation for a function $f(x)$ on an RDD `X`; for the two compute nodes shown, the elements of `X` in the local partitions are mapped to those of the RDD `Y`, which has an identical partitioning to `X` due to the narrow dependency.

Stages consist only of local map operations, and are bounded by *shuffle* operations that require communication and data transfer between the compute nodes. For example, shuffling is necessary to perform *reduce* operations on RDDs, wherein a scalar value is produced from an associative binary operator applied to each element of the dataset. In implementation, a shuffle is conducted by writing the results of the tasks in the preceding stage, $\{f(x_i)\}$, to a local file buffer. These shuffle files may or may not be written to disk, depending on the operating system's page table, and are fetched by remote nodes as needed in the subsequent stage via a TCP connection. Shuffle file fetches occur asynchronously, and multiple connections between compute nodes to transfer information are made in parallel. In addition, map tasks on the previous stage's results that are stored locally by a compute node will occur

concurrently with remote fetches.[2]

A reduce operation on an RDD of type `RDD`$[\mathcal{A}]$ produces a scalar value of type $\mathcal{A}$ by an application of an associative binary operator $\oplus : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ to all of the elements $\{a_i\}$ as

$$a_1 \oplus a_2 \oplus \cdots \oplus a_n = \bigoplus_{i=1}^{n} a_i. \tag{2.1}$$

Reduce operations require first performing *local* reductions on the partitions stored locally by each node before communicating the nodes' results to the driver. The local reduction of the partition of $\{a_i\}$ stored on node v_t is written $a^{[t]}$ and computed as $a^{[t]} = \bigoplus_{i_l \in \mathcal{I}_t} a_{i_l}$ for $\mathcal{I}_t = \{i : P(a_i) = v_t\}$ and requires no communication between nodes. The full reduction for multiple nodes is then $\bigoplus_{t=1}^{n_p} a^{[t]}$, which incurs communication costs dependent on n_p and the size (in bytes) of an element in $\mathcal{A}$: reducing scalars ($\mathcal{A} = \mathbb{R}$) is cheaper than reducing vectors ($\mathcal{A} = \mathbb{R}^p$). Additionally, reduce operations on an RDD may be performed in two ways: either through an all-to-one communication pattern (Fig. 2.3a) in which all n_p local results from the compute nodes are communicated to the host machine on which the driver program is running, or through a multi-level tree communication [3] where intermediary (locally reduced) results are aggregated by compute nodes in n_l levels before being transferred to the driver [109] (Fig. 2.3b).

[2] For more details on the Spark shuffle, see Ousterhout [86] and Davidson & Or [29]—but if you want more details than that, just throw in the towel. While the Spark project has many helpful people and a large amount of documentation about how to *deploy* Spark code, one notable shortcoming [in your authors' opinion] is the lack of precise and specific documentation about the system's internals. Ousterhout's aforementioned shuffle documentation was written in response to an email thread on the developer mailing list involving your authors' continued questions about how shuffle was implemented—she was kind enough to give authoritative answers replete with extended footnotes. But if you want to generally understand the control flow or hierarchical structure of the Spark codebase, it can be read the code or bust, which is a hidden cost for organizations adopting Spark if they ever reach a juncture at which the optimization of performance in their codebase or diagnosis of unexpected behaviour requires knowledge of system internals.

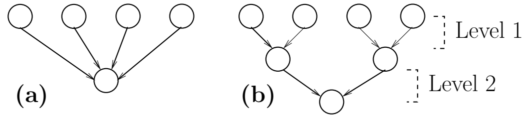


Figure 2.3: *Comparison of an (a) all-to-one communication where every node, depicted by circles, communicates results with the driver (bottom circle), and a (b) multi-level scheme with $n_l = 2$, such that the final reduction to the driver requires communication with only 2 nodes.*

In §1 we discussed how the communication costs of gradient evaluations can be an expensive component of the univariate line search. Having now reviewed the architecture of the Spark system, we recapitulate this idea, using the evaluation of the gradient $\nabla\mathcal{L}$ for a mean loss function as an illustrative example. We consider the dataset of observations $\mathcal{R} = \{(\mathbf{x}_i, y_i)\}$ to be stored as an RDD with type $\texttt{RDD}\,[(\mathbb{R}^p, \mathbb{R})]$, but assume the $\nabla\mathcal{L}$ vector is small enough such that the master node stores both it and the other vectors necessary in an optimization procedure such as Alg. 1 or Alg. 2. In such a case, the gradient computations are *farmed* out to the compute nodes, with the results shuffled back to the master node. Since $\nabla\mathcal{L}$ is the mean of all $\{\nabla f(\mathbf{w}; \mathbf{x}_i, y_i)\}$, it is easily expressible through map/reduce operations by first mapping $f(\mathbf{w}; \mathbf{x}_i, y_i)$ onto $\mathcal{R}$ to form a new RDD of gradients, and subsequently reducing this new RDD where the operator $\oplus$ is vector addition. This process is depicted in several stages in Fig. 2.4 for two compute nodes and a dataset with four records, where the shorthand f_i is used to denote $f(\mathbf{w}; \mathbf{x}_i, y_i)$. Fig. 2.4a shows the functional pseudocode for the calculation, and Fig. 2.4b illustrates the mapping of the gradient function onto the RDD of $\mathcal{R}$. In Fig. 2.4c, an implementation feature of a reduce operation is shown in which both nodes perform *map-side* reductions of locally stored gradient vectors. The writing of shuffle files occurs in Fig. 2.4d, and the network communication occurs in Fig. 2.4e. The final stage occurs in Fig. 2.4f, in which the gradients are reduced to the master node, which averages the results to form a consolidated vector. Note that the network shuffle in Fig. 2.4e is dependent on the number of compute nodes as well as the dimension of $\nabla\mathcal{L}$: the more nodes and the larger the vector,

the greater the cost of shuffling.[3]

To conclude this chapter, it behooves us to mention the fault tolerance built into the Apache Spark system, since it is often run on commodity Hadoop clusters. The system's fault tolerance is achieved through the functional lineage graph specified by any RDD, and is managed by the scheduler using the simple delay scheduling algorithm [113] which prioritizes data locality when submitting tasks to the available compute nodes. If the node v_α stores the needed parent partition containing element x_i to compute task $f(x_i)$, but is temporarily unavailable due to faults or stochastic delays, rather than submitting the task on another node, the scheduler will wait until v_α is free. In the case that v_α does not become available within a specified maximum delay time (several seconds in practice), the scheduler will resubmit the tasks to a different compute node. However, if the partition containing x_i is not available in memory on this other node, the lineage of the RDD containing x_i must be traversed further back in ancestry, and tasks required to recompute x_i afresh from the parent RDDs will be submitted for computation *in addition to* and *before* the task $f(x_i)$. In this way, fault tolerance is achieved in the system through recomputation.[4]

---

[3] It also bears noting that Fig. 2.4 shows an all-to-one communication pattern for simplicity, though it is generally less efficient than a multilevel scheme for large clusters. Furthermore, readers familiar with Spark should be quick to point out that terser idioms exist to compute $\nabla\mathcal{L}$ than explicitly using map and reduce operations. We are sympathetic to your cause, but confine the discussion to the simplest functional operations available for pedagogical purposes.

[4] Although it's no stretch of the imagination to consider pathological circumstances where the recomputation model stagnates if failures occur frequently enough to result in recomputation *in perpetuum*, we are unaware of any reported cases. At any rate, in commodity clusters large enough that the failure probability for at least one machine is high at any given time, it is possible to replicate the data contained within RDDs by sufficiently large integer factors (for Hadoop, this replication factor may be 3–4 [106]) in order that the data contained by any machine is accessible elsewhere should it become unresponsive. In addition, Spark provides a *checkpointing* mechanism that writes in-memory data to stable storage, such that any RDDs dependent on the stored data have shorter lineage graphs that require fewer stages of recomputation in the event of failures.

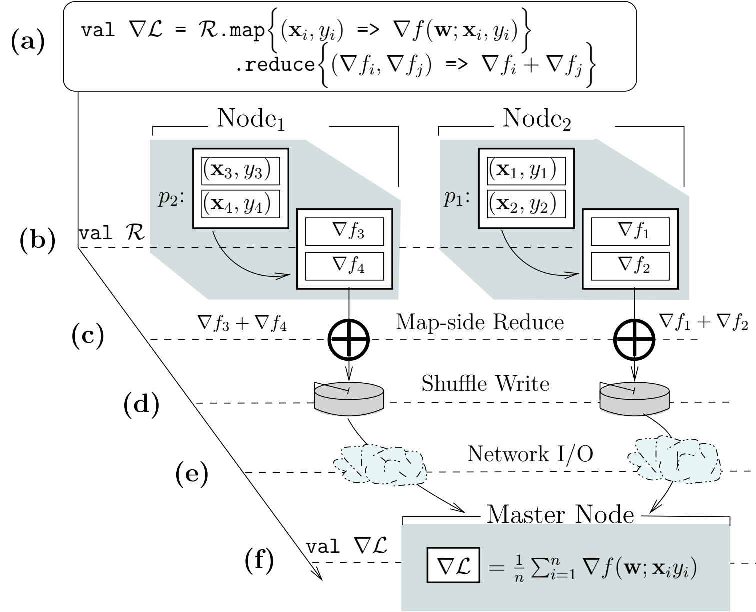


Figure 2.4: *Schematic of the procedure for computing $\nabla\mathcal{L}$ with map/reduce operations in Apache Spark with two example compute nodes with $\mathcal{R}$ stored in two partitions, p_1 and p_2. Subplot **(a)** shows sample map and reduce pseudocode to evaluate $\nabla\mathcal{L}$ on a distributed dataset $\mathcal{R}$. In **(b)** the gradients are evaluated for each $(\mathbf{x}_i, y_i)$ locally stored by the nodes. Map-side reduction of subgradients into* one *m-dimensional vector per node occurs in **(c)**, and these resultant vectors are written to shuffle files in **(d)**. Network shuffle transfer occurs in **(e)**, and the master node averages each partial gradient obtained from the network to obtain $\nabla\mathcal{L}$ in **(f)**.*

3

Polynomial Expansion Line Search

HAVING now some appreciation for the predicament that befalls optimization methods in parallel distributed systems, we turn our attention to the meat of the matter. In this chapter, we present the Polynomial Expansion Line Search (PELS) for solving the univariate line search problem

$$\alpha^* = \arg\min_{\alpha>0} \mathcal{L}(\mathbf{w} + \alpha\mathbf{p}) = \arg\min_{\alpha>0} \phi(\alpha), \tag{3.1}$$

where $\mathbf{w}, \mathbf{p} \in \mathbb{R}^m$ are fixed and $\mathcal{L}$ is a mean loss function:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{w}; \mathbf{x}_i, y_i) + \lambda R(\mathbf{w}). \tag{3.2}$$

The main idea of PELS is simple but powerful: we propose to approximate $\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ in a univariate line search by a low-degree polynomial expansion in α, and show that the coefficients of this polynomial may be computed in a single pass over the dataset with modest communication requirements, after which the polynomial approximation may be minimized with high accuracy. In each line search invocation, the expansion may be repeated iteratively until the expansion point and minimum are sufficiently accurate.

The advantages of PELS stem from two main improvements. Firstly, the PELS technique can obtain more accurate minima when the high intrinsic potential accuracy of the polynomial expansion is realized, which can lead to significantly fewer iterations of the optimization method to reach a desired accuracy than with classical approximate line searches. Secondly, if multiple iterations are required in a line search, the PELS method is much more efficient in terms of parallel communication than the iterations in approximate line searches that seek to impose the Wolfe conditions and require evaluating $\nabla\mathcal{L}$ in each line search iteration: aggregating the polynomial coefficients requires much less communication than aggregating the m-dimensional gradient vectors $\{\nabla f(\mathbf{w}; \mathbf{x}_i, y_i)\}$. Additionally, when $\phi(\alpha)$ is itself polynomial, the PELS expansion can be made *exact* with a sufficiently large degree.

The contributions of this chapter are organized as follows. In §3.1, the standard line search technique for solving (3.1) approximately using the Wolfe conditions is presented, followed by the PELS algorithm in §3.2. In §3.3 we detail our implementation in the Apache Spark framework for fault-tolerant distributed computing, where it has been used to accelerate the training of L_2-regularized logistic regression models by NCG and LBFGS on several large binary classification datasets in experiments described in §3.4. Finally, §3.5 compares the performance of these algorithms in Apache Spark using either PELS or a standard approximate line search scheme.

3.1 Wolfe Approximate Line Searches

Since computing $\phi(\alpha)$ and $\phi'(\alpha) = \mathbf{p}^T\nabla\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ are expensive operations, often an approximate solution to (3.1) is sought instead of an exact solution in order that the number of function evaluations required in the line search may be reduced. Approximate line searches compute a sequence of iterates $\{\alpha_j\}_{j\geq 0}$ until convergence criteria are satisfied, which are normally the Strong Wolfe conditions [85]. These conditions are a set of two inequalities guaranteeing (i) sufficient decrease as

$$\phi(\alpha) \leq \phi(0) + \nu_1\alpha\phi'(0), \tag{3.3}$$

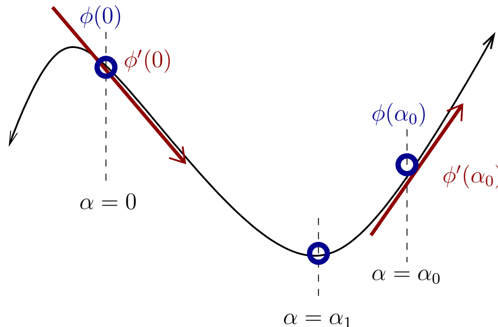


Figure 3.1: *Illustration of a cubic interpolating polynomial constructed from function & gradient values at the two control points, $\alpha = 0$ and $\alpha = \alpha_0$. The control values are $\phi(0)$ and $\phi'(0)$, and $\phi(\alpha_0)$ and $\phi'(\alpha_0)$, where the gradients are shown as arrows tangent to the interpolating curve at. The point α_1 is here determined as the (local) minimum of the constructed polynomial.*

and (ii) a curvature condition requiring

$$|\phi'(\alpha)| \leq \nu_2 |\phi'(0)| \tag{3.4}$$

for $0 < \nu_1 < \nu_2 < 1$ (where $\nu_1 \approx 10^{-4}$ and $\nu_2 \approx 0.9$ [85]).

There are a multitude of widely-used inexact univariate line search algorithms for satisfying (3.3) and (3.4) for general $\mathcal{L}(\mathbf{w})$ [80, 4, 47, 48], and we refer to an algorithm in this class as a *Wolfe approximate* (WA) line search. Most successful WA algorithms are variants of the following classic interpolation scheme, summarized in Alg. 3. In each jth iteration of the line search, an interval $I_j = [\alpha_l, \alpha_u]$ containing α^* is determined, and the next α_{j+1} is generated by the minimization of an interpolated cubic polynomial $P_j(\alpha)$ with control points $\phi(\alpha_l), \phi(\alpha_u), \phi'(\alpha_l)$, and $\phi'(\alpha_u)$ (i.e. $P_j(\alpha_l) = \phi(\alpha_l)$, $P_j'(\alpha_l) = \phi'(\alpha_l)$, etc as shown in Fig. 3.1). The next interval I_{j+1} is computed such that the endpoints are closer to α^*, and methods such as bisection, secant, and dual interpolation/minimization can be used to shrink the interval. Each evaluation of $\phi(\alpha)$ and $\phi'(\alpha)$ requires an $\mathcal{O}(n)$ pass over $\mathcal{R}$, and it is typical in publicly available codes to structure optimization routines with a function that computes both $\phi(\alpha)$ and $\phi'(\alpha)$ simultaneously by evaluating $\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ and $\nabla\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$, explicitly

Algorithm 3: Cubic Interpolating WA Line Search

Input: $\alpha_0 > 0$

Output: α_j satisfying (3.3) and (3.4) for $\phi(\alpha)$

```
1 I_0 ← [0, α_0]
2 j ← 0
3 while α_j does not satisfy (3.3) and (3.4) do
4     [α_l, α_u] ← I_j
5     P_j(α) ← interpolate φ(α_l), φ(α_u), φ'(α_l), φ'(α_u)
6     α_{j+1} ← arg min_α P_j(α)
7     I_{j+1} ← update interval using α_l, α_u, α_{j+1}
8     j ← j + 1
9 return α_j
```

returning a scalar and an m-dimensional vector such that, if the current step size is accepted, the computed value of $\nabla\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ provides $\nabla\mathcal{L}(\mathbf{w}_{k+1})$ for the next iteration. Note, however, that if the initial α_0 in Alg. 3 satisfies (3.3) and (3.4), it is accepted as the solution to (3.1) *without* constructing and minimizing $P_0(\alpha)$, regardless of the accuracy of α_0.

3.2 Polynomial Expansion Line Search

Our goal with the PELS method is to solve (3.1) accurately with the cheapest *distributed* operations possible when both the regularization $R(\mathbf{w})$ and individual loss $f(\mathbf{w}; \mathbf{x}_i, y_i)$ are smooth with respect to $\mathbf{w}$. In this case, we consider the Taylor expansion of $\phi(\alpha) = \mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ in terms of α, which requires summing the Taylor expansions of each $f(\mathbf{w}; \mathbf{x}_i, y_i)$ in (3.2) in addition to an expansion for $R(\mathbf{w})$. For an expansion about a step size α_j, we have

$$\mathcal{L}(\mathbf{w} + \alpha\mathbf{p}) = Q_\lambda(\alpha)|_{\alpha_j} + \sum_{i=1}^{n}\sum_{\ell=0}^{\infty} b_{\ell,i}(\alpha - \alpha_j)^\ell$$

where $Q_\lambda(\alpha)|_{\alpha_j}$ is a polynomial expansion of $\lambda R(\mathbf{w})$, and $\{b_{\ell,i}\}_{\ell=0}^{\infty}$ are the coefficients in the expansion of $f(\mathbf{w}; \mathbf{x}_i, y_i)$ that depend explicitly on $(\mathbf{x}_i, y_i)$ and the direction $\mathbf{p}$. To make this representation

amenable to a distributed setting, we reorder the summations and write the degree-d approximation to $\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ as

$$W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j} = Q_\lambda(\alpha)|_{\alpha_j} + \sum_{\ell=0}^{d} c_\ell(\alpha - \alpha_j)^\ell, \tag{3.5}$$

which has a truncation error $\varepsilon_{d+1}(\alpha)|_{\alpha_j}$ as

$$W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j} = \mathcal{L}(\mathbf{w} + \alpha\mathbf{p}) - \varepsilon_{d+1}(\alpha)|_{\alpha_j}. \tag{3.6}$$

This error is an $\mathcal{O}\big((\alpha - \alpha_j)^{d+1}\big)$ term and hence small for α near α_j. Each coefficient c_ℓ in (3.5) contains a summation over the dataset as

$$c_\ell = \sum_{i=1}^{n} b_{\ell,i} = \sum_{i=1}^{n} F_\ell(\mathbf{r}, \mathbf{p}; \mathbf{x}_i, y_i), \tag{3.7}$$

where $\mathbf{r} = \mathbf{w} + \alpha_j\mathbf{p}$, and the functions $\{F_\ell\}_{\ell=0}^{d}$ compute the coefficients for the ℓth terms for a single observation.

The PELS algorithm exploits the following facts: (i) computing the coefficients in (3.7) is a *parallelizable* operation, and (ii) once the $\{c_\ell\}$ are computed, $W(\alpha)|_{\alpha_j}$ is a useful approximation that may be used to estimate minima in a neighbourbood of α_j. The PELS method proceeds as follows. Starting from the iterate α_j, a polynomial approximation $W(\alpha)|_{\alpha_j}$ is constructed by computing the coefficients $\{c_\ell\}$ in parallel, after which the subsequent iterate is determined as

$$\alpha_{j+1} = \arg\min_{\alpha>0} W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}. \tag{3.8}$$

Solving (3.8) is a subproblem that will generate further iterates in a minimization routine, however, since the coefficients of $W(\alpha)|_{\alpha_j}$ are *fixed*, the minimization problem requires no further distributed operations. In addition, computing the first and second derivatives $W'|_{\alpha_j}$ and $W''|_{\alpha_j}$ are inexpensive $\mathcal{O}(d+1)$ operations in a minimization routine when performed with Horner's rule. If α_{j+1} is found to be insufficiently accurate, then the process repeats iteratively: α_{j+1} is chosen as the new expansion point, and the coefficients of $W(\alpha)|_{\alpha_{j+1}}$ are computed. This general form of the PELS algorithm is summarized in Alg. 4, where the coefficients

for $W(\alpha)|_{\alpha_j}$ are denoted by the vector $\mathbf{c}_j = [c_0, c_2, \ldots, c_d]^T$, and the procedure CALC_COEFFS for computing the coefficients in the Apache Spark framework will be described in §3.3.

As a termination condition at step 7 of Alg. 4, we propose using an error metric $\epsilon(\alpha)$ that estimates the fractional error in the value of $W(\alpha_{j+1})|_{\alpha_j}$. The estimation uses the difference between polynomial approximations to $\mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$ of degree d and degree $(d-1)$, and is based on the intuition that a degree-d Taylor polynomial is heuristically bounded above by the error in the polynomial of degree $(d-1)$ in the relevant neighbourhood of the expansion point α_j. Thus, we denote the polynomial of degree $(d-1)$ by W_{d-1}, and take

$$\epsilon(\alpha)|_{\alpha_j} = \left|\frac{W(\alpha)|_{\alpha_j} - W_{d-1}(\alpha)|_{\alpha_j}}{W(\alpha)|_{\alpha_j}}\right| = \left|\frac{c_d(\alpha - \alpha_j)^d}{W(\alpha)|_{\alpha_j}}\right|. \tag{3.9}$$

It can be readily seen that (3.9) uses the highest order term in $W(\alpha)|_{\alpha_j}$ as the approximate truncation error. While the true error in $W(\alpha)|_{\alpha_j}$ is an unknown $\mathcal{O}\big((\alpha - \alpha_j)^{d+1}\big)$ term, $\epsilon(\alpha)|_{\alpha_j}$ can be a particularly effective proxy when we note that $\epsilon(\alpha)|_{\alpha_j} \to 1$ as $|\alpha - \alpha_j| \to \infty$, which is in accordance with our knowledge that $W(\alpha)|_{\alpha_j}$ is a 100% incorrect approximation to $\mathcal{L}$ for large $|\alpha - \alpha_j|$. Alternatively, (3.9) measures the *dominance* of the degree-d term in the polynomial expansion: intuitively, when $c_d(\alpha - \alpha_j)^d$ is the largest term our approximation, it is almost certain that the approximation is wrong. By using (3.9) as the termination criterion of Alg. 4, the algorithm progresses until the expansion point α_j is sufficiently close to α^* such the $\mathcal{O}\big((\alpha - \alpha^*)^d\big)$ error metric is small.[1]

[1]The shrewd or incredulous reader may be chewing several questions at this point, e.g. Q_1: What about the Wolfe conditions—don't we need those and the Zoutendijk theorem? and Q_2: How you even know this converges? and Q_3: Do you have a feasible proof outline? A_1: While our implementation of PELS for the convex logistic regression loss function uses (3.9) as a convergence criterion and hence forgoes the convergence properties that the Wolfe conditions bring, it is actually both possible and simple to use the Wolfe conditions in the PELS method, since conditions (3.3) and (3.4) may be evaluated using $W(\alpha)|_{\alpha_j}$ either *exactly* at α_j or with an approximation error in a neighbourhood about α_j. In a combined manner, the $\epsilon(\alpha)|_{\alpha_j}$ error metric may best serve as a threshold

Algorithm 4: Polynomial Expansion Line Search

Input: $\mathbf{w} \in \mathbb{R}^m, \mathbf{p} \in \mathbb{R}^m, \alpha_0 > 0, \theta > 0$

Output: $\alpha_j \approx \alpha^* = \arg\min_\alpha \mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$

1. $j \leftarrow 0$
2. **repeat**
3. $\quad \mathbf{c}_j \leftarrow$ CALC_COEFFS$(\mathbf{w}, \mathbf{p}, \alpha_j)$
4. $\quad \alpha_{j+1} \leftarrow \arg\min_\alpha W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}$
5. $\quad \epsilon(\alpha_{j+1})|_{\alpha_j} \leftarrow$ approximate $\varepsilon_{d+1}(\alpha_{j+1})|_{\alpha_j}$
6. $\quad j \leftarrow j + 1$
7. **until** $\epsilon(\alpha_j)|_{\alpha_{j-1}} \leq \theta$
8. **return** α_j

Procedure CALC_COEFFS$(\mathbf{w}, \mathbf{p}, \alpha_0)$

1. $\quad \mathbf{r} \leftarrow \mathbf{w} + \alpha_0\mathbf{p}$
2. $\quad$ Broadcast $\mathbf{r}, \mathbf{p}$ to n_p compute nodes
3. $\quad$ **for** compute node $t \in \{1, \ldots, n_p\}$ **do**
4. $\quad\quad$ **for** $\ell \in \{0, \ldots, d\}$ **do**
5. $\quad\quad\quad c_\ell^{[t]} \leftarrow \sum_{\text{local } i_l} F_\ell(\mathbf{r}, \mathbf{p}; \mathbf{x}_{i_l}, y_{i_l})$
6. $\quad \mathbf{c}_\lambda \leftarrow \frac{\lambda}{2}\left[\|\mathbf{r}\|_2^2,\ 2\mathbf{r}^T\mathbf{p},\ \|\mathbf{p}\|_2^2, 0, 0, \ldots\right]^T$
7. $\quad \mathbf{c} \leftarrow \mathbf{c}_\lambda + \bigoplus_{t=1}^{n_p} \mathbf{c}^{[t]}$
8. $\quad$ **return** $\mathbf{c}$

---

for estimating a trust region in which the Wolfe conditions, if they hold according to $W(\alpha)$ and $W'(\alpha)$, would be believed such that the convergence guarantees of Zoutendijk-type proofs for optimization algorithms hold. A_2: The basic mechanism of Newton's method is to construct and minimize a degree-2 model function at α_j in each iteration. Increasing to degree-3 leads to Halley's method, and increasing the degree further then leads to the family of higher-order Householder methods. Newton's method is globally convergent on convex functions with positive definite Hessians [85], and it seems unreasonable that increasing the degree of the model (and thereby improving the accuracy) could alter the convergence; so for any problem for which Newton-Raphson iteration converges, the same would be expected of this formulation of PELS. A_3: A standard

When $\mathcal{L}(\mathbf{w})$ is itself polynomial, such as for least squares regression and low-rank matrix factorization as formulated by e.g. Gemulla et al. [42], then $\varepsilon_{d+1}(\alpha) = 0$ for sufficiently large d, and Alg. 4 can solve (3.1) *exactly* in a single step without further coefficient computations. For the more general case of analytic $\mathcal{L}(\mathbf{w})$, the advantages of Alg. 4 over Alg. 3 are twofold. Firstly, the communication costs are lesser in the distributed operations: the summation in (3.7) to compute the coefficients $\{c_\ell\}$ requires communicating $d+1$ *scalar* values for each $(\mathbf{x}_i, y_i)$, whereas the distributed computation of $\nabla\mathcal{L}$ in Alg. 3 communicates the m-dimensional gradient vectors $\{\nabla f(\mathbf{w}; \mathbf{x}_i, y_i)\}$. The second—and most important—reason is that, unlike standard line searches that compute values for $\mathcal{L}$ and $\nabla\mathcal{L}$ for only a *single* α_j, the distributed operation computing the $\{c_\ell\}$ in PELS produces a model which is valid for an entire neighbourhood of α_j; thus, if α_j is close to α^*, the PELS method can compute a very accurate approximation to α^* with only a single $\mathcal{O}(n)$ pass over the dataset to evaluate the polynomial coefficients. This is illustrated in Fig. 3.2, where both Alg. 3 and Alg. 4 have been applied to the function $\phi(\alpha) = \alpha\, e^{\alpha} + e^{-(\alpha-4)}$. In Fig. 3.2a, iterates produced by Alg. 3 (implemented as in [80] with $\nu_1 = 10^{-4}$ and $\nu_2 = 0.9$) are shown, however only a single iterate has been generated since (3.3) and (3.4) are satisfied at the input $\alpha_0 = 1$; as such, the algorithm terminates with a relatively inaccurate minimum, rather than compute $\nabla\mathcal{L}$ with an $\mathcal{O}(n)$ pass over the data for another step α_1. On the other hand, Fig. 3.2b shows PELS with a degree-3 polynomial approximation $W|_{\alpha_0}$ to $\phi(\alpha)$ for the same

---

proof technique for the convergence of Newton and other fixed point methods is to use the analytic formula for φ in a fixed point iteration $\alpha_{j+1} = \alpha_j - \varphi(\alpha_j)$, such that inequalities may be used to bound the term $|\alpha_j - \alpha^*|$ for each iterate in the sequence generated by φ. The problem induced by an arbitrary degree-d polynomial is that for $d > 4$, there are no *closed form* expressions for its roots (cf. Abel's impossibility theorem), and even the quartic formula is pretty hairy. Hence there is no analytical formula for the minima of an arbitrary degree-d polynomial [since the roots of a degree-d polynomial are the minima of the appropriate degree-$(d+1)$ polynomial] and thus no fixed point φ function to construct and grind through standard fixed point techniques. At current we are unaware of a standard method of analysis, and welcome suggestions.

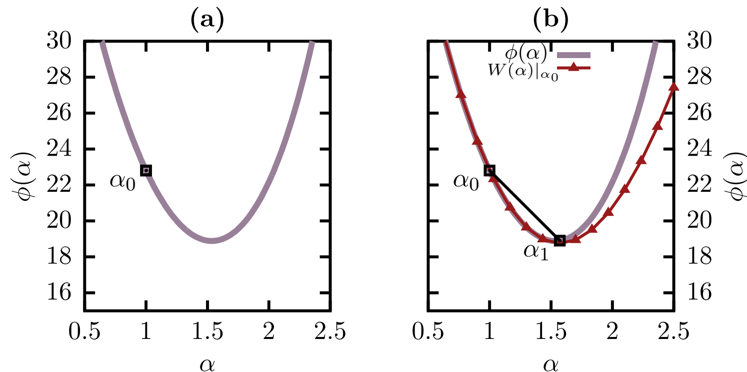


Figure 3.2: *Example iterates $\{\alpha_j\}$ produced by **(a)** Alg. 3 with $\nu_1 = 10^{-4}$ and $\nu_2 = 0.9$ (implemented as described by Moré & Thuente [80]) and **(b)** PELS with expansions using $d = 3$ for $\phi(\alpha) = \alpha\, e^{\alpha} + e^{-(\alpha-4)}$.*

$\alpha_0 = 1$. Here, α_1 is determined as the solution to (3.8), and is visibly more accurate than α_0 in Fig. 3.2a (only $d = 3$ is shown since $W|_{\alpha_0}$ could not be distinguished from $\phi(\alpha)$ at this scale for larger degrees). This example is not a toy problem, but a case that we have often observed experimentally for the LBFGS algorithm: though the initial step size $\alpha_0 = 1$ frequently obeys the Wolfe conditions and is accepted in Alg. 3, α_0 is an inaccurate solution to (3.1).

3.3 PELS Implementation In Spark

The PELS method in Alg. 4 was implemented in Apache Spark 1.5 and tested using an L_2-regularized logistic regression model for binary classification of a continuous vector $\mathbf{x}_i \in \mathbb{R}^m$ by the discrete label $y_i \in \{0, 1\}$. In implementing PELS, we mimicked the structure of the algorithms implemented within Spark 1.5's optimization library. The method assumes that the arrays storing the parameter vector $\mathbf{w}$, gradient $\nabla\mathcal{L}$, and associated search direction vector $\mathbf{p}$ fit into memory on the master node, which drives the outer optimization routines. The evaluation of $\mathcal{L}$ and $\nabla\mathcal{L}$ were farmed out to the compute nodes, as illustrated by Fig. 2.4 in §2. Analogously, the

PELS coefficient computations were evaluated through map and reduce operations, and the local minimization in (3.8) occurred on the master node.

The logistic loss function is derived from the maximum log likelihood of the logistic model probability $\Pr[y_i = 0|\mathbf{x}_i, \mathbf{w}]$ as [52]

$$\mathcal{L}(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{i=1}^{n}\Big(\log[e^{-\mathbf{w}^T\mathbf{x}_i} + 1] + \mathbb{N}(y_i)\mathbf{w}^T\mathbf{x}_i\Big) \tag{3.10}$$

where $\mathbb{N}(y_i) = 1 - \mathbb{I}(y_i)$ and $\mathbb{I}(y_i)$ is a boolean indicator function equal to 1 only if $y_i \neq 0$. It can be shown that (3.10) is strictly convex and has a unique global minimizer [66]. Since (3.10) is infinitely differentiable, we may use a Taylor expansion about α_0 for $\phi(\alpha) = \mathcal{L}(\mathbf{w} + \alpha\mathbf{p})$. Denoting the ray $\mathbf{w} + \alpha_0\mathbf{p}$ by $\mathbf{r}$, and using the simplifications $p_i = \mathbf{p}^T\mathbf{x}_i$ and $r_i = e^{-\mathbf{r}^T\mathbf{x}_i}$, the expansion up to 4th order is

$$\begin{aligned}
\phi(\alpha) = &\left[\frac{1}{n}\sum_{i=1}^{n}\big(\log[r_i + 1] + \mathbb{N}(y_i)\,\mathbf{r}^T\mathbf{x}_i\big) + \frac{\lambda}{2}\|\mathbf{r}\|_2^2\right] \\
&+ (\alpha - \alpha_0)\left[\frac{1}{n}\sum_{i=1}^{n} p_i\left(\mathbb{N}(y_i) - \frac{r_i}{r_i + 1}\right) + \lambda\mathbf{p}^T\mathbf{r}\right] \\
&+ (\alpha - \alpha_0)^2\left[\frac{1}{n}\sum_{i=1}^{n}\frac{p_i^2 r_i}{2(r_i + 1)^2} + \frac{\lambda}{2}\|\mathbf{p}\|_2^2\right] \\
&+ (\alpha - \alpha_0)^3\left[\frac{1}{n}\sum_{i=1}^{n}\frac{p_i^3 r_i\,(r_i - 1)}{6\,(r_i + 1)^3}\right] \\
&+ \mathcal{O}\big((\alpha - \alpha_0)^4\big).
\end{aligned} \tag{3.11}$$

Note that the functions $\{F_\ell(\mathbf{r}_k, \mathbf{p}_k; \mathbf{x}_i, y_i)\}_{\ell=0}^{d}$ are the coefficients of the $(\alpha - \alpha_0)$ terms in (3.11), and also that due to the structure of the loss function (i.e. the recursion of gradients of the exponential e^x), the high order coefficients in (3.11) depend only on the scalars p_i and r_i and thus do not have a significant computational cost beyond the initial calculation of p_i and r_i.

The observations $\mathcal{R} = \{(\mathbf{x}_i, y_i)\}$ were stored in an RDD with type `RDD` $[(\mathbb{R}^m, \mathbb{R})]$, where the vectors $\{\mathbf{x}_i\}$ were either sparse or dense vectors with double precision. The parameter and search di-

rection vectors $\{\mathbf{w}_k\}$ and $\{\mathbf{p}_k\}$ were not partitioned over the compute nodes, but stored as dense vector objects on the master node. Communicating the vectors $\mathbf{p}_k$ and $\mathbf{r}_k = \mathbf{w}_k + \alpha_j \mathbf{p}_k$ to the compute nodes in the cluster for any α_j in the line search was performed via a torrent broadcast [109]. To compute the coefficient vector $\mathbf{c}_j$ in any PELS iteration, the functions $\{F_\ell(\mathbf{r}_k, \mathbf{p}_k; \mathbf{x}_i, y_i)\}_{\ell=0}^{d}$ in (3.11) were calculated as parallel map operations on the RDD of $\mathcal{R}$, and the resulting $\mathbf{c}_j \in \mathbb{R}^{d+1}$ was summed via a multi-level reduce operation (recall Fig. 2.3b), where the operator $\oplus$ was vector addition.[2] The L_2 regularization terms were computed by the master node as the vector $\mathbf{c}_\lambda = \frac{\lambda}{2}[\|\mathbf{r}_k\|_2^2, 2\mathbf{r}_k^T\mathbf{p}_k, \|\mathbf{p}_k\|_2^2, 0, 0, \ldots]$ (with zeros appended so $\mathbf{c}_\lambda \in \mathbb{R}^{d+1}$) and added to $\mathbf{c}_j$. These steps are enumerated in procedure COMPUTE_COEFFS in Alg. 4 in which the local, *map-side* reduction (Fig. 2.4c) of coefficient vectors occurs in step 5, and the global reduction to the master node occurs in step 7.

In our implementation of PELS, we used the NR method as the optimization routine for minimizing the polynomial $W|_{\alpha_j}$ in step 4 of Alg. 4, which is equivalent to solving for the roots of $W'|_{\alpha_j}$. However, it is possible that $W'|_{\alpha_j} < 0$ for $\alpha > 0$ if α_j is far from α^* (i.e. $W'|_{\alpha_j}$ has no relevant real roots). In this case, instead of using a minimization routine, α_{j+1} was determined by NR iteration at α_j as

$$\alpha^{NR} = \alpha_j - \frac{\phi'(\alpha_j)}{\phi''(\alpha_j)} = \alpha_j - \frac{c_1}{2c_2}, \tag{3.12}$$

where only the coefficients c_1 and c_2 appear since $\phi(\alpha_j) = W(\alpha_j)|_{\alpha_j}$. Since it was observed that NR iteration converged within machine precision to a stationary point of $W|_{\alpha_j}$ in very few iterations, (3.12) was used as a default whenever the NR method starting from α_j failed to compute a zero gradient within a tolerance of

[2] To be precise for Spark aficionados, the map and reduce operations were performed using a multi-level `treeAggregate`, which operates more efficiently by using two functions, one that specifies a transformation $f_{\text{map}} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^{d+1}$ to create the Taylor polynomial coefficients for a single instance $(\mathbf{x}_i, y_i)$ in the dataset, and another $f_{\text{comb}} : \mathbb{R}^{d+1} \times \mathbb{R}^{d+1} \to \mathbb{R}^{d+1}$ that sums the coefficients for two distinct instances. The aggregation reduces the required memory usage by creating a minimal number of objects to be managed by the JVM [since the second *combiner* function operates on the objects in-place].

10^{-15} in fewer than 10 iterations.[3]

3.4 PELS Performance Tests

To evaluate the efficacy of PELS, performance tests were conducted in which the Gradient Descent (GD), NCG, and LBFGS algorithms were used to optimize the parameters of a logistic regression model. These algorithms each were run in two sets, using either PELS or a standard WA cubic interpolating line search [4, 85] from the open-source Breeze library [50] with $\nu_1 = 10^{-4}$ and $\nu_2 = 0.9$, which is used in Spark's native LBFGS implementation. The implementations using PELS will be henceforth denoted with a suffix *-P* as LBFGS-P, NCG-P, and GD-P. All algorithms using a WA line search were implemented such that the operations other than the line search were *identical* to those in the LBFGS-P, NCG-P and GD-P implementations, respectively, except where noted.[4]

[3] Just in case you were wondering whether this statement contradicts the NR convergence argument for convex functions in footnote 1, remember that the function being minimized by the NR routine is $W(\alpha)|_{\alpha_j}$ and *not* $\phi(\alpha)$. In reality, the NR iterate in (3.12) is the initial step computed in *every* invocation of our NR minimization routine solving (3.8) in PELS [since $W(\alpha_j)|_{\alpha_j} = c_0$, $W'(\alpha_j)|_{\alpha_j} = c_1$, etc], but the full NR procedure may diverge because $W(\alpha^{NR})|_{\alpha_0} \neq \phi(\alpha^{NR})$, and there aren't guarantees about the local boundedness of the gradient or Hessian of the approximating Taylor polynomial once you leave the sweet spot around α_j. Naturally it's possible to use an alternative minimizaton routine to (3.12)—one that minimizes $W(\alpha)|_{\alpha_j}$ on a *bounded* interval, with the bounds prudently chosen based on a maximal value of (3.9) above which solutions are disallowed [which is necessary unless you'd hazard accepting $-\infty$ as your minimum]. However you'll have to wait until §5 in which nonconvex functions are tackled for that trick; here it was sufficient to use (3.12) for the odd time that NR iteration diverged.

[4] The LBFGS implementation we are considering is *not* the native Spark LBFGS code. We found that it had significant overheads stemming—it seems—from the deeply nested data structures in Breeze. It was possible to re-implement the algorithm in an imperative style that took only circa 65% as much time as the Spark code on large problems. The issue with the native implementation, so far as can be gleaned from the code, is both the many layers of OOP required to compute the new

For both LBFGS and LBFGS-P, the initial step size in each invocation (i.e. the input α_0 in Alg. 3 and Alg. 4) was taken to be 1 for both the WA line search in LBFGS and PELS in LBFGS-P, which is the recommended trial step for the LBFGS algorithm [70]. A history of $n_c = 5$ corrections was used in our tests, as recommended for large problems [70], and the initial inverse Hessian approximation in each iteration used Liu and Nocedal's M3 scaling [70]. This is the same scaling formula used in the LBFGS code for Spark 1.5. The NCG-P update rule was the positive Polak-Ribière formula in (1.5), and the NCG and GD algorithms with and without PELS both used the scaling formula in (1.6) to determine the initial step size in their respective univariate line searches. For NCG-P and GD-P, the search directions used to compute the coefficients in PELS and subsequently update $\mathbf{w}_k$ were normalized as $\mathbf{p}_k \leftarrow \mathbf{p}_k/\|\mathbf{p}_k\|_2$. Finally, the NCG-P algorithm was run with a Powell restart condition of $\gamma = 0.2$ in (1.7), however NCG was given a restart threshold of $\gamma = 1$ because smaller values often triggered restarts in *every* iteration, reverting the method to GD and yielding poor performance.

The logistic regression models were trained on binary classification datasets procured from the LIBSVM repository [23], which are listed in in Tab. 5.1. This table also gives summary information about the datasets' respective sizes of n and m, mean number of nonzero elements in $\mathbf{x}_i$, ratio of the number of true (nonzero) y_i labels to false y_i labels as $n_+ : n_-$, and magnitudes of λ used in (3.10). The dense $\{\mathbf{x}_i\}$ in the Epsilon dataset were represented by contiguous dense vectors, while the other datasets' instances were stored in compressed sparse vector format. All $\mathbf{x}_i$ were further augmented as $\mathbf{x}_i^T \leftarrow [\mathbf{x}_i^T \; 1]^T$ to implicitly include a constant offset term in the inner product $\mathbf{w}^T\mathbf{x}_i$. For all datasets, each algorithm

$\mathbf{p}_k$ as well as the way in which the vectors $\{\mathbf{g}_{k+1} - \mathbf{g}_k\}$ and $\{\mathbf{w}_{k+1} - \mathbf{w}_k\}$ are stored for the quasi-Newton update. For the latter case, it seems that in Breeze a new sequence object of arrays is created from the previous one in each iteration by stripping the vector from the sequence's head and concatenating the most recent vector to the sequence's tail. For large m, it may be more efficient to create a statically-sized array for these vectors and in each iteration overwrite one entry at index $i^* = \text{mod}(k, n_c)$ of the array, such that the modular arithmetic takes care of all the heavy lifting in the LBFGS two-loop recursion procedure.

Table 3.1: *Properties of LIBSVM classification datasets used in numerical experiments, and respective magnitudes of λ.*

Dataset	n	m	$\text{nnz}(\mathbf{x}_i)$	$n_+ : n_-$	λ
EPSILON	400,000	2,001	2001	1.0 : 1.0	10^{-8}
RCV1 (test)	677,399	47,237	74 ± 54	1.1 : 1.0	10^{-7}
URL	2,396,130	3,231,962	117 ± 17	1.0 : 2.0	10^{-8}
KDD-A	8,407,752	20,216,831	37 ± 9	5.8 : 1.0	10^{-9}
KDD-B	19,264,097	29,890,095	29 ± 8	6.2 : 1.0	10^{-9}

was run with $\mathbf{w}_0 = \mathbf{0}$. In each iteration, computed values of $\mathcal{L}(\mathbf{w}_k)$ and $\|\nabla\mathcal{L}(\mathbf{w}_k)\|_2$ were written to the standard output filestream. The values of θ used in Alg. 4 were $\theta = 10^{-4}$ (approximately 0.01% error), except for the EPSILON and KDD-A datasets, on which 10^{-6} was used. While smaller θ generated more accurate step sizes, the additional PELS iterations increased the computational time; values of $\theta \in [10^{-6}, 10^{-4}]$ were a good compromise between accuracy and speed.

All performance tests were performed on a computing cluster composed of 16 homogeneous compute nodes, 1 storage node hosting a network filesystem, and 1 master node. The nodes were interconnected by a 10 Gb ethernet managed switch (PowerConnect 8164; Dell). Each compute node was a 64 bit rack server (PowerEdge R620; Dell) running Linux kernel 3.13 with two 8-core 2.60 GHz processors (Xeon E5-2670; Intel) and 200 GB of SDRAM. The master node had identical processor specifications and 512 GB of RAM. Compute nodes were equipped with six ext4-formatted 600 GB SCSI hard disks, each with 10,000 RPM nominal speed. The storage node (PowerEdge R720; Dell) contained two 6-core 2 GHz processors (Xeon E5-2620; Intel), 64 GB of memory, and a hard drive speed of 7,200 RPM. Further information about the cluster hardware specifications is available in §A.

Our Apache Spark assembly was built from a snapshot of the version 1.5 master branch using Oracle's Java 7 distribution, Scala 2.10, and Hadoop 2.0.2. Input files to Spark programs were stored on the storage node in plain text. The compute nodes' local SCSI drives were used for both Spark spilling directories and Java temporary files. Shuffle files were consolidated into larger files, as recom-

Table 3.2: *Default settings for Spark configuration parameters.*

Property Key	Value	Units
spark.kryoserializer.buffer.max	1024	MB
spark.driver.maxResultSize	∞	MB
spark.storage.memoryFraction	0.5	
spark.shuffle.spill	false	
spark.shuffle.file.buffer	256	kB
spark.shuffle.memoryFraction	0.4	
spark.reduce.maxMbInFlight	256	MB
spark.broadcast.compress	true	
spark.akka.threads	32	

mended for ext4 filesystems [29], and Kryo serialization was used. In our experiments, the Spark driver was executed on the master node in standalone client mode, and a single instance of a Spark executor was created on each compute node. All RDDs had 256 partitions, corresponding to 1 partition per available physical core; performance decreased in general as more partitions were used. Finally, n_l was set to $\log_2 16$ in all tree aggregations; empirically, this was faster for large datasets than the default of $n_l = 2$. All other relevant Spark configuration settings are listed in Tab. 3.2.[5]

3.5 Results & Discussion

Our performance results are presented in two parts. In our initial tests, we are interested purely in the convergence improvements possible with PELS, and hence consider the Epsilon and RCV1

[5]Spark practitioners interested specifically in these details should be aware that they reflect some bounds that may be permissible only with our particular cluster hardware; for instance, disallowing shuffle spill could [imaginably] be catastrophic without the oodles of RAM on our compute nodes. Setting the spark.driver.maxResultSize parameter to ∞ was also important for performing large reductions on RDDs to the driver program, and it caused some headaches before being made infinite. Additionally, the use of Kryo serialization is highly recommended since it significantly accelerated Spark runtimes for all of the experiments we have considered, sometimes by 25–50%.

datasets that have little noise and are well-posed with $m \ll n$. In these tests, we present high-accuracy convergence traces since $\mathcal{L}(\mathbf{w}^*)$ may be computed to machine precision. By contrast, real-life machine learning applications are often ill-posed and have statistical errors in the model or instances that exceed the numerical optimization error, such that $\mathbf{w}^*$ is computed to only a few significant digits until the training loss ceases to decrease appreciably [16]. To demonstrate that PELS is effective in this practical setting as well, we present results on the large and ill-posed URL, KDD-A, and KDD-B datasets, and compute the acceleration in reaching terminal values of the training label prediction accuracy, $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$, achievable by using PELS.

For the Epsilon and RCV1 datasets, respectively, Fig. 3.3 and Fig. 3.4 show the traces of $|\mathcal{L}(\mathbf{w}_k) - \mathcal{L}(\mathbf{w}^*)|$ as a function of **(a)** iterations and **(b)** clock time for the algorithms considered in §3.4. In both plots, $\mathcal{L}(\mathbf{w}^*)$ was determined as the minimal loss computed by any algorithm within the maximum number of iterations. That $\mathbf{w}^*$ was computed accurately is evinced by the gradient norm at $\mathbf{w}^*$: $\|\nabla\mathcal{L}(\mathbf{w}^*)\|_2$ was 1.3×10^{-11} for the Epsilon dataset and 8.7×10^{-12} for the RCV1 dataset, as computed by NCG-P. It is notable in both plots that NCG-P has drastically outperformed both the NCG and LBFGS algorithms that use a standard WA line search. LBFGS-P outperformed LBFGS in iterations and time for the Epsilon dataset, and performed similarly to LBFGS on the RCV1 dataset. GD-P and GD are virtually indistinguishable at the scale of Fig. 3.3 and Fig. 3.4 since there is little substantive difference in the two algorithms' traces.

The convergence traces for the large URL, KDD-A, and KDD-B datasets are shown in Fig. 3.5, Fig. 3.6, and Fig. 3.7, respectively (traces for GD and GD-P have been omitted from these plots since these algorithms made little progress towards the solution). These datasets required considerably more iterations and training time, and the LBFGS-P and NCG-P algorithms have outperformed their counterparts by multiple decimal digits in accuracy. However, since in many machine learning applications, the training procedure is halted once $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$ reaches a plateau, only $\log_{10}|\mathcal{L}(\mathbf{w}_k) - \mathcal{L}(\mathbf{w}^*)| \approx -3$ may be necessary in practice. Bearing this, the speedup factors as a function of training accuracy for LBFGS-P relative to LBFGS were computed for the URL and

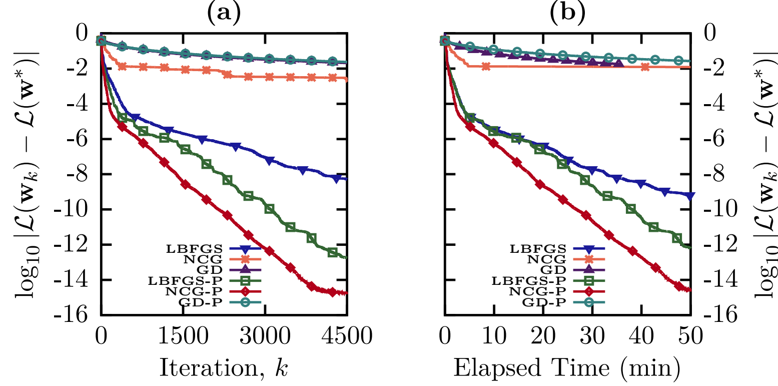


Figure 3.3: *Convergence traces in regularized loss for the* Epsilon *dataset in **(a)** iterations and **(b)** elapsed time.*

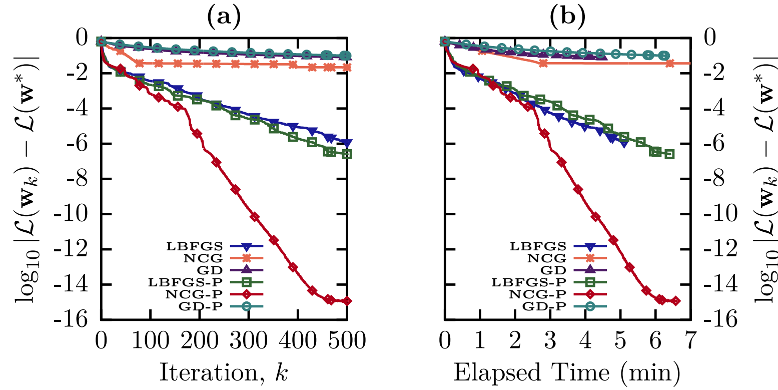


Figure 3.4: *Convergence traces in regularized loss for the* RCV1 *dataset in **(a)** iterations and **(b)** elapsed time.*

both KDD datasets, and are shown in Fig. 3.8, where the factors in Fig. 3.8a have been computed for the number of iterations required, and the factors in Fig. 3.8b have been computed for the amount of clock time required. These speedup factors were determined by finding the first iterate produced by LBFGS-P that reached the same or greater value of $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$ as LBFGS, and the error bars in this plot show the standard deviation about the mean speedup ratios computed in non-overlapping windows of

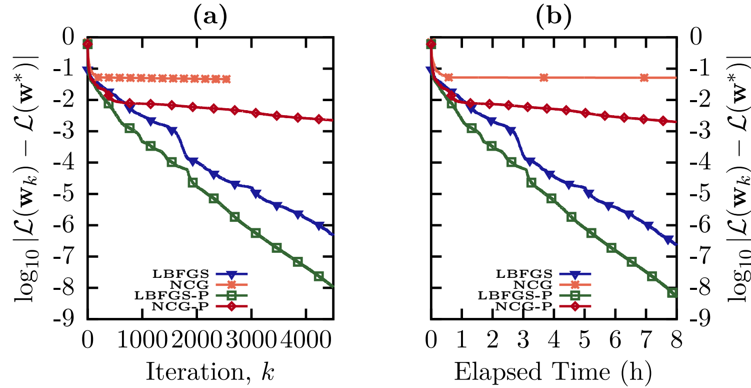


Figure 3.5: *Convergence traces in regularized loss for the* URL *dataset in **(a)** iterations and **(b)** elapsed time.*

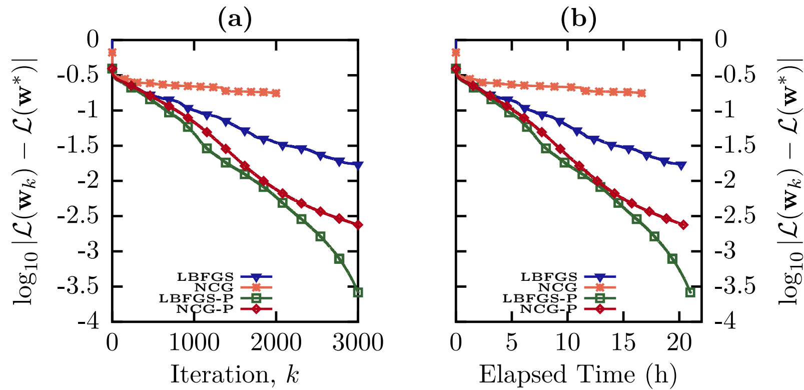


Figure 3.6: *Convergence traces in regularized loss for the* KDD-A *dataset in **(a)** iterations and **(b)** elapsed time.*

width 0.02 along the x-axis. To reach the terminal training accuracies (e.g. ∼97% for URL), Fig. 3.8 shows that factors of 1.8–2 in both iterations and clock time are achievable on the KDD-A, KDD-B, and URL problems.

To complement the convergence traces and explain the speedup in clock time of the PELS method in a distributed setting, Tab. 3.3 presents timing measurements for the distributed operations, averaged for each algorithm over all iterations. The quantity τ_{fg} repre-

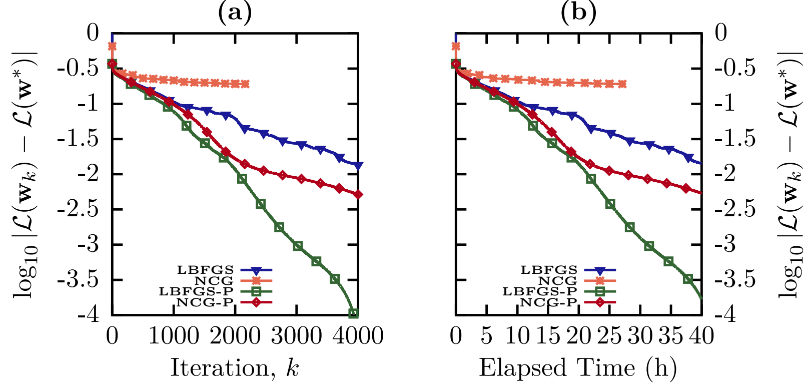


Figure 3.7: *Convergence traces in regularized loss for the* KDD-B *dataset in **(a)** iterations and **(b)** elapsed time.*

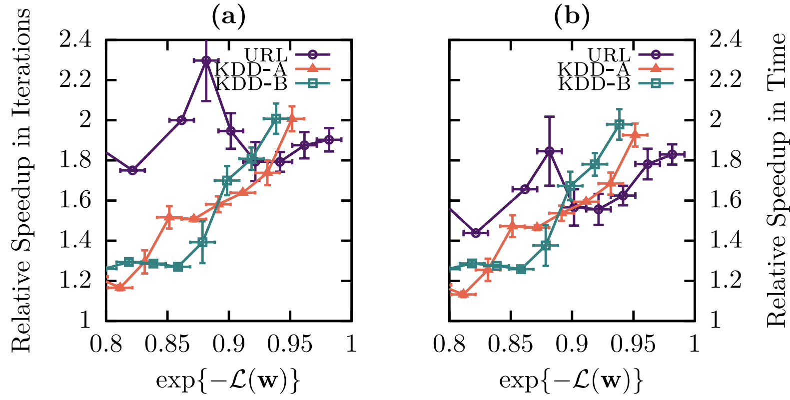


Figure 3.8: *Speedup factors of LBFGS-P over LBFGS as a function of approximate training accuracy, computed for* **(a)** *iterations and* **(b)** *time.*

sents the mean clock time required to evaluate $\mathcal{L}$ and $\nabla\mathcal{L}$ through a map operation on the RDD of $\mathcal{R}$ with a subsequent aggregation. However, since $\mathcal{L}$ does not need to be computed explicitly in the PELS method, τ_{fg} denotes the time required to compute *only* $\nabla\mathcal{L}$ for LBFGS-P, NCG-P, and GD-P (although evaluating $\nabla\mathcal{L}$ alone takes as much time as evaluating $\mathcal{L}$ and $\nabla\mathcal{L}$ simultaneously). For the algorithms using the PELS method, $\tau_{\ell\text{s}}$ gives the mean time to

both compute and aggregate the coefficient vector $\mathbf{c}_j$. The quantity n_e represents the average number of iterations per line search in the respective manners in which they were conducted: for the WA algorithms, n_e is the mean number of function/gradient evaluations per line search, and for the PELS algorithms, it represents the mean number of coefficient evaluations performed in Alg. 4 per line search. Thus, in each outer iteration of the optimization method, PELS algorithms perform 1 gradient computation taking τ_{fg} seconds and then n_e operations lasting $\tau_{\ell\mathrm{s}}$ seconds, while WA algorithms perform n_e operations taking τ_{fg} seconds. The total time per iteration is shown as τ_{tot}. All uncertainty bounds show the standard deviation about the mean value; no uncertainty is given for n_e since it was taken as the ratio of the total number of line search calls to outer iterations.

The advantage of PELS for accurate large-scale distributed line searches is apparent when comparing $\tau_{\ell\mathrm{s}}$ to τ_{fg} in Tab. 3.3, as well as the difference in n_e between PELS and WA algorithms. For all datasets, $\tau_{\ell\mathrm{s}} < \tau_{\mathrm{fg}}$, however when the problem size is large, such as for the KDD-A and KDD-B datasets, computing the PELS coefficients took only a quarter of the time required to compute $\nabla\mathcal{L}$. For all problems, LBFGS-P and NCG-P required a lower number of n_e evaluations in the line search than LBFGS and NCG, respectively, and in addition $n_e \approx 1$ for all PELS algorithms on the datasets considered. NCG-P is particularly effective when contrasted with NCG on the Epsilon, RCV1, and URL datasets: on these problems, NCG produced poorly scaled search directions requiring many line search iterations. While preconditioning in the NCG algorithm can be used to mitigate the poor scaling, it requires additional matrix-vector operations in each iteration, often constructing an approximation to the Hessian with LBFGS-type updates [49]; we thus find it notable that NCG-P often had better performance than LBFGS, *without* the need for preconditioning. In contrast to NCG, LBFGS generally had $n_e \approx 1$ in Tab. 3.3; as such, the performance gains of LBFGS-P over LBFGS stem principally from reducing the total number of required iterations by computing more accurate minima in each line search invocation.

In examining the average τ_{tot} for LBFGS on the large problems, it can be seen to be approximately the same for both PELS and WA implementations, however the standard deviation is far

Table 3.3: *Mean clock times for each iteration (τ_{tot}), evaluating $\mathcal{L}$ and $\nabla\mathcal{L}$ (τ_{fg}), and computing $\mathbf{c}_j$ ($\tau_{\ell s}$). The average number of function calls or line search iterations per outer iteration of the optimization algorithms are given by n_e.*

	Alg.	τ_{tot} (s)	τ_{fg} (s)	$\tau_{\ell\text{s}}$ (s)	n_e
EPSILON					
	LBFGS	0.5 ± 0.2	0.42 ± 0.07		1.15
	NCG	4 ± 2	0.43 ± 0.08		9.92
	GD	0.42 ± 0.10	0.42 ± 0.08		1.01
	LBFGS-P	0.7 ± 0.1	0.42 ± 0.07	0.32 ± 0.06	1.00
	NCG-P	0.7 ± 0.1	0.42 ± 0.07	0.32 ± 0.06	1.00
	GD-P	0.7 ± 0.1	0.42 ± 0.07	0.32 ± 0.06	1.00
RCV1					
	LBFGS	0.6 ± 0.3	0.47 ± 0.08		1.24
	NCG	5 ± 2	0.44 ± 0.07		10.48
	GD	0.5 ± 0.2	0.5 ± 0.1		1.06
	LBFGS-P	0.8 ± 0.2	0.45 ± 0.07	0.29 ± 0.06	1.05
	NCG-P	0.8 ± 0.2	0.46 ± 0.07	0.30 ± 0.06	1.08
	GD-P	0.8 ± 0.1	0.45 ± 0.07	0.29 ± 0.06	1.00
URL					
	LBFGS	6 ± 2	4.5 ± 0.2		1.14
	NCG	56 ± 15	4.6 ± 0.3		11.10
	LBFGS-P	6.1 ± 0.3	4.3 ± 0.2	1.0 ± 0.1	1.00
	NCG-P	5.9 ± 0.3	4.4 ± 0.2	1.0 ± 0.1	1.01
KDD-A					
	LBFGS	24 ± 5	19 ± 1		1.05
	NCG	30 ± 18	19 ± 1		1.38
	LBFGS-P	25 ± 2	18 ± 1	4.6 ± 0.4	1.03
	NCG-P	24 ± 2	18 ± 1	4.7 ± 0.6	1.03
KDD-B					
	LBFGS	37 ± 8	28 ± 2		1.07
	NCG	45 ± 28	28 ± 2		1.41
	LBFGS-P	38 ± 2	27 ± 2	6.7 ± 0.6	1.00
	NCG-P	36 ± 3	27 ± 2	7.1 ± 0.8	1.00

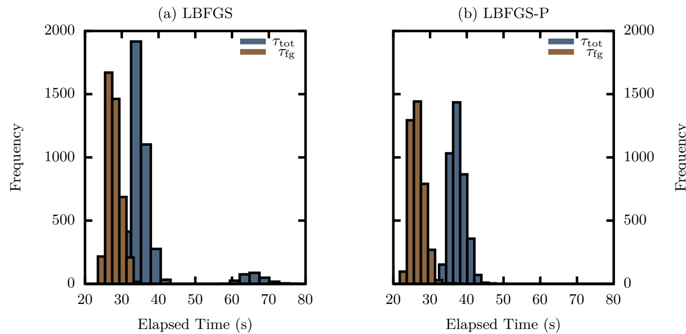
larger for LBFGS-P and there is more time unaccounted for in the values of the difference $\tau_{\text{tot}} - n_e \cdot \tau_{\text{fg}}$ for LBFGS than for LBFGS-P. Using the KDD-B dataset for instance, the difference is 4 ± 3 s for the LBFGS-P algorithm, but 7 ± 8 s for LBFGS.[6] Since the

[6] The absurdly large standard error here is propagated through quadrature of the [unrounded] values for standard deviation about the mean for τ_{tot} and τ_{fg} in Tab. 3.3 and afterwards rounded from 6.9397 ± 8.1553 to a single significant digit. Of course, such a large uncertainty begets questions—e.g. wherefore the large standard deviation in τ_{tot} for LBFGS in Tab. 3.3. With respect to the KDD-B dataset, the histograms below show the distribution of the τ_{tot} and τ_{fg} measurements

algorithm implementations differ only in the line search, the simplest conjecture is that there is a large amount of variability in the time required for some [yet unknown] component of the Breeze WA line search. If that variability could be removed by altering or re-implementing the Breeze code, it would not be unreasonable to expect a decrease of 3 seconds in the average time per iteration of LBFGS (i.e. a decrease at least equal to the difference in the *extra* times of the implementations of LBFGS-P and LBFGS, since their codes were otherwise identical). This would reduce the time taken for the KDD-B dataset by a factor of approximately 92%, with an accompanying reduction in the temporal speedup factors in Fig. 3.8b. While this does not dramatically alter the results presented here, it bears mention that further optimizations are possible.[7]

---

for LBFGS in panel (a) and LBFGS-P in panel (b), such that bimodality of τ_{tot} is starkly visible in (a). The right mode for LBFGS stems from the 260/4000 iterations that required 2 gradient evaluations; without those values, the standard deviation about the mean for the τ_{tot} of LBFGS was 2.0. The absence of a bimodal distribution for LBFGS-P is specific to the KDD-B dataset, since we observed $n_e = 1.00$.



[7]In fact, an obvious and straightforward improvement lies in wait in step 2 of procedure CALC_COEFFS of Alg. 4. As $\mathbf{w}, \mathbf{p}$ are *fixed* for the entirety of the line search, it is necessary to broadcast these vectors to the compute nodes only *once*. The required modification to the code is not particularly substantive, but requires that some closures [specific jargon for computer code that is serialized at program runtime and transmitted by the master node to the compute nodes to provide instructions for their map operations on local RDD partitions] in PELS line search be tweaked so that the broadcast variables remain in scope for subsequent

3.6 Conclusion

In this chapter, we have presented the Polynomial Expansion Line Search method for large-scale batch and minibatch optimization algorithms, applicable to smooth loss functions with L_2-regularization such as least squares regression, logistic regression, and low-rank matrix factorization. The PELS method constructs a truncated Taylor polynomial expansion of the loss function that may be minimized quickly and accurately in a neighbourhood of the expansion point, and additionally has coefficients that may be evaluated in parallel with little communication overhead. Performance tests with our implementations of LBFGS, NCG, and GD with PELS in the Apache Spark framework were conducted with a logistic regression model on large classification datasets on a 16 node cluster with 256 processing cores. It was found, perhaps surprisingly, that NCG with PELS often exhibited better convergence and faster performance than LBFGS with a standard Wolfe approximate line search. For large datasets, the PELS technique also significantly reduced the number of iterations and time required by the LBFGS algorithm to reach high training accuracies by factors of 1.8–2. The PELS technique may be used to accelerate parallel large-scale regression and matrix factorization computations, and is applicable to important classes of smooth optimization problems.

map operations.

4

When ALS Met NCG

DID you know that people's tastes in movies and music and consumer merchandise are easily modelled and predicted by something as simple as a mathematical *dot product*? Or that this technique powers the eager product recommendation systems of Amazon [68], Netflix [7], and Spotify [60], just to name a few? Well don't worry if you were out of the loop—this chapter is your how-to guide on finding out what people want using the data about what all the other people wanted.

Low-rank matrix factorization is one of the simplest yet effective techniques [7, 35] in the field of *collaborative filtering*, in which many people's preferences for products are used to build models for recommending those products to other people [95, 61]. The low-rank latent factor model works by associating with each user and product a vector of rank n_f, of which each component is a linear measurement of how much that user *likes* the *factor* or *feature* associated with that component (or conversely, how much of that component the item *has*). This may sound sort of vague, but that's why the factors are called *latent*: they're supposed to capture the je ne sais quoi of what you like and put it into a number. For movies the factors might be interpretable as genres or motifs,

such that work by filmmakers like Scorsese, Quentin Tarantino, and the Coen Bros. would be represented by vectors with similar measurements [in all the right places].

So how does this work? Inserting ourselves specifically in the user-movie framework, a low-rank matrix factorization procedure takes as its input a user-movie ratings matrix $\mathbf{R}$, in which each entry is a numerical rating of a movie by a user, and for which the known entries are notoriously few.[1] The procedure then determines the low-rank user and movie matrices, $\mathbf{U}$ and $\mathbf{M}$ respectively, where a column in these matrices represents a latent feature vector for a single user or movie, respectively, and (ideally) $\mathbf{R} = \mathbf{U}^T\mathbf{M}$ for the known values in $\mathbf{R}$. This relationship is illustrated in Fig. 4.1, in which a column $\mathbf{u}_i$ of $\mathbf{U}$ representing a user and a column $\mathbf{m}_j$ of $\mathbf{M}$ representing an movie align such that their inner product produces the numerical ranking of movie j by user i as $\mathbf{u}_i^T\mathbf{v} = r_{ij}$. The real trick is solving the resultant optimization problem of determining good values of $\mathbf{U}$ and $\mathbf{M}$ such that the relationship $\mathbf{R} \approx \mathbf{U}^T\mathbf{M}$ holds—which is where we come in.

The standard approach to determine $\mathbf{U}$ and $\mathbf{M}$ is by minimizing the sum of the squared difference $(r_{ij} - \mathbf{u}_i^T\mathbf{m}_j)^2$ for all known $\{r_{ij}\}$. And just like any optimization problem, there are a number of techniques on the market, such as distributed SGD by Gemulla et al. [42], coordinate descent by Yu et al. [110], and the Alternating Least Squares (ALS) algorithm [116]. In particular, the ALS algorithm is well-established as a choice method [62, 41]. ALS is easily parallelized [116, 56], and has recently been proposed as an efficient nonlinear preconditioner for the Nonlinear Conjugate Gra-

[1]For example, the MovieLens 20M dataset with 138,493 users and 27,278 movies contains only 20 million ratings (less than 1% of all possibilities), and in the famous Netflix Competition, the Netflix dataset of half a million ratings included information from customers who had mostly rated fewer than 5 movies. However, in this latter Netflix prize, there were users who had somehow legitimately rated more than 10,000 movies [81]. As this is on the order of two years' worth of nonstop moviegoing, it has to raise at least a few eyebrows—if not about the dataset, then about something else with a lot of that latent, je-ne-sais-quoi factor. To defer to Lily Tomlin: "If you read a lot of books, you're considered well-read. But if you watch a lot of TV, you're not considered well-viewed."

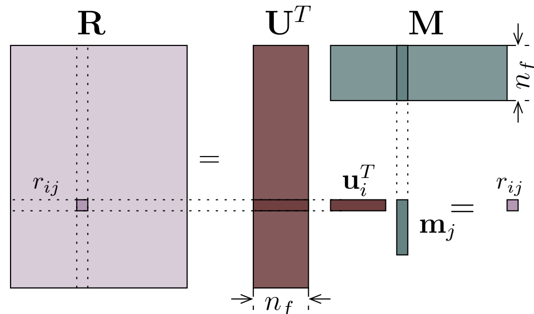


Figure 4.1: *Schematic of the relationship* $\mathbf{R} = \mathbf{U}^T\mathbf{M}$*. The column* $\mathbf{u}_i$ *of* $\mathbf{U}$ *representing the* i*th user and a column* $\mathbf{m}_j$ *of* $\mathbf{M}$ *representing the* j*th movie are aligned such that the inner product produces the ranking of movie* j *by user* i *as* $r_{ij} = \mathbf{u}_i^T\mathbf{v}$*.*

dient (Alg. 2) by De Sterck & Winlaw [30], who report that the combined ALS-NCG algorithm can converge faster for tensor decomposition problems than either of the two algorithms separately.

In this chapter, we present both a parallel implementation of the nonlinearly preconditioned ALS-NCG algorithm in the Apache Spark framework, built around Spark's existing ALS implementation for matrix factorization. In particular, because the ALS-NCG algorithm conducts a univariate line search in each iteration just like NCG, our ALS-NCG implementation is an ideal example to showcase the utility of the PELS algorithm from §3. We show that because of the structure of the matrix factorization loss function, the univariate line search function to be minimized is itself *polynomial*, and the Taylor coefficients computed by PELS are *exact*. As such, the line search routines in the ALS-NCG algorithm are fixed costs in each iteration, which improves the parallel efficiency.

The rest of this chapter is organized as follows. §4.1 formulates the matrix factorization loss function, which is part of the family of mean loss functions. The ALS algorithm and the existing implementation in Spark is then described in §4.2, followed by a description of the nonlinearly preconditioned ALS-NCG algorithm in §4.3 and its parallel implementation in Apache Spark. §4.4, describes our performance experiments using the MovieLens 20M dataset [51] and big synthetic datasets sampled from it with 6

million users and 800 million ratings. We present our observations that ALS-NCG was capable of optimizing the **U** and **M** matrices faster than the existing ALS implementation in Spark by factors of 2–5 in clock time. Furthermore, like the standalone ALS algorithm, the implementation is linearly scalable with the size of the input data.

4.1 Low Rank Matrix Factorization

In a low-rank factorization model [62, 116], we attempt to reconstruct the known entries of the sparse matrix $\mathbf{R} \in \mathbb{R}^{n_u \times n_m}$ within a rank-n_f feature space, with the final objective of correctly *predicting* the *unknown* values in **R**. In the user/movies setting we consider, we adopt the notation that n_u is the number of users, and n_m is the number of items. For the user feature matrix $\mathbf{U} = [\mathbf{u}_i] \in \mathbb{R}^{n_f \times n_u}$ and movie feature matrix $\mathbf{M} = [\mathbf{m}_j] \in \mathbb{R}^{n_f \times n_m}$, we construct a loss function measuring the prediction error for each of the known values of $\{r_{ij}\} \in \mathbf{R}$ such that $\mathbf{R} \approx \mathbf{U}^T\mathbf{M}$. Explicit in this relationship is that each vector $\mathbf{u}_i \in \mathbb{R}^{n_f}$ is associated with a user $1 \leq i \leq n_u$, and each vector $\mathbf{m}_j \in \mathbb{R}^{n_f}$ is associated with a movie $1 \leq j \leq n_m$, such that the interaction of user i with movie j is $\mathbf{u}_i^T\mathbf{m}_j \approx r_{ij}$. The objective is then to minimize a squared loss function dependent on *only* the index set $\mathcal{I}$ of known ratings,

$$\begin{aligned}\mathcal{L}(\mathbf{U},\mathbf{M}) = \sum_{(i,j)\in\mathcal{I}} (r_{ij} - \mathbf{u}_i^T\mathbf{m}_j)^2 + \\ \lambda\Big(\sum_i n_{u_i}\|\mathbf{u}_i\|^2 + \sum_j n_{m_j}\|\mathbf{m}_j\|^2\Big),\end{aligned} \tag{4.1}$$

where n_{u_i} denotes the number of ratings by user i, and n_{m_j} is the number of ratings of movie j. This latter term $\lambda(\sum_i n_{u_i}\|\mathbf{u}_i\|^2 + \sum_j n_{m_j}\|\mathbf{m}_j\|^2)$ for $\lambda > 0$ is a Tikhonov regularization [103] term included to prevent overfitting.

4.2 Alternating Least Squares

The ALS algorithm is a simple one. In each iteration, the method proceeds by cyclically *fixing* one of the unknown matrices—either **U**

or **M**—and then solving for the remaining *variable* matrix column by column, since each column can be written as the solution to a system of linear equations. In §4.2.1 we derive the expressions for these systems and show how ALS is immediately parallelizable, and then discuss in §4.2.2 the parallel implementation in the Apache Spark codebase based on the work by Johnson [59].

4.2.1 Derivation & Parallelization

We consider the first step of the ALS algorithm in which the movie matrix $\mathbf{M}$ is fixed. The least squares solution to (4.1) is then determined for each $\mathbf{u}_i$ by setting to zero each component of the gradient of (4.1) related to $\mathbf{u}_i$: $\frac{\partial \mathcal{L}_\lambda}{\partial u_{ki}} = 0$ for each kth component of $\mathbf{u}_i$ such that u_{ki} is an element of $\mathbf{U}$. Expanding the terms in the derivative of $\mathcal{L}$, we then have the equality

$$\sum_{j \in \mathcal{I}_i} 2(\mathbf{u}_i^T \mathbf{m}_j - r_{ij}) m_{kj} + 2\lambda n_{u_i} u_{ki} = 0 \;\; \forall\, i, k$$

in which m_{kj} is a scalar element of $\mathbf{M}$ and $\mathcal{I}_i$ is the index set of movies that user i has rated. This form may be rearranged and vectorized to achieve the resultant linear system for any $\mathbf{u}_i$,

$$(\mathbf{M}_{\mathcal{I}_i} \mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i} \mathbf{I}) \mathbf{u}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{R}^T(i, \mathcal{I}_i),$$

where $\mathbf{I}$ is the $n_f \times n_f$ identity matrix, and $\mathbf{M}_{\mathcal{I}_i}$ represents the sub-matrix of $\mathbf{M}$ where columns with indices from $\mathcal{I}_i$ are selected. Similarly, $\mathbf{R}(i, \mathcal{I}_i)$ is a row vector that represents the ith row of $\mathbf{R}$ with only the columns in $\mathcal{I}_i$ included. The explicit solution for any $\mathbf{u}_i$ is then given by

$$\mathbf{u}_i = \mathbf{A}_i^{-1} \mathbf{v}_i \;\; \forall\, i, \tag{4.2}$$

where $\mathbf{A}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i} \mathbf{I}$, and $\mathbf{v}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{R}^T(i, \mathcal{I}_i)$.

The analogous solution for the columns of $\mathbf{M}$ is found by fixing $\mathbf{U}$, such that each $\mathbf{m}_j$ is given by the solution of the system

$$\mathbf{m}_j = \mathbf{A}_j^{-1} \mathbf{v}_j \;\forall\, j \tag{4.3}$$

where $\mathbf{A}_j = \mathbf{U}_{\mathcal{I}_j} \mathbf{U}_{\mathcal{I}_j}^T + \lambda n_{m_j} \mathbf{I}$ and $\mathbf{v}_j = \mathbf{U}_{\mathcal{I}_j} \mathbf{R}(j, \mathcal{I}_j)$. Here, $\mathcal{I}_j$ is the index set of users that have rated movie j, $\mathbf{U}_{\mathcal{I}_j}$ represents the sub-matrix of $\mathbf{U} \in \mathbb{R}^{n_f \times n_u}$ where columns in the index set $\mathcal{I}_j$ are

Algorithm 5: Alternating Least Squares (ALS)

Output: $\mathbf{U}, \mathbf{M}$

1 Initialize $\mathbf{M}$
2 **while** *not converged* **do**
3   **for** $i = 1, \ldots, n_u$ **do**
4         $\mathbf{u}_i \leftarrow \mathbf{A}_i^{-1}\mathbf{v}_i$
5   **end**
6   **for** $j = 1, \ldots, n_m$ **do**
7         $\mathbf{m}_j \leftarrow \mathbf{A}_j^{-1}\mathbf{v}_j$
8   **end**
9 **end**

selected, and $\mathbf{R}(j, \mathcal{I}_j)$ is a column vector that represents the jth column of $\mathbf{R}$ with only the rows in $\mathcal{I}_j$ included. From (4.2) and (4.3), it is readily apparent that each of the columns of $\mathbf{U}$ and $\mathbf{M}$ may be computed independently since all $\{\mathbf{u}_i\}$ depend only on the *fixed* $\{\mathbf{m}_j\}$ and vice versa. Thus can ALS be easily parallelized, as done by Zhou et al. [116]. In addition, note that non-negativity bound constraints can be easily enforced in practice through the choice of linear solver for $\mathbf{u}_i$ and $\mathbf{m}_j$ (see e.g. an NCG variant for bound-constrained problems given by Polyak [88]). The full ALS algorithm for iteratively minimizing (4.1) repeats the procedure described above, and is summarized in Alg. 5.

4.2.2 ALS Implementation in Spark

The Spark implementation of ALS has several data structures for optimized communication patterns when forming the matrices $\mathbf{A}_i$ and $\mathbf{A}_j$ for all i, j. We first give an overview of the RDD partitioning, and then describe the *routing table* data structures that reduce the size of data shuffled between RDD partitions in each ALS iteration.

Overview. The ALS codebase stores the factor matrices $\mathbf{U}$ and $\mathbf{M}$ as single precision column block matrices, with each block as a single element of an RDD partition. A given $\mathbf{U}$ column block stores factor vectors for a subset of the users, and an $\mathbf{M}$ column block stores factor vectors for a subset of the movies. The rat-

ings matrix is stored twice: both $\mathbf{R}$ and $\mathbf{R}^T$ are stored in separate RDDs, partitioned in row blocks (i.e., $\mathbf{R}$ is partitioned by users and $\mathbf{R}^T$ by movies), both stored in a compressed sparse matrix format. The $\mathbf{U}$ and $\mathbf{R}$ RDDs have the same partitioning, such that a node that stores a partition of $\mathbf{U}$ also stores the partition of $\mathbf{R}$ such that the ratings for each user in its partitions of $\mathbf{U}$ are locally available. When updating a $\mathbf{U}$ block according to (4.2), the required ratings are available in a local $\mathbf{R}$ block, but the movie factor vectors in $\mathbf{M}$ corresponding to the movies rated by the users in a local $\mathbf{U}$ block must be shuffled across the network. These movie factors are fetched from different nodes, and, as explained below, an optimized routing table strategy is used that avoids sending duplicate information [59]. Similarly, updating a block of $\mathbf{M}$ according to (4.3) uses ratings data stored in a local $\mathbf{R}^T$ block, but requires shuffling of $\mathbf{U}$ factor vectors using a second routing table.

Block Partitioning. All RDDs are partitioned into n_b partitions[2], where, in practice, n_b is an integer multiple of the number of available compute cores. For example, $\mathbf{M}$ is divided into column blocks $\mathbf{M}_{j_b}$ with block (movie) index $j_b \in \{0, \ldots, n_b - 1\}$ by hash partitioning the movie factor vectors such that $\mathbf{m}_j \in \mathbf{R}_{j_b}$ if $j \equiv j_b \pmod{n_b}$ as in Fig. 4.2. Similarly, $\mathbf{U}$ is hash partitioned into column blocks $\mathbf{U}_{i_b}$ with block (user) index $i_b \in \{0, \ldots, n_b - 1\}$. The RDDs for $\mathbf{M}$ and $\mathbf{U}$ can be taken as type $\texttt{RDD}\left[(j_b, \mathbf{M}_{j_b})\right]$ and $\texttt{RDD}\left[(i_b, \mathbf{U}_{i_b}^T)\right]$, where the blocks are tracked by the indices j_b and i_b. $\mathbf{R}$ is partitioned by rows (users) into blocks with type $\texttt{RDD}\left[(i_b, \mathbf{R}_{i_b})\right]$ with the same partitioning as the RDD representing $\mathbf{U}$ (and similarly for the $\mathbf{R}^T$ and $\mathbf{M}$ RDDs). By sharing the same user-based partitioning scheme, the blocks $\mathbf{R}_{i_b}$ and $\mathbf{U}_{i_b}$ are normally located on the same compute node, except when faults occur. The same applies to $\mathbf{R}_{j_b}^T$ and $\mathbf{M}_{j_b}$ due to the movie-based partitioning scheme.

Routing Table. Fig. 4.3 shows how a routing table optimizes data shuffling in ALS. Suppose we want to update the user factor block $\mathbf{U}_{i_b}$ according to (4.2). The required ratings data, $\mathbf{R}_{i_b}$, is stored locally, but a global shuffle is required to obtain all the movie factor vectors in $\mathbf{M}$ that correspond to the movies rated by the users in $\mathbf{U}_{i_b}$. To optimize the data transfer, a routing table

[2] Strictly speaking, the $\mathbf{U}$ and $\mathbf{M}$ matrices may be partitioned with different numbers of blocks, but for simplicity we use n_b for both.

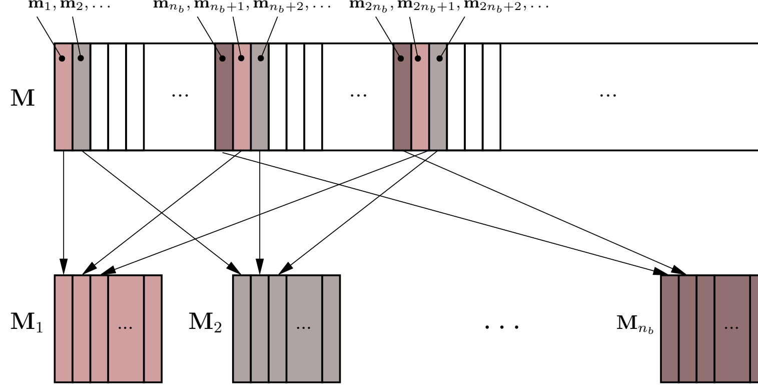


Figure 4.2: *Hash partitioning of the columns of* $\mathbf{M}$*, depicted as rectangles, into* n_b *blocks* $\mathbf{M}_1, \ldots, \mathbf{M}_{n_b}$*. Each block* M_{j_b} *and its index* j_b *forms a partition of the* $\mathbf{M}$ *RDD of type* `RDD` $[(j_b, M_{j_b})]$.

$T_m(\mathbf{m}_j)$ is constructed by determining, for each of the movie factor blocks $\mathbf{M}_{j_b}$, which factor vectors have to be sent to each partition of $\mathbf{R}_{i_b}$ (that may reside on other nodes). In Fig. 4.3a, the blocks $\mathbf{M}_{j_b}$ are *filtered* using $T_m(\mathbf{m}_j)$ such that a given $\mathbf{m}_j \in \mathbf{M}_{j_b}$ is written to the buffer destined for $\mathbf{R}_{i_b}$, $\mathbf{M}_{j_b}^{[i_b]}$, *only once* regardless of how many $\mathbf{u}_i$ in $\mathbf{U}_{i_b}$ have ratings for movie j, and *only* if there is at least one $\mathbf{u}_i$ in $\mathbf{U}_{i_b}$ that has rated movie j. This is shown by the hatching of each $\mathbf{m}_j$ vector in Fig. 4.3a; for instance, the first column in $\mathbf{M}_1$ is written only to $\mathbf{M}_1^{[1]}$ and has one set of hatching lines, but the last column is written to both $\mathbf{M}_1^{[2]}$ and $\mathbf{M}_1^{[n_b]}$ and correspondingly has two sets of hatching lines. Once the buffers are constructed, they are shuffled to the partitions of $\mathbf{R}$, as in Fig. 4.3b such that both the movies factors are ratings are locally available to compute the new $\mathbf{U}_{i_b}$ block, as in Fig. 4.3c. The routing table formulation for shuffling $\mathbf{U}$, with mapping $T_u(\mathbf{u}_i)$, is analogous. Note that the routing tables are constructed before computation begins in the ALS algorithm, and hence do *not* need recomputation in each iteration.

Updating $\mathbf{u}_i$ and $\mathbf{m}_j$. As in Fig. 4.3c, a compute node that stores $\mathbf{R}_{i_b}$, will obtain n_b buffered arrays of filtered movie factors.

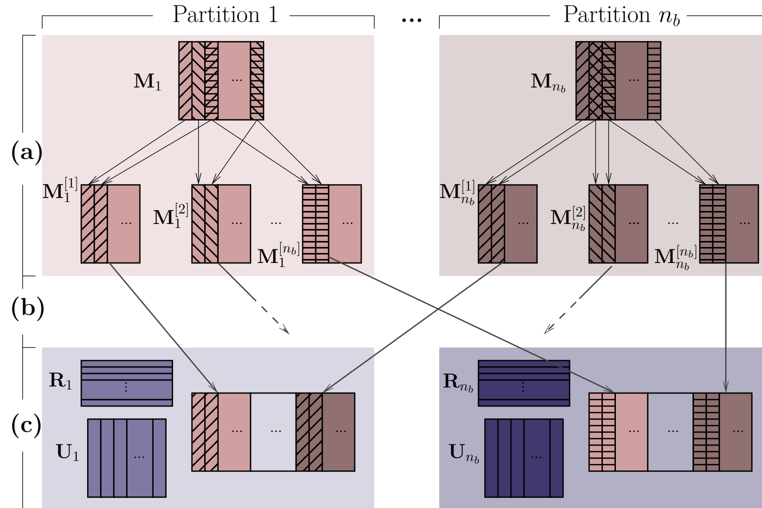


Figure 4.3: *Schematic of the use of the routing table $T_m(m_j)$ in the Spark ALS shuffle. In (a) the blocks $\{\mathbf{M}_{j_b}\}$ are filtered using $T_m(\mathbf{m}_j)$ for each destination $\mathbf{R}_{i_b}$ and shuffled to the respective blocks in (b), where arrows between the shaded backgrounds represent network data transfer between different compute nodes. In (c), when updating block $\mathbf{U}_{i_b}$, the ratings information is locally available in $\mathbf{R}_{i_b}$.*

Once the factors have been shuffled, $\mathbf{A}_j$ is computed for each $\mathbf{u}_i$ as $\sum_{j\in\mathcal{I}_i} \mathbf{m}_j\mathbf{m}_j^T + \lambda n_{u_i}\mathbf{I}$, and $\mathbf{v}_j$ as $\sum_{j\in\mathcal{I}_i} r_{ij}\mathbf{m}_j$ using the Basic Linear Algebra Subprograms (BLAS) library [11].[3] The resulting linear

[3] Jargon alert here and to follow. The BLAS library [technically, specification: there are several libraries available with different licenses and architecture-dependent optimizations that implement all the routines mandated by the specification] is a standard set of subroutines used ubiquitously in numerical linear algebra, like dot and matrix-vector products. The LAPACK library routines use BLAS routines in turn as components of solvers for linear systems and eigenvalue decomposition. The functions in all these libraries were written and debugged in Fortran once upon a time by early waves of computer scientists and mathematicians standardizing the first reusable numerical codes, and hence follow

system for $\mathbf{u}_i$ is then solved via the Cholesky decomposition using LAPACK routines [5] if no constraints are enforced, giving the computation to update $\mathbf{U}$ an asymptotic complexity of $\mathcal{O}(n_u n_f^3)$ since n_u linear systems must be solved. Solving for $\mathbf{M}$ is an identical operation with the appropriate routing table and has $\mathcal{O}(n_m n_f^3)$ complexity. If non-negativity constraints are specified, the systems are solved via a modified form of a bound-constrained NCG algorithm [88].

4.3 The ALS-NCG Algorithm

In this section we present the combined ALS-NCG algorithm. Our approach in §4.3.1 is to frame the hybrid algorithm as a nonlinearly preconditioned NCG method in the general framework given by De Sterck & Winlaw [30]. The contributions of parallelizing ALS-NCG for Spark then follow in §4.3.2, wherein the technical challenges of adding the additional NCG components to the existing Spark ALS framework are addressed. Since ALS-NCG requires a line search, the main consideration is how to compute the loss function (4.1) and its gradient in an efficient way in Spark so that good parallel performance is attainable. To this end, we demonstrate that the PELS coefficients computed for a fixed descent direction of (4.1) are exact and can be performed with a single Spark shuffle step. In addition, we show how to take advantage of the routing table mechanism to obtain a cheaper communication phase in computing the Taylor coefficients.

---

the notoriously terse naming conventions that were in vogue at the time. Example: the mnemonic function `axpby` performs the scaled vector addition, $a\mathbf{x} + b\mathbf{y}$. BLAS operations are divided into levels, such that level-1 is for vector-vector operations like dot products, level-2 is for matrix-vector operations such as $\mathbf{A} \cdot \mathbf{x}$, and level-3 operations are matrix-matrix operations such as full-blown multiplication, $\mathbf{A} \cdot \mathbf{B}$. Perhaps this footnote strikes you as overkill, but know that the initial published manuscript of this chapter contained several references to these specific routines [which seemed sensible at the time] that are preserved herein—just a heads up.

4.3.1 Nonlinearly Preconditioned NCG

Recall that the standard NCG algorithm in Alg. 2 generates search directions by the linear recombination of previous directions and gradient information. The nonlinearly preconditioned NCG technique builds on this idea by taking a *preconditioned* gradient vector rather than the *actual* gradient vector. First, with respect to the matrices $\mathbf{U}$ and $\mathbf{M}$, define the solution vector $\mathbf{x} \in \mathbb{R}^{n_f \times (n_u + n_m)}$ as the augmented vector

$$\mathbf{x}^T = \left[\mathbf{u}_1^T \; \mathbf{u}_2^T \dots \mathbf{u}_{n_u}^T \; \mathbf{m}_1^T \; \mathbf{m}_2^T \dots \mathbf{m}_{n_m}^T\right]. \tag{4.4}$$

For the iterate $\mathbf{x}_k$, the preconditioned direction $\bar{\mathbf{g}}_k$ is constructed through the application of a nonlinear preconditioning function $\mathbf{P}(\mathbf{x})$. Taking the preconditioned iterate $\bar{\mathbf{x}}_k$ as

$$\bar{\mathbf{x}}_k = \mathbf{P}(\mathbf{x}_k), \tag{4.5}$$

the preconditioned gradient direction is computed by the difference

$$\bar{\mathbf{g}}_k = \mathbf{x}_k - \bar{\mathbf{x}}_k. \tag{4.6}$$

Here, $-\bar{\mathbf{g}}_k$ is expected to be a *gradient-like descent* direction that is an improvement compared to the steepest descent direction, $-\mathbf{g}_k$. As such, the form of $\mathbf{P}(\mathbf{x})$ should be the result of an another optimization routine (like ALS) that improves upon the current parameter vector $\mathbf{x}_k$, but may still benefit from a line search and the conjugacy constructed through the NCG search direction update equation, which is adapted from (1.4) to use the preconditioned direction as

$$\mathbf{p}_{k+1} = \beta_k \mathbf{p}_k - \bar{\mathbf{g}}_{k+1}. \tag{4.7}$$

Here the update parameter β from (1.5) is redefined to include *both* the actual gradient and the preconditioned gradient vectors as

$$\beta_k = \max\left(\frac{\mathbf{g}_{k+1}^T(\bar{\mathbf{g}}_{k+1} - \bar{\mathbf{g}}_k)}{\mathbf{g}_k^T \bar{\mathbf{g}}_k}, 0\right). \tag{4.8}$$

Other forms for β_k and analysis thereof are given by De Sterck & Winlaw, accompanied by convergence guarantees [30]. However, (4.8) is the only form considered here. The general nonlinearly preconditioned NCG algorithm is summarized in Alg. 6. At current, the calculation of the search direction $\mathbf{p}_{k+1}$ in Alg. 6 is not capable of handling non-negativity constraints on $\mathbf{U}$ and $\mathbf{M}$.

Algorithm 6: Nonlinearly Preconditioned NCG

Input: $\mathbf{x}_0 \in \mathbb{R}^m$
Output: $\mathbf{x}_k \approx \mathbf{x}^* = \arg\min_{\mathbf{x}} \mathcal{L}(\mathbf{x})$

1. $k \leftarrow 0$
2. $\bar{\mathbf{g}}_0 \leftarrow \mathbf{x}_0 - \mathbf{P}(\mathbf{x}_0)$
3. $\mathbf{p}_0 \leftarrow -\bar{\mathbf{g}}_0$
4. **while** *not converged* **do**
5. $\quad \alpha_k \leftarrow \arg\min_\alpha \mathcal{L}(\mathbf{x}_k + \alpha \mathbf{p}_k)$
6. $\quad \mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$
7. $\quad \bar{\mathbf{g}}_{k+1} \leftarrow \mathbf{x}_{k+1} - \mathbf{P}(\mathbf{x}_{k+1})$
8. $\quad$ Compute β_k
9. $\quad \mathbf{p}_{k+1} \leftarrow \beta_k \mathbf{p}_k - \bar{\mathbf{g}}_{k+1}$
10. $\quad k \leftarrow k + 1$

4.3.2 ALS-NCG Implementation in Spark

This section describes the implementation of the additional vector operations of the preconditioned NCG algorithm in Alg. 6 alongside the existing datastructures in the Spark ALS framework. We first define how the additional NCG vectors were implemented, such that vector operations like addition were possible with co-aligned RDD data structures. The univariate line search procedure required in step 5 of Alg. 6 is then presented, in which the PELS methodology of §3 is used to evaluate the loss function in (4.1) [but reparameterized by $\mathbf{x}$ as in (4.4) for the ALS-NCG algorithm] for a fixed search direction, $\mathcal{L}(\mathbf{x} + \alpha\mathbf{p})$. Finally, some notes on the evaluation of the gradient of (4.1) are given.

Vector Storage. The additional vectors $\bar{\mathbf{x}}$, $\mathbf{g}$, $\bar{\mathbf{g}}$, and $\mathbf{p}$ for each iteration were each split into two separate RDDs, such that blocks corresponding to the components of $\mathbf{u}_i$ were stored in one RDD, partitioned in the same way as $\mathbf{U}$ with block index i_b. Analogously, blocks corresponding to components of $\mathbf{m}_j$ were stored in another RDD, partitioned in the same way as $\mathbf{M}$ with block index j_b. This ensured that all vector blocks were aligned component-wise for vector operations; furthermore, the $\mathbf{p}$ blocks could also be shuffled efficiently using the routing tables, which was important in the line search (to follow).

Algorithm 7: RDD Block Vector `BLAS.axpby`

Input: $\mathbf{x} = \texttt{RDD}\,[(i_b, \mathbf{x}_{i_b})]$; $\mathbf{y} = \texttt{RDD}\,[(i_b, \mathbf{y}_{i_b})]$; $a, b \in \mathbb{R}$
Output: $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$

1 $\mathbf{z} \leftarrow \mathbf{x}$.join($\mathbf{y}$).map{ $\forall i_b$
2 Allocate $\mathbf{z}_{i_b}$
3 $\mathbf{z}_{i_b} \leftarrow \mathbf{y}_{i_b}$
4 Call `BLAS.axpby`$(a, \mathbf{x}_{i_b}, b, \mathbf{z}_{i_b})$
5 Yield $(i_b, \mathbf{z}_{i_b})$
6 }

Vector Operations. RDDs have a standard operation termed a *join*, which concatenates the entries of two RDDs of key-value pairs by matching common keys. For RDDs R_1 and R_2 of tuples $\bigcup_i (k_i, a_i)$ and $\bigcup_i (k_i, b_i)$ with type $(\mathcal{K}, \mathcal{A})$ and $(\mathcal{K}, \mathbf{B})$, respectively, their join is an RDD R_3 of type $\texttt{RDD}\,[(\mathcal{K}, (\mathcal{A}, \mathbf{B}))]$, where $R_3 = R_1.\text{join}(R_2)$ represents the dataset $\bigcup_l (k_l, (a_l, b_l))$ of combined tuples and k_l is a key common to both R_1 and R_2. Parallel vector operations between RDD representations of blocked vectors $\mathbf{x}$ and $\mathbf{y}$ were implemented by joining the RDDs by their block index i_b and calling BLAS level-1 interfaces on the vectors within the resultant tuples of aligned vector blocks, $\{(\mathbf{x}_{i_b}, \mathbf{y}_{i_b})\}$. Since the RDD implementations of vectors had the same partitioning schemes, this operation was local to a compute node, and hence inexpensive. One caveat, however, is that BLAS subprograms generally perform modifications to vectors *in place*, overwriting their previous components. For fault tolerance, RDDs must be immutable; as such, whenever BLAS operations were required on the records of an RDD, an entirely new vector was allocated in memory and the RDD's contents copied to it. Alg. 7 shows the operations required for the vector addition $a\mathbf{x}+b\mathbf{y}$ using the `BLAS.axpby` routine (note that the result overwrites $\mathbf{y}$ in place). The inner product of two block vector RDDs and norm of a single block vector RDD were implemented in similar manners to vector addition, with an additional reduce operation.

Univariate Line Search. The PELS method from §3 yields a computationally cheap way to implement a line search in Spark for

the matrix factorization loss function. The form of $\mathcal{L}$ in (4.1) is exactly *quartic* for a a fixed descent direction. To see this, substitute the rays $\mathbf{x}_{\mathbf{u}_i} + \alpha \mathbf{p}_{\mathbf{u}_i}$ for $\mathbf{u}_i$ and $\mathbf{x}_{\mathbf{m}_j} + \alpha \mathbf{p}_{\mathbf{m}_j}$ for $\mathbf{m}_j$ in (4.1), where $\mathbf{x}_{\mathbf{u}_i}$ and $\mathbf{p}_{\mathbf{u}_i}$ refers to the components of $\mathbf{x}$ and $\mathbf{p}$ related to user i, respectively (of course, $\mathbf{x}_{\mathbf{u}_i} = \mathbf{u}_i$, but the notation attempts to remain analogous to the notation in step 5 of Alg. 6 and cirumvent the alternate and somewhat confusing substitution of $\mathbf{u}_i \leftarrow \mathbf{u}_i + \alpha \mathbf{p}_{\mathbf{u}_i}$). The vectors $\mathbf{p}_{u_i}$, $\mathbf{x}_{\mathbf{m}_j}$, and $\mathbf{p}_{\mathbf{m}_j}$ are defined identically, *mutatis mutandis*. Using the bilinearity of the inner product, we may expand the terms in the loss function for the squared difference between projected and actual ratings to obtain a degree-4 polynomial in α as

$$W(\alpha) = \sum_{n=0}^{4} \left(\sum_{(i,j) \in \mathcal{I}} C_{ij}^{[n]} \right) \alpha^n. \tag{4.9}$$

The second summation over the index set $\mathcal{I}$ is achievable through a Spark reduce overation once the coefficients $\{C_{ij}^{[n]}\}$ in (4.9) are computed. The full expression for $W(\alpha)$ is given below as

$$\begin{aligned}
W(\alpha) = {} & \sum_{\mathcal{I}} \left(\mathbf{x}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j} - r_{ij}\right)^2 + \lambda \Big(\sum_i n_{u_i} \|\mathbf{x}_{\mathbf{u}_i}\|_2^2 + \sum_j n_{m_j} \left\|\mathbf{x}_{\mathbf{m}_j}\right\|_2^2 \Big) \\
& + \alpha \Big[\sum_{\mathcal{I}} \left(\mathbf{x}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j} + \mathbf{p}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j}\right) \left(\mathbf{x}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j} - r_{ij}\right) \\
& \qquad + 2\lambda \Big(\sum_i n_{u_i} \mathbf{x}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{u}_i} + \sum_j n_{m_j} \mathbf{x}_{\mathbf{m}_j}^T \mathbf{p}_{\mathbf{m}_j} \Big) \Big] \\
& + \alpha^2 \Big[\sum_{\mathcal{I}} \Big(\mathbf{p}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j} \left[\mathbf{x}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j} - r_{ij}\right] + \left[\mathbf{x}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j}\right]^2 \\
& \qquad + \mathbf{p}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j} \left(\mathbf{x}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j} + 2\mathbf{p}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j}\right) \Big) \\
& \qquad + \lambda \Big(\sum_i n_{u_i} \|\mathbf{p}_{\mathbf{u}_i}\|_2^2 + \sum_j n_{m_j} \left\|\mathbf{p}_{\mathbf{m}_j}\right\|_2^2 \Big) \Big] \\
& + \alpha^3 \sum_{\mathcal{I}} 2\mathbf{p}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j} \left(\mathbf{x}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j} + \mathbf{p}_{\mathbf{u}_i}^T \mathbf{x}_{\mathbf{m}_j}\right) \\
& + \alpha^4 \sum_{\mathcal{I}} \left(\mathbf{p}_{\mathbf{u}_i}^T \mathbf{p}_{\mathbf{m}_j}\right)^2.
\end{aligned} \tag{4.10}$$

Note that the terms in the summations for each coefficient only require level-1 BLAS operations between the block vectors $\mathbf{x}_{\mathbf{u}_i}, \mathbf{p}_{\mathbf{u}_i}, \mathbf{x}_{\mathbf{m}_j}$, and $\mathbf{p}_{\mathbf{m}_j}$. The minimization in step 5 of Alg. 6 can be performed via minimizing the polynomial $W(\alpha)$ once its coefficients are computed, as in §3. By computing the coefficients of $W(\alpha)$ at the beginning of the line search, further evaluation of the loss function in each iteration is a constant time operation that requires very few operations; in practice, it took on the order of 10^{-3} s, while computing the initial coefficients was on the order of 10 s for the problems considered. Since each iteration of the line search was performed in constant time after computing the coefficients of $W(\alpha)$ and $W(\alpha)$, we used relatively stringent values in a simple backtracking line search (as formulated by Nocedal & Wright [85] with parameters $\rho = 0.9$, $c = 0.5$, and initial step size of 10) that searched intensively along the direction $\mathbf{p}$ to satisfy a sufficient decrease condition.[4] Finally, note that the summation over $\mathcal{I}$ in (4.1) may be performed in two ways; either $\mathbf{U}$ or $\mathbf{M}$ may be shuffled, depending on which routing table is used. Generally, $n_u \gg n_m$ and hence there is far more communication required to shuffle $\{\mathbf{u}_i\}$. Thus, in evaluating $\{C_{ij}^{[n]}\}$, the movie factors $\{\mathbf{m}_j\}$ were shuffled across the network.

Gradient Evaluation. We computed $\mathbf{g}_k$ with respect to a block for $\mathbf{u}_i$ using only BLAS level-1 operations as

$$\mathbf{g}_{\mathbf{u}_i} = 2\lambda n_{u_i}\mathbf{u}_i + 2\sum_{j\in\mathcal{I}_i}\mathbf{m}_j(\mathbf{u}_i^T\mathbf{m}_j - r_{ij}),$$

[4] With all of the work required to compute and reduce the coefficients $\{C_{ij}^{[n]}\}$ and emphasis on achieving better accuracy with PELS in §3, you might be aghast to hear that a *backtracking* line search was used. In retrospect, so are we. But this all becomes clearer once you understand the anecdotal history of the PELS method: (4.10) came about during frantic efforts to make the ALS-NCG implementation in Spark run fast enough. While initial serial experiments had shown that ALS-NCG could improve on the runtimes for ALS, the shuffle overhead for conducting multiple function and gradient evaluations in a line search made the method impractical. Realizing that (4.1) could be written as a polynomial in (4.10) came about in a moment of lucidity before the looming conference deadline for submitting the ALS-NCG manuscript. While the backtracking line search was later improved upon, it was less pressing than other matters at the time.

with an analogous operation for the gradients with respect to each $\mathbf{m}_j$. As this computation requires matching the $\mathbf{u}_i$ and $\mathbf{m}_j$ factors, the routing tables $T_u(\mathbf{u}_i)$ and $T_m(\mathbf{m}_j)$ were used to shuffle $\mathbf{u}_i$ and $\mathbf{m}_j$ consecutively. Evaluating $\mathbf{g}_k$ can be performed with $\mathcal{O}(n_f(n_u \sum_i n_{u_i} + n_m \sum_j n_{m_j}))$ operations, but requires two shuffles. As such, the gradient computation required as much communication as a single iteration of ALS in which both $\mathbf{U}$ and $\mathbf{M}$ are updated.

Evaluating β_k. The update parameter in (4.8) was used in the parallel implementation since it empirically produced smoother convergence traces. Computing the update parameter required the evaluation of $\mathbf{g}_{k+1}$ and storage of the gradient $\mathbf{g}_k$ from the previous iteration. The preconditioned direction $\bar{\mathbf{g}}_{k+1}$ was computed using RDD BLAS operations on the ALS-preconditioned vector $\bar{\mathbf{x}}_{k+1}$. However, to avoid additional the additional operation $\bar{\mathbf{g}}_{k+1} - \bar{\mathbf{g}}_k$, the inner product $\mathbf{g}_{k+1}^T \bar{\mathbf{g}}_{k+1}$ was stored between iterations, and β_k was computed using $\mathbf{g}_{k+1}^T \bar{\mathbf{g}}_{k+1} - \mathbf{g}_{k+1}^T \bar{\mathbf{g}}_k$ in the numerator of (4.8).

4.4 Parallel Performance of ALS-NCG

To compare the performance of ALS and ALS-NCG in Spark, the two algorithms' implementations were used to optimize user and movie matrices on the MovieLens 20M dataset with $\lambda = 0.01$ and $n_f = 100$. For each experimental run of ALS and ALS-NCG, two experiments with the same initial user and movie factors were performed: in one, the gradient norm in each iteration was computed and printed (incurring additional operations); in the other experiment, no additional computations were performed such that the elapsed times for each iteration were correctly measured. Since RDDs are materialized via lazy evaluation, to obtain timing measurements, actions were triggered to physically compute the partitions of each block vector RDD at the end of both each ALS subcomponent and each iteration of Alg. 6. In both algorithms, the RDDs were checkpointed to persistent storage every 10 iterations, since it is a widely known issue in the Spark community that RDDs with very long lineage graphs cause stack overflow errors when the scheduler recursively traverses their lineage [26].

Our comparison tests of the ALS and ALS-NCG algorithms

in Spark were performed on a computing cluster composed of 16 homogeneous compute nodes, 1 storage node hosting a network filesystem, and 1 master node. The nodes were interconnected by a 10 Gb ethernet managed switch (PowerConnect 8164). Each compute node was a 64 bit rack server (PowerEdge R620) running Ubuntu 14.04, with linux kernel 3.13 compiled with symmetric multiprocessing. The compute and master nodes all had two processors, both 8-core 2.60 GHz chips with 20M onboard cache (Xeon E5-2670). Compute nodes each had 256 GB of SDRAM, while the master node had 512 GB. The single storage node (PowerEdge R720) contained two 2 GHz processors, each with 6 cores (Xeon E5-2620), 64 GB of memory, and 12 hard disk drives of 4 TB capacity and 7200 RPM nominal speed. Finally, compute nodes were equipped with 6 ext4-formatted local SCSI 10k RPM hard disk drives, each with a 600 GB capacity.

Our Apache Spark assembly was built from a snapshot of the 1.3 release using Oracle's Java 7 distribution, Scala 2.10, and Hadoop 1.0.4. Input files to Spark programs were stored on the storage node in plain text. The SCSI hard drives on the compute nodes' local filesystems were used as Spark spilling and scratch directories, and the Spark checkpoint directory for persisting RDDs was specified in the network filesystem hosted by the storage node, and accessible to all nodes in the cluster. Shuffle files were consolidated into larger files, as recommended for ext4 filesystems [29]. In our experiments, the Spark driver was executed on the master node in standalone client mode, and a single instance of a Spark executor was created on each compute node. It was empirically found that the ideal number of cores to make available to the Spark driver was 16 per node, or the number of *physical* cores for a total of 256 available cores; hence the value of n_b was set to the number of cores in all experiments. Though the multithreading ability of our processors allowed for 32 *logical* cores to be specified, it was found that performance worsened in general by allocating more cores than physically available.

Fig. 4.4 shows the convergence in gradient norm, normalized by the degrees of freedom $N = n_f \times (n_u + n_m)$, for six separate runs of ALS and ALS-NCG on 8 compute nodes for the MovieLens 20M dataset. The subplots (a) and (b) show $\frac{1}{N}\|\mathbf{g}_k\|$ over 100 iterations and 25 minutes, respectively. This time frame was chosen since it

took just over 20 minutes for ALS-NCG to complete 100 iterations; note that in Fig. 4.4b, 200 iterations of ALS are shown, since with this problem size it took approximately twice as long to run a single iteration of ALS-NCG. The shaded regions in Fig. 4.4show the standard deviation about the mean value for $\frac{1}{N}\|\mathbf{g}_k\|$ across all runs, computed for non-overlapping windows of 3 iterations for subplot (a) and 30 s for subplot (b). From the width of these regions, we see that there is greater fluctuation in the gradient norm between iterations for ALS-NCG than ALS (however, traces of the loss in (4.1) computed in each iteration are smooth and monotonically decreasing). Yet even within this uncertainty bound, to reach accurate values of $\frac{1}{N}\|\mathbf{g}_k\|$ (e.g. below 10^{-3}), ALS-NCG requires much less time and many fewer iterations.

The operations that we have implemented in ALS-NCG that are additional to standard ALS in Spark have computational complexity that is linear in problem size. To verify the expected linear scaling, experiments with a constant number of nodes and increasing problem size up to 800 million ratings were performed on 16 compute nodes. Synthetic ratings matrices were constructed by sampling the MovieLens 20M dataset such that the synthetic dataset had the same statistical sparsity, realistically simulating the data transfer patterns in each iteration. To do this, first the dimension n_u of the sampled dataset was fixed, and for each user n_{u_i} was sampled from the empirical probability distribution $p(n_{u_i}|\mathbf{R})$ of how many movies each user ranked, computed empirically from the MovieLens 20M dataset. The n_{u_i} movies were them sampled from the empirical likelihood of sampling the jth movie, $p(\mathbf{m}_j|\mathbf{R})$, and the resultant rating value was sampled from the distribution of numerical values for all ratings. This method of scaling the users was chosen to model the situation in which an industry's user base grows far more rapidly than its items.

Fig. 4.5 shows the linear scaling in computation time for both ALS and ALS-NCG. The values shown are average times per iteration over 50 iterations, for n_u from 1 to 6 million, corresponding to the range from 133 to 800 million ratings. The time values were computed as a weighted average of time per iteration for iterations with and without checkpointing the RDDs to persistent storage, since the periodic checkpoint operation significantly increases the time required in that iteration. The error bars show the uncertainty

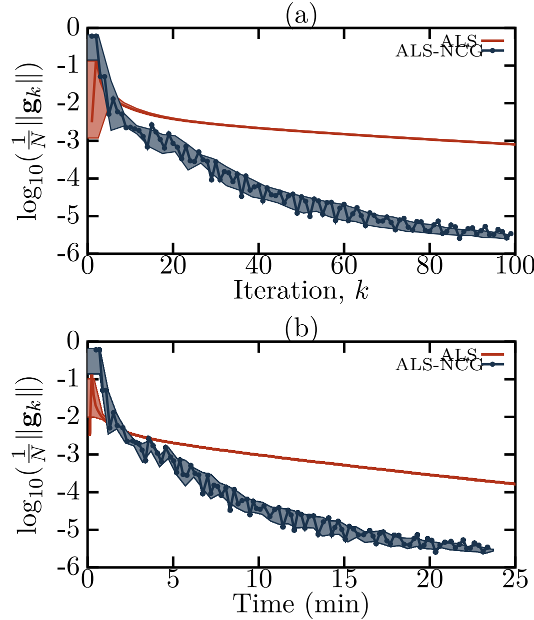


Figure 4.4: *Convergence in normalized gradient norm $\frac{1}{N}\|\mathbf{g}_k\|$ for 6 instances of ALS and ALS-NCG with different starting values in both (a) iteration and (b) clock time, for the MovieLens 20M dataset. The two solid lines in each panel show actual convergence traces, while the shaded regions show the standard deviation about the mean value over all instances, computed for non-overlapping windows of 3 iterations for (a), and 30 s for (b). The experiments were conducted on 8 nodes (128 cores).*

in this measurement, where the standard deviation about the mean values for both the checkpoint and non-checkpoint iterations have been propagated in the weighted average calculation. The uncertainty in the time per iteration for ALS-NCG is larger due to the greater overhead of memory management and garbage collection by the Java Virtual Machine required with more RDDs. While each iteration of ALS-NCG takes longer due to the additional line search and gradient computations, we note that many fewer iterations are

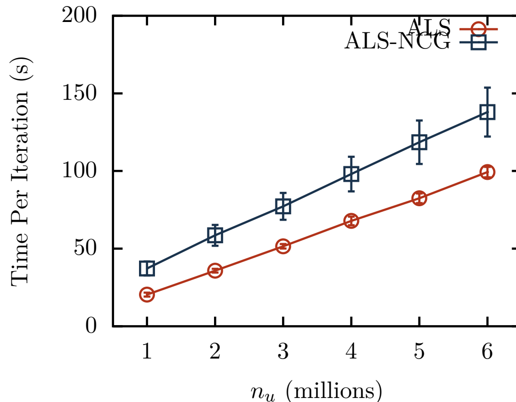


Figure 4.5: *Linear scaling of computation time per iteration with increasing n_u on a synthetic dataset for ALS and ALS-NCG on 16 compute nodes (256 cores) for up to 6M users, corresponding to 800M ratings.*

required to converge.

Finally, we compute the relative speedup that was attainable on the large synthetic datasets. For the value of $\frac{1}{N}\|\mathbf{g}_k\|$ in each iteration of ALS-NCG, we determined how many iterations of regular ALS were required to achieve an equal or lesser value gradient norm. Due to the local variation in $\frac{1}{N}\|\mathbf{g}_k\|$ (as in Fig. 4.4), a moving average filter over every two iterations was applied to the ALS-NCG gradient norm values. The total time required for ALS and ALS-NCG to reach a given gradient norm was then estimated from the average times per iteration in Fig. 4.5. The ratios of these total times for ALS and ALS-NCG are shown in Fig. 4.6 as the relative speedup factor for the 1M, 3M, and 6M users ratings matrices. When an accurate solution is desired, ALS-NCG often achieves faster convergence by a factor of 3–5, with the acceleration factor increasing with greater desired accuracy in the solution (i.e. decreasing $\frac{1}{N}\|\mathbf{g}_k\|$).[5]

[5]Note that while the speedup ratios in Fig. 4.6 agree with the acceleration in Fig. 4.4, the optimization problem being solved on synthetic data sampled from the distribution of movies in the MovieLens 20M

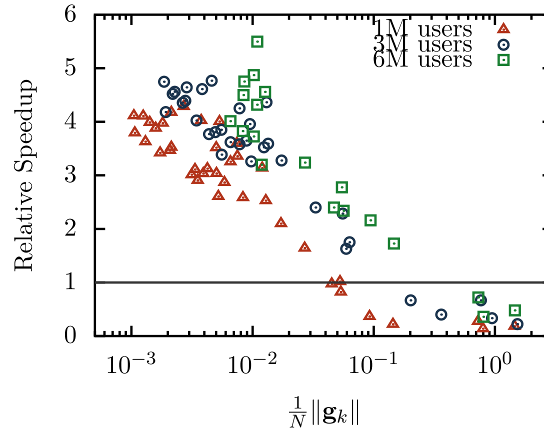


Figure 4.6: *Speedup of ALS-NCG over ALS as a function of normalized gradient norm on 16 compute nodes (256 cores), for a synthetic problem with up to 6M users and 800M ratings. ALS-NCG can easily outperform ALS by a factor of 4, especially when accurate solutions (small normalized gradients) are required.*

4.5 Conclusion

In this chapter, we have presented a case study in which the nonlinearly preconditioned NCG algorithm was successfully applied for

---

dataset is fundamentally different than the *actual* data, since the structure and nuance of people's likes and dislikes has been washed away and replaced by aggregate statistics. We can only conjecture whether this synthetic problem is easier or harder to solve (although a simple experiment to comment on this would be to compare the convergence rates in Fig. 4.4 with those from sampled datasets of the MovieLens 20M data with the same dimensions as the original), which is perhaps related to a broader question about how diverse or complex people's tastes in movies or stories are *really*. We shall not tackle that question here, bearing in mind that Kurt Vonnegut Jr. once proposed as a (rejected) thesis topic for his Master's degree in anthropology [which was also later rejected] an analysis of storylines that suggested that Cinderella and the Judeo-Christian Creation story were paradigmatically identical [105]. Apparently his commitee wasn't pleased.

optimizing feature matrices in low-rank matrix factorization problems. The particular form of the matrix factorization loss function lends itself naturally to the W method from §3, where it was here used to evaluate the *exact* Taylor coefficients and minimize the subsequent polynomials in univariate line searches. This technique circumvents having to evaluate the loss function multiple times in a line search invocation, since the polynomial may be minimized locally on the Spark driver program. The parallel ALS-NCG implementation is considered a highly preferable alternative to the standard ALS implementation in Spark for practitioners and organizations seeking efficient and scalable recommender systems, based on the significant speedup factors achievable with this method. We expect that our acceleration approach will be especially useful for advanced collaborative filtering models that achieve low root mean square error (RMSE), since these models require solving the optimization problem accurately, and that is precisely where accelerated ALS-NCG shows the most benefit over standalone ALS. Future interesting directions of research include experimentation with other distributed optimization methods as preconditioners, such as distributed Stochastic Gradient Descent [42, 101]. Furthermore, extensions of this approach for the popular ALS-based implicit feedback algorithm [56], which is already implemented in Spark, may be of interest to the collaborative filtering community.

5

PELS for the Multilayer Perceptron

Artificial neural networks (ANNs) are popular models in the fields of statistics and machine learning, and have been successfully applied to tasks such as image classification [31], speech recognition [45] and molecular interaction modelling [71]. Their success in pattern recognition likely stems from their hierarchical nature allowing representation of highly nonlinear relationships between network parameters, since the prototypical ANN function is composed of a sequence of densely coupled sigmoidal classification units. However, this flexibility comes with a price: deep ANNs are both very high-dimensional and nonconvex, and determining optimal network parameters is notoriously challenging for optimization algorithms [54, 74, 97].

The optimization algorithm of choice in the ANN community has long been Stochastic Gradient Descent (SGD) [15, 53, 97], which is a variant of the steepest descent method in which an approximate gradient instead of the full gradient is used to update the parameters, and a step size decreasing according to a preordained function—termed the *learning rate* in the machine learning community—is used for each update instead of a line search. While SGD has very impressive convergence properties when the step

size in each iteration is well-tuned, it is an inherently sequential algorithm. As such, parallelized extensions are an active research topic in which several camps exist. Some prominent methods include *minibatch* techniques [33] in which a subset of the dataset under consideration is sampled to compute a stochastic gradient, asynchronous methods [31, 90] in which the parameters are updated concurrently by multiple optimization routines running SGD without synchronization, and a surprising technique from people at Yahoo! that averages solution vectors determined by parallel and independent SGD routines on (non-disjoint) subsets of the dataset [118].

Admittedly, this chapter is not about any of those methods: we specifically confine ourselves to batch optimization. While an overview of optimization methods for ANNs cannot omit SGD's market dominance, there exist circumstances in which batch methods are [arguably] legitimately preferable. For instance, in Yoshua Bengio's discussion [8] of practical considerations in optimizing ANN parameters, he outlines both how having to tune SGD's learning rate meta-parameters is an inherent and time-consuming meta-algorithm that must be done in addition to the *bona fide* SGD routine,[1] Bengio also discusses having to practically determine the

[1]The meta-optimization of tuning parameters in SGD really is a well known hidden cost—so much so that meta-algorithms for choosing a *good* learning rate encompass an extensive field of research in their own right. The most renowned methods of this type are perhaps ADAGRAD [36] or ADADELTA [115], as well as the more recent *Pesky* paper by Schaul et al. [96] whose principles are advanced and extended in a production system described by Dalessandro et al. [28]. Recently in the proceedings of NIPS 2015, an interesting *probabilistic* line search with stochastic Wolfe conditions was proposed by Mahsereci & Hennig [73], which resonates in this thesis for obvious reasons. With the furor of development in this field, it's worth sharing a quotation from the original 1951 SGD paper by Robbins & Monro [91], who prefaced their convergence proof with an almost apologetic remark that "no claim is made that the procedure to be described has any optimum properties (i.e. that it is *efficient*)". Decades later the advent of Big Data entirely rephrased that efficiency question, and their work became a household name in machine learning—keep that in mind the next time someone bemoans tax dollars wasted on fusion reactors that are never going to work.

break-even point at which there is sufficient data for SGD to best LBFGS in running times, which is echoed in a fairly recent empirical case report of first order optimization methods by Le et al. [83] that compares minibatch SGD with minibatch versions of the NCG and LBFGS algorithms. They write that their particular formulation of minibatch LBFGS can outperform minibatch SGD in elapsed clock time for GPU workstations and small clusters (of 2–8 nodes), which is a parallel setting available to many researchers and small to medium size companies.[2] As such, a formulation of the PELS methodology may be a useful tool for practitioners who employ parallel batch algorithms in this setting and must conduct univariate line searches as part of those algorithms.

In this chapter, the PELS technique is applied to the smooth, sigmoidal multilayer perceptron (MLP), which is a classic but important type of neural network model. §5.1 presents a compact matrix formulation of the MLP, followed by a derivation of matrix expressions for the polynomial coefficients of a degree-d Taylor expansion of the MLP for a softmax output layer and a squared loss function in §5.2. Since the MLP is nonlinear and nonconvex, §5.2 also presents modifications to the PELS algorithm to adapt it to this setting. The experiments in §5.3 describe an initial feasibility study comparing LBFGS and NCG using a Wolfe approximate line search [80] with the LBFGS-P and NCG-P algorithms using the nonconvex PELS procedure on several datasets for binary and multiclass classification problems. While this initial effort has been

[2] There are currently many formulations of stochastic or minibatch LBFGS being proposed in the literature each year, so this statement begs further explanation if it is to lend support to our argument that batch and/or minibatch methods are of interest. The procedure of Le et al. [where Andrew Ng of Stanford counts among that *alia*, if that helps] for minibatch LBFGS and NCG is to sample a subset of the dataset and run the *deterministic* version of these algorithms on the subset for 20 and 3 iterations, respectively, before sampling anew and repeating *ad libitum*. While theirs is a very different approach from those of e.g. Byrd et al. [18, 20] or Sohl-Dickstein et al. [99] (from which the introduction is recommended as an overview of the current players in stochastic quasi-Newton research), and admittedly lacks an accompanying theoretical foundation, it has the benefits both of being simple and having been reproduced empirically in our own limited experiments.

done in a serial setting on small datasets, the results in §5.4 show that the PELS methodology can accelerate the convergence of these batch algorithms in terms of the number of iterations required to reach terminal values of the loss function, particularly for NCG.

5.1 Sigmoidal Multilayer Perceptron

An MLP is a nonlinear function $\mathcal{N}:\mathbb{R}^{p_0}\to\mathbb{R}^{p_N}$ that computes an output vector through the iterative application of nonlinear functions to a given input vector $\mathbf{y}_0 \in \mathbb{R}^{p_0}$. These updates are said to occur in *layers*, producing the sequence of intermediate vectors associated with each ℓth layer, $\{\mathbf{y}_\ell\}_{\ell=1}^N$. Each $\mathbf{y}_\ell$ is computed recursively from $\mathbf{y}_{\ell-1}$ using the *weight* matrices $\{\mathbf{W}_\ell\}_{\ell=1}^N$ and *bias* vectors $\{\mathbf{b}_\ell\}_{\ell=1}^N$ according to the update rule

$$\begin{cases} \mathbf{y}_\ell &= \sigma_\ell\left(\mathbf{z}_\ell\right) \\ \mathbf{z}_\ell &= \mathbf{W}_\ell \cdot \mathbf{y}_{\ell-1} + \mathbf{b}_\ell, \quad 1 \le \ell \le N. \end{cases} \tag{5.1}$$

where $\sigma_\ell : \mathbb{R}^{p_\ell} \to \mathbb{R}^{p_\ell}$ is the layer function. Each $\mathbf{W}_\ell$ has dimension $\mathbb{R}^{p_\ell \times p_{\ell-1}}$, projecting the previous vector $\mathbf{y}_{\ell-1}$ into the space $\mathbb{R}^{p_\ell}$. Similarly, $\mathbf{b}_\ell \in \mathbb{R}^{p_\ell}$ adds a constant offset in each layer. The quantity $\mathbf{z}_\ell$ is thus an *affine* transformation applied to the vector $\mathbf{y}_{\ell-1}$, which is in turn a nonlinear mapping applied to an affine transformation. Thus the network is completely characterized by its weights and biases as the tuple $(\{\mathbf{W}_\ell\},\{\mathbf{b}_\ell\})$. A sample network for $N = 2$ is depicted in the graph in Fig. 5.1, where the vectors $\{\mathbf{y}_0,\mathbf{y}_1,\mathbf{y}_2\}$ are shown as vertices, and the edges represent computations through the functions σ_1 and σ_2.

In an MLP, the vector function $\sigma_\ell(\cdot)$ at each layer is often *sigmoid-like* such that each component of its output vector ranges in $[0,1]$, and σ_ℓ is more specifically given by $\sigma_\ell:\mathbb{R}^{p_\ell}\to[0,1]^{p_\ell}$. Note that typically the functions $\{\sigma_\ell\}$ have identical operations regardless of their index for $\ell < N$, and are normally applied *element-wise* to their input vector such that each component of $\mathbf{y}_\ell$ depends only on the component of $\mathbf{z}_\ell$ in the same dimension. The typical form for a feedforward layer function with index $\ell < N$ is to take, for some $\mathbf{z} \in \mathbb{R}^{p_\ell}$,

$$\sigma_\ell(\mathbf{z}) = \begin{bmatrix} \frac{1}{1+\exp\{-z_1\}} & \frac{1}{1+\exp\{-z_2\}} & \cdots & \frac{1}{1+\exp\left\{-z_{p_\ell}\right\}} \end{bmatrix}^T, \tag{5.2}$$

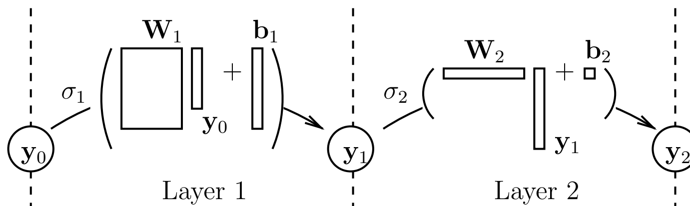


Figure 5.1: Schematic of a feedforward MLP with $N = 2$, where the vertices represent the vectors $\{\mathbf{y}_\ell\}_{\ell=0}^{2}$ in their respective layers. Edges denote computation, where the vector transformations have been drawn with dimensionally consistent block matrices to illustrate sample dimensions of $\mathbf{W}_1$ and $\mathbf{W}_2$.

where z_i is the ith component of $\mathbf{z}$. However, the final output layer function σ_N depends on the particular problem and does not normally have the element-wise restriction as the $\{\sigma_\ell\}_{\ell=1}^{N}$. For instance, for classification of the input vector into one of K classes, σ_N often takes the form of the *softmax* function $S(\mathbf{z})$, which maps a vector $\mathbf{z} \in \mathbb{R}^K$ to a *probability vector* by exponentially projecting its elements as

$$\begin{aligned} S(\mathbf{z}) =& \frac{1}{\sum_i^K \exp\{z_i\}} \begin{bmatrix} \exp\{z_i\} & \dots & \exp\{z_K\} \end{bmatrix}^T \\ =& \frac{\exp\{\mathbf{z}\}}{\|\exp\{\mathbf{z}\}\|_1}, \end{aligned} \tag{5.3}$$

where the shorthand $\exp\{\mathbf{z}\}$ produces a vector of identical size to $\mathbf{z}$ with components equal to the corresponding elements of $\mathbf{z}$ exponentiated. Due to the normalization, $\|S(\mathbf{z})\|_1 = 1$, and furthermore $\|S(\mathbf{z})\|_1$ is *smooth* despite the L_1-norm since the components of $\exp\{\mathbf{z}\}$ are non-negative. As each element of $S(\mathbf{z})$ ranges in $[0, 1]$ and all elements sum to 1, the softmax function is used to express a vector of probabilities where the magnitude of the ith component of $S(\mathbf{z})$ gives the relative probability of the ith class.

5.1.1 Multiclass Classifcation with MLPs

To use the MLP network output $\mathbf{y}_N$ for multiclass classification in a mean loss function $\mathcal{L}(\mathbf{w})$ over a set of observations $\{(\mathbf{x}_r, y_r)\}_{r=1}^{n}$, we

use the *squared loss* metric for the individual loss of a given $(\mathbf{x}_r, y_r)$. The softmax output layer in (5.3) renders each ith component of output vector $\mathbf{y}_N$ as the probability of the observation $\hat{\mathbf{x}}_r$ belonging to the ith class. As such, the scalar $y_r \in \{1, 2, \ldots, K\}$ is taken as a positive integer index, and the component $y_{N,y_r} \in [0, 1]$ is compared to the known probability of 1 for that label.[3] However, since the components $\{y_{N,i}\}_{i \neq y_r}$ are also valid probabilities of the known label being *other* than what it is, this information can also be used in the form of the loss function. Consider the constructed "observed" probability vector $\hat{\mathbf{y}}_r$ for each y_r, where $\hat{\mathbf{y}}_r$ is a *binary* vector nonzero *only* in the component y_r. The $\hat{\mathbf{y}}_r$ vector may then be compared with the MLP's *predicted* value for that r, $\mathbf{y}_N = \mathcal{N}(\mathbf{x}_r)$, where the input observation is taken as $\mathbf{y}_0 = \mathbf{x}_r$ using the squared L_2-norm, $\frac{1}{2}\|\hat{\mathbf{y}}_r - \mathcal{N}(\mathbf{x}_r)\|_2^2$, which is termed the squared loss. The full mean loss function is then

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{r=1}^{n} \frac{1}{2}\|\hat{\mathbf{y}}_r - \mathcal{N}(\mathbf{x}_r)\|_2^2 + \lambda R(\mathbf{w}), \tag{5.4}$$

where the vector $\mathbf{w}$ is defined as the *vectorized* listing of all weight matrix and bias parameters associated with the MLP network function:

$$\mathbf{w} = \begin{bmatrix} \mathbf{b}_1^T & \mathrm{vec}(\mathbf{W}_1)^T & \cdots & \mathbf{b}_N^T & \mathrm{vec}(\mathbf{W}_N)^T \end{bmatrix}^T. \tag{5.5}$$

Here, the vec$(\cdot)$ operator denotes the (column major) concatenation of all matrix elements; thus the dimension of the full $\mathbf{w}$ vector is $\sum_{\ell=1}^{N} p_\ell \cdot (p_{\ell-1} + 1)$.

5.1.2 MLP Gradient Computations

To determine the matrix formulation of the gradient $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{w}}$, expressions are needed for differentiating $\mathcal{L}$ with respect to each of the parameter matrices $\left\{\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_1}, \ldots, \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_N}\right\}$ and bias vectors

[3] The notation y_{N,y_r} is mildly unusual but not inconsistent. In case you're wrinkling your brow, remember that the ith component of the vector $\mathbf{y}_N$ is $y_{N,i}$, where the bold weight is neglected since the ith *element* of a vector is a scalar quantity. The rest follows since $i = y_r$ is a valid positive integer index.

$\left\{\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{b}_1}, \ldots, \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{b}_N}\right\}$. To formally use these quantities, we adopt the matrix calculus notation of Magnus and Neudecker [72] such that meaningful and self-consistent extensions of the chain and product rules from conventional calculus exist. In this convention, the derivative of a matrix $\mathbf{A} \in \mathbb{R}^{r \times s}$ by a matrix $\mathbf{B} \in \mathbb{R}^{p \times q}$ is the $r \cdot s \times p \cdot q$ matrix

$$\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}\mathbf{B}} = \frac{\mathrm{dvec}(\mathbf{A})}{\mathrm{dvec}(\mathbf{B})} = \begin{bmatrix} \frac{\mathrm{d}A_{1,1}}{\mathrm{d}B_{1,1}} & \frac{\mathrm{d}A_{1,1}}{\mathrm{d}B_{2,1}} & \cdots & \frac{\mathrm{d}A_{1,1}}{\mathrm{d}B_{p,q}} \\ \frac{\mathrm{d}A_{2,1}}{\mathrm{d}B_{1,1}} & \frac{\mathrm{d}A_{2,1}}{\mathrm{d}B_{2,1}} & \cdots & \frac{\mathrm{d}A_{2,1}}{\mathrm{d}B_{p,q}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mathrm{d}A_{r,s}}{\mathrm{d}B_{1,1}} & \frac{\mathrm{d}A_{r,s}}{\mathrm{d}B_{2,1}} & \cdots & \frac{\mathrm{d}A_{r,s}}{\mathrm{d}B_{p,q}} \end{bmatrix}, \tag{5.6}$$

such that $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_\ell}$ is represented by a $1 \times (p_{\ell+1} \cdot p_\ell)$ vector.

To obtain $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_\ell}$, the chain rule can be applied to $\mathcal{L}(\mathbf{W})$ and each $\mathbf{y}_\ell, \mathbf{z}_\ell$ pair in (5.1), yielding the product

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_N} \cdot \frac{\partial \mathbf{y}_N}{\partial \mathbf{z}_N} \frac{\partial \mathbf{z}_N}{\partial \mathbf{y}_{N-1}} \cdot \frac{\partial \mathbf{y}_{N-1}}{\partial \mathbf{z}_{N-1}} \frac{\partial \mathbf{z}_{N-1}}{\partial \mathbf{y}_{N-2}} \cdots \frac{\partial \mathbf{y}_\ell}{\partial \mathbf{z}_\ell} \frac{\mathrm{d}\mathbf{z}_\ell}{\mathrm{d}\mathbf{W}_\ell}.$$

Each derivative $\frac{\partial \mathbf{y}_\ell}{\partial \mathbf{z}_\ell}$ is specific to the layer function σ_ℓ, and is denoted by the matrix $\mathbf{\Sigma}_\ell \in \mathbb{R}^{p_\ell \times p_\ell}$, which is diagonal if σ_ℓ is a purely element-wise vector function. From (5.1), the derivative $\frac{\partial \mathbf{z}_\ell}{\partial \mathbf{y}_\ell}$ is merely equal to $\mathbf{W}_\ell$. Finally, the $1 \times (p_\ell \cdot p_{\ell+1})$ derivative $\frac{\mathrm{d}\mathbf{z}_\ell}{\mathrm{d}\mathbf{W}_\ell}$ is given by the kronecker product $\mathbf{y}_{\ell-1} \otimes \mathbf{I}_\ell$. The general formula for the derivatives may be neatly expressed for $1 \leq \ell \leq N$ as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}_\ell} = \mathbf{B}_\ell \cdot \mathbf{\Sigma}_\ell \cdot (\mathbf{y}_{\ell-1} \otimes \mathbf{I}_\ell) = \mathrm{vec}([\mathbf{B}_\ell \cdot \mathbf{\Sigma}_\ell]^T \cdot \mathbf{y}_{\ell-1}^T)^T, \tag{5.7}$$

where $\mathbf{I}_\ell$ is the $p_\ell \times p_\ell$ identity matrix and the *backpropagation* prefactor $\mathbf{B}_\ell \in \mathbb{R}^{1 \times p_\ell}$ is defined through the (backwards) recurrence relation

$$\begin{cases} \mathbf{B}_N &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_N} \\ \mathbf{B}_\ell &= \mathbf{B}_{\ell+1} \cdot \mathbf{\Sigma}_{\ell+1} \cdot \mathbf{W}_{\ell+1}, \quad 1 \leq \ell < N. \end{cases} \tag{5.8}$$

Similarly, the derivative $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{b}_\ell}$ for each ℓ is obtained as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{b}_\ell} = \mathbf{B}_\ell \cdot \mathbf{\Sigma}_\ell. \tag{5.9}$$

Note that the leading backpropagation matrix $\mathbf{B}_N$ depends on the form of $\mathcal{L}(\mathbf{w})$. For instance, for the squared loss, $\frac{1}{2}\frac{\partial}{\partial \mathbf{y}_N}\|\hat{\mathbf{y}} - \mathbf{y}_N\|_2^2 = (\mathbf{y}_N - \hat{\mathbf{y}})^T$. Finally, for σ_N equal to softmax output in (5.3) (with $K = p_N$), the $\mathbf{\Sigma}_N$ matrix is not diagonal, but will depend on the vector $\mathbf{z}_N$ since for a general $\mathbf{z} \in \mathbb{R}^K$,

$$\frac{\mathrm{d}S(\mathbf{z})}{\mathrm{d}\mathbf{z}} = \mathrm{diag}\left([S(\mathbf{z})]\right) - S(\mathbf{z}) \cdot S(\mathbf{z})^T \tag{5.10}$$

where the $\mathrm{diag}(\cdot)$ operator constructs a square diagonal matrix with the elements of its input vector along the diagonal.

From the recurrence relation in (5.8), the procedure of computing the derivatives can be appreciated as being computable via two passes through the layers of the MLP. The first (forward) pass iteratively computes the values of $\sigma_\ell(\mathbf{z}_\ell)$ from (5.1) required for the next layer as well as the elements of the $\mathbf{\Sigma}_\ell$ derivative matrix dependent on $\mathbf{z}_\ell$. The second (backward) pass then uses these stored values to calculate the components of $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{w}}$. This two-pass method for evaluating the output and gradient vector of an MLP is widely known in the neural network community as a procedural component in the *backpropagation algorithm*, which is that community's term for the SGD algorithm.[4]

[4]Please don't misconstrue this comment as being a shot fired in a naming convention turf war—this is just one example of many in which distinct academic communities have developed different nomenclatures. But this particular divergence is way more intriguing when you read the original 1986 Nature letter by Rumelhart al. [93], *Learning representations by back-propagating errors*, which does in fact state in its *body*—but *not* its abstract—that the underlying optimization method is Gradient Descent (cf. Eq. 8 therein). Their follow-up paper is [we opine] even clearer that they describe a procedure for computing a gradient direction in the Gradient Descent algorithm for a particular type of feedforward network function, e.g.: "To minimize E by gradient descent it is necessary to compute the partial derivative of E with respect to each weight in the network" [94]. In light of those papers, and having some slight suspicion that it's [perhaps] more common in academia to cite abstracts rather than *contents* of papers than most academics [including your authors] would like to admit, would it be unseemly to conjecture that this particular divergence in nomenclature could stem entirely from people having systematically *misread* or *not* read the paper they cited?

5.2 PELS Formulation for the MLP

In this section, we formulate the PELS methodology of constructing a degree-d Taylor polynomial approximation for the feedforward layers of the MLP function. We seek the polynomial coefficients of the Taylor expansion in α of $\mathcal{L}(\mathbf{w}+\alpha\boldsymbol{p})$ about the point α_0 in order to solve the univariate line search problem. As before, $\boldsymbol{p}$ is a fixed descent direction defined analogously to $\mathbf{w}$ in (5.5) as a vectorized listing of the matrices $\{\mathbf{P}_\ell\}$, each of which is the search direction for the respective parameter matrix $\mathbf{W}_\ell$ in an iterative optimization algorithm. Our approach determines the expansion coefficients of $\mathcal{L}(\mathbf{w}+\alpha\boldsymbol{p})$ through an iterative application of the chain rule to σ_ℓ at each layer, which in turn uses Taylor expansions of the preceding $\sigma_{\ell-1}$ to supply the derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_{\ell-1}\right\}_n$ needed in the chain rule procedure. Concretely, starting from $\ell=1$ in which $\mathbf{y}_0$ is fixed, we evaluate the derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_1([\mathbf{W}_1+\alpha\mathbf{P}_1]\cdot\mathbf{y}_0)\right\}_{n=1}^d$ at α_0, from which the degree-d Taylor expansion of σ_1 about α_0 is computed. The process then repeats in succession for layers 2 through N, where the chain rule is applied at each σ_ℓ to determine its derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_\ell\right\}_n$ in terms of the derivatives of the previous layer, $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_{\ell-1}\right\}_n$.

The Taylor expansion procedure for a feedforward MLP is presented in several parts. The primary goal is to determine a coefficient matrix $\mathbf{C}_\ell$ for the ℓth layer, where each ith row contains the coefficients in the Taylor expansion for the ith component of that layer's $\mathbf{y}_\ell$. In §5.2.1, the iterative procedure for determining $\mathbf{C}_\ell$ given the previous $\mathbf{C}_{\ell-1}$ matrix is described, where the functions $\{\sigma_\ell\}_{\ell=1}^{N-1}$ are assumed to be element-wise mappings (i.e. $\boldsymbol{\Sigma}_\ell$ is diagonal). §5.2.2 then gives the formulation for a softmax output layer of the network, which has a more complicated treatment due to each element of the output vector $\mathbf{y}_N=\sigma_N(\mathbf{z}_N)$ having wide dependences on each component of $\mathbf{y}_{N-1}$ such that $\boldsymbol{\Sigma}_N$ is non-diagonal. §5.2.3 then finishes with the coefficients for the squared loss output, and §5.2.4 discusses modifications to the PELS algorithm to cope with the MLP's nonconvexity.

To simplify presentation in the remainder of this section, we adopt the following conventions to present the expressions yielding the polynomial coefficients in the Taylor expansions in matrix for-

mat. First, we explicitly consider augmented parameter matrices that *include* the bias as an additional column: $\mathbf{W}_\ell \leftarrow \begin{bmatrix}\mathbf{b}_\ell & \mathbf{W}_\ell\end{bmatrix}$. As such, each $\mathbf{W}_\ell$ and $\mathbf{P}_\ell$ are understood to be $p_\ell \times (p_{\ell-1}+1)$ matrices. Correspondingly, each $\mathbf{y}_\ell$ is also implicitly augmented before its use in matrix-vector operations as $\mathbf{y}_\ell \leftarrow [1\ \mathbf{y}_\ell^T]^T$. We denote the coefficients of the expansion for the ith component of $\mathbf{y}_{\ell-1}(\alpha)$ about α_0 by the vector $\mathbf{c}_i^{[\ell-1]} \in \mathbb{R}^{d+1}$, such that all coefficients can be compactly represented by the coefficient matrix of dimension $(p_{\ell-1}+1) \times (d+1)$ given by

$$\mathbf{C}_{\ell-1} = \begin{bmatrix}\mathbf{e}_1 & \mathbf{c}_1^{[\ell-1]} & \mathbf{c}_2^{[\ell-1]} & \ldots & \mathbf{c}_{p_{\ell-1}}^{[\ell-1]}\end{bmatrix}^T, \tag{5.11}$$

where $\mathbf{e}_1 = \begin{bmatrix}1 & 0 & 0 & \ldots & 0\end{bmatrix}^T \in \mathbb{R}^{d+1}$ accounts for the bias vector $\mathbf{b}_\ell$ incorporated as the first column of the augmented $\mathbf{W}_\ell$ matrix. This arrangement of $\mathbf{C}_{\ell-1}$ aligns coefficients for terms with the same degree in α in each *column* such that the multiplication

$$\mathbf{C}_{\ell-1} \cdot \begin{bmatrix}1 & (\alpha-\alpha_0) & (\alpha-\alpha_0)^2 & \cdots & (\alpha-\alpha_0)^d\end{bmatrix}^T$$

yields approximations to $\mathbf{y}_{\ell-1}(\alpha)$ with $\mathcal{O}\big([\alpha-\alpha_0]^{d+1}\big)$ element-wise truncation error in a neighbourhood of α_0.

5.2.1 Feedforward Layer Polynomial Expansions

We consider the problem of computing the coefficients in a Taylor expansion for $\sigma_\ell([\mathbf{W}_\ell + \alpha\mathbf{P}_\ell] \cdot \mathbf{y}_{\ell-1}(\alpha))$ for a feedforward layer function σ_ℓ that is assumed to be an element-wise vector function, such that the derivative matrix $\mathbf{\Sigma}_\ell$ is diagonal. In this case, it is straightforward to compute the derivatives $\big\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_\ell\big\}_n$ by applying the chain rule to σ_ℓ, and we show that each coefficient vector $\mathbf{c}_i^{[\ell]}$ of the $\mathbf{C}_\ell$ matrix can be expressed as a matrix-vector product of a lower triangular matrix dependent on the derivatives from the previous layer and a vector formed from the nonzero diagonal entries of $\mathbf{\Sigma}_\ell$.

Let the ray $\mathbf{q}_\ell(\alpha) \in \mathbb{R}^{p_{\ell+1}}$ in the parameter search space be defined as

$$\mathbf{q}_\ell(\alpha) = [\mathbf{W}_\ell + \alpha\mathbf{P}_\ell] \cdot \mathbf{y}_{\ell-1}(\alpha), \tag{5.12}$$

where $\mathbf{y}_{\ell-1}(\alpha)$ is the output of the previous layer in the line search, $\sigma_{\ell-1}(\mathbf{q}_{\ell-1}(\alpha))$, that has explicit dependence on α. The nth derivative $\mathbf{q}_\ell^{(n)} = \frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\mathbf{q}_\ell(\alpha)$ can be arithmetically expanded to

$$\mathbf{q}_\ell^{(n)} = (\mathbf{W}_\ell + \alpha\mathbf{P}_\ell) \cdot \mathbf{y}_{\ell-1}^{(n)} + n\mathbf{P}_\ell \cdot \mathbf{y}_{\ell-1}^{(n-1)}, \tag{5.13}$$

where $\mathbf{y}_{\ell-1}^{(n)} = \frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\mathbf{y}_{\ell-1}(\alpha)$. Since $\mathbf{\Sigma}_\ell$ is diagonal, it simplifies the presentation to consider each ith component of $\mathbf{y}_{\ell-1}^{(n)}(\alpha)$ so that scalar calculus may be employed. For a functional composition $f \circ g$ of $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, repeated application of the chain rule to analytically produce the nth derivative $(f \circ g)^{(n)}$ yields an expression that depends only on the individual functions' derivatives $\{f^{(n)}\}_n$ and $\{g^{(n)}\}_n$, for which general formulas exist (see e.g. Huang et al. [57]). By factoring out terms related to the derivatives of f, it is straightforward to form a matrix-vector product that yields the derivatives up to an arbitrary degree n as

$$\begin{bmatrix} (f \circ g) \\ (f \circ g)' \\ (f \circ g)'' \\ (f \circ g)''' \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ 0 & g' & & & \\ 0 & g'' & (g')^2 & & \\ 0 & g''' & 3g'g'' & (g')^3 & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} f \\ f' \\ f'' \\ f''' \\ \vdots \end{bmatrix} \tag{5.14}$$

where all the $\{g, g', \dots\}$ terms are evaluated at a given input x, and each of the $\{f, f', \dots\}$ terms is evaluated at $g(x)$.

With the recipe in (5.14) for the derivatives of a scalar functional composition, the coefficients of a Taylor polynomial expansion for each ith component of $\mathbf{y}_\ell$, can be computed by scaling each $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_\ell(q_{\ell,i})$ by $1/n!$, where $q_{\ell,i}$ is the ith component of $\mathbf{q}_\ell$. A simple expression for $\mathbf{c}_i^{[\ell]}$ is then the matrix-vector product

$$\mathbf{c}_i^{[\ell]} = \mathbf{D}^{-1} \begin{bmatrix} 1 & & & & \\ 0 & q'_{\ell,i} & & & \\ 0 & q''_{\ell,i} & (q'_{\ell,i})^2 & & \\ 0 & q'''_{\ell,i} & 3q'_{\ell,i}q''_{\ell,i} & (q'_{\ell,i})^3 & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \sigma_\ell(q_{\ell,i}) \\ \sigma'_\ell(q_{\ell,i}) \\ \sigma''_\ell(q_{\ell,i}) \\ \sigma'''_\ell(q_{\ell,i}) \\ \vdots \end{bmatrix} \tag{5.15}$$

for $1 \leq i \leq p_\ell$ where $q_{\ell,i}^{(n)}$ denotes the ith component of $\mathbf{q}_\ell^{(n)}$, and $\mathbf{D}$ is a square diagonal scaling matrix of dimension $(d+1)$ in which

the nth diagonal entry is $n!$. Note that in (5.15), both $q_{\ell,i}$ and all its derivatives $\{q_{\ell,i}^{(n)}\}_n$ have been computed at α_0, as in (5.14).

Finally, we note that evaluating (5.13) for each i is straightforward due to the arrangement of the polynomial coefficient matrix in (5.11). If we denote $\mathbf{Q}_\ell$ as $p_\ell \times (d+1)$ block matrix

$$\mathbf{Q}_\ell = \begin{bmatrix} \mathbf{q}_\ell(\alpha_0) & \mathbf{q}_\ell^{(1)}(\alpha_0) & \mathbf{q}_\ell^{(2)}(\alpha_0) & \ldots \mathbf{q}_\ell^{(d)}(\alpha_0) \end{bmatrix}, \tag{5.16}$$

then at each layer $\mathbf{Q}_\ell$ may be computed from the previous $\mathbf{C}_{\ell-1}$ as

$$\mathbf{Q}_\ell = [\mathbf{W}_\ell + \alpha_0 \mathbf{P}_\ell]\mathbf{C}_{\ell-1}\mathbf{D} + \mathbf{P}_\ell \mathbf{C}_{\ell-1}\mathbf{D}\mathbf{G}, \tag{5.17}$$

where $\mathbf{G}$ is an off-diagonal square matrix with dimension $(d+1)$ that accounts for the $n\mathbf{P}_\ell \mathbf{y}_{\ell-1}^{(n-1)}$ term in (5.13) as

$$\mathbf{G} = \begin{bmatrix} 0 & 1 & & & \\ & 0 & 2 & & \\ & & \ddots & \ddots & \\ & & & 0 & d \\ & & & & 0 \end{bmatrix}. \tag{5.18}$$

5.2.2 Softmax Layer Polynomial Expansion

For an output layer σ_N equal to the softmax function in (5.3), the derivative matrix $\mathbf{\Sigma}_N$ is not diagonal, and the simplified treatment in (5.15) cannot be applied. A brute-force approach that attempts to evaluate the derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\mathbf{z}^n} S(\mathbf{z})\right\}_n$ would be computationally infeasible, but is unncecessary since it is instead possible to exploit the structure of softmax function as the functional composition $S(\mathbf{z}) = L(\mathbf{z}) \cdot \exp\{\mathbf{z}\}$ where $L(\mathbf{z}) = \|\exp\{\mathbf{z}\}\|_1^{-1}$, such that the product rule can be applied to $S[\mathbf{q}_N(\alpha)]$ using the derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n} L[\mathbf{q}_N(\alpha)]\right\}_n$ and $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n} \exp[\mathbf{q}_N(\alpha)]\right\}_n$. The advantage of this approach is twofold: $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n} L[\mathbf{q}_N(\alpha)]$ is scalar with a recurrent dependence the computable function $\frac{\mathrm{d}^{n-1}}{\mathrm{d}\alpha^{n-1}} L[\mathbf{q}_N(\alpha)]$, and the function $\exp\{\mathbf{q}_N(\alpha)\}$ is applied element-wise such that its derivatives with respect to α can be expressed by a matrix-vector product similar to (5.15), *mutatis mutandis*.[5]

[5] There is one very important detail in this procedure as described that should be altered in actual implementation. Numerical analysts will be

The recurrence formula for $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n} L[\mathbf{q}_N(\alpha)]$ can be obtained by several applications of the product rule. Recall first that for two functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, the derivatives of the product $f(x) \cdot g(x)$ of an arbitrary order of can be expressed as a binomial sum of the constituent functions' derivatives as

$$\frac{\mathrm{d}^n}{\mathrm{d}x^n}(f \cdot g) = \sum_{m=0}^{n} \binom{n}{m} \frac{\mathrm{d}^m}{\mathrm{d}x^m} f(x) \frac{\mathrm{d}^{n-m}}{\mathrm{d}x^{n-m}} g(x). \tag{5.19}$$

Considering $L[\mathbf{z}(\alpha)]$ for an arbitrary vector $\mathbf{z}(\alpha) \in \mathbb{R}^K$, its (scalar) first derivative is

$$\frac{\mathrm{d}L}{\mathrm{d}\alpha} = -\frac{1}{\|\exp\{\mathbf{z}\}\|_1^2} \exp\{\mathbf{z}(\alpha)\}^T \cdot \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\alpha}. \tag{5.20}$$

To apply the product rule in (5.19), (5.20) may be rewritten as a product of scalar real functions $\psi(\alpha)$ and $\phi(\alpha)$ defined as

$$\begin{cases} \psi(\alpha) & = L[\mathbf{z}(\alpha)]^2 \\ \phi(\alpha) & = -\exp\{\mathbf{z}(\alpha)\}^T \cdot \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\alpha}, \end{cases} \tag{5.21}$$

each of which have easily computable derivatives. Denoting $\mathbf{s} = -\exp\{\mathbf{z}(\alpha)\}$, the derivative $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\mathbf{s} \in \mathbb{R}^K$ can be written element-wise for each ith element s_i by a functional composition matrix as

---

all too aware of the potential numerical instabilities in the exponentials of the softmax function, which result from the finite precision available in a floating point number system (Bryant & O'Hallaron's textbook [17] is a highly recommended reference both for these details and cautionary parables of historical engineering catastrophe, such as missile systems failing due to arithmetic overflow). In particular, for IEEE double precision numbers, $\exp\{x\}$ is indiscernible from the numerical representation of infinity for $x > 720$. The simple way to circumvent errors in the softmax function is to take an equivalent but numerically stable form

$$S'(\mathbf{z}) = \frac{\exp\{\mathbf{z} - z_{\max}\}}{\|\exp\{\mathbf{z} - z_{\max}\}\|_1},$$

where $z_{\max} = \max(\mathbf{z})$. For all of the derivations presented here, practical numerical codes should ensure that any vector argument to the exponential function is suitably modified to prevent overflow.

in (5.14). Once the $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\mathbf{s}\right\}_n$ are computed, the derivatives of $\phi(\alpha)$ then follow immediately as

$$\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\phi(\alpha) = \sum_{m=0}^{n}\binom{n}{m}\left[\frac{\mathrm{d}^m}{\mathrm{d}\alpha^m}\mathbf{s}\right]^T \cdot \left[\frac{\mathrm{d}^{n-m}}{\mathrm{d}\alpha^{n-m}}\mathbf{z}\right] \tag{5.22}$$

Computing the derivatives $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}L$ and $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\psi$ follow from (5.19), and can be evaluated through a simple dynamic programming procedure, since $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}L$ depends on $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\psi$, and $\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\psi$ depends on $\frac{\mathrm{d}^{n-1}}{\mathrm{d}\alpha^{n-1}}L$. The final desired derivatives $\left\{\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n}\sigma_N\right\}_n$ result from one final application of the product rule to $S(\mathbf{z})$ with $\mathbf{z} = \mathbf{q}_N$ such that the $\mathbf{Q}_N$ matrix formed in (5.17) supplies $\mathbf{z}(\alpha)$ and its derivatives evaluated at the expansion point α_0.

5.2.3 Squared Loss Output Polynomial

Approximating the squared loss function in (5.4) by a Taylor polynomial adds one final matrix-vector operation to account for the $\|\hat{\mathbf{y}} - \mathbf{y}_N(\alpha)\|_2^2$ loss term, where $\mathbf{y}_N(\alpha)$ is the output of the network as a function of $\alpha > 0$ along the parameter ray $\mathbf{w} + \alpha\boldsymbol{p}$. The L_2-norm term for results in the summation $\sum_{i=1}^{p_N}(\hat{y}_i - y_{N,i})^2$ for each ith component of $(\hat{\mathbf{y}} - \mathbf{y}_N)$; as such, the loss function expansion coefficient vector $\mathbf{c}_i^{[\mathcal{L}]}$ for each dimension of the difference vector $(\hat{\mathbf{y}} - \mathbf{y}_N)$, once computed, need only be summed over all i to obtain the polynomial approximation to the full loss function $\mathcal{L}(\mathbf{w} + \alpha\boldsymbol{p})$ for a single observation $(\hat{\mathbf{x}}_r, \hat{\mathbf{y}}_r)$.

Since $\|\hat{\mathbf{y}} - \mathbf{y}_N\|_2^2$ is quadratic, the coefficient vectors $\left\{\mathbf{c}_i^{[\mathcal{L}]}\right\}$ result immediately from the functional composition matrix-vector product as in (5.14), given by

$$\mathbf{c}_i^{[\mathcal{L}]} = \mathbf{D}^{-1}\begin{bmatrix} 1 & & & & \\ 0 & y'_{N,i} & & & \\ 0 & y''_{N,i} & (y'_{N,i})^2 & & \\ 0 & y'''_{N,i} & 3y'_{N,i}y''_{N,i} & (y'_{N,i})^3 & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}\begin{bmatrix} \frac{(\hat{y}_i - y_{N,i})^2}{2} \\ y_{N,i} - \hat{y}_i \\ 1 \\ 0 \\ \vdots \end{bmatrix} \tag{5.23}$$

where $\mathbf{D}$ is a square diagonal scaling matrix as before, with dimension p_N.

Example Taylor approximations to $\mathcal{L}(\mathbf{w} + \alpha\boldsymbol{p})$ for the squared loss function in (5.4) using an $N = 4$ feedforward MLP with a softmax output layer is shown in Fig. 5.2a. Here, $\boldsymbol{p}$ is the steepest descent direction, and the expansion point is $\alpha_0 = 1/\|\boldsymbol{p}\|_2$. Taylor polynomial approximations of degree 6 and degree 5 are overlaid to show the agreement for small $(\alpha - \alpha_0)$. Correspondingly, Fig. 5.2b shows both the actual computed percentage error between the polynomial approximation $W_6(\alpha)$ to the loss $\mathcal{L}(\mathbf{w} + \alpha\boldsymbol{p})$, as well as the approximate error computed by the error metric from (3.9) that depends on the difference between degree-6 and degree-5 approximations. It can be seen that the approximate error corresponds well with the actual percentage difference between $W_6(\alpha)$ and $\mathcal{L}(\mathbf{w}+\alpha\boldsymbol{p})$. Fig. 5.2 gives an intuitive sense of how well a polynomial will approximate the loss function along a single dimension; in general the radius $(\alpha - \alpha_0)$ in which the approximation is valid will depend on the degree of curvature around α_0.

5.2.4 Modifications to PELS for Nonconvexity

Recall the univariate line search problem for fixed $\mathbf{w}$ and descent direction $\boldsymbol{p}$:

$$\alpha^* = \arg\min_{\alpha>0} \mathcal{L}(\mathbf{w} + \alpha\mathbf{p}) = \arg\min_{\alpha>0} \phi(\alpha). \tag{5.24}$$

The PELS line search in Alg. 4 operated by iteratively forming the Taylor polynomial model $W(\alpha; \mathbf{w}, \boldsymbol{p})|_{\alpha_j}$ approximating $\phi(\alpha)$ in the neighbourhood of α_j and determining each successive step size through the minimization of the model,

$$\alpha_{j+1} = \arg\min_{\alpha>0} W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}. \tag{5.25}$$

While in previous chapters we approached (5.24) with better behaved functions, the MLP is less amenable to simple techniques due to its nonlinearity. For instance, NR iteration as applied previously would be hard-pressed to succeed for the very flat regions of $\mathcal{L}(\mathbf{w} + \alpha\boldsymbol{p})$ in Fig. 5.2a in which the Hessian is close to zero within the limits of numerical representation. As such, to solve (5.25) for general non-convex functions, we modify the PELS algorithm to include the interpolation technique from the WA class of

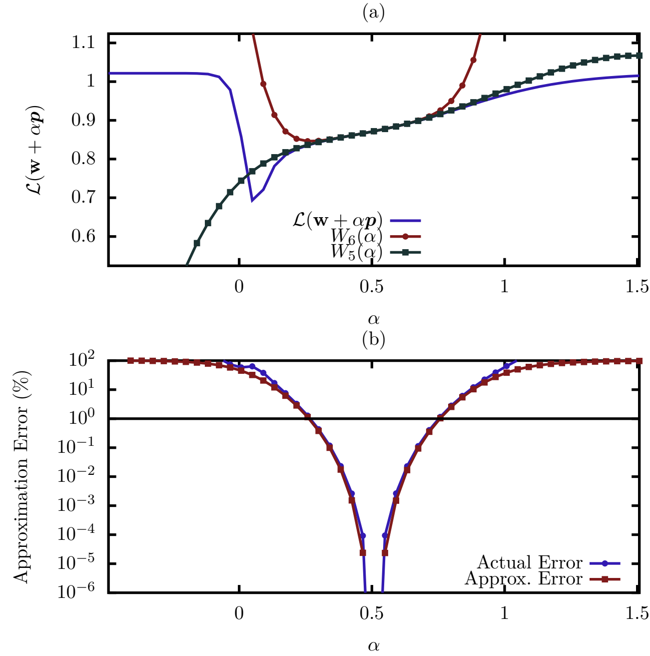


Figure 5.2: *(a) Example degree-6 polynomial approximation in α about $\alpha_0 = 1/\|\boldsymbol{p}\|_2$, where $\boldsymbol{p}$ is steepest descent direction of a mean squared loss function with a sample sigmoidal MLP and nonzero regularization for several randomly sampled instances from the* IRIS *dataset. Note that the sharpness at the minimum near $\alpha = 0$ is an artifact due to the number of gridpoints used to compute $\mathcal{L}(\mathbf{w}+\alpha\boldsymbol{p})$. In (b) the percentage error of the approximation relative to the actual network output as computed by the metric in (3.9) is shown on a logarithmic scale.*

line searches. This addition is used to recover from minimization failures for polynomials constructed with bad initial guesses for the step size in regions where NR iteration would diverge. Furthermore, it benefits naturally from the error metric previously proposed,

$$\epsilon(\alpha)|_{\alpha_j} = \left|\frac{c_d(\alpha-\alpha_j)^d}{W(\alpha)|_{\alpha_j}}\right|, \tag{5.26}$$

which may be used to determine a *trust radius* about α_j to bound a feasible solution to (5.25).

The cubic interpolating polynomial constructed in WA line searches uses the pairs of control points $\phi(\alpha_j)$ and $\phi'(\alpha_j)$, and $\phi(\alpha_{j-1})$ and $\phi'(\alpha_{j-1})$. Since the PELS Taylor coefficients of up to degree d at the bias points α_j and α_{j-1} are *exact*, they may be used to construct interpolating polynomials with higher degrees than cubic. Suppose that we wish to create an interpolant $P_{a,b}(\alpha)$ for the interval $[a, b]$ (which could be the steps $[\alpha_{j-1}, \alpha_j]$) and have evaluated the coefficients $\{c_a, i\}_{i=0}^{n_a}$ and $\{c_b, i\}_{i=0}^{n_b}$ of Taylor polynomials biased about a and b, respectively, with $1 \leq n_a, n_b \leq d$. The coefficients $\{c_{I,i}\}_{i=0}^{M}$ of the interpolant polynomial $P_{a,b}$ with degree $M = n_a + n_b + 1$ are then determined by constructing an augmented matrix system of two smaller Vandermonde-type systems. Specifically, we first take the coefficients $\{c_{a,i}\}_i$. The first (under-determined) Vandermonde-type linear system is

$$\underbrace{\mathbf{V}_M(a)}_{n_a \times (M+1)} \cdot \begin{bmatrix} c_{I,M} \\ c_{I,M-1} \\ \vdots \\ c_{I,1} \\ c_{I,0} \end{bmatrix} = \begin{bmatrix} c_{a,0} \\ c_{a,1} \\ \vdots \\ c_{a,n_a-1} \\ c_{a,n_a} \end{bmatrix} \tag{5.27}$$

where $\mathbf{V}_M(a) \in \mathbb{R}^{n_a \times (M+1)}$ is a Vandermonde-type matrix of a particular form. In $\mathbf{V}_M(a)$, each nth row corresponds to the interpolation equation for the nth derivative,

$$\frac{\mathrm{d}^n}{\mathrm{d}\alpha^n} \left(\sum_{m=0}^{M} c_{I,m} \alpha^m \right) \Bigg|_{\alpha=a} = c_{a,n}, \tag{5.28}$$

and is ordered such that each mth column aligns with the coefficient $c_{I,m}$ in (5.27). Thus we have that

$$\mathbf{V}_M(a) = \begin{bmatrix} a^M & a^{M-1} & \dots & a^2 & a & 1 \\ Ma^{M-1} & (M-1)a^{M-2} & \dots & 2a & 1 & 0 \\ \vdots & \vdots & \iddots & \iddots & \vdots & \vdots \end{bmatrix}. \tag{5.29}$$

The full (soluble) linear system for $\{c_{I,i}\}_i$ is then determined by

the solution of the $(M+1) \times (M+1)$ augmented system

$$\begin{bmatrix} \mathbf{V}_M(a) \\ \text{---} \\ \mathbf{V}_M(b) \end{bmatrix} \begin{bmatrix} c_{I,M} \\ \vdots \\ c_{I,0} \end{bmatrix} = \begin{bmatrix} \mathbf{c}_a \\ \text{---} \\ \mathbf{c}_b \end{bmatrix}, \tag{5.30}$$

where $\mathbf{c}_a = [c_{a,0}\ c_{a,1}\ \ \ldots, c_{a,n_a}]^T$ and $\mathbf{c}_b = [c_{b,0}\ c_{b,1}\ \ \ldots, c_{b,n_b}]^T$. Note that for $n_a = n_b = 1$, (5.30) may be solved for the coefficients of a cubic interpolant using function and gradient information at the points a and b. Allowing for $n_a, n_b > 1$ gives greater representation potential for the curvature of the interpolant, which may help in cases where the MLP function changes radically. The case $n_a \neq n_b$ is also permitted by this construction, and is useful since the known values $\phi(0)$ and $\phi'(0)$ may be combined with the degree-d information computed by PELS at α_0 in an interpolant P_{0,α_0} if a minimum to $W(\alpha)|_{\alpha_0}$ cannot be computed.[6]

The error metric $\epsilon(\alpha)|_{\alpha_j}$ in (5.26) has one further use: it is possible to solve for values of α at which the error metric is equal to a desired approximate fractional error μ. By inverting the error metric formula, a step size $\bar{\alpha}_j$ near α_j with a given μ may be found as one of the relevant roots of the equation

$$\left| 1 + \sum_{m=0}^{d-1} \frac{c_m}{c_d} (\alpha - \alpha_j)^{m-d} \right| - \frac{1}{\theta} = 0. \tag{5.31}$$

By *relevant*, we mean that the desired solution to (5.31) lies in the direction of descent for the expansionj point α_j (i.e. left if

[6]Another reason for which n_a may not equal n_b is one of numerical stability. (5.30) can only be solved when the columns of the Vandermonde matrix are sufficiently independent, but raising small values of a or b to a large power may leave the system insoluble. Hence it is necessary to verify that the solution is well conditioned, and *decrease* the number of coefficients in the vectors $\mathbf{c}_a$ and $\mathbf{c}_b$ otherwise. In the worst case, the cubic interpolation can be constructed, since this particular interpolant always exists and is unique [85]. In future references to (5.30), please mentally insert a note that in the PELS code, solving this linear system included a procedure that iteratively solved for $[c_{I,0}\ \ \ldots\ \ c_{I,M}]^T$ with decreasing M until the interpolant values at the control points differed from the known values by less than 10^{-9} (or until $M = 3$).

$\phi'(\alpha_j) > 0$ and right if $\phi'(\alpha_j) < 0$) since a lower value of $\phi(\alpha)$ than $\phi(\alpha_j)$ should be bracketed in the interval $(\alpha_j, \bar{\alpha}_j]$ or $[\bar{\alpha}_j, \alpha_j)$. Note that the roots of (5.31) were computed through bisection to avoid computing the large derivatives near α_j or vanishing derivatives as $\alpha \to \pm\infty$ for numerical robustness.[7]

With these two tools in hand, the PELS method is adapted as follows. After having computed the coefficients of $W(\alpha)|_{\alpha_j}$, we continue to use NR iteration to attempt a solution for α_{j+1} as in (5.25), since the fast convergence properties of NR iteration remain desirable, if possible. However, if the routine is unable to compute a valid solution α_{j+1}, taken as having achieved decrease in the polynomial value as $W(\alpha_{j+1})|_{\alpha_j} < W(\alpha_j)|_{\alpha_j}$ (in addition to standard numerical convergence critera as used previously in NR iteration; our implementation required that the iterative change to the step size be less than 10^{-14} within 50 iterations), then a high degree interpolant $P_{a,b}$ is constructed and minimized on a bounded interval, with bounds determined according to (5.31). The values of $[a, b]$

[7]Solving for the roots of either $\epsilon(\alpha)|_{\alpha_j} - \theta = 0$ or the inverted equation $\frac{1}{\epsilon(\alpha)|_{\alpha_j}} - \frac{1}{\theta} = 0$ should in principle be a six-in-one-hand, half-a-dozen-in-the-other type of question. We have not implemented the former and cannot comment on its robustness or practicalities, since in our initial design phase of the procedure, the latter equation seemed more appealing for the absence of pesky potential zeros in the denominator. Plus, it has the guarantee that $\frac{1}{\epsilon(\alpha)|_{\alpha_j}}$ diverges at α_j, entailing that that a numerical bisection procedure seeking zero on the right (say) of α_j, can bracket a zero pretty dependably [barring numerical representation] by starting from the lefthand value $\alpha_j + \mu_0$ with a tiny μ_0, and increasing that value to find a right bracket with a function value of opposite sign. This routine has worked reliably for us for a heuristic reason that we attribute to the numerical values of the coefficients $\{c_m\}$ in (5.31): we have observed $|c_d| \ll |c_{d-1}| \ll |c_{d-2}| < \ldots$ for the high order coefficients (no general inequality is claimed to relate c_0, c_1, and c_2). Thus the $\frac{1}{\epsilon\alpha|_{\alpha_j}}$ functions in our test problems increased or decreased monotonically in practice insofar as we could tell. Also note for completeness that our bisection implementation used the convergence criteria that either (i) the difference between the midpoint and endpoints was less than 10^{-8} or (ii) the machine epsilon for the midpoint was greater than 10^{-8} within 2000 iterations.

are updated in each iteration as in a conventional WA line search; initially $[a, b] = [0, \alpha_0]$, where $\phi(0)$ and $\phi'(0)$ are known. Once α_1 is found by either the minimization of $W(\alpha)|_{\alpha_0}$ or $P_{0,\alpha_0}(\alpha)$, the interval $[a, b]$ is updated to be either $[0, \alpha_1]$ if $|\alpha_1 - 0| \leq |\alpha_1 - \alpha_0|$ or $[\alpha_1, \alpha_2]$ otherwise (and ordered such that $a < b$). In each iteration, the interval I is determined from the relevant root of (5.31) for the current values of a and b (and where the initial left bound of the interval is 0). This procedure repeat is repeated iteratively until the error metric $\epsilon(\alpha_{j+1})|_{\alpha_{\text{last}}}$ for the iterate α_{j+1} is sufficiently small, where α_{last} is the closest to α_{j+1} of the latest expansion points stored in a or b (however, note that $\alpha = 0$ is never used as α_{last} since the PELS coefficients are not computed for this step size along the direction p under consideration). This procedure is summarized in Alg. 8, but note that several details discussed in the above paragraph are omitted from the summary for simplicity of presentation.

Alg. 8 has many similarities to the standard methodology for performing WA line searches with cubic interpolants in each iteration, which underlines that the PELS methodology of computing Taylor coefficients for conducting a univariate line search, rather than evaluating $\phi(\alpha)$ directly, is not at all incompatible with other techniques. For instance, enforcing the Wolfe conditions could be added as a criterion for convergence at the solution point in addition to the polynomial error metric, where the Wolfe conditions could be evaluated exactly at the expansion point of a Taylor polynomial or approximated using that polynomial's coefficients. The work by Hager & Zhang [48] may interest the reader as an example from the literature of *approximate* Wolfe conditions. In it they proffer a reformulation of the Wolfe sufficient decrease condition that depends on the gradient ϕ' as opposed to ϕ, but can be evaluated more accurately due to numerical representation errors. Other optimization researchers or practitioners may have related use cases in which a procedure as described here lends itself naturally.

5.3 PELS Feasibility Tests

To investigate the feasibility of the PELS method in batch optimization algorithms for training parameters in feedforward mul-

Algorithm 8: PELS for the MLP

Input: $\phi(0) \in \mathbb{R}, \phi'(0) < 0$, $\alpha_0 > 0$, $\theta > 0$, $\mu > 0$
Output: $\alpha_j \approx \alpha^* = \arg\min_{\alpha>0} \phi(\alpha)$

1 $[a, b] \leftarrow [0, \alpha_0]$
2 $\mathbf{c}_a \leftarrow [\phi(0)\ \phi'(0)]^T$
3 $j \leftarrow 0$
repeat
4 Compute $\{c_m\}_{m=0}^{d}$ for $W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}$
5 $\alpha_j^{\mathrm{NR}} \leftarrow \arg\min_{\alpha>0} W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}$
6 **if** α^{NR} *is valid* **then**
7 $\alpha_{j+1} \leftarrow \alpha_j^{\mathrm{NR}}$
else
8 Compute I from (5.31) on a and b with tolerance μ
9 Construct $P_{a,b}(\alpha)$
10 $\alpha_{j+1} \leftarrow \arg\min_{\alpha \in I} P_{a,b}(\alpha)$
11 Update and order $[a, b]$ to best bound α_{j+1}
12 Update $\mathbf{c}_a, \mathbf{c}_b$ with coefficients corresponding to $[a, b]$
13 $\alpha_{\mathrm{last}} \leftarrow \min(|\alpha_{j+1} - a|, |\alpha_{j+1} - b|)$
14 $j \leftarrow j + 1$
15 **until** $\epsilon(\alpha_j)|_{\alpha_{last}} \leq \theta$
16 **return** α_j

tilayer perceptron models, a comparison was made of the NCG and LBFGS using Alg. 8 or the cubic interpolating line search by Moré & Thuente [80] for univariate line searches. As before, the implementations using PELS are denoted by a suffix *-P* as NCG-P and LBFGS-P. While our focus in previous chapters has been on parallel implementations, our current PELS codebase is an unoptimized serial prototype; as such, the experiments were conducted on small binary and multiclass classification datasets that are standard and readily available online for prototyping and testing in the machine learning community. All results pertain to mean squared loss functions over labelled datasets as in (5.4) without regularization ($\lambda = 0$), where the predicted probability vectors were generated by multi-layer MLP functions with softmax output layers.

The PELS algorithm was implemented in the interpreted Oc-

tave language [37] in conjunction with the matrix-based MLP implementation formulated in §5.1 and run with $\theta = 10^{-8}$ and $\mu = 5 \times 10^{-2}$ (which entails approximately 5% error). The bound-constrained minimization problem in step 10 of Alg. 8 was performed using Octave's univariate `fminbnd` function with an input tolerance of 10^{-14} and a maximum of 250 iterations. The LBFGS and NCG algorithms were also implemented in Octave with a user-specifiable line search routine such that the LBFGS and LBFGS-P, and NCG and NCG-P algorithms had identical operations besides their univariate line searches, respectively. The NCG implementations used the positive Polak-Ribière update formula from (1.5), and the initial step size in each line search invocation was chosen through the scaling formula (1.6), except for restarts in some kth iteration, in which case the initial step was taken as $1/\|\mathbf{g}_k\|_2$. Furthermore, a Powell restart condition of $\nu = 0.9$ was used in (1.7). The LBFGS routine used the typical settings of $n_c = 5$ corrections, an initial step size of 1 in the line search, and Liu and Nocedal's M3 scaling [70] in the Hessian approximation. The Wolfe conditions in the WA algorithm were $\nu_1 = 10^{-4}$ and $\nu_2 = 0.9$. Lastly, a maximum of 20 iterations was allowed for both the WA and PELS algorithms in all experiments.

The datasets considered were: the IRIS [38] sepal and petal measurements for $K = 3$ classes of irises, the PARKINSON'S dataset [69] with 23 clinical voice measurements for binary classification of patients with Parkinson's disease, and an 500-image subset of the MNIST [64] dataset of vectorized pixels from handwritten postal code digits with $K = 10$ classes. For MNIST, the subset was sampled randomly and uniformly from all available images in the full set. The properties of these datasets and the layer dimensions of their respective MLPs classification functions are summarized in Tab. 5.1. Selecting the dimensions of the MLP functions is undeniably black magic, but the dimensions in Tab. 5.1 were chosen *ad hoc* such that there were sufficiently many parameters to drive $\mathcal{L}(\mathbf{w}_k)$ to zero for large k, as well as to minimic the relative dimensional ratios previously reported for MLPs for the MNIST dataset [64]. The observations vectors $\{\mathbf{x}_r\}_{r=1}^n$ in the IRIS and PARKINSON'S datasets were preprocessed such that the mean and variance of the values $\{x_{r,i}\}_{i=1}^m$ for each ith dimension were 0 and 1, respectively (this process is termed *standardization* in machine learning). The subset

Table 5.1: Summary of classification datasets and MLP model properties. Here, n gives the number of observations in the dataset, K gives the number of classes, and N gives the number of layers in the MLP function evaluated through (5.1). The dimensions of the weight matrices are shown, separated by hyphens, where each distinct digit corresponds to the number of rows in the subsequent layer. For example, the PARKINSON'S dataset had 2 layers with weight matrices $\mathbf{W}_1 \in \mathbb{R}^{23\times 32}$ and $\mathbf{W}_2 \in \mathbb{R}^{32\times 2}$, with dimensions denoted 23–32–2.

	n	K	N	Dimensions of $\{\mathbf{W}_\ell\}$
PARKINSON'S [69]	197	23	2	23–32–2
IRIS [38]	150	3	4	4–16–8–3
MNIST [64]	500	10	3	591–256–64–10

of the MNIST handwritten digit dataset was preprocessed slightly differently. Instead of standardizing the vectors, the dimensions for which *each* observation in the dataset had zero-valued entries were removed, which appreciably reduced the dimension of the input vectors since the images are very sparse. The nonzero pixel values contained in the images were otherwise unchanged. Finally, the initial values of $\mathbf{w}_0$ for all algorithms and experiments were drawn pseudorandomly from the standard unit normal distribution.

5.4 Results & Discussion

The convergence traces of $\mathcal{L}(\mathbf{w}_k)$ for the NCG-P and LBFGS-P algorithms are presented both as a function of iterations taken and the number of dataset reads (i.e. cumulative evaluations of $\mathcal{L}(\mathbf{w}_k)$ and $\nabla\mathcal{L}(\mathbf{w}_k)$). The two tales allow the feasibility of the PELS method to be considered in view of both the achievable convergence speedup, but also the cost in terms of passes over the dataset.[8] Note also that the *raw* loss function values are shown,

[8]A nagging question you could ask: why not also show the elapsed clock time for the algorithms as an alternate [and arguably more palpa-

rather than traces of $|\mathcal{L}(\mathbf{w}_k) - \mathcal{L}(\mathbf{w}^*)|$ since in the nonconvex neural network landscape, disparate solutions were computed by each algorithm (this was also verified numerically; each algorithm converged to a different minimum, most likely a local one [24]).

The loss traces for the IRIS, PARKINSON'S, and MNIST datasets are depicted in Fig. 5.3, Fig. 5.4, and Fig. 5.5, respectively. For each figure, subplot (a) shows the number of iterations along the horizontal axis, while the ordinates of subplot (b) show the number of cumulative dataset reads. As a general remark on the potential convergence improvements, the NCG-P algorithm performs substantially better on these example datasets than its counterpart, which is a similar result as found in §3 for logistic regression—in particular, the trace in Fig. 5.5a has comparable convergence to LBFGS. The LBFGS-P method exhibits slightly faster convergence on these problems than LBFGS, but not to the extent previously observed for logistic regression. To quantify this, the speedup factors in terms of the number of iterations required to reach a given value of $\mathcal{L}(\mathbf{w}_k)$ were computed and are shown in Fig. 5.6, where Fig. 5.6a shows the factors for NCG-P over NCG, and Fig. 5.6b shows those for LBFGS-P over LBFGS. The acceleration factors

ble] metric for efficiency? The answer is partly that the current implementation in Octave began as a feasibility study in a graduate project with the expedient goal of correctly implementing and investigating whether PELS improved convergence behaviour, and is sufficiently slow that plots with elapsed time in the horizontal axis would not be much to look at. However, the more important important detail not to be glossed over is that by showing either iterations or dataset passes, all the additional work required in to compute the Taylor coefficients in §5.2 has been *discounted*. The tacit assumption that the extra work is somehow inherently negligible would be wrong, and we do not wish to ignore that. While it may have been the case in §3, where the logistic regression Taylor coefficients in (3.11) took only a few extra flops after having evaluated the inner product terms, it is not necessarily the case for the MLP, which has far more dense linear algebra to do. Ultimately, just bear in mind that the convergence behaviour measured in iterations is an upper bound on what could be done if two methods took the same amount of time, and that for parallel systems the communications costs for PELS will be lesser. Beyond that, your mileage may vary based on how expensive communication is by comparison with dataset reads and CPU-bound computations.

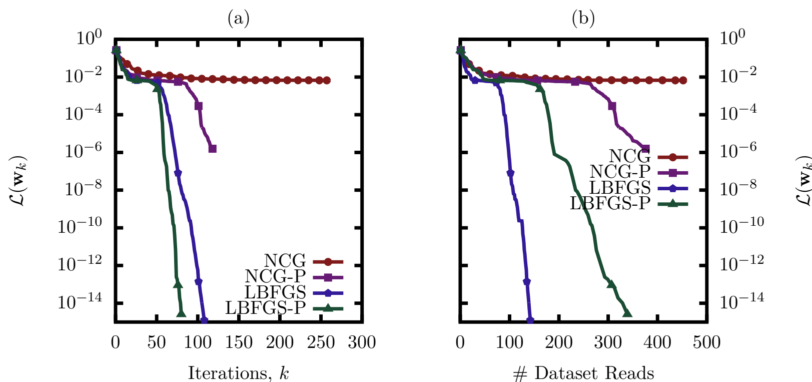


Figure 5.3: *Convergence traces compared between PELS and the cubic interpolating WA line search from Moré & Thuente for the* IRIS *dataset.*

for LBFGS-P fall mostly in the range of 1.1–1.3× for the traces on the datasets considered, while NCG-P can require fewer than half as many iterations as NCG. However, the loss traces dependent on the cumulative passes through the dataset also reveal that several additional function and gradient evaluations were made in each line search invocation. For instance, though in Fig. 5.5a only about 125 iterations of LBFGS-P were performed, over 400 coefficient or function/gradient evaluations were required in Fig. 5.5b, which is approximately 2.2 per line search (cf. that number with the values of $n_e \approx 1$ in Tab. 3.3). This motivates a discussion of whether or not the convergence gains for LBFGS-P over LBFGS justify further research on a large-scale system: does the additional work required in the PELS line search for the MLP make the improvements a pyrrhic victory?

Back in the parallel performance evaluation of PELS in §3, we saw that the distributed Apache Spark framework for large-scale processing had significant communication costs. Hence optimizing a logistic regression model in Spark was exemplary of a case in which the number of passes over the dataset was less important than the subsequent communication of the result of the computations on the data [evinced by how the end-to-end time for evaluating the Taylor coefficients in §3.5 was only about a quarter of the time needed to

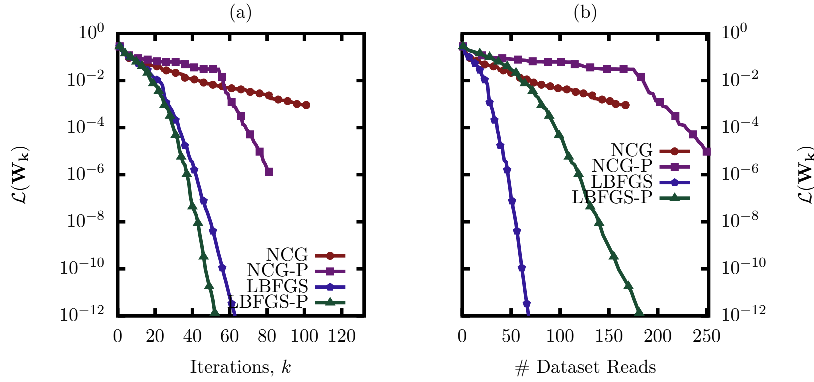


Figure 5.4: *Convergence traces compared between PELS and the cubic interpolating WA line search from Moré & Thuente for the* PARKINSON'S *dataset.*

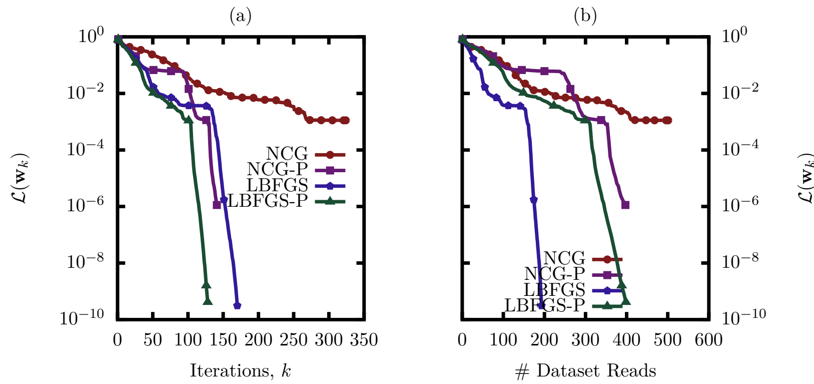


Figure 5.5: *Convergence traces compared between PELS and the cubic interpolating WA line search from Moré & Thuente for the* MNIST *dataset.*

evaluate the full gradient vectors]. The convergence enhancements produced by computing more accurate minima in the univariate line search invocations were thus realizable without significant additional work. Serial settings or systems without expensive communication overheads are the obverse: unless the convergence gains are sufficient to justify the extra work, doubling the time per iter-

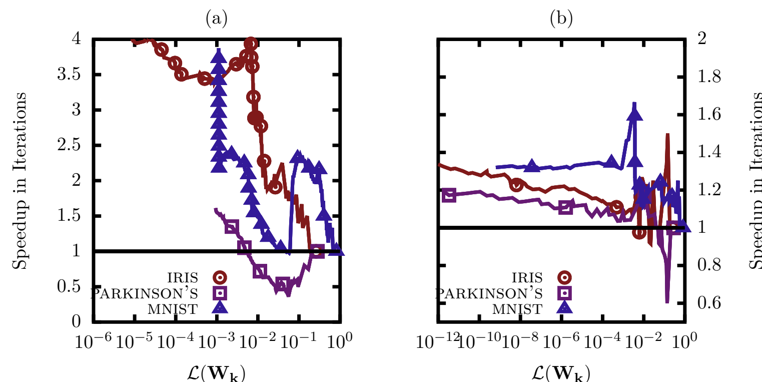


Figure 5.6: *Speedup factors in iterations required to reach a given value of $\mathcal{L}(\mathbf{w}_k)$, as computed for (a) NCG-P over NCG and (b) LBFGS-P over LBFGS on the datasets in Tab. 5.1.*

ation with additional passes over the dataset will only slow things down. Further experimentation with optimized implementations of methodology outlined here in systems such as Spark in which the gains could be substantial would be a better determinant of the applicability of PELS in this setting, particularly given our initial results showing that NCG can be substantially improved.

5.5 Conclusion

In this chapter, we have presented an extension of the Polynomial Expansion Line Search algorithm for nonconvex objective functions. One adaptation included the construction of polynomial interpolants constructed from the previously computed Taylor coefficients in the PELS line search procedure, similar to the cubic interpolation performed by classical Wolfe Approximate line searches. The second addition to the method was to use the polynomial error metric previously defined to compute trust intervals in which the interpolant was minimized in a bound-constrained minimization procedure. A matrix formulation of the multilayer perceptron ANNmodel was presented, as well as a derivation of its Taylor expansion coefficients replete with a procedure for their recursive computation. model. This derivation specifically considered multiclass

classification via a softmax output layer and mean squared loss error function. We implemented a serial prototype of PELS and conducted several initial performance comparisons with a cubic interpolating WA algorithm for the NCG and LBFGS algorithms for optimizing MLP parameters. On the small datasets considered, it was found that NCG-P could obtain speedup factors greater than 2 over NCG in the number of iterations required to minimize the loss function to a given order of magnitude. The speedup factors for LBFGS-P over LBFGS were approximately between 1.1–1.3 for the same problems. However, the increased number of passes through the dataset per line search invocation entail that the feasibility of PELS for accelerating LBFGS is less straightforward than for NCG, and it is thought likely that the communication costs of the particular system using a parallel implementation of PELS in the optimization of ANN parameters will determine the technique's suitability.

6

Conclusion

Now begins the fat lady's song. This thesis has considered the problem of how to efficiently conduct univariate line searches in commodity clusters in the context of gradient-based batch optimization algorithms. In it, the Polynomial Expansion Line Search (PELS) technique was proposed, which is a line search technique for cases where the underlying objective function is analytic, as in logistic regression and low rank matrix factorization. In this method we approximate the objective function by a truncated Taylor polynomial along a fixed search direction, and compute the Taylor coefficients for the expansion explicitly along that direction. The coefficients of this polynomial may be computed efficiently in parallel with far less communication than needed to transmit the high-dimensional gradient vector, after which the polynomial may be minimized locally by a driver program with high accuracy in a neighbourhood of the expansion point without further distributed operations.

Three applications of the PELS technique were presented for important classes of analytic functions: (i) logistic regression, (ii) low-rank matrix factorization models, and (iii) feedforward multilayer perceptrons (MLPs). Furthermore, the PELS implementa-

tions for logistic regression and matrix factorizations were presented in the Apache Spark framework for fault-tolerant cluster computing, where significant convergence enhancements were achievable using the technique. For instance, in training large-scale logistic regression models with the LBFGS algorithm, the number of iterations and time required to reach terminal training accuracies for was reduced by factors of 1.8–2. The PELS technique was also used as an integral component of a nonlinearly preconditioned Nonlinear Conjugate Gradient algorithm for solving the matrix factorization problem as formulated for recommender systems, for which the speedup over the conventional Alternating Least Squares algorithm reached 3–5 for obtaining accurate solutions to the optimization problem. Substantial acceleration was also observed for the Nonlinear Conjugate Gradient algorithm for MLP models, and may be of interest as a future tool for optimizing neural networks models.

The Polynomial Expansion Line Search technique as presented herein has been applicable to several models for Big Data processsing and large-scale optimization, and has been a useful component of batch optimization routines in certain circumstances. However, it is not necessarily a blanket solution: we have presented instances for MLP models in which it has had only modest gains or even hampered convergence for the LBFGS algorithm due to the extra computational work required. As such, the method presented is not a hammer, but a screwdriver[1] that has specific but invaluable use cases. At any rate, we release PELS into the wild and hope that it may find its way into researchers' and practitioners' numerical toolboxes for their own optimization problems.

[1] Possibly in the vein of a degree-d generalization of those strange double- and triple- square screw heads formed by the superposition of rotated concentric Robertson squares. That sufficiently large d would reduce such a screw head to a futile blind hole is also not lost on us.

References

[1] A. Acerbi, V. Lampos, P. Garnett, and R. A. Bentley. The expression of emotions in 20th century books. *PlOS ONE*, 8(3):e59030, 2013.

[2] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Nature*, 369:40, 1994.

[3] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[4] M. Al-Baali and R. Fletcher. An efficient line search for nonlinear least squares. *Journal of Optimization Theory and Applications*, 48(3):359–377, 1986.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[6] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[7] R. M. Bell and Y. Koren. Lessons from the Netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, 2007.

[8] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[9] R. Bird. *Introduction to functional programming using Haskell*. Prentice Hall, 1998.

[10] R. S. Bird. *An introduction to the theory of lists*. Springer, 1987.

[11] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[12] D. Borthakur. The Hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(1):21, 2007.

[13] L. Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):25, 1998.

[14] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics*, pages 177–186. Springer, 2010.

[15] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[16] O. Bousquet and L. Bottou. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[17] R. Bryant and D. O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2003.

[18] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[19] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, 134(1):127–155, 2012.

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

[21] F. Cardone and J. R. Hindley. Lambda-calculus and combinators in the 20th century. In D. M. Gabbay and J. Woods, editors, *Handbook of the History of Logic: Logic from Russell to Church*, volume 5, pages 723–817. Elsevier, 2009.

[22] S. Chakravorti and W. R. Emmons. Who pays for credit cards? *Journal of Consumer Affairs*, 37(2):208–230, 2003.

[23] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[24] A. Choromanska, M. Henaff, M. Mathieu, G. Ben Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 192–204, 2015.

[25] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[26] J. Dai. Experience and lessons learned for large-scale graph analysis using GraphX. Spark Summit East, 2015.

[27] Y.-H. Dai. Convergence properties of the BFGS algoritm. *SIAM Journal on Optimization*, 13(3):693–701, 2002.

[28] B. Dalessandro, D. Chen, T. Raeder, C. Perlich, M. Han Williams, and F. Provost. Scalable hands-free transfer learning for online advertising. In *Proceedings of the Special Interest Groupd on Knowledge Discovery and Data Mining*, pages 1573–1582. ACM, ACM, 2014.

[29] A. Davidson and A. Or. Optimizing shuffle performance in Spark. *UC Berkeley, Dept Elec Eng & Comp Sci*, 2013.

[30] H. De Sterck and M. Winlaw. A nonlinearly preconditioned conjugate gradient algorithm for rank-R canonical tensor approximation. *Numerical Linear Algebra with Applications*, 22:410–432, 2014.

[31] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[32] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[33] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research*, 13(1):165–202, 2012.

[34] J. E. Dennis, Jr and J. J. Moré. Quasi-Newton methods, motivation and theory. *SIAM Review*, 19(1):46–89, 1977.

[35] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! music dataset and KDD-cup '11. In *Proceedings of the Special Interest Group on Knowledge Discovery and Data Mining*, volume 18, pages 3–18. Journal of Machine Learning Research, 2012.

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

[37] J. W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. SoHo Books, 2007.

[38] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*[2], 7(2):179–188, 1936.

[2] Yes, *the* R. A. Fisher. And also, yes, *the* Eugenics of Galton [and Pearson and Fisher]. The early statisticians were all pretty sold on the idea of a thoroughbred human stock. The Annals of Eugenics, first established by Pearson in 1925, was rechristened the Annals of *Human Genetics* in 1954, which you may or may not find alarmingly recent.

[39] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964.

[40] S. H. Fuller, L. I. Millett, et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011.

[41] S. Funk. Netflix update: Try this at home. Blog Post, 2006. `http://sifter.org/~simon/journal/20061211.html`.

[42] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the Special Interest Group on Knowledge Discovery and Data Mining*, pages 69–77. ACM, 2011.

[43] C. Gerlitz and A. Helmond. The like economy: Social buttons and the data-intensive web. *New Media & Society*, 15(8):1348–1365, 2013.

[44] J. C. Gilbert and J. Nocedal. Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization*, 2(1):21–42, 1992.

[45] A. Graves, A.-R. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 6645–6649. IEEE, 2013.

[46] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[47] W. Hager. A derivative-based bracketing scheme for univariate minimization and the conjugate gradient method. *Computers and Mathematics with Applications*, 18(9):779–795, 1989.

[48] W. Hager and H. Zhang. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on Optimization*, 16(1):170–192, 2005.

[49] W. W. Hager and H. Zhang. A survey of nonlinear conjugate gradient methods. *Pacific Journal of Optimization*, 2(1):35–58, 2006.

[50] D. Hall. Breeze: numerical processing library for Scala. `https://github.com/scalanlp/breeze`.

[51] F. M. Harper and J. A. Konstan. The MovieLens datasets: history and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):19, 2016.

[52] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2001.

[53] G. Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[54] G. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[55] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, and P. D. Ravikumar. QUIC: quadratic approximation for sparse inverse covariance estimation. *Journal of Machine Learning Research*, 15(1):2911–2947, 2014.

[56] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Proceedings of the International Conference on Data Mining*, pages 263–272. IEEE, 2008.

[57] H. Huang, S. Marcantognini, and N. Young. Chain rules for higher derivatives. *The Mathematical Intelligencer*, 28(2):61–69, 2006.

[58] M. B. Hynes and H. De Sterck. A polynomial expansion line search for large-scale unconstrained minimization of smooth L2-regularized loss functions, with implementation in Apache Spark. In *Proceedings of the International Conference on Data Mining*, pages 603–611. SIAM, 2015.

[59] C. Johnson. Music recommendations at scale with Spark. Spark Summit, 2014.

[60] C. C. Johnson. Logistic matrix factorization for implicit feedback data. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[61] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM, 2008.

[62] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[63] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[64] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[65] D.-H. Li and M. Fukushima. A modified BFGS method and its global convergence in nonconvex minimization. *Journal of Computational and Applied Mathematics*, 129(1):15–35, 2001.

[66] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region Newton method for logistic regression. *Journal of Machine Learning Research*, 9:627–650, 2008.

[67] J. Lin and A. Kolcz. Large-scale machine learning at twitter. In *Proceedings of the Special Interest Group on the Management of Data*, pages 793–804. ACM, 2012.

[68] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Comput.*, 7(1):76–80, 2003.

[69] M. A. Little, P. E. McSharry, S. J. Roberts, D. A. Costello, and I. M. Moroz. Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *BioMedical Engineering OnLine*, 6(1):1, 2007.

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

[71] J. Ma, R. P. Sheridan, A. Liaw, G. E. Dahl, and V. Svetnik. Deep neural nets as a method for quantitative structure-activity relationships. *Journal of Chemical Information and Modeling*, 55(2):263–274, 2015.

[72] J. R. Magnus and H. Neudecker. Matrix differential calculus with applications to simple, Hadamard, and Kronecker products. *Journal of Mathematical Psychology*, 29(4):474–492, 1985.

[73] M. Mahsereci and P. Hennig. Probabilistic line searches for stochastic optimization. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, pages 181–189. Curran Associates, Inc., 2015.

[74] J. Martens. Deep learning via Hessian-free optimization. In *Proceedings of the International Conference on Machine Learning*, volume 27, pages 735–742. Journal of Machine Learning Research, 2010.

[75] J. McCarthy. History of LISP. In *History of programming languages I*, pages 173–185. ACM, 1978.

[76] L. Meertens. Algorithmics–towards programming as a mathematical activity. *Mathematics and Computer Science*, 1, 1986.

[77] L. Meertens. Constructing a calculus of programs. In *Mathematics of Program Construction*, pages 66–90. Springer, 1989.

[78] J. Millward. Deep inside: A study of 10,000 porn stars and their careers. Available: `http://jonmillward.com/blog/studies/deep-inside-a-study-of-10000-porn-stars/`, 2013.

[79] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[80] J. J. Moré and D. J. Thuente. Line search algorithms with guaranteed sufficient decrease. *Transactions on Mathematical Software*, 20(3):286–307, 1994.

[81] K. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

[82] I. Navon and D. M. Legler. Conjugate gradient methods for large-scale minimization in meteorology. *Monthly Weather Review*, 115(8):1479–1502, 1987.

[83] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the International Conference on Machine Learning*, volume 28, pages 265–272. Journal of Machine Learning Research, 2011.

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

[85] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[86] K. Ousterhout. Apache Spark shuffle internals. `https://cwiki.apache.org/confluence/display/SPARK/Shuffle+Internals`, 2015.

[87] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 17–29, 2007.

[88] B. T. Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4):94–112, 1969.

[89] M. J. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12(1):241–254, 1977.

[90] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[91] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[92] C. Rudder. *Dataclysm: Who We Are (When We Think No One's Looking).* Random House Incorporated, 2014.

[93] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[94] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3):213–219, 1988.

[95] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, pages 285–295, 2001.

[96] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. In *Proceedings of the International Conference on Machine Learning*, volume 28, pages 343–351. Journal of Machine Learning Research, 2013.

[97] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[98] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. In *Proceedings of the Special Interest Group on Measurement and Evaluation*. ACM, 2009.

[99] J. Sohl-Dickstein, B. Poole, and S. Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods. In *Proceedings of the International Conference on Machine Learning*, pages 604–612. Journal of Machine Learning Research, 2014.

[100] E. Tarone, S. Dwyer, S. Gillette, and V. Icke. On the use of the passive and active voice in astrophysics journal papers: With extensions to other languages and other fields. *English for Specific Purposes*, 17(1):113–132, 1998.

[101] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *Proceedings of the International Conference on Data Mining*, volume 12, pages 655–664. IEEE, 2012.

[102] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[103] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-posed problems*. John Wiley, New York, 1977.

[104] P. Tseng and S. Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Mathematical Programming*, 117(1-2):387–423, 2009.

[105] K. Vonnegut. *Palm Sunday: An Autobiographical Collage*. Random House Publishing Group, 2009.

[106] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2015.

[107] M. Winlaw. *Algorithms and Models for Tensors and Networks with Applications in Data Science*. PhD thesis, University of Waterloo, 2016. `https://uwspace.uwaterloo.ca/handle/10012/10158`.

[108] M. Winlaw, M. B. Hynes, A. Caterini, and H. De Sterck. Algorithmic acceleration of parallel ALS for Collaborative Filtering: Speeding up distributed Big Data recommendation in Spark. In *Proceedings of the International Conference on Parallel and Distributed Systems*, volume 21, pages 69–79, 2015.

[109] B. Yavuz and X. Meng. Performance improvements in MLlib. `https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html`, 2014.

[110] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 765–774. IEEE, 2012.

[111] G.-X. Yuan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. A comparison of optimization methods and software for large-scale L_1-regularized linear classification. *Journal of Machine Learning Research*, 11:3183–3234, 2010.

[112] G.-X. Yuan, C.-H. Ho, and C.-J. Lin. Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9):2584–2603, 2012.

[113] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the European Conference on Computer Systems*, pages 265–278. ACM, 2010.

[114] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the International Conference on Networked Systems Design & Implementation*, pages 15–39. USENIX, 2012.

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

[116] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the International Conference on Algorithmic Aspects in Information and Management*, volume 4, pages 337–348, 2008.

[117] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *Transactions on Mathematical Software*, 23(4):550–560, 1997.

[118] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances In Neural Information Processing Systems*, pages 2595–2603, 2010.

Appendices

A

Himrod Cluster Architecture

The `himrod` computing cluster is composed of 1 head node, 23 compute nodes, 4 *large* compute nodes with additional memory, and 1 storage node hosting a network filesystem. The head node is the entry point to the cluster via an `ssh` tunnel. Our experiements here have been conducted with the Spark master and driver programs on the large compute nodes; hence we refer to these nodes (homogenous) specifications as the *master* node. All nodes are 64 bit rack servers (PowerEdge R620; Dell, Round Rock, TX, US) running Ubuntu 14.04.2 LTS with linux kernel 3.13.0-54-generic, compiled with symmetric multiprocessing. The compute nodes all have two processors, both 8-core 2.6 GHz chips with 20M onboard cache (Xeon E5-2670; Intel, Santa Clara, CA, US) and 16 registered DIMMs with 16 GB of DDR3 SDRAM (M393B2G70BH0-YK0; Samsung, Seoul, KR) for a total of 256 GB of memory. The single storage node (PowerEdge R720; Dell) contains two 2 GHz processors, each with 6 cores (Xeon E5-2620; Intel), 64GB of memory, with 8 DIMMs, each slot providing 8 GB of DDR SDRAM (Hynix; Icheon, RK) and 12 hard disk drives of 4 TB capacity and 7,200 RPM nominal speed. The master node [i.e., any *large* compute node] differs from the [standard] compute

Table A.1: Processor information for each node in the himrod cluster. Each chip is a member of the Intel Xeon family.

	Master	Slave	Storage
Chip	E5-2670	E5-2670	E5-2620
Quantity	2	2	2
Speed (GHz)	2.6	2.6	2.0
Cores	8	8	6
Threads	16	16	12

Table A.2: Memory information for each node in the himrod cluster. The speeds given are the configured chip speeds as determined by `dmidecode`.

	Master	Slave	Storage
Chip	M393B4G70BM0-YH9	M393B2G70BH0-YK0	HMT31GR7EFR4A-PB
Quantity	16	16	8
Size (MB)	32,768	16,384	8,192
Speed (MHz)	1,333	1,600	1,600

nodes only by its onboard memory chips, which have 32 GB capacity (M393B4G70BM0-YH9; Samsung) making for 512 GB of total memory. The processor and RAM information for each node in the cluster is summarized in Tab. A.1 and Tab. A.2, respectively. Furthermore, the RAM chipset serial numbers are provided in Tab. A.2.

Each compute node's root filesystem is mounted on a local solid state drive with a read speed of 6 Gb/s (LB206M; SanDisk, Milpitas, CA, US) and formatted with an ext4 filesystem. For local storage, compute nodes also have 6 ext4-formatted SCSI 10,000 RPM hard disk drives, each with a 600 GB capacity. The network storage system hosted by the storage node is mounted on both the head and compute nodes via the `automount` daemon, and nodes are interconnected by a 10 Gb ethernet managed switch (PowerConnect 8164; Dell).

To determine the above hardware information, several standard methods were used. Firstly, basic information about the storage devices was taken from the nodes' local `/proc` directories. Specifically, `/proc/scsi/scsi` contained basic information about the vendor

and model of available SCSI [pronounced *scuzzy*] devices. For more detailed information about the hardware, the `lshw` program was invoked as:

```
$ sudo lshw -class disk -class storage
```

The processor and memory specifications were obtained via the `dmidecode` program, called as

```
$ sudo dmidecode --type processor
```

for the CPU listings, and

```
$ sudo dmidecode --type 17
```

for RAM information.