# Efficient Quasi-Newton Methods in Trust-Region Frameworks for Training Deep Neural Networks

Mahsa Yousefi

## Abstract

Deep Learning (DL), utilizing Deep Neural Networks (DNNs), has gained significant popularity in Machine Learning (ML) due to its wide range of applications in various domains. DL applications typically involve large-scale, highly nonlinear, and non-convex optimization problems. The objective of these optimization problems, often expressed as a finite-sum function, is to minimize the overall prediction error by optimizing the parameters of the neural network. In order to solve a DL optimization problem, interpreted as DNN training, stochastic second-order methods have recently attracted much attention. These methods leverage curvature information from the objective function and employ practical subsampling schemes to approximately evaluate the objective function and its gradient using random subsets of the available (training) data. Within this context, active research is focused on exploring strategies based on Quasi-Newton methods within both line-search and trust-region optimization frameworks. A trust-region approach is often preferred over the former one due to its ability to make progress even when some iterates are rejected, as well as its compatibility with both positive definite and indefinite Hessian approximations. Considering Quasi-Newton Hessian approximations, the thesis studies two classes of second-order trust-region methods in stochastic expansions for training DNNs as follows. In the class of standard trust-region methods, we consider well-known limited memory Quasi-Newton Hessian matrices, namely L-BFGS and L-SR1, and apply a half-overlapping subsampling for computations. We present an extensive experimental study on the resulting methods, discussing the effect of various factors on the training of different DNNs and filling a gap regarding which method yields more effective training. Then, we present a modified L-BFGS trust-region method by introducing a simple modification to the secant condition, which enhances the curvature information of the objective function, and extend it in a stochastic setting for training tasks. Finally, we devise a novel stochastic method that combines a trust-region L-SR1 second-order direction with a first-order variance-reduced stochastic gradient. Our focus in the second class is to develop standard trust-region methods for both non-monotone and stochastic expansions. Using regular fixed sample size subsampling, we investigate the efficiency of a non-monotone L-SR1 trust-region method in training through different approaches for computing the curvature information. We eventually propose a non-monotone trust-region algorithm that involves an additional sampling strategy in order to control the resulting error in function and gradient approximations due to subsampling. This novel method enjoys an adaptive sample size procedure and achieves almost sure convergence under standard assumptions. The efficiency of the algorithms presented in this study, implemented in MATLAB, is assessed by training different DNNs to solve specific problems such as image recognition and regression, and comparing their performance to well-known first- and second-order methods, including Adam and STORM.

# Acknowledgments

It is with great appreciation that I acknowledge Prof. Ángeles Martínez Calomardo, my supervisor, for all her unwavering support and encouragement throughout this journey. I hold a greater sense of gratitude towards her for all maternal favors which meant to me a lot.

I'd like to extend my gratitude to Prof. Nataša Krejić and Prof. Nataša Krklec Jerinkić for their expertise, professionalism, and dedication to the work and our collaboration. They made my visiting program at the University of Novi Sad an unforgettable success. Moreover, I wish to appreciate the late Prof. Daniela di Serafino and Dr. Marco Viola for their valuable contributions.

My sincere appreciation goes out to the director of the DMG, Prof. Daniele del Santo, and the coordinator of the Ph.D. program, Prof. Stefano Maset. I wish to convey my genuine heartfelt thanks to Prof. Paolo Novati; I am always more grateful to him than he will ever know. I am also deeply thankful to Sir. Piero Falconer, the technician of the DMG, for his kind help and delightful friendship.

It is my immense pleasure to have many lovely cousins and friends. Their presence in ups and downs, and their understanding mean the world to me. My forever thanks go out to my two closest friends, Elham and Marzieh.

I am truly blessed to have an open-minded and supportive family that sincerely embraces my educational pursuits. I am at a loss for words to fully express my utmost gratitude to my beloved ones for their lasting love and love.

Mahsa Yousefi

Trieste, May 2023

# Contents

**List of Figures** v

**List of Tables** vii

**List of Algorithms** viii

**Notation** x

**1 Introduction** 1

**2 Basic Background on Unconstrained Optimization** 7
2.1 Unconstrained Optimization Problem . . . 7
2.2 Line-Search and Trust-Region Strategies . . . 10

**3 An Overview on Optimization in Deep Learning** 21
3.1 Deep Neural Networks . . . 22
3.2 Deep Learning Optimization Problem . . . 29
3.3 Deep Learning Optimization Strategies . . . 30
3.4 Experimental Setups . . . 36

**4 Stochastic Trust-Region Methods** 40
4.1 Stochastic Quasi-Newton TR Algorithms . . . 40
4.1.1 Stochastic Limited-Memory BFGS TR . . . 42
4.1.2 Stochastic Limited-Memory SR1 TR . . . 44
4.1.3 Numerical Comparison . . . 48
4.2 A Stochastic Modified L-BFGS Trust-Region Method . . . 67
4.2.1 A modified L-BFGS update . . . 67
4.2.2 Algorithm Framework . . . 70
4.2.3 Numerical Evaluation . . . 71
4.3 A Stochastic Hybrid L-SR1 Trust-Region Method . . . 75
4.3.1 Algorithm Framework . . . 75
4.3.2 Numerical Evaluation . . . 79

**5 Stochastic Non-Monotone Trust-Region Methods** 82
5.1 Introduction . . . 82
5.2 A Stochastic Algorithm with Fixed-Size Sampling . . . 83
5.2.1 Algorithm Framework . . . 85
5.2.2 Numerical Evaluation . . . 89
5.3 A Stochastic Algorithm with Adaptive Sampling . . . 93
5.3.1 Algorithmic framework . . . 94

5.3.2 Convergence Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.3.3 Numerical Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 106

**6 Concluding Remarks 114**

**A Programming Comments 118**

**B Two Solvers for the TR Subproblem 126**

**C Additional Algorithms 132**

**D Overlap Batching and Computations 137**

**E Additional Experiments 139**

# List of Figures

3.1 An artificial neural network with one hidden layer. . . . . . . . . . . . . . . 23
3.2 Convolution operation using a kernel of size 2 and stride size 2. . . . . . . 26
3.3 A residual building block. . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

4.1 A schematic of the fixed-size half-overlapping scheme within one epoch. . 48
4.2 The effect of the limited memory parameter on sL-QN-TR with CIFAR10. . . . . 54
4.3 The accuracy of sL-QN-TR on MNIST with LeNet-like. . . . . . . . . . . . . . 55
4.4 The accuracy of sL-QN-TR on Fashion-MNIST with LeNet-like. . . . . . . . . 55
4.5 The accuracy of sL-QN-TR on Fashion-MNIST with ResNet-20. . . . . . . . . . 55
4.6 The accuracy of sL-QN-TR on Fashion-MNIST with ResNet-20(no BN). . . . . 56
4.7 The accuracy of sL-QN-TR on CIFAR10 with ResNet-20. . . . . . . . . . . . . 56
4.8 The accuracy of sL-QN-TR on CIFAR10 with ResNet-20(no BN). . . . . . . . . 56
4.9 The accuracy of sL-QN-TR on MNIST with ConvNet3FC2. . . . . . . . . . . . . 57
4.10 The accuracy of sL-QN-TR on MNIST with ConvNet3FC2(no BN). . . . . . . . . 57
4.11 The accuracy of sL-QN-TR on Fashion-MNIST with ConvNet3FC2. . . . . . . . 57
4.12 The accuracy of sL-QN-TR on Fashion-MNIST with ConvNet3FC2(no BN). . . . 58
4.13 The accuracy of sL-QN-TR on CIFAR10 with ConvNet3FC2. . . . . . . . . . . . 58
4.14 The accuracy of sL-QN-TR on CIFAR10 with ConvNet3FC2(no BN). . . . . . . . 58
4.15 The accuracy of sL-QN-TR vs time on MNIST with LeNet-like. . . . . . . . . 61
4.16 The accuracy of sL-QN-TR vs time on Fashion-MNIST with LeNet-like. . . . . 61
4.17 The accuracy of sL-QN-TR vs time on CIFAR10 with ConvNet3FC2. . . . . . . . 62
4.18 The accuracy of sL-QN-TR vs time on CIFAR10 with ConvNet3FC2(no BN). . . 62
4.19 Comparisons of sL-QN-TR and STORM (CIFAR10, ConvNet3FC2). . . . . . . . 63
4.20 Comparisons of sL-QN-TR and *tuned* Adam (CIFAR10, ConvNet3FC2). . . . . . 63
4.21 Comparisons of sL-QN-TR and STORM (Fashion-MNIST, ResNet-20). . . . . . 64
4.22 Comparisons of sL-QN-TR and *tuned* Adam (CIFAR10, ConvNet3FC2(no BN)). . 64
4.23 Comparisons of sL-QN-TR and *tuned* Adam (Fashion-MNIST, ResNet20). . . . 65
4.24 Comparisons of sL-QN-TR and *tuned* Adam (Fashion-MNIST, ResNet20(no BN)). 65
4.25 Comparisons of sL-QN-TR and *tuned* Adam (MNIST, LeNet-like). . . . . . . . 65
4.26 The comparative behavior of sM-LBFGS-TR (MNIST, LeNet-like). . . . . . . . 72
4.27 The comparative behavior of sM-LBFGS-TR (CIFAR10, ConvNet3FC2). . . . . . 73
4.28 Error bars of sM-LBFGS-TR, sL-BFGS-TR and *tuned* Adam. . . . . . . . . . . 74
4.29 The comparative behavior of sM-LBFGS-TR vs CPU time. . . . . . . . . . . . 74
4.30 The effect of L2 regularization parameter on the training of Adam . . . . . . . 80
4.31 The effect of L2 regularization parameter on the training of sCLSR1-TR . . . . 81
4.32 The accuracy vs iteration evolution of sCLSR1-TR and *tuned* Adam . . . . . . 81
4.33 The accuracy vs time evolution of sCLSR1-TR and *tuned* Adam . . . . . . . . 81

5.1 The effect of regularization over testing accuracy of sL-SR1-NTR ($bs = 500$). . . 89

5.2 The impact of curvature computing approaches of **pHv+Fv** types. . . . . . . . 90
5.3 The impact of different curvature computing approaches. . . . . . . . . . . . . 91
5.4 The comparative accuracy of sL-SR1-NTR using **Fv** vs GPU Time (3 hours). . 92
5.5 Comparative testing accuracy of sL-SR1-NTR with different curvature approaches. 92
5.6 The comparative accuracy sL-SR1-NTR using **Fv** vs iteration (3 hours). . . . . 93
5.7 The accuracy variations of STORM and ASNTR on `MNIST`. . . . . . . . . . 108
5.8 The accuracy variations of STORM and ASNTR on `Cifar10`. . . . . . . . . 108
5.9 The accuracy variations of STORM and ASNTR on `DIGITS`. . . . . . . . . . 108
5.10 The loss variation of STORM and ASNTR on `MNIST`. . . . . . . . . . . . . . 109
5.11 The loss variation of STORM and ASNTR on `CIFAR10`. . . . . . . . . . . . 109
5.12 The loss variation of STORM and ASNTR on `DIGITS`. . . . . . . . . . . . . 109
5.13 Tracking subsampling in ASNTR. . . . . . . . . . . . . . . . . . . . . . . . . 111
5.14 Batch size progress with initial `rng(42)`. . . . . . . . . . . . . . . . . . . . 113

A.1 Properties in the MATLAB object `dlNet`. . . . . . . . . . . . . . . . . . . . 121

D.1 An example of the overlapping batch formation within 2 epochs. . . . . . 138

E.1 `MNIST`, `LeNet-like`: The accuracy and loss evolution vs epoch. . . . . . . 140
E.2 `F-MNIST`, `LeNet-like`: The accuracy and loss evolution vs epoch. . . . . . 141
E.3 `F-MNIST`, `ResNet-20`: The accuracy and loss evolution vs epoch. . . . . . . 142
E.4 `CIFAR10`, `ResNet-20`: The accuracy and loss evolution vs epoch. . . . . . . 143
E.5 `F-MNIST`, `ResNet-20(no BN)`: The accuracy and loss evolution vs epoch. . 144
E.6 `CIFAR10`, `ResNet-20(no BN)`: The accuracy and loss evolution vs epoch. . 145
E.7 `MNIST`, `ConvNet3FC2`: The accuracy and loss evolution vs epoch. . . . . . . 146
E.8 `F-MNIST`, `ConvNet3FC2`: The accuracy and loss evolution vs epoch. . . . . 147
E.9 `CIFAR10`, `ConvNet3FC2`: The accuracy and loss evolution vs epoch. . . . . 148
E.10 `MNIST`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch. . 149
E.11 `F-MNIST`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch. 150
E.12 `CIFAR10`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch. 151
E.13 `MNIST` and `F-MNIST`: The accuracy evolution vs CPU time. . . . . . . . . . 152
E.14 `CIFAR10`: The accuracy evolution vs CPU time. . . . . . . . . . . . . . . . . 153
E.15 `MNIST`, `LeNet-like`: Comparison with *tuned* Adam. . . . . . . . . . . . . . 154
E.16 `F-MNIST`, `ResNet-20`: Comparison with *tuned* Adam. . . . . . . . . . . . . 155
E.17 `F-MNIST`, `ResNet-20(no BN)`: `ResNet-20`: Comparison with *tuned* Adam. . 156
E.18 `CIFAR10`, `ConvNet3FC2`: Comparison with *tuned* Adam. . . . . . . . . . . . 157
E.19 `CIFAR10`, `ConvNet3FC2(no BN)`: Comparison with *tuned* Adam. . . . . . . 158

# List of Tables

3.1 The details of the Networks. . . . . 39

4.1 The total number of networks' trainable parameters ($n$). . . . . 51
4.2 Summary of the best sL-QN-TR approaches for classification problems. . . . . 53

5.1 Experimental configuration of sL-SR1-NTR. . . . . 88
5.2 Hyper-parameters of STORM and ASNTR. . . . . 106

E.1 Set of figures for image classification problems. . . . . 139

# List of Algorithms

1 Backtracking . . . . . 16
2 DNNs training . . . . . 25
3 sL-BFGS-TR . . . . . 49
4 sL-SR1-TR . . . . . 50
5 sM-LBFGS-TR . . . . . 69
6 sCLSR1-TR . . . . . 78
7 sL-SR1-NTR . . . . . 84
8 ASNTR . . . . . 97
C.1 Trust-Region radius adjustment . . . . . 132
C.2 L-BFGS Hessian initialization . . . . . 133
C.3 Orthonormal Basis BFGS (OBB) . . . . . 133
C.4 L-SR1 Hessian initialization . . . . . 134
C.5 Orthonormal Basis SR1 (OBS) . . . . . 134
C.6 sL-BFGS-TR (regular) . . . . . 135
C.7 sL-SR1-TR (regular) . . . . . 136

# Notation

Throughout this thesis, vectors are typically represented by lowercase letters, while matrices are represented by uppercase letters unless otherwise specified in the context. To facilitate the description of certain concepts in this thesis, a summary of the notations used is provided below.

| **Symbols** | **Descriptions** |
|---|---|
| $=,\ \neq,\ \approx,\ \triangleq,\ \infty$ | equals, is not equal, is approximately, is defined by, infinity |
| $>,\ <$ | is greater than, is less than |
| $\geq,\ \leq$ | is greater than or equal, is less than or equal |
| $\phi'(\sigma)$ and $\phi''(\sigma)$ | derivative and second derivative of $\phi$ with respect to $\sigma$ |
| $\frac{\partial f}{\partial w}$ and $\frac{\partial^2 f}{\partial w^2}$ | partial and second partial derivatives of $f$ with respect to $w$ |
| $\nabla f,\ \nabla^2 f$ | gradient of $f$, Hessian of $f$ |
| $\to,\ \to a^+,$ lim, lim inf | approaches, approaches to $a$ from the right, limit, infimum limit |
| lim, lim inf | limit, infimum limit |
| $\perp,\ \square,\ \vec{}$ | perpendicular, (box) end of proof, vector sign as in $\vec{0}$ |
| $\mathbb{N},\ \mathbb{R},\ \emptyset$ | set of natural number, set of real number, the empty set |
| $\mathbb{R}^n$ | set of real column vector of dimension $n$ |
| $\mathbb{R}^{n\times m}$ | set of real matrix with $n$ rows and $m$ columns |
| $A^{-1},\ A^T,\ I$ | the inverse of matrix $A$, the transpose of $A$, the Identity matrix |
| $\mathrm{diag}(d_1,\ldots,d_m)$ | the diagonal matrix with elements $d_1,\ldots,d_m$ |
| $\{.\},\ \in,\ \notin$ | set or sequence, is member of, is not member of |
| $\subset,\ \subseteq,\ \cup,\ \cap$ | is proper subset of, is subset of, union, intersection |

( To be continued ... )

| Symbols | Descriptions |
|---|---|
| $a_i$ | indicates the $i$th element of the vector $a$ or the sequence $\{a_k\}_{k=1}^{\infty}$ |
| $J_{:,i}$ | the $i$th column of the matrix $J$ |
| $\|.\|_2,\ \|.\|_F$ | $L^2$ (Euclidean) norm of a vector, Frobenius norm of a matrix |
| $\lvert . \rvert$ | is the cardinality of a set or the absolute value of a number |
| $\mathbb{E}(x)$ | mathematical expectation of a random variable $x$ |
| $\mathbb{E}(x \mid \mathcal{F})$ | conditional expectation of the random variable $x$ given a $\sigma$-algebra $\mathcal{F}$ |
| $a.s$ | abbreviates the expression *"almost surely"* |

# 1

# Introduction

Deep Learning (DL), using Deep Neural Networks (DNNs) [32], is a prominent technique in Machine Learning (ML). It finds extensive application across various domains, including speech recognition, natural language processing, image recognition, image restoration, partial differential equations, advertising, bioinformatics, and drug discovery. In DL applications (e.g., image recognition, which is the main focus of this thesis), an optimization problem is encountered: $\min_{w \in \mathbb{R}^n} f(w)$, where the objective function $f(w)$ is typically expressed as a finite sum of $N$ loss functions, i.e., $f(w) = \sum_{i=1}^{N} f_i(w)$, corresponding to $N$ sample pairs with inputs $x_i$ (e.g., images) and targets $y_i$ (e.g., true classes). Each loss function, denoted as $f_i(w)$, quantifies the prediction error between the network's output $h(x_i; w)$ and the corresponding target $y_i$, given by the function $L$, i.e., $f_i(w) = L(h(x_i; w), y_i)$. Minimizing the total prediction error for the network's parameter vector $w$ is necessary to train a DNN and enable accurate predictions. This process is commonly referred to as the *training task*. While the architecture of networks and computational resources play crucial roles in determining the effectiveness of DNNs, there is an opportunity to enhance efficiency through improvements in the methodology employed for their training tasks. This is particularly challenging due to the inherent characteristics of large-scale DL problems, which exhibit high nonlinearity and non-convexity, making the application of traditional optimization strategies for the minimization nontrivial. Computing the true gradient with respect to the entire set of $N$ samples can be computationally expensive. Hence, many optimization strategies in-

troduced for ML and DL problems utilize stochastic extensions. For example, references such as [13, 20] describe a range of deterministic and stochastic methods employed in solving these problems. In stochastic extensions, a practical subsampling scheme is utilized to evaluate the objective function and its gradient by considering a *random* subset (mini-batch) of training data at each iteration. The prevailing DL optimization strategies primarily rely on first-order methods, such as stochastic gradient descent and its variants, due to their low per-iteration computational cost and ease of implementation. Nevertheless, these methods are not without their limitations. For instance, they may struggle to escape saddle points and often require exhaustive trial and error to fine-tune hyper-parameters, such as step length. As a result, stochastic second-order methods have received significant attention recently, aiming to address some of these shortcomings.

Second-order methods utilize the Hessian matrix or its approximations to incorporate the curvature of the objective function, allowing for potentially faster convergence. Among these methods, the Newton method is well-known, as it involves the exact calculation of the Hessian. However, in DL applications with millions of parameters, the computational time and memory storage required for this exact calculation can be extremely prohibitive. To address these limitations, several appealing alternatives have been developed to incorporate second-order information. For instance, Hessian-Free methods aim to estimate the Newton direction by computing Hessian-vector products, bypassing the direct calculation of the Hessian matrix. Another approach is the use of limited memory Quasi-Newton methods, which construct approximations of the true Hessian solely based on gradient information. To apply these methods in large-scale DL applications, one can either utilize the complete training set, fully leveraging modern computational architectures or work with a mini-batch of the data to define the objective function and its gradient. Looking towards (stochastic) Hessian-Free and (stochastic) Quasi-Newton methods is an active area of research. For this thesis, we have chosen to focus on the latter ones, while keeping the other methods for future study.

Regardless of whether one is working in a stochastic or deterministic setting, second-order methods for DL optimization are typically considered within two fundamental frameworks: line-search and trust-region [62]. Each framework provides a guideline for

determining an appropriate search direction to move from the current parameter values to the next. Most line-search methods require a descent direction and, consequently, a positive definite curvature matrix per iteration. However, in DL optimization with a non-convex objective function, the true Hessian may not be positive definite. A second-order trust-region framework can accommodate the Hessian or its approximations, whether positive definite or not. Moreover, based on the distinguishing characteristics of trust-region algorithms, unlike line-search methods, the progress of the learning/training will not stop or slow down due to the occasional rejection of the undesired search directions. More precisely, the parameter updating process using line-search methods is stopped if the necessary conditions are not met, but it continues using trust-region methods once the trust-region radius has been modified. These reasons have primarily motivated our choice to utilize trust-region frameworks in this thesis study. Moreover, it is recognized that non-monotonicity can have potential benefits by relaxing the typical requirement of monotonic decreasing conditions in the objective function values; see e.g. [2]. This relaxation allows for local increases in the values without significantly impacting the convergence properties. In the context of stochastic extensions used in DL approaches, we believe that it is also not desirable to impose strict decrease conditions on the approximate (subsampled) function since it serves as an estimation of the true objective function. Hence, we explore both standard and non-monotone trust-region regimes in the present study.

In light of the foregoing, this thesis studies stochastic Quasi-Newton trust-region methods for training DNNs in large-scale DL applications. Specifically, by leveraging different subsampling strategies and incorporating limited memory Quasi-Newton Hessian approximations, we describe several second-order trust-region algorithms in both standard and non-monotone regimes for training several DNNs in image classification and regression problems. The research findings can be also found in [45, 81, 82, 83, 80]. We follow the thesis objectives through the following chapters:

**Chapter 2** Since the generic DL optimization problem is an unconstrained minimization, we provide an overview of this type of problem in this chapter. We highlight the general properties of two fundamental optimization strategies: line-search and trust-

region methods. Although the line-search method is not the primary framework utilized in this study, its review provides some ideas for a better understanding.

**Chapter 3** We formulate the DL optimization problem as a particular unconstrained minimization and do a literature review of some existing optimization strategies for solving that. This in turn requires us to have an overview of DNNs at first because the main goal of solving a DL minimization problem is finding an optimal parametric model, i.e. $h(.; w)$, which is defined by the structure of a DNN. We conclude the chapter with some general experimental configurations required for the next chapters.

**Chapter 4** In Section 4.1 of this chapter, we describe two stochastic algorithms based on two well-known limited memory Quasi-Newton Hessian approximations, i.e. L-BFGS and L-SR1, applied in a standard trust-region framework with a specific subsampling strategy. We adopt a fixed-size batching approach, where successive mini-batches overlap by half. The performance of these algorithms is compared in training different DNNs, with the goal of addressing the question of whether more efficient training can be achieved by employing a positive definite L-BFGS update or an L-SR1 one which allows for an indefinite Hessian approximation. We present and discuss the results of an extensive experimental study according to the effect of some factors on training performance. In Section 4.2, we introduce a variant of the L-BFGS trust-region method by making a simple modification to the secant condition, which improves the curvature information. The stochastic expansion of this method is used for the DL problem to assess its training performance. in training tasks. In Section 4.3, we discuss the benefits of combining directions for training by a stochastic algorithm that utilizes both a second-order direction obtained from the L-SR1 trust-region subproblem and a first-order variance-reduced direction derived from the reduced memory SAGA gradient.

**Chapter 5** We focus on the efficiency of non-monotone trust-region methods in this chapter. In Section 5.2, we describe a non-monotone second-order trust-region framework that employs a regular fixed-size mini-batching approach and incorporates the L-SR1 Hessian approximation. The curvature information for the approximation is updated through subsampled Hessian-vector product techniques. Then, in Section 5.3, we

introduce a new stochastic second-order method that is designed to incorporate adaptive mini-batch sizes. Its foundation lies in an additional sampling strategy integrated into a non-monotone trust-region framework, which aims to control the non-martingale error resulting from subsampling and thus inexact approximation. Assuming standard assumptions, we provide an almost sure convergence analysis of the proposed algorithm.

**Chapter 6** This chapter presents the concluding remarks, a summary of the main findings, and potential routes for future work.

**Appendix A** In this appendix, we provide a tutorial concerning the MATLAB *Deep Learning Toolbox* in order to give basic intuition on constructing a DNN and performing the necessary computations within a single training loop. Using *Deep Learning Custom Training Loops*, we apply `dlarray` and `dlnetwork` MATLAB objects with automatic differentiation for designing (initialized) DNNs as well as implementing prescribed training algorithms which are not available as built-in functions in MATLAB, e.g. the second-order optimization algorithms considered in this work.

**Appendix B** Second-order trust-region methods generate a sequence of iterates in a trustful region by minimizing a quadratic model of the generic objective function for finding a search direction along which the function is reduced. Solving this minimization is the heart of any method of this type. In this appendix, we provide a comprehensive description of the two solvers which are employed in this thesis to solve the trust-region subproblem using Quasi-Newton Hessian approximations, i.e. L-BFGS and L-SR1.

**Appendix C** This appendix presents several specific algorithms that are referred to in the context of the thesis.

**Appendix D** We use a special subsampling strategy for implementation in Chapter 3, in which mini-batches are half-overlapped. In this appendix, however, we explain a more general overlap subsampling and the associated computations, which may be of interest to readers.

**Appendix E** This appendix consists of additional numerical results of Chapter 3.

In this thesis, we investigate the efficiency of the algorithms proposed for training various types of DNNs to tackle specific problems, such as image recognition and regression. We compare the performance of these algorithms against well-known first- and second-order methods, including Adam [42] and STORM [9, 17]. Adaptive moment estimation (Adam) is a popular efficient first-order optimizer used in ML and DL applications. Due to the high sensitivity of Adam to the values of its hyper-parameters, such as the learning rate, it is commonly used after determining near-optimal values through time-consuming grid search strategies. This algorithm utilizes a fixed-size sampling strategy during the training phase. On the other hand, according to the statements made in [9, 17], the number of true successful iterations in a trust-region approach will be increased if the stochastic functions are sufficiently accurate. Considering this, the authors of [17] introduced a second-order trust-region method called STORM that incorporates a progressive sampling strategy in which the sample size is determined by $\min(N, \max(b_0(k+1)+b_1, \lceil \frac{1}{{\delta_k}^2} \rceil))$. Here, with some positive constants $b_0$ and $b_1$, $\delta_k$ represents the trust-region radius at iteration $k$, and $N$ denotes the number of samples.

The performance of the algorithms presented in the thesis is assessed by tracking the evolution of accuracy and loss on both training and test datasets during the training phase. Due to the use of subsampling and forming mini-batches of training data, it is important to note that the training accuracy is determined by the percentage of accurate predictions made within the current mini-batch, and the training loss is obtained by calculating the average of all single-loss functions corresponding to the samples of that mini-batch. Likewise, throughout this thesis, the measurement of testing accuracy and testing loss is defined using the entire set of test samples. These evaluation measures are employed under different experimental conditions, such as fixed budgets of epochs, time, gradient evaluations, and iterations, depending on the specific study.

We remark that the experiments of Section 4.1 and Section 4.2 were performed on an Ubuntu Linux server virtual machine with 32 CPUs and 128GB RAM, and all experiments of Section 4.3 and Chapter 5 were conducted on an Ubuntu 20.04.4 LTS (64-bit) Linux server VMware with 20GB memory using a VGPU NVIDIA A100D-20C.

# 2

# Basic Background on Unconstrained Optimization

In the following two chapters, we concentrate on the essential concepts needed in our study, i.e., numerical algorithms in DL. Since a DL problem is often an unconstrained optimization problem, we generally explain in this chapter the basic analytic features of an unconstrained optimization problem according to the excellent materials provided in [62] and [8].

## 2.1 Unconstrained Optimization Problem

For making decisions in science and in the analysis of physical systems, *optimization* is a very important tool. To use this tool, we must first identify some *objective* as a quantitative measure of the performance of the system under study, which depends on certain characteristics of the system, i.e., *variables* or *parameters*. Our goal is to find values of the variables that optimize the objective. In optimization, an important distinction is between problems that have constraints on the variables (constrained problems) and those that do not (unconstrained problems).

**Remark 2.1.** *An optimization problem arising in DL applications is mainly categorized in the second class and nonlinear programming where the objective is a nonlinear function.*

In unconstrained optimization as follow

$$\min_{w \in \mathbb{R}^n} f(w), \tag{2.1}$$

an objective smooth function $f : \mathbb{R}^n \to \mathbb{R}$ depending on real variables $w \in \mathbb{R}^n$ with no restrictions on its $n \leq 1$ values is minimized. The global minimizer of $f$ is $w^* \in \mathbb{R}^n$ such that $f(w^*) \leq f(w)$ for all $w \in \mathbb{R}^n$; i.e., a point where the function attains its least value. The global minimizer can be difficult to find because our knowledge of $f$ is usually only local, and we do not have a good picture of the overall shape of $f$. On the other hand, most of the time, a local minimizer can be found such that $f(w^*) \leq f(w)$ for all $w$ in a neighborhood if $w^*$, i.e., $\{w : \|w - w^*\| < R\}$ for a given $R > 0$.[1] However, there are more efficient and practical ways to identify local optimality through necessary and sufficient conditions as follows. Let $\nabla f : \mathbb{R}^n \to \mathbb{R}$ and $\nabla^2 f : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ be respectively gradient and Hessian of the defined $f$.

**Definition 2.1** (Stationary Point). *If $\nabla f(w^*) = 0$, then $w^*$ is called a stationary point. This point can be a maximizer, a minimizer, or an inflection.*

**Theorem 2.1** (First-Order Necessary Conditions). *Let $f$ be continuously differentiable in an open neighborhood of $w^*$. If $w^*$ is a local minimize, then $\nabla f(w^*) = 0$; i.e., any local minimizer must be a stationary point.*

**Definition 2.2** (Positive Definiteness). *A matrix $B$ is positive definite if $p^T B p > 0$ for all $p \neq 0$, and positive semidefinite if $p^T B p \geq 0$ for all $p$.*

**Theorem 2.2** (Second-Order Necessary Conditions). *Let $\nabla^2 f$ exist and be continuous in an open neighborhood of $w^*$. If $w^*$ is a local minimizer of $f$, then $\nabla f(w^*) = 0$ and $\nabla^2 f(w^*)$ is positive semidefinite.*

**Theorem 2.3** (Second-Order Sufficient Conditions). *Suppose that $\nabla^2 f$ is continuous in an open neighborhood of $w^*$, and that $\nabla f(w^*) = 0$ and $\nabla^2 f(w^*)$ is positive definite. Then $w^*$ is a strict local minimizer of $f$.*

[1] The terminology *strict minimizer* is used when $\leq$ is substituted with $<$.

**Definition 2.3** (Convex and Non-convex Functions)**.** *Let $C$ be a convex subset of $\mathbb{R}^n$, i.e., $\alpha x + (1-\alpha)y \in C$ holds for all $x$, $y \in C$ and $\alpha \in [0,1]$. A function $f : C \to \mathbb{R}$ is called convex if $f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y)$; otherwise, it is non-convex.*

**Theorem 2.4.** *When $f$ is convex, any local minimizer $w^*$ is a global of $f$. In addition, If $f$ is differentiable, then any stationary point $w^*$ is a global minimizer of $f$.*

### Convergence Issues

As it is clear convexity plays a very important role in nonlinear programming. Moreover, the first- and second-order necessary conditions can fail to guarantee local optimality of $w^*$ if $f$ is not convex; see Chapter 1 in [8]. There are a collection of algorithms for solving (2.1), which are iterative. Beginning at $w_0$, optimization algorithms generate a sequence of iterates $\{w_k\}$ that terminates when either no more progress can be made or when it seems that a solution point has been approximated with sufficient accuracy. All good algorithms should possess some properties such as *robustness*, *efficiency*, and *accuracy*. Some algorithms accumulate information gathered at $w_0, w_1, \ldots, w_{k-1}$, while others use only local information obtained at the current point $w_k$. They use this information to find a new iterate $w_{k+1}$ with a lower function value than $w_k$, i.e., $f(w_k) < f(w_{k-1})$; in this sense, they rely on an important idea, called *iterative descent*. We note that non-monotone algorithms do not insist on a decrease in $f$ at every step, but even these algorithms require $f$ to be decreased after some prescribed number of iterations ($m$), i.e., $f(w_k) < f(w_{k-m})$. In deciding how to move from one iterate $w_k$ to the next, the algorithms are distinguished. They can be categorized into two fundamental optimization strategies, i.e., *line-search* and *trust-region*. We will address these strategies in the following subsection.

Given an algorithm for solving (2.1), we ideally would like the generated sequence $\{w_k\}$ to converge to a global minimum. Unfortunately, however, this is too much to expect at least when $f$ is not convex. The most we can expect from an optimization algorithm is that it converges to a stationary point. Generally, depending on the nature of the objective function $f$, the generated sequence $\{w_k\}$ need not have a limit point; in fact, the sequence is typically unbounded if $f$ has no local minima. However, this

sequence has at least one limit point if we know the level set $\{w : f(w) \leq f(w_0)\}$ is bounded, and $f$ is enforced to be descent at each iteration, then $\{w_k\}$ must be bounded since it belongs to this level set. Although convergence to a single limit point may not be easy to guarantee, the stationarity of limit points was proven [8] for a majority of optimization algorithms to solve (2.1). From a general point of view, optimization algorithms are usually guaranteed to converge to a stationary point [62].

**Definition 2.4.** *The term "globally convergent" is used to refer to algorithms for which the following property is satisfied*

$$\lim_{k \to \infty} \|\nabla f(w_k)\| = 0. \tag{2.2}$$

*This limit is the strongest global convergence result that can be obtained; i.e., we cannot guarantee that the optimization algorithm converges to a minimizer, but only that it is attracted by stationary points. Only by making additional requirements can we strengthen this result to include convergence to a local minimum.*

## 2.2 Line-Search and Trust-Region Strategies

Most of the computational algorithms for solving (2.1) follow the line-search approach that starts by fixing a direction $p_k$ and then identifying an appropriate distance, namely the step length $\alpha_k$, or the trust-region approach that chooses a region whose maximum distance from the current point is $\delta_k$ and then seeks a direction by minimizing a given model $m_k$ whose behavior is similar to the function in that region. In this subsection, we first describe some general properties of these strategies. Then, we outline their convergence results.

Without explaining the development of each computational algorithm in detail, we aim to only describe the proposed search direction which immediately specified that algorithm which can be classified into first-order optimization methods if the gradient is applied whether into second-order optimization methods if the Hessian matrix or an approximation of that is exploited.

**General Properties of Line-Search Methods**

In the line-search strategy, a search from the current iterate $w_k$ along a chosen direction $p_k$ is done for a new iterate with a lower function value. At each iteration, a limited number of trial step lengths is generated until one is found that loosely approximates the minimum of

$$\min_{\alpha>0} f(w_k + \alpha p_k). \tag{2.3}$$

This process including selecting a direction and seeking a step length is repeated for each new iterate. Now, we concentrate on the selective search direction $p_k$.

**Definition 2.5** (Descent Direction)**.** *The search direction $p_k$ that makes an angle of strictly greater than 90 degrees with $\nabla f(w_k)$ is descent. When $p_k$ is a descent direction, that is*

$$p_k^T \nabla f(w_k) = \|p_k\| \|\nabla f(w_k)\| \cos(\theta_k) < 0.$$

*In general, any descent direction is guaranteed to produce a decrease in $f$, provided that the step length (the distance of the current point) is sufficiently small. It follows that $f(w_k + \epsilon p_k) < f(w_k)$ for all positive but sufficiently small values of $\epsilon$.*

Let $D_k \in \mathbb{R}^{n\times n}$ be a symmetric and non-singular matrix. The search direction often has the general form as $p_k = -D_k \nabla f(w_k)$. Note that there are some other directions such as ***Conjugate Gradient (CG)*** that are not typically expressed as $p_k = -D_k \nabla f(w_k)$. The following most important directions are used directly in line-search frameworks.[2]

**Steepest Descent Direction.** The simplest choice along which $f$ decreases most rapidly is $p_k = -\nabla f$. Hence, the *unit* direction $p_k$ of the most rapid decrease is

$$p_k = \frac{-\nabla f(w_k)}{\|\nabla f(w_k)\|_2}.$$

This direction does not require the calculation of the second derivative and thus is low per cost; however, it can provide a slow rate of convergence on different problems.

[2] These directions also have an analog in trust-region frameworks that we will see later.

**Newton Direction.** Perhaps the most important search direction is the Newton direction $p_k$. This direction by the following formula is reliable when the difference between the true function $f(w_k + p)$ and its quadratic model $m_k(p)$ is not too large

$$p_k = -\nabla^2 f(w_k)^{-1} \nabla f(w_k).$$

This direction provides a fast rate of local convergence. Nevertheless, its main drawback is the need for the Hessian $\nabla^2 f(w_k)$ which is often costly in its explicit computation or is not available.

**Quasi-Newton Direction.** An attractive alternative to the Newton direction is obtained by

$$p_k = -{B_k}^{-1} \nabla f(w_k),$$

where $B_k$ is an approximation of the true Hessian $\nabla^2 f(w_k)$. The new matrix $B_{k+1}$ is chosen so that it mimics the property of $\nabla^2 f(w_k)$, i.e., $\nabla^2 f(w_k)(w_{k+1} - w_k) \approx \nabla f(w_{k+1}) - \nabla f(w_k)$ by holding the secant condition $B_{k+1} s_k = y_k$ where $s_k = w_{k+1} - w_k$ and $y_k = \nabla f(w_{k+1}) - \nabla f(w_k)$. Given an initial $B_0$, additional requirements on $B_{k+1}$ such as symmetry and a restriction that the difference $B_{k+1} - B_k$ be low rank are typically imposed.

**Gauss-Newton Direction.** $F = [F_1, F_2, \ldots, F_N] \in \mathbb{R}^N$. For specially minimizing the finite sum of $N$ real-valued functions as $f(w) = \frac{1}{2} \sum_{i=1}^{N} (F_i(w))^2$, the direction Gauss-Newton using the Gauss-Newton Hessian matrix $H_G = \nabla F(w_k) \nabla F(w_k)^T$ is obtained as

$$p_p = -{H_G}^{-1} \nabla F(w_k) F(w_k),$$

The directions mentioned above are respectively special forms of a general descent direction $p_k = -D_k \nabla f(w_k)$ with $D_k = I$, $D_k = \nabla^2 f(w_k)^{-1}$, $D_k = {B_k}^{-1}$, and $D_k = {H_G}^{-1}$. However, most line-search algorithms require $p_k$ to be descent; when $D_k$ is a symmetric positive definite (SPD) matrix, $p_k$ is a descent direction in order to guarantee that $f$ can be reduced along this direction. In the case where $D_k$ is not SPD, some modification is necessary; two strategies for ensuring that the step is always of good quality were

described in [62]. An approach consists in modifying $\nabla^2 f(w_k)$ before or during solving the linear system $\nabla^2 f(w_k)p_k = \nabla f(w_k)$ in order to have an SPD coefficient by adding either a positive diagonal matrix or a full matrix to the true Hessian $\nabla^2 f(w_k)$. There are some heuristic examples of an SPD matrix $D_k$; for example, ***Diagonally Scaled Steepest Descent*** known as a diagonal approximation to the Newton direction is a descent direction in which $D_k$ is a diagonal matrix whose elements are approximations to the inverted second partial derivative of $f$. However, solving the linear system based on factorization of the coefficient matrix can be expensive in large-scale problems. It is therefore appealing to apply an iterative method from a family of methods by the general name ***inexact Newton*** methods and terminate the iterations at some approximate (inexact) solution of this system. In addition, as noted below, one can implement these methods in a **Hessian-free** manner, so that the true Hessian need not be calculated or stored explicitly at all. An inexact method can be used in both line-search or trust-region frameworks; *Newton-CG* is one of the well-known examples of this type. This method does not require explicit knowledge of the true Hessian; rather, it requires us to supply matrix-vector products of the form $\nabla^2 f(w_k)d$ for any given vector $d$. Automatic differentiation (AD) and finite differences (FD) (see Chapter 8 in [62]) can be used to calculate these Hessian–vector products when the Hessian is not available in analytic form, and thus the user cannot supply code to calculate the second derivatives, or where the Hessian requires too much storage. To briefly illustrate the AD and FD techniques given $\nabla f(w)$ at $w = w_k$, we respectively use the chain rule and the approximation as follows

$$\frac{\partial \nabla f(w)^T d}{\partial w} = \frac{\partial \nabla f(w)^T}{\partial w} d + \nabla f(w)^T \frac{\partial d}{\partial w} = \frac{\partial \nabla f(w)^T}{\partial w} d = \nabla^2 f(w)d, \tag{2.4}$$

and

$$\frac{\nabla f(w_k + hd) - \nabla f(w_k)}{h} \approx \nabla^2 f(w)d, \tag{2.5}$$

for some small differencing interval $h$. Note that the price for bypassing the computation of the Hessian in both techniques is one new gradient evaluation per iteration.

### General Properties of Trust-Region Methods

In the trust-region (TR) strategy, the information gathered about $f$ is used to construct a model function $m_k$ whose behavior near the current point $w_k$ is similar to that of $f$. We restrict the search for a minimizer of $m_k$ to some trustable regions around $w_k$ with maximum distance $\delta_k$ to ensure $m_k$ is a good approximation of $f$. At each iteration, TR finds the candidate step $p$ by approximately solving the subproblem

$$\min_{p} m_k(w_k + p), \tag{2.6}$$

where $w_k + p$ lies inside a region. If the candidate solution $w_k + p$ does not produce a sufficient decrease in $f$, we conclude that the TR is too large. Thus, $\delta_k$ is shrunk and then the minimization is resolved. Throughout the thesis, the region is a ball as $\|p_k\|_2 \leq \delta_k$, where the $\delta_k > 0$ is called the TR radius at iteration $k$. In the following, we describe two well-known models in TR. The problem (2.6) with respect to each of these models is called the ***TR subproblem***.

**Second-Order TR Model.** The model $m_k$ in (2.6) is often defined to be a quadratic function of the form

$$m_k(w_k + p) = f(w_k) + p^T \nabla f(w_k) + \frac{1}{2} p^T B_k p, \tag{2.7}$$

with the function values $f(w_k)$ and gradient values $\nabla f(w_k)$ at the point $w_k$. The matrix $B_k$ is either the exact Hessian $\nabla^2 f(w_k)$ (TR Newton method) or some approximation to it such as a Quasi-Newton approximation (TR Quasi-Newton method). If $B_k$ is positive definite, solving the quadratic minimization produces Newton direction or Quasi-Newton direction described in line-search, i.e., $p_k = -\nabla^2 f(w_k)^{-1} \nabla f(w_k)$ or $p_k = -B_k^{-1} \nabla f(w_k)$, respectively. However, unlike line-search, there is no restriction on $B_k$ to be positive definite. This is the main justification behind our decision to use TR frameworks for this thesis study.

**First-Order TR Model.** If $B_k = 0$ in (2.7), then

$$m_k(w_k + p) = f(w_k) + p^T \nabla f(w_k). \tag{2.8}$$

Using (2.8), the solution of (2.6) can be written in closed form as $p_k = -\frac{\delta_k \nabla f(w_k)}{\|\nabla f(w_k)\|_2}$. This solution can be simply seen as one step of the Steepest Descent line-search where the step length $\alpha_k$ is determined by the TR radius $\delta_k$; in fact, the TR and line-search approaches are essentially the same in this case.

### Convergence Results in Line-Search Methods

One iteration of a line-search method is given by $w_{k+1} = w_k + \alpha_k p_k$ where $\alpha_k$ is called the step length by which the method decides how far to move along the given descent direction at each iteration. The ideal choice of $\alpha_k$ would be the global minimizer of (2.3) that identifying this value is too expensive. More practical strategies perform an *inexact line-search* to identify a step length that achieves adequate reductions in $f$ at minimal cost under a certain condition. A simple reduction condition is $f(w_k + \alpha_k p_k) < f(w_k)$ which should be modified in order to provide sufficient reduction in $f$.

The popular condition for providing a sufficient reduction in $f$ is measured by

$$f(w_k + \alpha p_k) \leq f(w_k) + c_1 \alpha \nabla f(w_k)^T p_k, \tag{2.9}$$

for some $c_1 \in (0, 1)$, say $c_1 = 10^{-4}$. In other words, the reduction in $f$ should be proportional to both the step length $\alpha_k$ and the directional derivative $\nabla f(w_k)^T p_k$. The sufficient decrease condition (2.9) called the *Armijo condition* is not enough by itself to ensure that the algorithm using line-search makes reasonable progress because it is satisfied for all sufficiently small values of $\alpha$. To rule out unacceptably short steps, a second criterion requires $\alpha_k$ to satisfy

$$f(w_k + \alpha p_k)^T p_k \geq c_2 \nabla f(w_k)^T p_k, \tag{2.10}$$

for some $c_2 \in (c_1, 1)$, say $c_2 = 0.9$ when $p_k$ is chosen as a Newton or Quasi-Newton direction. The inequality (2.10) is called the *Curvature condition.* Armijo and Curvature

**Algorithm 1** Backtracking

1: **Inputs:** $\bar{\alpha} > 0$, $\tau \in (0, 1)$, $\alpha = \bar{\alpha}$
2: **while** $f(w_k + \alpha p_k) > f(w_k) + c_1 \alpha \nabla f(w_k)^T p_k$ **do**
3: $\quad \alpha = \tau \alpha$
4: **end while**
5: $\alpha_k = \alpha$

conditions with $0 < c_1 < c_2 < 1$ are known collectively as *Wolfe conditions*.

Now, let define $\phi(\alpha) = f(w_k + \alpha p_k)$. Then (2.10) restates that $\phi'(\alpha_k) \geq c_2 \phi'(0)$. If the slope $\phi'(\alpha)$ is strongly negative, we have an indication that the function $f$ can be significantly reduced by moving further along the chosen direction $p_k$. However, if the slope is slightly negative or even positive, it is a sign that we can not expect much more reduction in $f$ along $p_k$; therefore, it makes sense to terminate the line-search. Thus, a step length may satisfy the Wolfe condition without being particularly close to a minimizer of $\phi$. In order to force $\alpha_k$ to lie in at least a broad neighborhood of a local minimizer or stationary point of $\phi$, the curvature condition is modified and requires $\alpha_k$ to satisfy the following *Strong Wolfe conditions*

$$
\begin{aligned}
f(w_k + \alpha p_k) &\leq f(w_k) + c_1 \alpha \nabla f(w_k)^T p_k, \\
|f(w_k + \alpha p_k)^T p_k| &\leq c_2 |\nabla f(w_k)^T p_k|.
\end{aligned}
$$

**Remark 2.2.** *When a line-search algorithm chooses its candidate step lengths appropriately, one can dispense with the extra curvature condition and use just* (2.9) *to terminate the line-search procedure. A proper approach is so-called* ***backtracking*** *outlined in Algorithm 1.*

**Remark 2.3.** *All line-search procedures generate a sequence* $\{\alpha_i\}$ *given* $\alpha_0$*, that either terminates with a step length satisfying the conditions specified by the user (e.g. Wolfe) or determines that such a step length does not exist. Typical procedures consist of two phases: a bracketing phase which finds an interval containing desirable candidates and a selection phase which zooms into the interval to locate the final step length. The selection phase usually reduces the bracketing interval during its search and interpolates some of the function and derivative information gathered on earlier steps to guess the location of the minimize; see [62] for the details.*

The following Theorem 2.5 shows that the steepest descent method is globally convergent. It also describes how far $p_k$ in other methods can deviate from the steepest descent direction and still produce a globally convergent iteration. Various line-search termination conditions can be used in place of the Wolfe conditions to establish the result.

**Theorem 2.5.** *Consider any iteration of the form* $w_{k+1} = w_k + \alpha_k p_k$*, where* $p_k$ *is a descent direction and* $\alpha_k$ *satisfies the Wolfe conditions. Suppose that* $f$ *is bounded below in* $\mathbb{R}^n$ *and that* $f$ *is continuously differentiable in an open set* $S$ *containing the level set* $\mathcal{L} \triangleq \{w : f(w) \leq f(w_0)\}$*, where* $w_0$ *is the starting point of the iteration. Assume also that the gradient* $\nabla f$ *is Lipschitz continuous on* $S$*, that is, there exists a constant* $L > 0$ *such that* $\|\nabla f(w) - \nabla f(\tilde{w})\| \leq L\|w - \tilde{w}\|$ *for all* $w$ *and* $\tilde{w} \in S$*. Then* ***Zoutendijk condition*** *holds, i.e.,* $\sum_{k\geq 0} \cos^2(\theta_k)\|\nabla f(w_k)\|^2 < \infty$.

**Remark 2.4.** *The result of Theorem 2.5 implies that* $\cos^2(\theta_k)\|\nabla f(w_k)\|^2 \to 0$*. If chosen search direction* $p_k$ *in* $w_{k+1} = w_k + \alpha_k p_k$ *ensures that the angle* $\theta_k$ *is bounded and away from 90 degrees, i.e., there is a positive constant* $\delta$ *for all* $k$ *such that* $\cos(\theta_k) = \frac{-p_k^T \nabla f(w_k)}{\|p_k\|\|\nabla f(w_k)\|} \geq \delta > 0$*, then* $\cos^2(\theta_k)\|\nabla f(w_k)\|^2 \to 0$ *immediately ends up with* (2.2)*. In other words, the global convergence results for line-search approaches can be driven provided that the search directions are never too close to orthogonality with the gradient.*

### Convergence Results in Trust-Region (TR) Methods

One step of a TR approach is given by $w_t = w_k + p_k$ as the trial point. The effectiveness of this step critically depends on the size of the region; in fact, if the region of radius $\delta_k$ is too small, the algorithm misses an opportunity to take a substantial step that will move it much closer to the minimizer of the objective function. The new radius choice is based on the agreement between the model function $m_k$ and the objective function $f$, which is measured as $\rho_k = \frac{f(w_t) - f(w_k)}{m_k(p_k) - m_k(0)}$. If $\rho_k$ is close to zero or negative, that means that the region is too large, and the minimizer of the model may be far from the minimizer of the objective function in the region, so we may have to reduce the size of the region and try again. On the other hand, if $\rho_k$ is close to 1, it is safe to expand the region for the next iteration; otherwise, we do not alter the size of the region. In a TR approach, the step

is accepted, i.e., $w_{k+1} = w_t$ only if

$$\rho_k > \eta, \tag{2.11}$$

where $\eta$ is zero or some small positive value, which means the new objective value $f(w_k + p_k)$ is less than the current value $f(w_k)$; otherwise, the step must be rejected, i.e., $w_{k+1} = w_k$.

We need now to focus on seeking an optimal $p_k$ through the TR subproblem. From now on, let us consider the minimization (2.6) with respect to the second-order TR model (2.7) and the region which is a ball of radius $\delta_k$, i.e., $\|p_k\|_2 \leq \delta_k$. By these assumptions, the exact solutions of the TR subproblem are primarily characterized by the following theorem.

**Theorem 2.6** (Gay [28], Moré and Sorensen [58]). *Let $\delta_k$ be a given positive constant. A vector $p_k^*$ is a global solution of the second-order TR subproblem at iteration $k$ if and only if $\|p_k^*\|_2 \leq \delta_k$ and there exists a unique $\sigma_k^* \geq 0$ such that $B_k + \sigma_k^* I$ is positive semi-definite with*

$$(B_k + \sigma_k^* I)p_k^* = -\nabla f(w_k), \quad \sigma_k^*(\delta_k - \|p_k^*\|_2) = 0. \tag{2.12}$$

*Moreover, if $B_k + \sigma_k^* I$ is positive definite, then the global minimizer is unique.*

As we saw, line-search methods can be globally convergent even when the optimal step length is not used at each iteration, i.e., the step length $\alpha_k$ need only satisfy fairly loose criteria. In a TR approach, in a similar fashion, it is enough for purposes of global convergence to find an approximate solution $p_k$ that lies within the region and gives a sufficient reduction in the model. The sufficient reduction can be quantified in terms of the *Cauchy point*.

**Definition 2.6** (Cauchy Point.). *A closed-form definition of the Cauchy point can be written as $p_k^C = \tau_k p_k^s$ where $p_k^s$ is the closed-form solution of the first-order TR subproblem and*

$$\tau_k = \begin{cases} 1 & \text{if} \quad \nabla f(w_k)^T B_k \nabla f(w_k) \leq 0, \\ \min(1, \dfrac{\|\nabla f(w_k)\|^3}{\delta_k \nabla f(w_k)^T B_k \nabla f(w_k)}) & \text{otherwise;} \end{cases}$$

*thus,*

$$p_k^C = -\tau_k \frac{\delta_k \nabla f(w_k)}{\|\nabla f(w_k)\|}.$$

The point $p_k^C$ is inexpensive to calculate and crucially important in deciding if an approximate solution to the TR subproblem is acceptable. The following Lemma 2.1 shows an estimate of the decrease in $m_k$ achieved by the Cauchy point.

**Lemma 2.1.** *Let an approximate solution $p_k$ of the TR subproblem produce an estimate of the decrease in $m_k$ as follows*

$$m_k(0) - m_k(p_k) \geq c_1 \|\nabla f(w_k)\| \min\left(\delta_k, \frac{\|\nabla f(w_k)\|}{\|B_k\|}\right), \tag{2.13}$$

*where $c_1 \in (0, 1]$. The Cauchy point $p_k \triangleq p_k^C$ satisfies (2.13) with $c_1 = \frac{1}{2}$.*

However, by always taking the Cauchy point as our step, we are simply implementing the steepest descent method with a particular choice of step length, which performs poorly even if an optimal step length is used at each iteration, see Chapter 3 in [62] for this observation. Moreover, the Cauchy point does not depend very strongly on the matrix $B_k$ while rapid convergence can be expected when $B_k$ plays a role in determining the direction, and if $B_k$ contains valid curvature information about the function. Therefore, TR algorithms compute the Cauchy point and then try to improve on it. Specifically, a TR method will be globally convergent if its computed direction $p_k$ achieves a reduction in the model $m_k$ at least as much as the reduction achieved by the Cauchy point.

**Theorem 2.7.** *Let $m_k(0) - m_k(p_k) \geq c_2 \left(m_k(0) - m_k(p_k^C)\right)$ for any search direction $p_k$ such that $\|p_k\| \leq \delta_k$. Then, $p_k$ satisfies (2.13) with $c_1 = \frac{c_2}{2}$.*

Global convergence results to stationary points for TR methods come in two varieties, depending on the value of the parameter $\eta$ in (2.11). If $\eta = 0$, one can show that the sequence of gradients $\nabla f(w_k)$ has a limit point at zero while the stronger result that $\nabla f(w_k) \to 0$ is obtained if $\eta > 0$.

**Theorem 2.8.** *Let $\eta = 0$ in (2.11). Suppose that $\|B_k\| \leq \beta$ for some constant $\beta$, that $f$ is bounded below on the level set $\{w | \quad f(w) \leq f(w_0)\}$ and Lipschitz continuously differentiable in an open neighborhood, and that all approximate solutions $p_k$ of the TR*

*subproblem satisfy* (2.13). *Let* $\|p_k\| \leq \gamma\delta_k$ *for some constant* $\gamma \geq 1$ *in order to allow the results to be applied more generally provided that* $p_k$ *stays still within some fixed multiple of the bound. Then*

$$\liminf_{k\to\infty} \|\nabla f(w_k)\| = 0.$$

**Theorem 2.9.** *Let the statement of Theorem 2.8 hold but* $\eta \in (0, \frac{1}{4})$. *Then*

$$\lim_{k\to\infty} \nabla f(w_k) = 0.$$

There are two well-known approaches namely Dogleg if $B_k$ is positive definite, and the 2D subspace minimization method if $B_k$ is indefinite in order to solve the TR subproblem for $p_k$, whose convergence results can be obtained by Theorem 2.8 or Theorem 2.9. The third method is CG-Steihaug (CG-Newton in TR framework) and is most appropriate when $B_k$ is the exact Hessian or any Quasi-Newton Hessian approximation which can be large and sparse. All these three strategies produce improvements on the Cauchy point; see [62] for details. There is also a strategy due to Theorem 5.1 that finds *nearly exact* solutions $(p_k^*, \sigma_k^*)$ at the cost of eigendecomposition of the matrix $B_k$ together with an ingenious application of a 1D Newton method in order to identify the value of $\sigma_k^*$ through $\|p^*(\sigma)\| = \delta_k$ such that $B_k + \sigma I$ is positive definite and $p(\sigma) = -(B_k + \sigma I)^{-1}\nabla f(w_k)$. In [58], a safeguarded version of the root-finding Newton method is described such that its termination criteria ensure that the approximate solution $p_k$ satisfied $m_k(0) - m_k(p_k) \geq c_1 (m_k(0) - m_k(p_k^*))$ and $\|p_k^*\| \leq \gamma\delta_k$ with $c_1 \in (0, 1]$ and $\gamma > 0$. These ensure that the approximate solution achieves a significant fraction of the maximum decrease possible in the model $m_k$; see [62, 58] for more details. We consider a variation of this latter approach for solving TR Quasi-Newton methods in the next chapters of this thesis. In this variant [15], the optimal Lagrange multiplier $\sigma_k^*$ is computed by the Newton method with a judicious initial guess that does not require safeguarding. This initial guess guarantees the method to converge monotonically to $\sigma_k^*$. With $\sigma_k^*$ in hand, $p_k^*$ is computed directly by a formula; the detail is provided in Appendix B.

# 3

# An Overview on Optimization in Deep Learning

Our focus is specifically on optimization strategies tailored for training DNNs. Thus, in Section 3.1, we have an overview of DNNs; detailed materials about DNNs can be found, for example, in [32, 70]. Then, in Section 3.2 and Section 3.3, we respectively describe the DL minimization problem and conduct a literature review of existing optimization strategies for solving it. In Section 3.4, we provide experimental configurations that are utilized throughout the thesis. Let us begin this chapter by introducing two concepts that are commonly encountered in DL and ML literature. The main utility of all ML models, such as those used for image recognition, lies in their capability to generalize knowledge learned from observed training data to unseen instances. Consequently, these models are associated with two primary phases, as follows:

**Training.** During the training phase, a neural network gradually learns by adjusting its parameters, facilitating progress in performing the specific task using extracted features from a given training dataset. The training process of neural networks is closely intertwined with optimizing an objective function defined in the underlying problem. Upon completion of the training phase, an optimized model with adapted parameters is ready for the inference phase. The selection of an optimization method is an important step in designing a learning approach, which forms the focus of this thesis.

**Inference.** During the inference phase, the trained neural network is deployed on a validation or testing dataset to compute an output. This output serves as an evaluation of the training phase and indicates the extent to which the trained DNN can generalize to similar problems.

## 3.1 Deep Neural Networks

An (artificial) neural network (ANN) is a popular machine learning technique that comes from the inspiration from a biological neural network of our brain. It is often visualized as a set of interconnected artificial neurons, also called units or nodes. As a computing system, it consists of a composition of many functions together, takes some input, and returns an output. Theoretically, an ANN is capable of learning any mathematical function with sufficient training data. A Feedforward network as the simplest ANN is often arranged in (fully connected) layers each of which by a width referring to the number of neurons in that layer. The model is associated with a directed acyclic graph describing how the functions are composed together. As Figure 3.1 illustrates, given the input $x \in \mathbb{R}^2$, there are three functions $a^{[1]}$, $a^{[2]}$, and $a^{[3]}$ associated with the first layer (input layer), the second layer (hidden layer), and the third layer (output layer), respectively, to form the output $\hat{y}$. Let $W^{[1]} \in \mathbb{R}^{4\times 2}$ and $W^{[2]} \in \mathbb{R}^{4\times 1}$ be weights parameters which are columns unrolled as $w^{[1]} \in \mathbb{R}^8$ and $w^{[2]} \in \mathbb{R}^4$, respectively. Thus, the output can be defined as

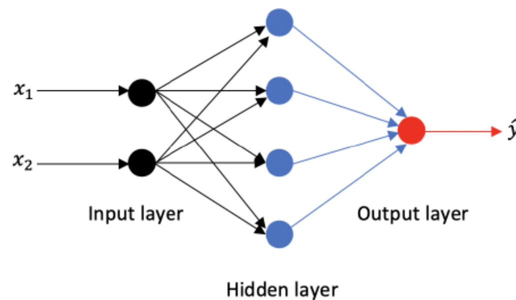
$$\hat{y} \triangleq h(x; w) = a^{[3]}(a^{[2]}(a^{[1]}(x; w^{[1]}, b^{[1]}); w^{[2]}, b^{[2]}))$$

where

$$a^{[1]}(x; w^{[1]}, b^{[1]}) \triangleq a^{[1]}(W^{[1]}x + b^{[1]}), \qquad a^{[2]}(x; w^{[2]}, b^{[2]}) \triangleq a^{[2]}(a^{[1]}(x; w^{[1]}, b^{[1]}) + b^{[2]}).$$

The single column vector of parameters $w$ includes $w^{[1]}$, $b^{[1]}$, $w^{[2]}$ and $b^{[2]}$, respectively. The chain structure of the output is the most commonly used structure in ANNs, whose overall length gives the *depth* of the model. The names Deep Learning and Deep Neural Networks (DNNs) are arisen from this terminology.

**Figure 3.1** An artificial neural network with one hidden layer.




In the chain structure described above, the weighted sum $z$ of each layer is the input to a *nonlinear* activation function, e.g. $a^{[1]}$, $a^{[2]}$, or $a^{[3]}$, which produces the value $a$ for the next layer; i.e., $a = \sigma(z)$. Activation functions work as gates that decide how the information should be passed on in a DNN. Two of the most common activation functions are the rectified linear unit (ReLU) function $\sigma(z) = \max(0, z)$, which often is the default one in modern neural networks, and the hyperbolic tangent function $\sigma(z) = \tanh(z)$, which is widely used as an alternative to the logistic sigmoid function $\sigma(z) = (1 + e^{-z})^{-1}$ whose value lies in (0, 1) but is not zero-centered. It is possible to have different activation functions in a network. For instance, corresponding $a^{[3]}$, the softmax function $\sigma(z) = e^{z_i} / \sum_{j=1}^{C} e^{z_i}$ for $i = 1, 2, \ldots, C$ can be used in the output layer in a classification problem with $C$ possible classes; the vector $\sigma(z)$ sum to 1, so the output can be interpreted as probabilities whose the highest value corresponds to what the network predicts.

**Forward and Backward Propagation.** Let us explore the process of training an ANN by considering the XOR ("exclusive or") function as an example. This will involve the concepts of forward and backward propagation. The XOR function is an operation on two binary values $x_1$ and $x_2$ as given input data. The XOR function returns 1 when exactly one of these binary values is equal to 1; otherwise, it returns 0. Let the XOR function provide the target function $y = f^*(x)$ that must be learned by an ANN. Given the input data, the (chain) model mentioned above provides the function $\hat{y} = h(x; w)$. To make the function $h$ as similar as possible to $f^*$, the model must adapt the parameters $w$; in fact, the training of an ANN is based on adapting the parameters vector $w$. The *loss* function $L(\hat{y}, y) \in \mathbb{R}$ can be seen as a distance metric that quantifies how far away the

network's prediction $\hat{y}$ is from $y$. In the *forward pass*, the execution starts by feeding the inputs through the first layer and then creating outputs for the subsequent layers. In fact, upon the computation of $\hat{y}$ and consequently $L$ with respect to the current parameters $w$, one forward pass is executed. During the process of *backward pass*, the model calculates the gradient of the loss function with respect to the current parameters. Subsequently, the parameters are updated in a direction that minimizes the loss. We notice that the derivative of the loss function, i.e. the gradient, is computed using the chain rule of calculus from the last layer to the input layer. Once the gradient is computed, it is used to perform a gradient-based optimizer for updating parameters. Note that each layer and also an optimizer may contain a set of hyper-parameters that cannot be trained and therefore have to be chosen (carefully) before the training process. The essential steps of the training process for a neural network are highlighted in Algorithm 2; the `forward`, `backward`, and `optimizer` are shorthands for calling the respective functions. The focus of this thesis is on optimization techniques for DNN training.

**Remark 3.1.** *The theory of backpropagation procedure for a feedforward network can be found in different textbooks, e.g. [32, 70]. This procedure can be generalized for other specialized kinds of neural networks by applying* ***automatic differentiation*** *[3, 62] which is a set of techniques to evaluate the partial derivative of a function specified by a computer program.*

**Remark 3.2.** *In Algorithm 2, the variable epoch counts the number of one pass through whole $N$ data examples. Moreover, a full gradient is computed with respect to all $N$ examples and then applied to a (deterministic) optimizer in line 15. However, one can apply a stochastic optimizer as* `optimizer`$(w, g)$ *in the inner loop, using a single gradient evaluated with respect to a randomly selected single example, e.g.* $(x, y)$.

### Convolutional Neural Networks (CNNs)

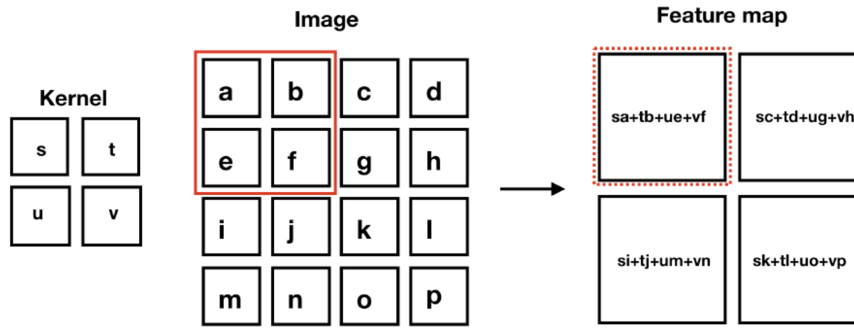
CNNs are a specialized kind of neural network for processing data that has a known grid-like topology, e.g. images. A convolutional layer is a fundamental component of a CNN that utilizes a specialized mathematical operation known as convolution. The parameters of a CNN's layer are comprised of a collection of trainable *filters* or *kernels*.

**Algorithm 2** DNNs training

```
1: Inputs: Training dataset with N samples {(x_i, y_i)}_{i=1}^N, initial parameters w, hyper-
   parameters
2: while Progress is made in terms of some measure of accuracy, do
3:     epoch = 0
4:     for i = 1, 2, ..., N do
5:         Loss = 0
6:         Gradient = 0⃗
7:         (x, y) = (x_i, y_i)
8:         ŷ ← forward(x, w)
9:         l ← loss(ŷ, y)
10:        g ← backward(l, w)
11:        Loss = Loss + l
12:        Gradient = Gradient + g
13:    end for
14:    Loss = Loss/N
15:    Gradient = Gradient/N
16:    w ← optimizer(w, Gradient)
17:    epoch = epoch + 1
18: end while
19: Result: The trained model is available for the inference phase.
```

The size of a filter refers to its spatial dimensions, specifically the width, and height. Typically, these filters are square-shaped, meaning they have equal width and height dimensions. In a given convolutional layer, all filters share the same filter size, and the number of filters determines the depth of the output volume, also known as feature maps. For example, if 10 different filters of size 2 are used for the convolution in Figure 3.2, the number of feature maps (the depth of output) is also 10. A convolution operation in a 2D grid is performed by letting a kernel slide over the input of each layer. Then, the obtained weighted sum is fed into a nonlinear function in order to produce an activation value for the next layer. Figure 3.2 illustrates the process of performing a convolution operation. The *stride* refers to the step size used to traverse the input vertically and horizontally. In the depicted scenario, the filter slides with a stride size of 2 over the image, resulting in a smaller feature map compared to the original image. To control the spatial size of feature maps, one can employ a *padding* process. Padding involves adding zeros to the outer edges of the layer's input. This approach prevents the (rapid) reduction of feature map dimensions that occurs during convolution without padding. In certain cases, it is desirable to precisely preserve the spatial size of a layer's input volume. For instance, in the context of Figure 3.2, the feature map's spatial size can match that of the original image by adding $\lceil \frac{\text{input size}}{\text{stride size}} \rceil = \lceil \frac{4}{2} \rceil$ layers of zeros to the outer edges of

**Figure 3.2** Convolution operation using a kernel of size 2 and stride size 2.




the image. This padded image can then be subjected to a $2 \times 2$ filter. *Pooling* layers are additional important layers utilized for downsampling in neural networks. In a pooling layer, each channel of an input volume is divided into non-overlapping rectangles, and specific values are outputted. The two widely used pooling functions are *max-pooling* and *average-pooling*. Max-pooling selects the maximum value within each subregion, while average-pooling computes the average value. Another commonly employed pooling operation is 2-D *global average-pooling*, which calculates the mean across the height and width dimensions of each channel in an input volume.

**Remark 3.3** (Overfitting vs. Underfitting.)**.** *Overfitting is a behavior of a learning approach, that occurs when the model is so closely fitted to the training data in the training phase but is unable to respond to new data in the inference phase. It can happen because (1) the model of a DNN is too complex, and (2) the training data size is too small for the model and/or contains large amounts of irrelevant information. Underfitting is the opposite concept of overfitting. In the early stages of training, it is common to observe underfitting behavior as the trained model is still far from the desired function. As training progresses over a longer duration, the network begins to grasp the underlying representation of the data. However, there is a risk of the network memorizing intricate patterns within the training data that may not generalize well during the inference phase. Regularization is a very important technique to prevent overfitting. When formulating the DL optimization problem in the next section, a common regularization technique relying on modifying the objective function is defined. We describe below the dropout layer that relies on modifying the network itself in order to reduce its complexity.*

**Dropout and Batch Normalization.** The dropout layer [71] and batch normalization layer [38] are additional layers that can be incorporated into the network architecture. Dropout is a regularization technique that modifies the architecture during training to prevent overfitting. The core concept is to randomly deactivate units in the neural network, where individual nodes are either dropped out of the network with a probability of $1-p$ or retained with a probability of $p$. This strategy helps improve generalization by reducing the network's reliance on specific nodes or features. During the inference phase, all units in this layer are considered, and the dropout operation is not applied.

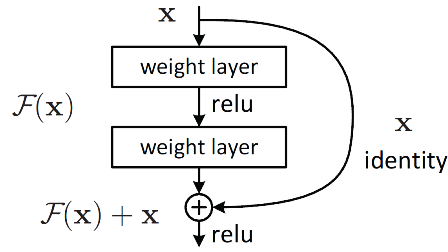
To accelerate the training process and mitigate sensitivity to network initialization, a widely adopted technique is to incorporate a batch normalization layer between a convolutional layer and an activation layer. This layer performs both standardization and normalization on the output of the preceding layer during training. By standardizing the outputs and normalizing them to have zero mean and unit variance, this layer helps stabilize the learning process and allows for more efficient training. Let $z_i^{[l]}$ be the $i$th sample of a batch of training data, e.g. $J$, in a batch normalization layer indicated as layer $l$. The values of this sample vector are standardized as $\hat{z_i}^{[l]} = \frac{z_i^{[l]}-\mu}{\sqrt{\sigma^2+\epsilon}}$, where the scalar $\epsilon$ improves numerical stability, and $\mu$ and $\sigma^2$ are respectively the mean and the variance of the batch $J$ of size $bs = |J|$

$$\mu = \frac{1}{bs}\sum_{i=1}^{bs} z_i^{[l]}, \qquad \sigma^2 = \frac{1}{bs}\sum_{i=1}^{bs}(z_i^{[l]} - \mu)^2. \tag{3.1}$$

Then, $\hat{z}_i^{[l]}$ is normalized by offset $\beta$ and scale $\gamma$ as $\tilde{z}_i^{[l]} = \gamma\hat{z}_i^{[l]} + \beta$. When network training finishes, the batch normalization layer requires a fixed mean and variance to normalize the new data for the inference phase. These quantities ($\bar{\mu}$ and $\bar{\sigma^2}$) can be tracked during the training phase. Let $X^{\{1\}}$ be the first batch of data used in one step of this phase, and let $\mu = [\mu^{\{1\}[1]}, \mu^{\{1\}[2]}]$ where $\mu^{\{1\}[1]}$ and $\mu^{\{1\}[2]}$ denote the mean of $X^{\{1\}}$ in first and second batch normalization layers, respectively. Then, set the *mean running average* as $\bar{\mu} = \mu$. Using $X^{\{2\}}$, in a similar fashion, we have $\mu = [\mu^{\{2\}[1]}, \mu^{\{2\}[2]}]$. Therefore, $\bar{\mu}$ is updated as follows

$$\bar{\mu} = \phi\mu + (1-\phi)\bar{\mu}, \qquad \bar{\sigma^2} = \phi\sigma^2 + (1-\phi)\bar{\sigma^2}, \tag{3.2}$$

**Figure 3.3** A residual building block.




where $\phi$ denotes the statistic decay value, say 0.1. This process must be continued to the end of the training phase. In a similar manner, the *variance running average* can be computed as $\bar{\sigma}^2$ in (3.2).

#### Residual Neural Networks (ResNets)

The intuition behind adding more layers is that these layers progressively learn more complex features extracted from low level to high level. But it has been found that there is a maximum threshold for depth with the traditional CNN model. In [36], a typical example shows that the percentage of the error for a 56-layer network is more than a 20-layer one in both the training and testing phases. The degradation of accuracy could be more related to the vanishing gradient problem, which has been diminished with the introduction of Residual Blocks for very DNNs which lead to ResNets [36]; see Figure 3.3. Each residual block has a *skip connection* or *shortcut*, which is a direct connection that bypasses certain layers in between (stacked layers). Without such a shortcut, the input $x$ undergoes a weighted sum and then passes through the activation function $\mathcal{F}$, resulting in the output $\mathcal{F}(x)$. Using an identity shortcut, the residual block produces the output $\mathcal{H}(x) = \mathcal{F}(x) + x$. This shortcut adds neither extra parameters nor computational complexity. A different shortcut can be also applied when the dimensions of $\mathcal{F}(x)$ differ from those of $x$. This shortcut can involve zero-padding, or it can be implemented by adding a $1 \times 1$ convolutional layer to match the dimensions. By enabling the gradient to flow through these skip connections, the issue of vanishing gradient is mitigated. Additionally, by enabling the model to learn identity functions, deeper networks exhibit performance comparable to, if not better than, shallower networks.

## 3.2 Deep Learning Optimization Problem

Deep Learning (DL) is defined as training DNNs to learn from given data in order to do a certain task. Let $\mathcal{N} = \{1, 2, \ldots, N\}$ denote the index set of the training dataset $\{(x_i, y_i)\}_{i=1}^{N}$ where $N$ represents the total number of sample pairs. Each pair consists of an input $x_i \in \mathbb{R}^d$ and a corresponding target $y_i \in \mathbb{R}^C$. Mathematically, a DL problem is often formulated as the unconstrained minimization of a finite sum function, expressed as

$$\min_{w \in \mathbb{R}^n} f(w) \triangleq \frac{1}{N} \sum_{i=1}^{N} f_i(w), \tag{3.3}$$

where $w \in \mathbb{R}^n$ is the vector of trainable parameters and $f_i(w) \triangleq L(y_i, \hat{h}(x_i; w))$ with a relevant loss function $L(\cdot)$ measuring the prediction error between the target $y_i$ and the network's output $h(x_i; w)$. In fact, a parametric function $h(x_i; \cdot) : \mathbb{R}^d \longrightarrow \mathbb{R}$ is found such that the overall prediction error is minimized. In deep learning applications, this function is then utilized to predict outputs for unseen data.

**Regularized DL Optimization Problem.** The objective function in (3.3) can be expressed with a regularization term. As we discussed earlier, overfitting occurs when a trained network performs accurately on the given data, but cannot generalize well to new data. Regularization is a broad term that describes attempts to avoid overfitting. Since large weights may lead to neurons that are sensitive to their inputs and hence less reliable when new data is presented, one can modify the objective function in order to encourage small weights by using $\mathbf{L_2}$ ***regularization*** as follow

$$\min_{w \in \mathbb{R}^n} f(w) \triangleq \frac{1}{N} (\sum_{i \in \mathcal{N}} f_i(w) + \frac{\lambda}{2} \|w\|_2^2), \tag{3.4}$$

where $\lambda > 0$ is the regularization factor. The backpropagation algorithm undergoes a negligible and affordable adjustment due to the use of this well-known regularization technique. Another technique for modifying the objective function in (3.3) is known as ***Jacobian regularization*** [37]. The primary goal of that is to penalize significant changes in the prediction $h(x_i; w) \in \mathbb{R}^C$ in response to small variations in the $i$th input $x_i \in \mathbb{R}^d$. Doing so makes the network more robust to input data polluted by noise. The

regularized DL problem using Jacobian regularization is achieved by

$$\min_{w\in\mathbb{R}^n} f(w) \triangleq \frac{1}{N}(\sum_{i\in\mathcal{N}} f_i(w) + \frac{\lambda}{2}\|J_{x_i}\|_F^2), \tag{3.5}$$

where $\|J_{x_i}\|_F$ is the Frobenius norm of the input-output Jacobian matrix $J \triangleq J_{x_i} \in \mathbb{R}^{C\times n}$ associated with the $i$th input $x_i$. Given $v_1, v_2, \ldots, v_{m_p}$ as $m_p$ random vectors in $\mathbb{R}^C$, and the network's output $\hat{y} = h(x_i; w) \in \mathbb{R}^C$, we have

$$\|J_{x_i}\|_F^2 \approx \frac{1}{m_p}\sum_{j=1}^{m_p} \frac{\partial(v_j^T \hat{y})}{\partial x}. \tag{3.6}$$

Given a random vector $v \in \mathbb{R}^C$ drawn from a standard normal distribution, the approximate value $\|J_{x_i}\|_F^2$ is obtained by the Hutchinson's trick, i.e., $\mathbb{E}_v[v^T J J^T v] = \text{trace}(JJ^T)$. According to (3.6), computing $\|J_{x_i}\|_F^2$ is equivalent at the cost of $m_p$ gradient evaluations. In practice, $m_p = 1$ in order to minimize the computational burden.

**Remark 3.4.** *Note that the biases in the parameter vector $w$ are not regularized.*

## 3.3 Deep Learning Optimization Strategies

To solve the large-scale, highly nonlinear, and often non-convex optimization problem defined in (3.3), traditional optimization algorithms are often ineffective. Consequently, significant efforts have been dedicated to the development of DL algorithms. This section aims to address some of these advancements.

#### Deterministic vs Stochastic Approaches

In many applications in statistics, machine learning, economics, and transportation (see [74] and references therein), the following stochastic optimization problem as *expected risk* arises

$$\min_{w\in\mathbb{R}^n} f(w) \triangleq \mathbb{E}(F(w, \xi)), \tag{3.7}$$

where $F : \mathbb{R}^n \times \mathbb{R}^d$ is continuously differentiable and possibly non-convex, $\mathbb{E}(.)$ is the mathematical expectation taken with respect to the random variable $\xi$, and the parameter vector $w$ is ideally chosen to minimize the expected loss. In some cases, e.g., when

the function $F(., \xi)$ is not given explicitly and/or the distribution function is unknown, the minimization of the expected risk is untenable. Thus, in practice, the objective is to find a solution to a problem that involves estimating the expected risk. There are two competing approaches, see e.g. [26], to solve (3.7): Stochastic Approximation (SA) and Sample Average Approximation (SAA). A classical SA method traced back to [65], as the prototypical stochastic optimization method is the stochastic gradient descent (SGD). This method mimics the steepest gradient descent method as follows $w_{k+1} = w_k - \alpha_k \nabla f_{i_k}(w_k)$ where $\alpha_k$ is the step length and $i_k$ refers to the $i_k$th realization $\xi_{i_k}$ which is chosen randomly for all $k \in \{0, 1, \ldots\}$, see e.g. [12]. Each iteration of this method is very cheap, involving only the computation of the stochastic gradient $\nabla f_{i_k}(w_k)$. The SAA approach, as a special case of (3.7), involves the minimization of the empirical risk given a set of realizations $\{\xi_{i_k}\}_{i=1}^{N}$ of $\xi$. This can be formulated as $\min_{w \in \mathbb{R}^n} f(w) \triangleq \frac{1}{N}\sum_{i=1}^{N} f_i(w)$, where $f_i(w) \triangleq f(w, \xi_{i_k})$ and $N$ is assumed to be extremely large. Notice that the DL problem (3.3) is exactly the empirical risk minimization described above, where $\{(x_i, y_i)\}_{i=1}^{N}$ corresponds to $\{\xi_{i_k}\}_{i=1}^{N}$. For solving this optimization problem, one can employ deterministic optimization methods in which the full gradient $\nabla f(w)$ is obtained as follows:

$$\nabla f(w) = \frac{1}{N} \sum_{i \in \mathcal{N}} \nabla f_i(w).$$

In DL applications with a very large number of training samples (large $N$), however, computing functions and gradients are expensive. Alternatively, optimization methods using *subsampling* can be applied. At each iteration of these types of methods, an approximation of the objective function is evaluated with respect to a random subset (mini-batch) of training data whose index set is $\mathcal{N}_k \subseteq \mathcal{N}$ with the sample size $N_k = |\mathcal{N}_k|$. Usually, $N_k$ is much smaller than $N$ for $k = 0, 1, \ldots$ so that algorithms operate in the stochastic approximation regime. In this regime, the subsampled function and its subsampled gradient are respectively defined as follows

$$f_{\mathcal{N}_k}(w) = \frac{1}{N_k} \sum_{i \in \mathcal{N}_k} f_i(w), \tag{3.8}$$

and

$$\nabla f_{\mathcal{N}_k}(w) = \frac{1}{N_k} \sum_{i \in \mathcal{N}_k} \nabla f_i(w). \tag{3.9}$$

**Agreement 3.1.** *In this study, any methods that employ a subsampling strategy, whether using a fixed sample size or an adaptive one, are classified in the stochastic class of DL methods.*

**Remark 3.5.** *Similar to algorithms in the deterministic regime, stochastic algorithms can also be classified in the first- or second-order class of methods. First-order methods, also known as gradient-based methods, utilize the gradient information of the objective function to guide the optimization process. They typically update the parameters based on the first derivative (gradient) of the objective function with respect to the parameters. Second-order methods, on the other hand, utilize additional information in the form of the second derivative (Hessian) of the objective function. These methods can provide more precise information about the curvature of the objective function and can potentially converge faster than first-order methods.*

We now turn our attention to the main focus of the thesis, specifically numerical methods for solving DL optimization, and begin with a literature review of several existing methods.

#### Existing Methods

In DL applications, stochastic first-order methods have been widely used due to their low per-iteration cost, optimal complexity, easy implementation, and proven efficiency in practice. The preferred method is the SGD method [65, 14], and its variance-reduced variants, e.g. SVRG [41], SAG [66], SAGA [21], SARAH [60] as well as adaptive variants, e.g. AdaGrad [25] and Adam [42]. However, due to the use of only first-order information, they come with several issues such as relatively slow convergence, high sensitivity to the choice of hyper-parameters, stagnation at high training loss [13], difficulty in escaping saddle points [84], limited benefits of parallelism due to usual implementation with small mini-batches and suffering from ill-conditioning [47]. The advantage of using second derivatives is that the loss function is expected to converge faster to a minimum due to

using curvature information. To address some of these issues, second-order approaches are available. The main second-order method incorporating the Hessian matrix is the Newton method [62], but it presents serious computational and memory usage challenges involved in the computation of the Hessian, in particular for large-scale DL problems with many parameters (large $n$); see [13] for details. Alternatively, Hessian-Free (HF) [53] and Quasi-Newton (QN) [62] methods are two techniques aimed at incorporating second-order information without computing and storing the true Hessian matrix.

An HF optimization method, also known as truncated Newton or inexact Newton method (see Section 2.2) attempts to efficiently estimate Hessian-vector products by a technique known as the *Pearlmutter trick*. This method computes an approximate Newton direction using e.g. the conjugate gradient method which can calculate the Hessian-vector product without explicitly calculating the Hessian matrix [51, 10, 78]; see (2.4) and (2.5) for the ways of producing the product. However, HF methods have shortcomings when applied to large-scale DNNs. This major challenge is addressed in [53] where an efficient HF method using only a small sample set (a mini-batch) to calculate the Hessian-vector product could reduce the cost. According to the comparison of complexity can be found in the table provided in [78], the number of iterations required for the (modified) CG method, whether utilizing the true or subsampled Hessian matrix-vector products, is higher compared to that of a limited memory QN method. Note that HF methods are not limited to inexact Newton methods; many algorithms employ approximations of the Hessian that maintain positive definiteness, such as those proposed in [67] and [79], where the Gauss-Newton Hessian matrix $H_G$ and diagonal Hessian approximation (see Section 2.2) are used, respectively. It is recognized that the curvature matrix (Hessian) related to objective functions in neural networks is predominantly non-diagonal. Therefore, there is a need for an efficient and direct approach to compute the inverse of a non-diagonal approximation to the curvature matrix (without depending on methods such as CG). This could potentially lead to an optimization method whose updates are as potent as HF methods while being (almost) computationally inexpensive. Kronecker-factored Approximate Curvature (K-FAC) [54] is such a method that can be much faster in practice than even highly tuned implementations of SGD with momentum

on certain standard DL optimization benchmarks. It is obtained by approximating several large blocks of the Fisher information matrix as the Kronecker product of two significantly smaller matrices. Precisely, this matrix is approximated by a block diagonal matrix, where the blocks are approximated with information from each layer in the network. Note that the Fisher information matrix is the expected value of $H_G$.

QN methods aim to merge the efficiency of the Newton method with the scalability of first-order methods. They build approximations of the Hessian matrix solely based on gradient information and demonstrate superlinear convergence. The primary focus lies on two widely recognized QN methods: Broyden-Fletcher-Goldfarb-Shanno (BFGS) and Symmetric Rank One (SR1), along with their limited memory variants, abbreviated as L-BFGS and L-SR1, respectively. These methods can leverage parallelization and exploit the finite-sum structure of the objective function in large-scale DL problems; see e.g. [13, 40, 5]. In stochastic settings, these methods, utilizing a subsampled gradient and/or subsampled Hessian approximation, have been investigated in the context of convex and non-convex optimization in ML and DL.

There are some algorithms for online convex optimization and for strongly convex problems, see e.g. [16, 68]. For strongly convex problems, a method was proved in [59] to be linearly convergent by incorporating a variance reduction technique to soothe the effect of noisy gradients; see also [33]. There is also a regularized method in [56] as well as an online method for strongly convex problems in [57] extended in [50] to incorporate a variance reduction technique. For non-convex optimization in DL, one can refer to e.g. [74] in which a damped method incorporating the SVRG approach was developed, [7] in which an algorithm using overlap batching scheme was proposed for stability and reducing the computational cost, or [11] where a progressive batching algorithm including the overlapping scheme was suggested. A K-FAC block diagonal QN method was also proposed, which takes advantage of network structures for required computations, see e.g [30]. Almost all previously cited articles are considered with whether a BFGS or L-BFGS update which is a symmetric positive definite Hessian approximation. A disadvantage of using a BFGS update with such a property may occur when it tries to approximate an indefinite (true Hessian) matrix in a non-convex setting while SR1 or L-SR1 updates can

allow for indefinite Hessian approximations. Moreover, almost all articles using BFGS are considered in line-search frameworks except e.g. [63] which adopts a trust-region (TR) approach. Obviously, TR approaches [18] present an opportunity to incorporate both L-BFGS and L-SR1 QN updates. As an early example, [27] can be referenced, where L-SR1 updates are utilized within a TR framework. To the best of our knowledge, no comparative study has yet explored the utilization of Quasi-Newton TR methods with L-SR1 and L-BFGS.

The potential benefits of non-monotonicity can be traced back to [35], where a non-monotone line-search technique was proposed for the Newton method. This technique aimed to relax certain standard line-search conditions in order to avoid slow convergence. Consequently, the resulting method allows for an increase in function values without affecting convergence properties. The application of non-monotonicity in TR methods can be found in [22], and later in various studies such as [2, 19]. Recently, while this thesis was being written, a noise-tolerant TR algorithm was introduced in [72]. This algorithm also incorporates relaxation for both the numerator and the denominator of the TR reduction ratio. Non-monotonicity was also utilized in stochastic regimes. To address problems with a finite-sum objective function, such as (3.3), a class of algorithms was introduced in [43]; these algorithms employ non-monotone line-search rules that adaptively adjust the sampling scheme at each iteration. None of the previously mentioned non-monotone methods have been specifically developed for training DNNs.

TR methods, whether employed in a standard or non-monotone regime, can take advantage of adaptive subsampling techniques. Various literature discusses adaptive sample size strategies. One particular type was implemented for a second-order method within a standard TR framework, known as the STORM algorithm [9, 17]. In the approach used in [27], a periodical progressive subsampling strategy is employed. Another form of adaptive subsampling, which utilizes inexact restoration, was proposed in [4] for a first-order standard TR approach. Variable size subsampling is not limited to TR frameworks; for example, in [11], a progressive subsampling technique was explored within a line-search method.

There are a few studies that utilize additional sampling to manage the non-martingale error associated with function and gradient approximations. References such as [39, 44, 23] provide further insights into these approaches. The final aim of this thesis study is to employ an adaptive subsampling procedure with additional sampling within a $2^{nd}$-order non-monotone TR framework.

## 3.4 Experimental Setups

In this subsection, we lay the groundwork for experimental purposes that will be utilized throughout the thesis.

**Classification and Regression.** We regard image classification and regression problems as specific DL applications in the thesis. In order to solve an image classification problem for images with unknown classes/labels, we need to seek an optimal classification model by using a $C$-class training dataset $\{(x_i, y_i)\}_{i=1}^N$ with an image $x_i \in \mathbb{R}^d$ and its one-hot encoded label $y_i \in \mathbb{R}^C$. To this end, the generic problem (3.3) is minimized, where its single loss function is *softmax cross-entropy* as follows

$$f_i(w) \triangleq L(y_i, h(x_i; w)) = -\sum_{k=1}^{C} (y_i)_k \log(h(x_i; w))_k, \quad i = 1, \dots, N. \tag{3.10}$$

In (3.10), the output $h(x_i; w)$ is a prediction provided by a DNN whose last layer includes the softmax function. On the other hand, the example of regression considered in this study shows how to fit a regression model using a neural network to be able to predict the angles of rotation of handwritten digits in images. The single loss in regression problem is *half-mean-squared error* as follows

$$f_i(w) \triangleq L(y_i, h(x_i; w)) = -\frac{1}{2}(y_i - h(x_i; w))^2, \quad i = 1, \dots, N. \tag{3.11}$$

**Networks.** A large variety of CNNs architectures with different numbers of fully connected (FC) layers at the end of their structure have been proposed and tested for image classification tasks. In this study, inspired by `LeNet-5` mainly used for character recognition tasks [49], we use the `LeNet-like` network with a structure described in Table 3.1.

We also employ a modern residual network `ResNet-20` [36] exploiting special skip connections (shortcuts) to avoid possible gradient vanishing that might happen due to the deep architecture. Finally, we consider `ConvNet3FC2` with larger numbers of parameters than the two previous networks for image classification and `CNN-Rn` for image regression with structures as indicated in Table 3.1.

**Datasets.** As already mentioned, the learning approach consists of two main phases: training and inference. During the training phase, the model is fitted using the training dataset with the objective of achieving good performance on unseen data. The unseen data is created by setting aside a portion of the entire dataset, known as the test set, for evaluation purposes. For experiments, we consider several image datasets which are split into training and test sets including $N$ and $\hat{N}$ samples, respectively. We use three popular benchmarks, all in $C = 10$ categories, for image classification tasks as follows: the `MNIST` [48] and `Fashion-MNIST` [77] datasets with $70 \times 10^3$ samples of handwritten gray-scale digit images of $28 \times 28$ pixels, and the `CIFAR10` dataset [46] with $60 \times 10^3$ RGB images of $32 \times 32$ pixels. Every single image of datasets is defined as a 3-D numeric array $x_i \in \mathbb{R}^d$ where $d = 28 \times 28 \times 1$ for `MNIST` and `Fashion-MNIST`, and $d = 32 \times 32 \times 3$ for `CIFAR10`. Every single label of these images, in order to be applicable in (3.10), must be converted into a one-hot encoded label as $y_i \in \mathbb{R}^C$. $\hat{N} = 10 \times 10^3$ images of each dataset are set aside as test images, and the remaining $N$ images are set as training images. For the image regression task, we use the `DIGITS` dataset[1] containing $10 \times 10^3$ synthetic images with $28 \times 28$ pixels as well as their angles (in degrees) by which each image is rotated. The response $y_i$ (the rotation angle in degrees) is approximately uniformly distributed between -45 and 45. Every single image is also defined as a 3-D numeric array $x_i \in \mathbb{R}^d$ where $d = 28 \times 28 \times 1$. In the image regression problem, each training and testing dataset has the same number of images ($N = \hat{N} = 5 \times 10^3$).

**Input Normalization.** All image datasets whose pixels are in the range $[0, 255]$ are divided by 255 so that the pixel intensity range is bounded in $[0, 1]$ (zero-one rescaling). Besides this, we may apply z-score normalization, i.e., data standardization to have a

[1] https://www.mathworks.com/help/deeplearning/ug/data-sets-for-deep-learning.html

mean of 0 and a standard deviation of 1. If so, we explicitly state it in the text. Input normalization must be indicated through the input layer when defining the architectures of networks; see Appendix A.

**Initialization.** The parameter $w \in \mathbb{R}^n$ of a DNN includes weights and biases of convolutional layers as well as scale and offset values of batch normalization layers. The parameters in $w \in \mathbb{R}^n$ are initialized by the Glorot (Xavier) initializer [29] and zeros for respectively weights and biases of convolutional layers as well as ones and zeros respectively for scale and offset variables of batch normalization layers.

Table 3.1: The details of the Networks.

| Regression | |
|---|---|
| | $(Conv(3\times 3@8,\ 1, same)/BN/ReLu/AvgPool(2\times 2,\ 2,0))$ |
| | $(Conv(3\times 3@16,\ 1, same)/BN/ReLu/AvgPool(2\times 2,\ 2,0))$ |
| `CNN-Rn` | $(Conv(3\times 3@32,\ 1, same)/BN/ReLu$ |
| | $(Conv(3\times 3@32,\ 1, same)/BN/ReLu/DropOut(0.2)$ |
| | $FC(1)$ |
| **Classification** | |
| | $(Conv(3\times 3@16,\ 1,1)/BN/ReLu)$ |
| | $B_1$ $\begin{cases}(Conv(3\times 3@16,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@16,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_2$ $\begin{cases}(Conv(3\times 3@16,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@16,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_3$ $\begin{cases}(Conv(3\times 3@16,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@16,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_1$ $\begin{cases}(Conv(3\times 3@32,\ 2,1)/BN/ReLu)\\(Conv(3\times 3@32,\ 1,1)/BN)\\(Conv(1\times 1@32,\ 2,0)/BN)+addition(2)/Relu\end{cases}$ |
| `ResNet-20` | $B_2$ $\begin{cases}(Conv(3\times 3@32,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@32,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_3$ $\begin{cases}(Conv(3\times 3@32,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@32,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_1$ $\begin{cases}(Conv(3\times 3@64,\ 2,1)/BN/ReLu)\\(Conv(3\times 3@64,\ 1,1)/BN)\\(Conv(1\times 1@64,\ 2,0)/BN)+addition(2)/Relu\end{cases}$ |
| | $B_2$ $\begin{cases}(Conv(3\times 3@64,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@64,\ 1,1)/BN)+addition(1)/Relu\end{cases}$ |
| | $B_3$ $\begin{cases}(Conv(3\times 3@64,\ 1,1)/BN/ReLu)\\(Conv(3\times 3@64,\ 1,1)/BN)+addition(1)/g.AvgPool/ReLu)\end{cases}$ |
| | $FC(C/Softmax)$ |
| **Classification** | |
| | $(Conv(5\times 5@20,\ 1,0)/ReLu/MaxPool(2\times 2,\ 2,0))$ |
| `LeNet-like` | $(Conv(5\times 5@50,\ 1,0)/ReLu/MaxPool(2\times 2,\ 2,0))$ |
| | $FC(500/ReLu)$ |
| | $FC(C/Softmax)$ |
| **Classification** | |
| | $(Conv(5\times 5@32,\ 1,2)/BN/ReLu/MaxPool(2\times 2,\ 1,0))$ |
| | $(Conv(5\times 5@32,\ 1,2)/BN/ReLu/MaxPool(2\times 2,\ 1,0))$ |
| `ConvNet3FC2` | $(Conv(5\times 5@64,\ 1,2)/BN/ReLu/MaxPool(2\times 2,\ 1,0))$ |
| | $FC(64,/BN/ReLu)$ |
| | $FC(C/Softmax)$ |

*Table's note:* The compound $(Conv(5\times 5@32,\ 1,2)/BN/ReLu/MaxPool(2\times 2,\ 1,0)))$ indicates a simple convolutional network (convnet) including a convolutional layer ($Conv$) using 32 filters of size $5\times 5$, stride 1, padding 2, followed by a batch normalization layer ($BN$), a nonlinear activation function ($ReLu$) and, finally, a 2-D max-pooling layer with a channel of size $2\times 2$, stride 1 and padding 0. The syntax $FC(C/Softmax)$ indicates a layer of $C$ fully connected neurons followed by the $softmax$ layer. Moreover, ($AvgPool$), ($g.Avg.Pool$), and ($DropOut$) show the 2D average-pooling, global average-pooling, and drop-out layers, respectively. The syntax $addition(1)/Relu$ indicates the existence of an *identity shortcut* with functionality such that the output of a given block, say $B_1$ (or $B_2$ or $B_3$), is directly fed to the *addition* layer and then to the ReLu layer. Furthermore, $addition(2)/Relu$ in a block shows the existence of a *projection shortcut* with functionality such that the output from the two first convnets is added to the output of the third convnet, then the output is passed through the $ReLu$ layer.

# 4

# Stochastic Trust-Region Methods

In this chapter, our main focus is on stochastic Quasi-Newton (QN) trust-region (TR) approaches for training deep neural networks (DNNs) in image classification tasks. In Section 4.1, we consider limited-memory variants of two well-known QN updates, namely L-BFGS and L-SR1, for approximating the Hessian matrix in the quadratic model of the TR approach. Through an extensive experimental study, our goal is to investigate whether more efficient training can be achieved by using a positive definite L-BFGS update or an L-SR1 one which allows for an indefinite approximation and thus could be beneficial for non-convex DL problems. In other words, the aim is to assess whether indefinite Hessian approximations can be useful for non-convex DL problems. Moreover, we present in Section 4.2 a method using an L-BFGS variant obtained by a simple modification of the secant condition that provides better curvature information. Finally, in Section 4.3, we describe a stochastic hybrid method that combines an L-SR1 direction obtained by the stochastic trust-region subproblem and a direction derived by the reduced memory SAGA gradient.

## 4.1 Stochastic Quasi-Newton TR Algorithms

Let $J_k$ be a random mini-batch of a dataset whose index set is $\mathcal{N}_k \subseteq \mathcal{N}$ of the size $N_k = |\mathcal{N}_k|$, by which subsampled function and corresponding gradient can be defined as (3.8) and (3.9), respectively. In the framework of TR, the original DL problem (3.3) is

reduced into a quadratic optimization for computing a search direction $p_k$ as follows

$$p_k = \arg\min_{p \in \mathbb{R}^n} Q_k(p) \triangleq \frac{1}{2} p^T B_k p + g_k^T p \quad \text{s.t.} \quad \|p\|_2 \leq \delta_k, \quad k = 0, 1, \ldots, \tag{4.1}$$

for some $\delta_k > 0$, where $B_k$ is an approximation of the Hessian and $g_k$ is the subsampled gradient (3.9) which is evaluated at $w_k$, i.e.,

$$g_k = \nabla f_{\mathcal{N}_k}(w_k).$$

In this fashion, a sequence $\{w_k\}$ is produced whether the trail point $w_t = w_k + p_k$ with computed $p_k$ is accepted or rejected. Moreover, the positive constant $\delta_k$ in (4.1) is a radius of the region in which the similarity between the model and (subsampled) function are compared. Adjusting $\delta_k$ as well as deciding to accept $w_t$ are subject to the value of the reduction ratio as follows

$$\rho_k = \frac{f_{\mathcal{N}_k}(w_t) - f_{\mathcal{N}_k}(w_k)}{Q_k(p_k) - Q_k(0)}. \tag{4.2}$$

Precisely, it is safe to expand $\delta_k \in (\delta_0, \delta_{max})$ with $\delta_0$, $\delta_{max} > 0$ when there is a *very good* agreement between the model and (subsampled) function. However, the current $\delta_k$ is not altered if there is a *good* agreement, or it is shrunk when there is *weak* agreement. Mathematically, this adjustment is done by measuring the value of $\rho_k$ in a given interval, e.g., $[\tau_2, \tau_3] \subset (0, 1)$; see Algorithm C.1. Moreover, since the denominator in (4.2) is nonpositive, we have $w_{k+1} \triangleq w_t$ if $\rho_k$ is positive ($\rho_k > \eta$); otherwise, $w_{k+1} \triangleq w_k$. The symmetric QN matrix $B_k$ from Broyden class [62] can iteratively be constructed as a Hessian approximation such that the following *secant equation*

$$B_{k+1} s_k = y_k \tag{4.3}$$

holds with carvature pair $(s_k, y_k)$ where

$$s_k = p_k, \qquad y_k = g_t - g_k, \tag{4.4}$$

and

$$g_t = \nabla f_{\mathcal{N}_k}(w_t).$$

The computational bottleneck of most TR methods is solving (4.1) for a search direction $p_k$. In [41], an efficient algorithm named OBS was proposed for solving (4.1) where the Hessian approximation $B_k$ is a Quasi-Newton update and is not necessarily positive definite. This algorithm can be considered a generalized variant of the algorithm solving (4.1) with positive definite $B_k$; see [1, 15, 63]. In this study, we term the latter algorithm as OBB. In order to clarify the mechanism of these solvers for solving (4.1), the details of their theory and algorithms (OBS, OBB) are provided in Appendix B and Appendix C, respectively. For finding an optimal pair $(\sigma_k^*, p_k^*)$, each solver solves the stochastic variant of the optimality conditions in Theorem 5.1, i.e.,

$$(B_k + \sigma_k^* I)p_k^* = -g_k, \quad \sigma_k^*(\delta_k - \|p_k^*\|_2) = 0. \tag{4.5}$$

What follows introduces two stochastic training algorithms in a TR framework described above, where $B_k$ is limited-memory variants of two well-known QN Hessian approximations from Broyden class.

### 4.1.1 Stochastic Limited-Memory BFGS TR

BFGS is the most popular QN update which provides a Hessian approximation for which (4.3) holds. It has the following general updating form

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}, \quad k = 0, 1, \ldots. \tag{4.6}$$

The difference between the symmetric approximations $B_k$ and $B_{k+1}$ is a rank-two matrix. Moreover, the constructed QN update is a positive definite matrix, i.e., $B_{k+1} \succ 0$ since $B_0 \succ 0$ and the *curvature condition* holds, i.e., $s_k^T y_k > 0$. When this condition is not satisfied, updating $B_k$ is skipped. In this study, $B_k$ is not updated if the following rule given $\tau = 10^{-2}$ does not hold

$$s_k^T y_k > \tau \|s_k\|^2, \qquad \tau \in (0, 1). \tag{4.7}$$

For large-scale DL optimization problems, the limited-memory BFGS update (L-BFGS) is more efficient. In practice, only $l \ll n$ (usually $l < 100$) recent pairs $\{s_j, y_j\}$ are stored in the following storage matrices

$$S_k \triangleq \begin{bmatrix} s_{k-l} & s_{k-(l-1)} & \dots & s_{k-1} \end{bmatrix}, \quad Y_k \triangleq \begin{bmatrix} y_{k-l} & y_{k-(l-1)} & \dots & y_{k-1} \end{bmatrix}, \tag{4.8}$$

for $k \geq l$. Using (4.8), the L-BFGS matrix $B_k$ can be represented in the following compact form

$$B_k = B_0 + \Psi_k M_k \Psi_k^T, \quad k = 1, 2, \dots, \tag{4.9}$$

where

$$\Psi_k = \begin{bmatrix} B_0 S_k & Y_k \end{bmatrix}, \qquad M_k = \begin{bmatrix} -S_k^T B_0 S_k & -L_k \\ -L_k^T & D_k \end{bmatrix}^{-1}. \tag{4.10}$$

In (4.10), $\Psi_k$ and $M_k$ have at most $2l$ columns. Moreover, $L_k$ is the strictly lower part, $U_k$ is the strictly upper part and $D_k$ is the diagonal part of the following matrix splitting

$$S_k^T Y_k = L_k + D_k + U_k. \tag{4.11}$$

As already mentioned, the initial Hessian approximation $B_0$ must be positive definite. It is often set to some multiple of the identity matrix as $B_0 = \gamma_k I$. Thus, the selection of $\gamma_k$ is important in generating L-BFGS Hessian approximations $B_k$. A heuristic and conventional method to determine the value of the multiple is

$$\gamma_k = \frac{y_{k-1}^T y_{k-1}}{y_{k-1}^T s_{k-1}} \triangleq \gamma_k^h. \tag{4.12}$$

The quotient of (4.12) is an approximation to an eigenvalue of $\nabla^2 F(w_k)$ and appears to be the most successful choice in practice [62]. Moreover, in DL optimization (3.3) where the true Hessian might be indefinite, the positive definite L-BFGS $B_k$ has a difficult task to approximate it; therefore, the choice of $\gamma_k$ would also be crucial for this second reason. Let $H \in \mathbb{R}^{n \times n}$ and $g \in \mathbb{R}^n$ are constant, and, for simplicity, the objective function of (3.3) be as follows

$$f(w) = \frac{1}{2} w^T H w + g^T w. \tag{4.13}$$

Since $\nabla f(w_{k+1}) - \nabla f(w_k) = H(w_{k+1} - w_k)$, we have $y_k = Hs_k$, and thus $S_k^T Y_k = S_k^T H S_k$ for all $k$. Moreover, using (4.9) for the quadratic function (4.13) with $H = \nabla^2 f(w)$, we obtain

$$S_k^T H S_k - \gamma_k S_k^T S_k = S_k^T \Psi_k M_k \Psi_k^T S_k. \tag{4.14}$$

According to (4.14), if $H$ is not positive definite, then its negative curvature information can be captured by $S_k^T \Psi_k M_k \Psi_k^T S_k$ as $\gamma_k > 0$. However, choosing $\gamma_k$ too big may cause false curvature information while $H$ is positive definite. To avoid this, see [27] for details and [63], $\gamma_k$ can be selected in $(0, \hat{\lambda})$ where $\hat{\lambda}$ is the smallest eigenvalue of the generalized eigenvalue problem (GEP) $S_k^T H S_k u = \lambda S_k^T S_k u$. Precisely, using (4.11), the following GEP is solved

$$(L_k + D_k + L_k^T)u = \lambda S_k^T S_k u. \tag{4.15}$$

If $\hat{\lambda} \leq 0$, then $\gamma_k = \max\{1, \gamma_k^h\}$; see Algorithm C.2 for computing $\gamma_k$ in L-BFGS update.

Given $\gamma_k$, the compact form (4.9) is applied in (4.5) where both optimality conditions together are solved for $p_k \triangleq p_k^*$ through the OBB algorithm (Algorithm C.3). This solver uses the spectral decomposition of $B_k$ as well as the Sherman-Morison-Woodbury formula for the inversion. By solving the TR subproblem for $p_k$ at each iteration, we generate a sequence $\{w_k\}$ for the DL problem (3.3). This introduces a stochastic limited-memory BFGS in a TR framework which is abbreviated as **sL-BFGS-TR (regular)**; see Algorithm C.6.

### 4.1.2 Stochastic Limited-Memory SR1 TR

Another popular QN update is the SR1 formula which generates good approximations to the true Hessian matrix, often better than the BFGS approximations [62]. The general SR1 updating formula verifying the secant condition (4.3) is given by

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}, \quad k = 0, 1, \ldots, \tag{4.16}$$

where the difference between the symmetric approximations $B_k$ and $B_{k+1}$ is a rank-one matrix. To prevent the denominator in (4.16) from vanishing, a simple safeguard that works well in practice is simply skipping the update if the denominator is small [62]; i.e.,

$B_{k+1} = B_k$. In this study, $B_k$ is updated only if the following rule given $\tau = 10^{-8}$ holds

$$|s^T(y_k - B_k s_k)| \geq \tau \|s_k\| \|y_k - B_k s_k\|, \qquad \tau \in (0, 1). \tag{4.17}$$

Unlike (4.6), if $B_k$ in (4.16) is positive definite, $B_{k+1}$ may have not the same property. In fact, the SR1 update generates a sequence of matrices that may be indefinite regardless of the sign of $y_k^T s_k$ for each $k$. We note that the value of the quadratic model in (4.1) evaluated at the descent direction $p_k$ is always smaller if this direction is also a direction of negative curvature. Therefore, the ability to generate indefinite approximations can actually be regarded as one of the chief advantages of SR1 updates in non-convex settings like DL applications. Using $S_k$ and $Y_k$ defined in (4.8), the limited-memory SR1 update (L-SR1) matrix $B_k$ can similarly be represented in a compact form as

$$B_k = B_0 + \Psi_k M_k \Psi_k^T, \quad k = 1, 2, , \ldots, \tag{4.18}$$

where

$$\Psi_k = Y_k - B_0 S_k, \qquad M_k = (D_k + L_k + L_k^T - S_k^T B_0 S_k)^{-1}. \tag{4.19}$$

In (4.19), $\Psi_k$ and $M_k$ have at most $l$ columns, and $L_k$ and $D_k$ are those defined in (4.11).

In [27], it was proven that the solution of the TR subproblem, i.e. $p_k$, becomes closely parallel to the eigenvector corresponding to the most negative eigenvalue of the L-SR1 approximation $B_k$. This shows the importance of $B_k$ to be able to capture curvature information correctly. In [27], a comprehensive discussion was provided showing how to avoid false curvature information of $B_k$ through computing $B_0$. In fact, it was highlighted how the choice of $B_0 = \gamma_k I$ for some $\gamma_k \neq 0$ affects $B_k$, and not judiciously choosing $\gamma_k$ in relation to $\hat{\lambda}$ as the smallest eigenvalue of (4.15) can have adverse effects. In particular, if $\gamma_k$ is too close to $\hat{\lambda}$ from below, then $B_k$ becomes ill-conditioned; on the other hand, if $\gamma_k$ is too close to $\hat{\lambda}$ from above, then the smallest eigenvalue of $B_k$ becomes negatively large arbitrarily. As already seen in L-BFGS, selecting $0 < \gamma_k < \hat{\lambda}$ results in $B_k \succ 0$ while $\gamma_k > \hat{\lambda}$ can result in false curvature information. According to [27], Lemma 4.1 suggests selecting $\gamma_k$ near but strictly less than $\hat{\lambda}$ to avoid asymptotically poor conditioning while improving the negative curvature properties of $B_k$.

**Lemma 4.1.** *Let the objective function of* (3.3) *be a quadratic function as* (4.13)*, and* $\hat{\lambda}$ *denote the smallest eigenvalue of* (4.15)*. Then, for all* $\gamma_k < \hat{\lambda}$*, the smallest eigenvalue of* $B_k$ *denoted by* $\lambda_{min}(B_k)$ *is bounded above by the smallest eigenvalue of* $H = \nabla^2 f(w)$ *in the span of* $S_k$*, i.e.*

$$\lambda_{min}(B_k) \leq \min_{S_k v \neq 0} \frac{v^T S_k^T H S_k v}{v^T S_k^T S_k v}.$$

In this work, we set $\gamma_k = \max\{10^{-6}, 0.5\hat{\lambda}\}$ in the case where $\hat{\lambda} > 0$; otherwise the $\gamma_k$ is set to $\gamma_k = \min\{-10^{-6}, 1.5\hat{\lambda}\}$; see Algorithm C.4 for computing $\gamma_k$. Given $\gamma_k$, the compact form (4.18) is applied in (4.5) where the optimality conditions together are solved for $p_k$ through the OBS solver [15] using the spectral decomposition of $B_k$ as well as the Sherman-Morison-Woodbury formula for the inversion; see Algorithm C.5. Thus, the sequence $\{w_k\}$ is generated for the DL problem (3.3) after solving the TR subproblem for $p_k$ at each iteration. The resulting stochastic limited-memory SR1 TR algorithm is abbreviated as **sL-SR1-TR (regular)**; see Algorithm C.6.

### Batch Formation and Computations

**Remark 4.1.** *The term* ***regular*** *refers to the regular sampling strategy applied in both algorithms, where the mini-batch* $J_k$ *with index set* $\mathcal{N}_k$ *of size* $N_k$ *is exploited through the iteration* $k$*. This was initially proposed in [68] as an alternative to the strategy in which batches changed from one iteration to the next using a BFGS-based algorithm. Thus* $y_k$ *is more stably computed as* $y_k = g_{k+1}^{J_k} - g_k^{J_k}$ *rather than* $y_k = g_{k+1}^{J_{k+1}} - g_k^{J_k}$ *where* $g_k^{J_k} \triangleq \nabla f_{\mathcal{N}_k}(w_k)$*. Therefore, the regular sampling could avoid poor curvature estimates due to stochastic gradient differences in* $y_k$ *and thus uninformative Hessian approximation* $B_{k+1}$*. However, the regular sampling requires* $g_k = \nabla f_{\mathcal{N}_k}$ *to be computed twice at* $w_k$ *and* $w_{k+1}$*. Alternatively, a cheaper* ***overlap sampling strategy*** *was proposed in [6] in which only a common part between every two consecutive mini-batches* $J_k$ *and* $J_{k+1}$ *whose index sets are respectively* $\mathcal{N}_{k+1}$ *and* $\mathcal{N}_k$ *is employed for computing* $y_k$*. Defining* $O_k = J_k \cap J_{k+1} \neq \emptyset$ *whose index set is* $\mathcal{O}_k$ *of size* $|\mathcal{O}_k|$*, we obtain* $y_k = g_{k+1}^{O_k} - g_k^{O_k}$ *where* $g_{k+1}^{O_k} \triangleq \nabla f_{\mathcal{O}_k}(w_{k+1})$*. Since* $O_k$*, and thus* $J_k$*, should be sizeable, the overlap sampling strategy is generally called* ***multibatch sampling****.*

**Remark 4.2.** *The reader may question whether it is valuable to employ the overlapping strategy in stochastic gradient differences required in the previously described algorithms, i.e.,*

$$s_k = p_k, \qquad y_k = g_t^{O_k} - g_k^{O_k}, \tag{4.20}$$

*where*

$$g_k^{O_k} \triangleq \nabla f_{\mathcal{O}_k}(w_k), \qquad g_t^{O_k} \triangleq \nabla f_{\mathcal{O}_k}(w_t). \tag{4.21}$$

*To address this question, we have performed experiments in [83] where an L-BFGS-based algorithm in a TR framework was proposed. We should note that if $y_k$ and $\rho_k$ are computed with respect to index overlapping set $\mathcal{O}_k$ and index set $\mathcal{N}_k$, see respectively (4.20) and (4.2), we can save altogether exactly $2|\mathcal{O}_k|$ single gradients at $w_k$ and $w_t$ per iteration. However, if one uses (4.4) rather than (4.20), only $|\mathcal{O}_k|$ single gradients at $w_k$ can be saved per iteration; but instead, $y_k$ can be computed with low-variance gradients because of using index set $\mathcal{N}_k$ instead of $\mathcal{O}_k$.*

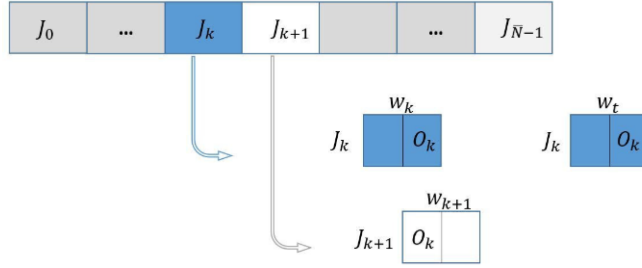
**Remark 4.3.** *Another natural question is whether it is meaningful to employ the overlapping scheme in stochastic function differences and alternatively describe (4.2) as follows*

$$\rho_k = \frac{f_{\mathcal{O}_k}(w_t) - f_{\mathcal{O}_k}(w_k)}{Q_k(p_k)}. \tag{4.22}$$

*In [83], we also addressed the second question in comparison using both measures (4.2) and (4.22). Theoretically, it was also suggested in [17] that one could compute $\rho_k$ as in (4.22). However, a stochastic algorithm in a TR framework may have a challenging task to hold a very good agreement between the quadratic model $Q_k(p_k)$ and function respectively computed with respect to $J_k$ and $O_k = J_k \cap J_{k+1}$. In some instances based on SR1 updates, we have found that (4.22) can lead to failure to converge; this observation has been also commented in [75].*

Considering Remarks 4.1–4.3, we aim to describe a specific variants of sL-BFGS-TR (regular) and sL-SR1-TR (regular). To this goal, we form mini-batches $J_k = O_{k-1} \cup O_k$, for $k = 0, 1, 2, \dots$ with two subsets $O_{k-1}$ and $O_k$ such that $|\mathcal{O}_k| = |\mathcal{O}_{k-1}| = os$, and apply **half-overlap batching strategy** as a specific variant of the overlap sampling. Let $\mathcal{N}_k$, $\mathcal{O}_k$ and $\mathcal{O}_{k-1}$ be respectively index sets of $J_k$, $O_k$ and $O_{k-1}$ such that $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$

**Figure 4.1** A schematic of the fixed-size half-overlapping scheme within one epoch.




with $|\mathcal{N}_k| = bs$. Utilizing the notation defined in (4.21), then, subsampled function and its gradient (see (3.8) and (3.9)) are computed as follows

$$f_{\mathcal{N}_k}(w_k) \triangleq f_k^{J_k} = \frac{1}{2}(f_{\mathcal{O}_{k-1}}(w_k) + f_{\mathcal{O}_k}(w_k)), \qquad g_k \triangleq g_k^{J_k} = \frac{1}{2}(g_k^{O_{k-1}} + g_k^{O_k}). \tag{4.23}$$

We note that at each iteration, the quantities $f_{\mathcal{O}_{k-1}}(w_k)$ and $g_k^{O_{k-1}}$ in (4.23) are obtained from previous iteration. Figure 4.1 schematically shows batches $J_k$ and $J_{k+1}$ overlapped in half at iterations $k$ and $k+1$, respectively. Now, we consider both sL-BFGS-TR (regular) and sL-SR1-TR (regular) with half-overlap batching, where the main quantities are obtained by (4.23), and the values of $y_k$ and $\rho_k$ are computed using (4.4) and (4.2), respectively. We outline these variants in Algorithm 3 and Algorithm 4 and name them sL-BFGS-TR and sL-SR1-TR in the respective order. We would like to determine whether more efficient training is obtained when using sL-BFGS-TR or sL-SR1-TR. This curiosity is experimentally addressed in the next section.

### 4.1.3 Numerical Comparison

In this section, we present the results of extensive experimentation of both Algorithm 3 and Algorithm 4, which are simply indicated as sL-QN-TR, for training DNNs in several image classification problems. Implementation details and programming codes of the current section are provided on the following page, see also Appendix A:

https://github.com/MATHinDL/sL_QN_TR/

**Algorithm 3** sL-BFGS-TR

1: **Inputs:** $w_0 \in \mathbb{R}^n$, $\text{epoch}_{max}$, $l$, $\gamma_0 > 0$, $S_0 = Y_0 = [\ ]$, $\delta_0 > 0$, $0 < \tau_1, \tau < 1$, bs
2: **while** epoch $<$ $\text{epoch}_{max}$ or training accuracy $< 100\%$ **do**
3:   **if** $k = 0$ **then**
4:     Take $\mathcal{O}_{-1}$ and $\mathcal{O}_0$ such that $|\mathcal{O}_{-1}| = |\mathcal{O}_0|$ and $\mathcal{N}_0 = \mathcal{O}_{-1} \cup \mathcal{O}_0$ of size $N_0 = bs$
5:     Evaluate $f_{\mathcal{N}_0}(w_0) \triangleq f_0^{J_0}$ and $g_0 \triangleq g_0^{J_0}$ by (4.23)
6:   **else**
7:     Take $\mathcal{O}_k$ such that $|\mathcal{O}_{k-1}| = |\mathcal{O}_k|$ and $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$ of size $N_k = bs$
8:     Evaluate $f_{\mathcal{N}_k}(w_k) \triangleq f_k^{J_k}$ and $g_k \triangleq g_k^{J_k}$ by (4.23)
9:     **if** $\text{mod}(k+1, \bar{N}) = 0$ **then**
10:       Shuffle the data and epoch $=$ epoch $+ 1$
11:     **end if**
12:   **end if**
13: {Compute search direction}
14:   **if** $k = 0$ **then**
15:     Obtain $p_k = -\delta_k \dfrac{g_k}{\|g_k\|}$
16:   **else**
17:     Obtain $p_k$ using Algorithm C.3
18:   **end if**
19: {Computation at $w_t$}
20:   Evaluate $f_{\mathcal{N}_k}(w_t) \triangleq f_t^{J_k}$ and $g_t \triangleq g_t^{J_k}$ by (4.23) at the trial $w_t = w_k + p_k$
21: {Compute curvature pair and $\rho_k$}
22:   Compute $(s_k, y_k)$ and $\rho_k$ by (4.4) and (4.2)
23:   **if** $\rho_k \geq \tau_1$ **then**
24:     $w_{k+1} = w_t$ {Update $w_k$}
25:   **else**
26:     $w_{k+1} = w_k$
27:   **end if**
28:   Update $\delta_k$ by Algorithm C.1 {Update $\delta_k$}
29:   **if** $s_k^T y_k > \tau \|s_k\|^2$ **then**
30:     **if** $k < l$ **then**
31:       Store $s_k$ and $y_k$ as new columns in $S_{k+1}$ and $Y_{k+1}$ {Update $B_k$}
32:     **else**
33:       Keep only $l$ recent $\{s_j, y_j\}_{j=k-l+1}^{k}$ in $S_{k+1}$ and $Y_{k+1}$
34:     **end if**
35:     Compute $\gamma_{k+1}$ for $B_0$ by Algorithm C.2 and $B_{k+1}$ by (4.9)
36:   **else**
37:     Set $B_{k+1} = B_k$
38:   **end if**
39:   $k = k + 1$
40: **end while**

*Algorithm's note:* $\delta_0 = 1,\ \gamma_0 = 1,\ \tau_1 = 10^{-4},\ \tau = 10^{-2},\ \text{epoch}_{max} = 10$.
$J_k$ represents the mini-batch of data associated with the index set $\mathcal{N}_k$.

**Algorithm 4** sL-SR1-TR

1: **Inputs:** $w_0 \in \mathbb{R}^n$, epoch$_{max}$, $l$, $\gamma_0 > 0$, $S_0 = Y_0 = [\,]$, $\delta_0 > 0$, $0 < \tau_1, \tau < 1$, bs
2: **while** epoch < epoch$_{max}$ or training accuracy < 100% **do**
3: **if** $k = 0$ **then**
4: Take $\mathcal{O}_{-1}$ and $\mathcal{O}_0$ such that $|\mathcal{O}_{-1}| = |\mathcal{O}_0|$ and $\mathcal{N}_0 = \mathcal{O}_{-1} \cup \mathcal{O}_0$ of size $N_0 = bs$
5: Evaluate $f_{\mathcal{N}_0}(w_0) \triangleq f_0^{J_0}$ and $g_0 \triangleq g_0^{J_0}$ by (4.23)
6: **else**
7: Take $\mathcal{O}_k$ such that $|\mathcal{O}_{k-1}| = |\mathcal{O}_k|$ and $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$ of size $N_k = bs$
8: Evaluate $f_{\mathcal{N}_k}(w_k) \triangleq f_k^{J_k}$ and $g_k \triangleq g_k^{J_k}$ by (4.23)
9: **if** $\mathrm{mod}(k+1, \bar{N}) = 0$ **then**
10: Shuffle the data and epoch = epoch + 1
11: **end if**
12: **end if**
13: {Compute search direction}
14: **if** $k = 0$ **then**
15: Obtain $p_k = -\delta_k \dfrac{g_k}{\|g_k\|}$
16: **else**
17: Obtain $p_k$ using Algorithm C.5
18: **end if**
19: {Computation at $w_t$}
20: Evaluate $f_{\mathcal{N}_k}(w_t) \triangleq f_t^{J_k}$ and $g_t \triangleq g_t^{J_k}$ by (4.23) at the trial $w_t = w_k + p_k$
21: {Compute curvature pair and $\rho_k$}
22: Compute $(s_k, y_k)$ and $\rho_k$ by (4.4) and (4.2)
23: **if** $\rho_k \geq \tau_1$ **then**
24: $w_{k+1} = w_t$ {Update $w_k$}
25: **else**
26: $w_{k+1} = w_k$
27: **end if**
28: Update $\delta_k$ by Algorithm C.1 {Update $\delta_k$}
29: **if** $|s^T(y_k - B_k s_k)| \geq \tau \|s_k\| \|y_k - B_k s_k\|$ **then**
30: **if** $k < l$ **then**
31: Store $s_k$ and $y_k$ as new columns in $S_{k+1}$ and $Y_{k+1}$ {Update $B_k$}
32: **else**
33: Keep only $l$ recent $\{s_j, y_j\}_{j=k-l+1}^{k}$ in $S_{k+1}$ and $Y_{k+1}$
34: **end if**
35: Compute $\gamma_{k+1}$ for $B_0$ by Algorithm C.4 and $B_{k+1}$ by (4.18)
36: **else**
37: Set $B_{k+1} = B_k$
38: **end if**
39: $k = k + 1$
40: **end while**

*Algorithm's note:* $\delta_0 = 1$, $\gamma_0 = 1$, $\tau_1 = 10^{-4}$, $\tau = 10^{-8}$, epoch$_{max} = 10$.
$J_k$ represents the mini-batch of data associated with the index set $\mathcal{N}_k$.

| | LeNet-like | ResNet-20 | ResNet-20(no BN) | ConvNet3FC2 | ConvNet3FC2(no BN) |
|---|---|---|---|---|---|
| MNIST | 431,030 | 272,970 | 271,402 | 2,638,826 | 2,638,442 |
| Fashion-MNIST | 431,030 | 272,970 | 271,402 | 2,638,826 | 2,638,442 |
| CIFAR10 | 657,080 | 273,258 | 271,690 | 3,524,778 | 3,525,162 |

Table 4.1: The total number of networks' trainable parameters ($n$).

### Configurations

We have applied three types of networks LeNet-like, ConvNet3FC2, ResNet-20 (see Table 3.1 for their architectures) as well as three benchmarks MNIST, Fashion-MNIST, and CIFAR10 for image classification problems. Table 4.1 shows the total number of trainable parameters, $n$, for each problem. To analyze the effect of batch normalization [38] on the performance of the algorithms, we have also considered variants of ResNet-20 and ConvNet3FC2 networks, named ResNet-20(no BN) and ConvNet3FC2(no BN), in which the batch normalization (BN) layers have been removed. We have also applied z-score normalization through the input layer of ConvNet3FC2 and ResNet-20 whether using batch normalization layers or not. We have used the same initial seed with the MATLAB random number generator in implementations.

### Experiments

We have compared sL-BFGS-TR and sL-SR1-TR to train the DNNs within at most 10 epochs, which was set as one of the stopping criteria. Besides, the training process was concluded upon achieving a training accuracy of 100%. Using training and testing datasets, we present the evolution of both accuracy (in percentage) and (total) loss values. During the training phase, training loss and training accuracy are obtained with respect to mini-batches while testing loss and testing accuracy are reported with respect to the testing data set; see Chapter 1 for the definition of these evaluation measurements. In this section, we report the numerical results based on training and testing accuracy and leave those based on overall loss values in Appendix E. To allow for better visualization, we have shown these measurements versus epochs using a determined frequency of display whose value is reported at the top of the figures.

What comes next is an analysis of how various elements including batch sizes, the limited memory parameter, and CPU training/running time affect the performance of the sL-QN-TR algorithms for training DNNs under the effect of batch normalization (BN) layers. We have also provided comparisons between sL-QN-TR algorithms involving particular fixed-size overlapping subsampling with a well-established second-order trust-region method involving adaptive size subsampling, i.e. STORM, and a widely used first-order algorithm involving regular fixed-size subsampling, i.e. Adam.

**Remark 4.4.** *Take into consideration that certain algorithms achieved their endpoints earlier than others in particular illustrations of the following experiments due to the influence of the considered stopping conditions.*

**Influence of the limited memory parameter.** The results reported in Figure 4.2 illustrate the effect of the limited memory parameter values $l = 5, 10$ and 20 on the accuracy achieved by the algorithms with batch sizes $bs = 500$ and 5000 within a fixed number of epochs on `CIFAR10`. As it is clearly shown in this figure, in particular for `ConvNet3FC2(no BN)`, the effect of the limited memory parameter is more pronounced when large batches are used ($bs = 5000$). For large batch sizes the larger the value of $l$ the higher the accuracy. No remarkable differences in the behavior of both algorithms with small batch size ($bs = 500$) are observed. It seems that incorporating more recently computed curvature vectors (i.e. larger $l$) does not increase the efficiency of the algorithms to train `ConvNet3FC2` while it does when BN layers are removed. Finally, we remark that we found that using larger values of $l$ (i.e. $l \geq 30$) was not helpful since it led to higher overfitting in some of our experiments.

**Influence of the batch size.** We have considered different values of the mini-batch size $bs \in \{100, 500, 1000, 5000\}$, or equivalently different values of the overlapping size $os \in \{50, 250, 500, 2500\}$, for all considered problems and DNNs. The results of these experiments for $bs = 100, 1000$, and $l = 20$ have been reported in Figures 4.3–4.14, see Appendix E for more numerical results. The general conclusion is that when training the networks for a fixed number of epochs, the achieved accuracy decreases when the batch size increases. This is due to the reduction in the number of parameter updates.

| | LeNet-like | ResNet-20 | ResNet-20(no BN) | ConvNet3FC2 | ConvNet3FC2(no BN) |
|---|---|---|---|---|---|
| MNIST | sL-SR1-TR | both | sL-SR1-TR | both | both |
| F-MNIST | sL-SR1-TR | both | sL-SR1-TR | both | sL-SR1-TR |
| CIFAR10 | sL-SR1-TR | sL-BFGS-TR | sL-SR1-TR | sL-BFGS-TR | sL-SR1-TR |

Table 4.2: Summary of the best sL-QN-TR approaches for classification problems.

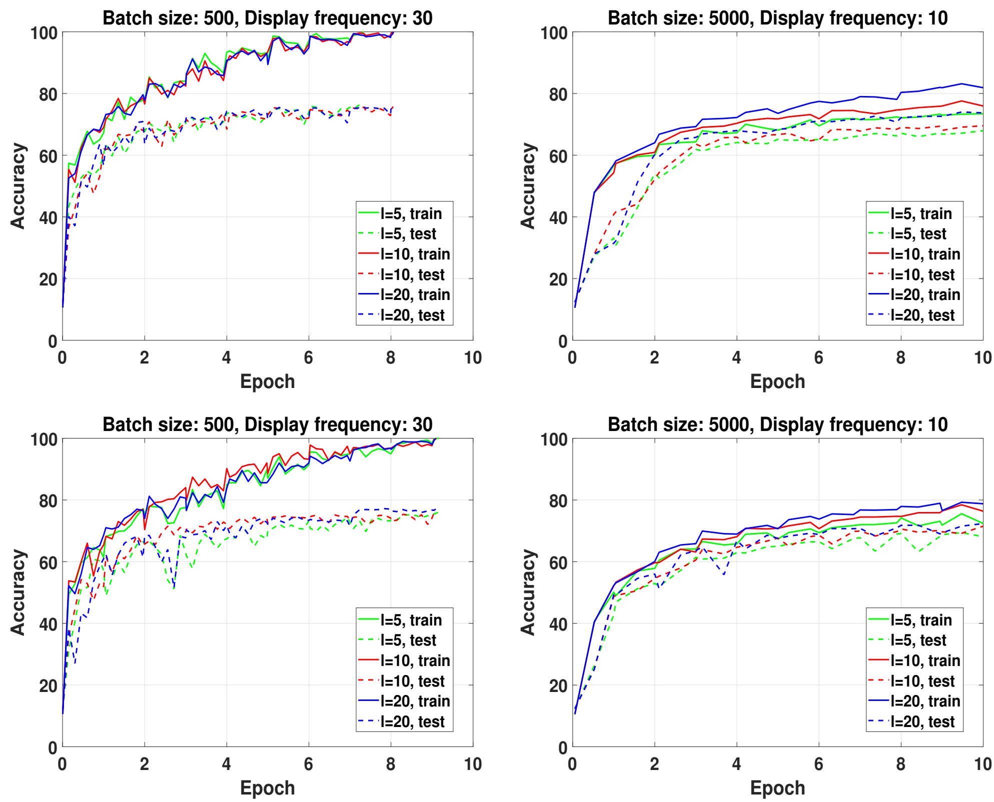
In fact, with a fixed number of epochs, the number of allowed iterations corresponding to large batch sizes is not enough to reach good accuracy. Table 4.2 summarises the relative superiority of one of the two stochastic QN algorithms over the other for all problems. sL-SR1-TR performs better than sL-BFGS-TR for training networks without BN layers while both QN updates exhibit comparable performances when used for training networks with BN layers. "Both" refers to similar behavior. More comments are given below.

***LeNet-like.*** Figures 4.3 and 4.4 show that both algorithms perform well in training `LeNet-like` within 10 epochs to classify `MNIST` and `Fashion-MNIST` datasets, respectively. Specifically, sL-SR1-TR provides better accuracy than sL-BFGS-TR.

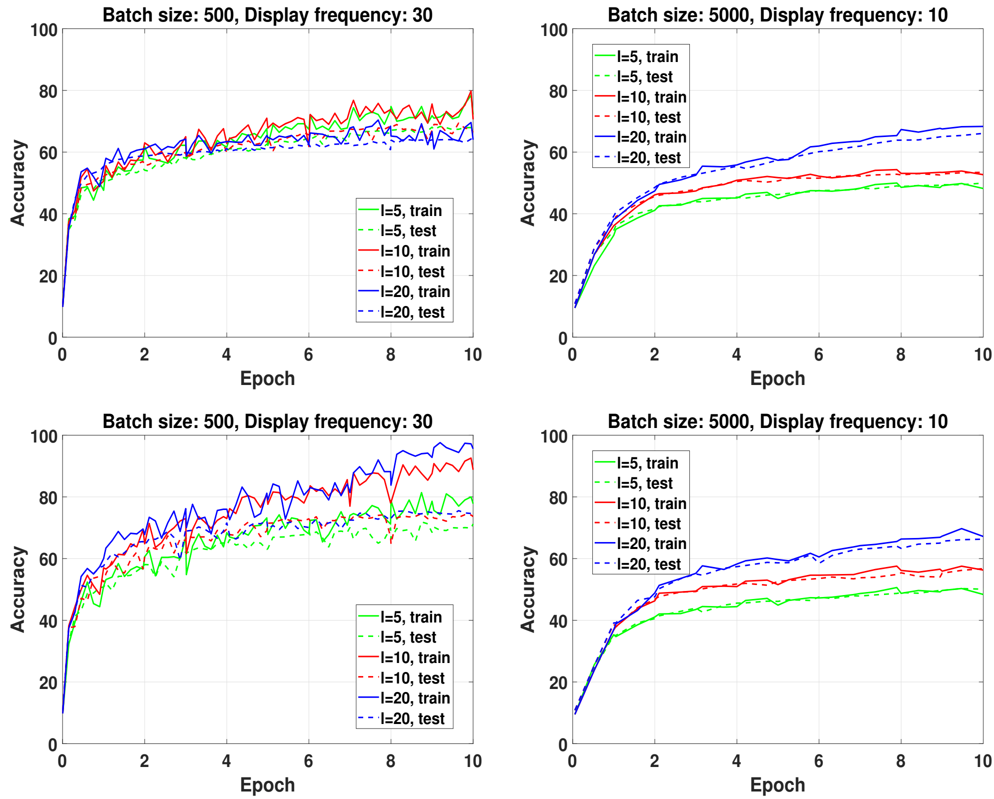
***ResNet-20.*** Figure 4.5 shows that the classification accuracy on `Fashion-MNIST` increases when using `ResNet-20` instead of `LeNet-like`, as expected. Both algorithms of interest exhibit comparable performances when BN layers are used; nevertheless, we point out the fact that sL-BFGS-TR using $bs = 100$ achieves higher accuracy than sL-SR1-TR in less time. According to Figures 4.5–4.8, the numerical results on `ResNet-20` with and without BN layers confirm that sL-SR1-TR performs better than sL-BFGS-TR when these layers are not used, but as it can be clearly seen, the elimination of BN layers causes a detriment of all methods performances.

***ConvNet3FC2.*** Figures 4.9–4.14 show that sL-BFGS-TR still produces better testing/training accuracy than sL-SR1-TR on CIFAR10 while both algorithms behave similarly on MNIST and Fashion-MNIST datasets. Besides, Figure 4.13 shows sL-BFGS-TR with $bs = 100$ within 10 epochs achieves the highest accuracy faster than sL-SR1-TR. This observation confirms once more that this method is more reliable for networks that are trained with smaller batch sizes in the presence of BN layers.

**Figure 4.2** The effect of the limited memory parameter on sL-QN-TR with `CIFAR10`.




**(a)** sL-BFGS-TR (up) and sL-SR1-TR (down) with `ConvNet3FC2`




**(b)** sL-BFGS-TR (up) and sL-SR1-TR (down) with `ConvNet3FC2(no BN)`

Figure 4.3 The accuracy of sL-QN-TR on `MNIST` with `LeNet-like`.

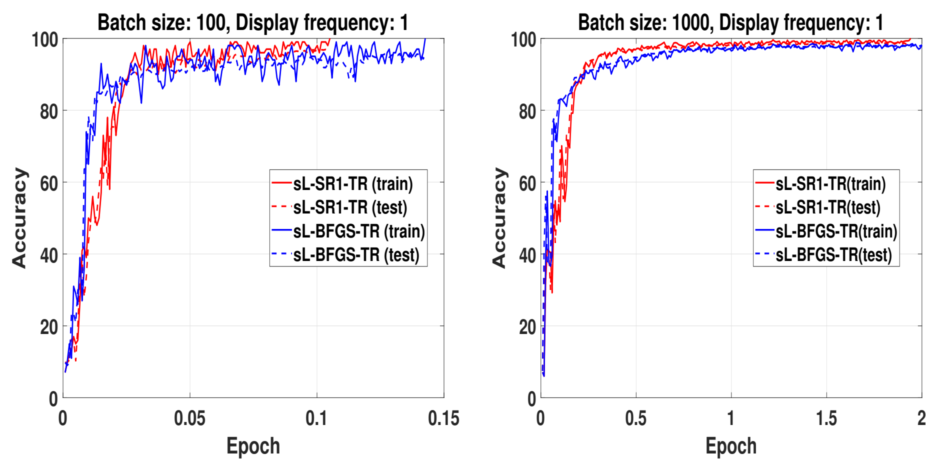



Figure 4.4 The accuracy of sL-QN-TR on `Fashion-MNIST` with `LeNet-like`.

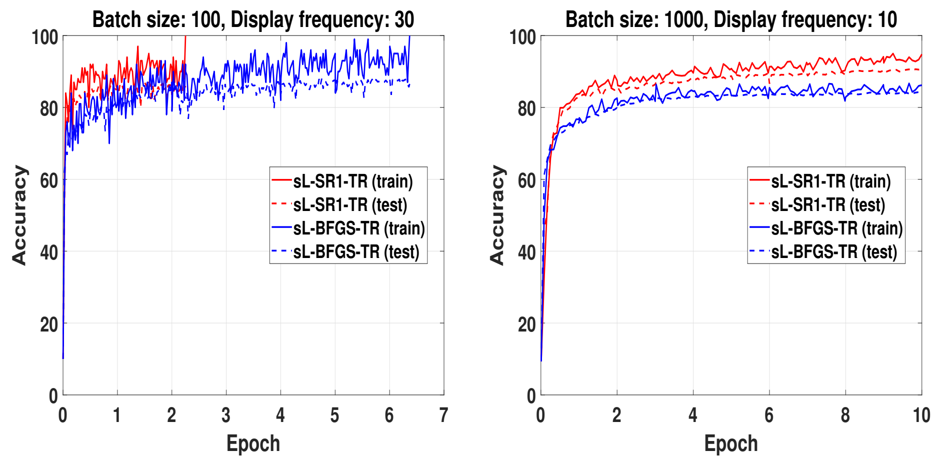



Figure 4.5 The accuracy of sL-QN-TR on `Fashion-MNIST` with `ResNet-20`.

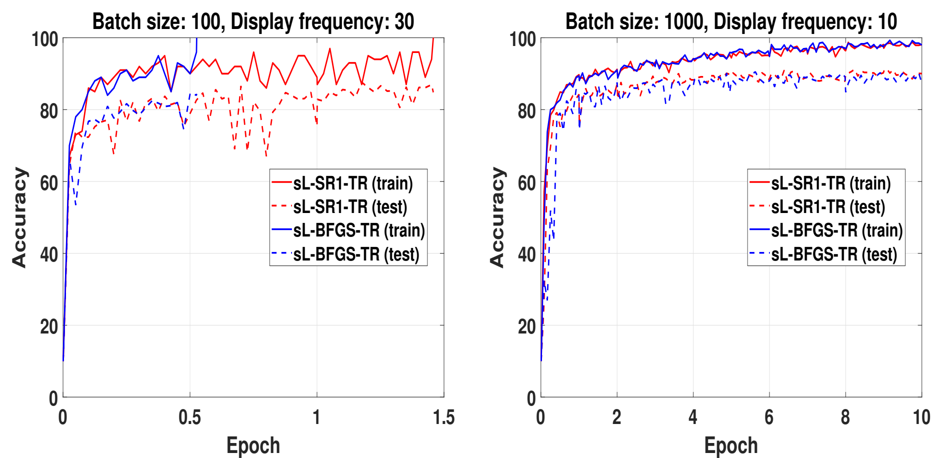


**Figure 4.6** The accuracy of sL-QN-TR on `Fashion-MNIST` with `ResNet-20(no BN)`.

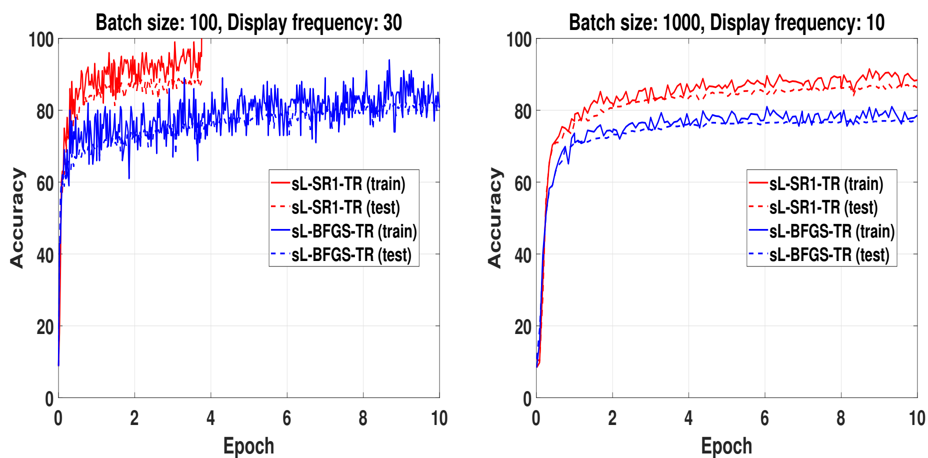



**Figure 4.7** The accuracy of sL-QN-TR on `CIFAR10` with `ResNet-20`.

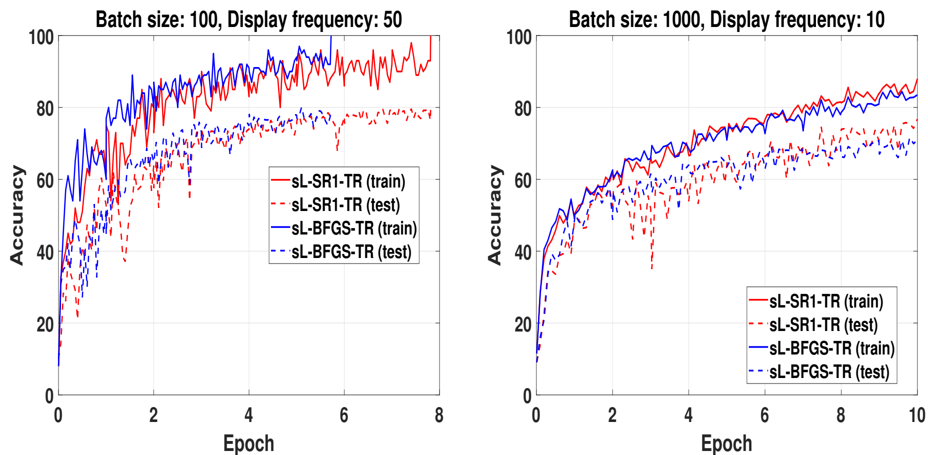



**Figure 4.8** The accuracy of sL-QN-TR on `CIFAR10` with `ResNet-20(no BN)`.

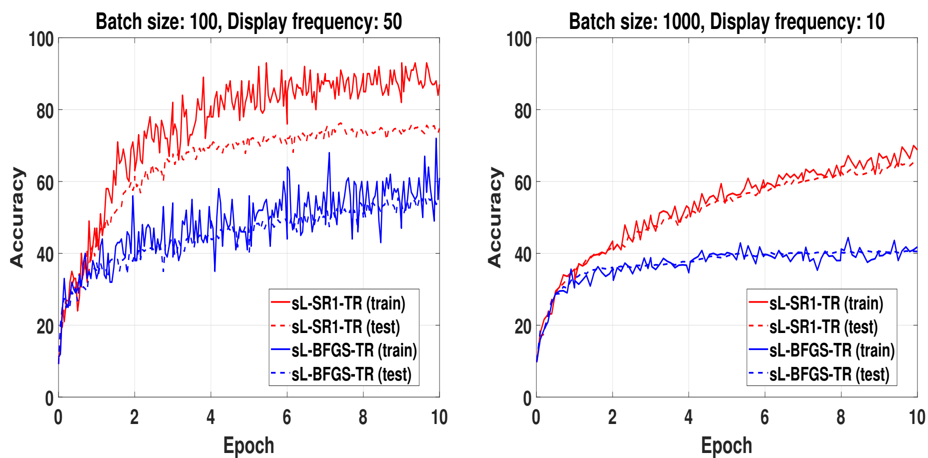


**Figure 4.9** The accuracy of sL-QN-TR on `MNIST` with `ConvNet3FC2`.

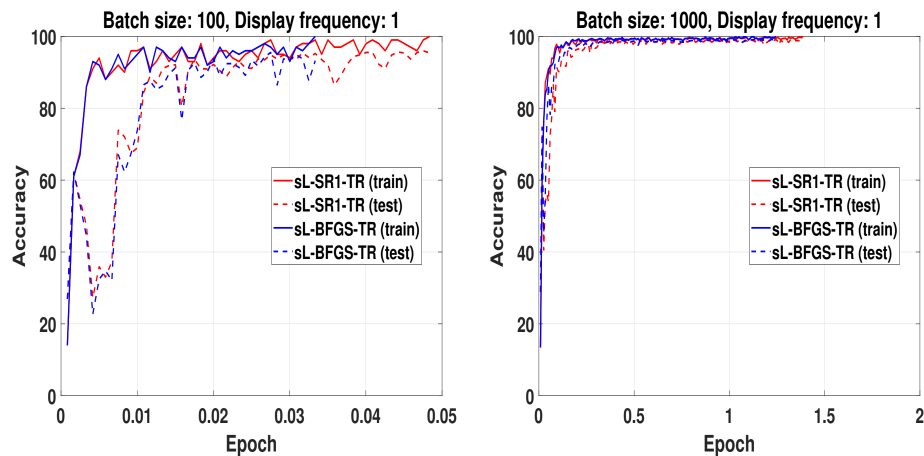



**Figure 4.10** The accuracy of sL-QN-TR on `MNIST` with `ConvNet3FC2(no BN)`.

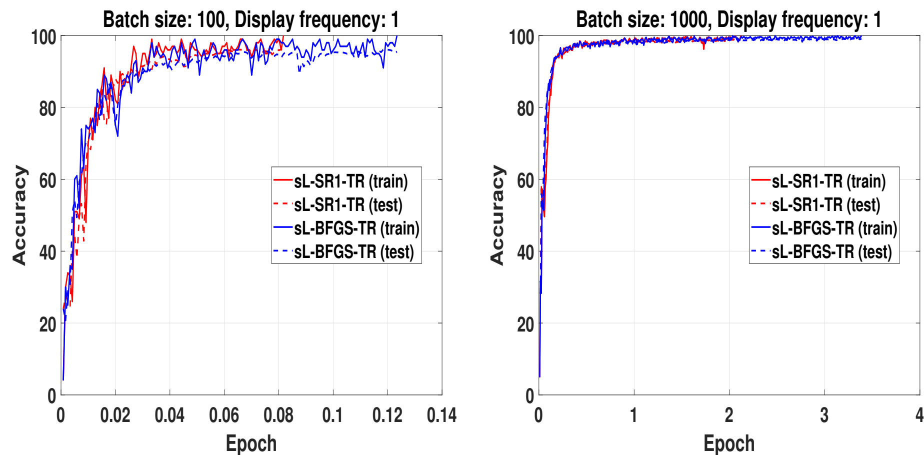



**Figure 4.11** The accuracy of sL-QN-TR on `Fashion-MNIST` with `ConvNet3FC2`.

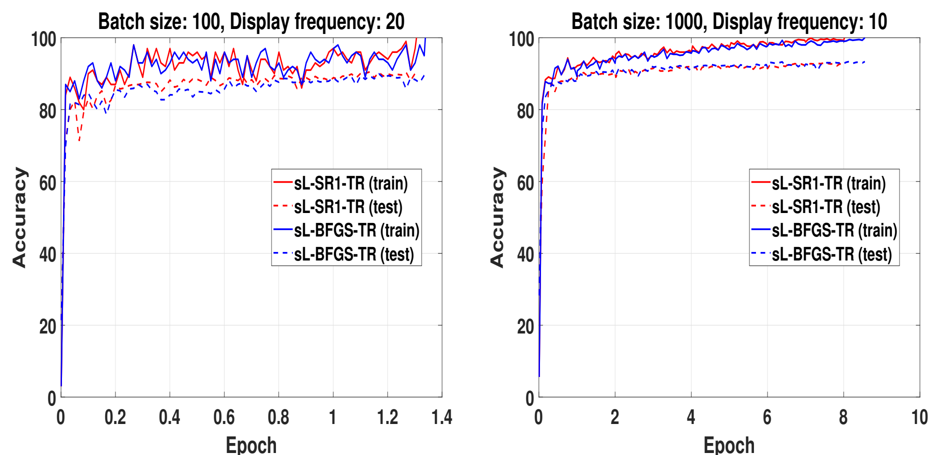


**Figure 4.12** The accuracy of sL-QN-TR on `Fashion-MNIST` with `ConvNet3FC2(no BN)`.

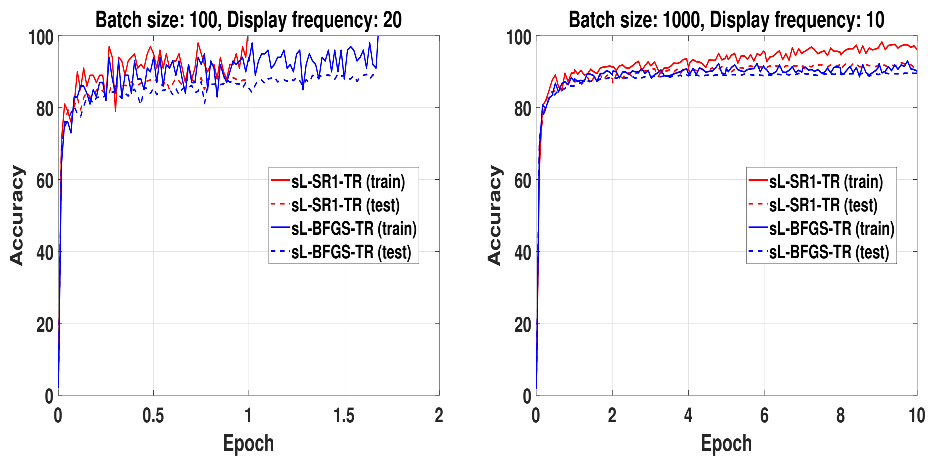



**Figure 4.13** The accuracy of sL-QN-TR on `CIFAR10` with `ConvNet3FC2`.

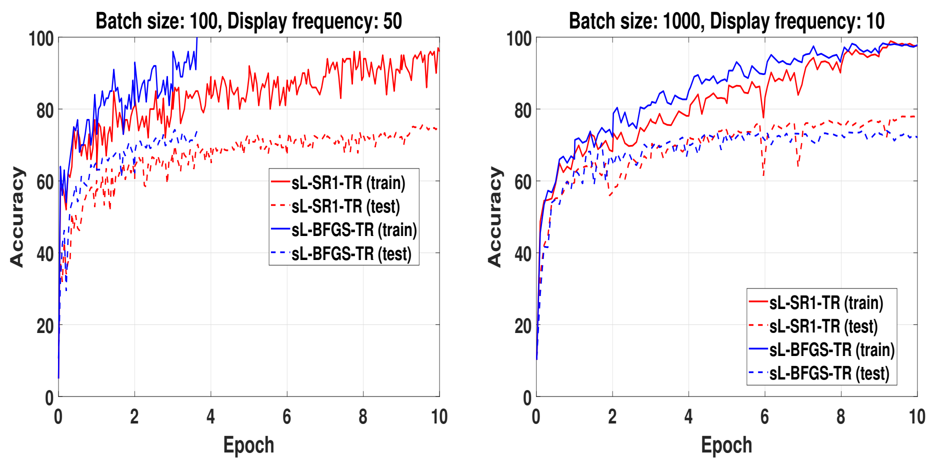
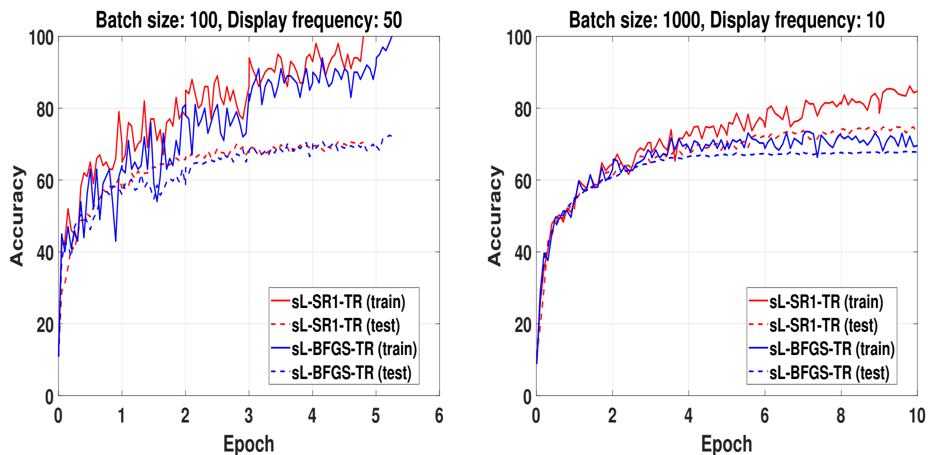



**Figure 4.14** The accuracy of sL-QN-TR on `CIFAR10` with `ConvNet3FC2(no BN)`.



**CPU training time.** We have run the sL-QN-TR algorithms ($l = 20$) for a maximum budget of CPU time indicated on the $x$-axes of Figures 4.15–4.18 in order to see which algorithm achieves the highest training accuracy faster. Figures show that sL-SR1-TR trains faster with better accuracy than sL-BFGS-TR for training `LeNet-like`. For training `ConvNet3FC2` with and without BN layers, both algorithms behave comparably within the selected interval of time when BN layers are used. Nevertheless, sL-SR1-TR is faster to pass 10 epochs even if it does not achieve higher training accuracy. This experiment illustrates that both algorithms can yield very similar training accuracy regardless of the batch size. Despite the small influence of the batch size on the final reached accuracy, it can be observed a slight increase in the accuracy when larger batch sizes are used.

**Comparison with STORM.** As mentioned in Chapter 1, STORM involves an adaptive sample size rule as $b_k = \min(N, \max(b_0(k+1) + b_1, \lceil \frac{1}{{\delta_k}^2} \rceil))$. We have set, as in [17], $b_0 = 100$, and $b_1$ is $32 \times 32 \times 3$ for `CIFAR10` and $28 \times 28 \times 1$ for `Fashion-MNIST`. We have implemented STORM as Algorithm 5 in [17] with L-SR1 and L-BFGS updates, and thus OBS and OBB solvers for solving its TR subproblem, respectively. We have considered half-overlapped batches of fixed-size $bs \in \{100, 500, 1000, 5000\}$, or equivalently $os \in \{50, 250, 500, 2500\}$, for the sL-QN-TR algorithms ($l = 20$) within at most 10 epochs. We have observed that STORM passed 10 epochs rapidly due to its progressive sampling behavior. Thus, we have allowed it to execute for more epochs, i.e., 50 epochs, as shown in Figures 4.19 and 4.21.

In both `Fashion-MNIST` and `CIFAR10` problems, the algorithms with $bs = 500$ and 1000 produce comparable or higher accuracy than STORM at the end of their own training phase. Even if we set a fixed budget of time corresponding to one needed for passing 50 epochs by STORM, sL-QN-TR algorithms with $bs = 500$ and 1000 provide comparable or higher accuracy. We need more consideration on the smallest and largest batch sizes. When $bs = 100$, the algorithms can not be better than STORM with any fixed budgets of time; however, they provide higher training accuracy and testing accuracy, except for `Fashion-MNIST` problem on `ResNet-20` trained by sL-BFGS-TR, at the end of their training phase. This makes sense due to training with batches of small size. In contrast,

when $bs = 5000$, sL-BFGS-TR algorithms only can produce higher or comparable training accuracy without any comparable testing accuracy. This is normal behavior as they could update only a few parameters within 10 epochs when $bs = 5000$; allowing longer training time or more epochs can compensate for this lower accuracy. This experiment also shows another finding that sL-BFGS-TR algorithms with $bs = 5000$ can be preferred to $bs = 100$ because they could yield higher accuracy within less time.

**Comparison with Adam.** In order to determine the optimal value of hyper-parameters allowing Adam[1] to achieve the highest testing accuracy, we have performed a grid search of learning rates $lr \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$ and different values of batch sizes $bs \in \{100, 500, 1000, 5000\}$. The gradient and squared gradient decay factors are set as $\beta_1 = 0.9$ and $\beta_2 = 0.999$, respectively, and the small constant to prevent divide-by-zero errors is set to $10^{-8}$ in Adam. It is worth noting that sL-QN-TR approaches do not require step-length tuning, and this particular experiment offers a comparison with the optimized Adam optimizer. In Figures 4.20 and 4.22–4.25 where $l = 20$, we have analyzed which algorithm achieves the highest training accuracy within at most 10 epochs for different batch sizes. In networks using BN layers, all methods achieve comparable training and testing accuracy within 10 epochs with $bs = 1000$. However, this cannot be generally observed when $bs = 100$; see Figure 4.22. The figure shows *tunned* Adam has higher testing accuracy than sL-SR1-TR. Nevertheless, sL-BFGS-TR is still faster to achieve the highest training accuracy, as we also previously observed, with comparable testing accuracy with *tunned* Adam. On the other hand, for networks without BN layers, sL-SR1-TR is the clear winner against both other algorithms. Another important observation is that Adam is more affected by batch sizes than the sL-QN-TR algorithms, thus the advantage of them over Adam can increase to enhance the parallel efficiency when using large batch sizes.

**Extended numerical results.** More numerical results are provided in Appendix E.

[1]Throughout the thesis, the Adam algorithm has been implemented using the MATLAB built-in function `adamupdate`.

**Figure 4.15** The accuracy of sL-QN-TR vs time on `MNIST` with `LeNet-like`.

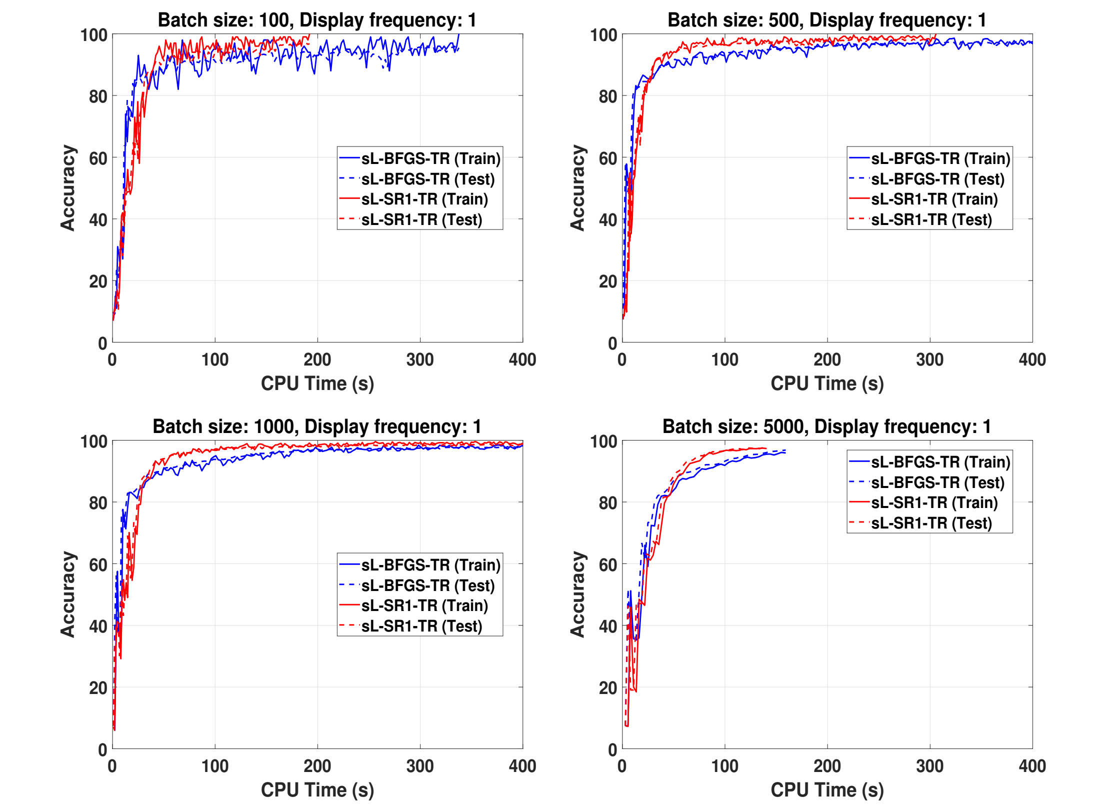



**Figure 4.16** The accuracy of sL-QN-TR vs time on `Fashion-MNIST` with `LeNet-like`.

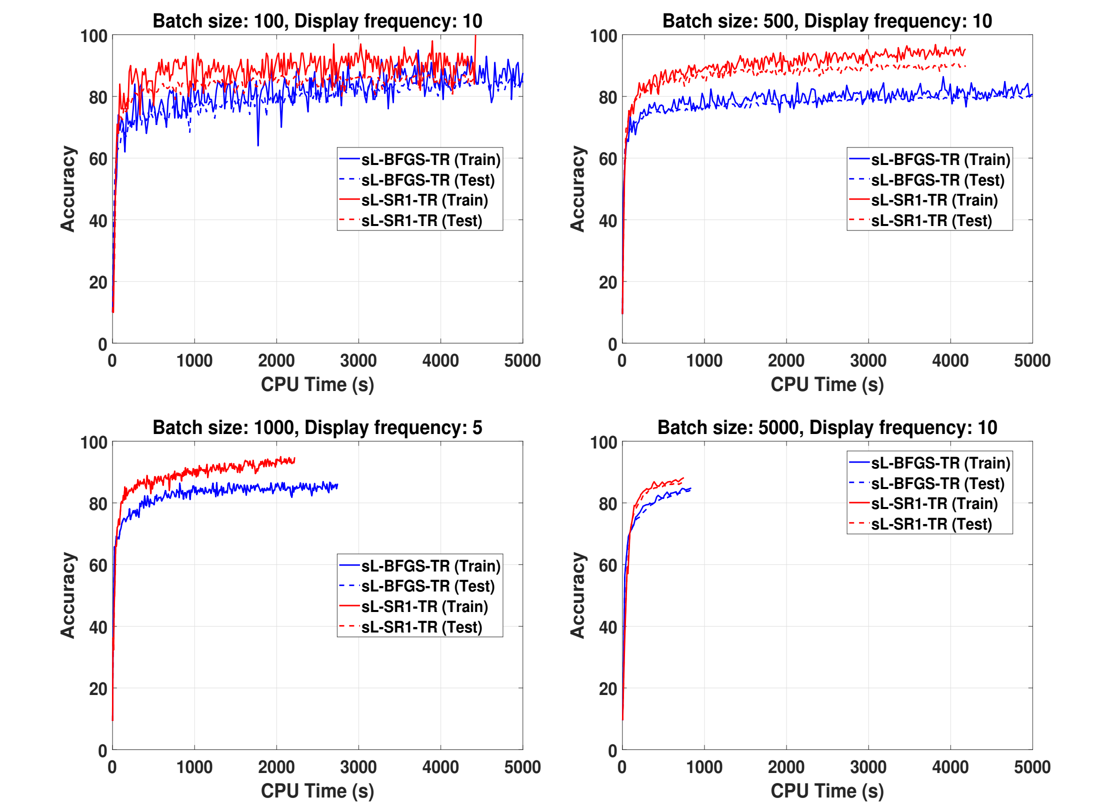


**Figure 4.17** The accuracy of sL-QN-TR vs time on `CIFAR10` with `ConvNet3FC2`.

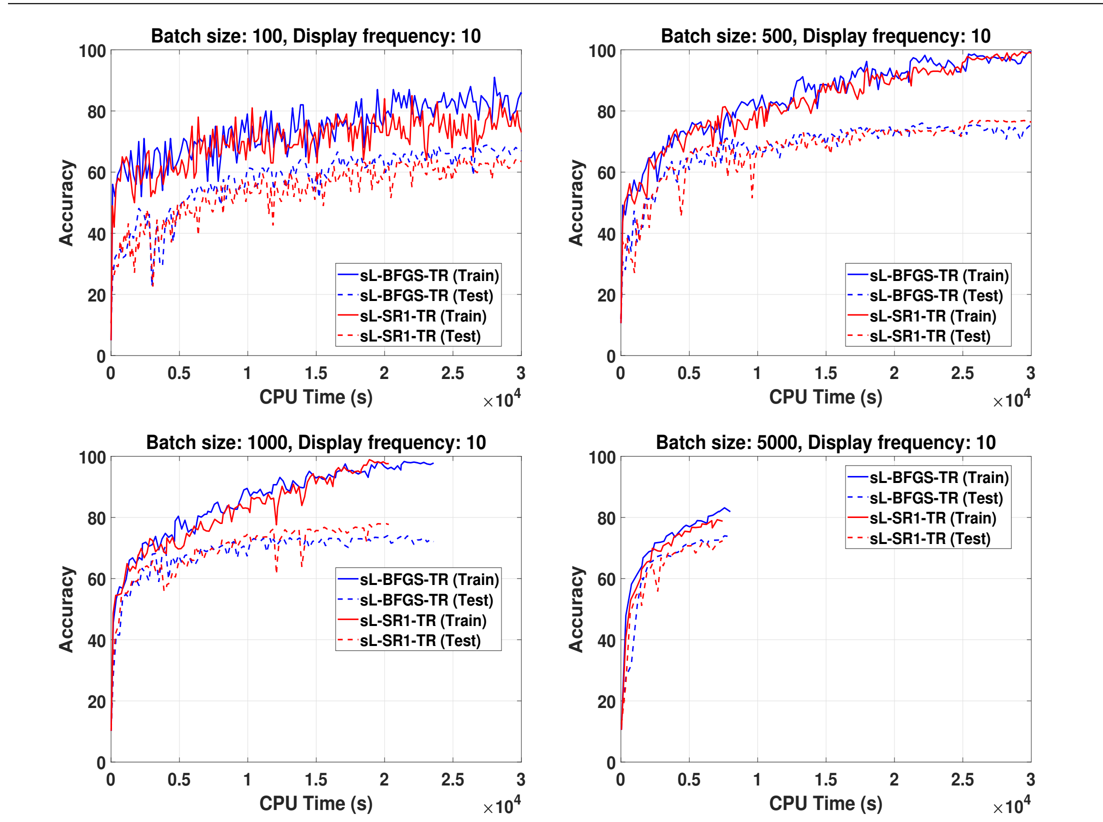



**Figure 4.18** The accuracy of sL-QN-TR vs time on `CIFAR10` with `ConvNet3FC2(no BN)`.

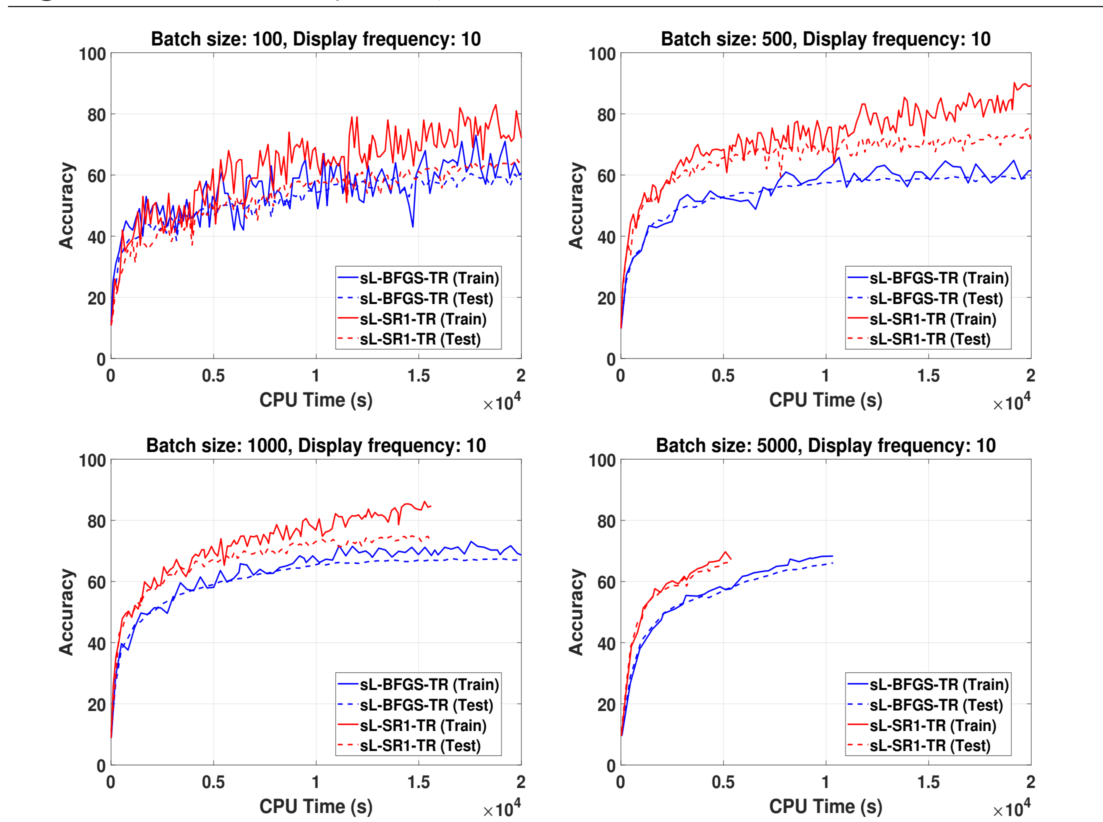


**Figure 4.19** Comparisons of sL-QN-TR and STORM (`CIFAR10`, `ConvNet3FC2`).

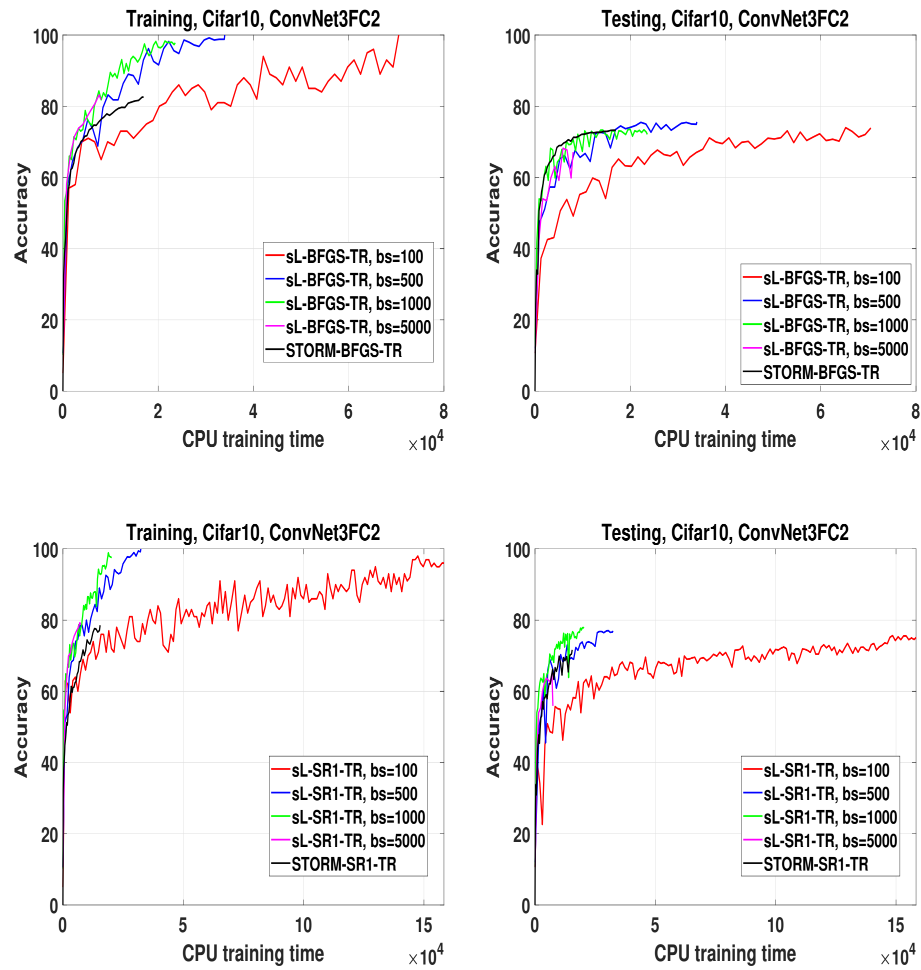



**Figure 4.20** Comparisons of sL-QN-TR and *tuned* Adam (`CIFAR10`, `ConvNet3FC2`).

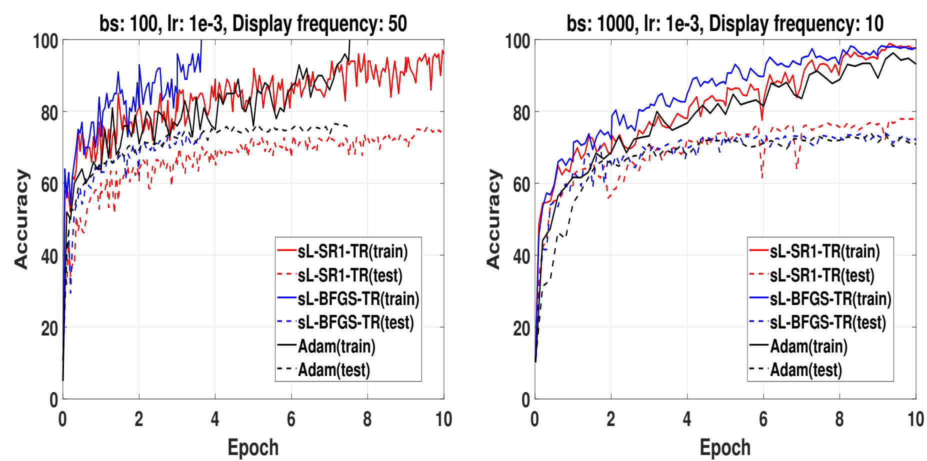


**Figure 4.21** Comparisons of sL-QN-TR and STORM (`Fashion-MNIST`, `ResNet-20`).

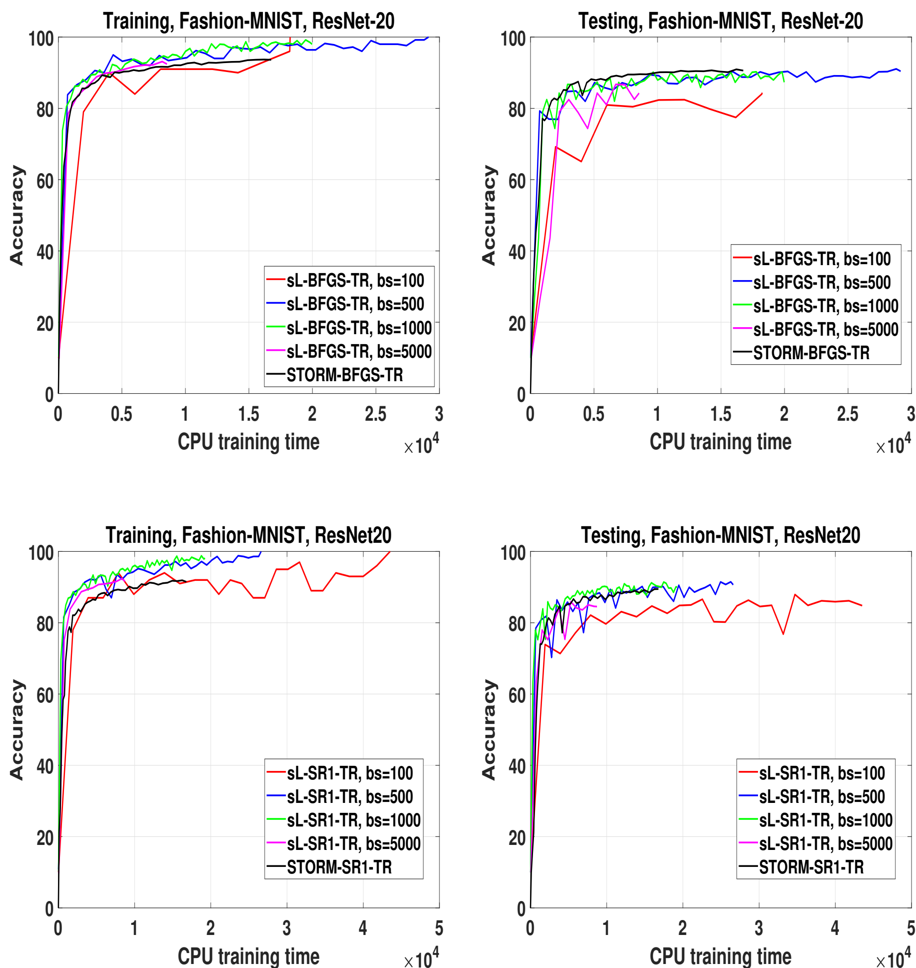



**Figure 4.22** Comparisons of sL-QN-TR and *tuned* Adam (`CIFAR10`, `ConvNet3FC2(no BN)`).

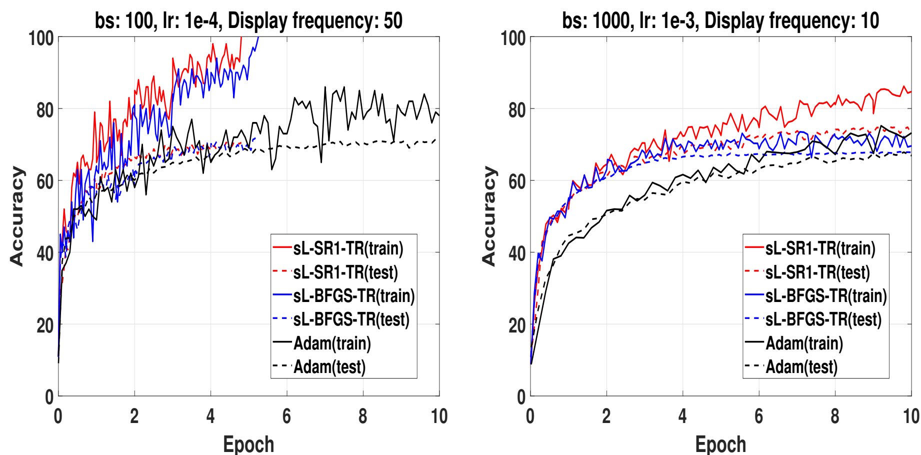


**Figure 4.23** Comparisons of sL-QN-TR and *tuned* Adam (`Fashion-MNIST`, `ResNet20`).

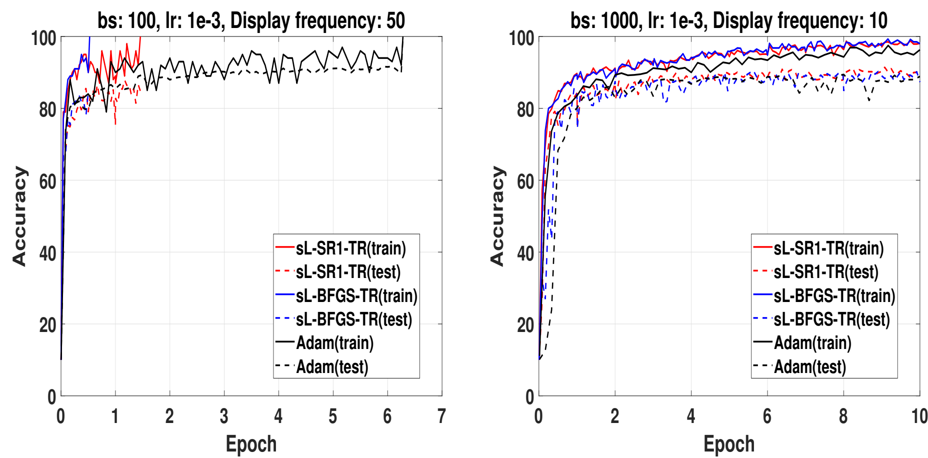



**Figure 4.24** Comparisons of sL-QN-TR and *tuned* Adam (`Fashion-MNIST`, `ResNet20(no BN)`).

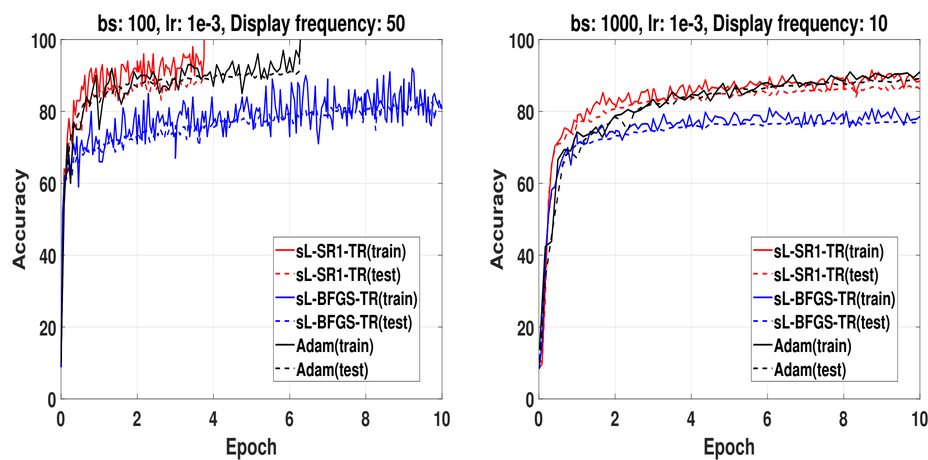
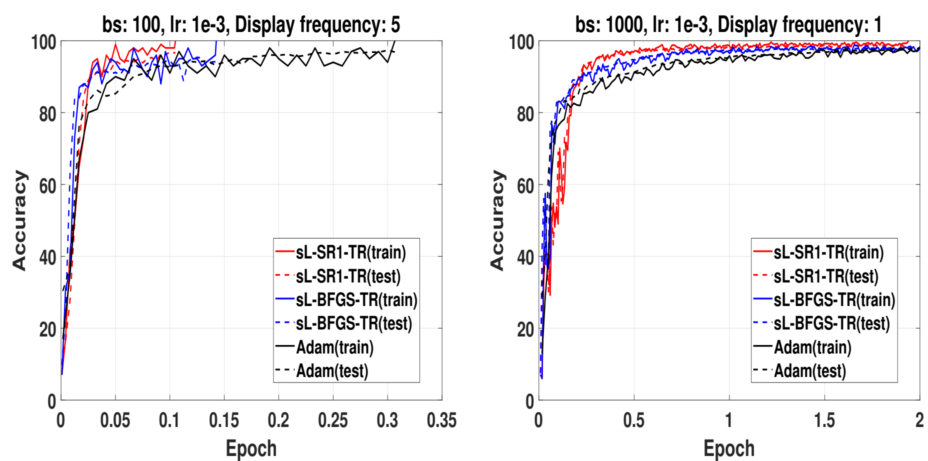



**Figure 4.25** Comparisons of sL-QN-TR and *tuned* Adam (`MNIST`, `LeNet-like`).



Now, we sum up the extensive experimental study to assess the performance of sL-BFGS-TR and sL-SR1-TR algorithms, limited memory Quasi-Newton trust-region methods using a specific fixed-size overlapping sampling.

1. Batch normalization (BN) layers: as the main findings, we have observed that BN is a key component for the performance of the algorithms with different sample sizes. sL-SR1-TR performs better than sL-BFGS-TR in the networks without BN layers. This behavior is in accordance with the property of L-SR1 updates allowing for indefinite Hessian approximations in non-convex optimization. However, sL-BFGS-TR behaves comparably or slightly better than sL-SR1-TR with BN layers.

2. Limited memory parameter $l$: the experiments, networks without BN layers, in particular, illustrated that larger values of $l$ produce higher accuracy. However, in order to reduce the computational cost, these values are not considered too big.

3. Batch size: larger batch sizes within a fixed number of epochs lead to lower training accuracy compared to using smaller batch sizes. However, longer training with larger batch sizes can help recover the lost accuracy.

4. Timing: the experiments on training time have shown a slight superiority in the accuracy reached by both algorithms when larger batch sizes are used within a fixed budget of time.

5. Comparison with STORM: within the fixed budget of time, except for the smallest batch size, both algorithms reveal more efficiency than the second-order STORM algorithm customized by L-BFGS and L-SR1 updates.

6. Comparison with Adam: the results show that sL-BFGS-TR and tuned Adam can not be better than sL-SR1-TR for the networks without BN layers. For the networks with BN layers, the sL-QN-TR algorithms, in particular sL-BFGS-TR, exhibited comparable accuracy with tunned Adam.

## 4.2 A Stochastic Modified L-BFGS Trust-Region Method

In this section, we define a modified L-BFGS update obtained through a modified secant condition which is theoretically shown to provide an increased order of accuracy in the Hessian approximation. We aim at extending this idea to a stochastic setting and devise a modified L-BFGS TR method for supervised learning applications.

### 4.2.1 A modified L-BFGS update

Let, for moments, $f(w)$ and $\nabla f(w)$ be any general deterministic objective function and its true gradient, respectively. A modified BFGS update $B_{k+1}$ as an approximation of the Hessian can be constructed by a *modified secant condition* as follows

$$B_{k+1}s_k = y_k^*, \tag{4.24}$$

where $(s_k, y_k^*)$ gives better curvature information than $(s_k, y_k)$ in the standard secant condition $B_{k+1}s_k = y_k$ with $s_k = w_{k+1} - w_k$ and $y_k = \nabla f(w_{k+1}) - \nabla f(w_k)$. In [76], the vector $y_k^*$ was computed as

$$y_k^* = y_k + \frac{\psi_k}{\|s_k\|^2} s_k, \tag{4.25}$$

where $\psi_k = (f(w_k) - f(w_{k+1})) + (\nabla f(w_k) + \nabla f(w_{k+1}))^T s_k$. Definition (4.25) together with (4.24) provides more accurate curvature information. In fact, it can be proved that

$$\begin{aligned} s_k^T(\nabla^2 f(w_{k+1})s_k - y_k^*) &= \frac{1}{3} s_k^T (T_{k+1}s_k)s_k + O(\|s_k\|^4), \\ s_k^T(\nabla^2 f(w_{k+1})s_k - y_k) &= \frac{1}{2} s_k^T (T_{k+1}s_k)s_k + O(\|s_k\|^4), \end{aligned} \tag{4.26}$$

where $T_{k+1}$ is the tensor of the true function $f(w)$ at $w_{k+1}$ in the Taylor series as

$$f(w_k) = f(w_{k+1}) - \nabla f(w_{k+1})^T s_k + \frac{1}{2} s_k^T \nabla^2 f(w_{k+1}) s_k - \frac{1}{6} s_k^T \left(T_{k+1}s_k\right) s_k + O(\|s_k\|^4). \tag{4.27}$$

In [55], a simple modification of (4.25) was proposed as $y_k^* = y_k + \text{sign}(\psi_k)\frac{\psi_k}{\|s_k\|^2} s_k$ to handle the case $\psi_k < 0$. However, we show below that this modification does not provide any improvement.

#### Sign correction

Considering the equations in (4.26) together yields

$$\psi_k = \frac{1}{6} s_k^T (T_{k+1} s_k) s_k + O(\|s_k\|^4). \tag{4.28}$$

Let $\psi_k < 0$. Therefore, we have $s_k^T y_k^* = s_k^T y_k - \psi_k$ which leads to derive

$$\begin{aligned} s_k^T \nabla^2 f(w_{k+1}) s_k - s_k^T y_k^* &= s_k^T \nabla^2 f(w_{k+1}) s_k - \left(s_k^T y_k + \psi_k\right) + 2\psi_k \\ &= \frac{2}{3} s_k^T \left(T_{k+1} s_k\right) s_k + O(\|s_k\|^4). \end{aligned} \tag{4.29}$$

Equation (4.29) shows that the dominant error is even worse than the one in (4.26). Therefore, we suggest to use $y_k$ whenever $\psi_k < 0$; otherwise we can use $y_k^*$.

#### A new modified secant condition

Taking the derivative of both sides of (4.27) with respect to $s_k$ and premultiplying it by $s_k^T$, we have

$$\begin{aligned} s_k^T \nabla f(w_k) &= s_k^T \nabla f(w_{k+1}) - s_k^T \nabla^2 f(w_{k+1}) s_k + \frac{1}{2} s_k^T \left(T_{k+1} s_k\right) s_k + O(\|s_k\|^4) \\ &= 3\psi_k + s_k^T y_k + O(\|s_k\|^4). \end{aligned} \tag{4.30}$$

To derive the second equality in (4.30), see the second equation in (4.26). Considering equations (4.27) and (4.30) together yields that the third order term disappears and

$$\begin{aligned} s_k^T \nabla^2 f(w_{k+1}) s_k &= 6(f(w_k) - f(w_{k+1})) + 3 s_k^T \left(\nabla f(w_{k+1}) + \nabla f(w_k)\right) + s_k^T y_k + O(\|s_k\|^4) \\ &= 3\psi_k + s_k^T y_k + O(\|s_k\|^4). \end{aligned} \tag{4.31}$$

Comparing (4.31) and the first equation in (4.26) suggests the choice of

$$y_k^* = \frac{3\psi_k}{\|s_k\|^2} s_k + y_k. \tag{4.32}$$

Obviously, the new vector $y_k^*$ provides a better curvature approximation, i.e., its error is of order $O(\|s_k\|^4)$, than the one of order $O(\|s_k\|^3)$ in equation (4.25).

**Algorithm 5** sM-LBFGS-TR

1: **Inputs:** $w_0 \in \mathbb{R}^n$, epoch$_{max}$, $l$, $\gamma_0 > 0$, $S_0 = Y_0 = [\,]$, $\delta_0 > 0$, $0 < \tau_1, \tau < 1$, bs
2: **while** epoch < epoch$_{max}$ or training accuracy < 100% **do**
3: **if** $k = 0$ **then**
4: Take index sets $\mathcal{O}_{-1}$ and $\mathcal{O}_0$ such that $|\mathcal{O}_{-1}| = |\mathcal{O}_0|$ and $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$ of size $N_k = bs$
5: Compute $f_{\mathcal{O}_{-1}}(w_0)$, $f_{\mathcal{O}_0}(w_0)$ and $g_0^{O_{-1}} \triangleq \nabla f_{\mathcal{O}_{-1}}(w_0)$, $g_0^{O_0} \triangleq \nabla f_{\mathcal{O}_0}(w_0)$
6: Evaluate $f_{\mathcal{N}_k}(w_0)$ and $g_0$ by (4.23)
7: **else**
8: Take $\mathcal{O}_k$ such that $|\mathcal{O}_{k-1}| = |\mathcal{O}_k|$ and $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$ of size $N_k = bs$
9: Compute $f_{\mathcal{O}_k}(w_k)$, $g_k^{O_k} \triangleq \nabla f_{\mathcal{O}_k}(w_k)$
10: Evaluate $f_{\mathcal{N}_k}(w_k)$ $g_k$ by (4.23)
11: **if** $\mathrm{mod}(k+1, \bar{N}) = 0$ **then**
12: Shuffle the data and epoch = epoch + 1
13: **end if**
14: **end if**
15: {Compute search direction}
16: **if** $k = 0$ **then**
17: Obtain $p_k = -\delta_k \dfrac{g_k}{\|g_k\|}$
18: **else**
19: Obtain $p_k$ using Algorithm C.3
20: **end if**
21: {Computation at $w_t$}
22: Evaluate $f_{\mathcal{O}_k}(w_t)$ and $g_t^{O_k} \triangleq \nabla f_{\mathcal{O}_k}(w_t)$ at the trial $w_t = w_k + p_k$
23: {Compute curvature pair and $\rho_k$}
24: Compute $(s_k, y_k)$ and $\rho_k$ by (4.20) and (4.22)
25: Compute $\psi_k = (f_{\mathcal{O}_k}(w_k) - f_{\mathcal{O}_k}(w_t)) + s_k^T(g_k^{O_k} + g_t^{O_k})$
26: **if** $\mathrm{sign}(\psi_k) > 0$ **then**
27: $y_k^* = y_k + \frac{3\psi_k}{\|s_k\|^2} s_k$
28: **end if**
29: **if** $\rho_k \geq \tau_1$ **then**
30: $w_{k+1} = w_t$ {Update $w_k$}
31: **else**
32: $w_{k+1} = w_k$
33: **end if**
34: Update $\delta_k$ by Algorithm C.1 {Update $\delta_k$}
35: **if** $s_k^T y_k > \tau \|s_k\|^2$ **then**
36: **if** $k < l$ **then**
37: Store $s_k$ and $y_k^*$ as new columns in $S_{k+1}$ and $Y_{k+1}$ {Update $B_k$}
38: **else**
39: Keep only $l$ recent $\{s_j, y_j^*\}_{j=k-l+1}^{k}$ in $S_{k+1}$ and $Y_{k+1}$
40: **end if**
41: Compute $\gamma_{k+1}$ for $B_0$ by Algorithm C.2 and $B_{k+1}$ by (4.9)
42: **else**
43: Set $B_{k+1} = B_k$
44: **end if**
45: $k = k + 1$
46: **end while**

*Algorithm's note:* $\delta_0 = 1,\ \gamma_0 = 1,\ \tau_1 = 10^{-6},\ \tau = 10^{-2},\ l = 20,\ \text{epoch}_{max} = 10.$

The modified L-BFGS Hessian approximation of the true objective function $f(w)$ can also be taken into account in a stochastic extension and integrated within a TR framework in a similar fashion described in Section 4.1. In the following subsection, we introduce the stochastic modified L-BFGS TR method (sM-LBFGS-TR).

### 4.2.2 Algorithm Framework

The sM-LBFGS-TR algorithm which is outlined in Algorithm 5 allows the random index set $\mathcal{N}_k$ of fixed-size to be chosen with half-overlapping such that $\mathcal{N}_k = \mathcal{O}_{k-1} \cup \mathcal{O}_k$ given the overlapped index set $\mathcal{O}_k$. The subsampled functions and subsampled gradients of the method are evaluated by (4.23) with respect to $\mathcal{N}_k$. By these stochastic quantities, we define a TR approach as follows:

- the L-BFGS Hessian approximations $B_k$ in the TR quadratic model satisfy the modified secant condition (4.24). Analogous to what was described in Section 4.1.1, the compact form of the modified L-BFGS matrix is constructed by only $l \ll n$ recent pairs $\{s_j, y_j^*\}$ rather than $\{s_j, y_j\}$ through storage matrices $S_k$ and $Y_k$. We note that computing $\psi_k$ for $y^*$ does not impose any additional cost. The resulting compact form of the modified L-BFGS matrix still benefits from the OBB solver to solve the corresponding TR subproblem for the search direction $p_k$.

- the standard vector $y_k$ in $y_k^* = y_k + \frac{3\psi_k}{\|s_k\|^2} s_k$ is obtained with respect to an overlap index set $\mathcal{O}_k$ as (4.20) in Remark 4.2.

- the value of reduction TR ratio $\rho_k$ is also evaluated with respect to $\mathcal{O}_k$ as (4.22) in Remark 4.3. Then, the value is applied for adjusting the radius of the region $(\delta_k)$ and accepting the trial point $(w_t = w_k + p_k)$.

Taking into account the aforementioned points, the framework of the sM-LBFGS-TR algorithm is similar to the sL-BFGS-TR (Algorithm 3). In the following subsection, we evaluate the performance of the proposed method for training DNNs.

### 4.2.3 Numerical Evaluation

Let us consider sM-LBFGS-TR and its naive variant named s-LBFGS-TR where the Hessian approximations $B_k$ satisfy the standard secant condition (4.3); in fact, we have implemented s-LBFGS-TR as sM-LBFGS-TR where $y_k^* = y_k$, and thus lines 25-28 are ignored. We study in this subsection the behavior of these algorithms on the training of `LeNet-like` and `ConvNet3FC2` for image classification of the benchmark datasets `MNIST` and `CIFAR10`. The training images of `CIFAR10` have been normalized by the z-score approach. We also provide a comparison with the most popular first-order method Adam by a grid search tuning effort on learning rate and batch sizes. The best learning rate for all batch sizes is $10^{-3}$. The algorithms have been implemented with the same random seed generator. The limited memory parameter for both QN TR methods was set to $l = 20$. The networks were trained for at most 10 epochs, and training was terminated if 100% accuracy has been reached. Figures 4.26 and 4.27 show the evolution of loss and accuracy for different batch sizes $|\mathcal{N}_k| \in \{100, 500, 2000, 5000\}$ in the classification of `MNIST` and `CIFAR10`, respectively. The results corresponding to the smallest batch size for the `MNIST` dataset are reported within the first epoch only to facilitate the comparison. The evolution curves for the `CIFAR10` have been filtered by a fixed display frequency for better visualization.

We observe from Figures 4.26 and 4.27 that both sL-BFGS-TR and sM-LBFGS-TR perform better than *tuned* Adam independently of the batch size. In all the experiments, sM-LBFGS-TR exhibits a comparable performance with respect to sL-BFGS-TR. The stochasticity could be the main reason why the responsiveness of modification in sM-LBFGS-TR is not seen in training tasks when compared to sL-BFGS-TR. However, due to the slightly higher accuracy of sM-LBFGS-TR compared to sL-BFGS-TR with the largest batch size, see Figure 4.26, using sM-LBFGS-TR can be recommended. Neither sL-BFGS-TR nor sM-LBFGS-TR is strongly influenced by batch sizes. Large batch sizes can be employed without a considerable loss of accuracy even though the performance of both methods decreases when larger batch sizes are used, due to the fewer number of iterations per epoch. The tuned Adam provides comparable accuracy to the second-order methods, but it is less accurate when large batch sizes are used.

**Figure 4.26** The comparative behavior of sM-LBFGS-TR (`MNIST`, `LeNet-like`).

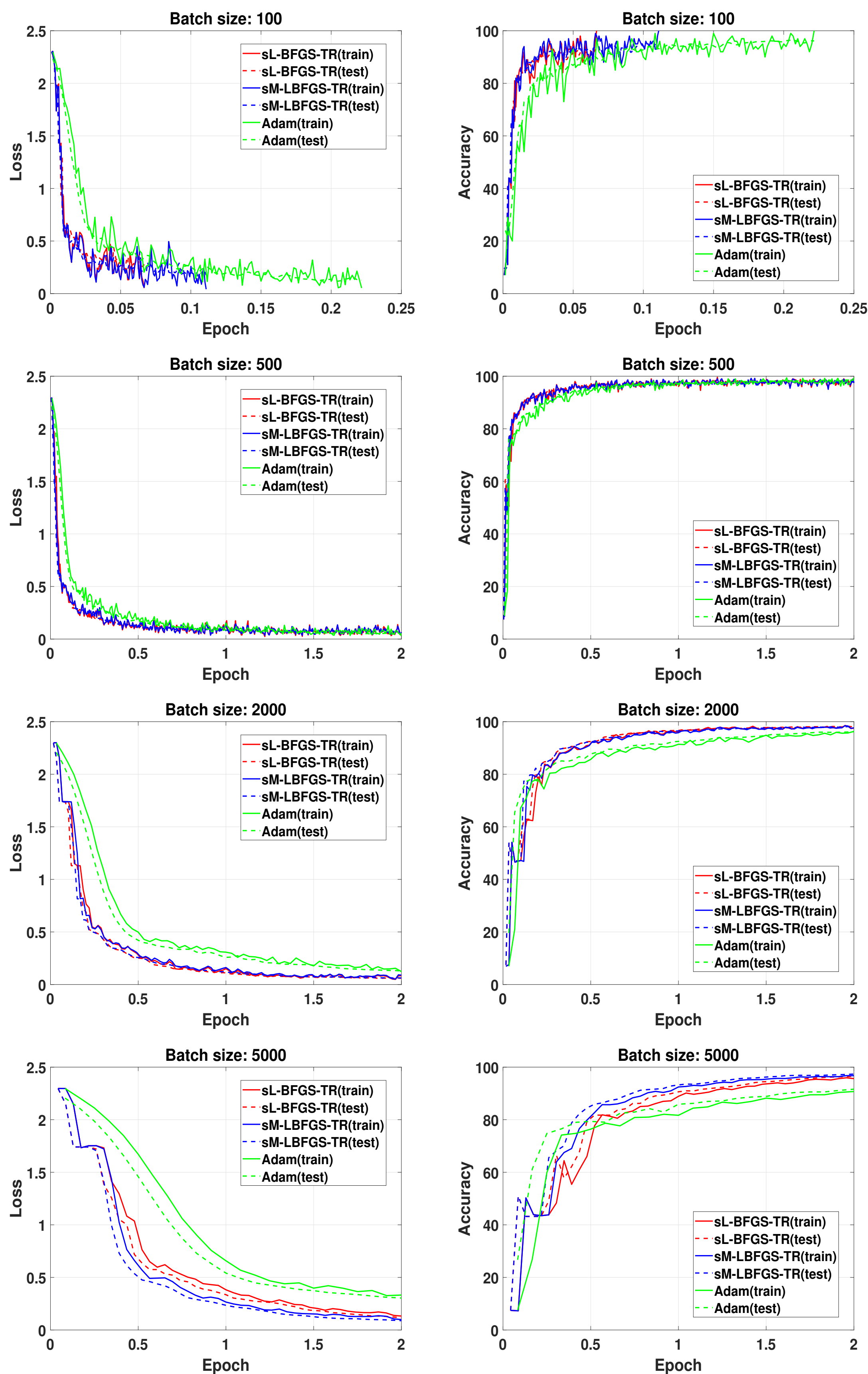


**Figure 4.27** The comparative behavior of sM-LBFGS-TR (CIFAR10, ConvNet3FC2).

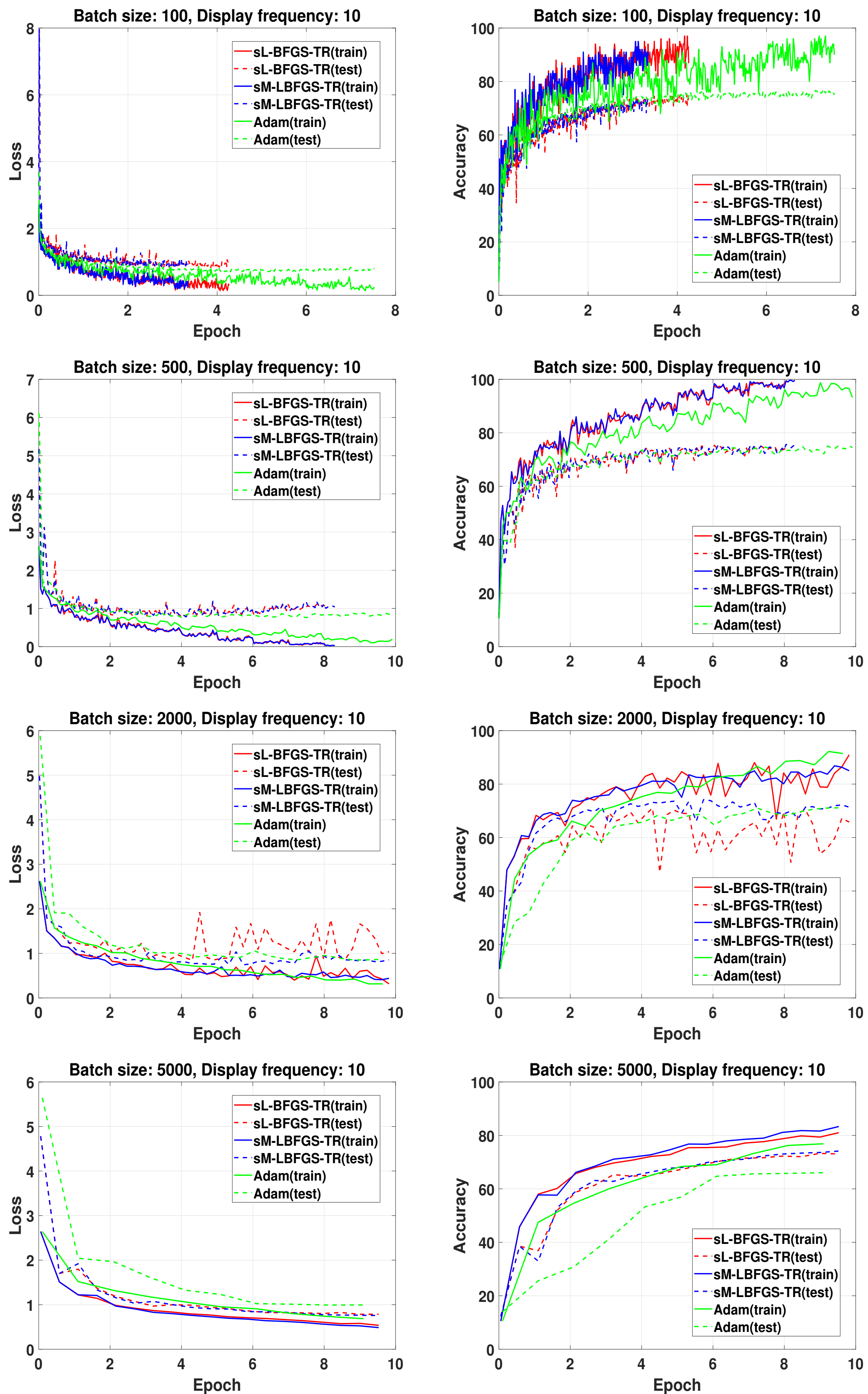


**Figure 4.28** Error bars of sM-LBFGS-TR, sL-BFGS-TR and *tuned* Adam.

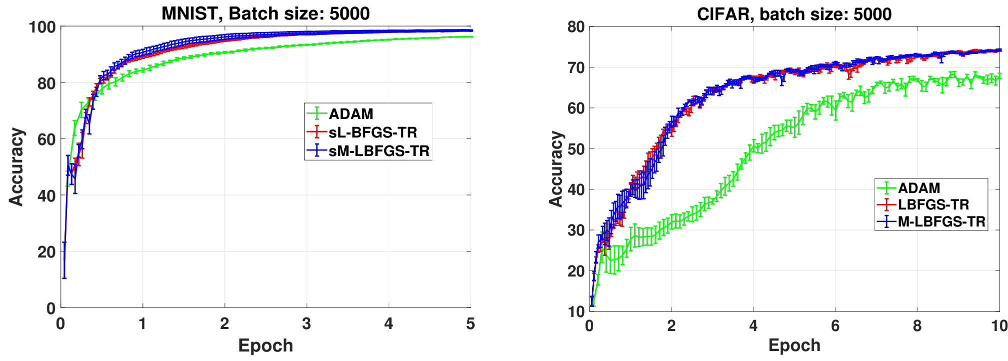



**Figure 4.29** The comparative behavior of sM-LBFGS-TR vs CPU time.

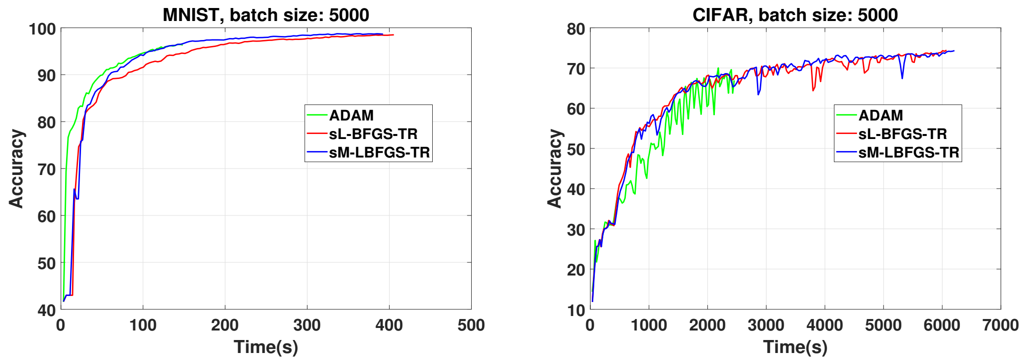



Figure 4.28 displays the variability of the obtained test accuracy computed over five runs corresponding to five different random seeds. It can be seen that the results are reliable and that tuned Adam exhibits larger variability than sL-BFGS-TR and sM-LBFGS-TR with the largest batch size.

It is not surprising that the measured CPU training time of sM-LBFGS-TR is comparable to that of M-LBFGS-TR due to their similar complexity, and that the training time of both algorithms is larger than that of Adam. Nevertheless, as Figure 4.29 illustrates, we underline the fact that with a fixed computational time budget both second-order methods provide comparable or better testing accuracy than tuned Adam.

In summary, although the modified secant condition provides theoretically an increased order of accuracy in the Hessian approximation, this modification in conjunction with the subsampled trust-region method could only produce a comparable or slightly better accuracy than its naive variant in which the standard L-BFGS is used instead of the modified L-BFGS, in particular when the largest batch size is used.

## 4.3 A Stochastic Hybrid L-SR1 Trust-Region Method

In Section 2.2, we mentioned that a line-search method for solving the unconstrained optimization problem (2.1) requires the search direction to be a descent direction, see Definition 2.5. However, this condition alone does not ensure convergence, and the search direction must satisfy the angle criterion (see e.g. [69])

$$\cos\left(-\nabla f(w_k),\, p_k\right) = \frac{-\nabla f(w_k)^T p_k}{\|\nabla f(w_k)\| \|p_k\|} \geq \epsilon_k, \quad \epsilon_k > 0, \tag{4.33}$$

where $\nabla f(w_k)$ is the gradient of the true objective function $f(w)$ at $w_k$, and the sequence $\{\epsilon_k\}$ is bounded away from 0, which means that the angle between the search direction ($p_k$) and the steepest descent direction ($-\nabla f(w_k)$) must be bounded away from the right angle. In order to address the case in which (4.34) does not hold, a globalization approach applicable to any Newton-type method was proposed in, e.g., [24]. The basic idea consists of linearly combining the Newton-type (e.g., Newton or Quasi-Newton) and Steepest Descent (SD) directions. The goal is to bring the iterates sufficiently close to a solution through the globally convergent Steepest Descent method so that once the iterates are in the basin of attraction of the Newton-type method, it can lead to faster convergence. We aim at extending the mentioned idea in a stochastic regime to devise an algorithm for DNNs training.

### 4.3.1 Algorithm Framework

We present a stochastic combined LSR1 TR method (sCLSR1-TR) and clearly described it in Algorithm 6. This method allows a random index set $\mathcal{N}_k$ of fixed-size to be chosen regularly, i.e., a random mini-batch of samples without overlapping. The subsampled functions and subsampled gradients of the method are evaluated by (4.23) with respect to $\mathcal{N}_k$. By these stochastic quantities, we define a stochastic TR approach. This approach involves L-SR1 Hessian approximation and produces corresponding second-order LSR1 direction $p_k$. The algorithm applies this direction as long as

$$\cos\left(-v_k,\, p_k\right) = \frac{-v_k^T p_k}{\|g_k\| \|p_k\|} \geq \epsilon_k, \quad \epsilon_k > 0, \tag{4.34}$$

where $v_k$ simulates the behavior of the SD direction in the stochastic expansion. In fact, $v_k$ is an approximation of the SD direction. At every iteration where the condition (4.34) is not satisfied, the algorithm computes a new direction by a combination of the first- and second-order directions. We propose a combination strategy by which a hybrid direction is defined as follows

$$p_k^h = \beta_k p_k - (1-\beta_k)\xi_k v_k, \tag{4.35}$$

where $0 \le \beta_k \le 1$ and $\xi_k > 0$. In (4.35), $\xi_k$ scales the first-order direction $v_k$ and $\beta_k$ guarantees that the combined direction $p_k^h$ in (4.35) satisfies (4.34), i.e., $\cos(-v_k, p_k^h) \ge \epsilon_k$. Our method computes the parameter $\beta_k$ as it is obtained according to Theorem 1 in [24] where $\beta_k$ is a lower bound of the smallest root in $(0,1)$ of the second-order polynomial; i.e.,

$$\beta_k = \frac{r_k}{r_k + \pi_k}, \tag{4.36}$$

where

$$r_k = \xi_k(1-\epsilon_k), \qquad \pi_k = \frac{v_k^T p_k}{\|v_k\|^2} + \epsilon_k \frac{\|p_k\|}{\|v_k\|}.$$

Given the search direction at iteration $k$, the proposed algorithm finds a suitable step length through a line-search step and then updates $w_k$. The line-search step in line 22 of sCLSR1-TR avoids resolving the TR subproblem due to occasional rejections of iterates. We also relax the progress of $\rho_k$ a stochastic term by averaging past ratios; in fact, we soften its effect. This averaging value, i.e., $\hat{\rho}_k$ in line 27 of sCLSR1-TR, is utilized to adjust the radius of the TR ($\delta_k$).

#### Selecting the scaling parameter $\xi_k$ and the first-order direction $v_k$

A suitable scaling of $v_k$ is a key issue in making the proposed approach effective. In deterministic cases, there are different strategies to choose this parameter, see [24] and references therein. In our algorithm, we adaptively set this parameter as follows

$$\xi_k = \frac{\delta_k}{\|v_k\|}, \tag{4.37}$$

to ensure that the combined direction $p_k^h$ is pushed to be still inside the trust-region. As already mentioned, the first-order direction $v_k$ in (4.35) plays the role of the SD direction even if it is only a noisy approximation of the true SD direction due to the error coming from the use of subsampling. There is a *variance reduction* family of algorithms that exploit the full gradient information strategically in an attempt to reduce or eliminate the noise present in gradient approximations; they save the information of a full gradient which is evaluated at a reference point, known as *snapshot gradient*, and modify it using stochastic gradient estimates to form the variance-reduced gradient in subsequent iterations. Therefore, a natural choice of $v_k$ can be a first-order variance reduced (VR) direction. Three representative VR directions obtained using mini-batch versions of SAGA, SVRG, and SARAH algorithms at iteration $k$ are listed below:

1. SAGA:

$$v_k = \nabla f_{\mathcal{N}_k}(w_k) - \frac{1}{|\mathcal{N}_k|} \sum_{i \in \mathcal{N}_k} J_{:,i}^k + \frac{1}{|\mathcal{N}|} \sum_{i \in \mathcal{N}} J_{:,i}^k, \tag{4.38}$$

   where

$$J_{:,i}^{k+1} = \begin{cases} J_{:,i}^k, & \text{if} \quad i \notin \mathcal{N}_k, \\ \frac{1}{|\mathcal{N}_k|} \sum_{i \in \mathcal{N}_k} \nabla f_i(w_k), & \text{if} \quad i \in \mathcal{N}_k, \end{cases}$$

   and $J_{:,i}^0 = \nabla f_i(w_0)$.

2. SVRG:

$$v_k = \nabla f_{\mathcal{N}_k}(w_k) - \nabla f_{\mathcal{N}_k}(w_s) + \nabla f(w_s), \tag{4.39}$$

   where $s$ is a previous iteration ($s < k$).

3. SARAH:

$$v_k = \nabla f_{\mathcal{N}_k}(w_k) - \nabla f_{\mathcal{N}_k}(w_{k-1}) + v_{k-1}, \quad v_s = \nabla f(w_s). \tag{4.40}$$

**Algorithm 6** sCLSR1-TR

1: Inputs: $k = 0,\ w_0,\ S_0 = Y_0 = [\,],\ \gamma_0, \delta_0 > 0,\ T = 0, \epsilon,\ \nu \in (0, 1), \mathrm{bs},\ \texttt{eps}$: machine epsilon
2: **for** $epoch = 1, 2, \ldots,$ **do**
3: Shuffle $N$ samples for randomly creating $N_b$ mini-batches
4: **for** $iter = 1, 2, \cdots, N_b$ **do**
5: Compute $f_{\mathcal{N}_k}(w_k)$ and $g_k \triangleq \nabla f_{\mathcal{N}_k}(w_k)$ with respect to given $\mathcal{N}_k$ of size $N_k = bs$
6: **if** Some stopping conditions hold **then**
7: Stop training
8: **end if**
9: **if** $k = 0$ or $S_k = [\,]$ **then**
10: Set $B_k = \gamma_0 I$, and compute $p_k = \frac{-\delta_k g_k}{\|g_k\|}$
11: **else**
12: Compute $B_k = \gamma_k I + \Psi_k M_k^{-1} \Psi_k^T$, and find $p_k$ by OBS solver Algorithm C.5
13: **end if**
14: Find $v_k$ as a first-order direction
15: **if** $\cos(-v_k,\, p_k) \geq \epsilon_k$ **then**
16: $p_k^h = p_k$
17: $\epsilon_{k+1} = \epsilon_k$
18: **else**
19: Compute $p_k^h = \beta_k p_k - (1 - \beta_k)\xi_k v_k$ such that $\cos(-v_k,\, p_k^h) \geq \epsilon_k$
20: $\epsilon_{k+1} = \max\{10\,\texttt{eps}, \nu\epsilon_k\}$
21: **end if**
22: Find a step-length $\alpha_k$ by backtracking and Armijo rule; i.e.

$$f_{\mathcal{N}_k}(w_k + \alpha_k p_k^h) \leq f_{\mathcal{N}_k}(w_k) + c_1 \alpha_k v_k^T p_k^h$$

23: Set $w_{k+1} = w_k + \alpha_k p_k^h$
24: Evaluate $f_{\mathcal{N}_k}(w_{k+1})$ and $g_{k+1} \triangleq \nabla f_{\mathcal{N}_k}(w_{k+1})$
25: Compute $s_k = \alpha_k p_k^h$, and $y_k = g_{k+1} - g_k$
26: Find $\gamma_{k+1}$ via Algorithm C.4, and construct a new *well-defined* $B_{k+1}$ as (4.18)
27: $\hat{\rho}_k = T\hat{\rho}_k + \rho_k$
28: $T = T + 1$
29: $\hat{\rho}_k = \frac{\hat{\rho}_k}{T}$
30: Update $\delta_k$ by Algorithm C.1 with $\hat{\rho}_k$ and $p_k^h$
31: **end for**
32: $k = k + 1$
33: **end for**

*Algorithm's note:* $\delta_0 = 1,\ \gamma_0 = 1,\ \epsilon_0 = 0.5,\ \nu = 0.95,\ c_1 = 10^{-4}$.

The *reduced memory* mini-batch SAGA described above requires computing the full gradient at the beginning of the algorithm and needs the *reduced memory* storage matrix of $N_b$ columns corresponding to the only $N_b$ previously computed stochastic gradients. In fact, since the same information is used for all the columns of $J^{k+1}$ that belong to $\mathcal{N}_k$, there is no need to store all identical information. Let's assume the index set $\mathcal{N}$ is partitioned into a fixed-number $N_b$ of random mini-batches of the same size. We initialize $J^0$ so that all the columns belonging to the same partition are the same, then they will be the same within each partition for all $k$. In such a case, we do not need to maintain all the identical copies; instead, we can update individually the columns of a compressed version of the Jacobian, with one column per partition set only, to reduce the total memory usage [34]. In contrast, the mini-batch SVRG does not require extra storage but it does require computing the full gradient periodically; see e.g. [64]. In fact, at each outer iteration, SVRG computes one full gradient $\nabla f(w_s)$, where $s < k$ is a previous iteration, then takes $t$ steps along random directions which are stochastic corrections of this full gradient. SARAH combines some ideas from SVRG and SAGA and differs from SVRG in terms of the inner loop step. The SARAH VR direction (4.40) involves a gradient snapshot which is also updated at each inner iteration and thus attains improved convergence properties relative to SVRG. See respectively e.g. [34], [64], and [61] for details on the VR directions listed above.

We select a reduced memory SAGA gradient as the first-order search direction $v_k$. With this approach, the hybrid algorithm needs one full gradient and a storage matrix with $N_b$ columns of length $n$. Moreover, we can get rid of finding the optimum value of $t$ as well as computing a full gradient periodically every $t$ iterations. We should notice that, in practice, SVRG can be quite sensitive to the choice of $t$; see [20, 41].

### 4.3.2 Numerical Evaluation

We applied Algorithm 6 (sCLSR1-TR) to the solution of the $L2$ regularized DL optimization problem (3.4) for the training of `ResNet20` to classify the `CIFAR10` dataset which is normalized by z-score approach. We conducted a performance comparison between sCLSR1-TR and tuned Adam. Both algorithms were implemented using the same seed

for the MATLAB random number generator. We considered two values of batch sizes as $N_k = bs$ with $bs = 500$ and $bs = 1000$. Different values of the $L2$ regularization parameter for sCLSR1-TR were tested, namely $\lambda \in \{10^{-2}, 10^{-3}, 10^{-4}, 0\}$. For grid-searching Adam, we tested regularization parameter $\lambda \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$ and learning rate $\alpha \in \{10^{-1}, 10^{-2}, 10^{-3}\}$. The limited memory parameter for LSR1 Hessian approximation was set to $l = 30$. The evolution of the accuracy in Figures 4.30–4.33 is reported by $c$-point mean values of the obtained accuracy at each iteration, where each mean value is computed over a sliding window of length $c$ across neighboring elements. We set $c = 20$.

Unlike Adam, which is sensitive to the values of its hyperparameters (including the $L2$ regularization parameter $\lambda$ and the learning rate $\alpha$), we can observe from Figure 4.30 and Figure 4.31 that sCLSR1-TR provides similar accuracy for all tested values of $\lambda$, except for the largest one, i.e., $\lambda = 10^{-2}$. This indicates that sCLSR1-TR can be employed for training purposes without the need to apply $L2$ regularization. Alternatively, if regularization is desired, it is recommended to use small values of the parameter $\lambda$. As displayed in Figure 4.32, the proposed algorithm, without regularization and within a fixed number of epochs, achieves similar or superior performance compared to tuned Adam. Regarding the training time, each iteration of the proposed hybrid algorithm requires more time on a GPU compared to tuned Adam, due to its higher iteration complexity. Nevertheless, given a fixed budget of time (90 minutes) in our experiments, the results in Figure 4.33 show that sCLSR1-TR ultimately achieves a higher level of testing accuracy ($bs = 500$) or the same level of it ($bs = 1000$) as tuned Adam.

**Figure 4.30** The effect of L2 regularization parameter on the training of Adam

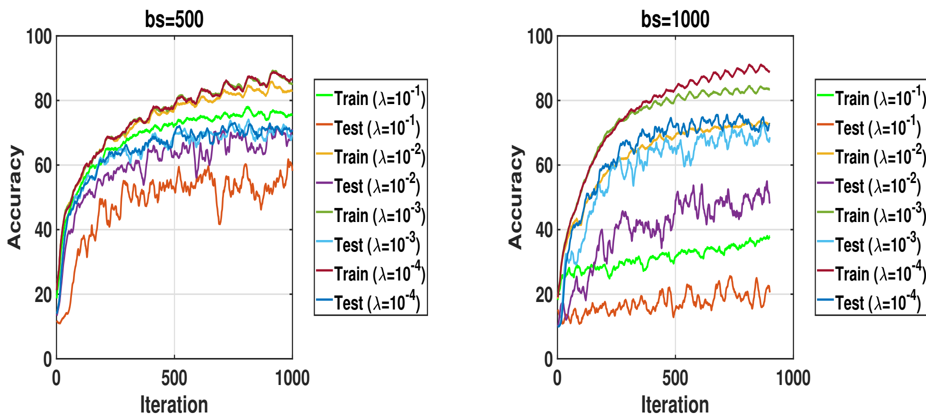


**Figure 4.31** The effect of L2 regularization parameter on the training of sCLSR1-TR

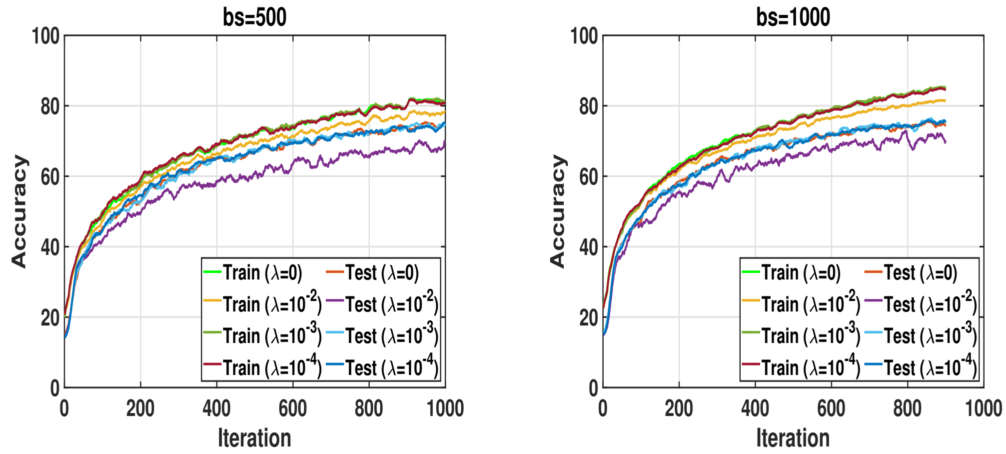



**Figure 4.32** The accuracy vs iteration evolution of sCLSR1-TR and *tuned* Adam

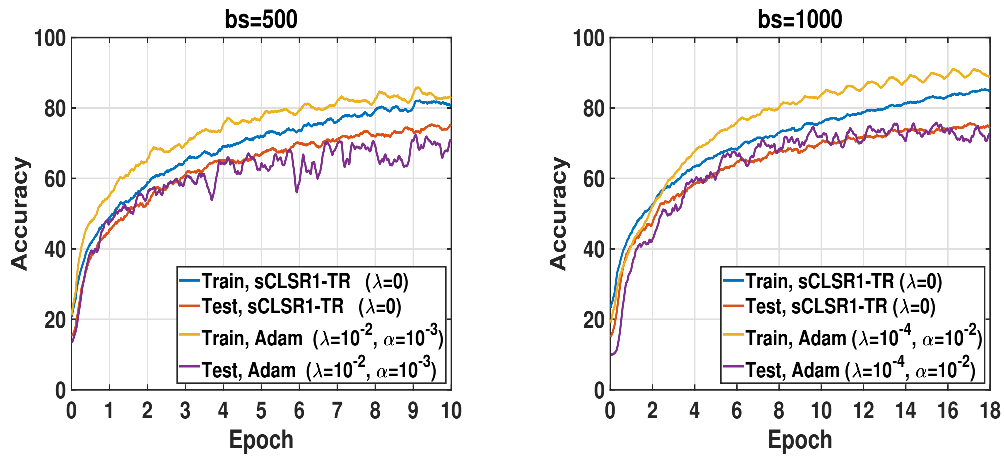
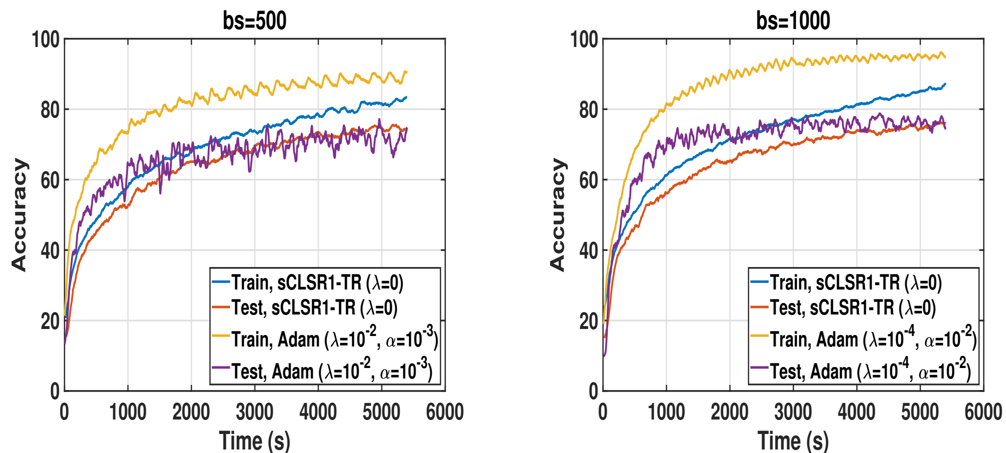



**Figure 4.33** The accuracy vs time evolution of sCLSR1-TR and *tuned* Adam




In summary, the proposed algorithm exhibits comparable or superior performance to the Adam while requiring significantly less tuning effort for the regularization parameter.

# 5

# Stochastic Non-Monotone Trust-Region Methods

## 5.1 Introduction

Throughout Chapter 4, our primary emphasis has been on stochastic trust-region (TR) methods, with a significant reliance on Quasi-Newton (QN) Hessian approximations. In the subsequent sections of this chapter, our objective shifts towards solving the DL optimization problem (3.3) within a non-monotone trust-region (NTR) framework in stochastic regimes, while considering inexact function and gradient approximations.

The potential usefulness of non-monotonicity can be traced back to [35], where a non-monotone line-search Newton method was proposed for solving general unconstrained minimization problems with any twice continuously differentiable function denoted as $F(.)$. Modifications of the Newton method for global convergence towards local minima require a line-search technique ensuring a monotonic decrease, which can sometimes slow down convergence. In contrast, non-monotone line-search relaxes some of the standard line-search conditions, allowing for an increase in function values without affecting convergence properties. In [35], the proposed non-monotone line-search requires the step-length $\alpha_k$ to satisfy the following condition

$$F(w_k + \alpha_k p_k) \leq F_{m(k)} + c_1 \alpha_k \nabla F(w_k)^T p_k, \tag{5.1}$$

where $c_1 \in (0, 0.5)$ and $F_{m(k)}$ is the maximum function value of a prefixed number ($L_n$) of previous iterates, i.e., $F_{m(k)} = \max_{0 \leq j \leq m(k)}\{F(w_{k-j})\}$ for $k = 0, 1, \ldots,$ in which $m(0) = 0$, $0 \leq m(k) \leq \min\{m(k-1)+1, L_n\}$ for all $k \geq 1$, and $L_n \geq 0$. The idea of incorporating non-monotonicity into a TR strategy, which is a generalization of the Armijo line-search approach [73], has been explored in previous works such as [22, 2, 19]. The most common TR reduction ratio in a non-monotone regime is defined as

$$\rho_k^{NTR} = \frac{F(w_k + p_k) - F_{m(k)}}{m_k(p_k)}, \tag{5.2}$$

where the denominator $m_k(\cdot)$ is a quadratic model associated with $F(.)$. There are also some modifications for (5.2) where the non-monotone term $F_{m(k)}$ is replaced with $c_k$ (see e.g. [19]) or $r_k$ (see e.g. [2]), where $c_k$ is defined as a weighted moving average of objective function values and $r_k$ which is a convex combination of $F_{m(k)}$ and latest objective function value.

To solve specific unconstrained optimization problems such as the DL minimization problem (3.3), one can also consider applying the aforementioned non-monotone techniques in both line-search and TR frameworks. However, the application of these techniques becomes computationally expensive when dealing with large sample sizes ($N$). Therefore, our objective is to extend these approaches to the stochastic regime to devise algorithms for the training of DNNs. In Section 5.2, we introduce a stochastic algorithm within a non-monotone TR framework, utilizing a regular fixed-size mini-batching approach. We further extend this study in Section 5.3 by presenting a novel stochastic non-monotone algorithm that incorporates adaptive subsampling.

## 5.2 A Stochastic Algorithm with Fixed-Size Sampling

In this section, we build upon the concept of non-monotonicity introduced in the TR framework proposed by [2] and extend it to the stochastic setting. Using inexact function and gradient approximations, we describe below a novel training algorithm with a stochastic non-monotone trust-region (NTR).

**Algorithm 7** sL-SR1-NTR

1: Inputs: $k = 0,\ w_0,\ S = Y = [\,],\ \gamma_0,\ \delta_0 > 0$
2: **for** *epoch* $= 1, 2, \ldots,$ **do**
3: Create randomly $N_b$ mini-batches of same carnality (sampling without replacement)
4: **for** *iter* $= 1, 2, \cdots, N_b$ **do**
5: Compute $f_{\mathcal{N}_k}(w_k)$ and $g_k \triangleq \nabla f_{\mathcal{N}_k}(w_k)$ with respect to given $\mathcal{N}_k$ of fixed-size $N_k$
6: **if** Some stopping conditions hold **then**
7: Stop training
8: **end if**
9: **if** $k = 0$ or $S = [\,]$ **then**
10: Set $B_k = \gamma_0 I$, and compute $p_k = \frac{-\delta_k g_k}{\|g_k\|}$
11: **else**
12: Compute $B_k = \gamma_k I + \Psi_k M_k^{-1} \Psi_k^T$, and find $p_k$ by OBS solver Algorithm C.5
13: **end if**
14: Set $w_t = w_k + p_k$ and compute $f_{\mathcal{N}_k}(w_t)$
15: Compute $\rho_k^{NTR}$ (5.3) using $r_k$ with $\tau_k$ defined in (5.5)
16: **if** $\rho_k > \eta_1$ **then**
17: Compute $s_k = p_k$, and $y_k$ as indicated in Table 5.1
18: Find $\gamma_{k+1}$ via Algorithm C.4, and construct a new *well-defined* $B_{k+1}$ as (4.18)
19: Set $w_{k+1} = w_t$
20: **else**
21: Find $\alpha_k$ by a stochastic non-monotone line-search (NLS); i.e.,

$$f_{\mathcal{N}_k}(w_k + \alpha_k p_k) \le r_k + c_1 \alpha_k g_k^T p_k$$

22: **if** NLS succeeds **then**
23: Set $w_t = w_k + \alpha_k p_k$ and $w_{k+1} = w_t$
24: Compute $s_k = \alpha_k p_k$, and $y_k$ as indicated in Table 5.1
25: Find $\gamma_{k+1}$ via Algorithm C.4, and construct a new *well-defined* $B_{k+1}$ as (4.18)
26: **else**
27: Skip updating $B_k$ and $w_k$
28: **end if**
29: **end if**
30: Update $\delta_k$ by Algorithm C.1 with $\rho_k$ and $s_k$
31: $k = k + 1$
32: **end for**
33: **end for**

*Algorithm's note:* $\delta_0 = 1,\ \gamma_0 = 1,\ c_1 = 10^{-4}$. See also Table 5.1.

### 5.2.1 Algorithm Framework

We present a stochastic L-SR1 non-monotone TR method (sL-SR1-NTR) and provide a clear description of it in Algorithm 7. Let's consider, for moments, the stochastic TR framework described in Section 4.1. In this framework, we have the evaluated quantities $f_{\mathcal{N}_k}(w_k)$ and $g_k$, representing the objective function and its gradient with respect to the index sample set $\mathcal{N}_k$ of size $N_k = |\mathcal{N}_k| = bs$. Additionally, we have an L-SR1 Hessian approximation $B_k$ for the TR quadratic model. Using these, we calculate a second-order search direction $p_k$ to find the trial point $w_t = w_k + p_k$. To adjust the TR radius ($\delta_k$) and determine whether to accept the trial point, the new method utilizes a stochastic NTR reduction ratio as follows

$$\rho_k^{NTR} = \frac{f_{\mathcal{N}_k}(w_t) - r_k}{Q_k(p_k)}, \tag{5.3}$$

where

$$r_k = \tau_k f_{m_k} + (1 - \tau_k) f_{\mathcal{N}_k}(w_k), \tag{5.4}$$

in which $\tau_k \in [\tau_{min}, \tau_{max}]$ with $\tau_{min} \in [0, 1)$, $\tau_{max} \in [\tau_{min}, 1]$, and $f_{m_k}$ is the non-monotone term. Considering function approximations in stochastic settings, the new ratio allows for a more relaxed agreement between the model and the approximated function. When the ratio $\rho_k$ indicates a good agreement between the model and the *relaxed* approximate function, the trial point is accepted, and the L-SR1 matrix, which is responsible for generating a reliable quadratic model, is updated. On the other hand, our algorithm incorporates a non-monotone line search step to handle situations where the ratios are small or negative due to an inaccurate model. Through this step, the algorithm first determines a suitable step length and then updates both the L-SR1 matrix and the parameter $w_k$. If the line search step fails to find a satisfactory solution, the algorithm proceeds to resolve the TR subproblem with the current parameter and L-SR1 matrix in the next iteration. See lines 16-29 of the sL-SR1-NTR algorithm.

What follows are some ways of selecting the non-monotonicity rate $\tau_k$ and the non-monotone term $f_{m_k}$ in the NTR reduction ratio (5.3). Additionally, we review some strategies for computing the curvature vector $y_k$ in order to update the LSR1 matrix.

**Selecting $\tau_k$ and $f_{m_k}$**

As considered in [2], the parameter $\tau_k$ determining the level of monotonicity can be updated as

$$\tau_k = \begin{cases} \frac{\tau_0}{2}, & \text{if } k = 1, \\ \frac{\tau_{k-1}+\tau_{k-2}}{2}, & \text{if } k \geq 2. \end{cases} \tag{5.5}$$

Considering the stochastic regime, however, the difference between our proposed NTR ratio and that proposed in [2] is the definition of $f_{m_k}$. Here, we list three possibilities in (5.4):

1. The trivial choice is the maximum recent (at most) $L_n$ subsampled functions' values, i.e.,

$$f_{m_k} = \max\{f_{\mathcal{N}_j}(w_j) \,|\, k - L_n + 1 \leq j \leq k\}, \qquad k = 0, 1, \ldots. \tag{5.6}$$

2. It may be possible to reduce the stochasticity of the first choice by taking $f_{m_k}$ as a reference point which is updated at specific iterations. Our second choice of $f_{m_k}$ is the maximum value of the recent $L_n$ which is computed every $L_n$ iterations, i.e.,

$$f_{m_k} = \begin{cases} \max\{f_{\mathcal{N}_j}(w_j) \,|\, k - L_n + 1 \leq j \leq k\}, & \text{if } k = 0 \text{ or } mod(k, L_n) = 0, \\ f_{m_{k-1}}, & \text{elsewhere.} \end{cases} \tag{5.7}$$

3. Since $f_{\mathcal{N}_k}(w_t)$ is evaluated with respect to $\mathcal{N}_k$, and $f_{m_k}$ might be obtained with respect to an index set different from $\mathcal{N}_k$, the numerator in (5.3) may suffer from the noise resulting from different mini-batches. Thus, our third (costly) choice of $f_{m_k}$ is the maximum value of the recent $L_n$ subsampled functions with respect to current index set $\mathcal{N}_k$, i.e.,

$$f_{m_k} = \max\{f_{\mathcal{N}_k}(w_j) \,|\, k - L_n + 1 \leq j \leq k\}, \qquad k = 0, 1, \ldots. \tag{5.8}$$

**Curvature Computing Strategies in $B_k$**

Now, let us describe different strategies for computing the curvature vector $y_k$ and how it is used to update $B_k$. It is known that a QN Hessian approximation such as an L-SR1 update is originally obtained by defining the iterate and gradient displacements as follows

$$s_k = p_k, \qquad y_k = g_t - g_k, \tag{5.9}$$

where $g_t \triangleq \nabla f_{\mathcal{N}_k}(w_t)$. The inherently overwriting process of updating the QN Hessian approximation can result in a single poor update that can have long-lasting effects on several subsequent iterations. This is because the curvature estimates $y_k$ need to accurately capture the behavior of the entire objective function's Hessian in the DL optimization problem (3.3), which is not achieved by using subsampled gradient differences based on small samples. To address this issue and achieve a more stable Hessian approximation, an effective approach is to separate the calculations of stochastic gradients used for parameter updates from the computations of $y_k$ [13, 16]. In this decoupling approach, if necessary, one can use a different and larger random mini-batch for computing both gradients involved in $y_k$ (5.9). By considering the first-order Taylor expansion to approximate the gradient difference, an alternative strategy is to use a subsampled Hessian-vector product, which can provide a better representation of the true Hessian's action [13]. To do so, a different and large enough random index set $\mathcal{N}_k^B$ of size $N_k^B = |\mathcal{N}_k^B|$ is considered for computing $y_k$ such that

$$s_k = p_k, \qquad y_k = \bar{B}_k s_k, \tag{5.10}$$

where

$$\bar{B}_k \triangleq \nabla^2 f_{\mathcal{N}_k^B} = \frac{1}{N_k^B} \sum_{i \in \mathcal{N}_k^B} \nabla^2 f_i(w_k).$$

Regardless of the definition of $y_k$ whether in (5.9) or (5.10), the cost of computing $y_k$ with respect to a larger set of samples is expensive. To address this issue, one approach is to compute the curvature estimate $y_k$ periodically, where the L-SR1 update is performed after a certain number of iterations (say, every $L$ iterations), and remains unchanged within these iterations. Note that the subsampled Hessian-vector product $\bar{B}_k s_k$ in (5.10)

can be coded directly in practice, without explicitly constructing $\bar{B}_k$; see the Hessian-Free technique described in (2.4). In fact, since

$$\frac{\partial \nabla f_{\mathcal{N}_k^B}(w)^T}{\partial w} s_k = \frac{\partial (\nabla f_{\mathcal{N}_k^B}(w)^T s_k)}{\partial w}, \tag{5.11}$$

we have $\bar{B}_k s_k = \frac{\partial (\nabla f_{\mathcal{N}_k^B}(w)^T s_k)}{\partial w}|_{w=w_k}$. Given $\bar{g}_k \triangleq \nabla f_{\mathcal{N}_k^B}(w_k)$, this equation shows how $\bar{B}_k s_k$ can be computed with the cost of only one additional gradient evaluation; in other words, we need (at most) two gradient evaluations with respect to $\mathcal{N}_k^B$ for computing $\bar{B}_k s_k$. We consider a third strategy for computing $y_k$ using a variant of empirical Fisher Information Matrix (eFIM), see e.g. [52], called accumulated eFIM. Given a memory budget of $L_f$, let $g_j$ be defined as $\nabla f_{\mathcal{N}_j}(w_j) \in \mathbb{R}^n$ with $j \in \{k-\mu, k-\mu+1, \ldots, k\}$ and $\mu = \min\{L_f, k\}$. Then, the accumulated eFIM-vector product for curvature computation is computed as follows

$$s_k = p_k, \qquad y_k = \frac{1}{\mu} \sum_{j=k-\mu}^{k} g_j g_j^T s_k. \tag{5.12}$$

Obviously, to implement this approach, it is necessary to allocate storage for a matrix with $L_f$ columns, each of length $n$, in order to store the computed stochastic gradients.

In the following subsection, we assess the performance of the new method in a supervised learning problem.

| | |
|---|---|
| **Gd:** | $N_k = bs$ |
| **Fv:** | $N_k = bs$ |
| **Hv-T0:** | $\mathcal{N}_k = \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **pHv-T0:** | $\mathcal{N}_k = \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **Hv-T1:** | $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **pHv-T1:** | $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **pHv-T2:** | $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = 3N_k$ |
| **pHv-T0 + Fv:** | $\mathcal{N}_k = \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **pHv-T1 + Fv:** | $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = N_k$ |
| **pHv-T2 + Fv:** | $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $N_k = bs$, $N_k^B = 3N_k$ |

| **hyper-parameters** |
|---|
| $bs = 1000$, $L = 5$, $L_f = 100$, $M = 10$, $\tau_0 = 0.25$, $\tau_1 = \frac{\tau_0}{2}$ |

Table 5.1: Experimental configuration of sL-SR1-NTR.

### 5.2.2 Numerical Evaluation

Experimentally, we have found that sL-SR1-NTR when using the choice $f_{m_k}$ defined as (5.7) yields slightly better performance in terms of the final testing accuracy. Consequently, we present the numerical results of Algorithm 7 incorporating this particular choice of $f_{m_k}$ to illustrate its performance compared to its naive variant (sL-SR1-NTR) and the well-known Adam optimizer for training `ResNet-20` on `CIFAR10` images. The naive sL-SR1-TR variant is described as Algorithm 7 where the non-monotone reduction ratio is replaced with the standard one defined in (4.2) and the step length $\alpha_k$ is also obtained by the standard stochastic line-search, i.e., $f_{\mathcal{N}_k}(w_k+\alpha_k p_k) \leq f_{\mathcal{N}_k}(w_k)+c_1\alpha_k g_k^T p_k$. he limited memory parameter for LSR1 Hessian approximation was set to $l = 30$. The `CIFAR10` dataset is normalized by zero-one rescaling and z-score normalization techniques. We set the seed of the MATLAB random number generator to a fixed value, ensuring that the initial parameter remained the same across all experiments. By using figures, we show the performance of the algorithms in terms of the testing accuracy of the classification model versus training time or iteration during the training phase.

We have solved the regularized DL optimization problem (3.5) by Jacobian regularization technique, with $\lambda = 1$, whose benefit can be observed in Figure 5.1. It is important to note that our algorithm uses subsampled loss, including the regularization term, for the required computations during the training phase. However, the quantity displayed as **Loss** in Figure Figure 5.1 does not include the regularization term

**Figure 5.1** The effect of regularization over testing accuracy of sL-SR1-NTR ($bs = 500$).

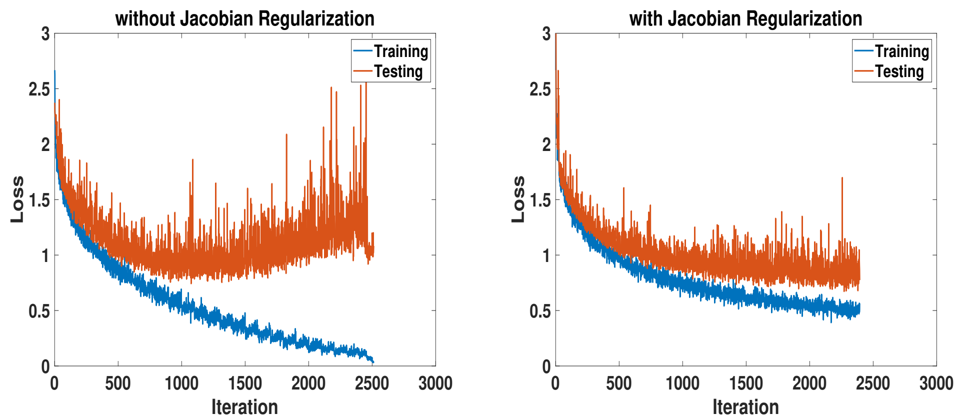


**Figure 5.2** The impact of curvature computing approaches of **pHv+Fv** types.

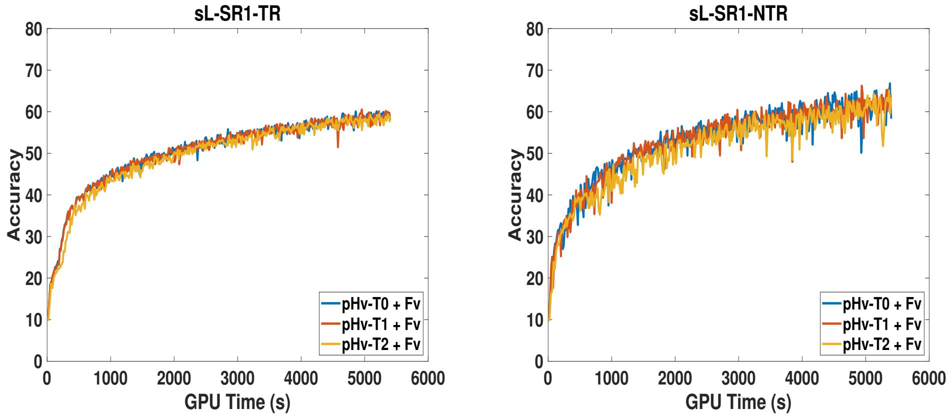



Table 5.1 specifies different approaches utilized for computing curvatures in our experiments conducted with the sLSR1-NTR algorithm. For computing vector $y_k$ in both sL-SR1-TR and sL-SR1-NTR algorithms, we have considered three different strategies, including the gradient difference (5.9), the accumulated eFIM-vector product (5.12), and the subsampled Hessian-vector product (5.10), respectively, denoted as **Gd**, **Fv**, and **Hv** in Table 5.1. In order to see the importance of sampling in the decoupling idea for computing $y_k$, in practice, we have examined the latter approach in different sampling settings to compute the subsampled gradient and subsampled Hessian approximations: same mini-batches (i.e., $\mathcal{N}_k = \mathcal{N}_k^B$), same batch sizes but different samples (i.e. $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $|\mathcal{N}_k| = |\mathcal{N}_k^B|$), or different mini-batches and sizes (i.e. $\mathcal{N}_k \neq \mathcal{N}_k^B$ and $|\mathcal{N}_k| \neq |\mathcal{N}_k^B|$). These cases are respectively denoted as **T0**, **T1** and **T2** in Table 5.1. We have also conducted experiments in the periodic setting to investigate the potential benefits of keeping $y_k$ unchanged between periods (e.g., pHv-T2) versus computing $y_k$ using the accumulated eFIM in the intermediate iterations between periods (e.g., pHv-T2+Fv); the prefix letter **p** stands for **p**eriodical computing $y_k$.

In Figure 5.2 and Figure 5.3, we report comparatively the testing accuracy reached with the different curvature computing strategies when using sL-SR1-TR or sL-SR1-NTR. Figure 5.3 shows that computing $y_k$ using **Hv** of type **T0** leads to more oscillation in testing curve than type **T1** for both algorithms sL-SR1-TR and sL-SR1-NTR. On the other hand, periodically computing $y_k$ produces smoother testing curves, in general. Nevertheless, there are some strong instabilities in the testing accuracy of sL-SR1-NTR

when using **pHv-T0** while they are damped by **pHv-T1**. This result shows that using different samples for computing $y_k$ with respect to the ones used for computing the subsampled gradient is more important in sL-SR1-NTR than in sL-SR1-TR. Moreover, Figure 5.3 shows that using a hybrid approach for computing $y_k$ leads to more stable testing accuracy for both sL-SR1-TR and sL-SR1- NTR even though it produces lower testing accuracy for sL-SR1-NTR.

Figure 5.2 illustrates that the periodical approaches behave similarly. Therefore, **pHv-T1 + Fv** from Figure 5.2 is selected as the better strategy for sL-SR1-NTR. Since **pHv-T1 + Fv** produces more stable testing accuracy than **pHv-T1**, we have considered this approach as the best **Hv** type strategy for computing $y_k$, and compared it with other approaches in Figure 5.5.

Finally, the most important conclusion that can be extracted from Figure 5.3 is that computing the curvature at each iteration by accumulated eFIM-vector product allows for faster training and higher final testing accuracy in all cases, but the improvement is greater when using sL-SR1-NTR.

**Figure 5.3** The impact of different curvature computing approaches.

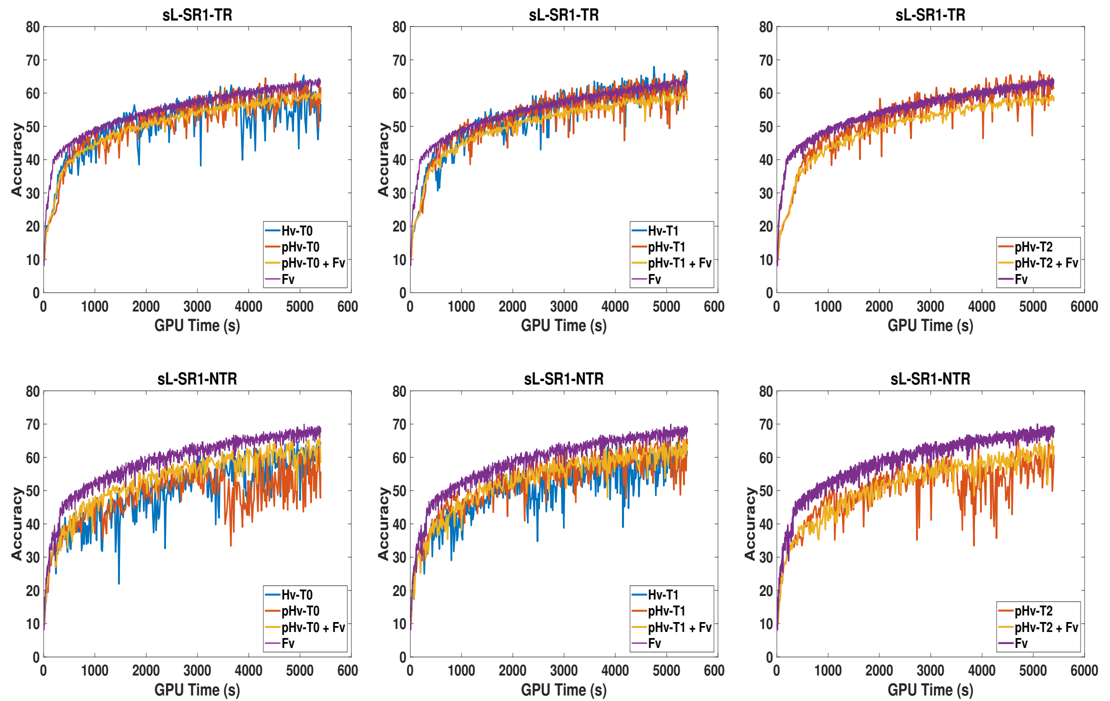


**Figure 5.4** The comparative accuracy of sL-SR1-NTR using **Fv** vs GPU Time (3 hours).

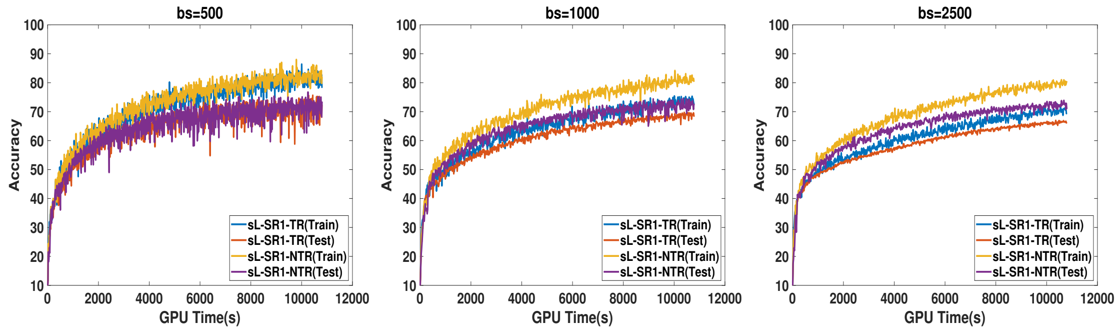



**Figure 5.5** Comparative testing accuracy of sL-SR1-NTR with different curvature approaches.

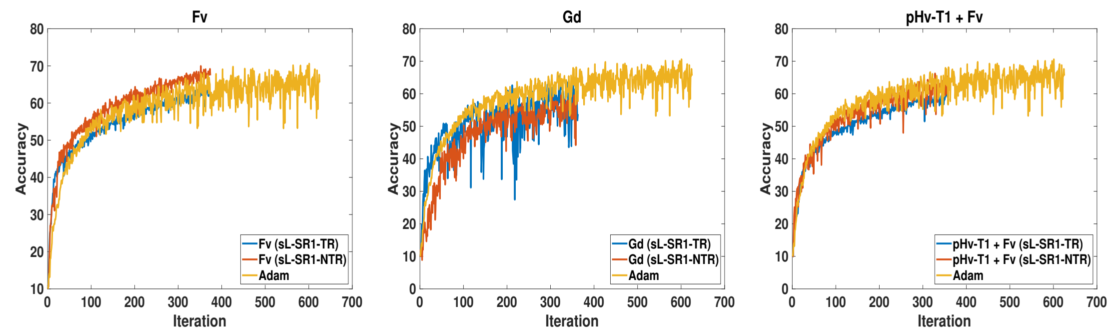
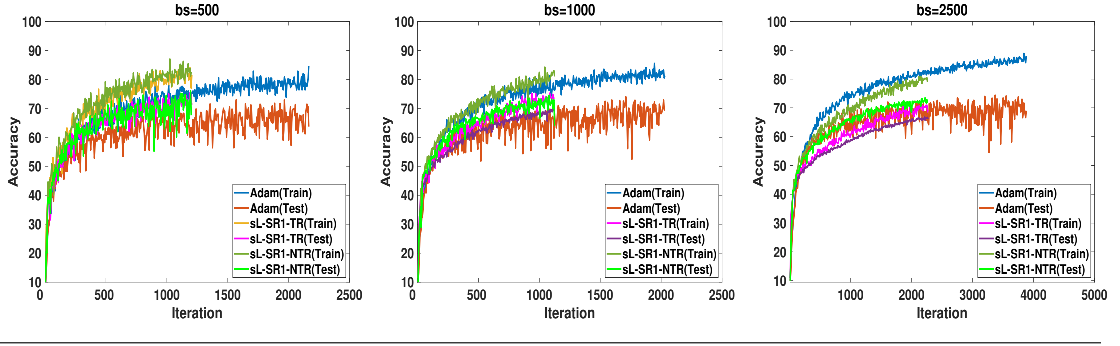



Figure 5.4 shows the results obtained by both algorithms using the curvature computing approach **Fv** and different batch sizes ($bs$). This time, we allowed the longer run for three hours; the reported results include both training and testing accuracy versus time and show that the best approach reveals always sL-SR1-NTR. Moreover, as expected, the experiments with respect to the smallest batch size ($bs = 500$) produce higher accuracy within the fixed running time. Figure 5.5 shows the testing accuracy versus iterations of sL-SR1-TR and sL-SR1-NTR ($bs = 1000$) using three different curvature strategies, i.e., from left to right **Fv**, **Gd** and **pHv-T1** + **Fv**. The results reported in this figure show that the best and worst performances of our proposed algorithm are obtained with **Fv**- and **Gd**-based curvature computing strategies, respectively. We include also the state-of-the-art Adam optimizer [42] for comparison, with learning rate $\alpha_k = 10^{-3}$ experimentally found to be the optimal one. It can be observed that sL-SR1-NTR provides a comparable or better testing accuracy than Adam in the same 90 minutes of running time. In Figure 5.6, we complete the study performed in Figure 5.5.

**Figure 5.6** The comparative accuracy sL-SR1-NTR using **Fv** vs iteration (3 hours).




To summarize, it is not recommended to compute curvature using the Hessian-vector product (**Hv**-types). We infer that the lower accuracy obtained with this approach stems from computing the Hessian-vector products using two inexact (noisy) gradient evaluations, which can amplify the noise. One reason for this conclusion is the use of periodic Hessian-vector products, which could reduce both the computational cost and the impact of the double noisy gradient evaluation. Additionally, the periodic curvature computing strategy works better with the decoupling idea, particularly with type **T1** (**pHv-T1**) rather than **T2**. Furthermore, for intermediate iterations of the algorithms, it is beneficial to employ the **Fv**-based curvature computing strategy in combination with **pHv-T1** (**pHv-T1** + **Fv**), as it leads to smoother accuracies. This approach is recommended for both algorithms, especially for sL-SR1-NTR, as it outperforms sL-SR1-TR. However, among **Fv**, **pHv-T1** + **Fv**, and **Gd**, the best choice for computing $y_k$ is **Fv**. By using this approach, sL-SR1-NTR can be applied instead of other considered methods within a fixed budget of time and a fixed budget of iterations (or epochs, assuming a fixed batch size for all algorithms), particularly when a large batch size is preferred.

## 5.3 A Stochastic Algorithm with Adaptive Sampling

Looking broadly at the idea of non-monotonicity within the stochastic TR framework described in the previous section, our objective is to develop an efficient algorithm with practical features for training DNNs. We propose a second-order non-monotone TR algorithm where the step and the candidate points for the next iteration are based on

the subsampled function and its gradient with respect to adaptive size mini-batches. The involving quadratic TR models are constructed by using Hessian approximations, without imposing a positive definiteness assumption, as the true Hessian in DL problems may not be positive definite due to their non-convex nature. Unlike the classical TR, our decision on acceptance of the trial point is not based only on the decreasing agreement between the model and the approximate objective function, but on an independent subsampled function. This "control" function which is formed through *additional sampling*, similar to one proposed in [23] for the line search framework, also has a role in controlling the sample average approximation error by adaptively choosing the sample size. Depending on the estimated progress of our algorithm, this can yield sample size scenarios ranging from mini-batch to full sample functions. We provide convergence analysis for all possible scenarios and show that the proposed method achieves almost sure convergence under standard assumptions for the TR framework such as Lipschitz-continuous gradients and bounded Hessian approximations.

In the following, we first outline the algorithm with a clear description of its steps, and then give some preliminary working hypotheses to provide its convergence results.

### 5.3.1 Algorithmic framework

Within this subsection, we describe the proposed method called ASNTR (Adaptive Subsample Non-monotone Trust-Region).

At iteration $k$, we construct a quadratic model, as explained in Section 4.1, using the subsampled gradient $g_k \triangleq \nabla f_{\mathcal{N}_k}(w_k)$ and an arbitrary Hessian approximation $B_k$ that satisfies the following condition:

$$\|B_k\| \leq L, \tag{5.13}$$

for some $L > 0$. The common TR subproblem (4.1) is then solved to obtain the relevant direction $p_k$. We assume that at least some fraction of the Cauchy decrease is obtained, i.e., the direction satisfies

$$Q_k(p_k) \leq -\frac{c}{2}\|g_k\| \min\{\delta_k, \frac{\|g_k\|}{\|B_k\|}\}, \tag{5.14}$$

for some $c \in (0,1)$. This is a standard assumption in TR, see Lemma 2.1, and it is not restrictive even in the stochastic framework. Given the search direction $p_k$, we calculate a trial point and then evaluate whether it should be accepted for the next iteration. Our new algorithm uses a modified acceptance strategy as follows. Motivated by the numerical study in Section 5.3, we use a non-monotone TR (NTR) framework instead of the standard TR one. This is because we are dealing with noisy approximations in a stochastic expansion and do not want to impose a strict decrease in the approximate function. To this end, we define the relevant ratio as follows

$$\rho_{\mathcal{N}_k} \triangleq \frac{f_{\mathcal{N}_k}(w_t) - r_{\mathcal{N}_k}}{Q_k(p_k)}, \tag{5.15}$$

where

$$r_{\mathcal{N}_k} \triangleq f_{\mathcal{N}_k}(w_k) + t_k \delta_k, \quad t_k > 0, \tag{5.16}$$

and

$$\sum_{k=0}^{\infty} t_k \leq t < \infty. \tag{5.17}$$

We allow $\mathcal{N}_k$ to be chosen in an arbitrary manner in ASNTR. However, even if we impose uniform sampling that yields an unbiased estimator of the objective function at $w_k$, the dependence between $\mathcal{N}_k$ and $w_t$ yields a biased estimator of $f(w_t)$. In order to overcome this difficulty that deteriorates straightforward convergence analysis, we apply an *additional sampling* strategy [23]. To this end, at every iteration at which $N_k < N$, we choose another independent subsample represented by the index set $\mathcal{D}_k \subset \mathcal{N}$ of size $D_k = |\mathcal{D}_k| < N$ and calculate $f_{\mathcal{D}_k}(w_k), f_{\mathcal{D}_k}(w_t)$ and $\bar{g}_k \triangleq \nabla f_{\mathcal{D}_k}(w_k)$ (see lines 5-6 of the ASNTR algorithm). Since this yields an additional cost per iteration, we suggest a modest value for $D_k$. In fact, in our experiments, we set $D_k = 1$ for all $k$. Furthermore, in the spirit of the TR framework, we also define a linear model as $L_k(v) \triangleq v^T \bar{g}_k$, and consider another agreement as follows

$$\rho_{\mathcal{D}_k} \triangleq \frac{f_{\mathcal{D}_k}(w_t) - r_{\mathcal{D}_k}}{L_k(-\bar{g}_k)}, \tag{5.18}$$

where

$$r_{\mathcal{D}_k} \triangleq f_{\mathcal{D}_k}(w_k) + \delta_k \tilde{t}_k, \quad \tilde{t}_k > 0, \tag{5.19}$$

and

$$\sum_{k=0}^{\infty} \tilde{t}_k \leq \tilde{t} < \infty. \tag{5.20}$$

Therefore, when $N_k < N$, the trial point is accepted only if both $\rho_{\mathcal{N}_k}$ and $\rho_{\mathcal{D}_k}$ are bounded away from zero; otherwise, if the full sample is used, the decision is made by $\rho_{\mathcal{N}_k}$ solely as in deterministic NTR (see lines 23-35 of the ASNTR algorithm). The reasoning behind this is as follows: If the point obtained by observing $f_{\mathcal{N}_k}$ is acceptable also to some independent approximation function $f_{\mathcal{D}_k}$, then we proceed with this point since it is probably acceptable for the original objective function as well because $f_{\mathcal{D}_k}(w_t)$ is an unbiased estimator of $f(w_t)$.

Another role of $\rho_{\mathcal{D}_k}$ is to control the sample size. If the obtained trial point $w_t$ yields an uncontrolled increase in $f_{\mathcal{D}_k}$ as specified in line 10 of ASNTR, we conclude that we need a better approximation of the objective function and we increase the sample size by choosing a new sample for the next iteration. The sample can also be increased if we are too close to a stationary point of the approximate function $f_{\mathcal{N}_k}$. This is stated in line 7 of ASNTR, where $h$ represents an approximation error estimate given by

$$h(N_k) \triangleq \frac{N - N_k}{N}.$$

This gives us more chances for an increase if we have a small sample size. However, the experiments reveal that this is not activated that often and it can be considered as a theoretical safeguard. The algorithm can also keep the same sample size (see lines 14 and 16 of the ASNTR algorithm). Keeping the same sample $\mathcal{N}_k$ in line 14 corresponds to the case where the trial point is acceptable with respect to $f_{\mathcal{D}_k}$, while we do not have the decrease in $f_{\mathcal{N}_k}$. Otherwise, we allow the algorithm in line 16 to choose a new sample of the current size and exploit some new data points in general. The strategy for updating the sample size is described in lines 7-19 of the ASNTR algorithm.

**Algorithm 8** ASNTR

1: Initialization: Set $k = 0$. Choose $\mathcal{N}_0, \subseteq \mathcal{N}$. $\{t_k\}$ satisfying (5.17), $\{\tilde{t}_k\}$ satisfying (5.20), $\delta_0, \delta_{max} \in (0, \infty)$, $\epsilon \in [0, \frac{1}{2})$, $\eta, \nu \in [0, 1/4)$, $0 < \tau_1 \leq 0.5 < \tau_2 < 1 < \tau_3$, $\eta < \eta_2 \leq 3/4$, $\eta_1 \in (\eta, \eta_2)$ .
2: Given $f_{\mathcal{N}_k}(w_k)$, $g_k$ and $B_k$, solve (4.1) for $p_k$ such that (5.14) holds, and then obtain the trial iterate $w_t = w_k + p_k$.
3: Given $f_{\mathcal{N}_k}(w_t)$, compute $\rho_{\mathcal{N}_k}$ using (5.15).
4: **if** $N_k < N$ **then**
5: Choose $\mathcal{D}_k \subset \mathcal{N}$ randomly and uniformly.
6: Given $f_{\mathcal{D}_k}(w_k)$, $\nabla f_{\mathcal{D}_k}(w_k)$, and $f_{\mathcal{D}_k}(w_t)$, compute $\rho_{\mathcal{D}_k}$ using (5.18).
7: **if** $\|g_k\| < \epsilon h(N_k)$ **then**
8: Increase $N_k$ to $N_{k+1}$ and choose $\mathcal{N}_{k+1}$.
9: **else**
10: **if** $\rho_{\mathcal{D}_k} < \nu$ **then**
11: Increase $N_k$ to $N_{k+1}$ and choose $\mathcal{N}_{k+1}$.
12: **else**
13: **if** $\rho_{\mathcal{N}_k} < \eta$ **then**
14: Set $N_{k+1} = N_k$ and $\mathcal{N}_{k+1} = \mathcal{N}_k$.
15: **else**
16: Set $N_{k+1} = N_k$ and choose $\mathcal{N}_{k+1}$.
17: **end if**
18: **end if**
19: **end if**
20: **else**
21: $N_{k+1} = N$
22: **end if**
23: **if** $N_k < N$ **then**
24: **if** $\rho_{\mathcal{N}_k} \geq \eta$ and $\rho_{\mathcal{D}_k} \geq \nu$ **then**
25: $w_{k+1} = w_t$.
26: **else**
27: $w_{k+1} = w_k$.
28: **end if**
29: **else**
30: **if** $\rho_{\mathcal{N}_k} \geq \eta$ **then**
31: $w_{k+1} = w_t$.
32: **else**
33: $w_{k+1} = w_k$.
34: **end if**
35: **end if**
36: **if** $\rho_{\mathcal{N}_k} < \eta_1$, **then**
37: $\delta_{k+1} = \tau_1 \delta_k$
38: **else if** $\rho_{\mathcal{N}_k} > \eta_2$ and $\|p_k\| \geq \tau_2 \delta_k$, **then**
39: $\delta_{k+1} = min\{\tau_3 \delta_k, \delta_{max}\}$,
40: **else**
41: $\delta_{k+1} = \delta_k$.
42: **end if**
43: **if** Some stopping conditions hold **then**
44: Stop training
45: **else**
46: Set $k = k + 1$ and go to step 2.
47: **end if**

Notice that the sample size can not be decreased, and once all samples are included, it remains unchanged until the end of the process. Moreover, it should be noted that ASNTR provides complete freedom in terms of the increment in the sample size as well as the choice of a sample $\mathcal{N}_k$. This leaves enough space for different sampling strategies within the Algorithm. Mostly depending on the problem, ASNTR can result in a mini-batch strategy, but it can also yield an increasing sample size procedure.

The radius $\delta_k$ is updated within lines 36-42 of ASNTR. It is a common update strategy for TR approaches, and also in our algorithm. It is completely based on $f_{\mathcal{N}_k}$ since it is related to the error of the quadratic model, and not to the approximation error which we control by additional sampling. We also require some predetermined fixed hyper-parameters for ASNTR, see Table 5.2. We set them based on the criteria set forth in e.g., [62, 27, 81], and as a result of the necessity of our convergence findings.

### 5.3.2 Convergence Analysis

We make the following standard assumption for the TR framework needed to prove the a.s. convergence result of ASNTR.

**Assumption 5.1.** *The functions $f_i, i = 1, ..., N$ are bounded from below and twice continuously-differentiable with L-Lipschitz-continuous gradients.*

First, we prove that the sequence of function values generated by ASNTR is uniformly bounded.

**Lemma 5.1.** *Suppose that Assumption 5.1 holds. Then the sequence $\{w_k\}$ generated by ASNTR algorithm satisfies*

$$f(w_k) \leq f(w_0) + \delta(t + \tilde{t}), \quad k = 0, 1, \dots$$

*where $t$ and $\tilde{t}$ correspond to those in (5.17) and (5.20), respectively, and $\delta \triangleq \delta_{max}$.*

**Proof.** *First scenario, when $N_k < N$ for each $k$.* In this scenario, we should notice that $\rho_{\mathcal{D}_k}$ is calculated at every iteration and influences the decision to accept or reject the trial point $w_t$. Let us consider an arbitrary iteration $k$ of the proposed algorithm under

this scenario. Notice that $\rho_{\mathcal{D}_k} \geq \nu$ is equivalent to

$$f_{\mathcal{D}_k}(w_{k+1}) \leq f_{\mathcal{D}_k}(w_k) - \nu\|\bar{g}_k\|^2 + \delta_k\tilde{t}_k.$$

Let us denote by $\mathcal{D}_k^+$ the subset of all possible outcomes of $\mathcal{D}_k$ at iteration $k$ that satisfy $\rho_{\mathcal{D}_k} \geq \nu$, i.e.,

$$\mathcal{D}_k^+ = \{\mathcal{D}_k \subset \mathcal{N} \mid f_{\mathcal{D}_k}(w_t) \leq f_{\mathcal{D}_k}(w_k) - \nu\|\nabla f_{\mathcal{D}_k}(w_k)\|^2 + \delta_k\tilde{t}_k\}. \tag{5.21}$$

Notice that if $\mathcal{D}_k \in \mathcal{D}_k^+$ and $\rho_{\mathcal{N}_k} \geq \eta$ then $w_{k+1} = w_t$ and there holds

$$f_{\mathcal{D}_k}(w_{k+1}) \leq f_{\mathcal{D}_k}(w_k) + \delta\tilde{t}_k.$$

On the other hand, if $\mathcal{D}_k \in \mathcal{D}_k^+$ and $\rho_{\mathcal{N}_k} < \eta$ or if $\mathcal{D}_k \in \mathcal{D}_k^-$, where

$$\mathcal{D}_k^- = \{\mathcal{D}_k \subset \mathcal{N} \mid f_{\mathcal{D}_k}(w_t) > f_{\mathcal{D}_k}(w_k) - \nu\|\nabla f_{\mathcal{D}_k}(w_k)\|^2 + \delta_k\tilde{t}_k\}, \tag{5.22}$$

then $w_{k+1} = w_k$ and there holds

$$f_{\mathcal{D}_k}(w_{k+1}) = f_{\mathcal{D}_k}(w_k) \leq f_{\mathcal{D}_k}(w_k) + \delta\tilde{t}_k.$$

Thus, we conclude that for all possible outcomes of $\mathcal{D}_k$, there holds

$$f_{\mathcal{D}_k}(w_{k+1}) = f_{\mathcal{D}_k}(w_k) \leq f_{\mathcal{D}_k}(w_k) + \delta\tilde{t}_k. \tag{5.23}$$

Now, let us denote by $\mathcal{F}_{k+1/2}$ the $\sigma$-algebra generated by $\mathcal{N}_0$, $\mathcal{D}_0$, ..., $\mathcal{N}_{k-1}$, $\mathcal{D}_{k-1}$, $\mathcal{N}_k$. Since $\mathcal{D}_k$ is chosen randomly and uniformly, we have

$$\mathbb{E}(f_{\mathcal{D}_k}(w_{k+1})|\mathcal{F}_{k+1/2}) = f(w_{k+1}).$$

By this fact, applying conditional expectation with respect to $\mathcal{F}_{k+1/2}$ on (5.23) leads to

$$f(w_{k+1}) \leq f(w_k) + \delta\tilde{t}_k \tag{5.24}$$

and we obtain that for all $k$ there holds

$$f(w_k) \leq f(w_0) + \delta \sum_{j=0}^{k-1} \tilde{t}_k \leq f(w_0) + \delta \tilde{t}. \tag{5.25}$$

*Second scenario, when $N$ is achieved at some finite iteration.* In this case, there exists a finite iteration $\tilde{k}$ such that $N_k = N$ for all $k > \tilde{k}$. Thus, the trial point for all $k > \tilde{k}$ is accepted if and only if $\rho_{\mathcal{N}_k} \geq \eta$. This implies that we either have

$$f(w_{k+1}) = f(w_k),$$

or

$$f(w_{k+1}) \leq f(w_k) + \delta_k t_k - \frac{c}{2}\|g_k\| \min\{\delta_k, \frac{\|g_k\|}{\|B_k\|}\} \leq f(w_k) + \delta t_k, \tag{5.26}$$

where $\|g_k\| = \|\nabla f(w_k)\|$ in this scenario. In any of the two cases, for all $s \in \mathbb{N}$, we can write

$$f(w_{\tilde{k}+s}) \leq f(w_{\tilde{k}}) + \delta \sum_{j=0}^{s-1} t_{\tilde{k}+j} \leq f(w_{\tilde{k}}) + \delta t. \tag{5.27}$$

Since we have already proved that the upper bound (5.25) holds for all $k$ when $N_k < N$, we can conclude that $f(w_{\tilde{k}}) \leq f(w_0) + \delta \tilde{t}$. Therefore

$$f(w_{\tilde{k}+s}) \leq f(w_0) + \delta(\tilde{t} + t).$$

Combining all together, we come to the conclusion that the objective function is uniformly upper-bounded in each scenario, which completes the proof. □

Next, we prove an important lemma that will help us prove the main convergence result. We prove a.s. convergence of ASNTR by analyzing both of the two possible scenarios with respect to the sample size sequence: 1) "mini-batch scenario" - where the subsampling is employed in each iteration; 2) "deterministic scenario" - where the full sample is reached eventually. In Lemma 5.2, we show that in the first scenario $\rho_{\mathcal{D}_k} \geq \nu$ holds for all possible realizations of $\mathcal{D}_k$ and all $k$ sufficiently large - otherwise we reach the full sample and fall into the "deterministic scenario".

**Lemma 5.2.** *Suppose that Assumption 5.1 holds. If $N_k < N$ for all $k \in \mathbb{N}$, then there must exist $k_1 \in \mathbb{N}$ such that $\rho_{\mathcal{D}_k} \geq \nu$ for all $k \geq k_1$ and for all possible realizations $\mathcal{D}_k$.*

**Proof.** Assume that there is no $k_1 \in \mathbb{N}$ such that $\rho_{\mathcal{D}_k} \geq \nu$ for all $k \geq k_1$. We will show that in that case the full sample is reached eventually. Assume the contrary, i.e., assume that there exists some $\bar{N} < N$ and $k_2 \in \mathbb{N}$ such that $N_k = \bar{N}$ for all $k \geq k_2$. Under the current assumptions, we know that there exists an infinite subsequence of iterations $K \subseteq \mathbb{N}$ such that $\rho_{\mathcal{D}_k} \geq \nu$ is not satisfied for all possible outcomes of $\mathcal{D}_k$. To be more precise, let us use the same notation as in the previous lemma regarding $\mathcal{D}_k^+$ and $\mathcal{D}_k^-$ where $|\mathcal{D}_k| = 1$ for simplicity. Then, we have that $\mathcal{D}_k^- \neq \emptyset$ for all $k \in K$. Since $\mathcal{D}_k$ is chosen randomly and uniformly, we have that $P(\mathcal{D}_k = i) = 1/N, i = 1, ..., N$ in any iteration $k$. Therefore, $P(\mathcal{D}_k \in \mathcal{D}_k^-) \geq 1/N$, i.e., $P(\mathcal{D}_k \in \mathcal{D}_k^+) \leq 1 - 1/N$ for all $k \in K$. So,

$$P(\mathcal{D}_k \in \mathcal{D}_k^+, k \in K) = \prod_{k \in K} \frac{N-1}{N} = 0,$$

which means that we will almost surely encounter an iteration at which the sample size will be increased due to $\rho_{\mathcal{D}_k}$, which is in contradiction with $N_k = \bar{N}$ for all $k$ large enough. This completes the proof. □

**Theorem 5.1.** *Suppose that Assumption 5.1 holds. Then the sequence $\{w_k\}$ generated by ASNTR algorithm satisfies*

$$\liminf_{k \to \infty} \|\nabla f(w_k)\| = 0 \quad a.s.$$

**Proof.** Assume two different scenarios with $N_k = N$ for all $k$ large enough, and $N_k < N$ for all $k$. Let us start with the first one in which $N_k = N$ for all $k \geq \tilde{k}$, where $\tilde{k}$ is random but finite. In this case, ASNTR reduces to the non-monotone *deterministic* trust-region algorithm applied on $f$. By $L$-Lipschitz continuity of $\nabla f$, we obtain

$$|f(w_k + p_k) - f(w_k) - \nabla^T f(w_k) p_k| \leq \frac{L}{2} \|p_k\|^2. \tag{5.28}$$

Now, let us denote $\rho_k \triangleq \rho_{\mathcal{N}_k}$ and assume that $\|\nabla f(w_k)\| \geq \varepsilon > 0$ for all $k \geq \tilde{k}$. Then,

we obtain

$$
\begin{aligned}
|\rho_k - 1| &= \frac{|f(w_k+p_k) - f(w_k) - \delta_k t_k - \nabla^T f(w_k)p_k - 0.5 p_k^T B_k p_k|}{|Q_k(p_k)|} \qquad (5.29)\\
&\leq \frac{0.5L\|p_k\|^2 + \delta_k t_k + 0.5L\|p_k\|^2}{0.5c\|g_k\| \min\{\delta_k, \frac{\|g_k\|}{\|B_k\|}\}}\\
&\leq \frac{L\|p_k\|^2 + \delta_k t_k}{0.5c\varepsilon \min\{\delta_k, \frac{\varepsilon}{L}\}},
\end{aligned}
$$

where $\|g_k\| = \|\nabla f(w_k)\|$ in this scenario. Let define $\tilde{\delta} = \frac{\varepsilon c}{20L}$. Without loss of generality, we can assume that $t_k \leq L\tilde{\delta}$ for all $k \geq \tilde{k}$. Let us now observe iterations after $\tilde{k}$. If $\delta_k \leq \tilde{\delta}$, then $\delta_{k+1} \geq \delta_k$. This comes from the following fact due to $\tilde{\delta} \leq \frac{\varepsilon}{L}$,

$$
|\rho_k - 1| \leq \frac{L\delta_k^2 + \delta_k t_k}{0.5c\varepsilon\delta_k} \leq \frac{2L\tilde{\delta}}{0.5c\varepsilon} < \frac{1}{4}, \qquad (5.30)
$$

i.e., $\rho_k > \frac{3}{4}$ which implies that the NTR radius is not decreased; see lines 36-42 in ASNTR. Further, there exists $\hat{\delta} > 0$ such that $\delta_k \geq \hat{\delta}$ for all $k \geq \tilde{k}$. Moreover, for all $k \geq \tilde{k}$, the assumption $\rho_k < \eta$ would yield a contradiction since it would imply $\lim_{k\to\infty} \delta_k = 0$. Therefore, there must exist an infinite set $K \subseteq \mathbb{N}$ as $K = \{k \geq \tilde{k} \,|\, \rho_k \geq \eta\}$. For all $k \in K$, we have

$$
f(w_{k+1}) \leq f(w_k) + \delta_k t_k - \frac{c}{8}\|g_k\| \min\{\delta_k, \frac{\|g_k\|}{\|B_k\|}\} \leq f(w_k) + \delta t_k - \frac{c}{8}\varepsilon \min\{\hat{\delta}, \frac{\varepsilon}{L}\},
$$

where $\delta \triangleq \delta_{max}$ in ASNTR. Now, let $\bar{c} \triangleq \frac{c}{8}\varepsilon \min\{\hat{\delta}, \frac{\varepsilon}{L}\}$. Since $t_k$ tends to zero, we have $\delta t_k \leq \frac{\bar{c}}{2}$ for all $k$ large enough. Without loss of generality, we can say that this holds for all $k \in K$; Denoting $K = \{k_j\}_{j\in\mathbb{N}}$, we have

$$
f(w_{k_j+1}) \leq f(w_{k_j}) - \frac{\bar{c}}{2}.
$$

Since $w_{k+1} = w_k$ for all $k \geq \tilde{k}$ and $k \notin K$, i.e. when $\rho_k < \eta$, we obtain

$$
f(w_{k_{j+1}}) \leq f(w_{k_j}) - \frac{\bar{c}}{2}.
$$

Thus, due to Lemma 5.1, we obtain for all $j \in \mathbb{N}$

$$f(w_{k_j}) \leq f(w_{k_0}) - j\frac{\bar{c}}{2} \leq f(w_0) + \delta(t + \tilde{t}) - j\frac{\bar{c}}{2}. \tag{5.31}$$

Letting $j$ tend to infinity in (5.31), we obtain $\lim_{j\to\infty} f(w_{k_j}) = -\infty$, which is in contradiction with the assumption of $f$ being bounded from below. Therefore, $\|\nabla f(w_k)\| \geq \varepsilon > 0$ for all $k \geq \tilde{k}$ can not be true, so we conclude that

$$\liminf_{k\to\infty} \|\nabla f(w_k)\| = 0.$$

Now let us assume the second scenario, i.e., $N_k < N$ for all $k$, i.e., the sample size is increased only finitely many times. According to Lemma 5.2 and the lines 7-8 of the algorithm, the currently considered scenario implies the existence of a finite $\tilde{k}_1$ such that

$$N_k = \tilde{N}, \quad \|g_k\| \geq \epsilon h(N_k) \triangleq \epsilon_{\tilde{N}} > 0 \quad \text{and} \quad \rho_{\mathcal{D}_k} \geq \nu, \tag{5.32}$$

for all $k \geq \tilde{k}_1$ and some $\tilde{N} < N$. Now, let us prove that there exists an infinite subset of iterations $\tilde{K} \subseteq \mathbb{N}$ such that $\rho_k \geq \eta$ for all $k \in K$, i.e., the trial point $w_t$ is accepted infinitely many times. Assume a contrary, i.e., there exists some finite $\tilde{k}_2$ such that $\rho_k < \eta$ for all $k \geq \tilde{k}_2$. Since $\eta < \eta_1$, this further implies that $\lim_{k\to\infty} \delta_k = 0$; see lines 36 and 37 in Algorithm 8. Moreover, line 13 of Algorithm 8 implies that the sample does not change, meaning that there exists $\tilde{\mathcal{N}} \subset \mathcal{N}$ such that $\mathcal{N}_k = \tilde{\mathcal{N}}$ for all $k \geq \tilde{k}_3 \triangleq \max\{\tilde{k}_1, \tilde{k}_2\}$. By $L$-Lipschitz continuity of $\nabla f_{\tilde{\mathcal{N}}}$, we obtain

$$|f_{\tilde{\mathcal{N}}}(w_k + p_k) - f_{\tilde{\mathcal{N}}}(w_k) - \nabla^T f_{\tilde{\mathcal{N}}}(w_k) p_k| \leq \frac{L}{2}\|p_k\|^2. \tag{5.33}$$

For every $k \geq \tilde{k}_3$, then we have

$$\begin{aligned} |\rho_k - 1| &= \frac{|f_{\tilde{\mathcal{N}}}(w_k + p_k) - f_{\tilde{\mathcal{N}}}(w_k) - \delta_k t_k - \nabla^T f_{\tilde{\mathcal{N}}}(w_k) p_k - 0.5 p_k^T B_k p_k|}{|Q_k(p_k)|} \\ &\leq \frac{0.5L\|p_k\|^2 + \delta_k t_k + 0.5L\|p_k\|^2}{0.5c\|g_k\| \min\{\delta_k, \frac{\|g_k\|}{\|B_k\|}\}} \\ &\leq \frac{L\delta_k^2 + \delta_k t_k}{0.5c\epsilon_{\tilde{N}} \min\{\delta_k, \frac{\varepsilon}{L}\}}. \end{aligned} \tag{5.34}$$

Since $\lim_{k\to\infty} \delta_k = 0$, there exists $\tilde{k}_4$ such that for all $k \geq \tilde{k}_4$ we obtain

$$|\rho_k - 1| \leq \frac{L\delta_k^2 + \delta_k t_k}{0.5c\epsilon_{\tilde{N}}\delta_k} = \frac{L\delta_k + t_k}{0.5c\epsilon_{\tilde{N}}}.$$

Due to the fact that $t_k$ tends to zero, we obtain that $\lim_{k\to\infty} \rho_k = 1$ which is in contradiction with $\rho_k < \eta < 1/4$. Thus, we conclude that there must exist $\tilde{K} \subseteq \mathbb{N}$ such that $\rho_k \geq \eta$ for all $k \in \tilde{K}$. Let us define $K_1 \triangleq \tilde{K} \cap \{\tilde{k}_1, \tilde{k}_1 + 1, \ldots\}$. Notice that we have $\rho_{\mathcal{D}_k} \geq \nu$ and $\rho_k \geq \eta$ for all $k \in K_1$. Thus, the following holds for all $k \in K_1$

$$\frac{f_{\mathcal{D}_k}(w_k + p_k) - r_{\mathcal{D}_k}}{L_k(-\bar{g}_k)} \geq \nu,$$

where $\bar{g}_k = \nabla f_{\mathcal{D}_k}(w_k)$. This further implies

$$f_{\mathcal{D}_k}(w_k + p_k) - r_{\mathcal{D}_k} \leq -\nu\|\bar{g}_k\|^2.$$

Moreover, the following holds for all $k \in K_1$

$$f_{\mathcal{D}_k}(w_{k+1}) = f_{\mathcal{D}_k}(w_k + p_k) \leq f_{\mathcal{D}_k}(w_k) - \nu\|\bar{g}_k\|^2 + \delta\tilde{t}_k. \tag{5.35}$$

As in Lemma 5.1, let us define by $\mathcal{F}_{k+1/2}$ the $\sigma$-algebra generated by $\mathcal{N}_0$, $\mathcal{D}_0$, ... , $\mathcal{N}_{k-1}$, $\mathcal{D}_{k-1}$, $\mathcal{N}_k$. Using the fact that $\mathcal{D}_k$ is chosen randomly and uniformly, applying the conditional expectation with respect to $\mathcal{F}_{k+1/2}$ on (5.35) we obtain for all $k \in K_1$

$$f(w_{k+1}) \leq f(w_k) - \nu\mathbb{E}(\|\bar{g}_k\|^2|\mathcal{F}_{k+1/2}) + \delta\tilde{t}_k. \tag{5.36}$$

Moreover, we also have

$$\mathbb{E}(\nabla f_{\mathcal{D}_k}(w_k)|\mathcal{F}_{k+1/2}) = \nabla f(w_k),$$

and

$$\begin{aligned}\|\nabla f(w_k)\|^2 = \|\mathbb{E}(\nabla f_{\mathcal{D}_k}(w_k)|\mathcal{F}_{k+1/2})\|^2 &\leq \mathbb{E}^2(\|\nabla f_{\mathcal{D}_k}(w_k)\||\mathcal{F}_{k+1/2}) \\ &\leq \mathbb{E}(\|\nabla f_{\mathcal{D}_k}(w_k)\|^2|\mathcal{F}_{k+1/2}).\end{aligned}$$

For all $k \in K_1$ in (5.36), we have

$$f(w_{k+1}) \leq f(w_k) - \nu \|\nabla f(w_k)\|^2 + \delta \tilde{t}_k. \tag{5.37}$$

Notice that $w_{k+1} = w_k$ in all other iterations $k \geq \tilde{k}_1$ and $k \notin K_1$. Denoting $T = \{k_j\}_{j \in \mathbb{N}}$, for all $j \in \mathbb{N}$, we obtain

$$f(w_{k_{j+1}}) = f(w_{k_j+1}) \leq f(w_{k_j}) - \nu \|\nabla f(w_{k_j})\|^2 + \delta \tilde{t}_{k_j}.$$

Then, due to the summability condition of $\tilde{t}_k$ and Lemma 5.1, the following holds for any $s \in \mathbb{N}$

$$f(w_{k_{s+1}}) \leq f(w_{k_0}) - \nu \sum_{j=0}^{s} \|\nabla f(w_{k_j})\|^2 + \delta \tilde{t} \leq f(w_0) + \delta(2t + \tilde{t}) - \nu \sum_{j=0}^{s} \|\nabla f(w_{k_j})\|^2 + \delta \tilde{t}.$$

Now, applying the expectation, using the boundedness of $f$ from below, and letting $s$ tend to infinity in the above inequality, we end up with

$$\sum_{j=0}^{\infty} \mathbb{E}(\|\nabla f(w_{k_j})\|^2) < \infty.$$

Moreover, the following holds by Markov's inequality for any $\epsilon > 0$

$$P(\|\nabla f(w_{k_j})\| \geq \epsilon) \leq \frac{\mathbb{E}(\|\nabla f(w_{k_j})\|^2)}{\epsilon^2}.$$

Therefore, we have

$$\sum_{j=0}^{\infty} P(\|\nabla f(w_{k_j})\| \geq \epsilon) < \infty.$$

Finally, Borel-Cantelli Lemma implies that almost surely $\lim_{j \to \infty} \|\nabla f(w_{k_j})\| = 0$. Combining all together, the result follows as in both scenarios we have at least

$$\liminf_{k \to \infty} \|\nabla f(w_{k_j})\| = 0 \quad \text{a.s.}$$

□

| **STORM** |
| --- |
| $\delta_0 = 1,\ \delta_{max} = 10,\ l = 30,\ \eta_1 = 10^{-4},\ \eta_2 = 10^{-3},\ \gamma = 2$ |
| **ASNTR** |
| $\delta_0 = 1,\ \delta_{max} = 10,\ l = 30,\ \eta = \nu = 10^{-4},\ \eta_1 = 0.1,\ \eta_2 = 0.75,\ \tau_1 = 0.5,\ \tau_2 = 0.8, \tau_3 = 2$ |

Table 5.2: Hyper-parameters of STORM and ASNTR.

### 5.3.3 Numerical Evaluation

We present some experimental results to make a comparison between ASNTR and STORM as Algorithm 5 in [17] for training DNNs in two problem types: regression and classification. Both datasets are normalized by zero-one rescaling and z-score normalization techniques. Although the TR quadratic models of ASNTR and STORM allow any Hessian approximations, we have customized both algorithms with an L-SR1 update and solved their associated TR subproblem by OBS solver, see Algorithm C.5. The hyper-parameters applied in both algorithms are described in Table 5.2. We have randomly (without replacement) chosen the index subset $\mathcal{N}_k \subseteq \{1, 2, \dots, N\}$ to generate a mini-batch of size $N_k$ for computing required quantities, i.e., subsampled functions and gradients. Note that in STORM, function estimates of the numerator of the TR ratio may not necessarily be obtained using this mini-batch. However, based on findings in [75], it is recommended to use the same mini-batch for acquiring all function estimations. Given $N_0 = d + 1$ where $d$ is the dimension of a single input sample $x_i \in \mathbb{R}^d$, we have adopted the linearly increased sampling rule that $N_k = \min(N, \max(100k + N_0, \lceil \frac{1}{{\delta_k}^2} \rceil)$ for STORM as in [17] while we have exploited the simple following sampling rule

$$N_{k+1} = \lceil 1.01 N_k \rceil, \tag{5.38}$$

for ASNTR only when it needed, otherwise $N_{k+1} = N_k$. Using the non-monotone TR framework in our algorithm, we set $t_k = \frac{C_1}{(k+1)^{1.1}}$ and $\tilde{t}_k = \frac{C_2}{(k+1)^{1.1}}$ for some $C_1, C_2 > 0$ in (5.16) and (5.19), respectively. We have selected $\mathcal{D}_k$ with cardinality 1, i.e., $|\mathcal{D}_k| = 1$

In our implementations, each algorithm was run with 5 different initial random seeds. The criteria of both algorithms' performance (loss and accuracy) are compared against the number of gradient calls ($N_g$) during the training phase. Both algorithms were terminated when $N_g$ reached the determined budget of the gradient evaluations ($N_g^{\max}$).

Due to the use of subsampling and forming mini-batches of training data, the training accuracy is measured as the percentage of correct predictions within each mini-batch, and the training loss is computed by evaluating the loss functions on the mini-batches. Training and testing accuracy are correspondingly reported; the accuracy of image classification is the number of correct predictions in percentage while the accuracy of image regression is the number of predictions in percentage within an acceptable error margin (threshold) that we have set the threshold to be 10 degrees. What follows is the evolution of the accuracy produced by the algorithms for training in considered problems.

#### Classification

Figures 5.7, 5.8, 5.10 and 5.11 shoe the accuracy and loss variations (mean and standard error) of the ASNTR algorithm in comparison with STORM. The experiments were conducted with different values of $C_2$ ($C_2 = 1, 10^2, 10^8$) in $\rho_{\mathcal{D}_k}$ and $C_1 = 1$ in $\rho_{\mathcal{N}_k}$, under fixed budgets of gradient evaluations ($N_g^{\max} = 6 \times 10^5$ for MNIST and $N_g^{\max} = 9 \times 10^6$ for CIFAR10).

The results demonstrate that ASNTR achieves higher training and testing accuracy compared to STORM, except in Figure 5.8 where ASNTR and STORM show comparable performance for $C_2 = 1$. Additionally, ASNTR achieves higher accuracy (lower loss) with fewer gradient evaluations ($N_g$) compared to STORM. The figures also indicate the robustness of ASNTR for larger values of $C_2$, indicating its ability with higher rates of non-monotonicity.

#### Regression

Figures 5.9 and 5.12 show also the accuracy and loss variations on both training and test `DIGITS` dataset within a fixed budget of gradient evaluations $N_g^{\max} = 2 \times 10^6$. These figures demonstrate the resilience of ASNTR, particularly for the highest value of $C_2$. Despite facing challenges in the early stages of the training phase with $C_2 = 1$ and $C_2 = 10^2$, ASNTR is able to overcome them and achieve accuracy levels comparable to those of the STORM algorithm. This indicates the robustness and effectiveness of ASNTR in handling different levels of non-monotonicity.

**Figure 5.7** The accuracy variations of STORM and ASNTR on MNIST.

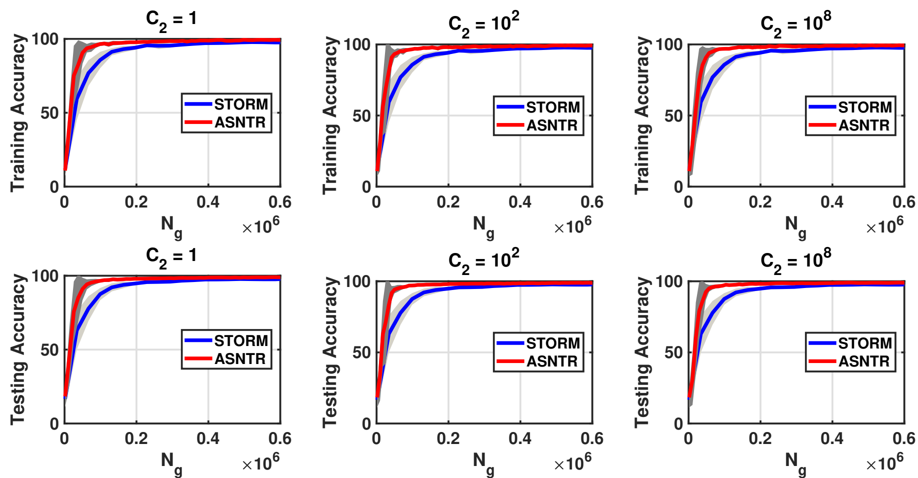



**Figure 5.8** The accuracy variations of STORM and ASNTR on Cifar10.

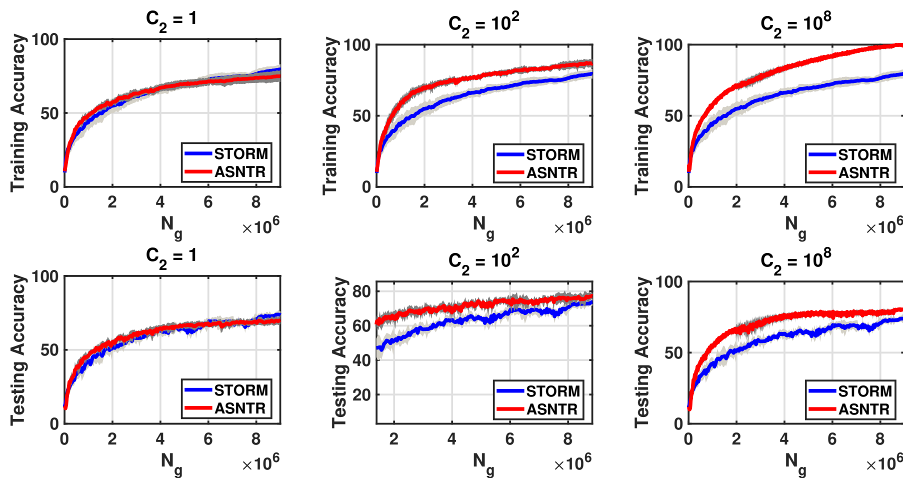



**Figure 5.9** The accuracy variations of STORM and ASNTR on DIGITS.

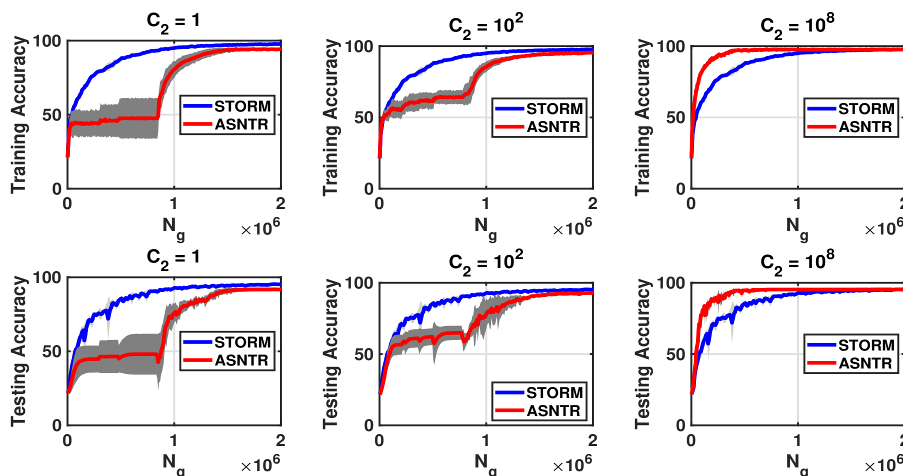


**Figure 5.10** The loss variation of STORM and ASNTR on `MNIST`.

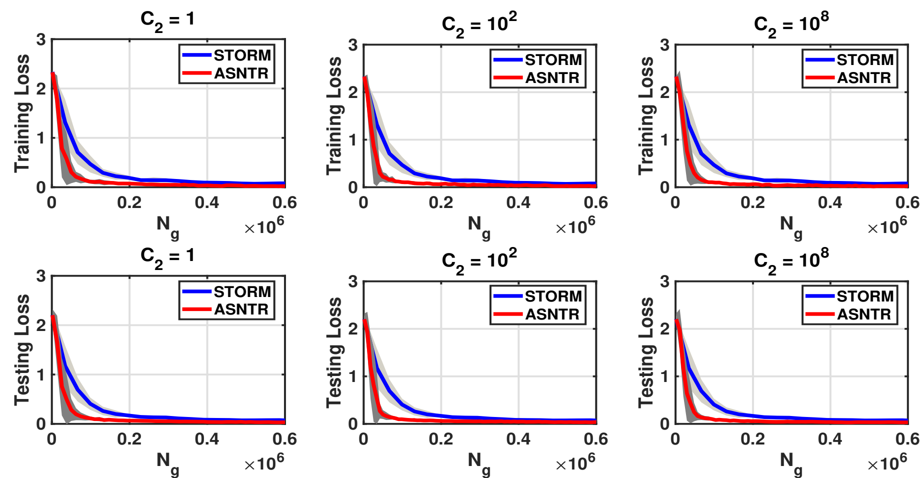



**Figure 5.11** The loss variation of STORM and ASNTR on `CIFAR10`.

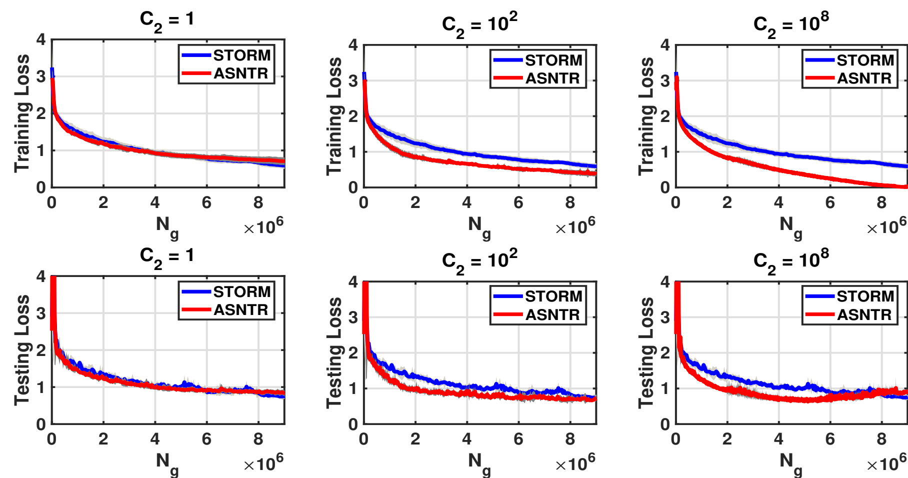
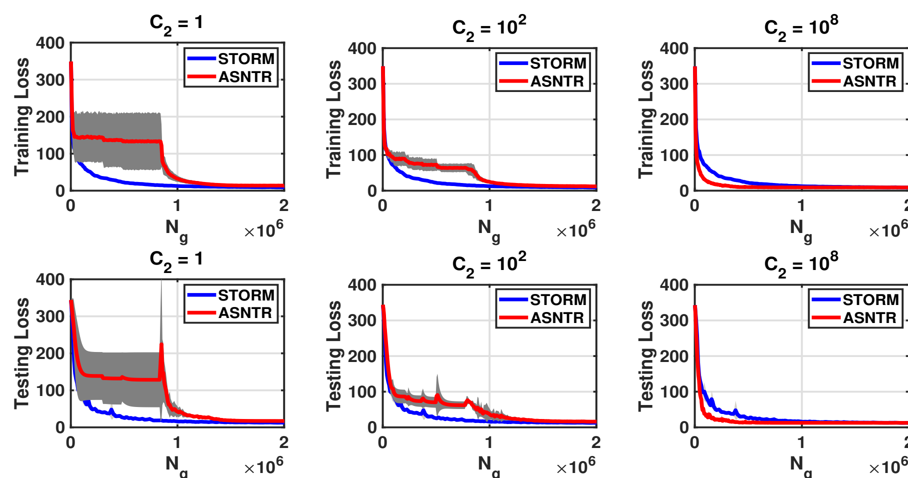



**Figure 5.12** The loss variation of STORM and ASNTR on `DIGITS`.



In summary, the experiments show the effectiveness of ASNTR in handling different levels of non-monotonicity, in particular when $C_2$ is large; in these cases, the testing accuracy of ASNTR is higher than that of STORM while requiring fewer numbers of gradient evaluations.

### Additional results

In Figure 5.13 and Figure 5.14, we present further details regarding our proposed algorithm, ASNTR, where $C_2 = 1,\ 10^2,\ 10^8$ in $\rho_{\mathcal{D}_k}$ and $C_1 = 1$ in $\rho_{\mathcal{N}_k}$. These results aim to give more insights with respect to the sampling behavior of ASNTR as follows.

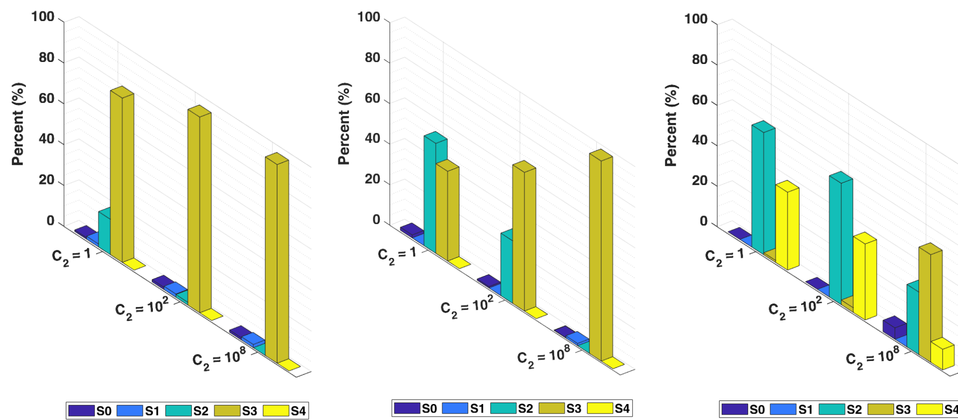
Let S1 and S2 indicate the iterations of ASNTR at which steps 7 and 10, respectively, using the increasing sampling rule (5.38) are executed. For the remaining option, i.e., $N_{k+1} = N_k$, let S3 and S0 show the iterations at which new samples through step 15 and current samples through step 13 are performed. We also define variable S4 representing the iteration of ASNTR at which whole samples ($N_k = N$) are needed.

Figure 5.13a shows the (average) contributions of the aforementioned sampling types in ASNTR running by five different initial random generators for `MNIST`, `CIFAR10`, and `DIGITS` with respectively predetermined $N_g^{\max} = 0.6 \times 10^6$, $9 \times 10^6$ and $2 \times 10^6$; however, considering only a specific initial seed, Figure 5.13b, Figure 5.13c and Figure 5.13d indicate when/where each of these sampling types is utilized in ASNTR.

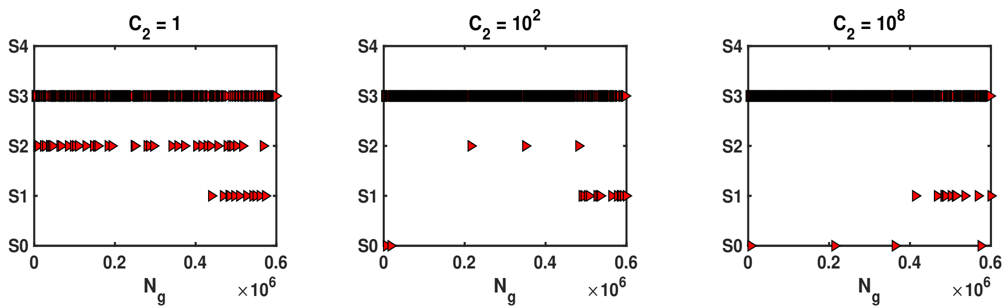
Obviously, the contribution rate of $S3$ is influenced by $S2$ where ASNTR has to increase the batch size if $\rho_{\mathcal{D}_k} < \nu$. The value of $C_2$ in $\rho_{\mathcal{D}_k}$ plays a significant role in determining the portion of $S3$ and $S2$. In fact, the larger $C_2$ results in the higher portion of $S3$ and the lower portion of $S2$. When $C_2$ is large, there is less need to increase the batch size unless the current iterate approaches a stationary point of the current approximate function, which leads to increasing the portion of $S1$. The increase portion of $S1$ usually happens at the end of the training stage.

Furthermore, we have observed that the choice of $C_2$ in $\tilde{t}_k$ significantly impacts the robustness of our proposed algorithm, as evident from the results presented in Figures 5.7–5.12; in other words, the higher the $C_2$, the more robust ASNTR is. This observation holds true for each individual dataset, as detailed below:

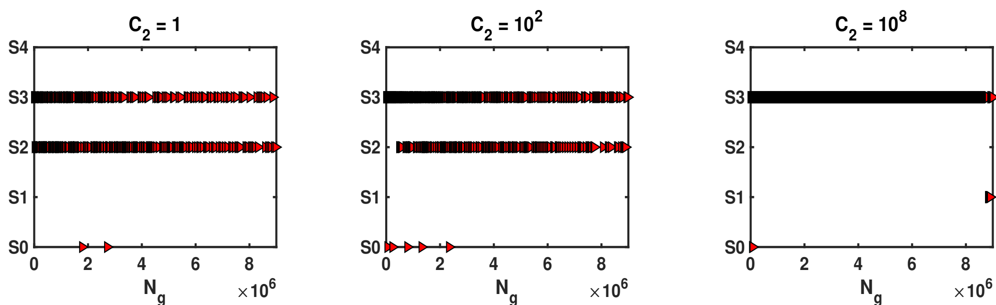
**Figure 5.13** Tracking subsampling in ASNTR.



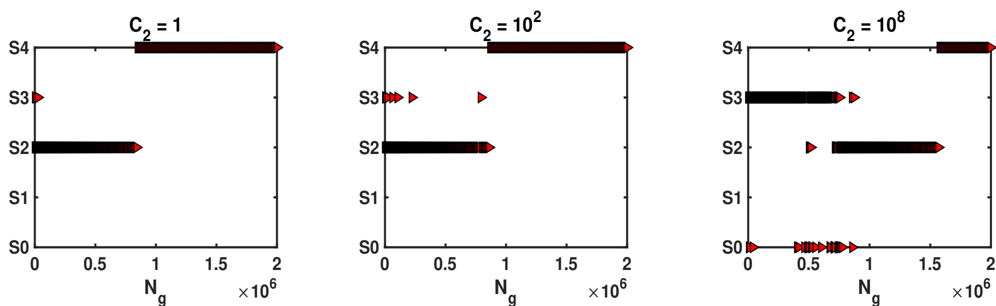

**(a)** Contribution of iterations on `MNIST` (left), `CIFAR10` (middle), `DIGITS` (right)




**(b)** Scatter of iterations with initial `rng(42)` on `MNIST`




**(c)** Scatter of iterations with initial `rng(42)` on `CIFAR10`




**(d)** Scatter of iterations with initial `rng(42)` on `DIGITS`

- `MNIST`: according to Figure 5.13a, the portion of the sampling type $S3$ is higher than others meaning. This means many new batches without needing to be increased are applied in ASNTR during training; i.e., the proposed algorithm with fewer samples, and thus fewer gradient evaluations, can train a network. Nevertheless, ASNTR with different values of $C_2$ in $\rho_{\mathcal{D}_k}$ increases the size of the mini-batches in some of its iterations; see the portions of $S1$ and $S2$ in Figure 5.13a or their scatters in Figure 5.13b. We should note that the sampling type $S1$ occurs almost at the end of the training phase where the algorithm tends to be close to a stationary point of the current approximate function; Figure 5.13b shows this fact.

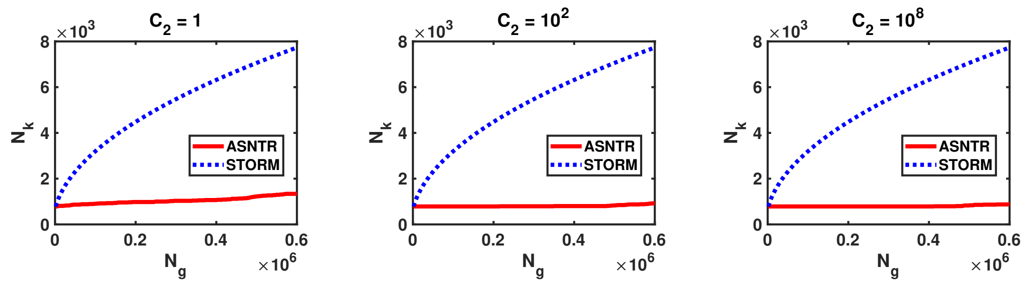
- `CIFAR10`: according to Figure 5.13a, the portion of the sampling type $S3$ is still high. Unlike `MNIST`, the sampling type $S1$ almost never occurs during the training phase. On the other hand, the increase of the sample size through $S2$ with a high portion may compensate for the lack of sufficiently accurate functions and gradients required in ASNTR. These points are also illustrated in Figure 5.13c, which shows how ASNTR successfully trained the `ResNet-20` model without the requirement of frequently enlarging the sample sizes throughout various iterations. For both the `MNIST` and `CIFAR10` problems, $S3$ as the predominant type corresponds to $C_2 = 10^8$.

- `DIGITS`: according to Figure 5.13a, we observe that the main sampling types are $S2$ and $S4$. As the portion of $S2$ increases, the portion of $S3$ decreases and the highest portion of $S3$ corresponds to the largest value of $C_2$. This pattern is similar to what is seen in the `MNIST` and `CIFAR10` datasets. However, in the case of `DIGITS`, the portion of $S4$ is higher. This higher portion of $S4$ in `DIGITS` may be attributed to the smaller number of samples in this dataset ($N = 5000$), which causes ASNTR to quickly encompass all the samples after a few iterations. Notably, the sampling type $S4$ occurs towards the end of the training phase, as shown in Figure 5.13d.

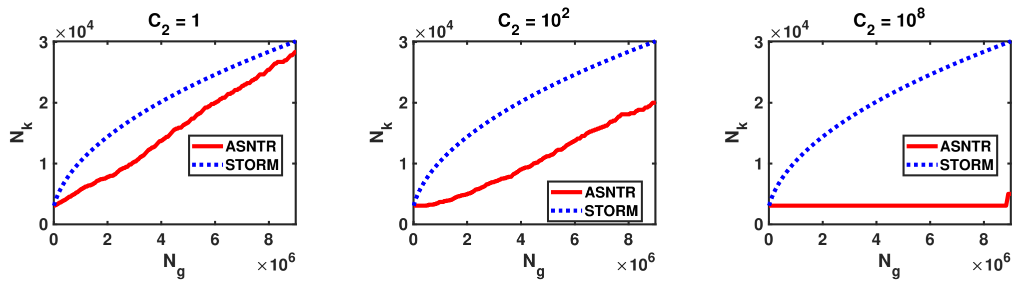
Figure 5.14 compares the progression of batch size growth in both ASNTR and STORM. In contrast to the STORM algorithm, ASNTR increases the batch size only when necessary, which can reduce the computational costs of gradient evaluations. This is considered a significant advantage of ASNTR over STORM. However, according to

this figure, the proposed algorithm needs fewer samples than STORM during the initial phase of the training task, but it requires more samples towards the later part of this phase. Nevertheless, we should notice that the increase in batch size that happened at the end of the training phase is determined by whether $S1$ or $S4$ (see Figure 5.13b-Figure 5.13d). In our experiments, we have observed that ASNTR does not need to be continued with very large $N_g^{\max}$, as it typically achieves the required training accuracy.

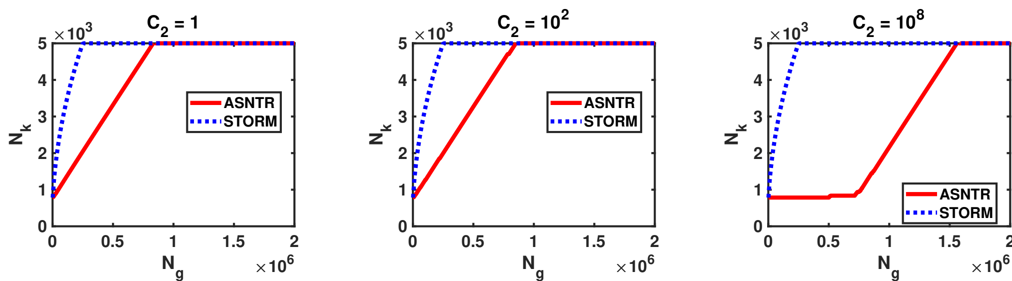
**Figure 5.14** Batch size progress with initial `rng(42)`.




**(a)** MNIST ($N = 6 \times 10^4$)




**(b)** `CIFAR10` ($N = 5 \times 10^4$)




**(c)** `DIGITS` ($N = 5 \times 10^3$)

# 6

# Concluding Remarks

Because of the breakthrough applications of Deep Learning (DL) in various fields, in this thesis, we have considered the target of solving the nonlinear and non-convex optimization problems that arise in the training of Deep Neural Networks (DNNs) for image classification and regression. While first-order methods are commonly used due to their ease of implementation, there has been growing interest in utilizing second-order methods, specifically Hessian-Free or Quasi-Newton approaches. These methods aim to capture curvature information of the objective function and overcome certain limitations of first-order methods. Motivated by this interest, our research has specifically concentrated on Quasi-Newton trust-region methods in stochastic regimes, where subsampling strategies are used to make these methods more practical for training DNNs in real-world scenarios.

**Summary of Contributions**

1. We have considered two well-known limited memory Quasi-Newton Hessian approximations, i.e. L-SR1 and L-BFGS updates, in a trust-region framework involving a particular fixed-size overlapping sampling. An extensive experimental study has been done to see the impact of different factors on the behavior of the algorithms (sL-BFGS-TR and sL-SR1-TR) for training DNNs under the effect of batch normalization (BN) layers. The main findings from our experiments can be summarized as follows. In image classification problems, we have observed that

BN is a key component for the performance of the algorithms with different sample sizes. sL-SR1-TR performs better than sL-BFGS-TR in networks without BN layers. This behavior is in accordance with the property of L-SR1 updates allowing for indefinite Hessian approximations in non-convex optimization. However, sL-BFGS-TR behaves comparably or slightly better than sL-SR1-TR when BN layers are used. Furthermore, the results have shown that using larger batch sizes within a fixed number of epochs leads to lower training accuracy compared to using smaller batch sizes. However, this decrease in accuracy can be mitigated by extending the training duration; in other words, longer training with larger batch sizes can help recover the lost accuracy. Our experiments on training time have shown a slight superiority in the accuracy reached by both algorithms when larger batch sizes are used within a fixed budget of time. This suggests the use of large batch sizes also in view of the parallelization of the algorithms, which is no longer a concern due to advancements in computational resources. Within the fixed budget of time, except for the smallest batch size, both algorithms reveal more efficiency than the second-order STORM algorithm customized by L-BFGS and L-SR1 updates. Finally, our results show that the algorithms in some instances outperform the well-known first-order tuned Adam optimizer.

2. We have presented a modified L-BFGS trust-region method, which improves upon the standard secant condition and has been theoretically shown to provide an increased order of accuracy in Hessian approximation. The stochastic variant, utilizing a particular overlapping subsampling scheme, was compared to its naive variant, where the Hessian approximation is obtained using the standard secant condition. In our experiments on image classification problems, both algorithms with different batch sizes exhibit comparable training performance. Restricted to the experiments with the largest considered batch size, the results show that with a fixed computational time budget both algorithms provide comparable or better testing accuracy than the first-order Adam optimizer. Despite the advantage of not requiring the time-consuming tuning effort needed for Adam, it should be noted that Quasi-Newton trust-region methods have a higher iteration complexity.

3. We have developed a novel second-order stochastic optimization algorithm based on the combination of first- and second-order information. The algorithm combines a trust-region limited memory SR1 second-order direction with a variance-reduced stochastic gradient. We have reported computational experiments showing that the proposed algorithm exhibits comparable or superior performance to the Adam optimizer while requiring significantly less tuning effort.

4. We have investigated an L-SR1 trust-region method incorporating non-monotonicity and a regular fixed-size subsampling. To our knowledge, this was the first attempt to analyze a non-monotone trust-region method in a stochastic setting and for training purposes. Additionally, three distinct approaches for computing the curvature vector required for the L-SR1 update have been examined. We have found that using accumulated empirical Fisher matrix-vector products (Fv) produces better training than when curvature is obtained by subsampled gradient differences or subsampled Hessian matrix-vector products. The results show that the proposed algorithm provides comparable or better testing accuracy than standard stochastic trust-region, with adopted curvature computing strategy, i.e. Fv, and outperforms the well-known Adam optimizer.

5. We have presented, a second-order non-monotone trust-region method that employs an adaptive subsampling strategy. We have incorporated additional sampling into the TR framework in order to control the noise and overcome issues in the convergence analysis coming from biased estimators. Depending on the estimated progress of the algorithm, this can yield different scenarios ranging from mini-batch to full sample functions. We provide convergence analysis for all possible scenarios and show that the proposed method achieves almost sure convergence under standard assumptions for the TR framework. The experiments show the efficiency of the proposed method in training tasks. In comparison to the state-of-the-art counterpart (STORM), our method achieves higher testing accuracy with a fixed budget of gradient evaluations.

**Future Works**

- In our experimental study, we observed that the inclusion or exclusion of batch normalization layers had a direct impact on the performance of the Quasi-Newton trust-region algorithms in training tasks. This intriguing behavior of the algorithms when dealing with networks that incorporate batch normalization layers calls for further investigation.
- Further efforts in the development of adaptive subsampled non-monotone trust-region algorithms could involve exploring more specific approaches for sample size updates and also refining the strategies for Hessian approximation.
- Another interesting future line of research will be devoted to (stochastic) Hessian-Free algorithms.
- Future work will also consist in studying other fields of DL applications that are common with the applications of inverse problems.

# Appendix A

# Programming Comments

In Chapter 2, we have comparatively analyzed the behavior of sL-BFGS-TR and sL-SR1-TR as two stochastic algorithms using QN Hessian approximations in a TR framework for training DNNs in image classification. Since the algorithms are not defined as MATLAB built-in functions, we have exploited the Deep Learning Custom Training Loops to customize their iterations. Implementation details are available at: `https://github.com/MATHinDL/sL_QN_TR/`. In this chapter, however, we provide basic intuition on designing and implementing a DNN.

While there are many Python-based codes provided by authors in DL literature and related repositories, there are limited resources for MATLAB users to customize their own optimizer for training a DNN. We have contributed to filling in this gap by presenting a tutorial based on the MATLAB Deep Learning toolbox on how to implement custom loops for training [82]. We provide general implementation comments by which one can learn how to define an initialized convolutional neural network (CNN) and how to compute functions, gradients, and other quantities required per single iteration of any gradient-based optimization algorithms.

### Introduction

Let us consider a supervised deep learning problem such as (3.3), where $w \in \mathbb{R}^n$ is the vector of trainable parameters and $(x_i, y_i)$ denotes the $i$th sample pair in a given $C$-class training dataset $\{(x_i, y_i)\}_{i=1}^{N}$ with input $x_i$ and target $y_i$. Moreover, $L_i(w)$ is a

single loss function defining the prediction error between the network's output $h(x_i; w)$ and $y_i$ which is converted into a one-hot target vector. In order to find an optimal classification model, the generic problem (3.3) is solved by employing the *softmax cross-entropy* function $L_i(w) = -\sum_{i=1}^{C}(y_i)_k \log(h(x_i; w))_k$ for $i = 1, \ldots, N$.

#### Network Construction

We would like to construct a neural network corresponding to $h(.; w)$ and train it using random data $\{(x_i, y_i)\}_{i=1}^{N_k} \subseteq \{(x_i, y_i)\}_{i=1}^{N}$ per iteration $k$ of an optimizer, where $N_k \leq N$.

The definition of a neural network is done by specifying an array of layers that creates the specified architecture of a network. This architecture is then established using the MATLAB function `layerGraph` that takes `layers` as an input parameter. Moreover, we would like to make use of training algorithms (optimizers) which are not built-in functions. In this case, we can use a model function `dlnetwork` to define architecture as well as customize training loops corresponding user's prescribed algorithm. The `1×1` object `dlnetwork` is a pack of properties including `Layers`, `Connections`, `Learnables`, `State`, `InputNames` and `OutputNames`. We illustrate with an example how to define a `dlnetwork` and show its main properties, i.e., `Layers` and `Learnables`.

```
>> layers = [
>>     imageInputLayer(inputSize,'Normalization','none','Name','input')
>>     convolution2dLayer(5,20, 'Padding', 'same', 'Name', 'conv1')
>>     batchNormalizationLayer('Name', 'bn1')
>>     reluLayer('Name','relu1')
>>     convolution2dLayer(5, 50, 'Padding', 1,'Name', 'conv2')
>>     batchNormalizationLayer('Name', 'bn2')
>>     reluLayer('Name','relu2')
>>     maxPooling2dLayer(2, 'Stride', 2, 'Name', 'maxpool1')
>>     fullyConnectedLayer(numClasses, 'Name', 'fc1')
>>     softmaxLayer('Name', 'softmax')];
>> lgraph = layerGraph(layers);
>> dlNet = dlnetwork(lgraph);
```

The instruction above defines different layers as well as `softmaxLayer` which is important to be defined for custom training loops in a classification task. Let us consider images with a determined size `inputSize` belonging to a number of classes `numClasses`. Through `imageInputLayer`, one can define different input normalization strategies such as `'zerocenter'` or `'zscore'` in place of `'none'`; the syntax above states that data input without normalization. The function `layerGraph` specifies the sequential ordered `layers` as the architecture of the network with a more complex graph structure. The object `layerGraph` must be converted into an initialized network, i.e., `dlnetwork` for the training task.

Figure A.1 shows how to see different properties of the `dlNet` object. For instance, `dlNet.Layers` contains the network's architecture while `dlNet.Learnables` contains all learnable parameters, i.e., parameter vector $w \in \mathbb{R}^n$ in (3.3). We note that `Weights` and `Bias` of the convolutional layers, and all `Offset` and `Scale` of the batch normalization layers [38] are included in `dlNet.Learnables`. The object `dlNet` represents an initialized DNN; in fact, initial values are given to `dlNet.Learnables` through `convolution2dLayer` and `batchNormalizationLayer`.

The DL toolbox provides some default initializers for `dlNet.Learnables`. For instance, weights and biases are respectively initialized by the Glorot (Xavier) initializer [29] and zeros through `convolution2dLayer`.

In order to implement the defined network, we should notice some points. Figure A.1 shows that `dlNet.Learnables` are layered and stored in a `table` format. For making calculations easier in the training loops of an optimizer, we unroll the values of `dlNet.Learnables` into a large parameter vector `w` representing $w \in \mathbb{R}^n$ in (3.3).

Figure A.1 also shows that variables are stored in `dlArray` object. In fact, storing variables and data in `dlArray` for custom training loops enables functions to compute and use derivatives through automatic differentiation. Therefore, we should use `extractdata` for extracting numeric values. If a GPU is available, we also use `gather` to collect the results of a GPU operation.

**Figure A.1** Properties in the MATLAB object `dlNet`.

```
Command Window
>> dlNet.Layers

ans =

  10x1 Layer array with layers:

     1   'input'       Image Input           28x28x1 images
     2   'conv-1'      Convolution           20 5x5x1 convolutions with stride [1  1] and padding 'same'
     3   'bn-1'        Batch Normalization   Batch normalization with 20 channels
     4   'relu-1'      ReLU                  ReLU
     5   'conv-2'      Convolution           50 5x5x20 convolutions with stride [1  1] and padding [1  1  1  1]
     6   'bn-2'        Batch Normalization   Batch normalization with 50 channels
     7   'relu-2'      ReLU                  ReLU
     8   'maxpool-1'   Max Pooling           2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     9   'fc-1'        Fully Connected       10 fully connected layer
    10   'softmax'     Softmax               softmax
>>
>> dlNet.Learnables

ans =

  10x3 table

     Layer      Parameter          Value
    ________    _________    ____________________

    "conv-1"    "Weights"    { 5x5x1x20  dlarray}
    "conv-1"    "Bias"       { 1x1x20    dlarray}
    "bn-1"      "Offset"     { 1x1x20    dlarray}
    "bn-1"      "Scale"      { 1x1x20    dlarray}
    "conv-2"    "Weights"    { 5x5x20x50 dlarray}
    "conv-2"    "Bias"       { 1x1x50    dlarray}
    "bn-2"      "Offset"     { 1x1x50    dlarray}
    "bn-2"      "Scale"      { 1x1x50    dlarray}
    "fc-1"      "Weights"    {10x8450    dlarray}
    "fc-1"      "Bias"       {10x1       dlarray}

>>
>> dlNet.State

ans =

  4x3 table

    Layer        Parameter              Value
    ______    _________________    _______________

    "bn-1"    "TrainedMean"        {1x1x20 single}
    "bn-1"    "TrainedVariance"    {1x1x20 single}
    "bn-2"    "TrainedMean"        {1x1x50 single}
    "bn-2"    "TrainedVariance"    {1x1x50 single}
```

```
>> w = [];
>> layeredParam = dlNet.Learnables.Value;
>> for layer = 1: size(layeredParam,1)
>>     val = double(gather(extractdata(layeredParam{layer,1})));
>>     w   = [w; val(:)];
>> end
```

### Network Updating

The initial parameter vector $w_0$ is gradually updated through the sequence $\{w_k\}$ according to an updating rule defined by an optimizer. Such an updating rule, for example, in algorithms based on QN methods is defined by $w_{k+1} = w_k + p_k$ where $w_k$ is the current parameter vector and $p_k$ is a search direction obtained by solving $B_k p_k = -g_k$ with Hessian approximation $B_k \in \mathbb{R}^{n \times n}$ and gradient $g_k \in \mathbb{R}^n$. Correspondingly, the *initialized*

dlNet must be gradually updated to produce a *trained* dlNet at the end of the training process. Therefore, we use MATLAB for- or while-training loops for updating the values of dlNet.Learnables. At each iteration of a training loop (algorithm), the loss function and its gradient are evaluated. What follows contains the core of implementation for computing these quantities by forward and backward propagation algorithms using automatic differentiation.

Given a batch of training data and their true labels respectively denoted by X and Y, the initialized network is iteratively trained by it a forward propagation pass to compute the overall loss (loss) and a backward propagation pass to compute the gradient (gradient). For computing loss and gradient, we primarily use a function handle such as, here, @modelgradient in order to define functions forward, crossentropy and dlgradient. Then, the values of predicted labels, loss, and gradient models are determined by the function dlfeval. In the DL toolbox, training batch $X$ must be converted in the dlarray format denoted by dlX where also labeled as SSCB standing for **S**patial, **S**patial, **C**hannel and **B**atch observations. As mentioned, dlarray format enables functions of the DL toolbox to compute derivatives by automatic differentiation. We should notice dlfeval works with dlX and dlNet including layered parameters stored in dlArray format. As a result, dlgradient and loss give respectively layered gradient variables and a loss variable in dlArray formats. Usually, we should convert the layered variables of dlgradient into a vector g and obtain its numeric values for computations needed in a training loop. The following instructions illustrate these comments.

```
>> dlX = dlarray(single(X),'SSCB');
>> [gradient, loss, state] = dlfeval(@modelgradient, dlNet, dlX, Y);
>>
>> function [gradient, loss, state] = modelgradient(dlNet, dlX, Y)
>>     [dlYp, state]  = forward(dlNet, dlX);
>>     loss           = crossentropy(dlYp, Y);
>>     gradient       = dlgradient(loss, dlNet.Learnables);
>> end

>> F = double(gather(extractdata(loss)));
```

```
>> g = [];
>> layeredGrad = gradient.Value;
>> for layer = 1: size(layeredGrad,1)
>>    val = double(gather(extractdata(layeredGrad{layer,1})));
>>    g   = [g; val(:)];
>> end
```

Besides function and gradient evaluations, an optimizer may have other required computations. For instance, as mentioned above, given vector $g_k$ and a QN Hessian approximation $B_k$, a search direction $p_k$ for updating $w_k$ as $w_k + p_k$ requires us to compute $B_k p_k = -g_k$ using any proper solver. This solver may use the computed gradient as the vector `g` and then provides `p` as a vector as well. Therefore, we should convert the numeric vector `p` into a layered variable in order to be able to update layered parameters `dlNet.Learnables` for the next iteration. The following syntax shows these commands.

```
>> Direction     = dlNet.Learnables;
>> end_array     = 0;
>> for layer = 1: size( Direction, 1)
>>     layer_size     = size( Direction.Value{layer,1} );
>>     start_array    = end_array + 1;
>>     end_array      = end_array + prod(layer_size);
>>     p_segment      = p(start_array : end_array);
>>     tensor         = dlarray(single(reshape(p_segment, layer_size)));
>>     Direction.Value{layer, 1} = tensor;
>> end
>> dlNet.Learnables = dlupdate(@(w,p) w + p, dlNet.Learnables, Direction);
```

**Network Evaluation**

We can evaluate the performance of the defined network in the training phase. In order to monitor the training accuracy (the accuracy of the network using a training dataset), we can use the following statements where we should note that each column of the `Y`

denotes a one-hot vector of a true label while every column of the `dlYp` is a probability coming from the softmax layer. Nevertheless, since the function `max` finds the maximum value and its corresponding location, transforming probabilities into one-hot vectors is not needed.

```
>> Yp          = extractdata(dlYp)
>> [~,ind_Yp] = max(Yp, [], 1);
>> [~,ind_Y]  = max(Y, [], 1);
>> accuracy   = mean(ind_Y == ind_Yp);
```

The training accuracy can be compared with the accuracy of the network using a validation dataset, which is denoted by `acc`. This comparison can help us to analyze the real performance of the network. Given `dlX_v` and `Y_v` respectively as validation examples in `dlArray` format and their one-hot targets, the evaluation of the network's performance on a validation dataset can be also monitored during the training process as follows

```
>>  dlYp        = predict(dlNet, dlX_v);
>>  loss        = crossentropy(dlYp, Y_v);
>>  Yp          = extractdata(dlYp)
>>  [~,ind_Yp] = max(Yp, [], 1);
>>  [~,ind_Y]  = max(Yv, [], 1);
>>  acc        = mean(ind_Y == ind_Yp);
```

There is another important point for consideration in the implementation. There are some layers that behave differently during the training and inference phases. For example, a dropout layer randomly sets input elements to zero during training while it does not change the input during inference. To set the network to the desired functionality, the functions `forward` and `predict` are respectively used to compute the outputs of the training and inference phases in proper ways. Another layer with different behavior in the training and inference phases is the batch normalization layer. However, applying `predict` cannot lonely help the network using this type of layer to perform well in the inference phase. To make predictions with the network after training, batch normalization requires a fixed mean and variance to normalize the data; this fixed mean and variance

can be approximated during training using running statistic computations. Specifying `state` as the second output of the function `forward` during training requires the function `forward` to compute the mean running average $\bar{\mu}$ and the variance running average $\bar{\sigma^2}$ during the training phase; see (3.2). Therefore, these statistics as `TrainedMean` and `TrainedVariance` parameters can be updated at the end of every single training iteration by the following syntax

```
>> dlNet.State = state;
```

When the function `predict` employs `dlNet` to make predictions, batch normalization layers in `dlNet` object use `TrainedMean` and `TrainedVariance` stored in `dlNet.State`; see Figure A.1. Failing to update `state` during the training phase causes batch normalization layers to use the initial mean and variance which results in a poor prediction in the inference phase.

# Appendix B

# Two Solvers for the TR Subproblem

### Computing TR subproblem with an L-BFGS matrix

This section describes how to solve the TR subproblem (4.1) using L-BFGS; see [1, 15, 63] for more details and Algorithm C.3 in Appendix C, where the following procedure is outlined.

Let $B_k$ be an L-BFGS compact matrix (4.9). Using Theorem 5.1, the global solution of the TR subproblem (4.1) can be obtained by exploiting the following two strategies:

**Spectral decomposition of $B_k$** By the thin QR factorization of the matrix $\Psi_k$, $\Psi_k = Q_k R_k$, or the Cholesky factorization of the matrix $\Psi_k^T \Psi_k$, $\Psi_k^T \Psi_k = R^T R$, and then spectral decomposition of the small matrix $R_k M_k R_k^T$ as $R_k M_k R_k^T = U_k \hat{\Lambda} U_k^T$, we have

$$B_k = B_0 + Q_k R_k M_k R_k^T Q_k^T = \gamma_k I + Q_k U_k \hat{\Lambda} U_k^T Q_k^T,$$

where $U_k$ and $\hat{\Lambda}$ respectively are orthogonal and diagonal matrices. Now, let $P_\parallel \triangleq Q_k U_k$ (or let $P_\parallel \triangleq (\Psi_k R_k^{-1} U_k)^T$) and $P_\perp \triangleq (Q_k U_k)^\perp$ where $\perp$ is orthogonal complement (perpendicular). By Theorem 2.1.1 in [31], we have $P^T P = P P^T = I$ where

$$P \triangleq \begin{bmatrix} P_\parallel & P_\perp \end{bmatrix} \in \mathbb{R}^{n \times n}. \tag{B.1}$$

Therefore, the spectral decomposition of $B_k$ is obtained as

$$B_k = P\Lambda P^T, \qquad \Lambda \triangleq \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} = \begin{bmatrix} \hat{\Lambda} + \gamma_k I & 0 \\ 0 & \gamma_k I \end{bmatrix}, \tag{B.2}$$

where $\Lambda = \operatorname{diag}(\hat{\lambda}_1, \dots, \hat{\lambda}_n) = \operatorname{diag}(\hat{\lambda}_1 + \gamma_k, \dots, \hat{\lambda}_k + \gamma_k, \gamma_k, \dots, \gamma_k) \in \mathbb{R}^{n\times n}$ with $\Lambda_1 \in \mathbb{R}^{2l\times 2l}$ and $\Lambda_2 \in \mathbb{R}^{(n-2l)\times(n-2l)}$ when $k > 2l$. We note that $\Lambda_1 \in \mathbb{R}^{k\times k}$ and $\Lambda_2 \in \mathbb{R}^{(n-k)\times(n-k)}$ when $k \leq 2l$. We also assume the eigenvalues in $\Lambda_1$ are ordered in increasing values. Notice that $\Lambda_1$ includes at most $2l$ elements with limited memory parameter $l$.

**Inversion by Sherman-Morrison-Woodbury formula** By dropping subscript $k$ in (4.9) and using the Sherman-Morrison-Woodbury formula to compute the inverse of the coefficient matrix in (2.12), we have

$$p(\sigma) = -(B + \sigma I)^{-1} g = -\frac{1}{\tau}\left(I - \Psi\left(\tau M^{-1} + \Psi^T\Psi\right)^{-1}\Psi^T\right) g, \tag{B.3}$$

where $\tau = \gamma + \sigma$. By using (B.2), the first optimality condition in (2.12) can be written as

$$(\Lambda + \sigma I)v = -P^T g, \tag{B.4}$$

where

$$v = P^T p, \qquad P^T g \triangleq \begin{bmatrix} g_{\parallel} \\ g_{\perp} \end{bmatrix} = \begin{bmatrix} P_{\parallel}^T g \\ P_{\perp}^T g \end{bmatrix}, \tag{B.5}$$

and therefore

$$\|p(\sigma)\| = \|v(\sigma)\| = \sqrt{\left\{\sum_{i=1}^{k} \frac{(g_{\parallel})_i^2}{(\lambda_i + \sigma)^2}\right\} + \frac{\|g_{\perp}\|^2}{(\gamma + \sigma)^2}}, \tag{B.6}$$

where $\|g_{\perp}\|^2 = \|g\|^2 - \|g_{\parallel}\|^2$. This makes the computation of $\|p\|$ feasible without computing $p$ explicitly. Let $p_u \triangleq p(0)$ as an unconstrained minimizer for (4.1) be the solution of the first optimality condition in (2.12), for which $\sigma = 0$ makes the second optimality condition hold. Now, we consider the following cases:

- If $\|p_u\| \leq \delta$, the optimal solution of (4.1) using (B.3) is computed as

$$(\sigma^*, p^*) = (0, p_u) = (0, p(0)). \tag{B.7}$$

- If $\|p_u\| > \delta$, then $p^*$ must lie on the boundary of the TR to hold the second optimality condition. To impose this, $\sigma^*$ must be the root of the following equation which is determined by the Newton method proposed in [15]:

$$\phi(\sigma) \triangleq \frac{1}{\|p(\sigma)\|} - \frac{1}{\delta} = 0. \tag{B.8}$$

Therefore, using (B.3), the global solution is computed as

$$(\sigma^*, p^*) = (\sigma^*, p(\sigma^*)). \tag{B.9}$$

#### Computing TR subproblem with an L-SR1 matrix

For solving (4.1) where $B_k$ is a compact L-SR1 matrix (4.18), an efficient algorithm called the *Orthonormal Basis L-SR1* (OBS) was proposed in [15]. We summarize this approach here; see Algorithm C.5 in Appendix C, where it describes the following procedure.

Let (B.2) be the eigenvalue decomposition of (4.18), where $\Lambda = \text{diag}(\hat{\lambda}_1, \dots, \hat{\lambda}_n) = \text{diag}(\hat{\lambda}_1 + \gamma_k, \dots, \hat{\lambda}_k + \gamma_k, \gamma_k, \dots, \gamma_k) \in \mathbb{R}^{n \times n}$ with $\Lambda_1 \in \mathbb{R}^{l \times l}$ and $\Lambda_2 \in \mathbb{R}^{(n-l)\times(n-l)}$ when $k > l$. We note that $\Lambda_1 \in \mathbb{R}^{k \times k}$ and $\Lambda_2 \in \mathbb{R}^{(n-k)\times(n-k)}$ when $k \leq l$. We also assume the eigenvalues in $\Lambda_1$ are ordered in increasing values. Notice that $\Lambda_1$ includes at most $l$ elements with limited memory parameter $l$. The OBS method exploits the Sherman-Morrison-Woodbury formula in different cases for L-SR1 $B_k$; by dropping subscript $k$ in (4.18), these cases are:

**$B$ is positive definite** In this case, the global solution of (4.1) is (B.7) or (B.9).

**$B$ is positive semi-definite (singular)** Since $\gamma \neq 0$ and $B$ is positive semi-definite with all nonnegative eigenvalues, then $\lambda_{min} = \min\{\lambda_1, \gamma\} = \lambda_1 = 0$. Let $r$ be the multiplicity of the $\lambda_{min}$; therefore,

$$0 = \lambda_1 = \lambda_2 = \cdots = \lambda_r < \lambda_{r+1} \leq \lambda_{r+2} \leq \cdots \leq \lambda_k.$$

For $\sigma > -\lambda_{min} = 0$, the matrix $(\Lambda + \sigma I)$ in (B.4) is invertible, and thus, $\|p(\sigma)\|$ in (B.6) is well-defined. For $\sigma = -\lambda_{min} = 0$, we consider the two following sub-cases[1]:

1. If $\lim_{\sigma\to 0^+} \phi(\sigma) < 0$, then $\lim_{\sigma\to 0^+} \|p(\sigma)\| > \delta$. Here, the OBS algorithm uses Newton's method to find $\sigma^* \in (0, \infty)$ so that the global solution $p^*$ lies on the boundary of trust-region, i.e., $\phi(\sigma^*) = 0$. This solution $p^* = p(\sigma^*)$ is computed using (B.3); by that, the global pair solution $(\sigma^*, p^*)$ satisfies the first and second optimal conditions in (2.12).

2. If $\lim_{\sigma\to 0^+} \phi(\sigma) \geq 0$, then $\lim_{\sigma\to 0^+} \|p(\sigma)\| \leq \delta$. It can be proved that $\phi(\sigma)$ is strictly increasing for $\sigma > 0$ (see Lemma 7.3.1 in [18]). This makes $\phi(\sigma) \geq 0$ for $\sigma > 0$ as it is non-negative at $0^+$, and thus, $\phi(\sigma)$ can only have a root $\sigma^* = 0$ in $\sigma \geq 0$. Here, we should notice that even if $\phi(\sigma) > 0$, the solution $\sigma^* = 0$ makes the second optimality condition in (2.12) hold. Since matrix $B + \sigma I$ at $\sigma^* = 0$ is not invertible, the global solution $p^*$ for the first optimality condition in (2.12) is computed by

$$\begin{aligned} p^* = p(\sigma^*) &= -(B + \sigma^* I)^\dagger g = -P(\Lambda + \sigma^* I)^\dagger P^T g \\ &= -P_\parallel(\Lambda_1 + \sigma^* I)^\dagger P_\parallel^T g - \frac{1}{\gamma + \sigma^*} P_\perp P_\perp^T g \\ &= -\Psi R^{-1} U(\Lambda_1 + \sigma^* I)^\dagger g_\parallel - \frac{1}{\gamma + \sigma^*} P_\perp P_\perp^T g, \end{aligned} \tag{B.10}$$

where $(g_\parallel)_i = (P_\parallel^T g)_i = 0$ for $i = 1, \ldots, r$ if $\sigma^* = -\lambda_{min} = -\lambda_1 = 0$, and

$$P_\perp P_\perp^T g = (I - P_\parallel P_\parallel^T) g = (I - \Psi R^{-1} R^{-T} \Psi^T) g.$$

Therefore, both optimality conditions in (2.12) hold for the pair solution $(\sigma^*, p^*)$.

**B is indefinite** Let $r$ be the algebraic multiplicity of the leftmost eigenvalue $\lambda_{min}$. Since $B$ is indefinite and $\gamma \neq 0$, we have $\lambda_{min} = \min\{\lambda_1, \gamma\} < 0$.
Obviously, for $\sigma > -\lambda_{min}$, the matrix $(\Lambda + \sigma I)$ in (B.4) is invertible, and thus, $\|p(\sigma)\|$ in (B.6) is well-defined. For $\sigma = -\lambda_{min}$, we discuss the two following cases:

1. If $\lim_{\sigma\to -\lambda_{min}^+} \phi(\sigma) < 0$, then $\lim_{\sigma\to -\lambda_{min}^+} \|p(\sigma)\| > \delta$. The OBS algorithm uses

[1] To have a well-defined expression in (B.6), we will discuss in limit setting (at $-\lambda_{min}^+$).

Newton's method to find $\sigma^* \in (-\lambda_{min}, \infty)$ as the root of $\phi(\sigma) = 0$ so that the global solution $p^*$ lies on the boundary of trust-region. By using (B.3) to compute $p^* = p(\sigma^*)$, the pair $(\sigma^*, p^*)$ satisfies the both conditions in (2.12).

2. If $\lim_{\sigma \to -\lambda_{min}^+} \phi(\sigma) \geq 0$, then $\lim_{\sigma \to -\lambda_{min}^+} \|p(\sigma)\| \geq \delta$. For $\sigma > -\lambda_{min}$, we have $\phi(\sigma) \geq 0$ but the solution $\sigma^* = -\lambda_{min}$ as the only root of $\phi(\sigma) = 0$ is a positive number, which cannot satisfy the second optimal condition when $\phi(\sigma)$ is strictly positive. Hence, we should consider the cases of equality and inequality separately:

**Equality.** Let $\lim_{\sigma \to -\lambda_{min}^+} \phi(\sigma) = 0$. Since matrix $B + \sigma I$ at $\sigma^* = -\lambda_{min}$ is not invertible, the global solution $p^*$ for the first optimality condition in (2.12) is computed using (B.10) by

$$p^* = \begin{cases} -\Psi R^{-1} U (\Lambda_1 + \sigma^* I)^\dagger g_\| - \dfrac{1}{\gamma + \sigma^*} P_\perp P_\perp^T g, & \sigma^* \neq -\gamma, \\ -\Psi R^{-1} U (\Lambda_1 + \sigma^* I)^\dagger g_\|, & \sigma^* = -\gamma, \end{cases} \tag{B.11}$$

where $g_\perp = P_\perp^T g = 0$, and thus $\|g_\perp\| = 0$ if $\sigma^* = -\lambda_{min} = -\gamma$. For $i = 1, \ldots, r$, we have $(g_\|)_i = (P_\|^T g)_i = 0$ if $\sigma^* = -\lambda_{min} = -\lambda_1$.

We note that both optimality conditions in (2.12) hold for the computed $(\sigma^*, p^*)$.

**Inequality.** Let $\lim_{\sigma \to -\lambda_{min}^+} \phi(\sigma) > 0$, then $\lim_{\sigma \to -\lambda_{min}^+} \|p(\sigma)\| < \delta$. As mentioned before, $\sigma = -\lambda_{min} > 0$ cannot satisfy the second optimality condition. In this case, so-called *hard case*, we attempt to find a solution which lies on the boundary. For $\sigma^* = -\lambda_{min}$, this optimal solution is given by

$$p^* = \hat{p}^* + z^*, \tag{B.12}$$

where $\hat{p}^* = -(B + \sigma^* I)^\dagger g$ is computed by (B.11) and $z^* = \alpha u_{min}$. Vector $u_{min}$ is a unit eigenvector in the subspace associated with $\lambda_{min}$ and $\alpha$ is obtained so that $\|p^*\| = \delta$; i.e.,

$$\alpha = \sqrt{\delta^2 - \|\hat{p}^*\|^2}. \tag{B.13}$$

The computation of $u_{min}$ depends on $\lambda_{min} = \min\{\lambda_1, \gamma\}$. If $\lambda_{min} = \lambda_1$ then the

first column of $P$ is a leftmost eigenvector of $B$, and thus, $u_{min}$ is set to the first column of $P_{\|}$. On other hand, if $\lambda_{min} = \gamma$, then any vector in the column space of $P_{\perp}$ will be an eigenvector of $B$ corresponding to $\lambda_{min}$. However, we avoid forming matrix $P_{\perp}$ to compute $P_{\perp}P_{\perp}^T g$ in (B.11) if $\lambda_{min} = \lambda_1$. By the definition (B.1), we have

$$\text{Range}(P_{\perp}) = \text{Range}(P_{\|})^{\perp}, \qquad \text{Range}(P_{\|}) = \text{Ker}(I - P_{\|}P_{\|}^T).$$

To find a vector in the column space of $P_{\perp}$, we use $I - P_{\|}P_{\|}^T$ as projection matrix mapping onto the column space of $P_{\perp}$. For simplicity, we can map one canonical basis vector at a time onto the column space of $P_{\perp}$ until a nonzero vector is obtained. This practical process, repeated at most $k+1$ times, will result in a vector that lies in $\text{Range}(P_{\perp})$; i.e.,

$$u_{min} \triangleq (I - P_{\|}P_{\|}^T)e_j, \tag{B.14}$$

for $j = 1, 2, \ldots k+1$ with $\|u_{min}\| \neq 0$; because $e_j \in \text{Range}(P_{\|})$ and

$$\text{rank}(P_{\|}) = \dim \text{Range}(P_{\|}) = \dim \text{Kerl}(I - P_{\|}P_{\|}^T) = k.$$

# Appendix C

# Additional Algorithms

This appendix contains a collection of specific algorithms that are referenced in the thesis. The algorithm notes at the bottom of some algorithms specify the hyper-parameters associated with them.

**Algorithm C.1** Trust-Region radius adjustment

1: **Inputs:**

Current iteration $k$, $p_k$, $\delta_k$, $\rho_k$, $0 < \tau_2 < 0.5 < \tau_3 < 1$, $0 < \eta_2 \leq 0.5$, $0.5 < \eta_3 < 1 < \eta_4$

2: **if** $\rho_k > \tau_3$ **then**

3:     **if** $\|p_k\| \leq \eta_3 \delta_k$ **then**

4:         $\delta_{k+1} = \delta_k$

5:     **else**

6:         $\delta_{k+1} = \min(\eta_4 \delta_k, \delta_{max})$

7:     **end if**

8: **else if** $\tau_2 \leq \rho_k \leq \tau_3$ **then**

9:     $\delta_{k+1} = \delta_k$

10: **else**

11:     $\delta_{k+1} = \eta_2 \delta_k$

12: **end if**

*Algorithm's note:* $\delta_{max} = 10,\ \eta_2 = 0.5,\ \eta_3 = 0.8,\ \eta_4 = 2,\ \tau_2 = 0.1, \tau_3 = 0.75.$

**Algorithm C.2** L-BFGS Hessian initialization

1: **Inputs:** Current iteration $k$ and storage matrices $S_{k+1}$, $Y_{k+1}$.

2: Compute the smallest eigenvalue $\hat{\lambda}$ of (4.15)

3: **if** $\hat{\lambda} > 0$ **then**

4: $\quad\gamma_{k+1} = \max\{1, c\hat{\lambda}\} \in (0, \hat{\lambda})$

5: **else**

6: $\quad$Compute $\gamma_k^h$ by (4.12) and set $\gamma_{k+1} = \max\{1, \gamma_k^h\}$

7: **end if**

*Algorithm's note:* $c = 0.9$.

**Algorithm C.3** Orthonormal Basis BFGS (OBB)

1: **Inputs:**

Current iteration $k$, $\delta \triangleq \delta_k$, $g \triangleq g_k$ and $B \triangleq B_k$ : $\Psi \triangleq \Psi_k$, $M^{-1} \triangleq M_k^{-1}$, $\gamma \triangleq \gamma_k$

2: Compute the thin QR factors $Q$ and $R$ of $\Psi$ or the Cholesky factor $R$ of $\Psi^T\Psi$

3: Compute the spectral decomposition of matrix $RMR^T$, i.e., $RMR^T = U\hat{\Lambda}U^T$

4: Set $\hat{\Lambda} = \text{diag}(\hat{\lambda}_1, \ldots, \hat{\lambda}_k)$ such that $\hat{\lambda}_1 \leq \ldots \leq \hat{\lambda}_k$ and $\lambda_{min} = \min\{\lambda_1, \gamma\}$

5: Compute the spectral of $B_k$ as $\Lambda_1 = \hat{\Lambda} + \gamma I$

6: Compute $P_{\|} = QU$ or $P_{\|} = (\Psi R^{-1}U)^T$ and $g_{\|} = P_{\|}^T g$

7: **if** $\phi(0) \geq 0$ **then**

8: $\quad$Set: $\sigma^* = 0$

9: $\quad$Compute $p^*$ with (B.3) as solution of $(B_k + \sigma^* I)p = -g$

10: **else**

11: $\quad$Compute a root $\sigma^* \in (0, \infty)$ of (B.8) by Newton method [15]

12: $\quad$Compute $p^*$ with (B.3) as solution of $(B_k + \sigma^* I)p = -g$

13: **end if**

**Algorithm C.4** L-SR1 Hessian initialization

1: **Inputs:** Current iteration $k$ and storage matrices $S_{k+1}$, $Y_{k+1}$
2: Compute the smallest eigenvalue $\hat{\lambda}$ of (4.15)
3: **if** $\hat{\lambda} > 0$ **then**
4: $\quad \gamma_{k+1} = \max\{c, c_1\hat{\lambda}\}$
5: **else**
6: $\quad \gamma_{k+1} = \min\{-c, c_2\hat{\lambda}\}$
7: **end if**

*Algorithm's note:* $c_1 = 0.5$, $c_2 = 1.5$, $c = 10^{-6}$.

**Algorithm C.5** Orthonormal Basis SR1 (OBS)

1: **Inputs:**
Current iteration $k$, $\delta \triangleq \delta_k$, $g \triangleq g_k$ and $B \triangleq B_k$ : $\Psi \triangleq \Psi_k$, $M^{-1} \triangleq M_k^{-1}$, $\gamma \triangleq \gamma_k$
2: Compute the thin QR factors $Q$ and $R$ of $\Psi$ or the Cholesky factor $R$ of $\Psi^T\Psi$
3: Compute the spectral decomposition of matrix $RMR^T$, i.e., $RMR^T = U\hat{\Lambda}U^T$
4: Set $\hat{\Lambda} = \text{diag}(\hat{\lambda}_1, \ldots, \hat{\lambda}_k)$ such that $\hat{\lambda}_1 \leq \ldots \leq \hat{\lambda}_k$ and $\lambda_{min} = \min\{\lambda_1, \gamma\}$
5: Compute the spectral of $B_k$ as $\Lambda_1 = \hat{\Lambda} + \gamma I$
6: Compute $P_{\|} = QU$ or $P_{\|} = (\Psi R^{-1}U)^T$ and $g_{\|} = P_{\|}^T g$
7: **if** Case I: $\lambda_{min} > 0$ and $\phi(0) \geq 0$ **then**
8: $\quad$ Set: $\sigma^* = 0$
9: $\quad$ Compute $p^*$ with (B.3) as solution of $(B_k + \sigma^* I)p = -g$
10: **else if** Case II: $\lambda_{min} \leq 0$ and $\phi(-\lambda_{min}) \geq 0$ **then**
11: $\quad$ Set: $\sigma^* = -\lambda_{min}$
12: $\quad$ Compute $p^*$ with (B.10) as solution of $(B_k + \sigma^* I)p = -g$
13: $\quad$ **if** Case III: $\lambda_{min} < 0$ **then**
14: $\quad\quad$ Compute $\alpha$ and $u_{min}$ with (B.12) for $z^* = \alpha u_{min}$
15: $\quad\quad$ Update: $p^* = p^* + z^*$
16: $\quad$ **end if**
17: **else**
18: $\quad$ Compute a root $\sigma^* \in (\max\{-\lambda_{min}, 0\}, \infty)$ of (B.8) by Newton method [15]
19: $\quad$ Compute $p^*$ with (B.3) as solution of $(B_k + \sigma^* I)p = -g$
20: **end if**

**Algorithm C.6** sL-BFGS-TR (regular)

1: **Inputs:** $w_0 \in \mathbb{R}^n$, $\epsilon > 0$, epoch$_{max}$, $l$, $\gamma_0 > 0$, $S_0 = Y_0 = [.]$, $0 < \tau, \tau_1 < 1$

2: **for** $k = 0, 1, \ldots$ **do**

3: Take a random and uniform minibatch (without replacement) $J_k$ of fixed-size $N_k$

4: Evaluate subsampled function $f_{\mathcal{N}_k}(w_k)$ and its gradients $g_k$

5: **if** epoch $\geq$ epoch$_{max}$ or training accuracy $\geq 100\%$ **then**

6: **Stop training**

7: **end if**

8: Compute $p_k$ using Algorithm C.3

9: Evaluate $f_{\mathcal{N}_k}(w_t)$ and $g_t$ at the trial point $w_t = w_k + p_k$

10: Obtain $(s_k, y_k) = (w_t - w_k, g_t - g_k)$ and $\rho_k = \dfrac{f_{\mathcal{N}_k}(w_t) - f_{\mathcal{N}_k}(w_k)}{Q(p_k)}$

11: **if** $\rho_k \geq \tau_1$ **then**

12: $w_{k+1} = w_t$

13: **else**

14: $w_{k+1} = w_k$

15: **end if**

16: Update $\delta_k$ by Algorithm C.1

17: **if** $s_k^T y_k > \tau \|s_k\|^2$ **then**

18: Update storage matrices $S_{k+1}$ and $Y_{k+1}$ by $l$ recent $\{s_j, y_j\}_{j=k-l+1}^{k}$

19: Compute $\gamma_{k+1}$ for $B_0$ by Algorithm C.2 and $B_{k+1}$ by (4.9)

20: **else**

21: Set $B_{k+1} = B_k$

22: **end if**

23: **end for**

**Algorithm C.7** sL-SR1-TR (regular)

1: **Inputs:** $w_0 \in \mathbb{R}^n$, $\epsilon > 0$, $\text{epoch}_{max}$, $l$, $\gamma_0 > 0$, $S_0 = Y_0 = [.]$, $0 < \tau, \tau_1 < 1$

2: **for** $k = 0, 1, \ldots$ **do**

3: Take a random and uniform minibatch (without replacement) $J_k$ of fixed-size $N_k$

4: Evaluate subsampled function $f_{\mathcal{N}_k}(w_k)$ and its gradients $g_k$

5: **if** epoch $\geq \text{epoch}_{max}$ or training accuracy $\geq 100\%$ **then**

6: **Stop training**

7: **end if**

8: Compute $p_k$ using Algorithm C.5

9: Evaluate $f_{\mathcal{N}_k}(w_t)$ and $g_t$ at the trial point $w_t = w_k + p_k$

10: Obtain $(s_k, y_k) = (w_t - w_k, g_t - g_k)$ and $\rho_k = \dfrac{f_{\mathcal{N}_k}(w_t) - f_{\mathcal{N}_k}(w_k)}{Q(p_k)}$

11: **if** $\rho_k \geq \tau_1$ **then**

12: $w_{k+1} = w_t$

13: **else**

14: $w_{k+1} = w_k$

15: **end if**

16: Update $\delta_k$ by Algorithm C.1

17: **if** $|s^T(y_k - B_k s_k)| \geq \tau \|s_k\| \|y_k - B_k s_k\|$ **then**

18: Update storage matrices $S_{k+1}$ and $Y_{k+1}$ by $l$ recent $\{s_j, y_j\}_{j=k-l+1}^{k}$

19: Compute $\gamma_{k+1}$ for $B_0$ by Algorithm C.4 and $B_{k+1}$ by (4.18)

20: **else**

21: Set $B_{k+1} = B_k$

22: **end if**

23: **end for**

# Appendix D

# Overlap Batching and Computations

In this chapter, we briefly describe the overlap sampling strategy for forming batches of fixed size. By this strategy, the stochastic function and gradient are required to be evaluated with respect to the overlap set of the form batch at each iteration.

**Batch Formation with Overlap Sampling**

Let batches $J_k$ for $k = 0, 1, 2, \ldots$ of the fixed size $N_k = bs$ be drawn without replacement in order to be sure about one actual pass through whole $N$ samples of a dataset within one epoch. Let $J_k = O_{k-1} \cup O_k$ where $O_{k-1}$ and $O_k$ are the overlapping samples of $J_k$ with batches $J_{k-1}$ and $J_{k+1}$, respectively. We assume that $|O_{k-1}| = |O_k| = os$ and thus overlap ratio $or \triangleq \frac{os}{bs} = \frac{1}{2}$ (half overlapping). Let $\bar{N} = \left\lfloor \dfrac{N}{os} \right\rfloor - 1$ indicates the number of batches in one epoch, where $\lfloor a \rfloor$ rounds $a$ to the nearest integer less than or equal to $a$. In order to create $\bar{N}$ mini-batches in one epoch, we have to notice two cases as $rs \triangleq \text{mod}(N, os) = 0$, and $rs \triangleq \text{mod}(N, os) \neq 0$ where the mod (modulo operation) of $N$ and $os$ returns the remainder after division of $N$ and $os$. In the first case, all $\bar{N}$ batches are duplex created by two subsets $O_{k-1}$ and $O_k$ as $J_k = O_{k-1} \cup O_k$. This also holds for the first $\bar{N} - 1$ mini-batches in the second case, but the $\bar{N}$th mini-batch is a triple one as $J_k = O_{k-1} \cup R_k \cup O_k$ where $R_k$ is a subset of size $rs \neq 0$. Considering the aforementioned cases, we assure that a DNN is trained by all samples per epoch.

### Function and Gradient Evaluations

Let $\mathcal{N}_k$ indicate the sample index of $J_k$. Then, the main quantities required in a QN-based algorithm, i.e., $f_{\mathcal{N}_k}(w_k) \triangleq f_k^{J_k}$ and $g_k \triangleq g_k^{J_k}$ (see (3.8) and (3.9)) are determined by

$$f_k^{J_k} = or(f_k^{O_{k-1}} + f_k^{O_k}), \qquad g_k^{J_k} = or(g_k^{O_{k-1}} + g_k^{O_k}),$$

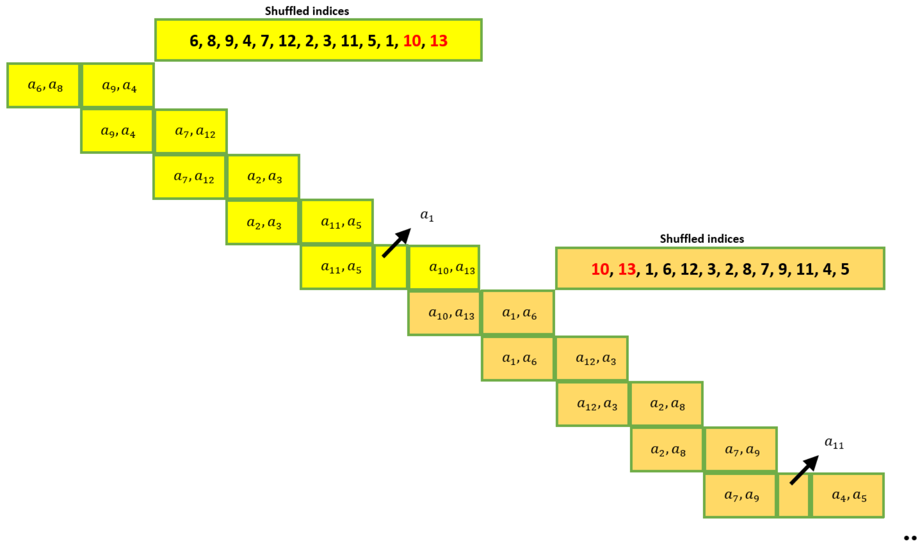
where $or = \frac{1}{2}$. Notice that these quantities with respect to the triple mini-batch in Case 2 are computed as follows

$$f_k^{J_k} = or(f_k^{O_{k-1}} + f_k^{O_k}) + (1-2or)f_k^{R_k}, \qquad g_k^{J_k} \quad = or(g_k^{O_{k-1}} + g_k^{O_k}) + (1-2or)g_k^{R_k}.$$

This figure illustrates an example of Case 2, where $\mathcal{N} = \{1, 2, \ldots, 13\}$, $bs = 4$, and $os = 2$. For this example obviously, we can have 5 mini-batches per epoch. The last mini-batch of each epoch allocates one subset to the remaining set $R_k$ of size $rs$, which in this example $rs = 1$. In overlapping batch formation, we notice the mini-batches when the epoch changes; where the subset of the last mini-batch in the current epoch includes the same samples as the first mini-batch in the new epoch; notice samples $a_10$ and $a_13$.

On https://github.com/MATHinDL/sL_QN_TR/, we offer readers MATLAB codes for implementing the overlapping strategy described in this section.

**Figure D.1** An example of the overlapping batch formation within 2 epochs.



# Appendix E

# Additional Experiments

More numerical results than those in Chapter 3 are provided below. Table E.1 gives straight addresses to the figures reporting training and testing accuracy as well as training and testing overall loss with more batch sizes. Moreover, additional observations on CPU training time, and comparisons between *tuned* Adam and the sL-QN-TR all with hyper-parameters used in Chapter 3 are presented.

| Comparison of the sL-LBFGS-TR and sL-SR1-TR algorithms vs epoch, ($l = 20$) | | | | | |
|---|---|---|---|---|---|
| | LeNet-like | ResNet-20 | ResNet-20(no BN) | ConvNet3FC2 | ConvNet3FC2(no BN) |
| MNIST | Figure E.1 | — | — | — | — |
| Fashion-MNIST | Figure E.2 | Figure E.3 | Figure E.5 | Figure E.11 | Figure E.8 |
| CIFAR10 | — | Figure E.4 | Figure E.6 | Figure E.9 | Figure E.12 |
| Comparison of the sL-LBFGS-TR and sL-SR1-TR algorithms vs CPU training time, ($l = 20$) | | | | | |
| MNIST | Figure E.13 | — | — | — | — |
| Fashion-MNIST | Figure E.13 | — | — | — | — |
| CIFAR10 | — | — | — | Figure E.14 | Figure E.14 |
| Comparison of the sL-LBFGS-TR and sL-SR1-TR algorithms with *tuned* Adam, ($l = 20$) | | | | | |
| MNIST | Figure E.15 | — | — | — | — |
| Fashion-MNIST | — | Figure E.16 | Figure E.17 | — | — |
| CIFAR10 | — | — | — | Figure E.18 | Figure E.19 |

Table E.1: Set of figures for image classification problems.

**Figure E.1** `MNIST`, `LeNet-like`: The accuracy and loss evolution vs epoch.

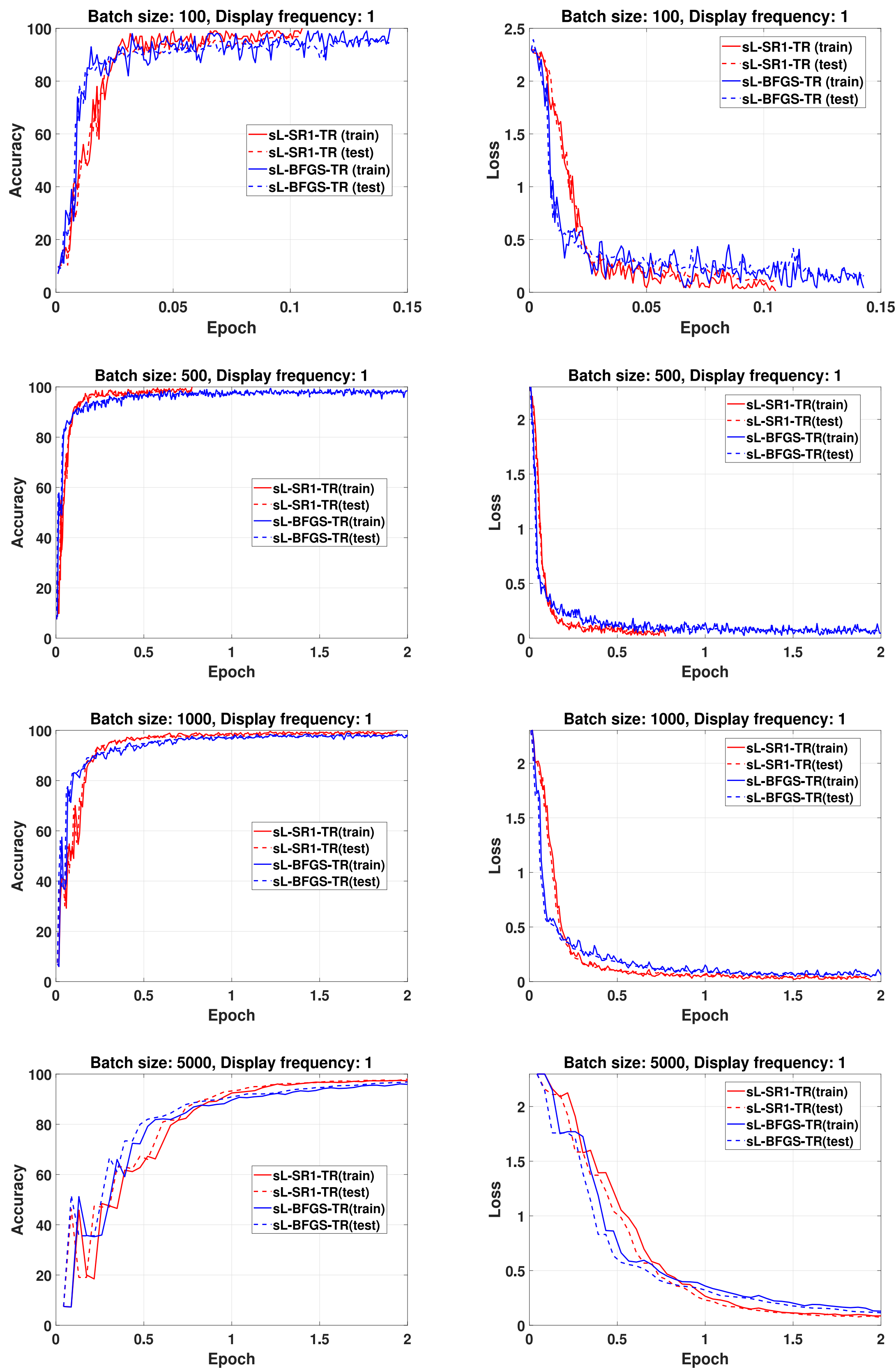


**Figure E.2** `F-MNIST`, `LeNet-like`: The accuracy and loss evolution vs epoch.

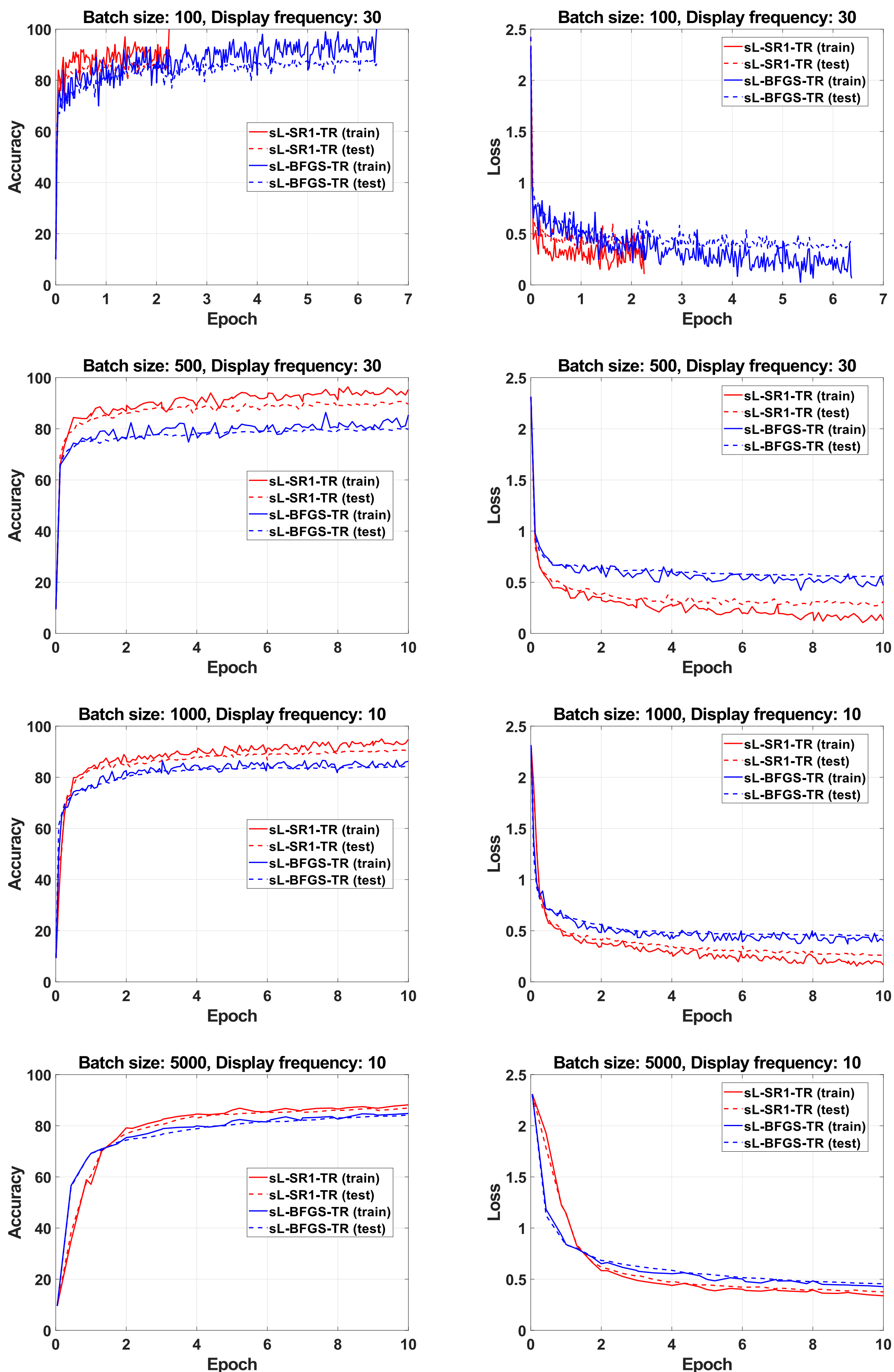


**Figure E.3** `F-MNIST`, `ResNet-20`: The accuracy and loss evolution vs epoch.

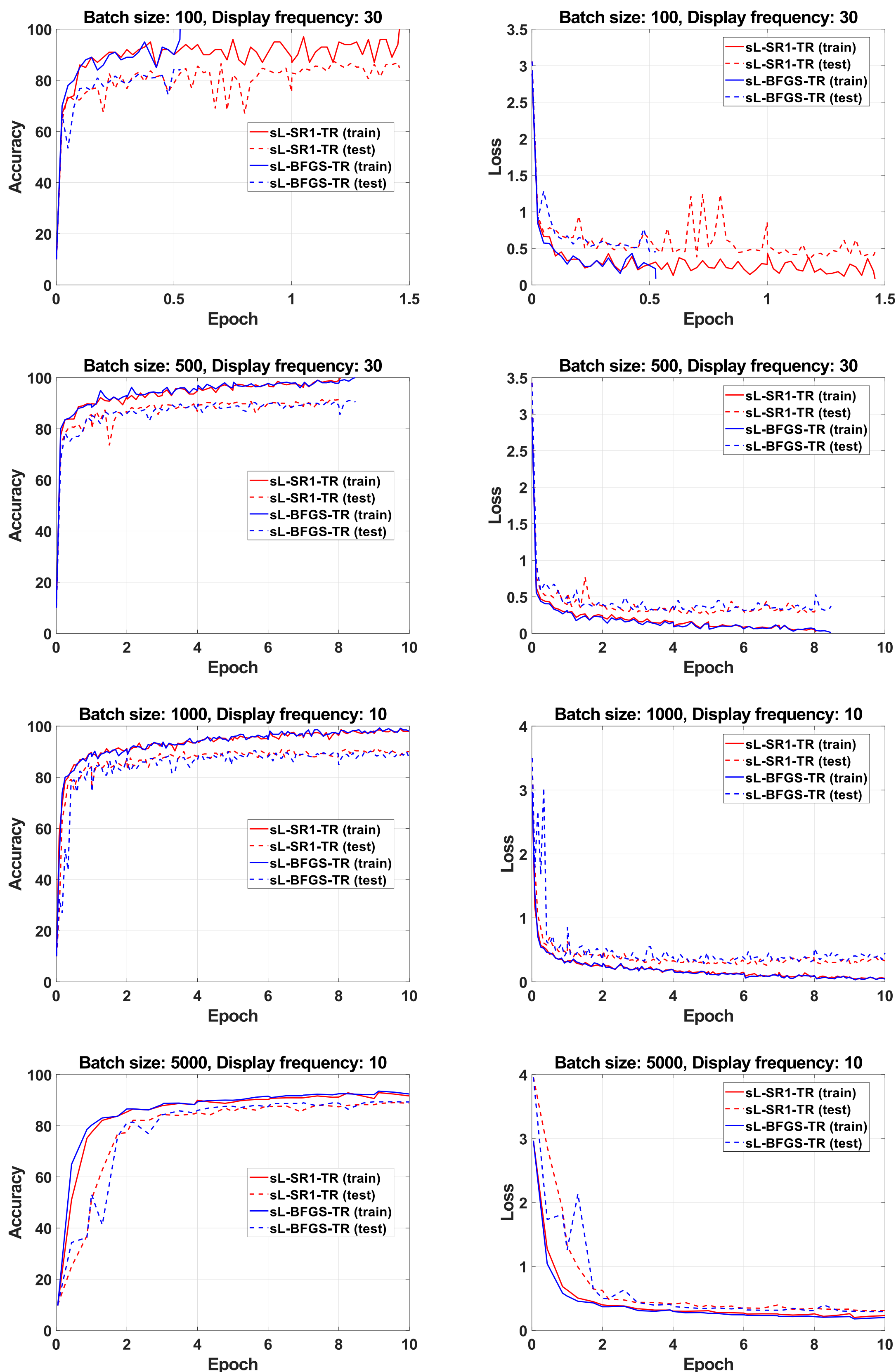


**Figure E.4** `CIFAR10`, `ResNet-20`: The accuracy and loss evolution vs epoch.

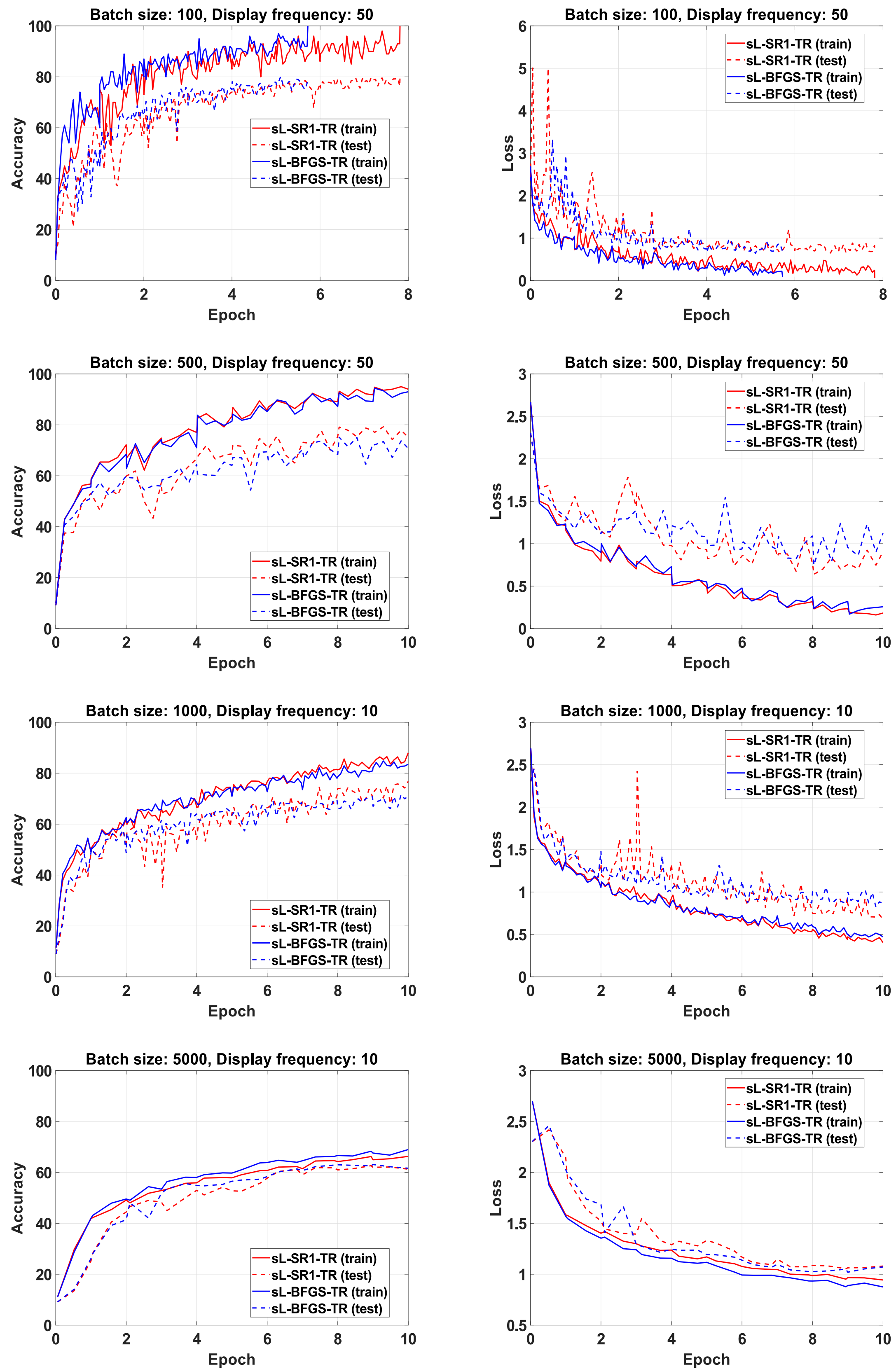


**Figure E.5** `F-MNIST`, `ResNet-20(no BN)`: The accuracy and loss evolution vs epoch.

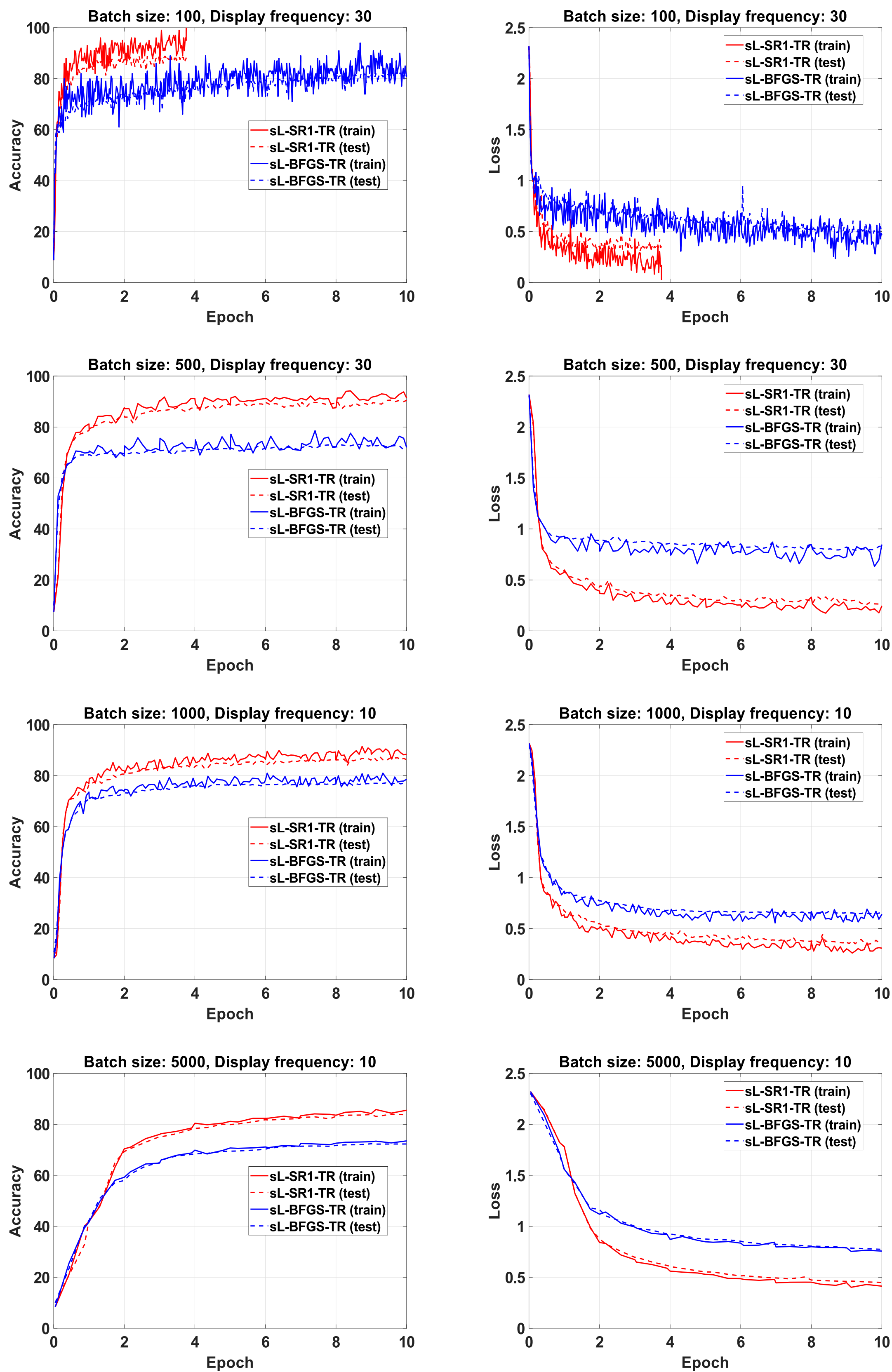


**Figure E.6** `CIFAR10`, `ResNet-20(no BN)`: The accuracy and loss evolution vs epoch.

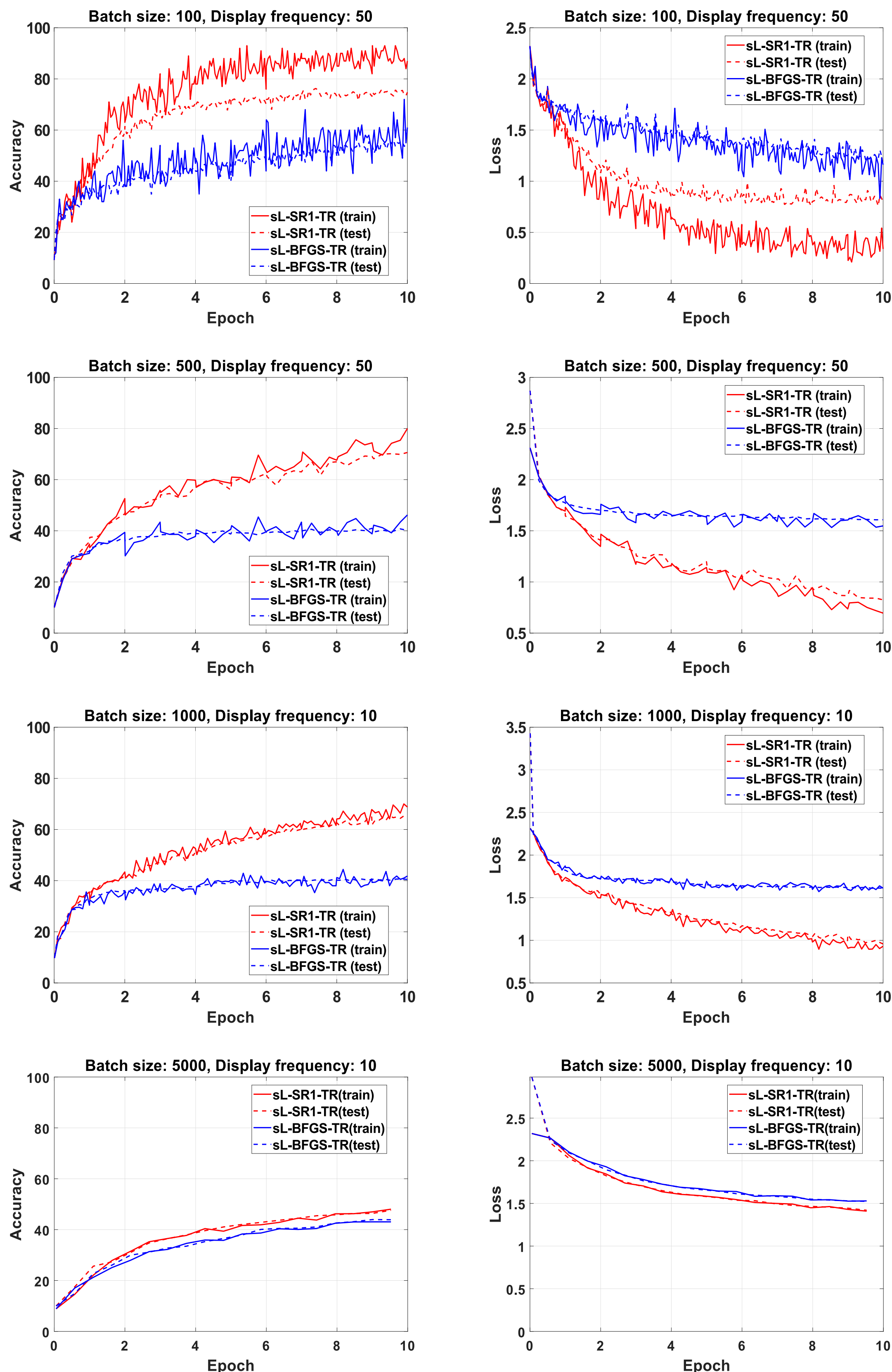


**Figure E.7** `MNIST`, `ConvNet3FC2`: The accuracy and loss evolution vs epoch.

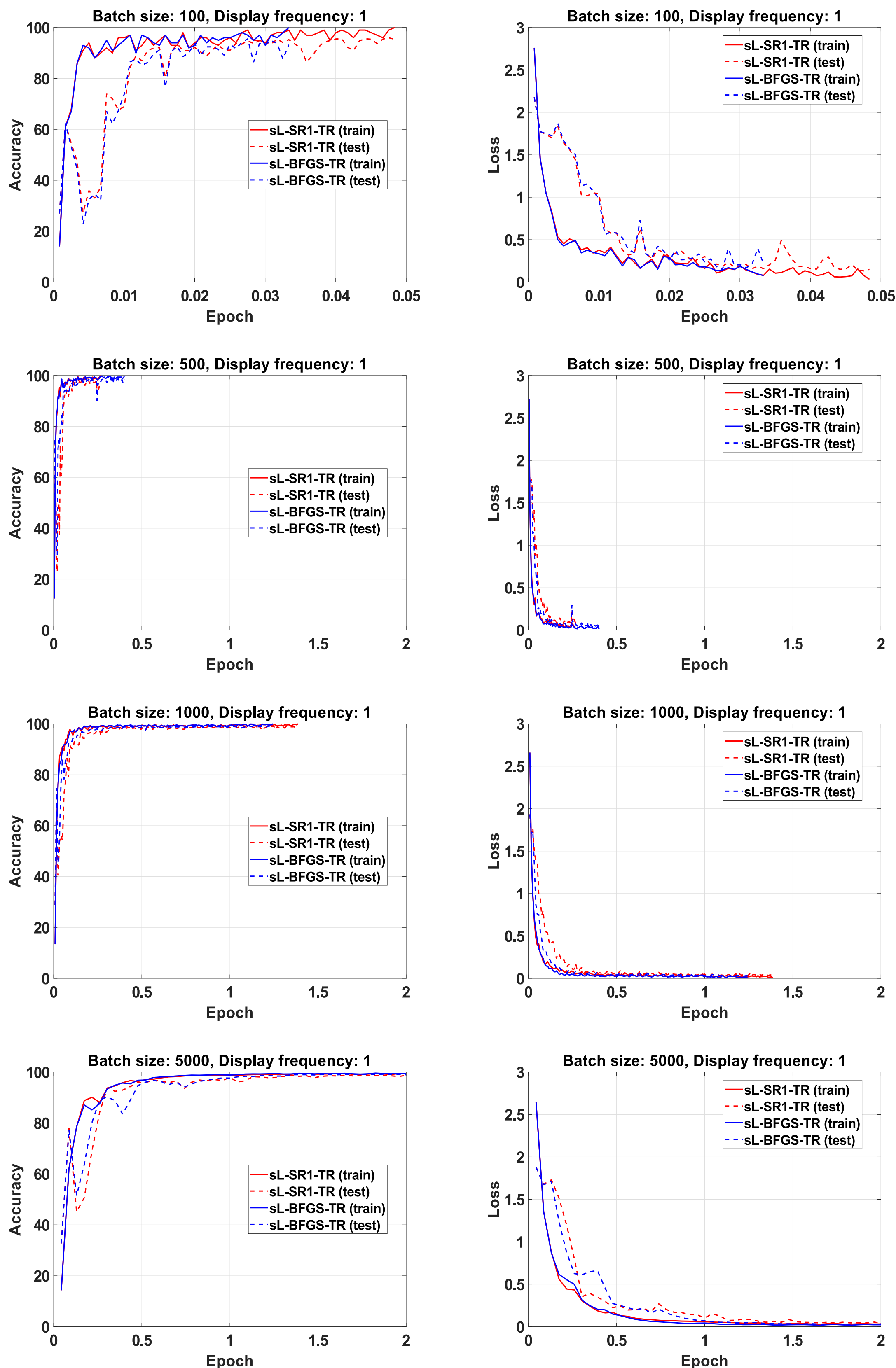


**Figure E.8** F-MNIST, ConvNet3FC2: The accuracy and loss evolution vs epoch.

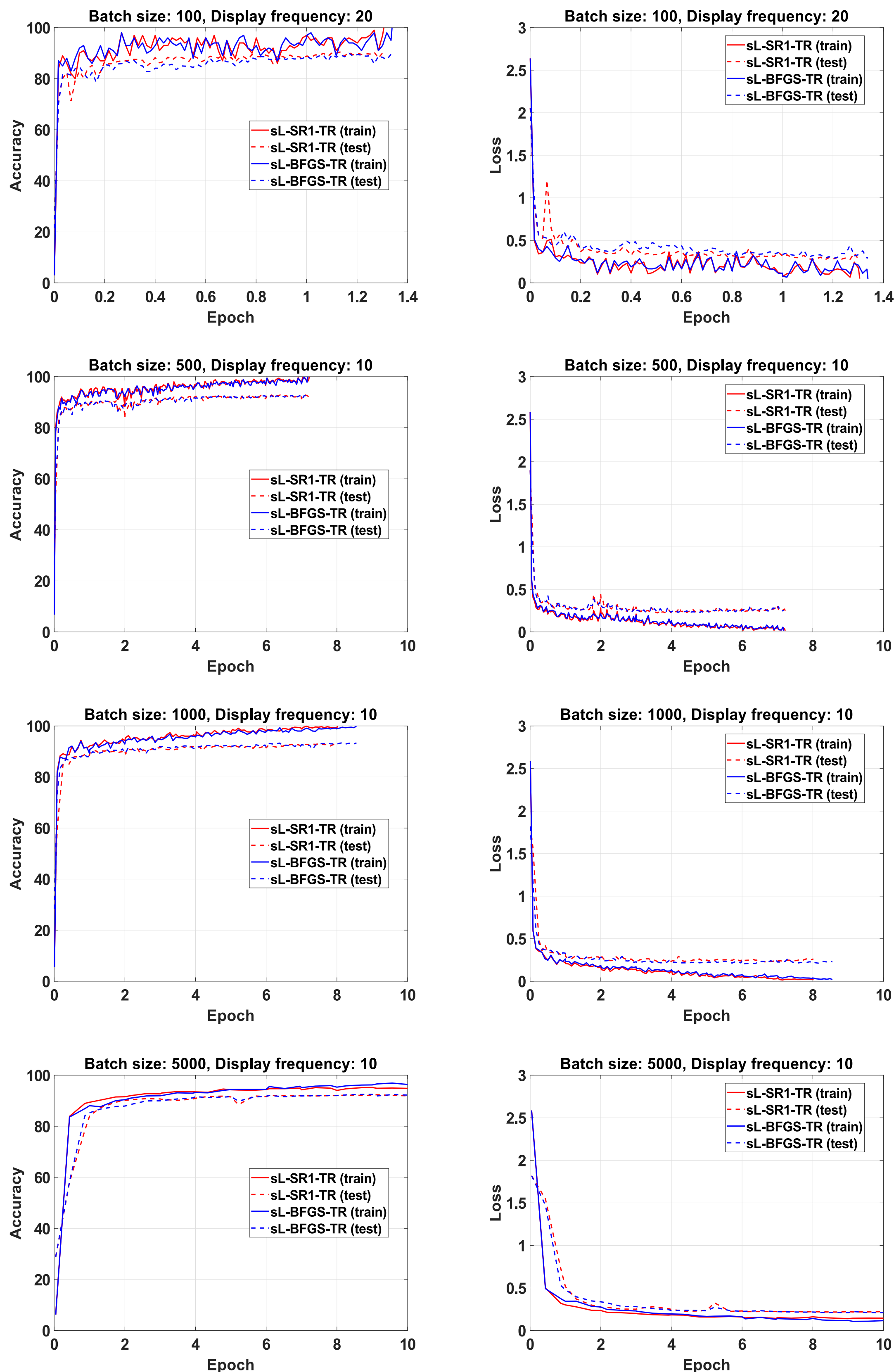


**Figure E.9** `CIFAR10`, `ConvNet3FC2`: The accuracy and loss evolution vs epoch.

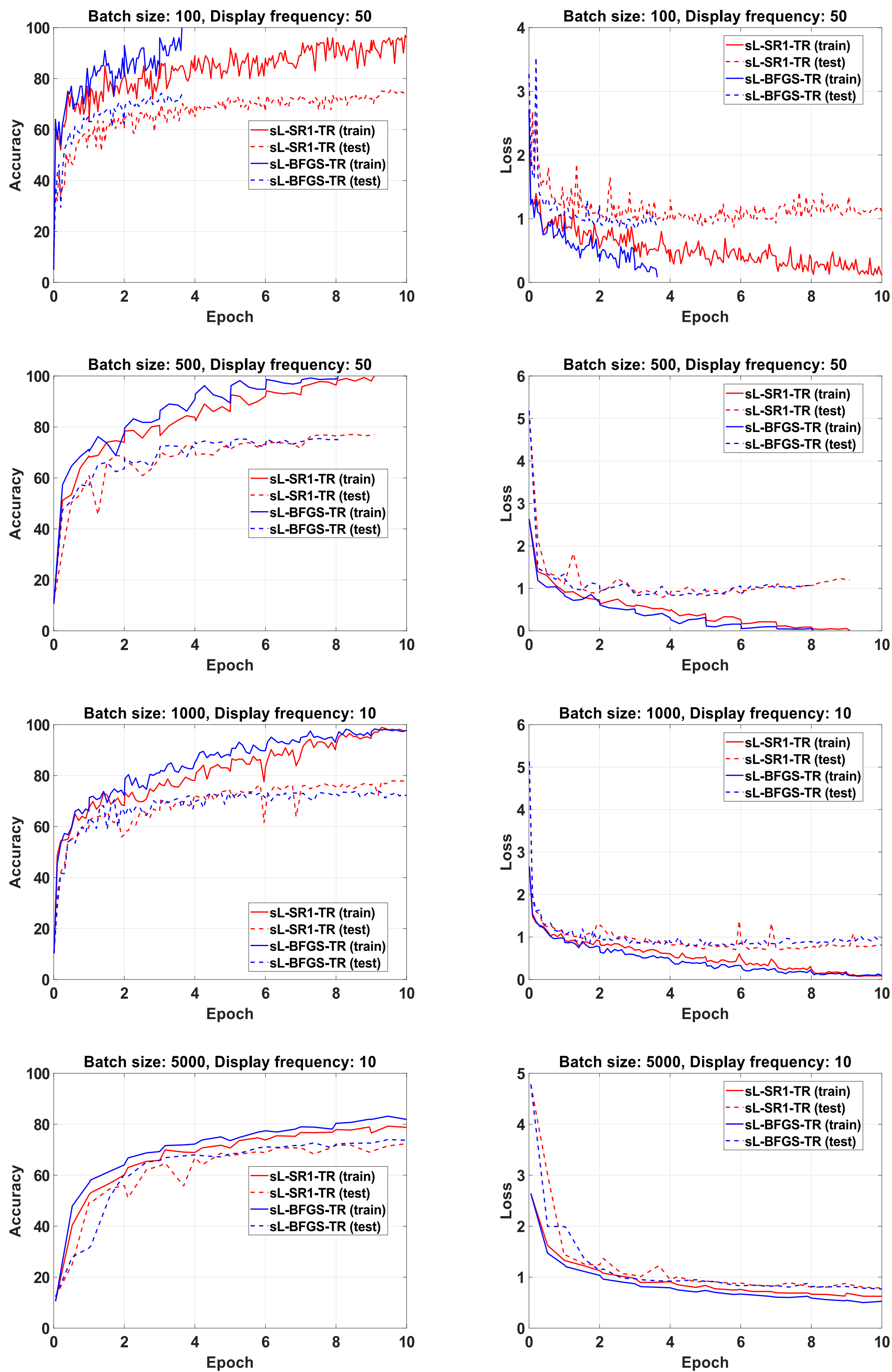


**Figure E.10** `MNIST`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch.

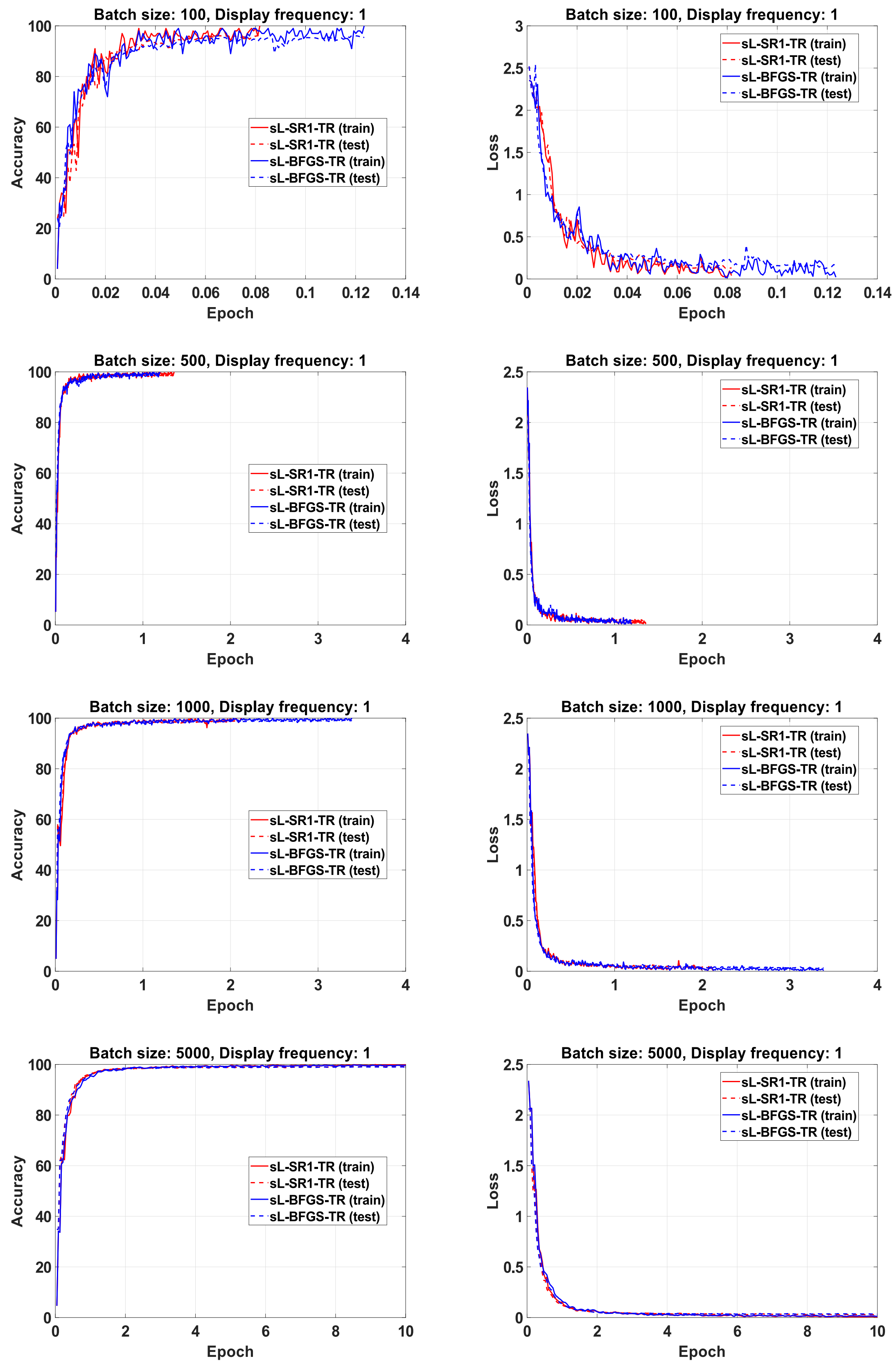


**Figure E.11** `F-MNIST`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch.

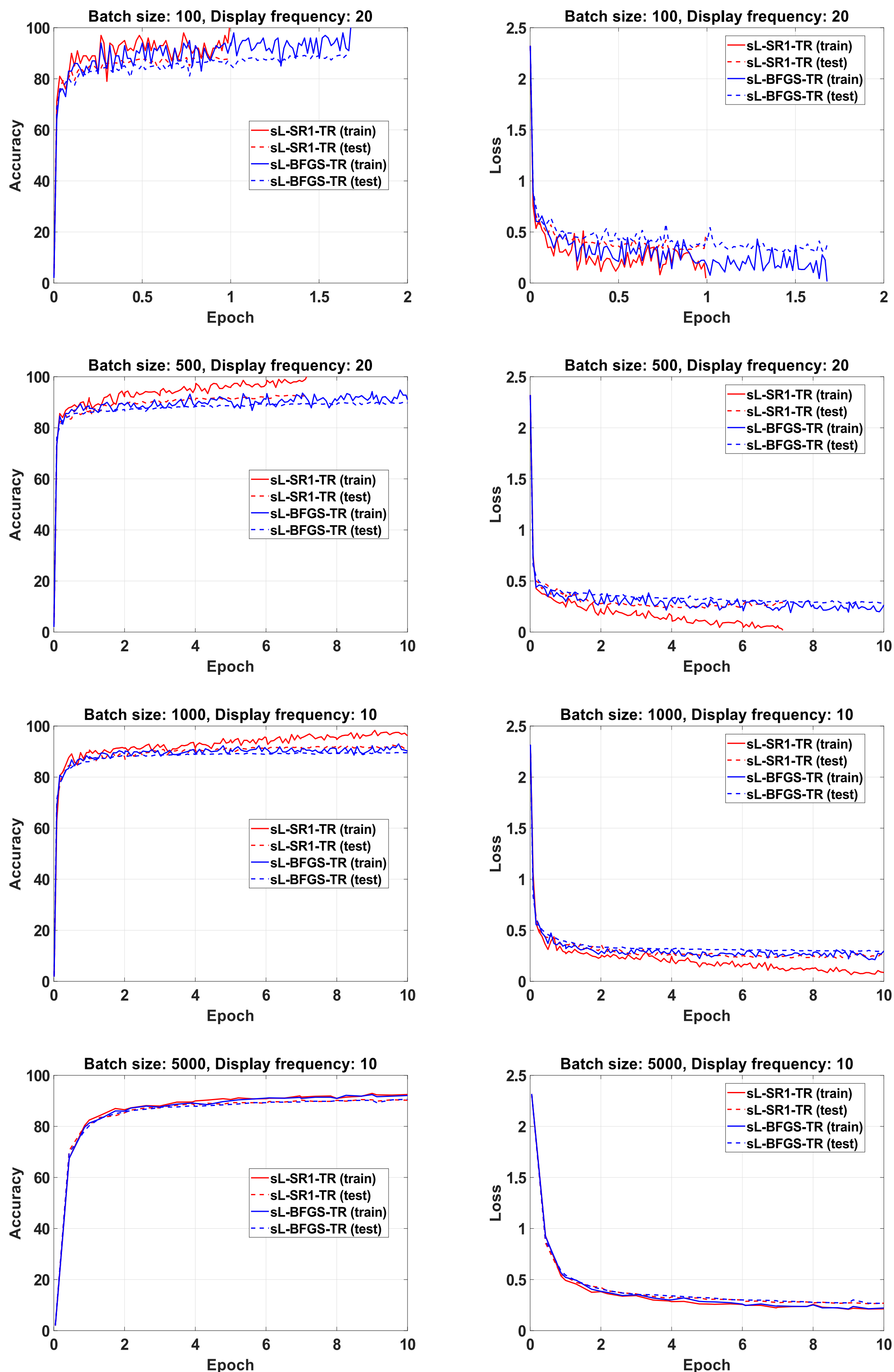


**Figure E.12** `CIFAR10`, `ConvNet3FC2(no BN)`: The accuracy and loss evolution vs epoch.

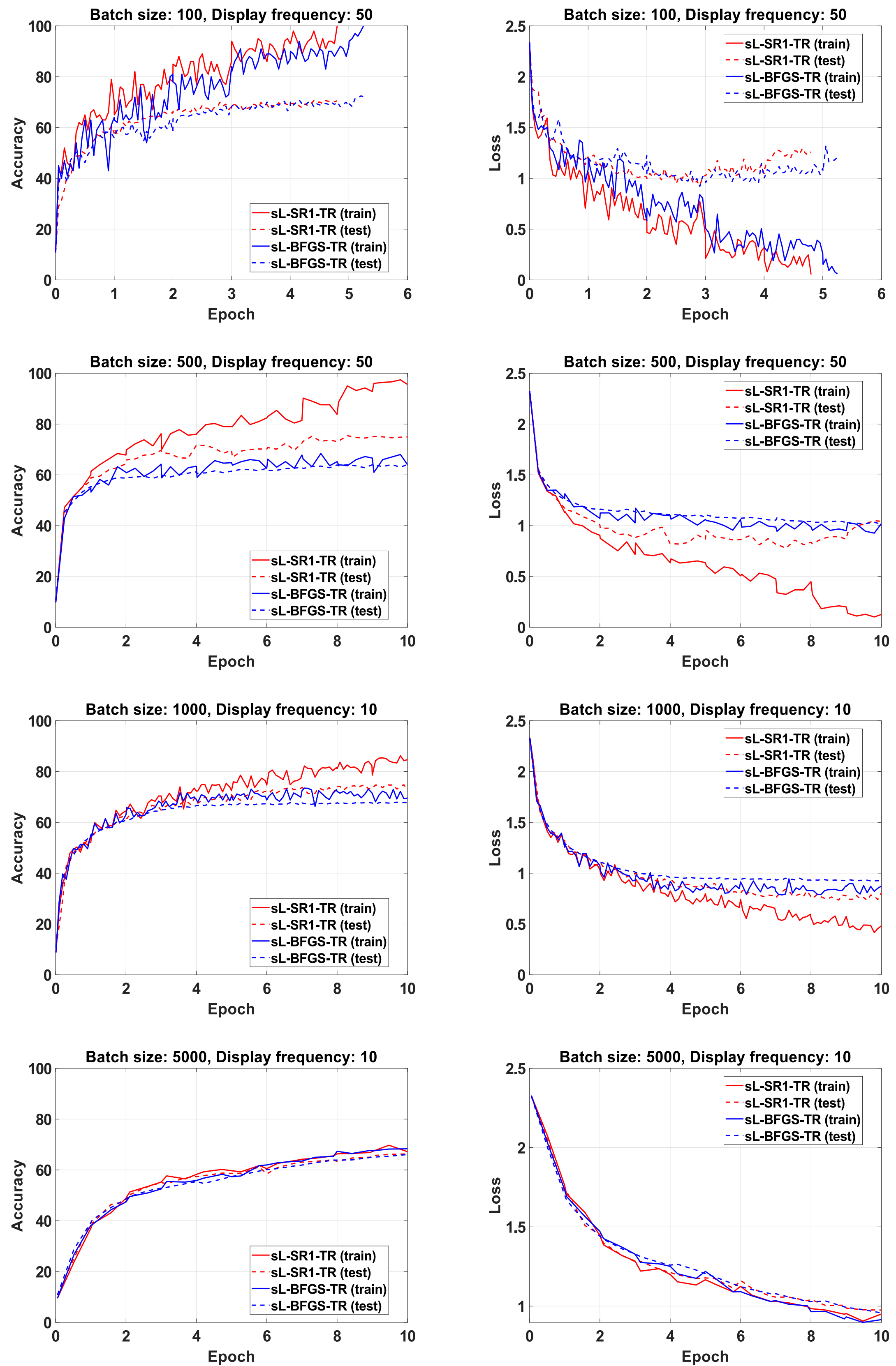
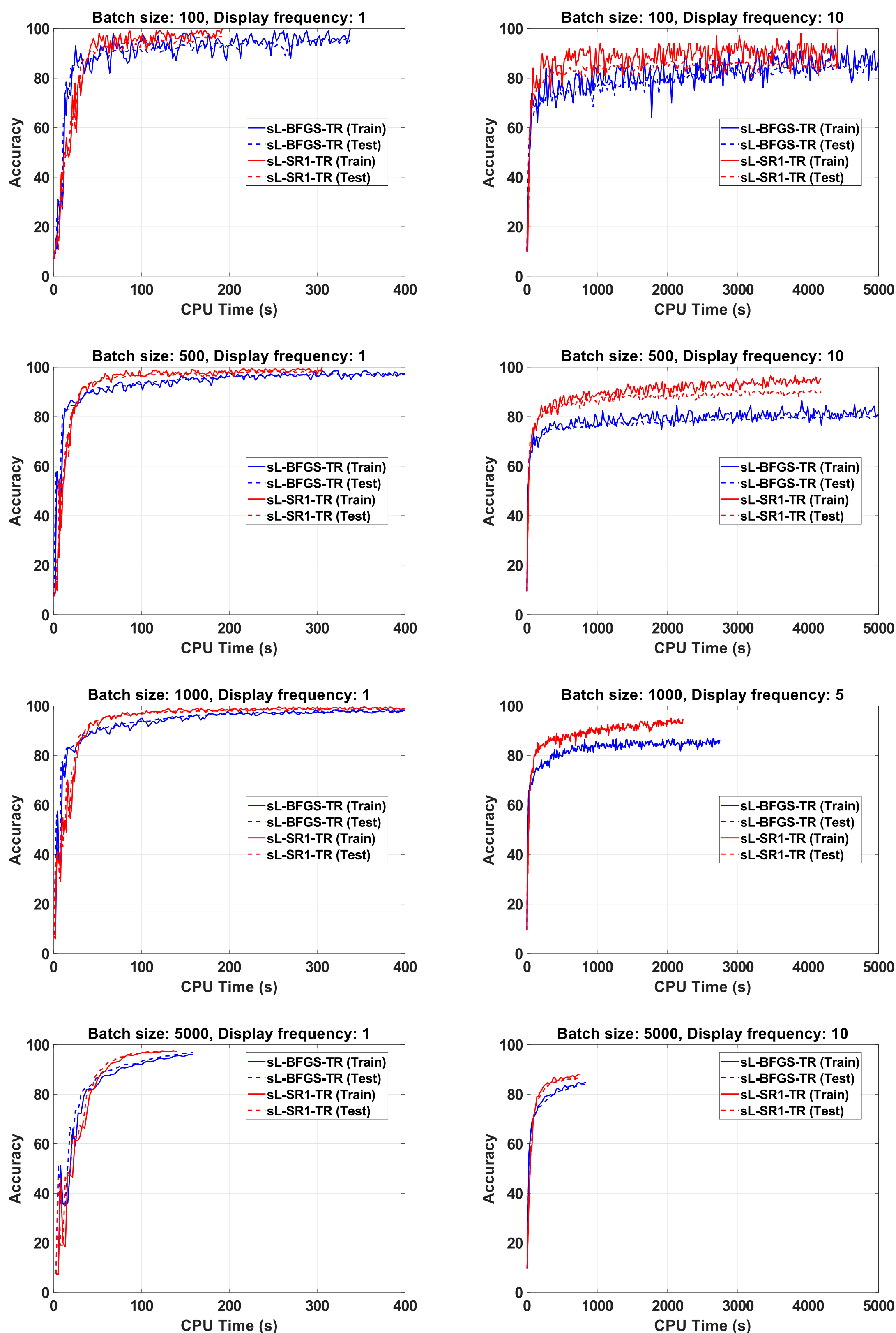
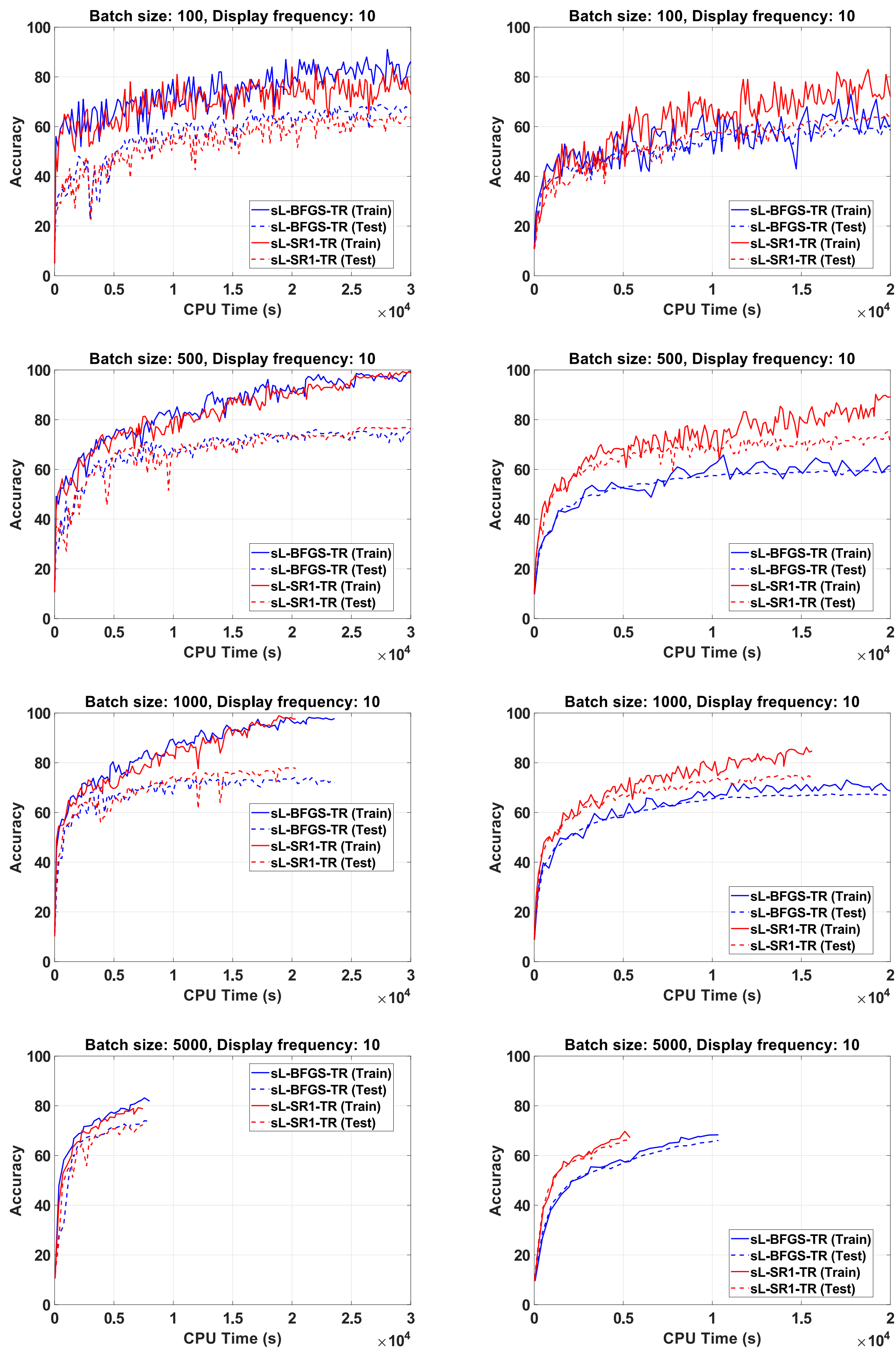


**Figure E.13** `MNIST` and `F-MNIST`: The accuracy evolution vs CPU time.




*(a) MNIST, LeNet-like* *(b) Fashion-MNIST, LeNet-like*

**Figure E.14** `CIFAR10`: The accuracy evolution vs CPU time.




(a) *CIFAR10, ConvNet3FC2* (b) *CIFAR10, ConvNet3FC2(no BN)*

**Figure E.15** MNIST, LeNet-like: Comparison with *tuned* Adam.

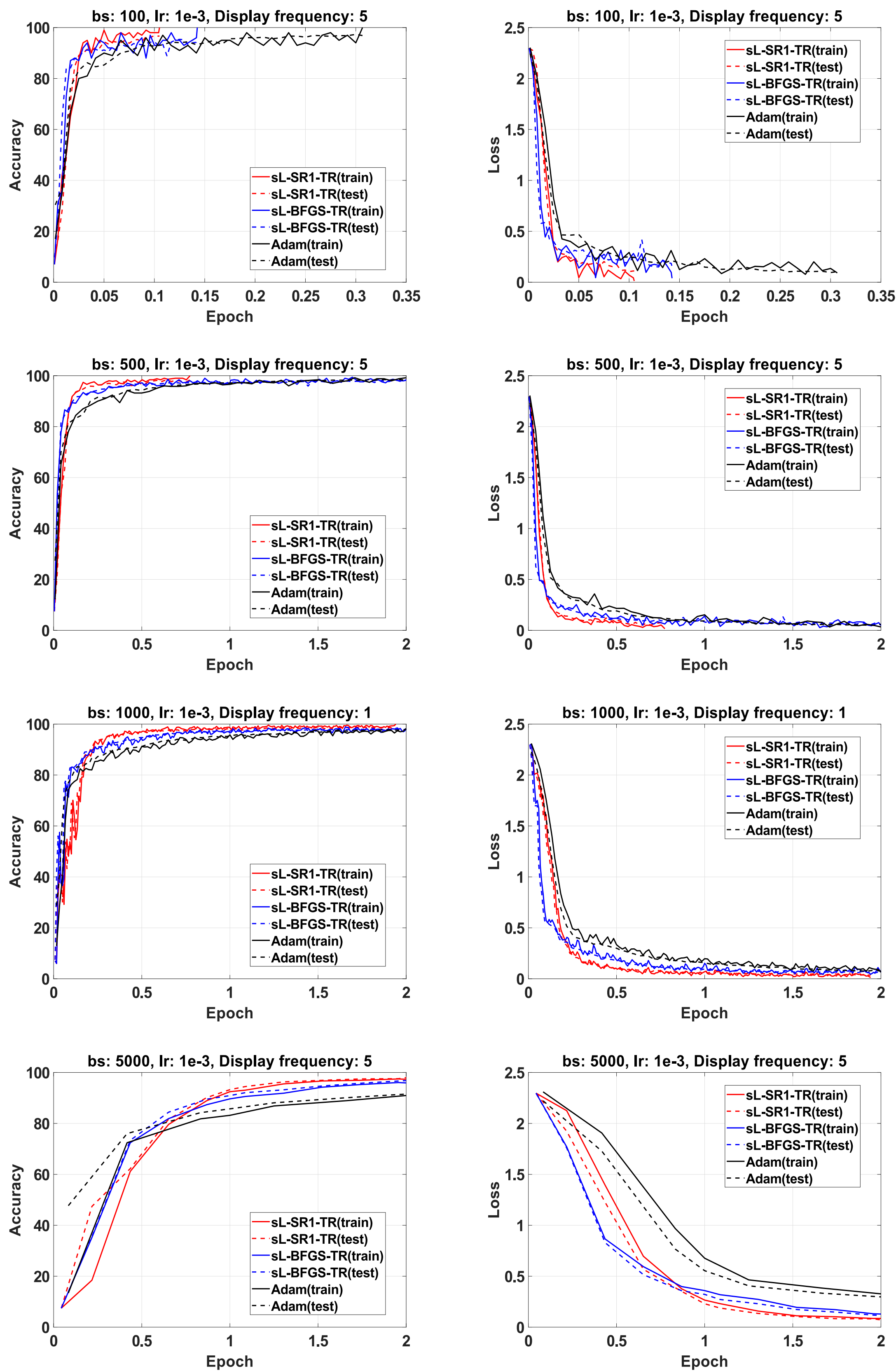


**Figure E.16** `F-MNIST`, `ResNet-20`: Comparison with *tuned* Adam.

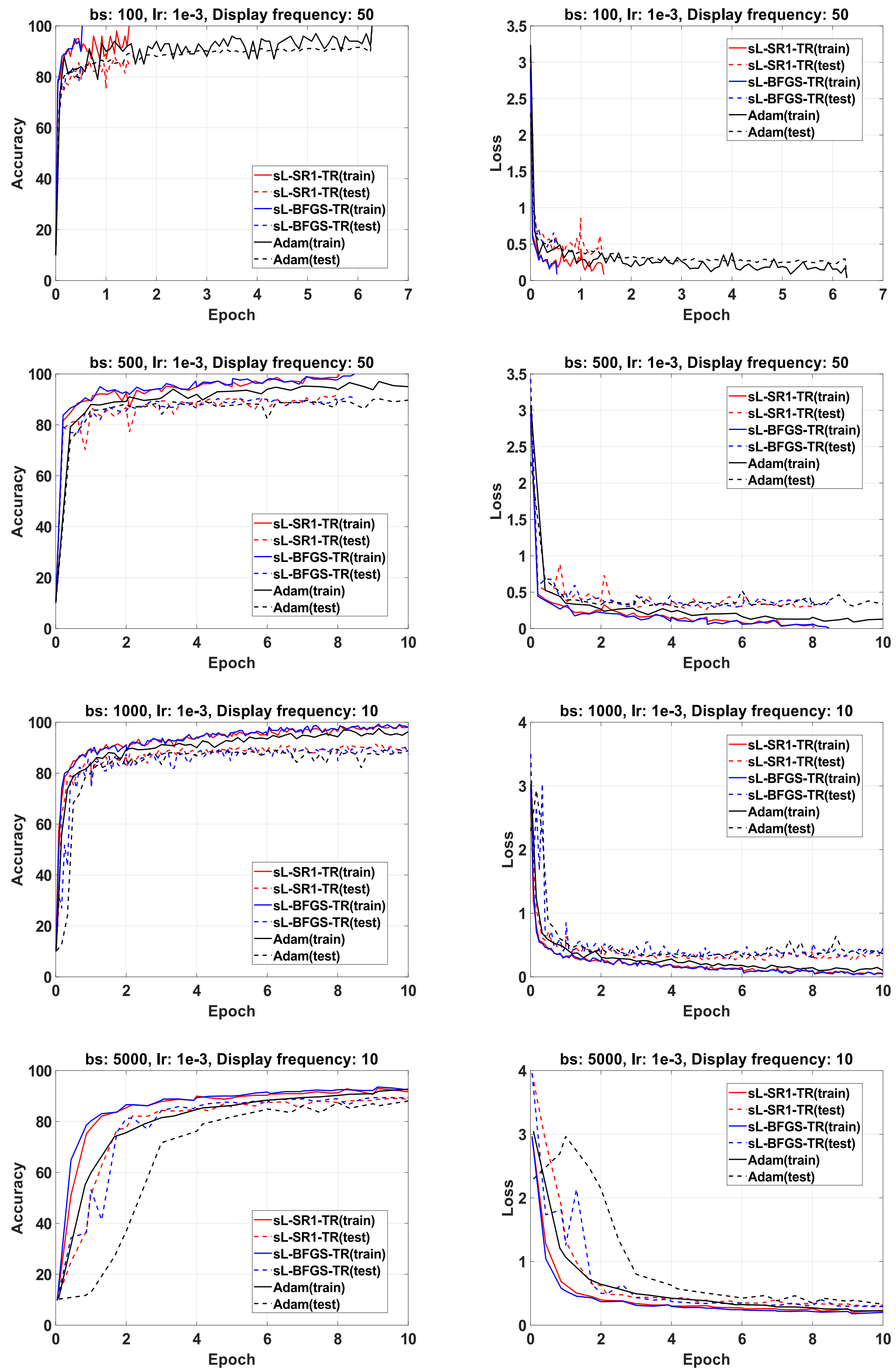


**Figure E.17** F-MNIST, ResNet-20(no BN): ResNet-20: Comparison with *tuned* Adam.

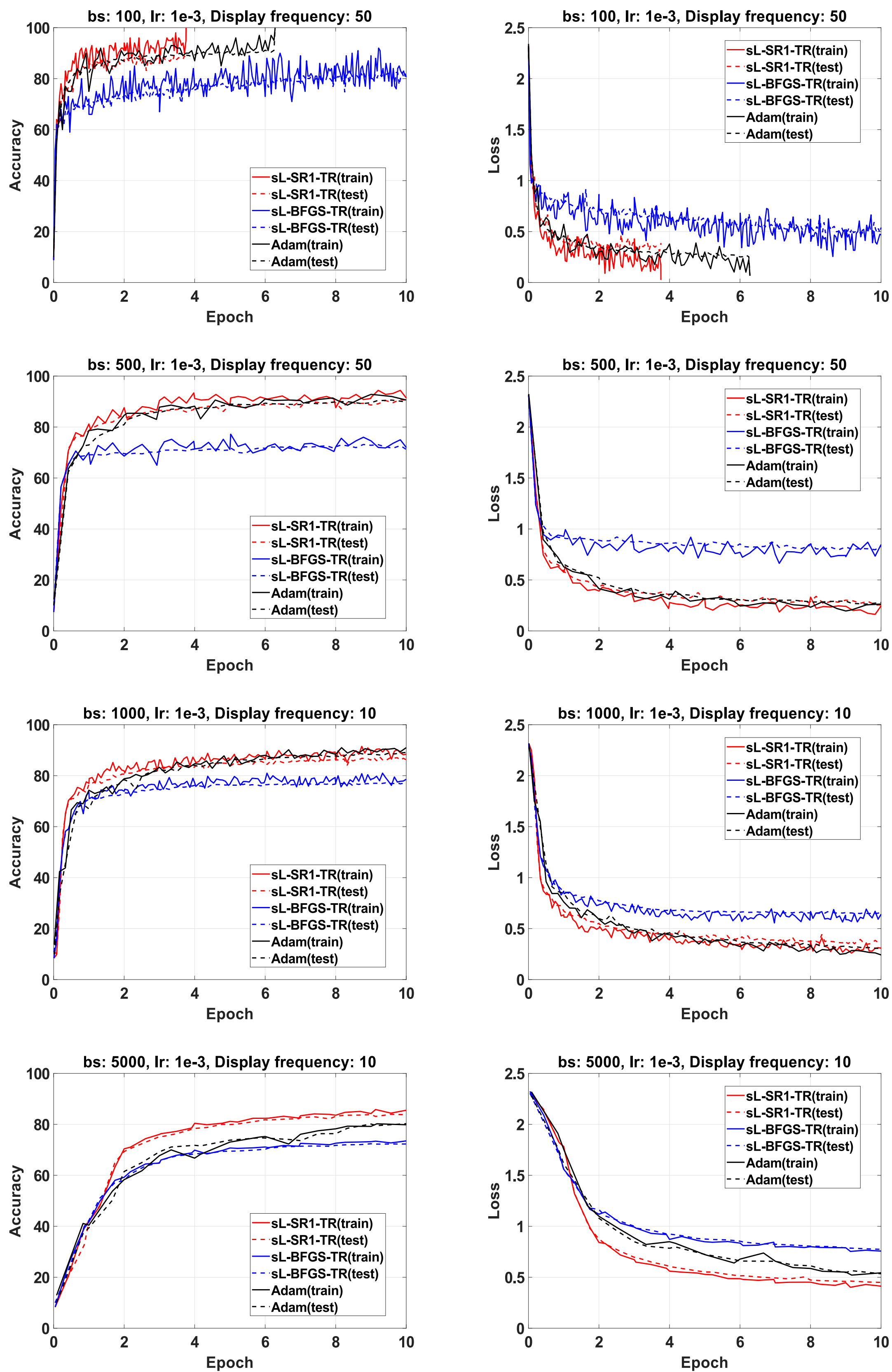


**Figure E.18** `CIFAR10`, `ConvNet3FC2`: Comparison with *tuned* Adam.

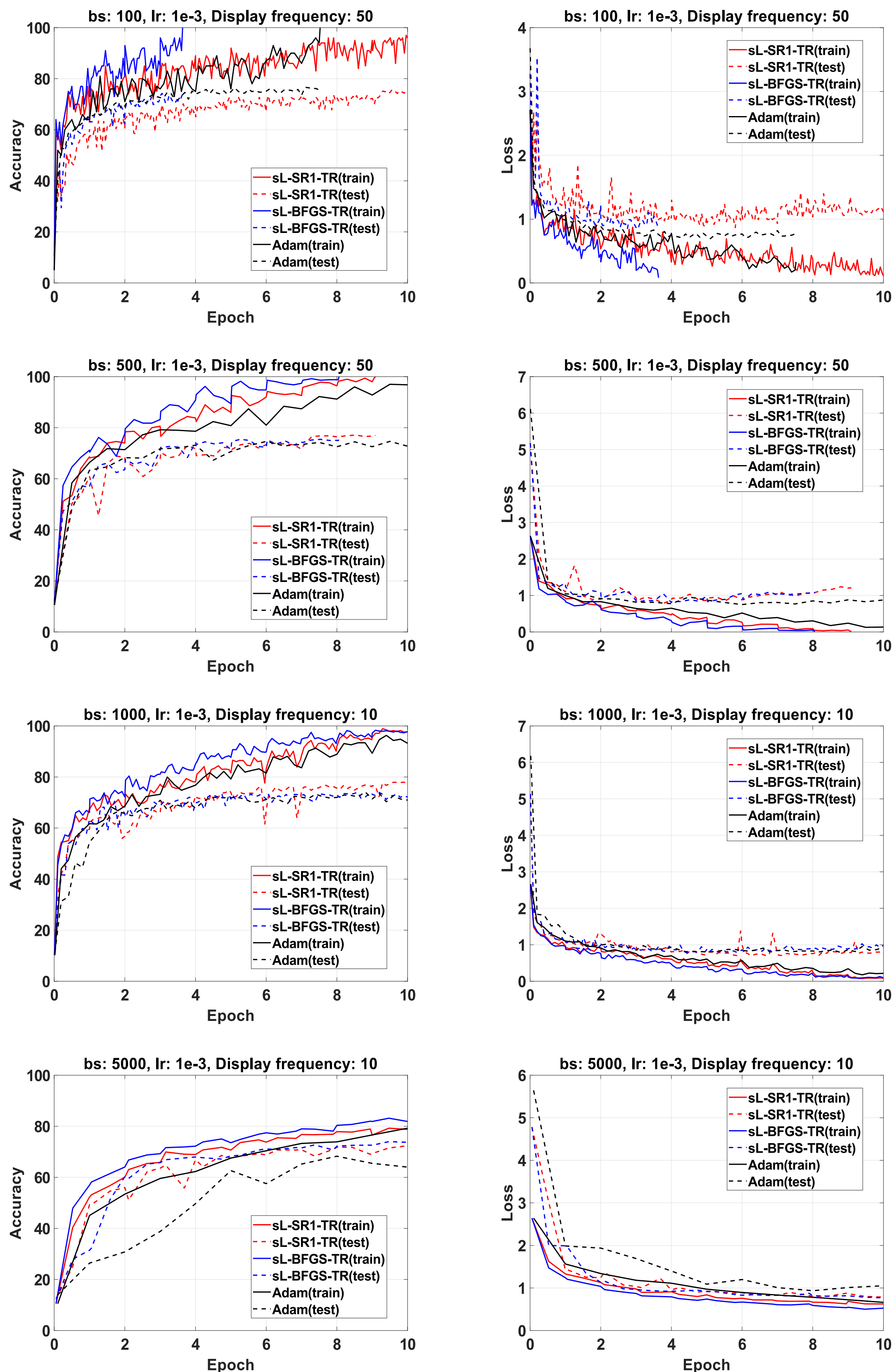
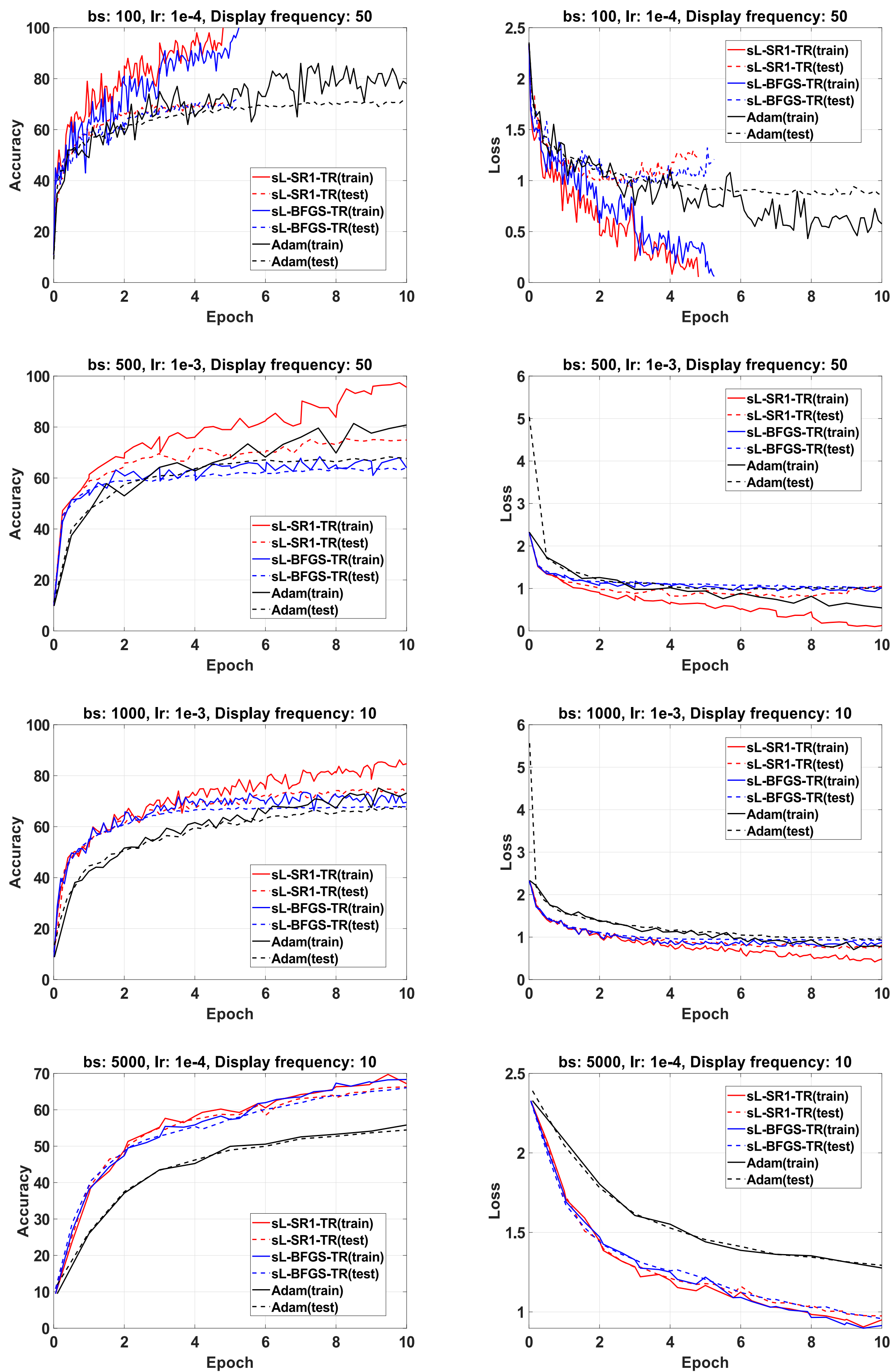


**Figure E.19** `CIFAR10`, `ConvNet3FC2(no BN)`: Comparison with *tuned* Adam.



# Bibliography


[1] Lasith Adhikari, Omar DeGuchy, Jennifer B Erway, Shelby Lockhart, and Roummel F Marcia. Limited-memory trust-region methods for sparse relaxation. In *Wavelets and Sparsity XVII*, volume 10394. International Society for Optical Engineering, 2017.

[2] Masoud Ahookhosh, Keyvan Amini, and Mohammad Reza Peyghami. A non-monotone trust-region line search method for large-scale unconstrained optimization. *Applied Mathematical Modelling*, 36(1):478–487, 2012.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[4] Stefania Bellavia, Nataša Krejić, Benedetta Morini, and Simone Rebegoldi. A stochastic first-order trust-region method with inexact restoration for finite-sum minimization. *Computational Optimization and Applications*, 84(1):53–84, 2023.

[5] Albert S Berahas, Majid Jahani, Peter Richtárik, and Martin Takáč. Quasi-Newton methods for machine learning: forget the past, just sample. *Optimization Methods and Software*, 37(5):1668–1704, 2022.

[6] Albert S Berahas, Jorge Nocedal, and Martin Takáč. A multi-batch L-BFGS method for machine learning. *In Advances in Neural Information Processing Systems*, pages 1055–1063, 2016.

[7] Albert S Berahas and Martin Takáč. A robust multi-batch L-BFGS method for machine learning. *Optimization Methods and Software*, 35(1):191–219, 2020.

[8] Dimitri P Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.

[9] Jose Blanchet, Coralia Cartis, Matt Menickelly, and Katya Scheinberg. Convergence rate analysis of a stochastic trust-region method via supermartingales. *INFORMS journal on optimization*, 1(2):92–119, 2019.

[10] Raghu Bollapragada, Richard H Byrd, and Jorge Nocedal. Exact and inexact subsampled newton methods for optimization. *IMA Journal of Numerical Analysis*, 39(2):545–578, 2019.

[11] Raghu Bollapragada, Jorge Nocedal, Dheevatsa Mudigere, Hao-Jun Shi, and Ping Tak Peter Tang. A progressive batching L-BFGS method for machine learning. In *International Conference on Machine Learning*, pages 620–629. PMLR, 2018.

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

[13] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[14] Léon Bottou and Yann LeCun. Large-scale online learning. *Advances in neural information processing systems*, 16:217–224, 2004.

[15] Johannes Brust, Jennifer B Erway, and Roummel F Marcia. On solving L-SR1 trust-region subproblems. *Computational Optimization and Applications*, 66(2):245–266, 2017.

[16] Richard H Byrd, Samantha L Hansen, Jorge Nocedal, and Yoram Singer. A stochastic Quasi-Newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[17] Ruobing Chen, Matt Menickelly, and Katya Scheinberg. Stochastic optimization using a trust-region method and random models. *Mathematical Programming*, 169(2):447–487, 2018.

[18] Andrew R Conn, Nicholas IM Gould, and Philippe L Toint. *trust-region methods*. SIAM, 2000.

[19] Zhaocheng Cui, Boying Wu, and Shaojian Qu. Combining non-monotone conic trust-region and line search techniques for unconstrained optimization. *Journal of computational and applied mathematics*, 235(8):2432–2441, 2011.

[20] Frank E Curtis and Katya Scheinberg. Optimization methods for supervised machine learning: From linear models to deep learning. In *Leading Developments from INFORMS Communities*, pages 89–114. INFORMS, 2017.

[21] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in neural information processing systems*, pages 1646–1654, 2014.

[22] NY Deng, Yi Xiao, and FJ Zhou. Nonmonotonic trust-region algorithm. *Journal of optimization theory and applications*, 76(2):259–285, 1993.

[23] Daniela Di Serafino, Nataša Krejić, Nataša Krklec Jerinkić, and Marco Viola. Lsos: Line-search second-order stochastic optimization methods for nonconvex finite sums. *Mathematics of Computation*, 92(341):1273–1299, 2023.

[24] Daniela di Serafino, Gerardo Toraldo, and Marco Viola. Using gradient directions to get global convergence of newton-type methods. *Applied Mathematics and Computation*, 409:125612, 2021.

[25] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[26] Darina Dvinskikh. Stochastic approximation versus sample average approximation for wasserstein barycenters. *Optimization Methods and Software*, 37(5):1603–1635, 2022.

[27] Jennifer B Erway, Joshua Griffin, Roummel F Marcia, and Riadh Omheni. Trust-region algorithms for training responses: machine learning methods using indefinite Hessian approximations. *Optimization Methods and Software*, 35(3):460–487, 2020.

[28] David M Gay. Computing optimal locally constrained steps. *SIAM Journal on Scientific and Statistical Computing*, 2(2):186–197, 1981.

[29] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[30] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical Quasi-Newton methods for training deep neural networks. *Advances in Neural Information Processing Systems*, 33:2386–2396, 2020.

[31] Gene H Golub and Charles F Van Loan. *Matrix computations, 4th Edition*. Johns Hopkins University press, 2013.

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[33] Robert Gower, Donald Goldfarb, and Peter Richtárik. Stochastic block BFGS: Squeezing more curvature out of data. In *International Conference on Machine Learning*, pages 1869–1878. PMLR, 2016.

[34] Robert M Gower, Peter Richtárik, and Francis Bach. Stochastic quasi-gradient methods: Variance reduction via jacobian sketching. *Mathematical Programming*, 188:135–192, 2021.

[35] Luigi Grippo, Francesco Lampariello, and Stephano Lucidi. A non-monotone line search technique for newton's method. *SIAM Journal on Numerical Analysis*, 23(4):707–716, 1986.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] Judy Hoffman, Daniel A Roberts, and Sho Yaida. Robust learning with jacobian regularization. *arXiv preprint arXiv:1908.02729*, 2019.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[39] Alfredo N Iusem, Alejandro Jofré, Roberto I Oliveira, and Philip Thompson. Variance-based extragradient methods with line search for stochastic variational inequalities. *SIAM Journal on Optimization*, 29(1):175–206, 2019.

[40] Majid Jahani, Mohammadreza Nazari, Sergey Rusakov, Albert S Berahas, and Martin Takáč. Scaling up Quasi-Newton algorithms: Communication efficient distributed SR1. In *Machine Learning, Optimization, and Data Science: 6th International Conference, LOD 2020, Siena, Italy, July 19–23, 2020, Revised Selected Papers, Part I 6*, pages 41–54. Springer, 2020.

[41] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems*, 26:315–323, 2013.

[42] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[43] Nataša Krejić and Nataša Krklec Jerinkić. Non-monotone line search methods with variable sample size. *Numerical Algorithms*, 68(4):711–739, 2015.

[44] Nataša Krejić, Zorana Lužanin, Zoran Ovcin, and Irena Stojkovska. Descent direction method with line search for unconstrained optimization in noisy environment. *Optimization Methods and Software*, 30(6):1164–1184, 2015.

[45] Nataša Krejić, Nataša Krklec Jerinkić, Ángeles Martínez, and Mahsa Yousefi. A non-monotone extra-gradient trust-region method with noisy oracles. *arXiv preprint arXiv:*, 2023.

[46] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. *Available at:* https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

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

[48] Yann LeCun. The MNIST database of handwritten digits. *Available at:* http://yann.lecun.com/exdb/mnist/, 1998.

[49] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[50] Aurelien Lucchi, Brian McWilliams, and Thomas Hofmann. A variance reduced stochastic newton method. *arXiv preprint arXiv:1503.08316*, 2015.

[51] J. Martens and I. Sutskever. Training deep and recurrent networks with Hessian-Free optimization. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer, 2012.

[52] James Martens. New insights and perspectives on the natural gradient method. *The Journal of Machine Learning Research*, 21(1):5776–5851, 2020.

[53] James Martens et al. Deep learning via Hessian-Free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[54] James Martens and Roger Grosse. Optimizing neural networks with Kronecker-Factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[55] Farzin Modarres, Abu Hassan Malik, and Wah June Leong. Improved Hessian approximation with modified secant equations for symmetric rank-one method. *Journal of computational and applied mathematics*, 235(8):2423–2431, 2011.

[56] Aryan Mokhtari and Alejandro Ribeiro. Res: Regularized stochastic BFGS algorithm. *IEEE Transactions on Signal Processing*, 62(23):6089–6104, 2014.

[57] Aryan Mokhtari and Alejandro Ribeiro. Global convergence of online limited memory BFGS. *The Journal of Machine Learning Research*, 16(1):3151–3181, 2015.

[58] Jorge J Moré and Danny C Sorensen. Computing a trust-region step. *SIAM Journal on Scientific and Statistical Computing*, 4(3):553–572, 1983.

[59] Philipp Moritz, Robert Nishihara, and Michael Jordan. A linearly-convergent stochastic L-BFGS algorithm. In *Artificial Intelligence and Statistics*, pages 249–258. PMLR, 2016.

[60] Lam M Nguyen, Jie Liu, Katya Scheinberg, and Martin Takáč. SARAH: A novel method for machine learning problems using stochastic recursive gradient. In *International Conference on Machine Learning*, pages 2613–2621. PMLR, 2017.

[61] Lam M Nguyen, Jie Liu, Katya Scheinberg, and Martin Takáč. Stochastic recursive gradient algorithm for nonconvex optimization. *arXiv preprint arXiv:1705.07261*, 2017.

[62] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Series in Operations Research and Financial Engineering. Springer Science & Business Media, 2006.

[63] Jacob Rafati and Roummel F Marcia. Improving L-BFGS initialization for trust-region methods in deep learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 501–508. IEEE, 2018.

[64] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, pages 314–323. PMLR, 2016.

[65] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[66] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[67] Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

[68] Nicol N Schraudolph, Jin Yu, and Simon Günter. A stochastic Quasi-Newton method for online convex optimization. In *Artificial intelligence and statistics*, pages 436–443. PMLR, 2007.

[69] Yixun Shi. Globally convergent algorithms for unconstrained optimization. *Computational Optimization and Applications*, 16:295–308, 2000.

[70] Sandro Skansi. *Introduction to Deep Learning: from logical calculus to artificial intelligence*. Springer, 2018.

[71] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[72] Shigeng Sun and Jorge Nocedal. A trust region method for noisy unconstrained optimization. *Mathematical Programming*, pages 1–28, 2023.

[73] Wenyu Sun. Non-monotone trust-region method for solving optimization problems. *Applied Mathematics and Computation*, 156(1):159–174, 2004.

[74] Xiao Wang, Shiqian Ma, Donald Goldfarb, and Wei Liu. Stochastic Quasi-Newton methods for nonconvex stochastic optimization. *SIAM Journal on Optimization*, 27(2):927–956, 2017.

[75] Xiaoyu Wang and Ya-xiang Yuan. Stochastic trust-region methods with trust-region radius depending on probabilistic models. *arXiv preprint arXiv:1904.03342*, 2019.

[76] Zengxin Wei, Guoyin Li, and Liqun Qi. New Quasi-Newton methods for unconstrained optimization problems. *Applied Mathematics and Computation*, 175(2):1156–1188, 2006.

[77] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[78] Peng Xu, Fred Roosta, and Michael W Mahoney. Second-order optimization for non-convex machine learning: An empirical study. In *Proceedings of the 2020 SIAM International Conference on Data Mining*, pages 199–207. SIAM, 2020.

[79] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second-order optimizer for machine learning. In *proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10665–10673, 2021.

[80] Mahsa Yousefi and Ángeles Martínez Calomardo. A stochastic non-monotone trust-region training algorithm for image classification. In *2022 16th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, pages 522–529, Los Alamitos, CA, USA, Oct 2022. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/SITIS57111.2022.00084.

[81] Mahsa Yousefi and Ángeles Martínez. On the efficiency of stochastic Quasi-Newton methods for deep learning. *arXiv preprint arXiv:2205.09121*, 2022.

[82] Mahsa Yousefi and Ángeles Martínez Calomardo. A matlab-based tutorial on implementing custom loops for training a deep neural network. 2022. http://doi.org/10.13140/RG.2.2.33008.94720.

[83] Mahsa Yousefi and Ángeles Martínez Calomardo. A stochastic modified limited memory BFGS for training deep neural networks. In *Intelligent Computing: Proceedings of the 2022 Computing Conference, Volume 2*, pages 9–28. Springer, 2022. https://doi.org/10.1007/978-3-031-10464-0_2.

[84] Liu Ziyin, Botao Li, and Masahito Ueda. SGD may never escape saddle points. *arXiv preprint arXiv:2107.11774*, 2021.