Advanced Methods for Scientific Computing

Prof. Luca Formaggia

Authors:

Giorgio Barocco, Francesco Bazzano, Tommaso Tron

Abstract

This project investigates the design, implementation, and performance analysis of advanced Quasi-Newton optimization methods, specifically **L-BFGS** and its stochastic variant **S-LBFGS** applied to the fields of neural networks and scientific computing. Originating from a foundational CPU implementation of BFGS, the work has expanded into a comprehensive framework capable of training Multi-Layer Perceptrons (MLPs) on both CPU and GPU architectures.

The core of the project relies on a modular optimization library developed for CPU execution, which leverages automatic differentiation. This library encompasses deterministic solvers, such as Gradient Descent and Newton's method, alongside a implementation of S-LBFGS.

To address scalability constraints and computational intensity, the project features a parallel backend engineered from scratch using **CUDA**.

The framework is validated through a custom automation suite that benchmarks these implementations against standard datasets, such as MNIST and Fashion-MNIST. These experiments focus on analyzing convergence rates, stability, and the computational trade-offs between deterministic and stochastic methods.

Finally, the scope of the project extends to Scientific Machine Learning through the exploration of Physics-Informed Neural Networks (PINNs). To overcome performance bottlenecks associated with higher-order derivative calculations in standard autodiff libraries, the project is currently integrating **EnzymeAD** to leverage differentiation at the LLVM IR level, applied to differential problems such as Burgers' equation.

A.Y. 2025-2026

Contents

1 Introduction **2**

2 Problem Formulation **3**
- 2.1 Problem Statement . . . 3
  - 2.1.1 Deterministic Optimization . . . 3
  - 2.1.2 Stochastic Optimization . . . 3
- 2.2 Mathematical Formulation and Algorithms . . . 4
  - 2.2.1 Stochastic Gradient Descent (SGD) . . . 4
  - 2.2.2 Limited-Memory BFGS (L-BFGS) . . . 4
  - 2.2.3 Stochastic L-BFGS (S-LBFGS) . . . 5

3 Implementation **6**
- 3.1 Architecture . . . 6
  - 3.1.1 CPU Backend (Eigen + OpenMP) . . . 6
  - 3.1.2 CUDA Backend . . . 7

4 Benchmark and Experiment **9**
- 4.1 Experimental Setup . . . 9
- 4.2 Benchmark Methodology . . . 9

5 Results **10**
- 5.1 Performance analysis . . . 10
  - 5.1.1 CPU Backend - MNIST . . . 10
  - 5.1.2 CPU Backend - Fashion-MNIST . . . 11
  - 5.1.3 CPU Backend - Full Fashion-MNIST Dataset . . . 13
  - 5.1.4 GPU Backend - MNIST and Fashion-MNIST . . . 14
- 5.2 Scalability and Runtime Analysis . . . 16
- 5.3 Stochastic LBFGS scalability with multi-core CPU . . . 17
- 5.4 Deeper Network with GPU and Fashion-MNIST . . . 18

6 PINN: Burgers' Equation **20**
- 6.1 Mathematical Formulation . . . 20
- 6.2 Differentiation Strategy . . . 21
- 6.3 Numerical Experiments . . . 21
  - 6.3.1 Experimental Setup . . . 21
  - 6.3.2 Results . . . 22

7 Conclusions **23**

1 Introduction

The training of deep neural networks represents one of the most significant computational challenges in modern artificial intelligence. From a mathematical perspective, this task is framed as a high dimensional, non convex optimization problem, where the objective is to minimize a loss function that quantifies the discrepancy between predictions and targets. As network architectures grow in depth and complexity, reaching millions or billions of parameters, the properties of the optimization landscape become increasingly critical to the success of the learning process.

Currently, the dominant paradigm relies on first order methods, specifically Stochastic Gradient Descent (SGD) and its adaptive variants such as Adam or RMSProp. While these methods are computationally efficient per iteration, they only use gradient information, making them sensitive to hyperparameter tuning and prone to slow convergence in ill conditioned problems.

The limitation suggest a theoretical advantage for second-order methods, which incorporate curvature information. Quasi-Newton methods, in particular L-BFGS offer a good trade off by approximating curvature information with linear memory complicity.

In this project we explore the design and performance of a custom optimization framework that implements both deterministic and stochastic variants of L-BFGS (S-LBFGS). By leveraging a dual-backend architecture (CPU and CUDA), we analyze the trade-offs between the generalization capabilities of stochastic methods and the convergence speed of Quasi-Newton algorithms, applying them to both standard image classification and the solution of partial differential equations.

2 Problem Formulation

2.1 Problem Statement

We consider the classical unconstrained non linear optimization problem:

$$\min_{w \in \mathbb{R}^d} F(w) \tag{2.1}$$

where $F : \mathbb{R}^d \to \mathbb{R}$ is a smooth objective function.

2.1.1 Deterministic Optimization

In the standard setting, we assume we can evaluate $F(w)$ and its full gradient $\nabla F(w)$ precisely. A standard second order method is **Newton's method**:

$$w_{k+1} = w_k - H_k^{-1} \nabla F(w_k) \tag{2.2}$$

where $H_k = \nabla^2 F(w_k)$ is the Hessian matrix. While Newton's method boasts quadratic convergence, it is computationally expensive ($O(d^3)$ per step to invert the Hessian or $O(d^2)$ to solve the linear system).

To mitigate this, Quasi-Newton methods approximate the curvature. The **BFGS (Broyden–Fletcher–Goldfarb–Shanno)** algorithm approximates the inverse Hessian $B_k \approx H_k^{-1}$ iteratively using gradient information. The update rule ensures the secant condition $B_{k+1} y_k = s_k$ is satisfied, typically using the Sherman-Morrison formulation:

$$B_{k+1} = (I - \rho_k s_k y_k^T) B_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.3}$$

where $s_k = w_{k+1} - w_k$, $y_k = \nabla F(w_{k+1}) - \nabla F(w_k)$, and $\rho_k = (y_k^T s_k)^{-1}$. While BFGS avoids explicit Hessian inversion, it still requires storing the dense $d \times d$ matrix B_k.

L-BFGS (Limited-memory BFGS) further refines this by storing only the last m pairs of vectors (s_i, y_i), implicitly representing the inverse Hessian. This reduces memory complexity to $O(md)$ and time complexity to $O(md)$, making it suitable for large-scale deterministic problems.

2.1.2 Stochastic Optimization

We specifically address the **Finite Sum Minimization** problem, central to machine learning:

$$F(w) = \frac{1}{N} \sum_{i=1}^{N} f_i(w) \tag{2.4}$$

where N is the number of data points. Computing the full gradient is often prohibitive.

Stochastic Gradient Descent (SGD) addresses this by approximating the gradient using a mini-batch $S \subset \{1, \ldots, N\}$:

$$w_{k+1} = w_k - \eta_k \left(\frac{1}{|S|} \sum_{i \in S} \nabla f_i(w_k) \right) \tag{2.5}$$

Although computationally efficient per iteration, SGD suffers from high variance introduced by the sampling process, necessitating diminishing step sizes η_k which leads to sublinear convergence rates.

S-LBFGS aims to combine the curvature benefits of Quasi-Newton methods with the efficiency of stochastic optimization [1]. It employs variance reduction techniques and stable curvature updates to achieve faster convergence than standard SGD in convex and non-convex settings.

2.2 Mathematical Formulation and Algorithms

2.2.1 Stochastic Gradient Descent (SGD)

The fundamental baseline for training neural networks is Stochastic Gradient Descent [2]. In the context of the finite-sum minimization problem $F(w) = \frac{1}{N}\sum_{i=1}^{N} f_i(w)$, the exact gradient $\nabla F(w)$ requires a pass over the entire dataset, which is computationally prohibitive for large N.

SGD approximates the true gradient by computing the gradient of a randomly sampled mini-batch $S_k \subset \{1, \ldots, N\}$ at iteration k. The update rule is defined as:

$$w_{k+1} = w_k - \eta_k \hat{g}(w_k), \quad \text{where} \quad \hat{g}(w_k) = \frac{1}{|S_k|}\sum_{i \in S_k} \nabla f_i(w_k) \tag{2.6}$$

Here, $\hat{g}(w_k)$ is an unbiased estimator of the full gradient, meaning $\mathbb{E}[\hat{g}(w_k)] = \nabla F(w_k)$. However, the variance of this estimator, $\mathbb{E}[\|\hat{g}(w_k) - \nabla F(w_k)\|^2]$, remains nonzero even as the iterates approach a minimizer. This noise prevents SGD from converging to the exact solution with a constant step size η. To ensure convergence, η_k must decay over time (e.g., $\eta_k \propto 1/k$), which inevitably leads to sub-linear convergence rates. Furthermore, SGD relies solely on first-order information, making it sensitive to ill-conditioned loss surfaces where the curvature varies significantly across dimensions.

2.2.2 Limited-Memory BFGS (L-BFGS)

To address the issues of conditioning and convergence speed, Quasi-Newton methods incorporate curvature information. The standard Newton update is $w_{k+1} = w_k - H_k^{-1}\nabla F(w_k)$, where $H_k = \nabla^2 F(w_k)$. Since computing the inverse Hessian is $O(d^3)$, BFGS approximates the inverse Hessian H_k^{-1} directly with a matrix B_k that is updated iteratively to satisfy the *secant condition*:

$$B_{k+1} y_k = s_k \tag{2.7}$$

where $s_k = w_{k+1} - w_k$ is the parameter displacement and $y_k = \nabla F(w_{k+1}) - \nabla F(w_k)$ is the gradient difference.

For large-scale problems ($d > 10^4$), storing the dense matrix B_k is impossible. **L-BFGS** [3] (Limited-memory BFGS) circumvents this by storing only the last m pairs of curvature vectors $\{(s_i, y_i)\}_{i=k-m}^{k-1}$. Instead of forming B_k explicitly, the matrix-vector product $p_k = -B_k \nabla F(w_k)$ is computed efficiently using the **Two-Loop Recursion** algorithm.

The Two-Loop Recursion

This algorithm, implemented in our `lbfgs_two_loop` function, computes the descent direction $r = H_k \nabla F(w_k)$ (using H to denote the inverse approximation) with $O(md)$ complexity:

1. **Backward Pass:** We start with the current gradient $q = \nabla F(w_k)$. We iterate backwards from the most recent curvature pair to the oldest (from $i = k-1$ down to $k-m$). For each i, we compute and store a scalar α_i:

   $$\alpha_i = \rho_i s_i^T q, \quad \text{where} \quad \rho_i = \frac{1}{y_i^T s_i} \tag{2.8}$$

   We then update $q \leftarrow q - \alpha_i y_i$. This step essentially "unrolls" the Hessian updates to project the gradient into a space where a simple initial approximation is valid.

2. **Scaling:** After the backward pass, q is scaled by an initial Hessian estimate H_0^k, typically $\gamma_k I$, where $\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$. This scaling ensures that the magnitude of the step is well-calibrated to the local curvature:

   $$r = \gamma_k q \tag{2.9}$$

3. **Forward Pass:** We iterate forwards from the oldest pair to the newest (from $i = k - m$ to $k - 1$). We compute $\beta_i = \rho_i y_i^T r$ and update the direction:

$$r \leftarrow r + s_i(\alpha_i - \beta_i) \tag{2.10}$$

The resulting vector r is the approximate Newton direction. L-BFGS combines this direction with a strong Wolfe line search to ensure global convergence in deterministic settings.

2.2.3 Stochastic L-BFGS (S-LBFGS)

Applying L-BFGS directly to stochastic gradients fails because the noise in the gradient differences y_k corrupts the curvature estimates, leading to unstable updates. Our implementation follows the robust S-LBFGS method[4], which stabilizes the algorithm through two key mechanisms: **Variance Reduction** and **Stable Curvature Updates**.

Variance Reduction via SVRG

To reduce the noise in the gradient while maintaining efficiency, we employ a semi-stochastic gradient estimator. The optimization proceeds in epochs. At the beginning of each epoch (every m iterations), a reference point $\tilde{w}$ is fixed (snapshot), and the *full gradient* $\mu = \nabla F(\tilde{w})$ is computed over the entire dataset.

During the inner iterations t, given the current weights w_t, we sample a mini-batch S_t and compute the variance-reduced gradient:

$$v_t = \underbrace{\nabla f_{S_t}(w_t) - \nabla f_{S_t}(\tilde{w})}_{\text{Mean zero noise reduction}} + \underbrace{\mu}_{\text{True gradient}} \tag{2.11}$$

As w_t converges towards $\tilde{w}$, the term in the brackets approaches zero, effectively eliminating the variance of the gradient estimator. This allows S-LBFGS to use constant step sizes without the noise barrier typical of SGD.

Hessian-Free Curvature Updates

To obtain reliable curvature pairs (s, y) in a stochastic setting, we decouple the curvature collection from the parameter updates.

1. **Averaging:** Every L iterations, we compute the average of the recent iterates $u_r = \frac{1}{L}\sum_j w_j$ to filter out high-frequency noise. The displacement is defined as $s_r = u_r - u_{r-1}$.
2. **Hessian-Vector Products (HVP):** The vector y_r should approximate $\nabla^2 F(w)s_r$. Instead of forming the Hessian, we use a **Finite Difference approximation** on a dedicated mini-batch b_H. As implemented in `finite_difference_hvp`:

$$y_r \approx \frac{\nabla f_{b_H}(u_r + \epsilon s_r) - \nabla f_{b_H}(u_r - \epsilon s_r)}{2\epsilon} \tag{2.12}$$

   This centered difference approach provides a second order accuracy approximation of the product $\nabla^2 F \cdot s$ using only gradient evaluations.

These stable pairs (s_r, y_r) are stored in the history buffer and used within the standard L-BFGS two loop recursion to condition the variance-reduced gradient v_t, effectively scaling the stochastic step by the inverse curvature.

3 Implementation

3.1 Architecture

The codebase is engineered around a dual backend architecture designed to maximize performance on both standard multi-core CPUs and NVIDIA GPUs. While sharing a unified high level workflow, defining a neural network topology, defining a loss function, and executing an iterative optimization strategy, the two backends diverge significantly in their internal data structures and memory management to exploit the specific hardware capabilities of each platform. The system is designed with modularity in mind, allowing the interchange of solvers and network configurations without altering the high level training loop.

3.1.1 CPU Backend (Eigen + OpenMP)

The CPU backend relies on the `Eigen` template library for high performance linear algebra operations, ensuring vectorization and cache friendly memory access patterns. To leverage multi-core processors, strictly computationally intensive loops, particularly within the stochastic optimization routines, are parallelized using OpenMP.

Optimization Framework

The core of the optimization logic is encapsulated in two main classes: `FullBatchMinimizer` for Gradient Descent, BFGS, LBFGS and `StochasticMinimizer` for Stochastic Gradient Descent and SLBFGS. These abstract base classes heavily rely on C++ templates to abstract the solver logic from the underlying vector and matrix types. The framework provides essential infrastructure for:

- **Line Search Strategies**: A robust line search mechanism implementing the Wolfe conditions is built directly into the base class, ensuring strictly decreasing objective values and sufficient curvature conditions during optimization steps.
- **Automatic Differentiation**: The framework supports flexible gradient computation strategies, including a custom reverse mode automatic differentiation implementation via the autodiff library.

The Stochastic Limited-memory BFGS (S-LBFGS) implementation, located in `src/minimizer/s_lbfgs.hpp`, is particularly notable. It employs a two loop recursion algorithm to approximate the inverse Hessian without explicit formation, maintaining a rolling history of curvature pairs (s_k, y_k) to efficiently handle high dimensional optimization landscapes.

Network Stack

The neural network architecture, defined in `src/network.hpp`, adopts a data oriented design. While the network is conceptually a sequence of layers (managed as `std::unique_ptr<Layer>`), the trainable parameters (weights and biases) are efficiently stored in contiguous memory blocks (`std::vector<double>`). This design choice allows the network to expose its entire parameter state as a single flat vector to the optimizer, decoupling the optimization algorithm from the network topology. Forward and backward propagation routines utilize efficient Eigen matrix-matrix multiplications, ensuring numerical stability.

3.1.2 CUDA Backend

The GPU backend is engineered for high throughput training on NVIDIA hardware, prioritizing data locality and minimizing host-device synchronization.

Memory Management and RAII

To manage the complexity of manual memory allocation, the backend utilizes a Resource Acquisition Is Initialization (RAII) pattern. The `DeviceBuffer<T>` class (`src/cuda/device_buffer.cuh`) serves as a smart pointer wrapper around raw device pointers, strictly managing the lifecycle of GPU memory (allocation via `cudaMalloc` and deallocation via `cudaFree`). Similarly, the `CublasHandle` class wraps the `cublasHandle_t` resource, ensuring proper initialization and destruction of the cuBLAS context.

Kernel Design and Execution

The computational part is divided between the cuBLAS library and custom CUDA kernels:

- **Linear Algebra**: GEMM (General Matrix Multiply) operations, for the dense layers of the Multi Layer Perceptron, are offloaded to optimized cuBLAS routines (`cublasSgemm`).
- **Custom Kernels**: Operations that do not map standard BLAS routines, such as activation functions (ReLU, Sigmoid, Tanh), their derivatives, and loss computations, are implemented as custom CUDA kernels in `src/cuda/kernels.cuh`. These kernels are designed for coalesced memory access to maximize memory bandwidth utilization.

Network and Optimization

The network implementation (`src/cuda/network.cuh`) mirrors the CPU structure but maintains all state (activations, parameters, gradients) in device memory. A key optimization in the CUDA training loop is the fusion of operations, where possible, to reduce kernel launch overhead. The network exposes a `compute_loss_and_grad` interface that performs the forward pass, computes the loss, and immediately triggers the backward pass on the device, returning only the scalar loss to the host. The optimization algorithms (e.g., in `src/cuda/minimizer_base.cuh`) are implemented directly on the device. Vector updates (AXPY operations) and norm calculations are performed via cuBLAS, ensuring that the parameter vectors remain in GPU memory throughout the entire training process.

Overall Architecture

To reconcile the divergent interfaces of the CPU and GPU backends, further complicated by the structural differences between full batch and stochastic optimization methods, we adopted a design based on the Adapter Pattern implemented via C++ Template Specialization. While this approach prioritizes performance over purely object oriented polymorphism, it effectively unifies the underlying architectures without runtime overhead.
The core of this abstraction is the `UnifiedOptimizer` class. This component serves as a standardized interface for all optimization strategies, masking the complexity of the underlying implementations. Whether the `Backend` is `CpuBackend` or `CudaBackend`, this class exposes a consistent API to perform training, effectively bridging the gap between the host-side Eigen iterations and the device-side cuBLAS kernels.
Complementing this, we introduced the `UnifiedLauncher` class, which acts as a high level Facade for the entire training lifecycle. This unified entry point manages the neural network construction and, crucially, handles backend specific data logistics. For the CPU backend, it

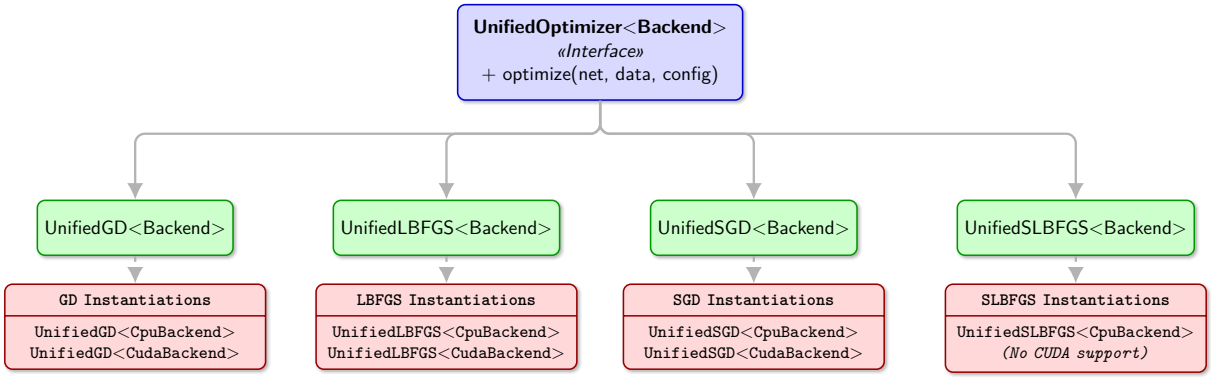


Figure 3.1: Implementation diagram of UnifiedOptimizer

simply wraps the Eigen matrices; for the CUDA backend, it automatically orchestrates the allocation of device memory and the transfer of training data to the GPU. By decoupling the experimental setup from the execution details, the `UnifiedLauncher` allows the user to define an experiment once and execute it transparently on either hardware infrastructure.

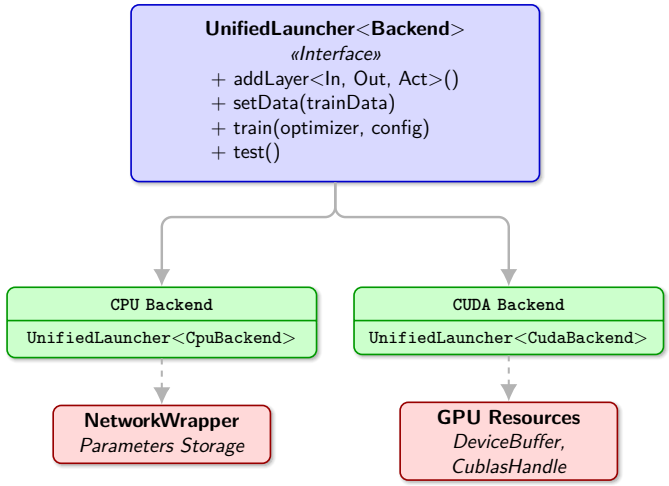


Figure 3.2: Implementation diagram of UnifiedLauncher

4 Benchmark and Experiment

The experiments analyze convergence rates, stability, and computational trade-offs between deterministic solvers (Gradient Descent, L-BFGS) and their stochastic counterparts (SGD, S-LBFGS). Furthermore, we evaluate the acceleration provided by the CUDA backend compared to the OpenMP-based CPU implementation.

4.1 Experimental Setup

All experiments are orchestrated via the `UnifiedLauncher` interface to ensure consistent initialization and execution across backends.

Datasets

We utilize two standard 28×28 grayscale image classification datasets, flattened into 784-dimensional vectors:

- **MNIST**: Handwritten digits (60k training, 10k test), serving as a baseline for correctness.
- **Fashion-MNIST**: Clothing items (60k training, 10k test) sharing the MNIST structure but presenting a more challenging optimization landscape due to higher intra-class variability.

Neural Network Architecture

The benchmarks focus on Multi Layer Perceptrons. While the framework supports arbitrary topologies, the reference configuration consists of:

- **Input**: 784 units (flattened image).
- **Hidden**: Dense layers utilizing ReLU activation.
- **Output**: 10 units with Linear activation.

4.2 Benchmark Methodology

The performance analysis is structured into three phases:

1. **Solver Validation:** We initially evaluate convergence behavior and classification accuracy on a single-hidden-layer network. This establishes a baseline for comparing deterministic and stochastic methods. We then increase network depth to assess optimizer stability in complex landscapes, particularly on the CUDA backend.
2. **Scalability Analysis:** We measure the computational trade-offs between CPU and GPU platforms by progressively increasing the number of trainable parameters.
3. **Parallel Efficiency:** Finally, we analyze the strong and weak scaling properties of the CPU backend, evaluating the effectiveness of the OpenMP implementation in exploiting multi-core architectures.

5 Results

5.1 Performance analysis

In this section, we present the experimental results obtained on the MNIST and Fashion-MNIST datasets. We employed an MLP with a single hidden layer of size 128 (topology $784 \to 128 \to 10$). To ensure a detailed analysis of the optimization dynamics within a reasonable computational time on the CPU backend (on a machine with 8 threads), the training was performed on a subset of $N = 5,000$ images. All algorithms were initialized with the same random seed to ensure a fair comparison of the optimization trajectories starting from the same point in the parameter space.

5.1.1 CPU Backend - MNIST

Figure 5.1 reports the optimization behavior on the MNIST dataset using the subset of 5,000 samples. The network, a shallow fully-connected architecture ($784 \to 128 \to 10$), was trained comparing deterministic first-order and second-order methods (GD, L-BFGS) against stochastic approaches (SGD, S-LBFGS).

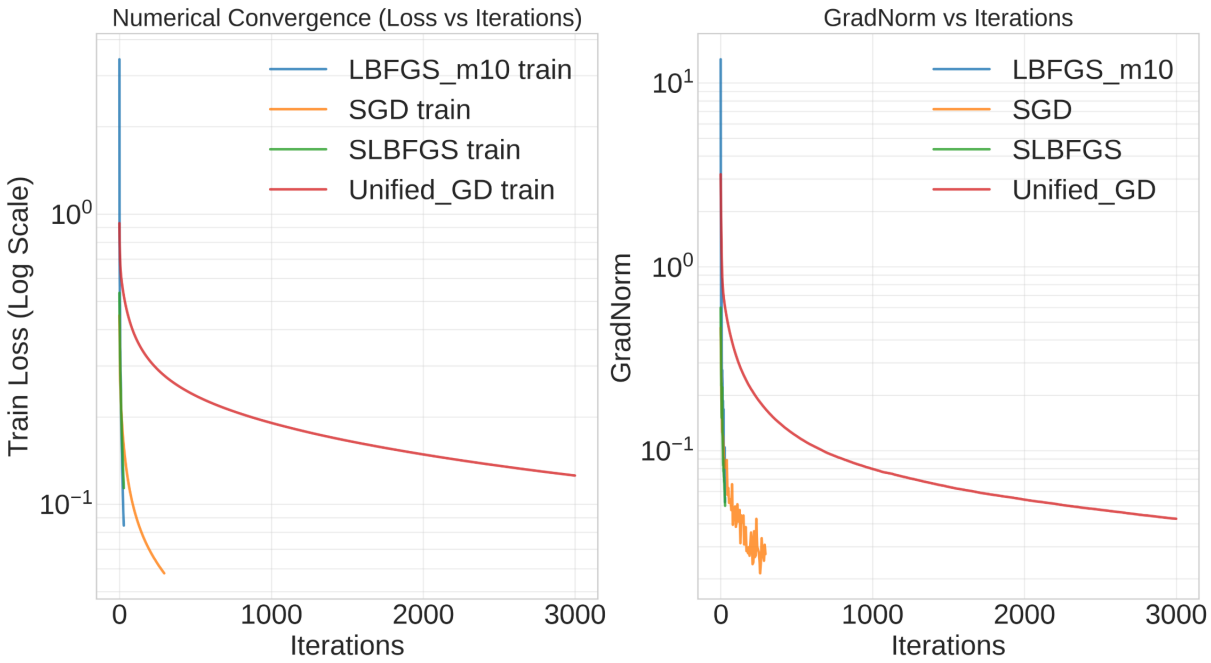


Figure 5.1: Performance Analysis on MNIST (CPU Backend). Comparison of deterministic (L-BFGS, GD) and stochastic (S-LBFGS, SGD) solvers.

Gradient Descent (GD) exhibits a stable and monotonic convergence behavior, but requires a relatively large number of iterations to achieve acceptable performance. Due to its full-batch nature GD progresses slowly along the optimization trajectory, resulting in the lowest final accuracy among the considered solvers.

Stochastic Gradient Descent (SGD) achieves the highest training and test accuracy in this setting. The use of mini-batches introduces gradient noise that promotes exploration of the loss landscape and acts as an implicit regularizer, ultimately leading to improved generalization. However, this comes at the cost of a larger number of iterations and a less regular optimization trajectory, as reflected by the noisier gradient norm evolution.

Stochastic L-BFGS (S-LBFGS) represents a compromise between first order stochastic optimization and second order curvature-aware methods. By incorporating approximate curvature information while retaining stochastic updates, S-LBFGS achieves faster loss reduction than standard GD and exhibits a more structured convergence behavior compared to SGD. Although its final accuracy does not surpass that of SGD, the method demonstrates improved efficiency in the early optimization stages, highlighting the benefit of second-order information even in noisy settings.

Deterministic L-BFGS ($m = 10$) stands out for its rapid convergence and efficient exploitation of curvature information. The method achieves a substantial reduction in training loss within a small number of iterations, as clearly visible in both the loss and gradient norm plots. This behavior confirms the effectiveness of quasi-Newton updates in navigating the parameter space and reaching high quality solutions on a limited computational budget. While the gains in training accuracy only partially translate to test performance, L-BFGS remains a highly effective optimizer when fast convergence and precise minimization of the empirical risk are primary objectives.

Solver	**Training Accuracy (%)**	**Test Accuracy (%)**
GD	91.98	89.50
SGD	97.64	93.15
S-LBFGS	93.42	90.94
L-BFGS ($m = 10$)	94.30	92.11

Table 5.1: Training and test performance on MNIST using a $784 \to 128 \to 10$ network (CPU backend).

5.1.2 CPU Backend - Fashion-MNIST

Figure 5.2 illustrates the optimization behavior on the Fashion-MNIST dataset, maintaining the same restricted dataset size of 5,000 images and the shallow architecture ($784 \to 128 \to 10$). Compared to standard MNIST, the Fashion-MNIST dataset poses a significantly more challenging classification problem, characterized by higher intra-class variability and less clearly separable patterns. These properties have a direct impact on the optimization dynamics and amplify the differences between first-order and curvature-aware methods.

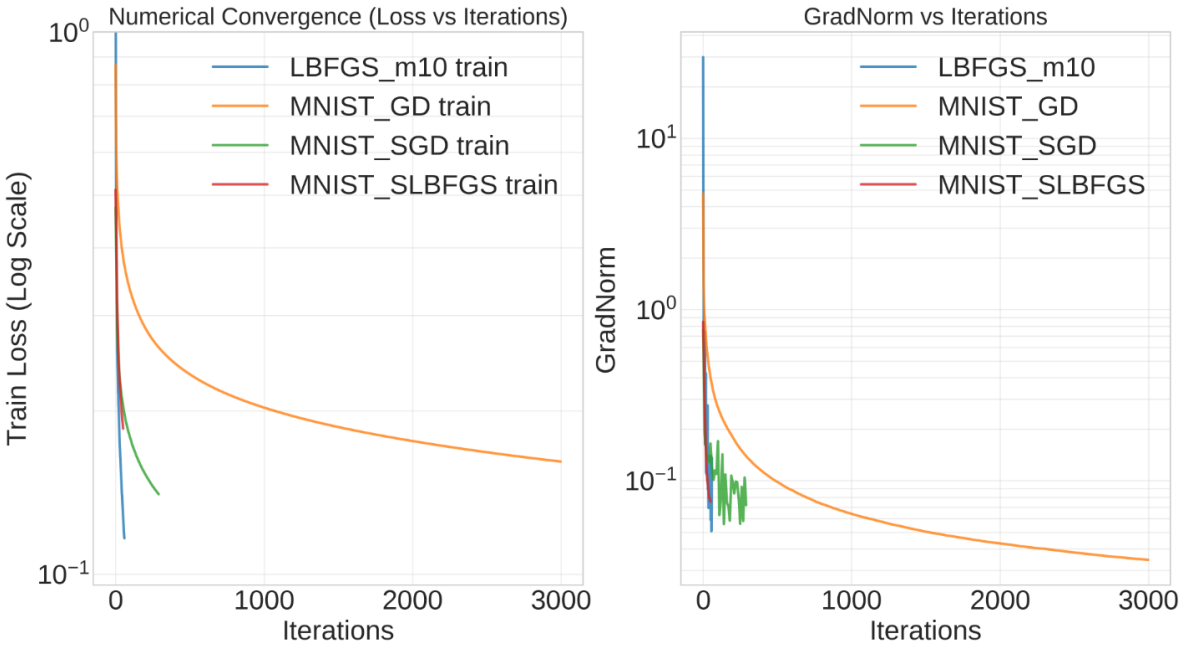


Figure 5.2: Performance Analysis on MNIST - Fashion (CPU Backend). Comparison of deterministic (L-BFGS, GD) and stochastic (S-LBFGS, SGD) solvers.

In this setting, Gradient Descent (GD) exhibits limited effectiveness. Although the method maintains a smooth and well behaved optimization trajectory, the increased complexity of the loss surface slows down progress substantially. As a result, GD converges to solutions with relatively high residual loss and reduced classification accuracy, highlighting the difficulty of relying solely on first-order information for more demanding datasets when operating on a CPU backend.

Stochastic Gradient Descent (SGD) benefits from its inherent noise, which enables broader exploration of the parameter space and leads to improved performance compared to GD. Nevertheless, the gains remain moderate, and convergence requires a large number of iterations. This suggests that, for Fashion-MNIST, stochasticity alone is not sufficient to fully compensate for the increased curvature and nonlinearity of the optimization landscape.

Stochastic L-BFGS (S-LBFGS) introduces approximate second-order information into the stochastic optimization process, resulting in a more structured descent direction during the early optimization phase. Compared to its behavior on MNIST, the advantages of S-LBFGS are less pronounced on Fashion-MNIST, as gradient noise partially obscures curvature estimates. However, the method still demonstrates faster initial loss reduction than GD, indicating that even imperfect curvature information can be beneficial in navigating more complex loss surfaces.

Deterministic L-BFGS ($m = 10$) emerges as the most effective solver in terms of optimization efficiency. By explicitly exploiting curvature information, L-BFGS rapidly adapts step directions to the local geometry of the objective function, achieving a substantial reduction in training loss within a limited number of iterations. While the improvement in test accuracy over SGD remains modest, the method consistently attains the best overall performance among the second order variants. This behavior highlights the strength of quasi-Newton methods in addressing increased problem complexity and reinforces their relevance as powerful optimization tools, particularly when fast convergence and accurate empirical risk minimization are central to the study.

Solver	Training Accuracy (%)	Test Accuracy (%)
GD	84.40	79.58
SGD	86.40	81.42
S-LBFGS	83.84	79.72
L-BFGS ($m = 10$)	88.10	82.38

Table 5.2: Training and test performance on Fashion-MNIST using a $784 \to 128 \to 10$ network (CPU backend).

5.1.3 CPU Backend - Full Fashion-MNIST Dataset

Building on the promising results obtained on the reduced subset, we extended the experimental evaluation to the full Fashion-MNIST dataset, comprising $N = 60,000$ training images. This scaling allows us to assess the robustness and computational efficiency of the solvers in a more realistic training scenario. While the reduced subset provided initial insights into convergence dynamics, the full dataset significantly increases the computational load, particularly for the CPU backend, making the trade-off between iteration cost and convergence speed critical.

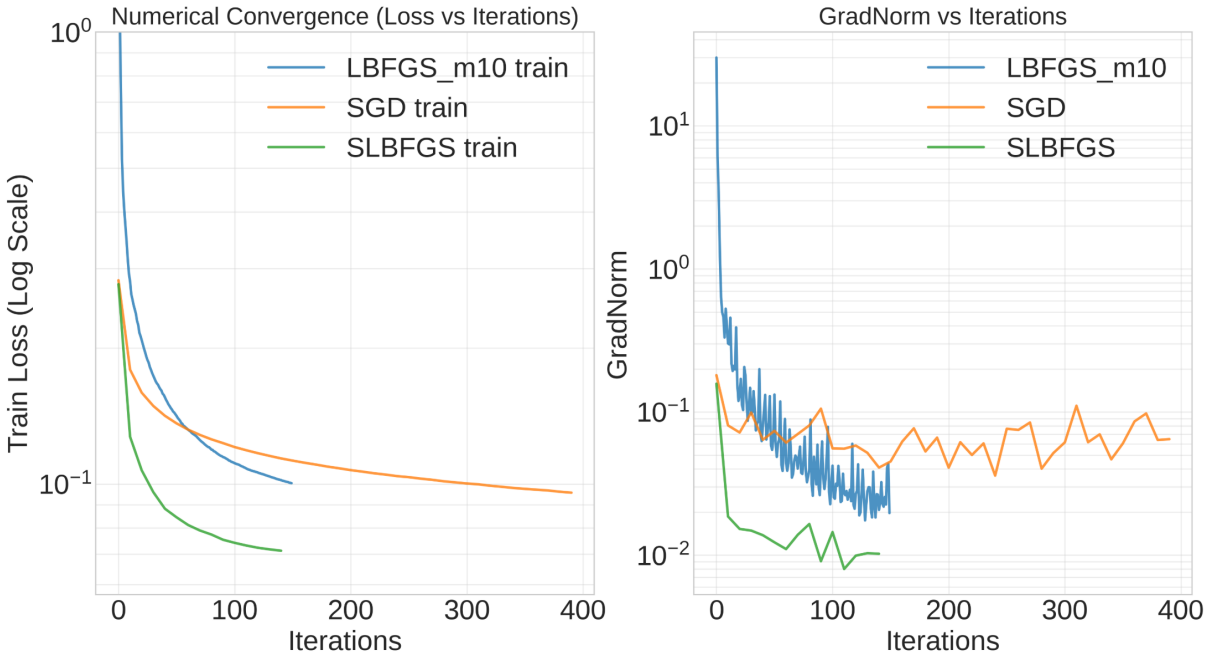


Figure 5.3: Comparison of optimization trajectories on the full Fashion-MNIST dataset (60k samples). Time-to-solution analysis highlights the efficiency gap between stochastic and quasi-Newton methods.

Computational Efficiency and Time-to-Solution

The transition to the full dataset exposes a dramatic divergence in wall-clock execution time. Standard Stochastic Gradient Descent (SGD) requires approximately 2000 seconds to complete the training process. While its per-iteration cost remains low, the sheer number of updates required to traverse the dataset and reach convergence accumulates into a substantial total training time.

L-BFGS solver completes the optimization in approximately 250 seconds, nearly an order of magnitude faster than SGD. This result underscores the superior scalability of second-order methods on CPU architectures for this class of problems: the ability to take larger, curvature-

informed steps significantly reduces the number of iterations needed, more than compensating for the overhead of the L-BFGS two loop recursion. S-LBFGS also demonstrates excellent efficiency, converging in roughly 450 seconds. It successfully bridges the gap, offering a training speed comparable to the deterministic algorithm while retaining the benefits of stochastic sampling.

Generalization and Accuracy Analysis

In terms of model quality, S-LBFGS emerges as the top performer. It achieves the highest training accuracy (93.46%) and, crucially, the best test accuracy (87.98%). This suggests that S-LBFGS effectively combines the rapid convergence properties of quasi-Newton methods with the regularization effect inherent in stochastic noise, allowing it to escape sharp local minima that might trap a fully deterministic solver.

Deterministic L-BFGS, while the fastest, yields a slightly lower test accuracy (86.62%) compared to both S-LBFGS and SGD. This phenomenon is consistent with the known properties of full-batch methods in deep learning, which tend to converge to sharp minimizers that may generalize less effectively than the flatter minima found by stochastic processes. SGD, despite its excessive training duration, achieves a respectable test accuracy (86.98%), slightly outperforming deterministic L-BFGS but falling short of the hybrid S-LBFGS approach.

In summary, for the full Fashion-MNIST dataset on a CPU, S-LBFGS proves to be the most balanced choice, delivering a good accuracy with a training time that is competitive with the fastest deterministic solver and significantly superior to standard SGD.

Solver	**Time (s)**	**Train Acc. (%)**	**Test Acc. (%)**
SGD	≈ 2000	89.43	86.98
L-BFGS ($m = 10$)	≈ 250	88.59	86.62
S-LBFGS	≈ 450	**93.46**	**87.98**

Table 5.3: Performance comparison on the full Fashion-MNIST dataset (60,000 samples) on CPU Backend.

5.1.4 GPU Backend - MNIST and Fashion-MNIST

Figures 5.4 and 5.5 report the optimization performance on MNIST and Fashion-MNIST using a fully-connected neural network with architecture $784 \to 128 \to 10$, ReLU activation in the hidden layer, and linear output units. Training is performed on GPU (Nvidia RTX 5070) using deterministic (GD, L-BFGS) and stochastic (SGD) solvers on the full 60k image datasets.

Despite the different nature of the two datasets, the qualitative behavior of the optimization algorithms is nearly identical across MNIST and Fashion-MNIST on the considered architecture. For this reason, the following discussion applies to both benchmarks.

Gradient Descent (GD) exhibits a smooth but slow convergence profile, both in terms of iterations and wall-clock time, due to its full-batch nature. Stochastic Gradient Descent (SGD), while introducing higher variance in the gradient norm, achieves faster practical convergence and consistently better generalization performance, benefiting from its intrinsic regularization effect.

L-BFGS clearly outperforms first order methods in terms of training loss minimization and convergence speed. However, this advantage does not translate into improved test accuracy. On the contrary, L-BFGS attains significantly higher training accuracy while yielding comparable test performance, indicating a pronounced overfitting tendency. Increasing the memory parameter from $m = 10$ to $m = 100$ further improves training accuracy but exacerbates this effect, confirming the sensitivity of second-order methods to over-parameterization in the absence of explicit regularization.

Overall, SGD provides the best trade-off between optimization efficiency and generalization, while L-BFGS proves to be an excellent deterministic optimizer for minimizing empirical risk but less suitable for generalization-critical classification tasks.

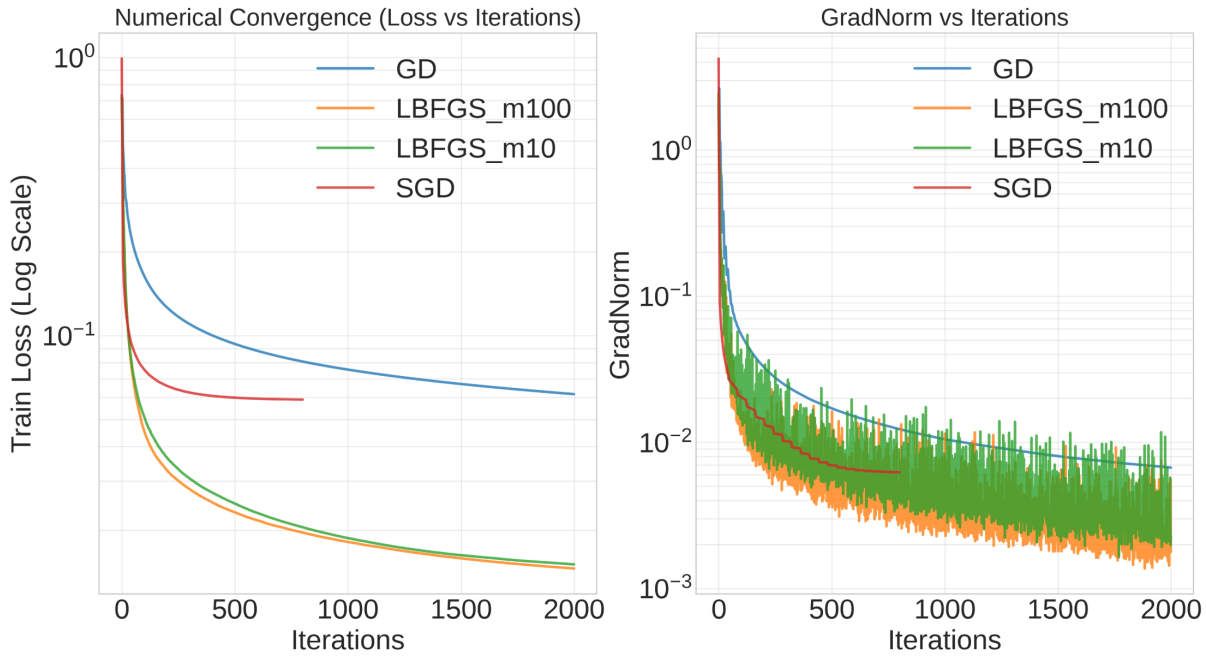


Figure 5.4: Performance Analysis on MNIST (GPU Backend). Comparison of deterministic (L-BFGS, GD) and stochastic (SGD) solvers.

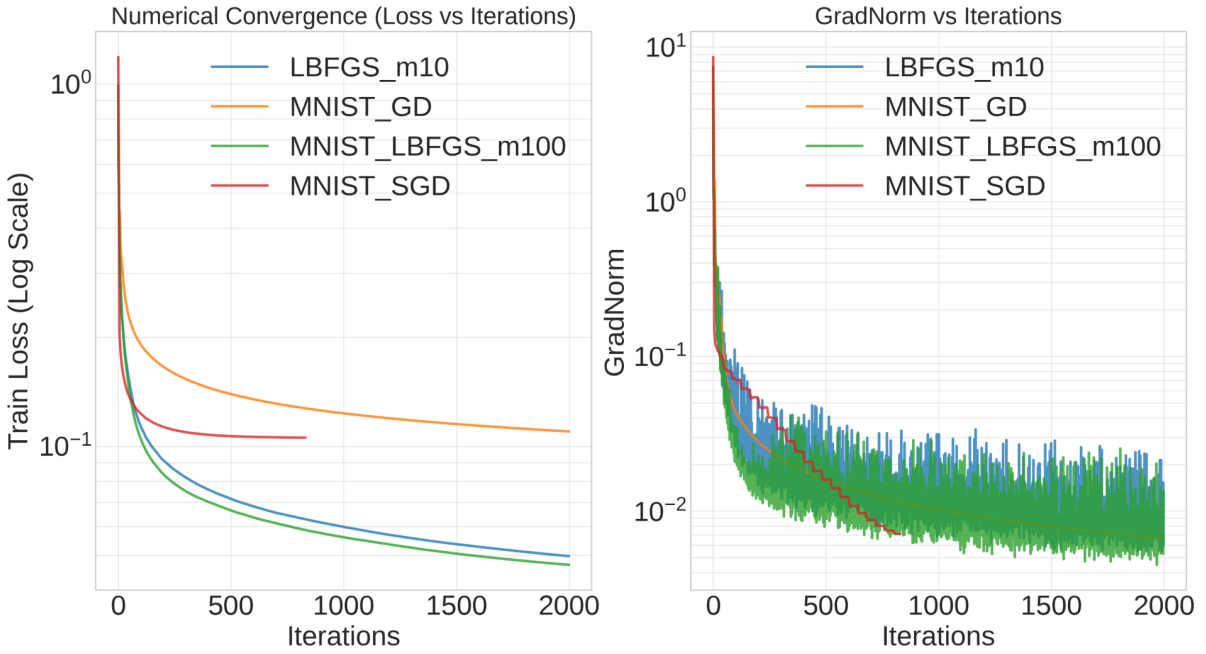


Figure 5.5: Performance Analysis on Fashion-MNIST (GPU Backend). Comparison of deterministic (L-BFGS, GD) and stochastic (SGD) solvers.

Solver	Training MSE	Training Accuracy (%)	Test Accuracy (%)
GD	0.02196	87.82	85.79
SGD	0.02112	88.52	86.62
L-BFGS ($m = 10$)	0.00995	95.99	86.74
L-BFGS ($m = 100$)	0.00941	96.21	86.37

Table 5.4: Training and test performance on Fashion-MNIST using a $784 \to 128 \to 10$ network (GPU backend).

5.2 Scalability and Runtime Analysis

The scalability of the proposed framework was evaluated by measuring the total training time across networks of increasing complexity, ranging from $20,000$ to $250,000$ parameters, using the MNIST dataset (Figure 5.7).

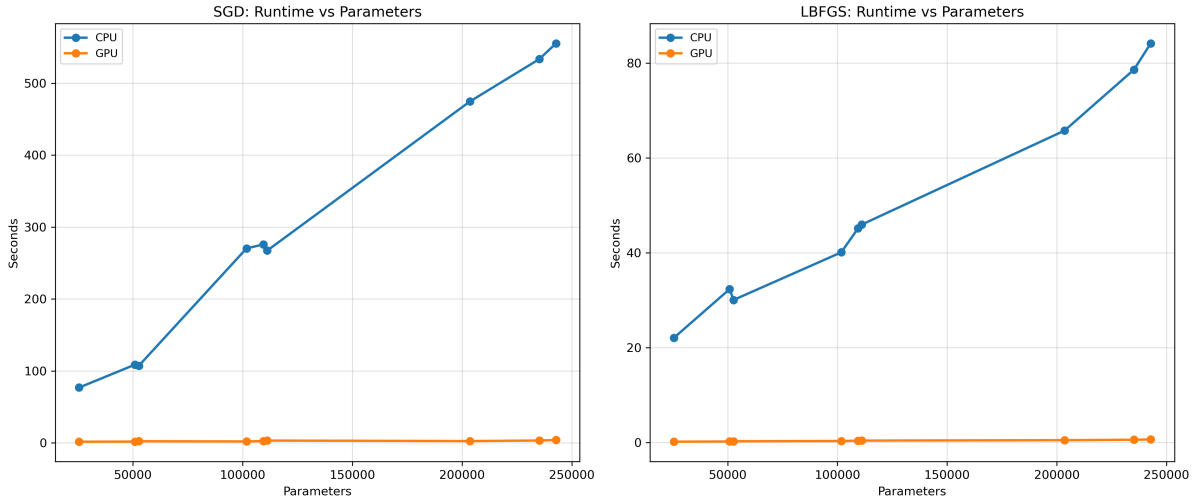


Figure 5.6: Scalability Analysis: Total runtime (in seconds) as a function of network parameters for SGD (Left) and L-BFGS (Right) on CPU vs. GPU backends.

Hardware Impact. The results demonstrate a clear distinction between the backends. The CPU implementation (blue line) exhibits a linear increase in runtime proportional to the number of parameters, reflecting the memory-bound nature of matrix operations on host memory. In contrast, the GPU backend (orange line) shows a near-constant runtime profile. This behavior indicates that for the tested network sizes and the relatively lightweight MNIST workload, the GPU is not compute-bound.

Algorithmic Efficiency on CPU. A notable observation from the CPU results is the performance gap between the optimization algorithms. Despite L-BFGS having a higher computational complexity per iteration ($O(md)$) compared to SGD ($O(d)$), it achieves a significantly lower total runtime. As shown in Figure 5.7, training a $250k$-parameter network takes approximately 550 seconds with SGD but less than 90 seconds with L-BFGS. This result highlights the trade-off between iteration cost and convergence rate: the super-linear convergence of the quasi-Newton method drastically reduces the total number of iterations required to reach the target loss, more than compensating for the additional arithmetic operations performed at each step. Furthermore, the L-BFGS implementation effectively leverages the vectorization capabilities of

the CPU (via Eigen) for its dense linear algebra operations, maintaining high throughput even without GPU acceleration.

5.3 Stochastic LBFGS scalability with multi-core CPU

Building upon the analysis presented in the previous sections, the Stochastic LBFGS (S-LBFGS) algorithm emerged as the most balanced solver in terms of accuracy/time tradeoff. Motivated by these results, we extended our investigation to evaluate the algorithm's parallel efficiency, conducting a scalability analysis with respect to the number of CPU cores on a computing cluster node (28 cores).

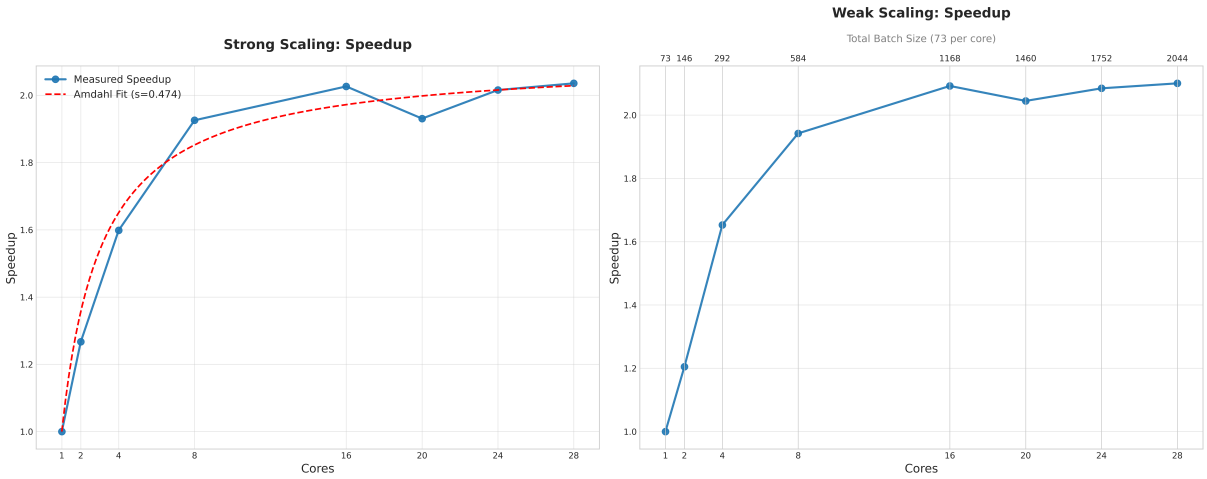


Figure 5.7: Scalability Analysis: Measured speedup ($S = T_1/T_N$) as a function of the number of active cores for Strong Scaling with a fixed global batch size (Left) and Weak Scaling with a scaled batch size $B = 73 \cdot N$ (Right). The top axis of the right plot indicates the resulting Total Batch Size.

Strong Scalability To analyze strong scalability, we adopted a fixed workload model, running 30 iterations of S-LBFGS on the full MNIST dataset (60,000 images). The results exhibit a textbook example of Amdahl's Law behavior: as the number of processing cores increases, the speedup does not grow linearly but rapidly approaches an asymptotic maximum. By fitting the experimental data with Amdahl's Law equation:

$$S(N) = \frac{1}{s + \frac{1-s}{N}} \tag{5.1}$$

we obtained a serial fraction parameter $s \approx 0.474$ (47.4%). This high value suggests that nearly half of the execution time is dominated by non-parallelizable components, such as I/O operations, OpenMP thread synchronization, and, crucially, memory bandwidth contention. Consequently, the maximum theoretical speedup is capped at approximately 2.1×, regardless of the number of cores used.

Weak Scalability We evaluated weak scalability by scaling the workload proportionally to the number of cores (batch size $B = 73 \cdot N$). While Gustafson's Law predicts linear speedup, our results saturated at approximately 2.1×, mirroring the strong scaling trend. This confirms the algorithm is *memory-bound*: despite a constant per-core computational load, S-LBFGS's low arithmetic intensity operations saturate the shared memory bandwidth, creating a bottleneck that prevents effective parallelism beyond a small number of threads.

5.4 Deeper Network with GPU and Fashion-MNIST

In order to further stress the optimization capabilities of the considered solvers, a deeper fully-connected architecture was employed, namely $784 \to 256 \to 128 \to 64 \to 10$, trained on the Fashion-MNIST dataset using a GPU backend. The corresponding optimization dynamics are reported in Figure 5.8.

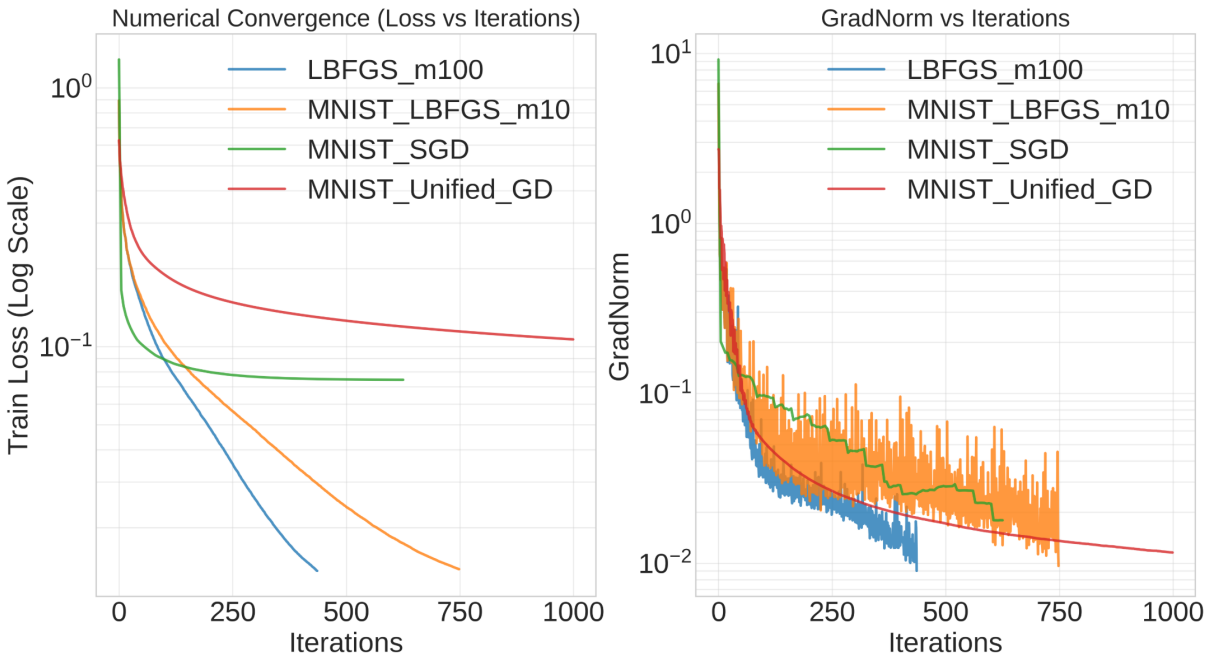


Figure 5.8: Performance analysis on Fashion-MNIST using a deep $784 \to 256 \to 128 \to 64 \to 10$ fully-connected network (GPU backend). Comparison of convergence behavior, computational efficiency, and gradient norm evolution for deterministic (GD, L-BFGS) and stochastic (SGD) optimization methods

Compared to the shallow network, the increased depth and parameter count significantly amplify the differences between first order and second order optimization methods. Gradient Descent (GD) shows limited benefits from the deeper architecture, reaching training and test accuracies comparable to the shallow case, while exhibiting slower convergence. This behavior highlights the difficulty of full-batch first order methods in efficiently navigating the more complex loss landscape induced by deeper networks.

Stochastic Gradient Descent (SGD) benefits from the increased model capacity, achieving a notable improvement in both training and test accuracy. The stochasticity of the updates continues to provide an implicit regularization effect, allowing SGD to exploit the expressive power of the deeper architecture while maintaining good generalization performance.

L-BFGS exhibits the most pronounced behavior in this setting. Both memory configurations ($m = 10$ and $m = 100$) achieve near-perfect optimization of the training objective, with training accuracies exceeding 98% and extremely low training MSE values. This confirms the effectiveness of second order curvature information in accelerating convergence and minimizing the empirical risk, even in deeper architectures. However, despite the substantial gains on the training set, test accuracy improves only marginally with respect to SGD and remains significantly lower than the corresponding training performance. This gap provides clear evidence of overfitting, which becomes more evident as model depth and capacity increase.

Interestingly, the difference between $m = 10$ and $m = 100$ remains limited in terms of test accuracy, suggesting that increasing the L-BFGS memory primarily affects the sharpness of the solution found rather than its generalization properties. Overall, these results reinforce the

conclusion that while L-BFGS is highly effective for deterministic optimization and rapid loss minimization, SGD remains the most robust choice when generalization is the primary objective.

Solver	**Training MSE**	**Training Accuracy (%)**	**Test Accuracy (%)**
GD	0.02133	87.55	85.37
SGD	0.01489	91.37	88.16
L-BFGS ($m = 10$)	0.00277	98.32	88.75
L-BFGS ($m = 100$)	0.00272	98.40	88.30

Table 5.5: Training and test performance on Fashion-MNIST using a deep $784 \to 256 \to 128 \to 64 \to 10$ network (GPU backend).

6 PINN: Burgers' Equation

The previous results rely on a framework tailored to supervised learning. We now consider a physics-informed neural network (PINN) as a complementary case study to highlight the relevance of second order optimization methods beyond data driven tasks.

In this setting the objective function is typically smooth and evaluated in a full-batch regime, while requiring the computation of higher order derivatives, making curvature aware optimization a natural choice. Moreover, this setting provides an opportunity to test our implementations on a deeper neural network, as multiple layers are commonly required in physics informed applications.

6.1 Mathematical Formulation

As a representative example, we considered the one dimensional Burgers' equation, which is commonly adopted as a benchmark in physics informed neural networks given the availability of reference solutions. The equation reads:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}, \quad x \in (-1, 1),\ t \in [0, 1], \tag{6.1}$$

where:

- $u(x, t)$ denotes the velocity field as a function of space x and time t,
- $\nu > 0$ is the kinematic viscosity,
- $\frac{\partial u}{\partial t}$ is the temporal derivative,
- $u\frac{\partial u}{\partial x}$ represents the nonlinear advection term,
- $\nu\frac{\partial^2 u}{\partial x^2}$ corresponds to the diffusion term.

We considered the equation with the following initial and boundary conditions:

$$u(x, 0) = u_0(x), \quad x \in (-1, 1) \tag{6.2}$$
$$u(-1, t) = g(-1, t),\ t \in [0, 1] \tag{6.3}$$
$$u(1, t) = g(1, t),\ t \in [0, 1] \tag{6.4}$$

The total loss function of our physics-informed neural network combines contributions from three sources: the initial condition, the boundary conditions, and the PDE residual. Denoting the network parameters as ω, the loss is defined as

$$F(\omega) = w_{\text{IC}}\, F_{\text{IC}} + w_{\text{BC}}\, F_{\text{BC}} + w_{\text{PDE}}\, F_{\text{PDE}}, \tag{6.5}$$

where w_{IC}, w_{BC}, and w_{PDE} are weighting factors for each contribution, and F_{IC}, F_{BC}, and F_{PDE} denote the losses associated with the initial condition, boundary conditions, and PDE residual, respectively. These are defined as follows:

$$F_{\text{IC}} = \frac{1}{N_{\text{IC}}} \sum_{i=1}^{N_{\text{IC}}} \left(u_\omega(x_i, 0) - u_0(x_i)\right)^2, \tag{6.6}$$

where $u_\omega(x_i, 0)$ is the network prediction at the initial time, and $u_0(x_i)$ is the prescribed initial condition. In our case, $u_0(x) = \sin(\pi x)$.

$$F_{\text{BC}} = \frac{1}{N_{\text{BC}}} \sum_{i=1}^{N_{\text{BC}}} \left(u_\omega(x_i, t_i) - g(x_i, t_i)\right)^2, \tag{6.7}$$

where $u_\omega(x_i, t_i)$ is the network prediction at the boundary points/initial points (x_i, t_i), and $g(x_i, t_i)$ represents the prescribed boundary values: $g(-1, t_i) = g(1, t_i) = 0$.

The PDE residual loss measures the deviation from the Burgers' equation:

$$F_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{i=1}^{N_{\text{PDE}}} \left(\frac{\partial u_\omega}{\partial t} + u_\omega \frac{\partial u_\omega}{\partial x} - \nu \frac{\partial^2 u_\omega}{\partial x^2}\right)^2, \tag{6.8}$$

6.2 Differentiation Strategy

A critical bottleneck in solving this problem arose from the requirement for nested differentiation. Training a PINN requires computing the derivative of the physics residual with respect to the network weights; however, the physics residual itself contains derivatives with respect to the spatial inputs (u_x, u_{xx}, u_t).

Standard operator overloading frameworks (such as the aforementioned autodiff library) failed in this context due to computation graph breakage. When computing the inner spatial derivatives, these libraries evaluate the gradient numerically and return a constant value (stopping the recording tape), rather than a symbolic expression linked to the network parameters. Consequently, the outer optimization loop perceived these derivatives as constant numbers independent of the weights, preventing the backpropagation of gradients required for learning.

To resolve this, we implemented automatic differentiation using EnzymeAD[5]. Unlike standard libraries, Enzyme operates at the LLVM IR (compiler) level. This allows it to generate differentiable code for the derivative calculations themselves, effectively maintaining a continuous, unbroken dependency graph from the physics residuals back to the network weights. This capability enabled the exact computation of higher order derivatives.

6.3 Numerical Experiments

With the differentiation pipeline in place, we investigated the robustness of the optimization framework with different viscosity parameters ν. This parameter plays a crucial role in the regularity of the solution. High values of viscosity dominate the advection term leading to a smooth and velocity field which is relatively straightforward for a neural network to approximate. In contrast, as ν decreases, the advection term prevails, causing the solution to develop steep gradients, which massively increase the difficulty of the problem.

6.3.1 Experimental Setup

To solve the described PDE, we employed a fully connected neural network with the following architecture:

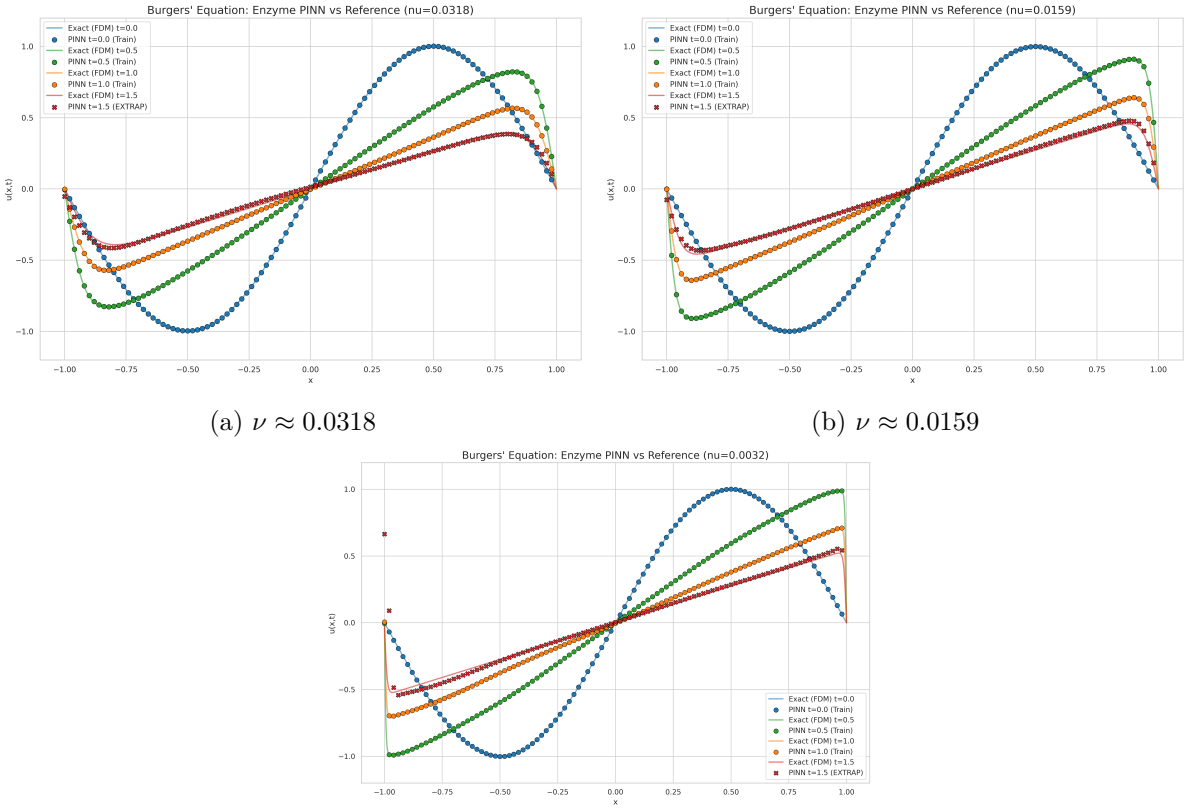
- **Input Layer**: 2 neurons (representing space x and time t).
- **Hidden Layers**: 2 layers composed of 20 neurons.
- **Activation Function**: Hyperbolic Tangent ($tanh$), chosen for its smoothness (C^∞) which is essential to computing higher order derivatives.
- **Output Layer**: 1 neuron (velocity u).

The training was performed using L-BFGS.

6.3.2 Results

To evaluate the solver's robustness to increasingly stiff physical constrains, we performed a sensitivity analysis by varying the kinetic viscosity ν. We tested three values corresponding to increasing Reynolds numbers : $\nu_1 = \frac{0.1}{\pi}$, $\nu_2 = \frac{0.05}{\pi}$ and $\nu_3 = \frac{0.01}{\pi}$.

- At $\nu_1 \approx 0.0318$ (Figure 6.1a), the diffusion term effectively smooths the solution, preventing the formation of sharp gradients. The wavefront remains relatively wide, posing a standard regression task for the neural network.
- At $\nu_2 \approx 0.0159$ (Figure 6.1b), the non-linear advection begins to dominate, causing the wavefront to steepen significantly as $t \to 1$.
- At $\nu_3 \approx 0.0032$ (Figure 6.1c), the solution approaches the inviscid limit. The diffusion is insufficient to counteract the steepening, resulting in a near-discontinuous shock at $x = 0$. This regime represents the critical stress test for the optimization framework, as the loss landscape becomes highly ill-conditioned due to the spectral bias of neural networks.



(a) $\nu \approx 0.0318$

(b) $\nu \approx 0.0159$

(c) $\nu \approx 0.0032$

Figure 6.1: Comparison of the predicted solution $u(x,t)$ at $t = 0$, $t = 0.5$, $t = 1$, $t = 1.5$ for decreasing viscosity values. As ν decreases, the wavefront transitions from a smooth curve to a sharp shock-like structure.

The profile at $t = 1.5$ represents a pure temporal extrapolation: the model was trained on the domain $t \in [0, 1]$. The prediction is obtained by evaluating the learned continuous function outside the training window, demonstrating the network's ability to generalize the physical dynamics beyond the observed time horizon.

7 Conclusions

The comparative analysis between L-BFGS and SGD revealed a fundamental trade-off intrinsic to learning optimization. The deterministic L-BFGS algorithm demonstrated superior convergence rates and numerical precision, consistently achieving the lowest training loss in the shortest wall clock time on CPU, and exhibiting good results on minimization of neural networks. However, this minimization often led to overfitting, as observed in the generalization gap on the Fashion-MNIST dataset. Conversely, Stochastic Gradient Descent (SGD), while computationally inefficient per epoch and characterized by noisy convergence trajectories, acted as a powerful implicit regularizer. It consistently provided the most robust test accuracy, confirming its status as the preferred choice for tasks where generalization capabilities outweigh pure fitting speed.

The implementation of Stochastic L-BFGS on the CPU backend proved to be a compelling middle ground. By effectively combining variance reduction with curvature information, it achieved the highest accuracy on the full Fashion-MNIST dataset, successfully mitigating the noise of SGD.

From a computational perspective, the scalability analysis highlighted the hardware constraints inherent to these algorithms. As evidenced by the weak scaling benchmarks, the performance of the CPU backend is strictly memory-bound, saturating the bandwidth at approximately 8 cores. This finding underscores the necessity of GPU acceleration for larger workloads, where our CUDA implementation demonstrated that the throughput benefits can outweigh the latency of data transfers.

Bibliography

[1] Tian-De Guo, Yan Liu, and Cong-Ying Han. An overview of stochastic quasi-newton methods for large-scale machine learning. 2023. Published online: 25 February 2023. (page 3.)

[2] Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent, 2026. Accessed: 2026-01-20. (page 4.)

[3] Limited-memory bfgs. https://en.wikipedia.org/wiki/Limited-memory_BFGS, 2026. Accessed: 2026-01-20. (page 4.)

[4] Philipp Moritz, Robert Nishihara, and Michael I. Jordan. Proceedings of the 19th international conference on artificial intelligence and statistics (aistats). pages 249–258. PMLR, 2016. (page 5.)

[5] Enzyme automatic differentiation framework. https://enzyme.mit.edu, 2026. Accessed: 2026-01-20. (page 21.)