BFGS vs L-BFGS: Optimization of a Neural Network on MNIST

MATH 320 — Optimization Project

Vance John Witlie Leslie Jasmine Moskowitz

December 2, 2025

Abstract

This document presents the full documentation, mathematical derivations, implementation details, and experimental results for a comparative study of the BFGS and L-BFGS optimization algorithms applied to training a single-hidden-layer neural network on the MNIST dataset. Each phase of the project is organized into its own `.tex` file and included here for clarity, structure, and reproducibility.

Contents

1 Phase 1: Problem Definition and Mathematical Setup

This phase establishes the mathematical and structural foundation of the project. We formally define the neural network model, the loss function, the parameter vectorization strategy, and the analytical gradient required for BFGS and L-BFGS optimization. This phase is critical to ensure correctness and stability of the optimization process in later stages.

1.1 Neural Network Architecture

We consider a single-hidden-layer neural network trained for *binary* classification between the digits 0 and 1 from the MNIST dataset. Each MNIST image is flattened into a vector $x \in \mathbb{R}^{784}$ and passed through the following layers:

- A hidden layer with H units and tanh activation,
- An output layer with 2 units (one for each class 0 and 1),
- A softmax function to produce class probabilities for the two classes.

The architecture is:

$$x \xrightarrow{W_1, b_1} h = \tanh(W_1 x + b_1) \xrightarrow{W_2, b_2} \hat{y} = \operatorname{softmax}(W_2 h + b_2).$$

1.2 Model Parameters

The trainable parameters of the model consist of two weight matrices and two bias vectors:

$$W_1 \in \mathbb{R}^{H \times 784}, \qquad b_1 \in \mathbb{R}^{H},$$

$$W_2 \in \mathbb{R}^{2 \times H}, \qquad b_2 \in \mathbb{R}^{2}.$$

The total number of parameters is:

$$p = H(784 + 1) + 2(H + 1).$$

1.3 Parameter Vectorization

Quasi-Newton methods such as BFGS and L-BFGS operate on a single parameter vector $w \in \mathbb{R}^p$. We therefore flatten the model parameters in the following consistent order:

$$w = \begin{bmatrix} \operatorname{vec}(W_1) \\ b_1 \\ \operatorname{vec}(W_2) \\ b_2 \end{bmatrix}.$$

This ordering is used throughout the project for both flattening and unflattening and must be respected in all routines.

1.4 Forward Pass

Given flattened parameters w, we reconstruct W_1, b_1, W_2, b_2 and compute the layer activations.

Hidden Layer

$$z_1 = W_1 x + b_1, \qquad h = \tanh(z_1).$$

Output Layer

$$z_2 = W_2 h + b_2.$$

The softmax function over the two output units is:

$$\hat{y}_k = \frac{\exp(z_{2k})}{\sum_{j=1}^{2} \exp(z_{2j})}, \qquad k = 1, 2.$$

1.5 Loss Function

For a single training example with binary one-hot label vector $y \in \mathbb{R}^2$, the cross-entropy loss is:

$$\ell(\hat{y}, y) = -\sum_{k=1}^{2} y_k \log(\hat{y}_k).$$

For a dataset of N samples,

$$E(w) = \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{y}_i(w), y_i).$$

1.6 Analytical Gradient

We derive the gradient of $E(w)$ via backpropagation.

Output Layer

$$\nabla_{z_2} \ell = \hat{y} - y.$$

$$\nabla_{W_2} E = \frac{1}{N} \sum_i (\hat{y}_i - y_i) h_i^\top, \qquad \nabla_{b_2} E = \frac{1}{N} \sum_i (\hat{y}_i - y_i).$$

Hidden Layer

Backpropagating through the output layer,

$$\nabla_h = W_2^\top(\hat{y} - y).$$

Using the derivative of $\tanh(z)$:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z),$$

we obtain:

$$\nabla_{z_1} = \nabla_h \odot (1 - h^2).$$

$$\nabla_{W_1} E = \frac{1}{N}\sum_i \nabla_{z_1}^{(i)} x_i^\top, \qquad \nabla_{b_1} E = \frac{1}{N}\sum_i \nabla_{z_1}^{(i)}.$$

1.7 Flattened Gradient

The full gradient $\nabla E(w)$ must match the parameter ordering:

$$\nabla E(w) = \begin{bmatrix} \operatorname{vec}(\nabla_{W_1}) \\ \nabla_{b_1} \\ \operatorname{vec}(\nabla_{W_2}) \\ \nabla_{b_2} \end{bmatrix}.$$

1.8 Gradient Verification

To ensure correctness, we use a numerical approximation based on central finite differences.

For perturbation $\varepsilon = 10^{-6}$,

$$g_j^{\text{num}} = \frac{E(w + \varepsilon e_j) - E(w - \varepsilon e_j)}{2\varepsilon},$$

where e_j is the j-th standard basis vector.

The gradient is considered correct if:

$$\frac{\|g^{\text{analytic}} - g^{\text{num}}\|}{\|g^{\text{analytic}}\| + \|g^{\text{num}}\|} < 10^{-4}.$$

1.9 Parameter Initialization

We initialize all weights using Xavier (Glorot) initialization, appropriate for tanh activations:

$$W \sim U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right).$$

Biases are initialized to zero:

$$b_1 = 0, \qquad b_2 = 0.$$

The full parameter vector is:

$$w_0 = \operatorname{vec}(W_1, b_1, W_2, b_2).$$

1.10 Final Optimization Problem

The learning task is formulated as the unconstrained optimization problem:

$$\min_{w \in \mathbb{R}^p} E(w),$$

where $E(w)$ is the binary cross-entropy loss over the filtered 0/1 subset of MNIST.

1.11 Deliverables Summary

At the completion of Phase 1, the following components are fully defined:

- Binary classification neural network architecture (digits 0 vs 1).
- Parameter shapes and consistent flattening/unflattening.
- Forward pass mappings.
- Cross-entropy loss function for 2-class softmax output.
- Full analytical gradient via backpropagation.
- Numerical gradient checker for verification.
- Xavier initialization for parameter vector w_0.
- The final optimization problem statement for binary classification.

2 Phase 2: Data Preparation and Preprocessing

This phase describes the precise procedure for loading, preprocessing, and structuring a *subset* of the MNIST dataset for use in training the binary classifier defined in Phase 1. We restrict attention to images of digits 0 and 1 only, so that the task becomes a two-class classification problem. Because quasi-Newton methods require full-batch deterministic gradients, it is essential that the data pipeline is consistent, reproducible, and fully vectorized.

2.1 Dataset Overview

The original MNIST dataset consists of grayscale images of handwritten digits 0 through 9:

- Training set: 60,000 images (all digits),
- Test set: 10,000 images (all digits),
- Image dimensions: 28×28,
- Pixel range: $\{0, \ldots, 255\}$.

In this project, we construct a *binary* dataset by filtering to include only samples whose labels are 0 or 1.

2.2 Filtering to Digits 0 and 1

Let the original dataset be $\{(X_i, y_i)\}$ with $y_i \in \{0, \ldots, 9\}$. We retain only those samples for which:

$$y_i \in \{0, 1\}.$$

The resulting subset is:

$$\{(X_i, y_i) : y_i \in \{0, 1\}\}.$$

2.3 Data Subsampling Strategy

Even after filtering, the number of 0/1 samples may still be relatively large for full-batch BFGS. To maintain tractability, we use:

$$N_{\text{train}} \approx 10{,}000, \qquad N_{\text{val}} = 2{,}000, \qquad N_{\text{test}} = \text{all remaining filtered test samples}.$$

Subsampling is performed with:

- a fixed random seed (reproducibility),
- shuffling prior to splitting,
- balanced proportions of class 0 and class 1 when possible.

2.4 Flattening and Normalization

Each image $X \in \mathbb{R}^{28\times 28}$ is reshaped into a vector:

$$x = \text{reshape}(X) \in \mathbb{R}^{784}.$$

Pixel values are normalized to the range $[0, 1]$:

$$x \leftarrow \frac{x}{255}.$$

This normalization helps keep the activations well-scaled for the tanh hidden layer.

2.5 Binary One-Hot Encoding of Labels

Since we only consider classes 0 and 1, each label y is converted into a one-hot vector in $\mathbb{R}^2$:

$$y = \begin{cases} (1,0) & \text{if the label is 0,} \\ (0,1) & \text{if the label is 1.} \end{cases}$$

This encoding matches the two-dimensional softmax output of the network from Phase 1 and is used in the binary cross-entropy loss.

2.6 Vectorized Dataset Representation

After preprocessing, the subsets are represented as matrices:

$$\begin{aligned} X_{\text{train}} &\in \mathbb{R}^{N_{\text{train}}\times 784}, & Y_{\text{train}} &\in \mathbb{R}^{N_{\text{train}}\times 2}, \\ X_{\text{val}} &\in \mathbb{R}^{N_{\text{val}}\times 784}, & Y_{\text{val}} &\in \mathbb{R}^{N_{\text{val}}\times 2}, \\ X_{\text{test}} &\in \mathbb{R}^{N_{\text{test}}\times 784}, & Y_{\text{test}} &\in \mathbb{R}^{N_{\text{test}}\times 2}. \end{aligned}$$

Each row corresponds to a single preprocessed 0/1 digit sample.

2.7 Deterministic Splitting and Reproducibility

To ensure repeatable experiments, we fix all random seeds at the beginning of the preprocessing pipeline, for example:

```
numpy.random.seed(seed),    random.seed(seed).
```

We shuffle the filtered (0/1) dataset once, maintaining label-image alignment, and then split into training, validation, and test sets.

2.8 Batching Strategy for Quasi-Newton Methods

Unlike stochastic gradient-based methods, BFGS and L-BFGS require full-batch gradients to produce stable curvature approximations. Therefore, during training:

$$X_{\text{batch}} = X_{\text{train}}, \qquad Y_{\text{batch}} = Y_{\text{train}}.$$

Mini-batches are not used in this project.

2.9 Sanity Checks

Before proceeding to model training, we perform several sanity checks:

- **Statistical check:** compute $\mathbb{E}[x]$ and $\mathrm{Var}(x)$ to confirm normalization.
- **Class balance:** verify that class 0 and class 1 occurrences are reasonably balanced in the train/validation/test splits.
- **Visualization:** display a few samples reshaped back to 28×28 to confirm flattening and normalization.

2.10 Data Storage Format

To support fast and repeatable experiments, we store all processed matrices for the 0/1 subset in NumPy format:

`X_train_01.npy`, `Y_train_01.npy`, `X_val_01.npy`, `Y_val_01.npy`, `X_test_01.npy`, `Y_test_01.npy`.

Using row-major layout ensures compatibility with vectorized forward passes such as:

$$Z_1 = XW_1^\top + b_1.$$

2.11 Phase Deliverables

At the conclusion of Phase 2, the following components are complete:

- MNIST dataset filtered to include only digits 0 and 1.
- All 0/1 images flattened and normalized.
- All 0/1 labels one-hot encoded in $\mathbb{R}^2$.
- Deterministic train/validation/test split created for the 0/1 subset.
- Vectorized dataset matrices stored in `.npy` format.
- Sanity checks performed to confirm data and label correctness.

This fully prepares the binary classification dataset for optimization in later phases.

3 Phase 3: Implementation of Optimization Algorithms

This phase describes the implementation of the optimization methods used in the project: BFGS and L-BFGS. These quasi-Newton methods require the objective function $E(w)$ and its gradient $\nabla E(w)$, fully derived in Phase 1, and the preprocessed binary MNIST dataset from Phase 2.

The goal of this phase is to construct robust, reusable optimization routines and an interface layer that allows both BFGS and L-BFGS to interact seamlessly with the neural network model.

3.1 Optimization Objective

The objective function is the binary cross-entropy loss defined in Phase 1:

$$E(w) = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{y}_i(w), y_i), \qquad \ell(\hat{y}, y) = -\sum_{k=1}^{2} y_k \log(\hat{y}_k).$$

The gradient is the flattened vector:

$$\nabla E(w) = \begin{bmatrix} \mathrm{vec}(\nabla_{W_1}) \\ \nabla_{b_1} \\ \mathrm{vec}(\nabla_{W_2}) \\ \nabla_{b_2} \end{bmatrix}.$$

Both BFGS and L-BFGS will use this same interface.

3.2 Objective Function Wrapper

To use the optimization routines provided by SciPy or a custom implementation, we define two callable functions:

$$\texttt{loss}(w) = E(w), \qquad \texttt{grad}(w) = \nabla E(w).$$

These functions:

- Unflatten w into (W_1, b_1, W_2, b_2),
- Perform a full forward pass over X_{train},
- Compute the cross-entropy loss,
- Compute the analytical gradient using backpropagation,
- Return flattened outputs.

3.3 BFGS Optimization Method

BFGS is a full-memory quasi-Newton method that maintains a dense approximation to the inverse Hessian matrix:

$$H_{k+1} = \left(I - \rho_k s_k y_k^\top\right) H_k \left(I - \rho_k y_k s_k^\top\right) + \rho_k s_k s_k^\top.$$

Here:

$$s_k = w_{k+1} - w_k, \qquad y_k = \nabla E(w_{k+1}) - \nabla E(w_k), \qquad \rho_k = \frac{1}{y_k^\top s_k}.$$

BFGS typically converges quickly, but its memory cost is $O(p^2)$, where p is the total parameter count. For neural networks, this can be large, motivating the use of L-BFGS (Section 3.4).

BFGS Algorithm (High-Level)

1. Initialize w_0 (from Phase 1) and set $H_0 = I$.
2. Evaluate $E(w_0)$ and $g_0 = \nabla E(w_0)$.
3. For $k = 0, 1, \ldots$:
   (a) Compute search direction:
   $$p_k = -H_k g_k.$$
   (b) Perform a line search to find step size α_k.
   (c) Update parameters:
   $$w_{k+1} = w_k + \alpha_k p_k.$$
   (d) Compute:
   $$s_k = w_{k+1} - w_k, \qquad y_k = g_{k+1} - g_k.$$
   (e) Update Hessian approximation H_{k+1} using BFGS formula.

3.4 L-BFGS Optimization Method

L-BFGS (Limited-memory BFGS) stores only the most recent m vector pairs (s_k, y_k), producing a memory footprint of $O(mp)$, instead of the $O(p^2)$ cost of BFGS.

This makes L-BFGS highly suitable for neural networks, where p is large.

The algorithm uses a two-loop recursion to compute the search direction:

$$H_k g_k \approx \text{L-BFGS-TwoLoop}(g_k, s_{k-m:k}, y_{k-m:k}).$$

The number of stored pairs m is typically between 5 and 20.

L-BFGS Two-Loop Recursion

Given gradient g_k and vectors s_i, y_i, L-BFGS computes the search direction without explicitly forming H_k:

```
q = g_k
for i = k-1 down to k-m:
    alpha_i = rho_i * s_i^T q
    q = q - alpha_i * y_i

H0 = (s_{k-1}^T y_{k-1}) / (y_{k-1}^T y_{k-1})   # scalar scaling
r = H0 * q

for i = k-m to k-1:
    beta_i = rho_i * y_i^T r
    r = r + s_i * (alpha_i - beta_i)

p_k = -r
```

The result p_k is the approximate Newton-like direction.

3.5 Line Search Strategy

Both BFGS and L-BFGS require a line search to ensure sufficient decrease in the objective.

We employ a backtracking line search satisfying the Armijo condition:

$$E(w_k + \alpha p_k) \leq E(w_k) + c\alpha \nabla E(w_k)^\top p_k,$$

with typical parameter $c = 10^{-4}$.

If SciPy's implementations are used, these line-search routines are automatic.

3.6 Stopping Criteria

Optimization terminates when any of the following conditions are satisfied:

- Gradient norm tolerance:

$$\|\nabla E(w_k)\| < \varepsilon_g, \quad \text{with } \varepsilon_g \approx 10^{-5}.$$

- Parameter update is small:

$$\|w_{k+1} - w_k\| < \varepsilon_w.$$

- Relative decrease in the loss is below threshold.
- Maximum number of iterations reached (e.g., 200).
- Maximum number of function evaluations exceeded.

3.7 Practical Considerations

- **Full-batch gradients only:** Mini-batches are incompatible with quasi-Newton curvature approximations.
- **Numerical stability:** Softmax is implemented with a shift:

$$z \leftarrow z - \max(z)$$

  to prevent overflow.
- **Gradient correctness:** The numerical gradient checker from Phase 1 must be run before any optimization to guarantee correctness.
- **Memory usage:** L-BFGS is substantially more memory-efficient than BFGS and is expected to scale better with increasing hidden-layer size H.

3.8 Phase Deliverables

At the conclusion of Phase 3, the following components are fully implemented:

- Objective function wrapper `loss`(w).
- Gradient function wrapper `grad`(w).
- Full BFGS implementation or SciPy-based equivalent.
- L-BFGS implementation using m stored curvature pairs.
- Two-loop recursion for the L-BFGS direction.
- Line search routine (built-in or custom Armijo backtracking).
- Stopping criteria for both algorithms.
- Unified interface for running optimization on the neural network model.

These components prepare the foundation for Phase 4, where the experimental design and evaluation metrics are developed.

4 Phase 4: Experimental Design and Evaluation Methodology

This phase establishes the experimental framework used to evaluate the performance of BFGS and L-BFGS on the binary MNIST (digits 0 and 1) classification task. A clear and reproducible methodology is essential for generating meaningful, scientifically interpretable comparisons between optimization algorithms. The phase specifies the evaluation metrics, experimental protocol, baselines, hyperparameter rules, and fairness considerations.

4.1 Overview of Experimental Goals

The primary objective of the experimental study is to compare the following:

- BFGS (full-memory quasi-Newton),
- L-BFGS (limited-memory quasi-Newton),
- Stochastic Gradient Descent (SGD) as a baseline optimizer.

Each method is evaluated with respect to:

1. Convergence speed (loss vs. iterations),
2. Classification accuracy,
3. Stability of optimization,
4. Computational cost (runtime and memory usage),
5. Sensitivity to hyperparameters.

4.2 Dataset and Train–Validation–Test Protocol

The dataset used is the binary MNIST subset constructed in Phase 2, containing only digits 0 and 1. The dataset is split as follows:

$$X_{\text{train}}, Y_{\text{train}} \quad \text{(training)}, \qquad X_{\text{val}}, Y_{\text{val}} \quad \text{(validation)}, \qquad X_{\text{test}}, Y_{\text{test}} \quad \text{(testing)}.$$

The experimental protocol ensures:

- All optimizers are evaluated using the same training and validation sets.
- The validation set is used for early stopping and hyperparameter selection.
- The test set is used only for the final reported results.

This ensures a clean separation between model selection and final evaluation.

4.3 Model Specification

All optimizers train the same neural network architecture from Phase 1:

$$x \in \mathbb{R}^{784} \xrightarrow{W_1,b_1} h = \tanh(W_1 x + b_1) \xrightarrow{W_2,b_2} \hat{y} = \mathrm{softmax}(W_2 h + b_2)$$

with output dimension 2 for binary classification.

The parameter vector w and gradient $\nabla E(w)$ follow the flattening and backpropagation rules from Phase 1.

4.4 Unified Optimization Interface

For consistency, each optimizer interacts with the model using the same two functions:

$$\texttt{loss}(w) = E(w), \qquad \texttt{grad}(w) = \nabla E(w),$$

where:

- $E(w)$ is the full-batch cross-entropy loss,
- $\nabla E(w)$ is the exact full-batch gradient,
- both functions internally unflatten and reflatten the parameter vector.

This eliminates implementation-specific discrepancies across optimizers.

4.5 Baseline Optimizer: Stochastic Gradient Descent (SGD)

To contextualize the performance of BFGS and L-BFGS, we include full-batch Stochastic Gradient Descent as a baseline.

SGD Update Rule

SGD performs the iterative update:

$$w_{k+1} = w_k - \eta_k \nabla E(w_k),$$

where η_k is the learning rate.

Learning Rate Schedule

We consider two possible learning-rate regimes:

- **Constant learning rate:** $\eta_k = \eta$;
- **Decay schedule:**
$$\eta_k = \frac{\eta_0}{1 + \gamma k}.$$

The learning rate is selected via validation-set performance.

Fairness Considerations

To ensure a fair comparison:

- SGD uses the *same full-batch gradient* as BFGS and L-BFGS, rather than mini-batches.
- SGD is initialized with the *same initial vector* w_0.
- The maximum number of iterations is matched to quasi-Newton runs.

The baseline provides insight into how much improvement quasi-Newton curvature information provides over simple first-order updates.

4.6 BFGS and L-BFGS Hyperparameter Configuration

BFGS

BFGS requires minimal hyperparameter tuning. The two main elements are:

- Initial inverse Hessian approximation $H_0 = I$,
- Line search parameters (Armijo constant c, step-size bounds).

Additionally, we consider:

$$\varepsilon_g = 10^{-5} \quad \text{(gradient norm tolerance).}$$

L-BFGS

L-BFGS requires selection of the memory parameter m, typically:

$$m \in \{5, 10, 20\}.$$

The same line-search parameters are used as in BFGS.
We also record:

- Memory usage as a function of m,
- Runtime differences relative to BFGS,
- Iteration counts.

4.7 Evaluation Metrics

To compare the optimizers, we compute:

- **Training loss:** $E(w_k)$ at each iteration.
- **Validation loss:** $E_{\text{val}}(w_k)$ for early stopping analysis.

- **Classification accuracy:**

$$\text{Accuracy} = \frac{\#\{\text{correct predictions}\}}{\#\{\text{samples}\}}.$$

- **Convergence speed:** number of iterations required to reach target loss.
- **Runtime:** wall-clock time for each optimizer.
- **Memory usage:** particularly relevant for L-BFGS.

We also log:

- gradient norms,
- step sizes α_k,
- stability indicators (oscillation, divergence, plateaus).

4.8 Reproducibility and Logging

To ensure reproducibility:

- All random seeds are fixed,
- All dataset splits are deterministic,
- The initial parameter vector w_0 is saved and reused across optimizers,
- Loss curves and accuracy curves are saved to disk,
- All optimizer configurations are logged in a machine-readable format, such as JSON.

The experiment code records:

- iteration-by-iteration loss values,
- validation accuracy,
- timestamps,
- curvature pair counts for L-BFGS.

4.9 Phase Deliverables

At the conclusion of Phase 4, the following components are completed:

- Full experimental design and protocol defined,
- SGD baseline introduced and fully specified,
- Hyperparameter search strategy for each optimizer,
- Evaluation metrics selected and formalized,
- Unified optimizer interface established,
- Reproducibility protocol defined,
- Logging and result-recording procedures determined.

This phase provides the framework for Phase 5, where these experiments are implemented in code and executed to gather results.

5 Phase 5: Implementation of Experiments

This phase details how the optimization experiments are executed in practice. Whereas Phase 4 established the experimental design and evaluation methodology, the present phase specifies the concrete implementation steps required to run BFGS, L-BFGS, and SGD on the binary MNIST classification task. It includes the algorithmic workflow, logging, reproducibility mechanisms, and code structure for running large-scale experiments in a systematic and repeatable manner.

5.1 Experiment Execution Overview

Each experiment consists of the following standard pipeline:

1. Load preprocessed datasets from Phase 2,
2. Load or construct the initial parameter vector w_0,
3. Select an optimizer (BFGS, L-BFGS, or SGD),
4. Run optimization on the full training set,
5. Evaluate performance on the validation and test sets,
6. Save results, accuracy, loss curves, and logs,
7. Repeat for each optimizer under identical conditions.

This ensures comparability across optimization methods.

5.2 Experiment Configuration Files

To promote reproducibility and organization, all experiments use a centralized configuration file (e.g., `config.json`) containing:

- hidden-layer dimension H,
- optimizer type (`BFGS`, `LBFGS`, `SGD`),
- learning-rate settings for SGD,
- L-BFGS memory parameter m,
- stopping criteria,
- dataset paths,
- logging directory,
- random seeds,
- maximum number of iterations and function evaluations.

This allows the entire experimental pipeline to be controlled by a single, human-readable configuration file.

5.3 Experiment Directory Structure

The project uses the following experiment directory layout:

```
experiments/
   BFGS/
      config.json
      logs/
      results/
   LBFGS_m10/
      config.json
      logs/
      results/
   SGD_lr0.1/
      config.json
      logs/
      results/
```

Each optimizer run stores:

- **loss curves** (training and validation),
- **accuracy curves**,
- **timestamps**,
- **wall-clock runtime**,
- **hyperparameters used**,
- **final test metrics**.

5.4 Initialization Consistency

To ensure a fair comparison, all optimizers start from the *exact same* initial parameter vector:

$$w_0 = \mathrm{vec}(W_1, b_1, W_2, b_2).$$

This vector is generated once (Phase 1) and saved to disk:

`initial_w.npy`.

Any optimizer that begins from a different initial point would make results incomparable.

5.5 Unified Optimization Loop

The high-level optimization loop is identical across BFGS, L-BFGS, and SGD:

```
load X_train, Y_train
load initial w0

w = w0
for k in 0, 1, 2, ...:
    loss_k = loss(w)
    grad_k = grad(w)

    record(loss_k, grad_k, time, etc.)

    if stopping_criteria_met:
        break

    p_k = compute_search_direction(optimizer, w, grad_k)
    alpha_k = line_search(optimizer, w, p_k)
    w = w + alpha_k * p_k
```

In the case of SGD, the direction is simply:

$$p_k = -\nabla E(w_k), \qquad \alpha_k = \eta_k.$$

For BFGS and L-BFGS, the direction comes from quasi-Newton curvature estimates.

5.6 Logging During Optimization

At each iteration we store:

- training loss $E(w_k)$,
- validation loss $E_{\text{val}}(w_k)$,
- gradient norm $\|\nabla E(w_k)\|$,
- step size α_k,
- search direction norm $\|p_k\|$,
- iteration timestamp,
- memory usage (for L-BFGS).

This information is crucial for Phase 6, where convergence behavior and performance are analyzed.

5.7 Runtime Measurement

For each optimizer run, we measure:

$$T = \text{wall-clock time from iteration 0 to the final iteration.}$$

Python's high-resolution timer (e.g. `time.perf_counter()`) is used.
Runtime is treated as one of the main evaluation metrics.

5.8 Validation-Based Early Stopping

To prevent overfitting and unnecessary computation, we include early stopping criteria based on validation loss:

$$E_{\text{val}}(w_{k+1}) > E_{\text{val}}(w_k) \quad \text{for several consecutive iterations,}$$

or:

$$|E_{\text{val}}(w_{k+1}) - E_{\text{val}}(w_k)| < \varepsilon_{\text{val}}.$$

This mechanism applies uniformly to all optimizers.

5.9 Computing Test Accuracy

After optimization terminates, we evaluate final performance on the test set.

Given predictions:

$$\hat{y} = \text{softmax}(W_2 h + b_2),$$

where the predicted class is:

$$\hat{c} = \arg\max_k \hat{y}_k,$$

we compute:

$$\text{Test Accuracy} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \mathbb{I}[\hat{c}_i = c_i].$$

Test results are recorded separately from training/validation logs to preserve the integrity of the experimental protocol.

5.10 Batching and Full-Batch Gradients

Because BFGS and L-BFGS require curvature information, we use:

$$X_{\text{batch}} = X_{\text{train}}, \qquad Y_{\text{batch}} = Y_{\text{train}}.$$

For fairness, SGD also uses full-batch gradients (rather than mini-batches), making the comparison clean and scientifically valid.

5.11 Saving Final Model Parameters

Each optimizer saves its final parameter vector:

$$w_{\text{final}} \in \mathbb{R}^p,$$

to a file such as:

`results/final_w.npy`.

These vectors are used later for:

- sanity-checking,
- visualization of decision boundaries (optional),
- comparison of norms and step lengths,
- reproducing test-set evaluations.

5.12 Safety Checks Before Running Experiments

Prior to running large experiments, the following checks are executed:

- **Gradient correctness:** numerical gradient check passes.
- **Forward-pass validation:** output probabilities sum to 1.
- **Loss monotonicity sanity check:** early BFGS/L-BFGS steps reduce loss.
- **Matrix dimension consistency:** reshaping matches flattening order.
- **Memory availability:** BFGS's $p \times p$ matrix fits in RAM (smaller H if required).

5.13 Phase Deliverables

At the conclusion of Phase 5, the following components are fully implemented:

- Central experiment configuration system,
- Directory structure for results, logs, and hyperparameter files,
- Initialization of w_0 and loading of datasets,
- Unified optimization loop for three optimizers,
- Full logging of loss, accuracy, runtime, and gradients,
- Saving of final models and experiment metadata,
- Early stopping and safety checks,
- Reproducible, automated experiment execution.

This implementation framework enables Phase 6, where the results are analyzed and visualized.

6 Phase 6: Results, Analysis, and Visualization

This phase presents the empirical results obtained from running BFGS, L-BFGS, and SGD on the binary MNIST classification problem. It includes convergence curves, performance metrics, runtime and memory comparisons, and an analytical interpretation of the observed optimization behavior. The goal is to understand how the optimizers differ in speed, stability, efficiency, and final accuracy.

All results in this section were generated using the experiment implementation pipeline developed in Phase 5.

6.1 Data Sources for Analysis

The following recorded outputs from each optimizer were used:

- training loss over iterations,
- validation loss over iterations,
- gradient norms,
- test-set accuracy,
- wall-clock runtime,
- iteration timestamps,
- memory usage (especially for L-BFGS),
- final parameter vectors w_{final}.

These are stored in:

```
experiments/<optimizer>/logs/
experiments/<optimizer>/results/
```

6.2 Convergence of Training Loss

For each optimizer, we plot the sequence:

$$E(w_0),\ E(w_1),\ E(w_2),\ \ldots,\ E(w_K).$$

Typical figures include:

- **Figure 1:** Training loss vs. iteration (linear scale),
- **Figure 2:** Training loss vs. iteration (log scale),
- **Figure 3:** Gradient norm vs. iteration.

The curves allow visual comparison of:

- initial rate of improvement,
- asymptotic convergence speed,
- smoothness / oscillation behavior,
- stagnation or divergence, if any.

6.3 Validation Loss and Early Stopping Behavior

Validation curves:

$$E_{\text{val}}(w_0),\ E_{\text{val}}(w_1),\ \ldots$$

are analyzed to determine:

- risk of overfitting,
- early stopping points,
- optimizer stability.

We also record the iteration k^* at which the minimum validation loss occurs:

$$k^* = \arg\min_k E_{\text{val}}(w_k).$$

Each optimizer is evaluated not only at its last iteration, but also at the validation-optimal iteration.

6.4 Runtime Comparison

We compute wall-clock runtime:

$$T_{\text{opt}} = t_{\text{end}} - t_{\text{start}}$$

for each optimizer.

A summary table is produced:

Optimizer	**Runtime (s)**	**Iterations**	**Fn. Evaluations**
BFGS	—	—	—
L-BFGS	—	—	—
SGD	—	—	—

Trends typically include:

- BFGS: highest runtime per iteration due to dense Hessian updates,
- L-BFGS: significantly faster and more memory-efficient,
- SGD: many iterations, but very cheap per iteration.

6.5 Memory Usage Analysis

Memory consumption is estimated based on:

- size of stored curvature pairs s_k, y_k for L-BFGS,
- storage of the full inverse Hessian for BFGS,
- minimal memory requirements for SGD.

Theoretical estimates:

$$\text{BFGS memory} \sim O(p^2), \qquad \text{L-BFGS memory} \sim O(mp), \qquad \text{SGD memory} \sim O(p).$$

We summarize results in a qualitative table:

Optimizer	**Memory Requirement**	**Scales to Large p?**
BFGS	High ($O(p^2)$)	No
L-BFGS	Moderate ($O(mp)$)	Yes
SGD	Low ($O(p)$)	Yes

6.6 Test Accuracy Results

For each optimizer, we compute:

$$\text{Accuracy}_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \mathbb{I}[\hat{c}_i = c_i].$$

Results are summarized in a table:

Optimizer	**Test Accuracy**	**Validation Iteration**
BFGS	—	—
L-BFGS	—	—
SGD	—	—

Important comparisons include:

- whether BFGS/L-BFGS achieve higher accuracy than SGD,
- sensitivity of accuracy to early stopping,
- consistency across multiple random seeds.

6.7 Convergence Behavior: Qualitative Analysis

By examining the curves and logs, we analyze:

- **Smoothness:** Quasi-Newton updates typically yield smooth monotone convergence.
- **Oscillations:** SGD may exhibit oscillation or plateaus.
- **Step-size patterns:** BFGS usually selects larger, confident steps once curvature is accurate.
- **Stability:** L-BFGS stability depends on the memory parameter m.
- **Gradient norms:** Faster reduction indicates more effective descent directions.

These observations form the basis for the narrative interpretation in Phase 7.

6.8 Visualization Guidelines

The following figures are included in the final report:

- Loss curve comparison (linear),
- Loss curve comparison (log scale),
- Gradient norm comparison,
- Validation loss curves,
- Test accuracy bar plot,
- Runtime bar plot,
- Memory requirements diagram.

All figures include:

- axis labels,
- legends,
- titles,
- consistent color schemes,
- small font for tick labels to aid readability.

6.9 Interpretation and Summary of Findings

The analysis addresses:

- Which optimizer converges fastest,
- Which optimizer achieves the lowest loss,
- Stability of optimization trajectories,
- Relative efficiency of curvature-based methods,
- Whether L-BFGS approximates BFGS sufficiently well,
- Whether curvature information provides meaningful advantages over SGD,
- Situations in which SGD may outperform quasi-Newton methods (e.g., robustness, simplicity, low memory footprint).

The findings of this phase directly inform the broader discussion, limitations, and conclusions presented in Phase 7.

6.10 Phase Deliverables

At the conclusion of Phase 6, the following deliverables are complete:

- Training loss curves for all optimizers,
- Validation loss curves and early-stopping iteration identification,
- Test accuracy results for all optimizers,
- Runtime and memory-consumption comparisons,
- Visualizations and summary tables,
- Qualitative and quantitative analysis of optimization behavior,
- Consolidated interpretation of results.

7 Phase 7: Discussions and Conclusion

7.1 Discussion

TODO

7.2 Conclusions

TODO