STA378 Final Report

Felix Gao

September 2025

1 Abstract

Optimization methods such as L-BFGS rely on hyperparameters that strongly influence convergence. In practice, these values are often chosen by rules of thumb or trial-and-error, which may lead to suboptimal performance on different problem types. We investigate a machine-learning approach to systematically select L-BFGS hyperparameters based on problem characteristics. We extract analytical features (e.g., problem size, sparsity) and empirical features (e.g., condition number estimate, initial gradient norm, intial objective evaluation), and train a machine-learning model that predicts the optimal memory parameter *mem*. Experiments on both regular and scalable problems show that our approach improves runtime and reduces memory compared to standard default settings. This work highlights the potential of learning-based parameter selection to enhance classical optimization algorithms.

2 Background and Introduction

2.1 Unconstrained Optimization Algorithms

In the field of continuous numerical optimization, the aim is to develop solvers that can locate a local minimum of unconstrained optimization problems:

$$\min_{x \in \mathbb{R}^n} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and using first-order (gradient vector of size n) and eventually using second-order derivatives (Hessian matrix - symmetric matrix of size n) of the objective function f. There exists a variety of algorithms for such problem and you can find a list of implementations in the Julia package JSOSolver.jl. These algorithms are iterative algorithms that start from an initial guess $x_0 \in \mathbb{R}^n$ and compute a follow-up iterate until a stationary point is reached, i.e, $\nabla f(x) \approx 0$.

2.2 Line Search Methods

Line search methods form a broad class of iterative optimization algorithms in which, at each iteration, a search direction $\mathbf{p}_k$ is computed and a suitable step length $\alpha_k > 0$ is chosen to determine how far to move along that direction. The general update rule is given by

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

where α_k is referred to as the *step length* or *learning rate*.

The performance of a line search method depends critically on two components: the choice of the search direction $\mathbf{p}_k$ and the strategy used to determine α_k. Typical search directions include the negative gradient direction used in the Steepest Descent method and the Newton direction in second-order methods.

An ideal step length achieves a sufficient reduction in the objective function while maintaining numerical stability. In practice, α_k is often determined through a line search procedure that satisfies certain conditions, such as the *Wolfe* or *Armijo* conditions, which ensure both sufficient decrease and appropriate curvature. These conditions balance convergence speed with robustness, preventing steps that are excessively short or that overshoot the minimum.

2.3 Steepest Descent (Gradient Descent)

The Steepest Descent method, also known as Gradient Descent, is an iterative optimization algorithm used to find a local minimum of a differentiable function.

We consider a scalar-valued objective function

$$f : \mathbb{R}^n \to \mathbb{R}, \quad \mathbf{w} \mapsto f(\mathbf{w}),$$

and aim to find the point $\mathbf{w}^*$ that minimizes $f(\mathbf{w})$.

The procedure can be summarized as follows:

- Initialize with an initial guess $\mathbf{w}_0 \in \mathbb{R}^n$.
- At each iteration $k = 0, 1, 2, \ldots$, compute the gradient $\nabla f(\mathbf{w}_k)$.
- Update the parameter vector according to

  $$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla f(\mathbf{w}_k),$$

  where $\alpha_k > 0$ is the step size (or learning rate).

The process is repeated until a convergence criterion is met. In theory, convergence is achieved when the gradient norm becomes sufficiently small, i.e., $\|\nabla f(\mathbf{w}_k)\| < \epsilon$. In practice, termination is often based on a combination of factors such as a maximum number of iterations, a small relative change in $f(\mathbf{w})$, or when the gradient norm falls below a predefined threshold.

To minimize the function $F(w)$, we use the update rule

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} F(\mathbf{w})$$

2.4 Newton's Method

Newton's Method is a second-order iterative optimization algorithm that uses both the gradient and the Hessian of the objective function to determine the search direction. It can achieve significantly faster convergence compared to first-order methods such as Steepest Descent, particularly near the optimal point.

We consider a twice continuously differentiable scalar-valued function

$$f : \mathbb{R}^n \to \mathbb{R}.$$

At iteration k, Newton's Method computes the search direction $\mathbf{p}_k^N$ by solving the linear system

$$\nabla^2 f(\mathbf{x}_k)\, \mathbf{p}_k^N = -\nabla f(\mathbf{x}_k),$$

where $\nabla f(\mathbf{x}_k)$ is the gradient and $\nabla^2 f(\mathbf{x}_k)$ is the Hessian matrix of second derivatives. The parameter vector is then updated as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k^N,$$

where $\alpha_k > 0$ is the step size, often obtained by a line search procedure to ensure sufficient decrease in f.

In an ideal case with exact line search, $\alpha_k = 1$ is typically accepted for all sufficiently large k, and the method achieves *quadratic convergence*. Formally, if the Hessian $\nabla^2 f(\mathbf{x})$ is Lipschitz continuous in a neighborhood of the solution $\mathbf{x}^*$, and if $\nabla^2 f(\mathbf{x}^*)$ is positive definite, then:

1. For a starting point $\mathbf{x}_0$ sufficiently close to $\mathbf{x}^*$, the sequence $\{\mathbf{x}_k\}$ converges to $\mathbf{x}^*$.
2. The convergence rate is quadratic; that is,

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq C\|\mathbf{x}_k - \mathbf{x}^*\|^2,$$

   for some constant $C > 0$.
3. The gradient norms $\|\nabla f(\mathbf{x}_k)\|$ also converge quadratically to zero.

Despite its theoretical efficiency, Newton's Method can be computationally expensive for high-dimensional problems, since it requires forming and inverting the Hessian matrix at every iteration. Moreover, if the Hessian is not positive definite, the computed direction $\mathbf{p}_k^N$ may fail to be a descent direction, potentially leading to divergence. In practice, modified Newton methods or quasi-Newton algorithms (such as BFGS and L-BFGS) are employed to approximate curvature information efficiently while maintaining desirable convergence behavior.

2.5 Quasi-Newton Methods

Optimization methods such as steepest descent and Newton's Method are widely used in scientific computing to find local minima of differentiable functions. However, for large-scale problems, these methods can become computationally expensive and memory-intensive. Quasi-Newton methods, similar to the steepest descent approach, require only the gradient of the objective function at each iteration. By observing how the gradient changes between steps, these methods build an approximate model of the objective function that is accurate enough to achieve superlinear convergence. Compared to steepest descent, their improvement in speed and stability is substantial, particularly on challenging problems. Because they avoid computing second derivatives, quasi-Newton methods can even be more efficient than full Newton's Method. Modern optimization libraries now include many variants of quasi-Newton algorithms designed for unconstrained, constrained, and large-scale optimization problems.

2.6 Implementations of Quasi-Newton Methods

- The BFGS Method

Algorithm 6.1 (BFGS Method).
Given starting point x_0, convergence tolerance $\epsilon > 0$,
inverse Hessian approximation H_0;
$k \leftarrow 0$;
while $\|\nabla f_k\| > \epsilon$;
Compute search direction

$$p_k = -H_k \nabla f_k; \tag{6.18}$$

Set $x_{k+1} = x_k + \alpha_k p_k$ where α_k is computed from a line search
procedure to satisfy the Wolfe conditions (3.6);
Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;
Compute H_{k+1} by means of (6.17);
$k \leftarrow k + 1$;
end (while)

Figure 1: BFGS Algorithm (adapted from Nocedal & Wright, 2006)

- The L-BFGS Method

Algorithm 7.5 (L-BFGS).
Choose starting point x_0, integer $m > 0$;
$k \leftarrow 0$;
repeat
Choose H_k^0 (for example, by using (7.20));
Compute $p_k \leftarrow -H_k \nabla f_k$ from Algorithm 7.4;
Compute $x_{k+1} \leftarrow x_k + \alpha_k p_k$, where α_k is chosen to
satisfy the Wolfe conditions;
if $k > m$
Discard the vector pair $\{s_{k-m}, y_{k-m}\}$ from storage;
Compute and save $s_k \leftarrow x_{k+1} - x_k$, $y_k = \nabla f_{k+1} - \nabla f_k$;
$k \leftarrow k + 1$;
until convergence.

Figure 2: L-BFGS Algorithm (adapted from Nocedal & Wright, 2006)

Algorithm 7.4 (L-BFGS two-loop recursion).

$q \leftarrow \nabla f_k$;
for $i = k-1, k-2, \ldots, k-m$
  $\alpha_i \leftarrow \rho_i s_i^T q$;
  $q \leftarrow q - \alpha_i y_i$;
end (for)
$r \leftarrow H_k^0 q$;
for $i = k-m, k-m+1, \ldots, k-1$
  $\beta \leftarrow \rho_i y_i^T r$;
  $r \leftarrow r + s_i(\alpha_i - \beta)$
end (for)
stop with result $H_k \nabla f_k = r$.

Figure 3: L-BFGS two-loop recursion (adapted from Nocedal & Wright, 2006)

2.7 Rate of Convergence Analysis

The convergence behavior of an optimization algorithm measures how quickly its iterates approach the optimal solution. Designing algorithms with both good global convergence properties and rapid local convergence can be challenging, as these objectives often conflict. For example, the Steepest Descent method guarantees global convergence under mild assumptions but tends to converge slowly in practice, especially on ill-conditioned problems. In contrast, Newton's Method achieves much faster local (quadratic) convergence but may fail globally if the initial point is far from the optimum or if the Hessian is not positive definite.

To study convergence quantitatively, consider the quadratic objective function

$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{b}^T\mathbf{x},$$

where Q is symmetric and positive definite. For this problem, the gradient is $\nabla f(\mathbf{x}) = Q\mathbf{x} - \mathbf{b}$, and the minimizer $\mathbf{x}^*$ satisfies $Q\mathbf{x}^* = \mathbf{b}$. When the Steepest Descent method is applied with an exact line search, the optimal step size at iteration k is

$$\alpha_k = \frac{\nabla f(\mathbf{x}_k)^T \nabla f(\mathbf{x}_k)}{\nabla f(\mathbf{x}_k)^T Q \nabla f(\mathbf{x}_k)}.$$

The resulting iteration can be expressed as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k).$$

It can be shown that the decrease in the objective function satisfies the inequality

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_Q \leq \frac{\kappa(Q) - 1}{\kappa(Q) + 1} \|\mathbf{x}_k - \mathbf{x}^*\|_Q,$$

where $\|\mathbf{x}\|_Q = \sqrt{\mathbf{x}^T Q \mathbf{x}}$ and $\kappa(Q)$ is the *condition number* of Q, defined as the ratio of its largest to smallest eigenvalue.

This result implies that Steepest Descent converges *linearly*, and its rate depends heavily on the condition number of the Hessian. If $\kappa(Q)$ is large (i.e., Q is ill-conditioned), the iterates tend to "zigzag" toward the solution and convergence becomes slow. This inefficiency motivates second-order methods such as Newton's Method and Quasi-Newton algorithms, which exploit curvature information to improve convergence speed.

3 Methodology and Implementation

3.1 Benchmark Suite Construction

The benchmark problems were drawn from the *OptimizationProblems.jl* package. To focus on meaningful instances of smooth, large-scale optimization, the initial benchmark selection was restricted to problems that were *unconstrained*, *free of bound constraints*, and of dimension at least five. These criteria exclude trivial toy problems and ensure that curvature approximation and memory effects in L-BFGS are non-negligible. Formally, the first-stage problem set was obtained by filtering according to

$$\texttt{contype == :unconstrained}, \qquad \neg\texttt{has\_bounds}, \qquad \texttt{nvar} \geq 5.$$

For the scalable phase of the benchmark, we focused on problems whose dimension can be chosen programmatically at construction time. In the metadata of the *OptimizationProblems.jl* suite, this capability is indicated by the flag `variable_nvar = true`, which serves as a direct marker of scalability. Since the project centers on unconstrained smooth optimization, we retained only those problems explicitly classified as unconstrained and free of bound constraints. Although `meta.contype = :unconstrained` already implies the absence of general constraints, we additionally filtered for `ncon = 0` as a defensive measure to ensure that no equality or inequality constraints were included due to potential inconsistencies in the metadata. This refined subset of scalable, unconstrained, bound-free problems constituted the core of the large-scale experiments.

3.2 Solver Configuration

All optimization experiments were performed using the L-BFGS implementation from *JSOSolvers.jl*. Since the primary objective of the project was to understand how the limited-memory parameter affects performance across a wide range of problem instances, the memory values were not chosen manually. Instead, for each problem we queried the problem-specific L-BFGS parameter space provided by `LBFGSParameterSet`, and exhaustively evaluated every admissible memory value between the lower and upper bounds of the domain `param_set.mem`. This ensured that the experiments captured the full range of memory configurations that the solver considers meaningful for each problem.

Although several other algorithms (including trust-region solvers) were explored informally during the development phase to better understand the solver landscape within the *JuliaSmoothOptimizers* ecosystem, the final benchmarking results focus exclusively on L-BFGS to maintain a consistent and interpretable comparison across problems. All runs used the default algorithmic settings

supplied by *JSOSolvers*, with the exception of the line search. Because some problems may exhibit pathological or slow line-search behaviour, a wall-clock time limit of 60 seconds was imposed on each run to prevent individual cases from dominating the overall computation time.

For each problem and each memory value, the solver was initialized from the same starting point to ensure a controlled comparison. Before running the solver, we also explicitly evaluated the initial objective and gradient, allowing us to separately record their computation time, memory usage, and allocation statistics. These measurements were included in the final dataset to facilitate a deeper analysis of initialization overhead and objective/gradient evaluation behaviour across problems. All remaining solver parameters were left at their default values to isolate the impact of the memory parameter as the sole variable under investigation.

3.3 Experimental Protocol

The experimental workflow was designed to produce consistent and reproducible benchmarks across both fixed-dimension and scalable problem families. For problems supporting an arbitrary variable dimension, we constructed instances at several increasing sizes (e.g., n = 1,000, 10,000, and higher whenever permitted by the available HPC resources). These settings allow us to characterize how the computational behaviour of L-BFGS evolves as the dimension increases, complementing the fixed-dimension benchmarks from the first part of the project. In addition to the full memory sweep performed for each scalable instance, a secondary experiment was conducted on a representative problem in which the dimension was systematically varied while holding the memory parameter fixed. This provided a clearer picture of dimension-driven scaling behaviour independent of the limited-memory parameter.

For every problem instance, the solver was always initialized from the same starting point, and each run was subject to a fixed wall-clock time limit to prevent divergent or pathological cases from dominating total runtime. Because Julia's JIT compilation can introduce substantial one-time overhead, especially for large-scale problems, each experiment included a warm-up evaluation phase whose results were discarded to ensure that reported timings reflect steady-state solver performance. Throughout the experiments, objective and gradient evaluations were also measured independently of L-BFGS iterations, allowing us to track how fundamental evaluation costs scale with problem dimension.

For very large instances, particularly those approaching $n = 10^5$ and beyond, the memory requirements of the model evaluation and matrix-free operations exceeded the available RAM on the 32 GB HPC nodes, causing early termination or timeouts. These resource-induced failures were recorded explicitly, as they provide practical insight into the scalability limits of both the underlying

problems and the L-BFGS implementation. All results were logged incrementally to CSV files to ensure robustness during long-running experiments.

3.4 Performance Metrics

A comprehensive set of performance metrics was recorded for every solver run to enable a multi-faceted analysis of L-BFGS behaviour across different memory settings and problem dimensions. For each run, we logged standard optimization metrics including the final solver status, number of iterations, number of objective evaluations, number of gradient evaluations, and the number of matrix-free vector–product operations (`nvmops`). These quantities provide insight into both convergence behaviour and the computational effort required by the algorithm.

In addition to iteration-level metrics, we recorded detailed timing and memory information. Each call to L-BFGS was timed end-to-end, and the reported runtime includes both the solver iterations and the initial model construction cost. Peak memory usage was measured in megabytes to assess the interaction between problem dimension, memory parameter, and algorithmic overhead. To better understand initialization costs, the objective and gradient were evaluated once prior to the L-BFGS call, and their evaluation time, memory usage, and allocation counts were logged separately. These initialization statistics are particularly important for large-scale problems where evaluation cost may dominate solver iterations.

All measurements were logged incrementally to CSV files, with each row representing a single solver run and each column corresponding to a recorded metric. Separate CSV files were generated for the fixed dimension benchmarks and the scalable problem set, allowing the experiments to run independently and recover gracefully from long execution times or interrupted HPC sessions. For downstream analysis, including exploratory statistics, factor analysis, PCA conducted with collaborators, and the construction of several classifiers for predictive modelling, the completed CSV files were merged into a unified dataset. One of the large scale experiments is still running, and its results will be incorporated once the computation completes.

A complete description of all recorded metrics is provided in Table 1. The table summarizes every column in the benchmarking dataframe, including solver status codes, problem identifiers, memory parameters, dimensionality, iteration statistics, objective and gradient evaluation counts, as well as detailed timing, allocation, and memory measurements. These fields together form the core dataset used in the subsequent analysis, enabling comparisons across memory settings, problem dimensions, and solver outcomes.

3.5 Expression Tree Analysis

In addition to benchmarking solver performance, we also examined the structural complexity of the objective functions by analyzing their symbolic expression trees. For every unconstrained and bound free problem of dimension at least five, we converted the Symbolics.jl representation of the objective into an expression tree using the ExpressionTreeForge.jl framework. Two structural metrics were computed for each tree. The first is the tree length, defined recursively as the total number of nodes, where leaf nodes contribute a count of one and internal nodes contribute the sum of the lengths of their children. The second is the tree depth, defined as the maximum root to leaf path length, with the root assigned depth one. These metrics provide a coarse but informative measure of algebraic and compositional complexity, offering insight into how problem structure may influence solver behaviour. The resulting dataset, which includes the problem name, dimension, objective type, indicator of arbitrary variable dimension, tree length, and tree depth, was stored in a separate CSV file and later merged with the solver based benchmarking data to support a richer comparative analysis.

Column	Type	Description
`status`	Symbol	**first_order**: solver successfully reached a first-order stationary point; **max_time**: solver couldn't solve the problem within the time limit; **unbounded**: the minimum of the objective function goes to $-\infty$.
`name`	String	Identifier for the optimization problem.
`solver`	String	Name of the solver/algorithm applied.
`mem`	Int	Hyperparameter of the solver.
`nvar`	Int	Number of decision variables in the problem.
`time`	Float64	Total time(s) for solving the problem.
`memory`	Float64	Total memory(MB) used for solving the problem.
`num_iter`	Int	Total number of iterations performed.
`nvmops`	Int	Number of vector–matrix products.
`neval_obj`	Int	Number of objective function evaluations.
`init_eval_obj_time`	Float64	Time (seconds) for the initial objective evaluation.
`init_eval_obj_mem`	Float64	Memory (MB) for the initial objective evaluation.
`init_eval_obj_alloc`	Float64	Number of heap allocations for the initial objective evaluation.
`neval_grad`	Int	Number of gradient evaluations.
`init_eval_grad_time`	Float64	Time (seconds) for the initial gradient evaluation.
`init_eval_grad_mem`	Float64	Memory (MB) for the initial gradient evaluation.
`init_eval_grad_alloc`	Float64	Number of heap allocations for the initial gradient evaluation.
`is_init_run`	Bool	Indicates whether this is the initial run.

Table 1: Description of columns in the solver benchmark data table.

4 Results

5 Conclusions

6 Future work