A Machine Learning Approach for HyperParameter Tuning of Unconstrained Optimization Solvers

Final Project Report

Felix Gao
Department of Mathematical and Computational Sciences
University of Toronto

Contents

Abstract

Optimization methods such as L-BFGS rely on hyperparameters that strongly influence convergence. In practice, these values are often chosen by heuristics 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, memory usage) and empirical features (e.g., initial objective evaluation, initial gradient evaluation), and train a machine-learning model that predicts the optimal memory parameter *mem*. As *mem* increases, the memory used by the algorithm increases as well. This introduces a fundamental trade-off where increasing *mem* yields a more accurate curvature approximation that potentially reduces the total iteration count, but it proportionally increases memory consumption and per-iteration computational overhead. 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.

1 Background and Introduction

1.1 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, $\nabla f(x) \approx 0$.

1.2 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.

1.3 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.

1.4 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.

1.5 Implementations of Quasi-Newton Methods

Algorithm 1 (BFGS Method).

Given starting point x_0, convergence tolerance $\epsilon > 0$,
    inverse Hessian approximation H_0;
$k \leftarrow 0$;
while $\|\nabla f_k\| > \epsilon$ **do**
    Compute search direction
    $p_k = -H_k \nabla f_k$; (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
end (while)

Algorithm 2 (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 $k > m$ **then**
        Discard the vector pair $\{s_{k-m}, y_{k-m}\}$ from storage;
    end if
    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.

Algorithm 3 (L-BFGS two-loop recursion).

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

1.6 Expression Tree

An expression tree is a standard data structure in computer science for representing algebraic expressions in a form that makes their structure explicit. Each internal node encodes an operator (for example addition, multiplication, or a nonlinear function), and each leaf encodes an operand such as a variable or constant. This representation induces a natural notion of complexity: the tree length corresponds to the total number of nodes in the expression, while the tree depth corresponds to the maximum number of nested operations along any root to leaf path. Because these quantities reflect how many operations are composed and how deeply they are nested, they provide compact, problem specific descriptors of the objective that complement purely numeric timing features.

1.7 Oracle Optimal Memory Parameter

We refer to the oracle-optimal memory parameter as the value that minimizes the observed wall-clock runtime for a given problem instance when evaluated exhaustively over all candidate memory values.

2 Methodology and Implementation

2.1 Benchmark Suite Construction

Source and scope. Benchmark problems were obtained from *OptimizationProblems.jl*, a maintained library of smooth test problems with consistent metadata and evaluation routines. Since this project targets large scale unconstrained smooth optimization with L-BFGS, we restricted attention to problems that are unconstrained, free of bound constraints, and satisfy `nvar >= 5`. These criteria exclude trivial instances and ensure that curvature approximation and limited memory effects are practically relevant.

Regular and scalable problems. The benchmark contains two types of problem families. Regular problems have a fixed dimension determined by the problem definition. Scalable problems allow the dimension to be specified at construction time, as indicated by the metadata flag `variable_nvar = true`. For scalable families, we generate multiple instances by varying the dimension n in order to study how performance and the best choice of `mem` change with problem size.

Selection criteria. The first stage selection criteria are summarized in Listing 1. In addition, we require `ncon = 0` as a consistency check to verify the absence of equality or inequality constraints.

Listing 1: First stage filtering criteria.

```
contype == :unconstrained
!has_bounds
nvar >= 5
```

Benchmark summary. After filtering, the benchmark contains 105 distinct problem families, including 84 scalable families. The experiments comprise 19,015 solver runs in total, partitioned into 10,605 runs on regular (fixed dimension) instances and 8,410 runs on scalable instances. For the regular subset, the dimension satisfies $n \in [5, 100]$, with median 100 and mean 77.94. For the scalable subset, the dimension satisfies $n \in [961, 1000]$, with median 1000 and mean 998.16. In both subsets, for all problems, we exhaustively evaluate the L-BFGS memory parameter over the integer range $\texttt{mem} \in \{1, 2, \ldots, 100\}$.

Initialization runs. To avoid bias from one time initialization overhead (for example compilation and cache setup), we exclude the first solver call for each newly constructed problem instance from the primary analysis. We retain these runs and label them with `is_init_run = true` to enable a direct comparison between the first call and the subsequent call at the same memory setting (typically $\texttt{mem} = 1$).

2.2 Solver Configuration

All experiments use the L-BFGS solver from *JSOSolvers.jl*. Our objective is to isolate the effect of the limited-memory parameter `mem`. Accordingly, we vary only `mem` and keep all other solver settings at their default values. For each problem instance, we obtain the admissible integer domain of `mem` from `LBFGSParameterSet` and evaluate all values in that domain.

Each run starts from the same initial point provided by the test problem. To prevent pathological cases from dominating total compute time, we impose a wall-clock time limit of 60 seconds per run for regular instances and 300 seconds per run for scalable instances. In addition to solver outcomes (for example runtime, iteration count, and termination status), we record the cost of evaluating the objective and gradient at the starting point. These measurements are used as empirical features and to characterize initialization overhead.

2.3 Data Collection and Target Definition

For each solver run, we record the termination status, runtime, iteration count, and solver statistics including the number of objective and gradient evaluations and memory or allocation measurements when available. Runs that exceed the wall-clock limit or terminate abnormally are treated as failures and are excluded when defining the optimal parameter.

For each problem instance, we define the optimal memory parameter $\texttt{mem}^*$ as the value of `mem` that minimizes runtime among successful runs within the tested domain. When multiple values achieve the same runtime within numerical tolerance, we break ties by selecting the smaller `mem` to favor lower memory usage.

Initialization runs flagged by `is_init_run = true` are excluded from the primary analysis to avoid one time overhead effects, but are retained for secondary comparisons.

2.4 Feature Extraction

Gradients are computed via automatic differentiation through *ADNLPModels.jl*. We construct the learning feature set from three sources. First, we use analytical features from problem metadata, including nvar, `is_scalable`, `variable_nvar`, and the objective type label. Second, we include empirical features measured at the starting point, namely the objective evaluation cost and gradient evaluation cost (recorded in time, memory, and allocation statistics). Third, we augment these features with symbolic structure descriptors by converting the objective into an expression tree using *ExpressionTreeForge.jl* and computing two recursion based quantities: the number of leaves in the tree (`expr_leaf_count`) and the maximum root to leaf path length (`expr_max_depth`).

2.5 Implementation Details

All experiments are implemented in Python using `pandas` for data processing and `scikit-learn` for model training. Problem instances are uniquely identified by $(\texttt{problem}, \texttt{nvar})$ and are split at the instance level into training (70%), validation (15%), and test (15%) sets by shuffling the unique instance list with a fixed random seed and merging back into the full table, ensuring no leakage across splits. We train a random forest regressor to predict the log-transformed wall-clock runtime $\log(1 + \texttt{time})$ from a fixed feature vector consisting of problem features and the candidate memory value `mem`; hyperparameters are selected by validation-set MSE and the final model is retrained on train+validation before test evaluation. At inference time, memory selection is performed by evaluating the trained regressor over all candidates $\texttt{mem} \in \{1, \ldots, 100\}$ for each unseen instance and choosing the value that minimizes the predicted runtime, yielding a discrete memory selector derived from the learned runtime surface.

Column	Type	Description
status	Symbol	Termination status of the solver run (for example **first_order**, **max_time**, **unbounded**).
problem	String	Identifier of the optimization problem instance (problem name as stored in metadata).
solver	String	Name of the solver used for the run (here L-BFGS).
mem	Int	L-BFGS memory parameter controlling the number of stored curvature pairs.
nvar	Int	Number of decision variables n.
time	Float64	Wall-clock solve time in seconds.
memory	Float64	Memory usage associated with the solver run (MB).
num_iter	Int	Total number of iterations performed by the solver.
nvmops	Int	Number of vector–matrix style operations reported by the solver.
neval_obj	Int	Number of objective function evaluations.
init_eval_obj_time	Float64	Wall-clock time (seconds) for the initial objective evaluation at the starting point.
init_eval_obj_mem	Float64	Memory usage (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	Wall-clock time (seconds) for the initial gradient evaluation at the starting point.
init_eval_grad_mem	Float64	Memory usage (MB) for the initial gradient evaluation.
init_eval_grad_alloc	Float64	Number of heap allocations for the initial gradient evaluation.
is_init_run	Bool	Whether the row corresponds to the first solver call for a newly constructed instance (initialization run).
is_scalable	Bool	Indicator that the problem family supports programmatic variation of dimension.
objtype	String	Objective type label provided by the problem metadata (for example least squares versus other).
variable_nvar	Bool	Metadata flag indicating whether nvar can be specified at construction time.
expr_leaf_count	Int	Number of leaves in the objective expression tree (nodes with no children), computed by a recursive traversal.
expr_max_depth	Int	Maximum root-to-leaf path length in the objective expression tree, computed by a recursive traversal.

Table 1: Description of columns in the solver benchmark dataset.

3 Results

We report out-of-sample runtime prediction metrics for the random forest model and evaluate memory selection quality on the held-out test set of unseen problem instances.

3.1 Runtime Prediction Performance

Table 2 reports standard out-of-sample regression metrics on the test set.

Model	MSE	MAE	R^2
Random Forest	2.57	0.84	−1.84

Table 2: Out-of-sample runtime prediction performance on the test set.

Predictive accuracy is not the primary objective of this work. The learned model is instead used to compare runtimes across candidate memory values in order to guide memory selection. Accordingly, the coefficient of determination is treated as a diagnostic metric. Unlike the classical in-sample least-squares setting where R^2 is non-negative due to orthogonal projection, the out-of-sample predictive R^2 reported here may be negative when evaluated on unseen problem instances with noisy runtime measurements.

3.2 Memory Selection Performance

Let x denote a problem instance identified by (`problem`, `nvar`). Given a trained runtime regressor $\widehat{T}(x, m)$ and candidate set $m \in \{1, \ldots, 100\}$, we select

$$\hat{m}(x) = \arg\min_{m \in \{1,\ldots,100\}} \widehat{T}(x, m), \qquad m^\star(x) = \arg\min_{m \in \{1,\ldots,100\}} T(x, m),$$

and report the surrogate performance ratio

$$r(x) = \frac{\widehat{T}(x, \hat{m}(x))}{T(x, m^\star(x))},$$

where $T(x, m)$ is the observed wall-clock runtime. Since the numerator is model-predicted, $r(x)$ may be below 1; smaller values indicate that the selector is predicted to be closer to the oracle-best configuration.

Table 3 reports the resulting exact match accuracy and ratio-based performance metrics on the test set.

Metric	Value
Exact match accuracy	0.07
Median ratio r	0.68
Fraction with $r \leq 1.05$	0.72
Fraction with $r \leq 1.10$	0.76

Table 3: Memory selection performance on the test set.

4 Conclusions

We studied the problem of selecting the L-BFGS memory parameter using a data-driven approach based on runtime prediction. Although exact recovery of the oracle memory parameter is rare due to the large discrete search space, the learned selector consistently identifies configurations with near-optimal performance. On the held-out test set, the selected memory values achieve predicted runtimes within 10% of the oracle optimum for 76% of problem instances, and within 5% for 72% of instances, with a median performance ratio of 0.68. These results indicate that learning a runtime surface and selecting parameters by minimization can substantially improve efficiency compared to uninformed or fixed parameter choices, even when precise parameter recovery is difficult.

5 Future work

A natural next step is to deploy the learned selector as a lightweight web tool. Given a user-specified problem identifier and dimension (`problem`, `nvar`), the service would return a recommended memory parameter `mem` together with an estimated runtime profile. This interface could be extended to support preference-aware recommendations by letting users specify an optimization priority or constraint, for example minimize wall-clock time subject to a memory budget or reduce memory usage within a tolerated runtime slowdown, and returning the corresponding recommended `mem`.