The ensmallen library for flexible numerical optimization

Ryan R. Curtin  
ryan@ratml.org — RelationalAI, Atlanta, GA, USA

Marcus Edel  
Free University of Berlin, Germany

Rahul Ganesh Prabhu  
Birla Institute of Technology and Science Pilani, India

Suryoday Basak  
University of Texas at Arlington, USA

Zhihao Lou  
Epsilon, Chicago, IL, USA

Conrad Sanderson  
Data61/CSIRO, Australia, and Griffith University, Australia

Editor: Antti Honkela

Abstract

We overview the ensmallen numerical optimization library, which provides a flexible C++ framework for mathematical optimization of user-supplied objective functions. Many types of objective functions are supported, including general, differentiable, separable, constrained, and categorical. A diverse set of pre-built optimizers is provided, including Quasi-Newton optimizers and many variants of Stochastic Gradient Descent. The underlying framework facilitates the implementation of new optimizers. Optimization of an objective function typically requires supplying only one or two C++ functions. Custom behavior can be easily specified via callback functions. Empirical comparisons show that ensmallen outperforms other frameworks while providing more functionality. The library is available at https://ensmallen.org and is distributed under the permissive BSD license.

Keywords: Numerical optimization, mathematical optimization, function minimization.

1. Introduction

The problem of numerical optimization is generally expressed as \( \text{argmin}_x f(x) \) where \( f(x) \) is a given objective function and \( x \) is typically a vector or matrix. Such optimization problems are fundamental and ubiquitous in the computational sciences (Nocedal and Wright, 2006). Many frameworks or libraries for specific machine learning approaches have an integrated optimization component for distinct and limited use cases, such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019) and LibSVM (Chang and Lin, 2011). There are also many general numerical optimization toolkits aimed at supporting a wider range of use cases, including SciPy (Virtanen et al., 2020), opt++ (Meza, 1994), and OR-Tools (Perron and Furnon, 2019) among many others. However, such toolkits still have limitations in several areas, including: (i) types of supported objective functions, (ii) selection of available optimizers, (iii) support for custom behavior via callback functions, (iv) support for various underlying element and matrix types used by objective functions, and (v) extensibility, to facilitate adding more optimizers.

These shortcomings have motivated us to create the ensmallen library, which explicitly supports numerous types of user-defined objective functions, including general, differentiable, separable, categorical, and constrained objective functions, as well as semidefinite programs. Custom behavior during optimization can be specified via callback functions, for purposes such as printing progress, early stopping, inspection and modification of an
optimizer's state, and debugging of new optimizers. A large set of pre-built optimizers is
provided; at the time of writing, 46 optimizers are available. This includes simulated an-
nealing (Kirkpatrick et al., 1983), several Quasi-Newton optimizers (Liu and Nocedal, 1989;
Mokhtari et al., 2018), and many variants of Stochastic Gradient Descent (Ruder, 2016).

The user interface to the optimizers is intuitive and matches the ease of use of popular
optimization toolkits mentioned above; for more details, see the online documentation at
https://ensmallen.org/docs.html. Typically, a user only needs to implement one or two
C++ functions, and then they can use any optimizer matching the type of their objective.

Importantly, the ease-of-use does not come at the cost of efficiency; instead, ensmallen
uses C++ template metaprogramming techniques (hidden from the user) to provide ac-
celerations and simplifications where possible. The use of various underlying element and
matrix types is supported, including single- and double-precision floating point, integer val-
ues, and sparse data. Lastly, ensmallen provides an extensible framework to easily allow
the implementation of new optimization techniques.

2. Functionality

The task of optimizing an objective function with ensmallen is straightforward. The type
of objective function defines the implementation requirements. Each type has a minimal
set of methods that must be implemented; typically between one and four methods. Apart
from the requirement of an implementation of \( f(x) \), characteristics of \( f(x) \) can be exploited
through additional functions. For example, if \( f(x) \) is differentiable, an implementation
of \( f'(x) \) can be used to accelerate the optimization process. Then, one of the pre-built
differentiable function optimizers, such as L-BFGS (Liu and Nocedal, 1989), can be used.

Whenever possible, ensmallen will automatically infer methods that are not provided.
For example, given a separable objective function \( f(x) = \sum_i f_i(x) \) where an implementa-
tion of \( f_i(x) \) is provided (as well as the number of such separable objectives), an implementa-
tion of \( f(x) \) can be automatically inferred. This is done at compile-time, and so there is
no additional runtime overhead compared to a manual implementation. C++ template
metaprogramming techniques (Abrahams and Gurtovoy, 2004; Alexandrescu, 2001) are in-
ternally used to automatically produce efficient code during compilation.

To implement a new optimizer, the user only needs to implement a class with an
Optimize() method taking an external implementation of \( f(x) \) (and other functions specific
to the class of objective function). As such, ensmallen is easily extensible.

When an optimizer (either pre-built or new) is used with a user-provided objective func-
tion, the requirements for that optimizer are checked (e.g., presence of an implementation
of \( f'(x) \)), resulting in user-friendly error messages at compile-time if there are any issues.
For example, as L-BFGS is suited for differentiable functions, a compile-time error will be
printed if an attempt is made to use it with non-differentiable (general) functions.

3. Example Usage & Empirical Comparison

For an example implementation and comparison, let us first consider linear regression. In
this problem, predictors \( X \in \mathbb{R}^{d \times n} \) and associated responses \( y \in \mathbb{R}^n \) are given. We wish
```cpp
#include <ensmallen.hpp>

struct LinearRegressionFn
{
    LinearRegressionFn(const arma::mat& in_X, const arma::vec& in_Y) : X(in_X), y(in_Y) {}

double Evaluate(const arma::mat& phi)
    { const arma::vec tmp = X.t() * phi - y; return arma::dot(tmp, tmp); }

void Gradient(const arma::mat& phi, arma::mat& grad)
    { grad = 2 * X * (X.t() * phi - y); }

const arma::mat& X; const arma::vec& y;
};

int main()
{
    arma::mat X; arma::vec y;
    // ... set the contents of X and y here ...
    arma::mat phi_star(X.n_rows, 1, arma::fill::randu); // initial point (uniform random)
    LinearRegressionFn f(X, y);
    ens::L_BFGS optimizer; // create an optimizer object with default parameters
    optimizer.Optimize(f, phi_star); // after here, phi_star contains the optimized parameters
}
```

Figure 1: Example implementation of an objective function class for linear regression and usage of the L-BFGS optimizer. The optimizer can be easily changed by replacing `ens::L_BFGS` with another optimizer, such as `ens::GradientDescent`, or `ens::SA` which implements simulated annealing (Kirkpatrick et al., 1983).

to find the best linear model $\Phi \in \mathcal{R}^d$, which translates to finding $\Phi^* = \arg\min_{\Phi} f(\Phi)$ for $f(\Phi) = \|X^T \Phi - y\|^2$. This gives the gradient $f'(\Phi) = 2X(X^T \Phi - y)$.

To find $\Phi^*$ using a differentiable optimizer, we simply need to provide implementations of $f(\Phi)$ and $f'(\Phi)$. For a differentiable function, `ensmallen` requires only two methods: `Evaluate()` and `Gradient()`. The pre-built L-BFGS optimizer can then be used to find $\Phi^*$. Figure 1 shows an example implementation. Via the use of the Armadillo library (Sanderson and Curtin, 2016), the linear algebra expressions to implement the objective function and its gradient are compact and closely match natural mathematical notation. Armadillo efficiently translates the expressions into standard BLAS and LAPACK function calls (Anderson et al., 1999), allowing easy exploitation of high-performance implementations such as the multi-threaded OpenBLAS (Xianyi et al., 2020) and Intel MKL (Intel, 2020) libraries.

Table 1 compares the performance of `ensmallen` against other frameworks for the linear regression problem on various dataset sizes. We compare against SciPy, Optim.jl (Mogensen and Riseth, 2018), and the bfgsmin() function from GNU Octave (Eaton et al., 2018). We also compare against the automatic differentiation implementations of PyTorch, TensorFlow, and the Python library Autograd (Maclaurin et al., 2015). In each framework, the provided L-BFGS optimizer is limited to 10 iterations. Highly noisy random data with a slight linear pattern is used. The runtimes are the average of 5 runs. The experiments were performed on an AMD Ryzen 7 2700X with 64GB RAM, with g++ 10.2.0, Julia 1.5.2, Python 3.8.5, and Octave 6.1.0. For fairness, all tools used the CPU only.
Next, we consider the common machine learning problem of logistic regression using two-class versions of various real datasets from the UCI dataset repository (Lichman, 2013). The setup of our experiments is the same as for the previous example; results are in Table 2.

Both simulations show that ensmallen achieves the lowest runtimes, sometimes by large margins. This is due to multiple factors, including the efficiency of the optimizer implementations in ensmallen, template metaprogramming optimizations in Armadillo and ensmallen, and minimal overhead and dependencies compared to the competitors.

4. Conclusion

The ensmallen numerical optimization provides a flexible framework for optimization of user-supplied objective functions in C++. Unlike other frameworks, ensmallen supports many types of objective functions, provides a diverse set of pre-built optimizers, supports custom behavior via callback functions, and handles various element and matrix types used by objective functions. The underlying framework facilitates the implementation of new optimization techniques, which can be contributed for inclusion into the library.

The library has been successfully used by open source projects such as the mlpack machine learning toolkit (Curtin et al., 2018). The library uses the permissive BSD license (St. Laurent, 2008), with the development done in an open and collaborative manner. The source code and documentation are freely available at https://ensmallen.org.

Further details, such as internal use of template metaprogramming for automatic generation of efficient code, automatic function inference, clean error reporting, and various approaches for obtaining efficiency are all discussed in the accompanying technical report (Curtin et al., 2020).

<table>
<thead>
<tr>
<th>Framework</th>
<th>d: 100, n: 1k</th>
<th>d: 100, n: 10k</th>
<th>d: 100, n: 100k</th>
<th>d: 1k, n: 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>ensmallen</td>
<td>0.0016s</td>
<td>0.0067s</td>
<td>0.1460s</td>
<td>1.4011s</td>
</tr>
<tr>
<td>Optim.jl</td>
<td>0.0069s</td>
<td>0.0117s</td>
<td>0.1672s</td>
<td>1.3985s</td>
</tr>
<tr>
<td>SciPy</td>
<td>0.0028s</td>
<td>0.0110s</td>
<td>0.2247s</td>
<td>1.8461s</td>
</tr>
<tr>
<td>Autograd</td>
<td>0.0073s</td>
<td>0.0163s</td>
<td>0.2416s</td>
<td>1.8733s</td>
</tr>
<tr>
<td>PyTorch</td>
<td>0.0469s</td>
<td>0.0986s</td>
<td>0.5670s</td>
<td>5.6041s</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>0.1876s</td>
<td>0.2306s</td>
<td>0.6925s</td>
<td>6.6764s</td>
</tr>
<tr>
<td>bfgsmin()</td>
<td>1.9773s</td>
<td>18.0515s</td>
<td>123.437s</td>
<td>9710.6750s</td>
</tr>
</tbody>
</table>

Table 1: Runtimes for optimizing linear regression parameters on various dataset sizes, where n is the number of samples, and d is the dimensionality of each sample.

<table>
<thead>
<tr>
<th>Framework</th>
<th>MNIST 60k × 784</th>
<th>coverytype 407k × 55</th>
<th>pokerhand 700k × 10</th>
<th>font 832k × 407</th>
<th>isole 7.8k × 617</th>
</tr>
</thead>
<tbody>
<tr>
<td>ensmallen</td>
<td>0.6546s</td>
<td>0.9038s</td>
<td>0.5186s</td>
<td>6.1678s</td>
<td>0.0510s</td>
</tr>
<tr>
<td>Optim.jl</td>
<td>1.4231s</td>
<td>1.2067s</td>
<td>0.6754s</td>
<td>10.9051s</td>
<td>0.1214s</td>
</tr>
<tr>
<td>SciPy</td>
<td>0.8101s</td>
<td>1.1388s</td>
<td>1.0231s</td>
<td>7.5838s</td>
<td>0.07519s</td>
</tr>
<tr>
<td>Autograd</td>
<td>0.8012s</td>
<td>1.4241s</td>
<td>2.6005s</td>
<td>7.1224s</td>
<td>0.0876s</td>
</tr>
<tr>
<td>PyTorch</td>
<td>6.5710s</td>
<td>8.8340s</td>
<td>3.2404s</td>
<td>59.0194s</td>
<td>0.8172s</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>9.3662s</td>
<td>5.4231s</td>
<td>2.6005s</td>
<td>70.1122s</td>
<td>0.7563s</td>
</tr>
<tr>
<td>bfgsmin()</td>
<td>539.1358s</td>
<td>43.9067s</td>
<td>8.2561s</td>
<td>2358.1680s</td>
<td>48.8020s</td>
</tr>
</tbody>
</table>

Table 2: Runtimes for training a logistic regression model on real data with L-BFGS.
References


