Journal of Machine Learning Research 1 (2000) 1-48 Submitted 4/00; Published 10/00

Flexible numerical optimization with ensmallen

Ryan R. Curtin RYAN@RATML.ORG
RelationalAI,
Atlanta, GA 30318, USA

Marcus Edel
Free University of Berlin,
Berlin, Germany

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

Suryoday Basak
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, TX, USA

Zhihao Lou
Epsilon
Chicago, IL, USA

Conrad Sanderson
Griffith University
Australia

Editor: not currently known

Abstract

We introduce the `ensmallen` numerical optimization library, and give an overview of its functionality and how it works. The library provides a fast and flexible C++ framework for mathematical optimization of arbitrary user-supplied functions. A large set of pre-built optimizers is provided, including many variants of Stochastic Gradient Descent and Quasi-Newton optimizers. Several types of objective functions are supported, including differentiable, separable, constrained, and categorical objective functions. Implementation of a new optimizer requires only one method, while a new objective function requires typically only one or two C++ methods. Through internal use of C++ template metaprogramming, `ensmallen` provides support for arbitrary user-supplied callbacks and automatic inference of unsupplied methods without any runtime overhead. Empirical comparisons show that `ensmallen` outperforms other optimization frameworks (such as Julia and SciPy), sometimes by large margins. The library is available at `https://ensmallen.org` and is distributed under the permissive BSD license.

Keywords: Optimization, numerical optimization, blah blah, math or something

1. Introduction

The problem of mathematical optimization is fundamental in the computational sciences **?**. In short, this problem is expressed as

$$\operatorname*{argmin}_{x} f(x) \tag{1}$$

where $f(x)$ is a given objective function and x is typically a vector or matrix. The ubiquity of this problem gives rise to the proliferation of mathematical optimization toolkits, such as SciPy **?**, opt++ **?**, OR-Tools **?**, CVXOPT **?**, NLopt **?**, Ceres **?**, and RBFOpt **?**. Furthermore, in the field of machine learning, many deep learning frameworks have integrated optimization components. Examples include Theano **?**, TensorFlow **?**, PyTorch **?**, and Caffe **?**.

Mathematical optimization is generally quite computationally intensive. For instance, the training of deep neural networks is dominated by the optimization of the model parameters on the data **??**. Similarly, other popular machine learning algorithms such as logistic regression are also expressed as and dominated by an optimization process **??**. Computational bottlenecks occur even in fields as wide-ranging as rocket landing guidance systems **?**, motivating the development and implementation of specialized solvers.

The necessity for both efficient and specializable function optimization motivated us to implement the `ensmallen` library, originally as part of the `mlpack` machine learning library **?**. The `ensmallen` library provides a large set of pre-built optimizers for optimizing user-defined objective functions in C++; at the time of writing, 46 optimizers are available. The external interface to the optimizers is intuitive and matches the ease of use of popular optimization toolkits mentioned above.

However, unlike many of the existing optimization toolkits, `ensmallen` explicitly supports numerous function types: arbitrary, differentiable, separable, categorical, constrained, and semidefinite. Furthermore, custom behavior during optimization can be easily specified via *callbacks*. Lastly, the underlying framework facilitates the implementation of new optimization techniques, which can be contributed upstream and incorporated into the library. Table **??** contrasts the functionality supported by `ensmallen` and other optimization toolkits.

This report is a revised and expanded version of our initial overview of `ensmallen` **?**. It also provides a deep dive into the internals of how the library works, which can be a useful resource for anyone looking to contribute to the library or get involved with its development.

We continue the report as follows. We overview the available functionality and show example usage in Section **??**. Advanced usage such as callbacks is discussed in Section **??**. We demonstrate the empirical efficiency of `ensmallen` in Section **??**. The salient points are summarized in Section **??**.

	unified framework	constraints	separable functions / batches	arbitrary functions	arbitrary optimizers	sparse gradients	categorical	arbitrary types	callbacks
`ensmallen`	●	●	●	●	●	●	●	●	●
Shogun ?	●	-	●	●	●	-	-	-	-
Vowpal Wabbit ?	-	-	●	-	-	-	●	-	-
TensorFlow ?	●	-	●	◐	-	◐	-	◐	-
PyTorch ?	●	-	●	◐	◐	-	-	◐	-
Caffe ?	●	-	●	◐	◐	-	-	◐	●
Keras ?	●	-	●	◐	◐	-	-	◐	●
scikit-learn ?	◐	-	◐	◐	-	-	-	◐	-
SciPy ?	●	●	-	●	-	-	-	◐	●
MATLAB ?	●	●	-	●	-	-	-	◐	-
Julia (`Optim.jl`) ?	●	-	-	●	-	-	-	●	-

Table 1: Feature comparison: ● = provides feature, ◐ = partially provides feature, - = does not provide feature. *unified framework* indicates if there is a form of generic/unified optimization framework; *constraints* and *separable functions / batches* indicate support for constrained functions and separable functions; *arbitrary functions* means arbitrary objective functions are easily implemented; *arbitrary optimizers* means arbitrary optimizers are easily implemented; *sparse gradient* indicates that the framework can natively take advantage of sparse gradients; *categorical* refers to if support for categorical features exists; *arbitrary types* mean that arbitrary types can be used for the parameters x; *callbacks* indicates that user-implementable callback support is available.

2. Overview

The task of optimizing an objective function with `ensmallen` is straightforward. The class of objective function (e.g., arbitrary, constrained, differentiable, etc.) defines the implementation requirements. Each objective function type has a minimal set of methods that must be implemented. Typically this is only between one and four methods. As an example, to optimize an objective function $f(x)$ that is differentiable, implementations of $f(x)$ and $f'(x)$ are required. One of the optimizers for differentiable functions, such as L-BFGS ?, can then be immediately employed.

Whenever possible, `ensmallen` will automatically infer methods that are not provided. For instance, given a separable objective function $f(x) = \sum_i f_i(x)$ where an implementation of $f_i(x)$ is provided (as well as the number of such separable objectives), an implementation 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 **???** are internally used to produce efficient code during compilation.

Not every type of objective function can be used with every type of optimizer. For instance, since L-BFGS is a differentiable optimizer, it cannot be used with a non-differentiable object function type (e.g. an arbitrary function). When an optimizer is used with a user-provided objective function, an internal mechanism automatically checks the requirements, resulting in user-friendly error messages if any required methods are not detected.

2.1 Types of Objective Functions

In most cases, the objective function $f(x)$ has inherent attributes; for example, as mentioned in the previous paragraphs, $f(x)$ might be differentiable. The internal framework in `ensmallen` can optionally take advantage of such attributes. In the example of a differentiable function $f(x)$, the user can provide an implementation of the gradient $f'(x)$, which in turn allows a first-order optimizer to be used. This generally leads to significant speedups when compared to using only $f(x)$. To allow exploitation of such attributes, the optimizers are built to work with many types of objective functions. The classes of objective functions are listed below.

For details on the required signatures for each objective function type (such as `DifferentiableFunctionTyp`
see the online documentation at `https://ensmallen.org/docs.html`.

- **Arbitrary functions** (`ArbitraryFunctionType`). No assumptions are made on function $f(x)$ and only the objective $f(x)$ can be computed for a given x.

- **Differentiable functions** (`DifferentiableFunctionType`). A differentiable function $f(x)$ is an arbitrary function whose gradient $f'(x)$ can be computed for a given x, in addition to the objective.

- **Partially differentiable functions** (`PartiallyDifferentiableFunctionType`). A partially differentiable function $f(x)$ is a differentiable function with the additional property that the gradient $f'(x)$ can be decomposed along some basis j such that $f'_j(x)$ is sparse. Often, this is used for coordinate descent algorithms (i.e., $f'(x)$ can be decomposed into $f'_1(x)$, $f'_2(x)$, etc.).

- **Arbitrary separable functions** (`ArbitrarySeparableFunctionType`). An arbitrary separable function is an arbitrary function $f(x)$ that can be decomposed into the sum of several objective functions: $f(x) = \sum_i f_i(x)$.

- **Differentiable separable functions** (`DifferentiableSeparableFunctionType`). A differentiable separable function is a separable arbitrary function $f(x)$ where the individual gradients $f_i'(x)$ are also computable.

- **Categorical functions** (`CategoricalFunctionType`). A categorical function type is an arbitrary function $f(x)$ where some (or all) dimensions of x take discrete values from a set.

- **Constrained functions** (`ConstrainedFunctionType`). A constrained function $f(x)$ is a differentiable function[1] subject to constraints of the form $c_i(x)$; when the constraints are satisfied, $c_i(x) = 0 \ \forall \ i$. Minimizing $f(x)$ then means minimizing $f(x) + \sum_i c_i(x)$.

- **Semidefinite programs** (SDPs). *(These are a subset of constrained functions.)* `ensmallen` has special support to make optimizing semidefinite programs **?** straightforward.

2.2 Pre-Built Optimizers

For each class of the objective functions, `ensmallen` provides a set of pre-built optimizers:

- **For arbitrary functions:** Simulated annealing **?**, CNE (Conventional Neural Evolution) **?**, DE (Differential Evolution) **?**, PSO (Particle Swarm Optimization) **?**, SPSA (Simultaneous Perturbation Stochastic Approximation) **?**.

- **For differentiable functions:** L-BFGS **?**, Frank-Wolfe **?**, gradient descent.

- **For partially differentiable functions.** SCD (Stochastic Coordinate Descent) **?**.

- **For arbitrary separable functions:** CMA-ES (Covariance Matrix Adaptation Evolution Strategy) **?**.

- **For differentiable separable functions:** AdaBound **?**, AdaDelta **?**, AdaGrad **?**, Adam **?**, AdaMax **?**, AMSBound **?**, AMSGrad **?**, Big Batch SGD **?**, Eve **?**, FTML (Follow The Moving Leader) **?**, Hogwild! **?**, IQN (Incremental Quasi-Newton) **?**, Katyusha **?**, Lookahead **?**, SGD with momentum **?**, Nadam **?**, NadaMax **?**, SGD with Nesterov momentum **?**, Optimistic Adam **?**, QHAdam (Quasi-Hyperbolic Adam) **?**, QHSGD (Quasi-Hyperbolic Stochastic Gradient Descent) **?**, RMSProp **?**, SARAH/SARAH+ **?**, stochastic gradient descent, SGDR (Stochastic Gradient Descent with Restarts) **?**, Snapshot SGDR **?**, SMORMS3 **?**, SVRG (Stochastic Variance Reduced Gradient) **?**, SWATS **?**, SPALeRA (Safe Parameter-wise Agnostic LEarning Rate Adaptation) **?**, WNGrad **?**.

1. Generally, constrained functions do not need to be differentiable. However, this is a requirement here, as all of the current optimizers in `ensmallen` for constrained functions require a gradient to be available.

- **For categorical functions:** Grid search.
- **For constrained functions:** Augmented Lagrangian method, primal-dual interior point SDP solver, LRSDP (low-rank accelerated SDP solver) **?**.

2.3 Example Usage

Let us consider the problem of linear regression, where we are given a matrix of predictors $\boldsymbol{X} \in \mathcal{R}^{n \times d}$ and a vector of responses $\boldsymbol{y} \in \mathcal{R}^n$. Our task is to find the best linear model $\boldsymbol{\theta} \in \mathcal{R}^d$; that is, we want to find $\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ for

$$f(\boldsymbol{\theta}) = \|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|^2 = (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}). \tag{2}$$

From this we can derive the gradient $f'(\boldsymbol{\theta})$:

$$f'(\boldsymbol{\theta}) = 2\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}). \tag{3}$$

To find $\boldsymbol{\theta}^*$ using a differentiable optimizer[2], we simply need to provide implementations of $f(\boldsymbol{\theta})$ and $f'(\boldsymbol{\theta})$ according to the signatures required by the `DifferentiableFunctionType` of objective function. For a differentiable function, only two methods are necessary: `Evaluate()` and `Gradient()`. The pre-built L-BFGS optimizer can be used to find $\boldsymbol{\theta}^*$.

Figure **??** shows an example implementation. We hold `X` and `y` as members of the `LinearRegressionFunction class`, and `theta` is used to represent $\boldsymbol{\theta}$. Via the use of Armadillo **?**, the linear algebra expressions to implement the objective function and gradient are readable in a way that closely matches Equations (**??**) and (**??**).

3. Experiments

To show the efficiency of mathematical optimization with `ensmallen`, we compare its performance with several other commonly used optimization frameworks, including some that use automatic differentiation.

3.1 Simple Optimizations and Overhead

For our first experiment, we aim to capture the overhead involved in various optimization toolkits. In order to do this, we consider the simple and popular Rosenbrock function **?**:

$$f([x_1, x_2]) = 100(x_2 - x_1^2)^2 + (1 - x_1^2). \tag{4}$$

This objective function is useful for this task because the computational effort involved in computing $f(\cdot)$ is trivial. Therefore, if we hold the number of iterations of each toolkit constant, then this will help us understand the overhead costs of each toolkit. For the

2. Typically, in practice, solving a linear regression model can be done directly by taking the pseudoinverse: $\theta^* = X^{\dagger}y$. However, the objective function is easy to describe and useful for demonstration, so we use it in this document.

```
#include <ensmallen.hpp>

class LinearRegressionFunction
{
 public:
  LinearRegressionFunction(const arma::mat& in_X, const arma::vec& in_y) : X(in_X), y(in_y)  { }

  double Evaluate(const arma::mat& theta)  { return (X * theta - y).t() * (X * theta - y); }

  void Gradient(const arma::mat& theta, arma::mat& gradient)  { gradient = 2 * X.t() * (X * theta -

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

int main()
{
  arma::mat X;
  arma::vec y;

  // ... set the contents of X and y here ...

  ens::LinearRegressionFunction f(X, y);

  ens::L_BFGS optimizer; // create the optimizer with default parameters
  arma::mat theta_best(X.n_rows, 1, arma::fill::randu);  // initial starting point (uniform random

  optimizer.Optimize(f, theta_best);
  // at this point theta_best contains the best parameters
}
```

Figure 1: An example implementation of an objective function class for linear regression and usage of the L-BFGS optimizer in `ensmallen`. The online documentation for all ensmallen optimizers is at `https://ensmallen.org/docs.html`. The `arma::mat` and arma::vec types are dense matrix and vector classes from the Armadillo linear algebra library **?**, with the corresponding online documentation at `http://arma.sf.net/docs.html`.

optimizer, we use simulated annealing **?**, a gradient-free optimizer. Simulated annealing will call the objective function numerous times; for each simulation we limit the optimizer to 100K objective evaluations.

The code used to run this simulation for `ensmallen` (including the implementation of the Rosenbrock function) is given in Figure ??. Note that the `RosenbrockFunction` is actually implemented in `ensmallen`'s source code, in the directory `include/ensmallen_bits/problems/`.

We compare four frameworks for this task:

(i) `ensmallen`,
(ii) `scipy.optimize.anneal` from SciPy 0.14.1 ?,
(iii) simulated annealing implementation in `Optim.jl` with Julia 1.0.1 ?,
(iv) `samin` in the `optim` package for Octave ?.

While another option here might be `simulannealbnd()` in the Global Optimization Toolkit for MATLAB, no license was available. We ran our code on a MacBook Pro i7 2018 with 16GB RAM running macOS 10.14 with clang 1000.10.44.2, Julia version 1.0.1, Python 2.7.15, and Octave 4.4.1.

Our initial implementation for each toolkit, corresponding to the line "default" in Table ??, was as simple of an implementation as possible and included no tuning. This reflects how a typical user might interact with a given framework. Only Julia and `ensmallen` are compiled, and thus are able to avoid the function pointer dereference for evaluating the Rosenbrock function and take advantage of inlining and related optimizations. The overhead of both `scipy` and `samin` are quite large—`ensmallen` is nearly three orders of magnitude faster for the same task.

However, both Python and Octave have routes for acceleration, such as Numba ?, MEX bindings and JIT compilation. We hand-optimized the Rosenbrock implementation using Numba, which required significant modification of the underlying `anneal.anneal()` function. These techniques did produce some speedup, as shown in the second row of Table ??. For Octave, a MEX binding did not produce a noticeable difference. We were unable to tune either `ensmallen` or `Optim.jl` to get any speedup, suggesting that novice users will easily be able to write efficient code in these cases.

3.2 Large-Scale Linear Regression Problems

Next, we consider the linear regression example described in Section ??. For this task we use the first-order L-BFGS optimizer ?, implemented in `ensmallen` as the `L_BFGS` class.

	`ensmallen`	`scipy`	`Optim.jl`	`samin`
default	**0.004s**	1.069s	0.021s	3.173s
tuned		0.574s		3.122s

Table 2: Runtimes for 100K iterations of simulated annealing with various frameworks on the simple Rosenbrock function. Julia code runs do not count compilation time. The *tuned* row indicates that the code was manually modified for speed.

```cpp
#include <ensmallen.hpp>

struct RosenbrockFunction
{
  template<typename MatType>
  static typename MatType::elem_type Evaluate(const MatType& x) const
  {
    return 100 * std::pow(x[1] - std::pow(x[0], 2), 2) + std::pow(1 - x[0], 2);
  }
};

int main()
{
  arma::wall_clock clock;

  RosenbrockFunction rf;
  ens::ExponentialSchedule sched;
  // A tolerance of 0.0 means the optimization will run for the maximum number of iterations.
  ens::SA<> s(sched, 100000, 10000, 1000, 100, 0.0);

  // Get the initial point of the optimization.
  arma::mat parameters = rf.GetInitialPoint();

  // Run the optimization and time it.
  clock.tic();
  s.Optimize(rf, parameters);
  const double time = clock.toc();
  std::cout << time << std::endl << "Result (optimal 1, 1): " << parameters.t();
  return 0;
}
```

Figure 2: Code to use `ensmallen` to optimize the Rosenbrock function using 100K iterations of simulated annealing.

Using the same four packages, we implement the linear regression objective and gradient. Remembering that `ensmallen` allows us to share work across the objective function and gradient implementations (Section ??), for `ensmallen` we implement a version with only `EvaluateWithGradient()`, denoted as `ensmallen-1`. We also implement a version with both `Evaluate()` and `Gradient()`: `ensmallen-2`. We also use automatic differentiation for Julia via the `ForwardDiff.jl` ? package and for Python via the `Autograd` ? package. For GNU Octave we use the `bfgsmin()` function.

Results for various data sizes are shown in Table ??. For each implementation, L-BFGS was allowed to run for only 10 iterations and never converged in fewer iterations. The

algorithm	d: 100, n: 1k	d: 100, n: 10k	d: 100, n: 100k	d: 1k, n: 100k
`ensmallen-1`	**0.001s**	**0.009s**	**0.154s**	**2.215s**
`ensmallen-2`	0.002s	0.016s	0.182s	2.522s
`Optim.jl`	0.006s	0.030s	0.337s	4.271s
`scipy`	0.003s	0.017s	0.202s	2.729s
`bfgsmin`	0.071s	0.859s	23.220s	2859.81s
`ForwardDiff.jl`	0.497s	1.159s	4.996s	603.106s
`autograd`	0.007s	0.026s	0.210s	2.673s

Table 3: Runtimes for the linear regression function on various dataset sizes, with n indicating the number of samples, and d indicating the dimensionality of each sample. All Julia runs do not count compilation time.

datasets used for training are highly noisy random data with a slight linear pattern. Note that the exact data is not relevant for the experiments here, only its size. Runtimes are reported as the average across 10 runs.

The results indicate that `ensmallen` with `EvaluateWithGradient()` is the fastest approach. Furthermore, the use of `EvaluateWithGradient()` yields non-negligible speedup over the `ensmallen-2` implementation with both the objective and gradient implemented separately. In addition, although the automatic differentiation support makes it easier for users to write their code (since they do not have to write an implementation of the gradient), we observe that the output of automatic differentiators is not as efficient, especially with `ForwardDiff.jl`. We expect this effect to be more pronounced with increasingly complex objective functions.

3.3 Easy Pluggability of Various Optimizers

Lastly, we demonstrate the easy pluggability in `ensmallen` for using various optimizers on the same task. Using a version of `LinearRegressionFunction` from Section **??** adapted for separable differentiable optimizers, we run six optimizers with default parameters in just 8 lines of code, as shown in Fig. **??**. Applying these optimizers to the `YearPredictionMSD` dataset from the UCI repository **?** yields the learning curves shown in Fig. **??**.