Journal of Machine Learning Research 22 (2021) 1-6 Submitted 5/20; Revised 2/21; Published 6/21

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 $\operatorname{argmin}_x f(x)$ 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 annealing (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 accelerations and simplifications where possible. The use of various underlying element and matrix types is supported, including single- and double-precision floating point, integer values, 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 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 (Abrahams and Gurtovoy, 2004; Alexandrescu, 2001) are internally 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 function, 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 $\boldsymbol{X} \in \mathcal{R}^{d \times n}$ and associated responses $\boldsymbol{y} \in \mathcal{R}^n$ 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 $\mathbf{\Phi} \in \mathcal{R}^d$, which translates to finding $\mathbf{\Phi}^* = \text{argmin}_{\mathbf{\Phi}} f(\mathbf{\Phi})$ for $f(\mathbf{\Phi}) = \|\boldsymbol{X}^\top \mathbf{\Phi} - \boldsymbol{y}\|^2$. This gives the gradient $f'(\mathbf{\Phi}) = 2\boldsymbol{X}(\boldsymbol{X}^\top \mathbf{\Phi} - \boldsymbol{y})$.

To find $\mathbf{\Phi}^*$ using a differentiable optimizer, we simply need to provide implementations of $f(\mathbf{\Phi})$ and $f'(\mathbf{\Phi})$. For a differentiable function, `ensmallen` requires only two methods: `Evaluate()` and `Gradient()`. The pre-built L-BFGS optimizer can then be used to find $\mathbf{\Phi}^*$. 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).

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

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.

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

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

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1999.

Chih-Chung Chang and Chih-Jen Lin. LibSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011.

Ryan R. Curtin, Marcus Edel, Mikhail Lozhnikov, Yannis Mentekidis, Sumedh Ghaisas, and Shangtong Zhang. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software*, 3(26):726, 2018.

Ryan R. Curtin, Marcus Edel, Rahul Ganesh Prabhu, Suryoday Basak, Zhihao Lou, and Conrad Sanderson. Flexible numerical optimization with ensmallen. *arXiv:2003.04103*, 2020.

John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 4.4.0 manual: a high-level interactive language for numerical computations*, 2018.

Intel. Math Kernel Library (MKL), 2020. URL `https://software.intel.com/mkl`.

Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

M. Lichman. UCI Machine Learning Repository, 2013. `http://archive.ics.uci.edu/ml`.

Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in NumPy. In *AutoML Workshop at ICML*, 2015.

Juan C. Meza. OPT++: An object-oriented class library for nonlinear optimization. Technical report, Sandia National Labs., Livermore, CA (United States), 1994.

Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018.

Aryan Mokhtari, Mark Eisen, and Alejandro Ribeiro. IQN: An incremental quasi-Newton method with local superlinear convergence rate. *SIAM Journal on Optimization*, 28(2): 1670–1698, 2018.

Jorge Nocedal and Stephen Wright. *Numerical Optimization.* Springer, 2nd edition, 2006.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035, 2019.

Laurent Perron and Vincent Furnon. OR-tools, 2019. URL `https://developers.google.com/optimization/`.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 2016.

Conrad Sanderson and Ryan R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016.

Andrew St. Laurent. *Understanding Open Source and Free Software Licensing.* O'Reilly Media, 2008.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020.

Zhang Xianyi, Wang Qian, and Werner Saar. OpenBLAS, 2020. URL `http://www.openblas.net`.