
A generic and fast C++ optimization framework

Ryan R. Curtin
Symantec Corporation
Atlanta, GA, USA 30318
ryan@ratml.org

Shikhar Bhardwaj
Delhi Technological University
Delhi - 110042
shikhar_bt2k15@dtu.ac.in

Marcus Edel
Free University of Berlin
Arnimallee 7, 14195 Berlin
marcus.edel@fu-berlin.de

Yannis Mentekidis
mentekid@gmail.com

Abstract

The development of the **mlpack** C++ machine learning library (<http://www.mlpack.org/>) has required the design and implementation of a flexible, robust optimization system that is able to solve the types of arbitrary optimization problems that may arise all throughout machine learning problems. In this paper, we present the generic optimization framework that we have designed for mlpack. A key priority in the design was ease of implementation of both new optimizers and new objective functions to be optimized; therefore, implementation of a new optimizer requires only one method and implementation of a new objective function requires at most four functions. This leads to simple and intuitive code, which, for fast prototyping and experimentation, is of paramount importance. When compared to optimization frameworks of other libraries, we find that mlpack's supports more types of objective functions, is able to make optimizations that other frameworks do not, and seamlessly supports user-defined objective functions *and* optimizers.

1 Introduction

Machine learning is a field that is inextricably intertwined with the field of optimization. Countless machine learning techniques depend on the optimization of a given objective function; for instance, classifiers such as logistic regression [4], metric learning methods like NCA [7], manifold learning algorithms like MVU [17], and the extremely popular field of deep learning [14]. Thanks to the attention focused on these problems, it is increasingly important in the field to have fast, practical optimizers. This can explain the current focus on optimization: this year at ICML (2017), every single session had at least one track devoted solely to optimization techniques.

Therefore, the need is real to provide a robust, flexible framework in which new optimizers can be easily developed. Similarly, the need is also real for a flexible framework that allows new objective functions to be easily implemented and optimized with a variety of possible optimizers.

However, the current landscape of optimization frameworks for machine learning is not particularly comprehensive. A variety of tools such as Caffe [8], TensorFlow [1], and Keras [3] have optimization frameworks, but they are limited to SGD-type optimizers and are only able to optimize deep neural networks or related structures. Thus expressing arbitrary machine learning objective functions can be difficult or in some cases not possible. Other libraries, like scikit-learn [12], do have optimizers, but generally not in a coherent framework and often the implementations may be specific to an individual machine learning algorithm. At a higher level, many programming languages may have generic optimizers, like SciPy [9] and MATLAB [11], but typically these optimizers are not suitable for large-scale machine learning tasks where, e.g., calculating the full gradient of all of the data may not be feasible. Table 1 provides a rough overview.

Given this situation, we have developed a flexible optimization infrastructure in the **mlpack** C++ machine learning library [5]. This infrastructure makes it easy to combine nearly any type of optimizer

	has framework	constraints	batches	arbitrary functions	arbitrary optimizers	sparse gradients	categorical
mlpack [5]	●	●	●	●	●	●	●
Shogun [15]	●	-	●	●	●	-	-
Vowpal Wabbit [10]	-	-	●	-	-	-	●
TensorFlow [1]	●	-	●	◐	-	◐	-
Caffe [8]	●	-	●	◐	◐	-	-
Keras [3]	●	-	●	◐	◐	-	-
scikit-learn [12]	◐	-	◐	◐	-	-	-
SciPy [9]	●	●	-	●	-	-	-
MATLAB [11]	●	●	-	●	-	-	-

Table 1: Feature comparison: ●= provides feature, ◐= partially provides feature, - = does not provide feature. *has framework* the library has some kind of generic optimization framework; *constraints* and *batches* indicate support for constrained problems and batches; *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; and *categorical* refers to whether support for categorical features exists.

with nearly any type of objective function, and has allowed us to minimize the effort necessary to both implement new optimizers and to implement new machine learning algorithms that depend on optimization. Since the framework is implemented in C++ and uses template metaprogramming, we are able to preserve clean syntax while simultaneously allowing compile-time speedups that can result in a generic optimizer that is as fast as an optimizer written for one specific objective function.

In this short paper, we describe mlpack’s optimization infrastructure in detail. First, we describe the types of optimization problems we would like to solve, which then allows us to build a generic and intuitive API for objective functions to implement and optimizers to depend upon. We show a very simple example objective function and optimizer, and a system that we have built to detect when a user passes an objective function that cannot be optimized by a certain optimizer. Lastly, we show the optimizers we have implemented in this framework, and some example usage of mlpack’s optimizers.

2 The mlpack machine learning library

mlpack is a C++ machine learning library with an emphasis on speed, flexibility, and ease of use [5]; it has been continuously developed since 2007. mlpack uses the Armadillo library [13] for linear algebra and matrix primitives. Both mlpack and Armadillo exploit C++ language features such as policy-based design and template metaprogramming to provide compile-time optimizations that result in fast code [13], while retaining a simple and easy-to-use interface [6].

Many of the speedups in mlpack depend on the technique of *policy-based design* [2]. With this design paradigm, mlpack provides a number of classes with modular functionality. As a simple example, mlpack’s mean shift clustering implementation (as of version 2.2.5) takes three template parameters:

```
template<bool UseKernel, typename KernelType, typename MatType>
class MeanShift;
```

Thus a user wishing to perform kernelized mean shift clustering with the Gaussian kernel might simply use the class `MeanShift<true, GaussianKernel, arma::mat>` where `arma::mat` is Armadillo’s dense matrix type and `GaussianKernel` is a class that provides one simple `Evaluate()` method.

Since the `KernelType` argument is a template argument, any class can be used—it does not need to be part of mlpack. Therefore, a user can simply implement their own `KernelType` class with the single `Evaluate()` method and use that as a template argument to the `MeanShift` class, and the compiler will generate a specialized implementation using the custom kernel. Similar support could be accomplished by, e.g., function pointers (and this is often done in other languages); however, templates allow us to avoid the runtime cost of dereferencing the function pointer by generating code

that directly calls the correct method. This cost can be non-negligible, especially in situations where the method is called many times, such as the `Evaluate()` method of an optimization problem.

mlpack uses policy-based design throughout the library, so that any functionality can be easily extended or modified by the user without needing to dig into the internals of the code. Our optimization framework is built around this paradigm, allowing for fast prototyping of either new optimizers or new objective functions.

3 Requirements for optimizers and functions

In general, we want to be able to consider any solver of the problem

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

for any function $f(x)$ that takes some vector input x . But it is impossible to implement something both so generic and fast—for instance, gradient-based approaches converge far more quickly than gradient-free approaches (in general), so we must design an abstraction that is able to simultaneously generalize to many problem types, as well as take advantage of accelerations and optimizations.

Let us describe the class of functions to be optimized with some non-exclusive properties:

- **arbitrary**: no assumptions can be made on $f(x)$
- **differentiable**: $f(x)$ has a computable gradient $f'(x)$
- **separable**: $f(x)$ is a sum of individual components: $f(x) = \sum_i f_i(x)$
- **categorical**: x contains elements that can only take discrete values
- **sparse**: the gradient $f'(x)$ or $f'_i(x)$ (for a separable function) is sparse
- **partially differentiable**: the gradient $f'_j(x)$ is computable for individual elements x_j of x
- **bounded**: x is limited in the values that it can take

Needless to say, it is impossible to create a system where every optimizer can work with every possible type of function: a gradient-based optimizer cannot reasonably be expected to work with an arbitrary function $f(x)$ where the gradient is not computable or available.

Instead, the best we can hope to achieve is to maximize the flexibility available, so that a user can easily implement a function $f(x)$ and have it work with as many optimizers as possible. For this, C++ policy-based design aids us greatly: when implementing a function to be optimized, a user can implement only a few methods and we can use C++ template metaprogramming to check that the given functions match the requirements of the optimizer that is being used. When implementing an optimizer, a user can assume that the given function to be optimized meets the required assumptions of the optimizers, and encode those requirements. Since we are using templates and C++, the resulting code generated by the compiler can be identical to what a developer would write if they were writing an optimizer specifically for the function $f(x)$ at hand—this can provide significant efficiency gains.

In some cases, some subclasses of functions can still be optimized with more general optimizers. For example, separable functions can still be optimized with an optimizer that does not specifically support them. Similarly, a sparse differentiable function may be optimized with any optimizer that supports differentiable functions; however, the sparseness might not be taken advantage of.

Now, with an understanding of the types of functions that we wish to support and their individual characteristics, we can develop an API that a given function can implement the relevant parts of.

4 FunctionType API

In order to facilitate consistent implementations, we have defined a `FunctionType` API that describes all the methods that an objective function may implement. mlpack offers a few variations of this API, each designed to cover some of the function characteristics of the previous section. Any `FunctionType` to be optimized requires the implementation of an `Evaluate()` method. The interface used for that can be one of the following two methods:

```
// For non-separable objectives.  
double Evaluate(const arma::mat& parameters);  
// For separable objectives.  
double Evaluate(const arma::mat& parameters,
```

```

    const size_t start,
    const size_t batchSize);

```

Both of these methods should calculate the objective given the parameters matrix `parameters`; however, the second overload is for *separable* functions, and should calculate the partial objective starting at the separable function indexed by `start` and calculate `batchSize` partial objectives and return the sum. Functions implementing the first overload are used by optimizers like the GradientDescent optimizer; functions implementing the second are used by SGD-like optimizers.

Note that, importantly, it is easy to calculate the objective for a non-separable function with the second overload just by setting `start` to 0 and `batchSize` to 1. Therefore, it is easy to make an ‘adapter’ that can allow separable optimizers to work with non-separable functions (and vice versa).

Next, any *differentiable* function must implement some `Gradient()` method.

```

// For non-separable differentiable sparse and non-sparse functions.
template<typename GradType>
void Gradient(const arma::mat& parameters, GradType& gradient);
// For separable differentiable sparse and non-sparse functions.
template<typename GradType>
void Gradient(const arma::mat& parameters,
              const size_t start,
              GradType& gradient,
              const size_t batchSize);

```

Both of these methods should calculate the gradient and place the results into the matrix object `gradient` that is passed as an argument. Note that the method accepts a template parameter `GradType`, which may be `arma::mat` (dense Armadillo matrix) or `arma::sp_mat` (sparse Armadillo matrix). This allows support for both sparse-supporting and non-sparse-supporting optimizers.¹

Next, if the objective function is *partially differentiable*, we can implement the following method:

```

// For partially differentiable sparse and non-sparse functions.
template<typename GradType>
void PartialGradient(const arma::mat& parameters, const size_t j, GradType& gradient);

```

This should calculate the gradient of the parameters `parameters` with respect to the parameter `j` and store the results (either sparse or dense) in the gradient matrix object.

In addition, *separable* and *partially differentiable* functions must implement the `NumFunctions()` and `NumFeatures()` functions, respectively:

```

// For separable functions: return the number of parts the optimization problem
// can be decomposed into.
size_t NumFunctions();
// For partially differentiable functions: return the number of partial derivatives.
size_t NumFeatures();

```

Finally, *separable* functions must implement the method `void Shuffle()`, which shuffles the ordering of the functions (note that the optimizer is not required to call it). This is useful for data-based problems, where it may not be desirable to loop over the separable objective functions—which usually correspond to individual data points—in the same order.

These simple functions, however, do not specify how to handle *categorical* or *bounded* functions. In those cases, the `FunctionType` should accept, in its constructor, its bounds and which dimensions are categorical or numeric. Since constraint types can differ greatly, it is up to an individual optimizer to define the format in which it should receive its constraints. More information can be found in the documentation of `mlpack`’s `LRSDP`, `AugLagrangian`, and `FrankWolfe` optimizers.

5 Optimizer API

In addition to implementing functions, users can also add new optimizers easily if they implement an optimizer with a simple API. Fortunately, the requirements for implementing optimizers are much simpler than for objective functions. An optimizer must implement only the method

¹One could write a non-templated `Gradient()` method for just `arma::mat`, and it would work fine for non-sparse optimizers. But to us it seems just as easy to template it.

```
template<typename FunctionType>
double Optimize(FunctionType& function, arma::mat& parameters);
```

The `Optimize()` method should check that the given `FunctionType` satisfies the assumptions the optimizer makes (see Section 7) and optimize the given function `function`, storing the best set of parameters in the matrix `parameters` and returning the best objective value.

6 Example function and optimizer

This section details the usage of `mlpack`'s optimization framework. In this example, we would like to minimize a simple function, where each dimension has a parabola with a distinct minimum. In this example, we show how to use `mlpack`'s framework for this task, and minimize the function below.

```
struct ObjectiveFunction {
  ObjectiveFunction() { // A separable function consisting of four quadratics.
    in = arma::vec("20 12 15 100"); bi = arma::vec("-4 -2 -3 -8");
  }

  size_t NumFunctions() { return 4; }
  void Shuffle() { ord = arma::shuffle(arma::uvec("0 1 2 3")); }

  double Evaluate(const arma::mat& para, size_t s, size_t bs) {
    double cost = 0;
    for (size_t i = s; i < s + bs; i++)
      cost += para(ord[i]) * para(ord[i]) + bi(ord[i]) * para(ord[i]) + in(ord[i]);
    return cost;
  }

  void Gradient(const arma::mat& para, size_t s, arma::mat& g, size_t bs) {
    g.zeros(para.n_rows, para.n_cols);
    for (size_t i = s; i < s + bs; i++)
      g(ord[i]) += (1.0 / bs) * 2 * para(ord[i]) + bi(ord[i]);
  }

  arma::vec in /* intercepts */, bi /* coeffs */; arma::uvec ord /* function order */;
};
```

Note that in this code, we maintain an ordering with the vector `ord`; in other situations, such as training neural networks, we could simply shuffle the columns of the data matrix in `Shuffle()`. This objective function will work with any `mlpack` optimizer that supports separable or differentiable functions; this includes all SGD-like optimizers, L-BFGS, simulated annealing, and others.

Next, we wish to define a simple example optimizer that can be used with `ObjectiveFunction` and other `mlpack` objective functions. For this, we must implement only an `Optimize()` method, and a constructor to set some parameters. The code is given below.

```
struct SimpleOptimizer {
  SimpleOptimizer(size_t bs = 1, double lr = 0.02) : bs(bs), lr(lr) { }

  template<typename FunctionType>
  double Optimize(FunctionType& function, arma::mat& parameter) {
    arma::mat gradient;
    for (size_t i = 0; i < 5000; i += bs) {
      if (i % function.NumFunctions() == 0) { function.Shuffle(); }
      function.Gradient(parameter, i % function.NumFunctions(), gradient, bs);
      parameter -= lr * gradient;
    }
    return function.Evaluate(parameter, 0, function.NumFunctions());
  }
  size_t bs; double lr; // Store batch size and learning rate internally.
};
```

Note that for the sake of brevity we have omitted checks on the batch size (this optimizer assumes that `function.NumFunctions()` is a multiple of the batch size) and other typical parts of real implementations, as well as any static type checking to ensure separability and differentiability. Still,

SimpleOptimizer can work with any mlpack objective function satisfying those conditions. This includes mlpack's neural network code, logistic regression, and other objective functions.

Now, we can find a minimum of ObjectiveFunction with SimpleOptimizer using this code:

```
ObjectiveFunction function; arma::mat parameter("0 0 0 0;");
SimpleOptimizer optimizer;
std::cout << "objective: " << optimizer.Optimize(function, parameter);
```

When we run this code, we receive the output: objective: 123.75

The final value of the objective function should be close to the optimal value, which is the sum of values at the vertices of the parabolas. This simple example, of course, does not discuss all the intricacies like a more complex learning rate update routine, but instead presents how the optimization framework could be used in a simple data science context. Adapting the example to a real-life application would be straightforward.

An important point to re-emphasize is that the use of templates and policy-based design allows easy control of behavior by users, simply by specifying template parameters—or writing custom classes when needed. In addition, there is no runtime performance penalty for this flexibility, as there would be when providing this type of support through inheritance or in other languages such as Python or C.

7 Statically checking function properties

Unfortunately, template metaprogramming can result in some very lengthy error messages. Therefore, we must be careful to ensure that a user is able to easily debug a problem when they implement an objective function or gradient. To improve error message output, we can use C++'s template metaprogramming support to determine what methods a type has available. Similarly, we can also use static compile-time constants to denote the methods that are required by a specific optimizer. This is implemented via SFINAE [16]. A `static_assert()` is raised when a given objective function does not implement the methods required by the optimizer used.

For instance, when attempting to use the L-BFGS optimizer without having a `Gradient()` function implemented, the user will receive a (comparatively) simple error message of the form:

```
error: static assertion failed: the FunctionType does not have a correct
definition of a Gradient() function
```

8 Supported optimizers and functions in mlpack

Thanks to the easy abstraction, we have been able to provide support for a large set of diverse optimizers and objective functions. Below is a list of what is currently available.

- **SGD variants:** Stochastic Gradient Descent (SGD), Stochastic Coordinate Descent (SCD), Parallel Stochastic Gradient Descent (Hogwild!), Stochastic Gradient Descent with Restarts (SGDR), SMORMS3, AdaGrad, AdaDelta, RMSProp, Adam, AdaMax
- **Quasi-Newton variants:** Limited-memory BFGS (L-BFGS), incremental Quasi-Newton method (IQN), Augmented Lagrangian Method
- **Genetic variants:** Conventional Neuro-evolution (CNE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
- **Other:** Conditional Gradient Descent, Frank-Wolfe algorithm, Simulated Annealing
- **Objective functions:** Neural Networks, Logistic regression, Matrix completion, Neighborhood Components Analysis, Regularized SVD, Reinforcement learning, Softmax regression, Sparse autoencoders, Sparse SVM

In addition, many methods are currently in development and will be released in the future.

9 Conclusion

We have identified that the support for generic and robust optimization is not currently available in most machine learning toolkits, and acted upon this observation to provide an easy framework for both implementing new optimizers and new objective functions to be optimized inside of the mlpack machine learning library. The framework provided by mlpack supports a wide array of special cases, and already has implemented specialized algorithms that outperform their classic generic alternatives.

References

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [3] François Chollet. keras. <https://github.com/fchollet/keras>, 2015.
- [4] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.
- [5] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.
- [6] Ryan R. Curtin and Marcus Edel. Designing and building the mlpack open-source machine learning library. *arXiv preprint arXiv:1708.05279*, 2017.
- [7] Jacob Goldberger, Geoffrey E Hinton, Sam T Roweis, and Ruslan R Salakhutdinov. Neighbourhood components analysis. In *Advances in neural information processing systems*, pages 513–520, 2005.
- [8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.
- [9] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: open source scientific tools for Python. 2014.
- [10] J. Langford, Li L., and A. Strehl. Vowpal wabbit open source project. Technical report, Yahoo!, 2007.
- [11] Mathworks. Matlab optimization toolbox, 2017. The MathWorks, Natick, MA, USA.
- [12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.
- [13] Conrad Sanderson and Ryan R. Curtin. Armadillo: C++ template metaprogramming for compile-time optimization of linear algebra. In *Platform for Advanced Scientific Computing (PASC) Conference*, 2017.
- [14] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [15] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. de Bona, A. Binder, C. Gehl, and V. Franc. The SHOGUN machine learning toolbox. *The Journal of Machine Learning Research*, 99:1799–1802, 2010.
- [16] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, 2002.
- [17] Kilian Q Weinberger and Lawrence K Saul. An introduction to nonlinear dimensionality reduction by maximum variance unfolding. In *AAAI*, volume 6, pages 1683–1686, 2006.