ensmallen: a flexible C++ library for efficient function optimization

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

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

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

Yannis Mentekidis
Independent Researcher
mentekid@gmail.com

Conrad Sanderson
Data61, CSIRO, Australia
University of Queensland, Australia
conradsand@ieee.org

Abstract

We present ensmallen, a fast and flexible C++ library for mathematical optimization of arbitrary user-supplied functions, which can be applied to many machine learning problems. Several types of optimizations are supported, including differentiable, separable, constrained, and categorical objective functions. The library provides many pre-built optimizers (including numerous variants of SGD and Quasi-Newton optimizers) as well as a flexible framework for implementing new optimizers and objective functions. Implementation of a new optimizer requires only one method and a new objective function requires typically one or two C++ functions. This can aid in the quick implementation and prototyping of new machine learning algorithms. Due to the use of C++ template metaprogramming, ensmallen is able to support compiler optimizations that provide fast runtimes. Empirical comparisons show that ensmallen is able to outperform other optimization frameworks (like Julia and SciPy), sometimes by large margins. The library is distributed under the BSD license and is ready for use in production environments.

1 Introduction

Mathematical optimization is the workhorse of virtually all machine learning algorithms. For a given objective function $f(\cdot)$ (which may have a special structure or constraints), almost all machine learning problems can be boiled down to the following optimization form:

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

Optimization is often computationally intensive and may correspond to most of the time taken to train a machine learning model. For instance, the training of deep neural networks is dominated by the optimization of the model parameters on the data [31]. Even popular machine learning models such as logistic regression have training times mostly dominated by an optimization procedure [13].

The ubiquity of optimization in machine learning algorithms highlights the need for robust and flexible implementations of optimization algorithms. We present ensmallen, a C++ optimization toolkit that contains a wide variety of optimization techniques for many types of objective functions. Through the use of C++ template metaprogramming [2, 34], ensmallen is able to generate efficient code that can help with the demanding computational needs of many machine learning algorithms.

Although there are many existing machine learning optimization toolkits, few are able to take explicit advantage of metaprogramming based code optimizations, and few offer robust support for various types of objective functions. For instance, deep learning libraries like Caffe [11], PyTorch [24], and TensorFlow [1] each contain a variety of optimization techniques. However, these techniques

are limited to stochastic gradient descent (SGD) and SGD-like optimizers that operate on small batches of data points at a time. Other machine learning libraries, such as `scikit-learn` [25] contain optimization algorithms but not in a coherent or reusable framework. Many programming languages have higher-level packages for mathematical optimization. For example, `scipy.optimize` [12], is widely used in the Python community, and MATLAB's function optimization support has been available and used for many decades. However, these implementations are often unsuitable for modern machine learning tasks—for instance, computing the full gradient of the objective function may not be feasible because there are too many data points.

In this paper, we describe the functionality of `ensmallen` and the types of problems that it can be applied to. We discuss the mechanisms by which `ensmallen` is able to provide both computational efficiency and ease-of-use. We show a few examples that use the library, as well as empirical performance comparisons with other optimization libraries.

`ensmallen` is open-source software licensed under the 3-clause BSD license [27], allowing unencumbered use in both open-source and proprietary projects. It is available for download from `https://ensmallen.org`. Armadillo [30] is used for efficient linear algebra operations, with optional interface to GPUs via NVBLAS [23].

2 Types of Objective Functions

`ensmallen` provides a **set of optimizers** for optimizing **user-defined objective functions**. It is also easy to implement a new optimizer in the `ensmallen` framework. Overall, our goal is to provide an easy-to-use library that can solve the problem $\operatorname{argmin}_x f(x)$ for any function $f(x)$ that takes a vector or matrix input x. In most cases, $f(x)$ will have special structure; one example might be that $f(x)$ is differentiable. Therefore, the abstraction we have designed for `ensmallen` can optionally take advantage of this structure. For example, in addition to $f(x)$, a user can provide an implementation of $f'(x)$, which in turn allows first-order gradient-based optimizers to be used. This generally leads to significant speedups.

There are a number of other properties that `ensmallen` can use to accelerate computation. These are listed below:

- **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 separable gradient $f'_i(x)$ is also separable for a different axis j
- **bounded**: x is limited in the values that it can take

	unified framework	constraints	batches	arbitrary functions	arbitrary optimizers	sparse gradients	categorical
`ensmallen`	●	●	●	●	●	●	●
Shogun [33]	●	-	●	●	●	-	-
Vowpal Wabbit [16]	-	-	●	-	-	-	●
TensorFlow [1]	●	-	●	◐	-	◐	-
Caffe [11]	●	-	●	◐	◐	-	-
Keras [4]	●	-	●	◐	◐	-	-
scikit-learn [25]	◐	-	◐	◐	-	-	-
SciPy [12]	●	●	-	●	-	-	-
MATLAB [19]	●	●	-	●	-	-	-
Julia (`Optim.jl`) [20]	●	-	-	●	-	-	-

Table 1: Feature comparison: ●= provides feature, ◐= partially provides feature, - = does not provide feature. *unified framework* indicates if there is some kind of generic/unified 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 if support for categorical features exists.

Due to its straightforward abstraction framework, `ensmallen` is able to provide a large set of diverse optimization algorithms for objective functions with these properties. Below is a list of currently available optimizers:

- **SGD variants** [29]: Stochastic Gradient Descent (SGD), SGD with Restarts, Parallel SGD (Hogwild!) [22], Stochastic Coordinate Descent, AdaGrad [6], AdaDelta [35], RMSProp, SMORMS3, Adam [13], AdaMax
- **Quasi-Newton variants**: Limited-memory BFGS (L-BFGS) [36], incremental Quasi-Newton method [21], Augmented Lagrangian Method [10]
- **Genetic variants**: Conventional Neuro-evolution [3], Covariance Matrix Adaptation Evolution Strategy [9]
- **Other**: Conditional Gradient Descent, Frank-Wolfe algorithm [8], Simulated Annealing [14]

In `ensmallen`'s framework, if a user wants to optimize a differentiable objective function, they only need to provide implementations of $f(x)$ and $f'(x)$, and then they can use any of the gradient-based optimizers that `ensmallen` provides. Table 1 contrasts the classes of objective functions that can be handled by `ensmallen` and other popular frameworks and libraries.

Not every optimization algorithm provided by `ensmallen` can be used by every class of objective function; for instance, a gradient-based optimizer such as L-BFGS cannot operate on a non-differentiable objective function. Thus, the best the library can attain 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.

To accomplish the flexibility, `ensmallen` makes heavy use of C++ template metaprogramming. When implementing an objective function to be optimized, a user may only need to implement a few methods; metaprogramming is then automatically used to check that the given functions match the requirements of the optimizer that is being used. When implementing an optimizer, we can assume that the given function to be optimized meets the required assumptions of the optimizers, and encode those requirements as compile-time checks (via `static_assert`); this can provide much easier-to-understand error messages than typical compiler output for templated C++ code.

For the most common case of a differentiable $f(x)$, the user only needs to implement two methods:

- `double Evaluate(`x`)`: given coordinates x, this function returns the value of $f(x)$.
- `void Gradient(`x`, `g`)`: given coordinates x and a reference to g, set $g = f'(x)$.

Alternatively, the user can simply implement a `EvaluateWithGradient()` function that computes both $f(x)$ and $f'(x)$ simultaneously, which is useful in cases where both the objective and gradient depend on similar computations.

The required API for separable differentiable objective functions (i.e. those that would use an optimizer like SGD) is very similar, except that `Evaluate()`, `Gradient()` and `EvaluateWithGradient()` should operate only on mini-batches, and utility methods `Shuffle()` and `NumFunctions()` must be added. The same pattern applies for other types of objective functions: only a few methods specific to class of objective function itself must be implemented and then any optimizer may be used.

3 Example: Learning Linear Regression Models

As an example of usage, consider the linear regression objective function[1]. Given a dataset $X \in \mathcal{R}^{n \times d}$ and associated responses $y \in \mathcal{R}^n$, the model of linear regression is to assume that $y_i = x_i \theta$ for each point and response (x_i, y_i). To fit this model $\theta \in \mathcal{R}^d$ to the data, we must find

$$\underset{\theta}{\operatorname{argmin}} f(\theta) = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - x_i \theta)^2 = \underset{\theta}{\operatorname{argmin}} \|y - X\theta\|_F^2. \tag{2}$$

The objective function $f(\theta)$ has the associated gradient

$$f'(\theta) = \sum_{i=1}^{n} -2x_i(y_i - x_i\theta) = -2X^T(y - X\theta). \tag{3}$$

We can implement these two functions in a class named `LinearRegressionFunction`, as shown in Fig. 1. This is the entire required implementation to optimize the linear regression model with any of the gradient-based optimizers in `ensmallen`.

[1]For simplicity, we ignore the bias term. It can be rederived by taking $x_i^* = (x_i, 1)$.

```
class LinearRegressionFunction {
 public:
  // Construct the LinearRegressionFunction with the given data
  LinearRegressionFunction(arma::mat& X_in, arma::vec& y_in) : X(X_in), y(y_in) {}

  // Compute the objective function
  double Evaluate(const arma::mat& theta) {
    return std::pow(arma::norm(y - X * theta), 2.0);
  }
  // Compute the gradient and store in 'gradient'
  void Gradient(const arma::mat& theta, arma::mat& gradient) {
    gradient = -2 * X.t() * (y - X * theta);
  }
 private:
  arma::mat& X; arma::vec& y;
};
```

Figure 1: Implementation of objective and gradient functions for linear regression, used by optimizers in `ensmallen`. The types `arma::mat` and `arma::vec` are matrix and vector types from Armadillo [30].

Given the user-defined `LinearRegressionFunction` class, the code snippet below shows how the L-BFGS optimizer can be used to find the best parameters θ:

```
LinearRegressionFunction lrf(X, y); // we assume X and y already hold data
ens::L_BFGS lbfgs; // create L-BFGS optimizer with default parameters

arma::vec theta(X.n_rows, arma::fill::randu); // random uniform initialization
lbfgs.Optimize(lrf, theta); // after this call, theta holds the solution
```

To use the small-batch SGD-like optimizers provided by `ensmallen`, only a slight variation on the signature of `Evaluate()` and `Gradient()` would be needed, plus the `Shuffle()` and `NumFunctions()` utility methods.

4 Automatic Metaprogramming for Ease of Use and Efficiency

When optimizing a given function $f(x)$, the computation of the objective function $f(x)$ and its derivative $f'(x)$ often involve the computation of identical intermediate results. Consider the linear regression objective function described in Section 3, $f(\theta) = \|y - X\theta\|_F^2$. For this objective function, the derivative $f'(\theta)$ has a related form of $-2X^T(y - X\theta)$. Both the objective function and the derivative depend on the computation of the vector term $(y - X\theta)$, which can be computationally expensive to compute if X is a large matrix. Existing optimization frameworks do not have an easy way to avoid this duplicate computation. In many cases, an optimization algorithm may need the values of both $f(\theta)$ and $f'(\theta)$ for a given θ.

Using template metaprogramming, `ensmallen` provides an easy (and optional) way for users to avoid this extra computational overhead. Instead of specifying individual `Evaluate()` and `Gradient()` functions, a user may simply write an `EvaluateWithGradient()` function that returns both the objective value and the gradient value for an input θ. As an example, for the `LinearRegressionFunction` class in Fig. 1, we can replace `Evaluate()` and `Gradient()` with an implementation of `EvaluateWithGradient()` that computes $(y - X\theta)$ only once:

```
double EvaluateWithGradient(const arma::mat& theta, arma::mat& gradient) {
  const arma::vec v = (y - X * theta); // Cache result
  gradient = -2 * X.t() * v;  // Store gradient in the provided matrix
  return arma::accu(v % v); // Take squared norm of v
}
```

Template metaprogramming techniques are automatically used to detect which methods exist, and a wrapper class will use suitable mix-ins in order to provide 'missing' functionality [32]. For instance, if `EvaluateWithGradient()` is not provided, a version will be automatically generated that calls both `Gradient()` and `Evaluate()` in turn. Similarly, if `Evaluate()` or `Gradient()` does not exist, then `EvaluateWithGradient()` is called, and the unnecessary part of the result will be discarded.

The use of template metaprogramming in this manner also allows for compiler optimizations that would not otherwise be possible (and that are often not possible in other frameworks). Firstly, because the objective function class itself is a template parameter, the compiler is able to avoid the overhead of a function pointer dereference, which would not be easily possible when using a language with virtual

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

Table 2: Runtimes for 100000 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.

inheritance. The compiler is also able to use inlining and any optimizations that may imply, including removing temporary values and dead code elimination. Further, if `ensmallen` automatically generates an `Evaluate()` or `Gradient()` method from a user-supplied `EvaluateWithGradient()` method, the compiler can in some cases recognize and remove the computation of unnecessary results. For instance, in an automatically generated `Evaluate()` method, the computation of the gradient from `EvaluateWithGradient()` can be avoided entirely.

Overall, the automatic code generation functionality in `ensmallen` reduces the requirements for users when they are implementing their own objective functions to be optimized, and allows users a way to provide more efficient implementations of their objective functions. This leads to quicker development, quicker results, and reduces the likelihood of bugs.

At the time of writing, the automatic code generation is implemented for the most commonly-used cases: full-batch and small-batch `Evaluate()`, `Gradient()`, and `EvaluateWithGradient()`. We aim to expand this support to other sets of methods for other types of objective functions.

5 Experiments

To demonstrate the benefits of the metaprogramming based code optimizations that `ensmallen` can exploit, we compare the performance of `ensmallen` with several other optimization frameworks, including some that use automatic differentiation.

For our first experiment, we consider the simple and popular Rosenbrock function [28]: $f([x_1, x_2]) = 100(x_2 - x_1^2)^2 + (1 - x_1^2)$. For the optimizer, we use simulated annealing [14], a gradient-free optimizer. Simulated annealing will call the objective function numerous times; for each simulation we limit the optimizer to 100k objective evaluations. Since the objective function is straightforward and is called many times, this can help us understand the overheads of various frameworks.

We compare four frameworks for this task: **(i)** `ensmallen`, **(ii)** `scipy.optimize.anneal` from SciPy 0.14.1 [12], **(iii)** simulated annealing implementation in `Optim.jl` with Julia 1.0.1 [20], **(iv)** `samin` in the `optim` package for Octave [7]. 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.

Initially, we implemented these functions as simply as possible and ran them without any tuning. This reflects how a typical user might interact with a given framework. The results are shown in the first row of Table 2. Only Julia and `ensmallen` are compiled, and thus are able to avoid the function pointer dereference and take advantage of inlining and related optimizations.

However, both Python and Octave have routes for acceleration, such as Numba [15], 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 2. 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.

Next, we consider the linear regression example described in Sec. 3. For this task we use the first-order L-BFGS optimizer [36]. Using the same four packages, we implement the linear regression objective and gradient. 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` [26] package and for Python via the `Autograd` [18] package. For GNU Octave we use the `bfgsmin()` function.

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.

Results for various data sizes are shown in Table 3. For each implementation, L-BFGS was allowed to run for only 10 iterations and never converged in fewer iterations. The 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 considerable 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, 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.

Lastly, we demonstrate the easy pluggability in `ensmallen` for using various optimizers on the same task. Using a version of `LinearRegressionFunction` from Sec. 3 adapted for separable differentiable optimizers, we run six optimizers with default parameters in just 8 lines of code, as shown in Fig. 2(a). Applying these optimizers to the `YearPredictionMSD` dataset from the UCI repository [17] yields the learning curves shown in Fig. 2(b). Any other optimizer for separable differentiable objective functions can be dropped into place in the same manner.

6 Conclusion

We have described `ensmallen`, a flexible C++ library for function optimization that provides an easy interface for the implementation and optimization of user-defined objective functions. Many types of functions can be optimized, including separable and constrained functions. The library provides many pre-built optimizers (including numerous variants of SGD and Quasi-Newton optimizers). The library internally exploits template metaprogramming to maximise opportunities for efficiency gains, as well as to make the implementation of objective functions easier by automatically generating missing methods.

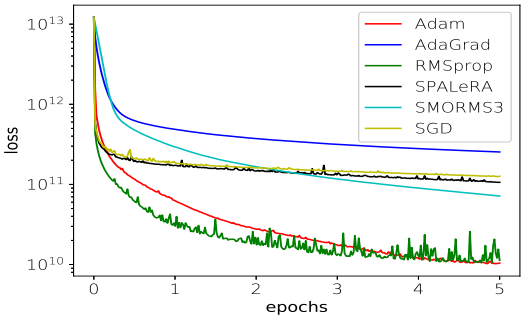
We aim to expand the library with further optimization techniques as the need arises. Since `ensmallen` is open source, external contributions to the codebase are welcome. For more information on the library, see the website and documentation at `https://ensmallen.org`. The library is already in use for function optimization in the `mlpack` machine learning toolkit [5].

Acknowledgements. We would like to thank the many contributors to `ensmallen`, who are listed on the associated website.

```
// X and y are data
LinearRegressionFunction lrf(X, y);

using namespace ens;
StandardSGD<>().Optimize(lrf, sgdModel);
Adam().Optimize(lrf, adamModel);
AdaGrad().Optimize(lrf, adagradModel);
SMORMS3().Optimize(lrf, smorms3Model);
SPALeRASGD().Optimize(lrf, spaleraModel);
RMSProp().Optimize(lrf, rmspropModel);
```

(a) Code.



(b) Learning curves.

Figure 2: Example usage of six `ensmallen` optimizers to optimize a linear regression function on the `YearPredictionMSD` dataset [17] for 5 epochs of training. The optimizers can be tuned for better performance.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[3] R. K. Belew, J. McInerney, and N. N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning, 1990. CSE Technical Report #CS90-174, University of California at San Diego.

[4] F. Chollet. Keras. https://github.com/fchollet/keras, 2015.

[5] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software*, 3:726, 2018.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[7] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 4.4.0 manual: a high-level interactive language for numerical computations*, 2018.

[8] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[9] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[10] M. R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4(5):303–320, 1969.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia*, pages 675–678, 2014.

[12] E. Jones, T. Oliphant, and P. Peterson. SciPy: open source scientific tools for Python, 2014. http://www.scipy.org/.

[13] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[14] S. Kirkpatrick, C. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[15] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based python JIT compiler. In *Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7, 2015.

[16] J. Langford, L. Li, and A. Strehl. Vowpal wabbit open source project. Technical report, Yahoo!, 2007.

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

[18] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. In *AutoML Workshop at ICML*, 2015.

[19] Mathworks. Matlab optimization toolbox, 2017. The MathWorks, Natick, MA, USA.

[20] P. K. Mogensen and A. N. Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018.

[21] A. Mokhtari, M. Eisen, and A. Ribeiro. IQN: An incremental quasi-newton method with local superlinear convergence rate. *SIAM Journal on Optimization*, 28(2):1670–1698, 2018.

[22] F. Niu, B. Recht, C. Ré, and S. J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, 2011.

[23] NVIDIA. NVBLAS Library. http://docs.nvidia.com/cuda/nvblas, 2015.

[24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. *Autodiff Workshop at NIPS*, 2017.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[26] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892*, 2016.

[27] L. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, 2004.

[28] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3:175–184, 1960.

[29] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 2016.

[30] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[31] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[32] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. In *International Symposium on Generative and Component-Based Software Engineering, Lecture Notes in Computer Science*, volume 2177, pages 164–178. Springer, 2000.

[33] 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. *Journal of Machine Learning Research*, 11:1799–1802, 2010.

[34] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2nd edition, 2017.

[35] M. D. Zeiler. ADADELTA: An adaptive learning rate method. *arXiv:1212.5701*, 2012.

[36] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.