Auto-Differentiating Linear Algebra

Matthias Seeger
Amazon
matthis@amazon.de

Asmus Hetzel
Amazon
ahhetzel@amazon.de

Zhenwen Dai
Amazon
zhenwend@amazon.co.uk

Neil Lawrence
Amazon
lawrennd@amazon.co.uk

Abstract

Development systems for deep learning, such as Theano, Torch, TensorFlow, or MXNet, are easy-to-use tools for creating complex neural network models. Since gradient computations are automatically baked in, and execution is mapped to high performance hardware, these models can be trained end-to-end on large amounts of data. However, it is currently not easy to implement many basic machine learning primitives in these systems (such as Gaussian processes, least squares estimation, principal components analysis, Kalman smoothing), mainly because they lack efficient support of linear algebra primitives as differentiable operators. We detail how a number of matrix decompositions (Cholesky, LQ, symmetric eigen) can be implemented as differentiable operators. We have implemented these primitives in MXNet, running on CPU and GPU in single and double precision. We sketch use cases of these new operators, learning Gaussian process and Bayesian linear regression models. Our implementation is based on BLAS/LAPACK APIs, for which highly tuned implementations are available on all major CPUs and GPUs.

1 Introduction

Deep neural networks, trained on vast amounts of data, have recently revolutionized several machine learning applications, ranging from object classification and detection in computer vision over large vocabulary speech recognition to machine translation of natural language text. Apart from large amounts of data and high-performance computing hardware, a major driver of this success has been the flexibility and ease of use of modern deep learning development systems (DLDS), such as Theano, Torch, TensorFlow, or MXNet. These systems map algorithms to *computation graphs*, where nodes are multi-dimensional arrays, and vertices are differentiable operators. Given this abstraction, gradients w.r.t. any node are obtained *automatically* by way of reverse mode differentiation, a generalization of error backpropagation. While some DLDS maximize flexibility by unfolding the graph on the fly, others may apply graph optimization in order to minimize runtime or memory usage on a specific target hardware. All a user has to do is to specify this symbolic graph, or even just re-combine existing components, and bind it to input data. Not only has this simple means of specifying a complex model lowered the technical bar to entry into cutting-edge ML, it is also dramatically shortening the time from idea to large data experimentation, which is so crucial for data-driven innovation.

Given this massive advance in ease of modeling, it is remarkable to observe how many "pre-deep-learning" algorithmic primitives remain out of reach of current DLDS. Here are some examples, which can be found in any ML standard textbook [2, 1, 4, 6]:

- Least squares estimation, solving linear systems

Submission to Autodiff Workshop, NIPS 2017.

- Gaussian process models
- Principal components analysis, linear discriminant analysis, canonical correlation analysis
- Kalman filtering and smoothing in linear dynamical systems

Why are none of these easily available in most DLDS, as are LSTM or Inception modules? Inspecting the former algorithms, they make use of certain linear algebra primitives, such as Cholesky decomposition, backsubstitution, LQ decomposition, or symmetric eigendecomposition. As we will demonstrate in this work, *this limited number of linear algebra primitives is all that is missing* in order to cover a substantial number of ML cornerstone algorithms, *as long as they are available as differentiable operators*.

In this work, we show how the following linear algebra primitives can be incorporated into a DLDS as fully differentiable operators:

- Cholesky decomposition, backsubstitution
- LQ decomposition[1]
- Symmetric eigendecomposition

We contributed implementations for all operators detailed here to the MXNet deep learning development system[2]. Our implementations work both on CPUs and GPUs, and can be used in `float64` or `float32`. We provide a number of concrete examples of how core ML methodology is implemented on top of our novel operators.

There are essentially two ways of including complex linear algebra operators into a DLDS. First, we can re-implement the forward computation in terms of primitives of the DLDS, relying on autograd for backward. Second, we can derive the backward expression, and implement both forward and backward ourselves. The first strategy is often propagated in the automatic differentiation research field, with the final goal of transforming native-language forward code such that gradients are obtained as well. For our work, we chose the latter strategy. While a bit more effort is needed for the derivations, we do not have to re-implement anything difficult, but can instead build on top of existing highly tuned numerical linear algebra software (implementations of BLAS and LAPACK), available for both CPUs and GPUs. As we will demonstrate, this opens the door to incorporating LQ and eigendecomposition into DLDS, robust implementations of which go far beyond of what can be re-implemented with reasonable effort and limited expertise in numerical mathematics.

Full details of our work, including a discussion of related work and all derivations, can be found in [7]. In the sequel, we sketch some machine learning applications which are realized in MXNet using our operators, and provide some details about the linear algebra operators and their implementation in MXNet.

2 Machine Learning Examples

In this section, we provide some examples for how a small number of linear algebra operators can enable a range of machine learning methodologies inside a DLDS (we make use of MXNet). Details are found in [7], and complete code is given in Jupyter notebooks referenced below.

2.1 Gaussian Processes

Powerful non-parametric regression and classification models are obtained by representing unknown functions by Gaussian processes (GPs). Details about GPs can be found in [6]. Say we want to fit a random function $f(\boldsymbol{x})$ to data $\{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$, where $y_i \in \mathbb{R}$. Free hyperparameters, such as kernel parameters or noise variance, can be learned by minimizing the negative log marginal likelihood of the observed data, given by:

$$\phi = -\log N(\boldsymbol{y}|\boldsymbol{0}, \boldsymbol{A}) = \frac{1}{2}\left(\boldsymbol{y}^T \boldsymbol{A}^{-1} \boldsymbol{y} + \log|2\pi \boldsymbol{A}|\right), \quad \boldsymbol{A} = \boldsymbol{K} + \lambda_y \boldsymbol{I}, \quad \boldsymbol{y} = [y_i].$$

[1] Transpose of the more well-known QR decomposition

[2] `https://github.com/apache/incubator-mxnet`

Here, $\boldsymbol{K} = [K(\boldsymbol{x}_i, \boldsymbol{x}_j)]_{i,j} \in \mathbb{R}^{n \times n}$ is the kernel matrix, and λ_y is the noise variance. Following [6]:

$$\phi = \frac{1}{2}\left(\|\boldsymbol{z}\|^2 + n\log(2\pi)\right) + \log|\boldsymbol{L}|, \quad \boldsymbol{z} = \boldsymbol{L}^{-1}\boldsymbol{y},$$

where $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$ is the Cholesky factorization. ϕ is easily implemented in MXNet, using our differentiable operators `linalg.potrf` (compute Cholesky factor $\boldsymbol{L}$ of matrix $\boldsymbol{A}$) and `linalg.trsm` (compute $\boldsymbol{z} = \boldsymbol{L}^{-1}\boldsymbol{y}$). A Jupyter notebook that implements the full GP model is available at `http://github.com/ARCambridge/MXNet_linalg_examples`.

2.2 Least Squares Estimation. Bayesian Linear Regression

The linear regression problem, also known as least squares estimation, is a cornerstone of applied mathematics. The gold-standard approach to solving the corresponding normal equations is applying the LQ decomposition (or its transpose, the QR decomposition). Here, we focus on *Bayesian linear regression* [2, Sect. 3.5], which is closely related to ℓ_2 regularized least squares estimation.[3]

In [7], we detail an MXNet model graph for the negative log marginal likelihood of a BLR model, which we can optimize for hyperparameter and feature learning. While this could once more be computed with a Cholesky factorization, it is well known that both least squares estimation and BLR training are much more numerically robust if expressed in terms of the LQ factorization. We do so in [7], using our novel MXNet operator `linalg.gelqf`. A Jupyter notebook that implements the Bayesian linear model is available at `http://github.com/ARCambridge/MXNet_linalg_examples`.

Note that numerical robustness is particularly important in this context. If the BLR features are represented by a DNN, computations will run efficiently on a GPU only with the `float32` datatype (single precision). Our LQ factorization expression of the final BLR layer uses matrices of much smaller condition number, so we can compute them in `float32`, while the equivalent expression in terms of a Cholesky factorization would normally require double precision computations.

2.3 Kalman Filtering

A final machine learning example is Kalman filtering for inference in Gaussian linear dynamical systems. A complete MXNet implementation can be found at `http://gluon.mxnet.io/chapter12_time-series/lds-scratch.html`. It makes use of our operators `linalg.potrf` and `linalg.trsm`.

3 Linear Algebra Operators

We provide full details and derivations of differentiable operators for Cholesky factorization, LQ factorization, symmetric eigendecomposition, triangular backsubstitution, and support functions in [7]. Here, we just sketch the backward (or pullback) expression for the LQ factorization, which is novel to our knowledge (the Cholesky backward appeared in [5], the eigendecomposition in [3], but they do not provide implementations).

Recall that reverse mode differentiation (RMD) is a simple algorithm operating on a bipartite computation graph, whose nodes are variables (multi-dimensional tensors) and operators. Each operator is defined by *forward* and *backward* (the latter is also known as "pullback"). For the LQ factorization, *forward* is:

$$(\boldsymbol{Q}, \boldsymbol{L}) = \text{gelqf}(\boldsymbol{A}), \quad \boldsymbol{A} = \boldsymbol{L}\boldsymbol{Q}, \quad \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I}_m, \quad l_{ij} = 0\ (i < j).$$

Here, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, where $m \le n$, $\boldsymbol{Q} \in \mathbb{R}^{m \times n}$ is row-orthonormal, and $\boldsymbol{L} \in \mathbb{R}^{m \times m}$ is lower triangular with non-zero diagonal. We require that $\boldsymbol{A}$ has full rank m, as otherwise derivatives are not defined. In our implementation, $\boldsymbol{Q}$ can overwrite $\boldsymbol{A}$.

The *backward* mapping has inputs $\bar{\boldsymbol{Q}} = \partial_{\boldsymbol{Q}}\phi$, $\bar{\boldsymbol{L}} = \partial_{\boldsymbol{L}}\phi$ (output gradients), $\boldsymbol{Q}$, $\boldsymbol{L}$ (outputs of forward), its output is the input gradient $\bar{\boldsymbol{A}} = \partial_{\boldsymbol{A}}\phi$. Here, ϕ is the loss function given by the full

[3] BLR can be seen as a neural network, where the weights of the final linear layer are random variables, which are integrated out.

graph. By the rules of differential calculus, the backward mapping does not depend on ϕ beyond the local operator. We have:

$$\bar{\boldsymbol{A}} = \boldsymbol{L}^{-T}\left(\bar{\boldsymbol{Q}} + \text{copyltu}(\boldsymbol{M})\boldsymbol{Q}\right), \quad \boldsymbol{M} = \boldsymbol{L}^T\bar{\boldsymbol{L}} - \bar{\boldsymbol{Q}}\boldsymbol{Q}^T.$$

Here, copyltu($\boldsymbol{X}$) for a square matrix $\boldsymbol{X}$ generates a symmetric matrix by copying the lower triangle to the upper triangle: $[\text{copyltu}(\boldsymbol{X})]_{ij} = x_{\max(i,j),\min(i,j)}$. Except for a temporary $\mathbb{R}^{m\times m}$ matrix, this expression can be computed in-place, and $\bar{\boldsymbol{A}}$ can overwrite $\bar{\boldsymbol{Q}}$.

Our operators are implemented by calling tuned numerical linear algebra code, adhering to the BLAS and LAPACK APIs. For our MXNet implementation, we adopt their naming scheme. For the LQ factorization, we have to call two LAPACK routines: `gelqf`, followed by `orglq`. The former returns an internal representation of the $\boldsymbol{Q}$ matrix. Also, the routines require working space, the amount of which is determined by issuing workspace queries. For GPU, we use the CUBLAS and CUSolver libraries from NVidia. Note that internally, we have to switch between row-major ordering (MXNet, NumPy) and column-major ordering (BLAS, LAPACK). In order to avoid unnecessary transpositions and memory allocation, we simply compute the QR factorization of $\boldsymbol{A}^T$ inside.

4 Discussion

In the full version [7] of this abstract, we show how to introduce advanced linear algebra routines, such as Cholesky factorization, backsubstitution, LQ factorization, and symmetric eigen decomposition, as differentiable operators into a deep learning development system (DLDS). Implemented in a time and memory efficient manner, they open the door to realizing a host of "pre-deep" machine learning models within a DLDS, and enable combinations of Gaussian processes, Bayesian linear models, or linear dynamical systems with deep multi-layer maps. We provide a range of machine learning examples, complete notebooks for which are available for download.

All operators mentioned here are implemented in MXNet (`https://github.com/apache/incubator-mxnet`). In contrast to similar operators in TensorFlow and Theano, our implementation is tuned for maximum memory efficiency. We call highly optimized BLAS and LAPACK library functions directly on the C++ level. In particular, we take care to use the minimum required amount of temporary storage. Most of our operators are implemented in-place, meaning that no additional memory is needed beyond inputs and outputs. If memory-intensive machine learning or scientific computing methods (such as Gaussian processes, least squares estimation, Kalman smoothing, or principal components analysis) are to be powered by easy-to-use DLDS, it will be essential to implement key linear algebra operators with the same level of care that is applied to convolution or LSTM layers today.

References

[1] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 1st edition, 2012.

[2] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1st edition, 2006.

[3] M. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation. Springer LNCSE 64*, pages 35–44, 2008.

[4] K. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. MIT Press, 1st edition, 2012.

[5] I. Murray. Differentiation of the Cholesky decomposition. Technical Report 1602.07527v1 [stat.CO], ArXiv, 2016.

[6] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[7] M. Seeger, A. Hetzel, Z. Dai, and N. Lawrence. Auto-differentiating linear algebra. Technical Report arXiv:1710.08717 [cs.MS], ArXiv, 2017.