Optimizing Matrix Computations for Learning

(and learning to optimize matrix computations)

Oliver Williams OLLIEW@MICROSOFT.COM
Microsoft Research
Mountain View, CA.

Andrew Fitzgibbon AWF@MICROSOFT.COM
Microsoft Research
Cambridge, UK.

Matrix-vector notation is the predominant idiom in which machine learning formulae are expressed; some models, like Gaussian processes (Rasmussen & Williams, 2006), would be extremely difficult to describe without it. Turning a matrix expression into a computer program is not always easy, however. Although good implementations of primitive matrix operations are available (Golub & Van Loan, 1996) as are packages like MATLAB[1], which provide a high-level interface to these primitives, two important tasks must still be carried out manually: *(i)* computing derivatives of matrix functions and *(ii)* turning a matrix expression into an efficient computer program. Not having tools to do this can and does harm research: even for the relatively simple example of fitting a linear regression model with gradient methods, the number of types and combinations of basis functions a researcher can experiment with is limited by the need to manually differentiate the objective function and write code for each version. We have addressed these issues by combining a symbolic matrix algebra engine with a superoptimizing compiler: an interesting learning problem in itself. We call our system Coconut.

Coconut transforms matrix expressions into fast programs for computing them. For the most part, Coconut treats an expression symbolically and by maintaining the semantic structure of a matrix expression, high-level knowledge can be used to provide two powerful features: differentiation and simplification. Differentiation requires the recursive application of the rules of matrix calculus (Magnus & Neudecker, 1999). This blind application frequently results in large expressions that are inefficient to compute: the real power of Coconut is its ability to simplify expressions.

We define simplification as the transformation of a source expression into one with equivalent output, but lower computational cost. To do this, Coconut is provided with a list of matrix identities (e.g., the matrix inversion lemma) and by matching identities to a source expression, a new set of equivalent expressions is generated. Performing this recursively gives rise to an infinite graph of equivalent matrix expressions, in which we would like to find the node with the minimum cost. Greedy application of identities, which is effectively `peephole' optimization in the compiler sense (Cooper & Torczon, 2003), performs very poorly in this domain. Even basic simplifying actions, such as expanding and collecting terms, reveal that frequent `uphill' moves are needed to make significant `downhill' progress. We approach this problem as one of reinforcement learning in which an `agent' traverses the

1. http://www.mathworks.com/products/matlab/

graph is search of good local optima. Initially, our agent performs a random walk around the graph by applying matching identities at random. However, by recording how trajectories affect the objective function, we are able to learn an improved proposal distribution from which future trajectories are sampled.

While existing symbolic algebra packages, e.g., Maple[2] provide differentiation, simplification and code generation functionality, they handle matrices at the level of scalar entries. This is limiting in two ways. First, problem sizes are severely restricted: matrices of sizes much larger than 10×10 overflow modern computers. More importantly, the derivative of an expression such as AX^{-1} is computed by explicitly expanding the inverse, leading to computationally unstable as well as enormous code. In contrast, Coconut treats scalars as special cases of matrices (not the other way around), and maintains the separation between a formula and the algorithm used to implement it, so that stable and efficient numerical methods (e.g. for computing AX^{-1}) can be employed. Automatic differentiation (Rall, 1981) is another alternative to our approach, but this is slow and memory-hungry when dealing with a large number of variables.

We demonstrate Coconut's effectiveness by fitting a Gaussian mixture model to data under a variety of priors, and by learning a Gaussian process regression, both using Newton's method. The only programming required to do this is the definition of the objective function $f(x)$. Differentiation to obtain the gradient $\nabla f(x)$ and the Hessian $\nabla^2 f(x)$, simplification and code generation are all performed automatically.

References

Cooper, K., & Torczon, L. (2003). *Engineering a Compiler*. Morgan Kaufmann.

Golub, G., & Van Loan, C. (1996). *Matrix Computations*. Johns Hopkins University Press.

Magnus, J., & Neudecker, H. (1999). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Wiley.

Rall, L. (1981). *Automatic Differentiation: Techniques and Applications*. Springer Verlag.

Rasmussen, C., & Williams, C. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

2. http://www.maplesoft.com/