Vectorizing Mathematical Expressions (Extended Abstract)

Joachim Giesen, Julien Klaus, Sören Laue*

Friedrich-Schiller-University Jena, Germany
{joachim.giesen,julien.klaus,soeren.laue}@uni-jena.de

Abstract. Mathematical expressions are ubiquitous in science, engineering and business. They are mostly given in index form, where elements of a data set are addressed by an index. A different, but equivalent representation for mathematical expressions is their vectorized form, where data sets are represented by vectors or matrices. Typically the vectorized representation of an expression can be evaluated much faster, because it can easily be mapped onto highly tuned libraries for basic linear algebra subroutines (BLAS). Evaluating expressions in vectorized form can be a few orders of magnitude faster than evaluating the same expression in index form.
In this paper we present a tool that transforms a mathematical expression in index form, into an equivalent vectorized form. We define a simple grammar for index form expressions, which is parsed into an expression tree. The expression tree is then transformed into another tree that only contains tensors and operations on tensors. Finally the vectorized expression is derived by applying simple substitution rules on the tensor tree. Numerical tests demonstrate the efficiency and correctness of our approach.

Keywords: mathematical expressions, vectorizing, vector unit

1 Introduction

Interpreted languages like Matlab or Python often suffer from slow execution of loops [4,5]. Loops occur naturally in mathematical expressions containing, among others, a universal quantifier or a sum symbol. Two examples of mathematical problems that contain these symbols are logistic regression (LR) or support vector machines (SVM). Both problems are used for classifying labeled data points. Given a data matrix $X \in \mathbb{R}^{m\times n}$ and a label vector $y \in \{-1,+1\}^m$, the goal of both problems is finding a hyperplane that separates the points with label -1 from the points with label $+1$. A separating hyperplane $\{x \in \mathbb{R}^n | x^\top w + b = 0\}$ is represented by $w \in \mathbb{R}^n$ the normal vector to the hyperplane and the offset $b \in \mathbb{R}$. Note that $\frac{|b|}{\|w\|}$ is the distance of the hyperplane from the origin.

* Sören Laue acknowledges the support of Deutsche Forschungsgemeinschaft (DFG) under grant LA-2971/1-1.

The logistic regression problem reads as:

$$\min_{w \in \mathbb{R}^m} \quad \sum_{i=1}^{n} \log\left(1 + \exp\left(\sum_{j=1}^{m} -y_i \cdot w_j \cdot X_{ij} + b\right)\right), \tag{LR}$$

and the support vector machine problem is the following optimization problem:

$$\begin{aligned} \min_{w,\xi} \quad & \frac{1}{2}\sum_{i=1}^{n} w_i^2 + c\sum_{i=1}^{m} \xi_i \\ \text{s.t.} \quad & \forall i : \sum_{j=1}^{m} y_i \cdot (w_j \cdot X_{ij} + b) \geq 1 - \xi_i \\ & \forall i : \xi_i \geq 0. \end{aligned} \tag{SVM}$$

The equivalent representation for the logistic regression in vectorized form is the following:

$$\min_{w \in \mathbb{R}^m} \quad \mathbb{1}^{\mathrm{T}} \cdot \log(\mathbb{1} + \exp(-y \odot (X \cdot w + \mathbb{1} \cdot b))), \tag{LRV}$$

and the support vector machine is given as:

$$\begin{aligned} \min_{w,\xi} \quad & \frac{1}{2} \cdot \|w\|^2 + c \cdot \mathbb{1}^{\mathrm{T}} \cdot \xi \\ \text{s.t.} \quad & \mathbb{1}^{\mathrm{T}} \cdot (y \odot (X \cdot w + \mathbb{1} \cdot b)) \geq \mathbb{1} - \xi \\ & \xi \geq \mathbb{0}, \end{aligned} \tag{SVMV}$$

where $\mathbb{0}$ and $\mathbb{1}$ are the vectors with only zeros and ones, $\|\cdot\|^2$ the Euclidean norm and $\odot$ is the element-wise multiplication. We are interested in a tool, that derives the vectorization automatically.

Usually two nested loops over the indices i and j are used to evaluate the expressions from above. It has been suggested by many authors [2,7] that the problems should be vectorized by hand, since the execution of these loops can be slow.

Vectorization is often nontrivial especially for more complex expressions. Furthermore, many different numerical packages such as Theano [6], Tensorflow [3], or Numexpr [1] require the input to be in vectorized form, i.e., in the form of linear algebra matrix expressions.

In the remainder of this paper we demonstrate our approach for automatically vectorizing mathematical expressions, using logistic regression and support vector machines as examples. After briefly describing the transformation and validation in Section 2, we demonstrate the speedup in evaluating the vectorized expressions experimentally in Section 3, concluding in Section 4.

2 Building Blocks of the Vectorization Tool

Our vectorization tool has three components, namely a parser, a transformation and a validation module that we briefly describe in this section.

2.1 Parsing

For our implementation we have restricted the allowed operations and variables. For the allowed mathematical expressions we have defined a simple grammar, that is shown in Figure 1 using simple EBNF rules.

⟨formula⟩ ::= { 'forall' '[' ⟨index⟩ {',' ⟨index⟩ } ']' '} ⟨compassign⟩

⟨compassign⟩ ::= ⟨expr⟩ [(':=' | '==' | '>' | '>=' | '<' | '<=') ⟨expr⟩]

⟨expr⟩ ::= ⟨term⟩ {('+' | '-') ⟨term⟩ }

⟨term⟩ ::= ⟨factor⟩ {('*' | '/')}

⟨atom⟩ ::= number | ⟨function⟩ '(' ⟨expr⟩ ')' | ⟨variable⟩

⟨index⟩ ::= alpha

⟨variable⟩ ::= alpha+ ['[' ⟨index⟩ [',' ⟨index⟩']']

⟨function⟩ ::= 'sin' | 'cos' | 'exp' | 'log' | 'sign' | 'sqrt' | 'abs' | 'sum' '[' ⟨index⟩ ']'

Fig. 1. Grammar for mathematical expression.

Using this grammar we can represent the logistic regression problem as follows:

$$\mathrm{sum}[i](\log(1 + \exp(\mathrm{sum}[j](-y[i] * w[j] * X[i,j] + b)))). \quad (1)$$

The first constraint of the support vector machine can be represented as:

$$\mathrm{forall}[i] : \mathrm{sum}[j](y[i] * (w[j] * X[i,j] + b)) >= 1 - xi[i]. \quad (2)$$

After successfully parsing the expression into an expression tree, the transformation into vectorized form follows.

2.2 Transformation

The transformation into vectorized form works in a bottom-up fashion, by iterating through the nodes of the expression tree and replacing the nodes according to their operation. During the transformation the dimensions of the nodes are tracked and adjusted if needed. For example, while transforming the logistic regression expression tree, the algorithm reaches the node $+$ with already transformed subtrees 1 and $\exp(-y. * (X * w + \mathrm{vector}(1) * b))$. Since the right subtree is a vector and the left subtree is a scalar, the algorithm changes the left subtree to $\mathrm{vector}(1) * 1$. After this change the $+$ operator can be processed. Adapting the whole transformation step to tensors reduces the number of dimension checks and simplifies the algorithm. Finally, the resulting vectorization is determined by simple substitutions of tensor algebra expressions by linear matrix algebra.

After a successful transform the algorithm produces the following vectorization for logistic regression example:

$$\text{sum}(\log(\text{vector}(1) + \exp(-y. * (X * w + \text{vector}(1) * b)))), \tag{3}$$

and for the first constraint of the support vector machine:

$$y. * (X * w + \text{vector}(1) * b) >= \text{vector}(1) - xi, \tag{4}$$

where vector(1) is the vector of all ones and $.*$ is the element-wise multiplication.

2.3 Verification

For validating the vectorization against the given formula in index form we perform a numerical check as well as a dimension check. The numerical verification simply calculates values for the unvectorized formula and the vectorized form and checks for equality. The second test compares the dimension of the two formulas. Both must be equal to the number of unbound indices. Only if these tests are correct the vectorization is assumed to be valid.

3 Experiments

For demonstrating the efficiency of our vectorization we have implemented two different versions for each problem. The first version uses loops to implement Equations 1 and 2, whereas the second version directly implements the vectorizations (Equations 3 and 4) using Numpy [8]. From Equations 2 and 4 we only use the left term for the time measurements which is the most time consuming part. Figures 2 and 3 show the evaluation times for the two problems. The run-

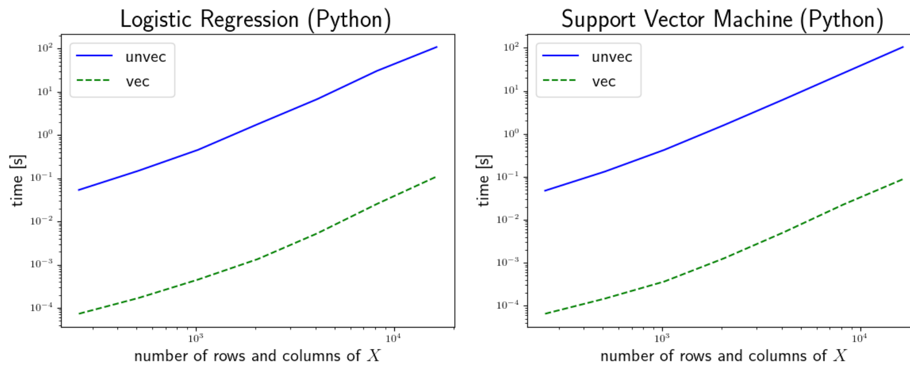

Fig. 2. The comparison of unvectorized formulas against its vectorized form in Python in a logarithmic plot.

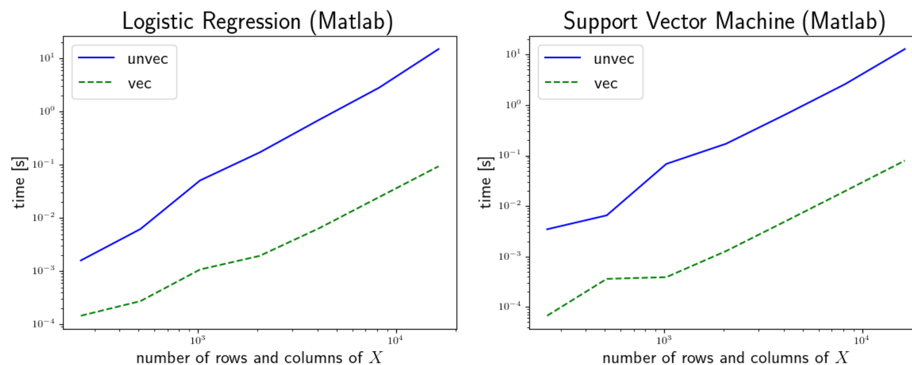

Fig. 3. The comparison of unvectorized formulas against its vectorized form in Matlab in a logarithmic plot.

ning times where obtained with Python 3.6 and Matlab R2016a. We generated random data sets $X \in \mathbb{R}^{m \times n}$ and $y \in \{-1, +1\}^m$ with $m = n$. During the experiments the best of three runs for each problem size is reported. We observe, that the vectorized expression for logistic regression can be evaluated three orders of magnitude faster than the unvectorized expression. Similar results hold for the support vector machine problem.

4 Conclusion

We have presented a tool for automatically transforming some mathematical expressions into their vectorized form. The experiments corroborate the practical benefits of vectorizing mathematical expressions. Especially for large data sets the vectorization mathematical expressions can result in a speedup of a few orders of magnitude. In the future we will include more operators and hence extend the set of mathematical expressions that can be transformed.

The presented tool is available at `http://www.autovec.org`.

References

1. Numexpr: Fast numerical expression evaluator for numpy, `https://github.com/pydata/numexpr`, accessed: 06.07.2017
2. Techniques to improve performance, `https://de.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html`, accessed: 06.07.2017
3. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V.,

Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
4. Cai, X., Langtangen, H.P., Moe, H.: On the performance of the python programming language for serial and parallel scientific computations. Sci. Program. 13(1), 31–56 (Jan 2005), http://dx.doi.org/10.1155/2005/619804
5. David Ascher, Paul F. Dubois, K.H.J.H.T.O.: Numerical python. Tech. rep., Lawrence Livermore National Laboratory, Livermore, CA 94566 (2001)
6. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints abs/1605.02688 (May 2016), http://arxiv.org/abs/1605.02688
7. Varoquaux, G.: Writing faster numerical code, http://www.scipy-lectures.org/advanced/optimizing/#writing-faster-numerical-code, accessed 06.07.2017
8. van der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science Engineering 13(2), 22–30 (March 2011)