TENLAB
A MATLAB Ripoff for Tensors

Y. Cem Sübakan, ys2939
Mehmet K. Turkcan, mkt2126
Dallas Randal Jones, drj2115

February 9, 2016

Introduction

MATLAB is a great language for manipulating arrays. However, its syntactic ease is more or less limited to 2D arrays; working with higher dimensional objects involves a lot of 'bsxfun'[1] (pun definitely intended) and in turn nonnegligible willpower and experience. In this project we intend to develop a streamlined tensor manipulation language for those who would prefer to avoid bsxfun's and dimension shifts.

Tensor research is an important field in data analysis. There exist a plethora of applications ranging from audio source separation to image analysis, from chemometrics to wireless communications [1, 2]. A good programming language with tensor support would be helpful in any application involving multi-dimensional arrays. For instance, when learning probabilistic models parameters often have multiple dimensions and we need to utilize generalized tensor product operations. Of course one can attempt to utilize a (potentially) huge number of for loops as well, but in this project we aim to develop a general purpose MATLAB like language that would easily handle operations with multi-way arrays.

Goals

- Create an extremely simple, easy to learn language strong enough for specialized tensor-utilizing applications.
- Minimize the amount of code one has to write to work with tensors, make the obvious approach the right one.
- Compile into C; this is excellent since people tend to work with ginormous tensors in real life.
- Build something we would like to use in the future.

[1] `http://www.mathworks.com/help/matlab/ref/bsxfun.html`

Mathematical Motivation

Let us start with the most basic tensors, vectors. Let $a, b \in \mathbb{R}^L$ (column vectors), and let $z \in \mathbb{R}^L$ be the output vector. The inner product of a and b is given as

$$z = \sum_{i=1}^{L} a_i b_i \tag{1}$$

Now, in a language like C, we need to run a boring for loop to do this summation. On the other hand, in MATLAB we can simply write

```
z = a'*b;
```

where $'$ transposes the vector a. Moving on to matrices, let $A \in \mathbb{R}^{L_1 \times K}$, and $B \in \mathbb{R}^{K \times L_2}$. Let $Z \in \mathbb{R}^{L_1 \times L_2}$ the output matrix. In index notation, Z is the following,

$$Z_{i,j} = \sum_{k=1}^{L} A_{ik} B_{kj}. \tag{2}$$

which requires three nested loops in C. In MATLAB, this is simply

```
 Z = A*B;
```

So far so good. But now let's move on to tensors. Let $A \in \mathbb{R}^{I_1, I_2, I_3}$, and $B \in \mathbb{R}^{J_1, J_2}$. Let's fancy that we 'need' to perform the following ugly operation (again Z is the output):

$$Z_{i_1,i_3,j_1} = \sum_{i_2=1}^{I_2} A_{i_1,i_2,i_3} B_{j_1,i_2}, \tag{3}$$

where we need $I_2 = J_2$, in order to be able to define the operation. Notice that the output ordering of the indices is arbitrary and it needs to be specified. In C, this requires illegible number of nested for loops, and in MATLAB people can get confused when reading statements like

```
Z = reshape(sum(bsxfun(@times,A, permute(reshape(B ,[1 1 J1 J2] ), ...
                                                [1 2 4 3])),3),[I1 I3 J1]);
```

in which the matrix B is first reshaped into a 4 dimensional object, after which the indices are permuted to get the summation along the desired dimensions. After an another sum over the 3rd dimension, the result is finally reshaped into the form originally desired. Note that we may need to use a permute operation if we want to have the output indices in a different order. The point is that this is *unnecessarily difficult*. Indeed, it would be nice if we could write this instead:

```
Z{1, 3, 2} = {1} A{2}.B{1};
```

where we specify the indices over which we would like to sum for each tensor on the right hand side and the index ordering for the output tensor on the left hand side. This automatically matches the second dimension of A with the first dimension of B

for summation. Then, the output is permuted according to index list specified on the left of the assignment operator =, to the right of which we specify a binary for each index specifying whether we would like to sum over the matched dimensions. If we don't specify a list next to the output, it will return a default index ordering (it won't permute the output indices). Similarly, if we want to code

$$Z_{i_1} = \sum_{i_3=1}^{I_3} \sum_{i_2=1}^{I_2} A_{i_1,i_2,i_3} B_{i_3,i_2}, \tag{4}$$

we can hope to type:

```
Z = {1, 1} A{2, 3}.B{2, 1};
```

If we do not wish to collapse (sum over) one of the matched dimensions, we simply put a zero to the corresponding entry in the list to the right of the assignment operator. For example,

```
Z = {0, 1} A{2, 3}.B{2, 1};
```

corresponds to the operation

$$Z_{i_1,i_2} = \sum_{i_3=1}^{I_3} A_{i_1,i_2,i_3} B_{i_3,i_2}, \tag{5}$$

Notice that this generalized tensor product operation 'includes' the outer product, therefore we will simply leave the lists empty for outer products. That is,

$$Z_{i_1,i_2,i_3,j_1,j_2} = A_{i_1,i_2,i_3} B_{j_1,j_2}. \tag{6}$$

will be produced by

```
Z  = {} A {} . B {}
```

A Trivial Example

Here's an example in pseudocode written in the "For style" for 'Variational Bayes for Infinite Mixtures of Hidden Markov Models':

Algorithm 1 Variational Bayes for an infinite mixture of HMMs

```
for $e = 1 : maxiter$ do
    for $k = 1 : K$ do
        for $j = 1 : J$ do
            $N_{k,j} = \sum_{n=1}^{N} \sum_{t=1}^{T_n} p^h_{k,n} p^r_{j,t,n}$    (update pseudo counts)
            $\widetilde{O}_{k,j} = \frac{\sigma^2 \mu_0 + \sigma_0^2 \sum_{n=1}^{N} \sum_{t=1}^{T_n} p^h_{k,n} p^r_{j,t,n} x_{t,n}}{N_{k,j}\sigma_0^2 + \sigma^2}$    (update the means)
            $\bar{\sigma}^2_{k,j} = \frac{\sigma^2 \sigma_0^2}{N_{k,j}\sigma_0^2 + \sigma^2}$    (update cluster variances)
        end for
    end for
    for $k = 1 : K$ do
        for $j_1 = 1 : J$ do
            for $j_2 = 1 : J$ do
                $\widetilde{A}_{k,j_1,j_2} = \beta - 1 + \sum_{n=1}^{N} p^h_{k,n} \sum_{t=2}^{T_n} p^r_{j,t,n} p^r_{j,t-1,n}$    (update transition matrix parameters)
            end for
        end for
    end for
    for $k = 1 : K$ do    (update stick breaking parameters)
        $\widetilde{\alpha}_k = \alpha + \sum_{n=1}^{N} p^h_{k,n}$
        $\widetilde{\gamma}_k = 1 + \sum_{n=1}^{N} \sum_{k'=k+1}^{K} p^h_{k',n}$
    end for
    ⋮
end for
```

Note that p^h is a two dimensional tensor (matrix) and p^r is a three dimensional tensor (h and r are superscripts, they are not indices!). Now, look how much simpler it would be to code this in our language:

```
for e = 1 : maxiter
   N = {1} Ph{2}.reshape(sum(Pr,2),[J N]){2}; %notice that we are using the
       add operation
   O = ({1}(Ph{2}.reshape( {1 0}Pr{2 3}.X{1 2} ),[J N]){2})*sigma0 + c1 ) *
       c2;
   Sig = 1./ (N*sig0 + sig); %just like MATLAB
   A =    {1}Ph{2} reshape( {1}Pr(:,2:end,:){2}Pr(:.1:end-1,:){2} , [ J N] )
       {2}  + b - 1;
   Al = a + sum(Ph,2);
   Gam = 1 + cumsum(sum(Ph,2)); %cumsum stands for cumulative sum, from
       MATLAB
end
```

Syntax and Thoughts

We will try to stick to MATLAB's syntax as much as possible since (a) a lot of people are familiar with it, and (b) it is awesome for the most part. That is, we are going to use the same syntax for comments, while loops, for loops, if statements, array access (that is, things like A(:,1), A(2,:)).

One other prospect is to generalize the tensor product into a general binary tensor operation. That is, If we use + instead og ., that would mean that we would add the matching dimensions instead of multiplying. For example, we can write

```
Z ={1} A{4} + B{8};
```

We can generalize this into any binary operation, as specified in the help page of bsxfun[2]. This would practically mean that we have a more intuitive version of bsxfun, though perhaps not as capable.

Another very interesting direction to take would be to allow varying lengths across dimensions. In the example we have given the data matrix (which is in fact a list) is definitely so. Each slice n of the data list X, contains a matrix with a differently sized second dimension. In real data applications this is indeed something to worry about and one needs to deal with cell arrays in MATLAB which are not easily amenable to algebraic operations.

References

[1] Kolda, T. G. and B. W. Bader, "Tensor Decompositions and Applications", *SIAM Review*, Vol. 51, pp. 455–500, 2009.

[2] Cichocki, A., D. P. Mandic, A. H. Phan, C. F. Caiafa, G. Zhou, Q. Zhao and L. D. Lathauwer, "Tensor Decompositions for Signal Processing Applications From Two-way to Multiway Component Analysis", *CoRR*, Vol. abs/1403.4462, 2014, `http://arxiv.org/abs/1403.4462`.

[2]`http://www.mathworks.com/help/matlab/ref/bsxfun.html`