MATLAB: Convolution of Matrix Valued Function

Asked 13 years, 8 months ago Modified 7 years, 7 months ago Viewed 4k times

4

I've written this code to perform the 1-d convolution of a 2-d matrix valued function (k is my time index, kend is on the order of 10e3). Is there a faster or cleaner way to do this, perhaps using built in functions?

```
for k=1:kend
  C(:,:,k)=zeros(3);
  for l=0:k-1
      C(:,:,k)=C(:,:,k)+A(:,:,k-l)*B(:,:,l+1);
  end
end
```

matlab matrix optimization convolution

Share
Improve this question
Follow

edited May 20, 2017 at 3:51
gnovice
126k ● 16 ● 258 ● 363

asked Apr 23, 2011 at 16:28
user719918
53 ● 1 ● 6

3 Answers

Sorted by: Highest score (default)

4

NEW SOLUTION:

This is a newer solution built on the older solution, which solved the previously given formula. The code in the question is actually a modification of that formula, in which the overlap between the two matrices in the third dimension is repeatedly shifted (it's akin to a convolution along the third dimension of the data). The previous solution I gave only computed the result for the *last* iteration of the code in the question (i.e. `k = kend`). So, here's a full solution that should be *much* more efficient than the code in the question for `kend` on the order of 1000:

```
kend = size(A,3);                         %# Get the value for kend
C = zeros(3,3,kend);                      %# Preallocate the output
Anew = reshape(flipdim(A,3),3,[]);        %# Reshape A into a 3-by-3*kend
matrix
Bnew = reshape(permute(B,[1 3 2]),[],3);  %# Reshape B into a 3*kend-by-3
matrix
for k = 1:kend
  C(:,:,k) = Anew(:,3*(kend-k)+1:end)*Bnew(1:3*k,:);  %# Index Anew and Bnew so
end                                                   %#   they overlap in
```

```
steps
                                                     %#    of three
```

Even when using just `kend = 100`, this solution came out to be about 30 times faster for me than the one in the question and about 4 times faster than a pure for-loop-based solution (which would involve **5 loops**!). Note that the discussion below of floating-point accuracy still applies, so it is normal and expected that you will see slight differences between the solutions on the order of the relative floating-point accuracy.

OLD SOLUTION:

Based on this formula you linked to in a comment:

$$C_{ij} := \sum_{r} \sum_{l=0}^{k} A_{ir}(k-l) B_{rj}(l)$$

it appears that you actually want to do something different than the code you provided in the question. Assuming `A` and `B` are 3-by-3-by-k matrices, the result `C` should be a 3-by-3 matrix and the formula from your link written out as a set of nested for loops would look like this:

```
%# Solution #1: for loops
k = size(A,3);
C = zeros(3);
for i = 1:3
  for j = 1:3
    for r = 1:3
      for l = 0:k-1
        C(i,j) = C(i,j) + A(i,r,k-l)*B(r,j,l+1);
      end
    end
  end
end
```

Now, it **is** possible to perform this operation without any for loops by reshaping and reorganizing `A` and `B` appropriately:

```
%# Solution #2: matrix multiply
Anew = reshape(flipdim(A,3),3,[]);         %# Create a 3-by-3*k matrix
Bnew = reshape(permute(B,[1 3 2]),[],3);  %# Create a 3*k-by-3 matrix
C = Anew*Bnew;                             %# Perform a single matrix multiply
```

You could even rework the code you have in your question to create a solution with a single loop that performs a matrix multiply of your 3-by-3 submatrices:

```
%# Solution #3: mixed (loop and matrix multiplication)
k = size(A,3);
C = zeros(3);
for l = 0:k-1
  C = C + A(:,:,k-l)*B(:,:,l+1);
end
```

So now the question: Which one of these approaches is faster/cleaner?

Well, "cleaner" is very subjective, and I honestly couldn't tell you which of the above pieces of code makes it any easier to understand what the operation is doing. All the loops and variables in the first solution make it a little hard to track what's going on, but it clearly mirrors the formula. The second solution breaks it all down into a simple matrix operation, but it's difficult to see how it relates to the original formula. The third solution seems like a middle-ground between the two.

So, let's make speed the tie-breaker. If I time the above solutions for a number of values of `k`, I get these results (in seconds needed to perform 10,000 iterations of the given solution, MATLAB R2010b):

```
 k   |  loop  | matrix multiply |  mixed
-----+--------+-----------------+--------
 5   | 0.0915 |      0.3242     | 0.1657
 10  | 0.1094 |      0.3093     | 0.2981
 20  | 0.1674 |      0.3301     | 0.5838
 50  | 0.3181 |      0.3737     | 1.3585
 100 | 0.5800 |      0.4131     | 2.7311   * The matrix multiply is now fastest
 200 | 1.2859 |      0.5538     | 5.9280
```

Well, it turns out that for smaller values of `k` (around 50 or less) the for-loop solution actually wins out, showing once again that for loops are not as "evil" as they used to be considered in older versions of MATLAB. Under certain circumstances, they can be more efficient than a clever vectorization. However, when the value of `k` is larger than around 100, the vectorized matrix-multiply solution starts to win out, scaling much more nicely with increasing `k` than the for-loop solution does. The mixed for-loop/matrix-multiply solution scales *atrociously* for reasons that I'm not exactly sure of.

So, if you expect `k` to be large, I'd go with the vectorized matrix-multiply solution. One thing to keep in mind is that the results you get from each solution (the matrix `C`) will differ ever so slightly (on the level of the floating-point precision) since the order of additions and multiplications performed for each solution are different, thus leading to a difference in accumulation of rounding errors. In short, the difference between the results for these solutions should be negligible, but you should be aware of it.

Share

Improve this answer

edited Apr 26, 2011 at 18:26

answered Apr 25, 2011 at 15:15

gnovice

126k ● 16 ● 258 ● 363

Follow

Thanks for all your work! The lhs of the formula I wrote probably should have been C_{ij}(k) for clarity. My code is correct, the result C should be 3-by-3-by-kend. – user719918 Apr 25, 2011 at 17:47

@user719918: OK, I *think* I see what you're doing. The answer I gave is basically the solution for the **last** iteration of your code (i.e. `k = kend`). I'll have a new solution for you soon. – gnovice Apr 25, 2011 at 20:07

2

Have you looked into Matlab's conv method?

I can't compare it against your provided code, because what you provided gives me a problem with trying to access the zeroth element of A. (When `k=1`, `k-1=0`.)

Share Improve this answer Follow

answered Apr 23, 2011 at 16:38

Ben Hocking
8,032 ● 5 ● 38 ● 53

I have. As far as I can tell conv does 1-d convolution of real valued data, and convn does n-d convolution of real valued data. I don't run into a zero index problem with that code. When `k=1`, we look from `l=0:0`, so we only access `A(:,:,k-0)`. Are you mistaking the "l" for a "1"? – user719918 Apr 23, 2011 at 16:53

2 I was mistaking the "l" for a "1", which is one reason why the letter should be avoided as a variable name. :) – Ben Hocking Apr 23, 2011 at 18:47

thats another reason why copy-paste is better than look-write :P – twerdster Apr 25, 2011 at 17:21

1

Have you considered using FFTs to convolve? A convolution operation is simply a point-wise multiplication in the frequency domain. You'll have to take some precaution with finite sequences, as you'll end up with circular convolution if you're not careful (but this is trivial to take care of).

Here's a simple example for a 1D case.

```
>> a=rand(4,1);
>> b=rand(3,1);
>> c=conv(a,b)

c =

    0.1167
    0.3133
    0.4024
    0.5023
    0.6454
    0.3511
```

The same using FFTs

```
>> A=fft(a,6);
>> B=fft(b,6);
>> C=real(ifft(A.*B))

C =

    0.1167
    0.3133
    0.4024
    0.5023
    0.6454
    0.3511
```

A convolution of an `M` point vector and an `N` point vector results in an `M+N-1` point vector. So, I've padded each of the vectors `a` and `b` with zeros before taking the FFT (this is automatically taken care of when I take the `4+3-1=6` point FFT of it).

EDIT

Although the equation that you showed is similar to a circular convolution, it's not exactly it. So you can ditch the FFT approach, and the built-in `conv*` functions. To answer your question, here's the same operation done without *explicit* loops:

```
dim1=3;dim2=dim1;
dim3=10;
a=rand(dim1,dim2,dim3);
b=rand(dim1,dim2,dim3);


mIndx=cellfun(@(x)(1:x),num2cell(1:dim3),'UniformOutput',false);
fun=@(x)sum(reshape(cell2mat(cellfun(@(y,z)a(:,:,y)*b(:,:,z),num2cell(x),num2cel
[dim1,dim2,max(x)]),3);
c=reshape(cell2mat(cellfun(@(x)fun(x),mIndx,'UniformOutput',false)),
[dim1,dim2,dim3]);
```

- `mIndx` here is a cell, where the `i`th cell contains a vector `1:i`. This is your `l` index (as others have noted, please don't use `l` as a variable name).
- The next line is an anonymous function that does the convolution operation, making use of the fact that the `k` index is just the `l` index flipped around. The operations are carried out on individual cells, and then assembled.
- The last line actually performs the operations on the matrices.

The answer is the same as that obtained with the loops. However, you'll find that the looped solution is actually an order of magnitude *faster* (I averaged 0.007s for my code and 0.0006s for the loop). This is because the loop is pretty straightforward, whereas with this sort of nested construction, there's plenty of function call overheads and repeated reshaping that slow it down.

MATLAB's loops have come a long way since the early days when loops were dreaded. Certainly, vectorized operations are blazing fast; but not everything can be vectorized, and sometimes, loops are more efficient than such convoluted anonymous functions. I could probably shave off a few more tenths here and there by optimizing my construction (or maybe taking a different approach), but I'm not going to do that.

Remember that good code should be readable, as well as efficient and minor optimization at the cost of readability serves no one. Although I wrote the code above, I certainly will not be able to decipher what it does if I revisited it a month later. Your looped code was clear, readable *and* fast and I would suggest that you stick with it.

Share
Improve this answer
Follow

edited Apr 24, 2011 at 6:24

answered Apr 23, 2011 at 19:42
abcd
42.2k ● 7 ● 84 ● 99

Thanks for the response. I knew this worked for 1-d functions, but I hadn't considered it here. Unfortunately, it appears fft() has the same problem as conv(); it only takes vectors as an inputs. – user719918 Apr 23, 2011 at 20:07

fft takes arrays as inputs too. you can specify along which dimension the operation should be performed. – abcd Apr 23, 2011 at 20:33

You're right, sorry I didn't read the documentation as closely as I should have. I have a feeling this is the right track, but I still can't get it work. codetidy.com/600 – user719918 Apr 24, 2011 at 0:26

I don't think your implementation of convolution in the question is entirely correct. You're not zero-padding the matrices, so you'll only get the first `N` outputs from the convolution when there are actually `2N-1` (assuming you start with two `N` point vectors). You'll see what I mean if you simplify your code to work with a 1D vector and look at the output and compare with that from `conv`. I think that is the reason you're seeing a discrepancy with the FFT solution. – abcd Apr 24, 2011 at 2:07

(continued...) Also, I think that it should be a Hadamard product (element-wise multiplication) and not a matrix multiplication in the `C(:,:,k)=...` line, but I could be wrong. – abcd Apr 24, 2011 at 2:08