Tutorial

2D CONVOLUTION AS MATRIX MULTIPLICATION USING TOEPLITZ MATRICES

By: Ali Salehi
AliSaaalehi@gmail.com
August 20, 2018

Contents

1 What is the purpose?

Instead of using for-loops to perform 2D convolution on images (or any other 2D matrices) we can convert the filter to a Toeplitz matrix and image to a vector and do the convolution just by one matrix multiplication (and of course some post-processing on the result of this multiplication to get the final result)

2 Why do we do that?

There are many efficient matrix multiplication algorithms, so using them we can have an efficient implementation of convolution operation.

3 What is in this document?

Mathematical and algorithmic explanation of this process. I will put a naive Python implementation of this algorithm to make it more clear.
Let's start with some definition and basic operation:

4 What is a Toeplitz matrix?

Toeplitz matrix is a matrix in which each values along the main diagonal and sub diagonals are constant. Matrix G is an example:

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & 1 \\ 5 & 2 & -1 & 0 & & & & \vdots \\ -8 & 5 & 2 & -1 & \ddots & & & \vdots \\ \vdots & -8 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & & & \ddots & 5 & 2 & -1 & 0 \\ \vdots & & & & -8 & 5 & 2 & -1 \\ 1 & \cdots & \cdots & \cdots & \cdots & -8 & 5 & 2 \end{pmatrix} \tag{1}$$

In a $N \times N$ matrix, its elements are determined by a $(2N-1)$ length sequence

$$\{t_n | -(N-1) \leq n \leq (N-1)\}$$

. So given a sequence t_n we can create a Toeplitz matrix by following these steps:

- put the sequence in the first column of the matrix.
- shift it and put it in the next column. When shifting, the last element disappears and a new element of the sequence appears. If there is no such an element, put zero in that location.

specifically:

$$T(m,n) = t_{m-n}$$

$$\begin{pmatrix} t_0 & t_{-1} & t_{-2} & \cdots & \cdots & \cdots & \cdots & t_{-(N-1)} \\ t_1 & t_0 & t_{-1} & t_{-2} & & & & \vdots \\ t_2 & t_1 & t_0 & t_{-1} & \ddots & & & \vdots \\ \vdots & t_2 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & t_{-2} & \vdots \\ \vdots & & & \ddots & t_1 & t_0 & t_{-1} & t_{-2} \\ \vdots & & & & t_2 & t_1 & t_0 & t_{-1} \\ t_{(N-1)} & \cdots & \cdots & \cdots & \cdots & t_2 & t_1 & t_0 \end{pmatrix} \tag{2}$$

Be aware that when we are working with sequences that are defined just for $n \geq 0$ values for t_n when $n \leq 0$ should be considered as 0. For example 4×4 Toeplitz matrix for the sequence $f[n] = [1, 2, 3]$ will be:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 0 & 3 & 2 & 1 \end{pmatrix} \tag{3}$$

5 One more definition: Doubly Blocked Toeplitz mtrix

In the matrix A, all A_{ij} are matrices. If the structure of A, with respects to its sub-matrices is Toeplitz, then matrix A is called **Block-Toeplitz**.

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ A_{M1} & A_{M2} & \cdots & A_{MN} \end{pmatrix} \tag{4}$$

If each individual Aij also is a Toeplitz matrix then A is called **Doubly Blocked Toeplitz**

6 Is this Convolution or Cross Correlation?

Most of the time, the word *convolution* in the deep learning literature is used instead of cross-correlation, but here I am explaining the process for convolution as is known in the signal processing community. Simply, for convolution we need to flip the filter (kernel) in both vertical and horizontal directions, but for cross-correlation we don't.

The method explained here performs the convolution (not correlation). Because of the way it is implemented here, there is no need to flip the filter, but if you are doing an example by hand and want to compare the results with the implemented method, remember to consider the flipping step in your calculation.

7 Step by Step

Let's explain the algorithm step by step using an example. Codes are written in python and the numpy library is used all over the code.

Note: Remember that convolution is a commutative operation, so it does not change the output if we switch the inputs for this operation. For simplicity, I will be calling one of the inputs **input or I** and the other **filter or F**

7.1 Input and Filter

Input matrix that the filter will be convolved with it is:

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{5}$$

And let the filter be:

$$F = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} \tag{6}$$

Python code to define these two matrices is:

```python
import numpy as np

# input signal
I = np.array([[1, 2, 3], [4, 5, 6]])

# filter
F = np.array([[10, 20], [30, 40]])
```

7.2 Calculate the final output size

If the input signal is $m_1 \times n_1$ and filter is $m_2 \times n_2$ the size of the convolution will be

$$(m_1 + m_2 - 1) \times (n_1 + n_2 - 1)$$

This is the size of full discrete linear convolution. One might just use some part of the output based on the application. For example in deep learning literature you can use "valid" or "same" as your padding mode. In these case just parts of the full output is used.

Proper zero padding should be done to get the correct output. Zero padding is the next step.

```
# number columns and rows of the input
I_row_num, I_col_num = I.shape

# number of columns and rows of the filter
F_row_num, F_col_num = F.shape


#  calculate the output dimensions
output_row_num = I_row_num + F_row_num - 1
output_col_num = I_col_num + F_col_num - 1
```

7.3 Zero-pad the filter matrix

The next step is to zero pad the filter to make it the same size as the output. Zeros should be added to the top and right sides of the filter.

$$ZeroPadded\ F = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 10 & 20 & 0 & 0 \\ 30 & 40 & 0 & 0 \end{bmatrix} \tag{7}$$

```
# zero pad the filter
F_zero_padded = np.pad(F, ((output_row_num - F_row_num, 0),
                           (0, output_col_num - F_col_num)),
                        'constant', constant_values=0)

'''
F_zero_padded:
[[ 0   0   0   0]
 [10  20   0   0]
 [30  40   0   0]]
'''
```

7.4 Toeplitz matrix for each row of the zero-padded filter

For each row of the zero-padded filter (`F_zero_padded`) create a Toeplitz matrix and store them in a list. Matrix created using the last row goes to the first cell of this list.

$$ZeroPadded\ F = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 10 & 20 & 0 & 0 \\ 30 & 40 & 0 & 0 \end{bmatrix} \tag{8}$$

$$F_0 = \begin{bmatrix} 30 & 0 & 0 \\ 40 & 30 & 0 \\ 0 & 40 & 30 \\ 0 & 0 & 40 \end{bmatrix} F_1 = \begin{bmatrix} 10 & 0 & 0 \\ 20 & 10 & 0 \\ 0 & 20 & 10 \\ 0 & 0 & 20 \end{bmatrix} F_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{9}$$

Why these matrices have three columns? Why not two or 5? What is the rule here?
The important point is that the number of columns of these generated Toeplitz matrices should be same as the number of columns of the input (I) matrix.
In the code, I am using *toeplitz()* function from *scipy.linalg library*. One row of the F is passed to this function and the function puts it as the first column of the its output matrix. Then as it is explained before, this vector should be shifted down and be putted in the second column. For this function, in addition to the first column, we need to define the first row, otherwise, the output of the function would be different than what we expect here. The first element of this first row is same as the first element of the first column, and the rest of the elements should be set to zero.

```
        toeplitz_list = []
        # iterate from last row to the first row
        for i in range(F_zero_padded.shape[0]-1, -1, -1):
         c = F_zero_padded[i, :] # copy i'th row of the F to c
         r = np.r_[c[0], np.zeros(I_col_num-1)]

         # toeplitz function is in scipy.linalg library
         toeplitz_m = toeplitz(c,r)
         toeplitz_list.append(toeplitz_m)
         print('F␣'+ str(i)+'\n', toeplitz_m)
```

7.5 Create doubly blocked toeplitz matrix

Now all these small toeplitz matrices should be arranged in a big doubly blocked toepltiz matrix 5.
In this example F_0, F_1, F_2 are corresponding toeplitz matrices for each row of the filter. They should be filled in the doubly blocked toeplitz matrix in this way:

$$doubly\ blocked = \begin{bmatrix} F_0 & 0 \\ F_1 & F_0 \\ F_2 & F_1 \end{bmatrix} \tag{10}$$

Number of columns in this symbolic matrix should be same as the number of rows in the input signal I.
The following code stores the indexes of F_0, F_1, F_2 in this format. This will be used to fill out the doubly blocked toepltiz matrix later.

```
c = range(1, F_zero_padded.shape[0]+1)
r = np.r_[c[0], np.zeros(I_row_num-1, dtype=int)]

doubly_indices = toeplitz(c, r)
print('doubly␣indices␣\n', doubly_indices)

'''
doubly indices
[[1 0]
[2 1]
[3 2]]
'''
```

Now let's fill in the doubly blocked toepltiz matrix. Following code does this part:

```
# shape of one of those small toeplitz matrices
h = toeplitz_shape[0]*doubly_indices.shape[0]
w = toeplitz_shape[1]*doubly_indices.shape[1]
doubly_blocked_shape = [h, w]
doubly_blocked = np.zeros(doubly_blocked_shape)

# tile the toeplitz matrix
b_h, b_w = toeplitz_shape # hight & withs of each block
for i in range(doubly_indices.shape[0]):
 for j in range(doubly_indices.shape[1]):
  start_i = i * b_h
  start_j = j * b_w
  end_i = start_i + b_h
  end_j = start_j + b_w
  doubly_blocked[start_i: end_i, start_j:end_j] =
                   toeplitz_list[doubly_indices[i,j]-1]

print(doubly_blocked)
'''
```

```
    [30.   0.   0.   0.   0.   0.]
    [40.  30.   0.   0.   0.   0.]
    [ 0.  40.  30.   0.   0.   0.]
    [ 0.   0.  40.   0.   0.   0.]
    [10.   0.   0.  30.   0.   0.]
    [20.  10.   0.  40.  30.   0.]
    [ 0.  20.  10.   0.  40.  30.]
    [ 0.   0.  20.   0.   0.  40.]
    [ 0.   0.   0.  10.   0.   0.]
    [ 0.   0.   0.  20.  10.   0.]
    [ 0.   0.   0.   0.  20.  10.]
    [ 0.   0.   0.   0.   0.  20.]
    '''
```

For this example the result will be the following matrix. I've colored parts of the matrix that is related to each of the small toeplitz matrices.

$$doubly\ blocked = \begin{bmatrix} 30 & 0 & 0 & 0 & 0 & 0 \\ 40 & 30 & 0 & 0 & 0 & 0 \\ 0 & 40 & 30 & 0 & 0 & 0 \\ 0 & 0 & 40 & 0 & 0 & 0 \\ 10 & 0 & 0 & 30 & 0 & 0 \\ 20 & 10 & 0 & 40 & 30 & 0 \\ 0 & 20 & 10 & 0 & 40 & 30 \\ 0 & 0 & 20 & 0 & 0 & 40 \\ 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 20 & 10 & 0 \\ 0 & 0 & 0 & 0 & 20 & 10 \\ 0 & 0 & 0 & 0 & 0 & 20 \end{bmatrix} \tag{11}$$

7.6 Convert the input matrix to a vector

Now that the filter has converted to a doubly blocked Toeplitz matrix, we just need to convert the input signal to a vector and multiply them.
All the rows of the input should be transposed to a column vector and stacked on top of each other. The last row goes first!

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow Vectoriaed\ I = \begin{bmatrix} 4 \\ 5 \\ 6 \\ 1 \\ 2 \\ 3 \end{bmatrix} \tag{12}$$

The following function does the vectorization. I am sure that there are much simpler ways to do so, but for the purpose of explanation, this function is implemented in this way.

```
def matrix_to_vector(input):
 input_h, input_w = input.shape
 output_vector = np.zeros(input_h*input_w,
                          dtype=input.dtype)
 # flip the input matrix up down
 input = np.flipud(input)
 for i,row in enumerate(input):
  st = i*input_w
  nd = st + input_w
  output_vector[st:nd] = row

return output_vector
```

7.7 Multiply doubly blocked toeplitz matrix with vectorized input signal

Do the matrix multiplication between these two matrices. In this example, the doubly blocked Toeplitz matrix is 12×6 and the vectorized input is 6×1 so the result of the multiplication is 12×1.

```
# get result of the convolution by matrix mupltiplication
result_vector = np.matmul(doubly_blocked, vectorized_I)
print('result_vector:␣', result_vector)

'''
result_vector: [120 310 380 240 70 230 330 240 10 40 70 60]
'''
```

7.8 Last step: reshape the result to a matrix form

From section 7.2 we know that the output of the convolution should be $(m_1 + m_2 - 1) \times (n_1 + n_2 - 1)$. First $(n_1 + n_2 - 1)$ elements in the output vector form the last row of the final output and the second (n_1+n_2-1) elements go to the second-to-last row of the output matrix. Repeat this process to form the final output matrix.

In this example $n_1 = 3$ and $n_2 = 2$, so every 4 element goes to one row of the output matrix.

$$result\ vector = \begin{bmatrix} 120 \\ 310 \\ 380 \\ 240 \\ 70 \\ 230 \\ 330 \\ 240 \\ 10 \\ 40 \\ 70 \\ 60 \end{bmatrix} \Rightarrow output = \begin{bmatrix} 10 & 40 & 70 & 60 \\ 70 & 230 & 330 & 240 \\ 120 & 310 & 380 & 240 \end{bmatrix} \tag{13}$$

Following function performs this step:

```
def vector_to_matrix(input, output_shape):
 output_h, output_w = output_shape
 output = np.zeros(output_shape, dtype=input.dtype)
 for i in range(output_h):
  st = i*output_w
  nd = st + output_w
  output[i, :] = input[st:nd]

 # flip the output matrix up-down to get correct result
 output=np.flipud(output)
 return output
```

8 Compare the result with other convolution methods

We can compare the output of this method with *convolve2d()* function from the *scipy* library.

```
from scipy import signal

result = signal.convolve2d(I, F, "full")
print('result: \n', result)
```

```
        '''
        result:
        [[ 10   40   70   60]
        [ 70  230  330  240]
        [120  310  380  240]]
        '''
```

As you can see the result on the same I and F matrices is same as the result of the implemented method. The parameter "full" is passed to the *signal.convolve2d()* function to get the full convolution results.

9 To Do

- Add notebook to the project
- Rewrite an efficient code
- Extend it to handle multi-channel input and filters
- Make it work with parameters padding='same' or 'valid'

10 References

The steps explained here are based on Christophoros Nikou's slides on Filtering in the Frequency Domain (Circulant Matrices and Convolution)

This post on dsp.stackexchange also helped in understanding this algorithm.