Implementation

Chih-Jen Lin
National Taiwan University

Last updated: May 25, 2020

Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$
5. Discussion

Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$
5. Discussion

Introduction I

- After checking formulations for gradient calculation we would like to get into implementation details
- Take the following operation as an example

$$\frac{\partial \xi_i}{\partial W^m} = \frac{\partial \xi_i}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T$$

- It's a matrix-matrix product
- We all know that a three-level for loop does the job
- Does that mean we can then write an efficient implementation?
- The answer is no

Introduction II

- To explain this, we check some details of matrix-matrix products
- We also introduce optimized BLAS (Basic Linear Algebra Subprograms)

Matrix Multiplication I

- We know that

$$C = AB$$

is a mathematics operation with

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Optimized BLAS: an Example by Using Block Algorithms I

- Let's test the matrix multiplication
- A C program:

```
#define n 2000
double a[n][n], b[n][n], c[n][n];

int main()
{
   int i, j, k;
   for (i=0;i<n;i++)
      for (j=0;j<n;j++) {
```

Optimized BLAS: an Example by Using Block Algorithms II

```
                a[i][j]=1; b[i][j]=1;
            }

        for (i=0;i<n;i++)
            for (j=0;j<n;j++) {
                c[i][j]=0;
                for (k=0;k<n;k++)
                    c[i][j] += a[i][k]*b[k][j];
            }
    }
```

Optimized BLAS: an Example by Using Block Algorithms III

- The result

```
cjlin@linux6:~$ gcc -O3 mat.c
cjlin@linux6:~$ time ./a.out
real 0m59.251s
user 0m58.994s
sys 0m0.096s
```

- Let's try another way:

Optimized BLAS: an Example by Using Block Algorithms IV

```
#define n 2000
double a[n][n], b[n][n], c[n][n];

int main()
{
   int i, j, k;
   for (i=0;i<n;i++)
      for (j=0;j<n;j++) {
         a[i][j]=1; b[i][j]=1;
         c[i][j]=0;
```

Optimized BLAS: an Example by Using Block Algorithms V

```
        }

      for (j=0;j<n;j++) {
          for (k=0;k<n;k++)
              for (i=0;i<n;i++)
                  c[i][j] += a[i][k]*b[k][j];
      }
  }
```

- The result

Optimized BLAS: an Example by Using Block Algorithms VI

```
cjlin@linux6:~$ gcc -O3 mat1.c
cjlin@linux6:~$ time ./a.out
real 2m13.199s
user 2m12.810s
sys 0m0.060s
```

- We see that first approach is faster. Why?
- For each of

```
    c[i][j] a[i][k] b[k][j];
```

we do column-access

Optimized BLAS: an Example by Using Block Algorithms VII

- C is row-oriented rather than column-oriented
- Now we sense that memory access can be an issue
- Let's try a Matlab program on the same computer

```
n = 2000;
A = randn(n,n); B = randn(n,n);
t = cputime; C = A*B; t = cputime -t
```

- To remove the effect of multi-threading, use

```
matlab -singleCompThread
```

- Timing is an issue

Optimized BLAS: an Example by Using Block Algorithms VIII

Elapsed time versus CPU time

```
cjlin@linux6:~$ matlab -singleCompThread
>> n = 2000;
>> A = randn(n,n); B = randn(n,n);
>> tic; C = A*B; toc
Elapsed time is 1.139780 seconds.
>> t = cputime; C = A*B; t = cputime -t
t =
    1.1200
```

- If using multiple cores,

Optimized BLAS: an Example by Using Block Algorithms IX

```
cjlin@linux6:~$ matlab
>> tic; C = A*B; toc
Elapsed time is 0.227179 seconds.
>> t = cputime; C = A*B; t = cputime -t
t =
    1.6800
```

- Matlab is much faster than a code written by ourselves. Why ?
- Optimized BLAS: data locality is exploited
- Use the highest level of memory as possible

Optimized BLAS: an Example by Using Block Algorithms X

- Block algorithms: transferring sub-matrices between different levels of storage

  They localize operations to achieve good performance

Memory Hierarchy I

CPU

$\downarrow$

Registers

$\downarrow$

Cache

$\downarrow$

Main Memory

$\downarrow$

Secondary storage (Disk)

Memory Hierarchy II

- $\uparrow$: increasing in speed
- $\downarrow$: increasing in capacity
- When I studied computer architecture, I didn't quite understand that this setting is so useful
- But from optimized BLAS I realize that it is extremely powerful

Memory Management I

- Page fault: operand not available in main memory
  transported from secondary memory
  (usually) overwrites page least recently used
- I/O increases the total time
- An example: $C = AB + C$, $n = 1,024$
- Assumption: a page 65,536 doubles = 64 columns
- 16 pages for each matrix
  48 pages for three matrices

Memory Management II

- Assumption: available memory 16 pages, matrices access: column oriented

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

  column oriented: 1 3 2 4

  row oriented: 1 2 3 4

- access each row of A: 16 page faults, $1024/64 = 16$
- Assumption: each time a continuous segment of data into one page
- Approach 1: inner product

Memory Management III

```
for i =1:n
  for j=1:n
    for k=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

We use a matlab-like syntax here

- At each (i,j): each row a(i, 1:n) causes 16 page faults

Memory Management IV

Total: $1024^2 \times 16$ page faults

- at least 16 million page faults
- Approach 2:

```
for j =1:n
  for k=1:n
    for i=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

Memory Management V

- For each j, access all columns of A
  A needs 16 pages, but B and C take spaces as well
  So A must be read for every j
- For each j, 16 page faults for A
  1024×16 page faults
  C, B : 16 page faults
- Approach 3: block algorithms (nb $= 256$)

Memory Management VI

```
for j =1:nb:n
  for k=1:nb:n
    for jj=j:j+nb-1
      for kk=k:k+nb-1
        c(:,jj) = a(:,kk)*b(kk,jj)+c(:,jj);
      end
    end
  end
end
```

In MATLAB, `1:256:1025` means 1, 257, 513, 769

Memory Management VII

- Note that we calculate

$$
\begin{bmatrix} A_{11} & \cdots & A_{14} \\ & \vdots & \\ A_{41} & \cdots & A_{44} \end{bmatrix}
\begin{bmatrix} B_{11} & \cdots & B_{14} \\ & \vdots & \\ B_{41} & \cdots & B_{44} \end{bmatrix}
$$

$$
= \begin{bmatrix} A_{11}B_{11} + \cdots + A_{14}B_{41} & \cdots \\ \vdots & \ddots \end{bmatrix}
$$

Memory Management VIII

- Each block: 256×256

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + \cdots + A_{14}B_{41} \\
C_{21} &= A_{21}B_{11} + \cdots + A_{24}B_{41} \\
C_{31} &= A_{31}B_{11} + \cdots + A_{34}B_{41} \\
C_{41} &= A_{41}B_{11} + \cdots + A_{44}B_{41}
\end{aligned}
$$

- For each (j, k), $B_{k,j}$ is used to add $A_{:,k}B_{k,j}$ to $C_{:,j}$

Memory Management IX

- Example: when $j = 1, k = 1$

$$C_{11} \leftarrow C_{11} + A_{11}B_{11}$$
$$\vdots$$
$$C_{41} \leftarrow C_{41} + A_{41}B_{11}$$

- Use Approach 2 for $A_{:,1}B_{11}$
- $A_{:,1}$: 256 columns, $1024 \times 256/65536 = 4$ pages. $A_{:,1}, \ldots, A_{:,4}$: $4 \times 4 = 16$ page faults in calculating $C_{:,1}$
- For A: 16×4 page faults
- B: 16 page faults, C: 16 page faults

Optimized BLAS Implementations

- OpenBLAS

  http://www.openblas.net/

  It is an optimized BLAS library based on GotoBLAS2 (see the story in the next slide)
- Intel MKL (Math Kernel Library)

  https://software.intel.com/en-us/mkl

Some Past Stories about Optimized BLAS

- BLAS by Kazushige Goto
  https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2
- See the NY Times article: "Writing the fastest code, by hand, for fun: a human computer keeps speeding up chips"
  http://www.nytimes.com/2005/11/28/technology/28super.html?pagewanted=all

Discussion I

- This discussion roughly explains why GPU is used for deep learning
- Somehow we can do fast matrix-matrix operations on GPU
- Note that we did not touch multi-core implementations, though parallelization is possible
- Anyway, the conclusion is that for some operations, using code written by experts is more efficient than our own implementation
- How about other operations besides matrix-matrix products?

Discussion II

- If they can also be done by calling others' efficient implementation, then a simple and efficient CNN implementation can be done
- The MATLAB implementation in simpleNN is a good experimental environment for us to study this
- We will explain details and use it in our subsequent projects

Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$
5. Discussion

Storage I

- In the earlier discussion, we check each individual data.
- However, for practical implementations, all (or some) instances must be considered together for memory and computational efficiency.
- Recall we do mini-batch stochastic gradient
- In our discussion we use l to denote the number of data instances in calculating the gradient (or the sub-gradient)

Storage II

- In our implementation, we store $Z^{m,i}, \ \forall i = 1, \ldots, l$ as the following matrix.

$$\begin{bmatrix} Z^{m,1} & Z^{m,2} & \ldots & Z^{m,l} \end{bmatrix} \in R^{d^m \times a^m b^m l}. \qquad (1)$$

- Similarly, we store

$$\frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T}, \ \forall i$$

as

$$\begin{bmatrix} \frac{\partial \xi_1}{\partial S^{m,1}} & \ldots & \frac{\partial \xi_l}{\partial S^{m,l}} \end{bmatrix} \in R^{d^{m+1} \times a^m_{\text{conv}} b^m_{\text{conv}} l}. \qquad (2)$$

Storage III

- We will explain our decision.
- Note that (1)-(2) are only the main setting to store these matrices because for some operations they may need to be re-shaped.
- For an easy description we may follow past discussion to let

$$Z^{\text{in},i} \text{ and } Z^{\text{out},i}$$

be the input and output images of a layer, respectively.

Operations of a Convolutional Layer I

- Recall that we conduct the following operations

$$
\begin{aligned}
&\frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T} \\
&= \left(\frac{\partial \xi_i}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \mathrm{vec}(I[Z^{m+1,i}])^T \right) P_{\mathrm{pool}}^{m,i}
\end{aligned} \tag{3}
$$

$$
\frac{\partial \xi_i}{\partial W^m} = \frac{\partial \xi_i}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T \tag{4}
$$

$$
\frac{\partial \xi_i}{\partial \mathrm{vec}(Z^{m,i})^T} = \mathrm{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)^T P_\phi^m P_{\mathrm{pad}}^m, \tag{5}
$$

Operations of a Convolutional Layer II

- Based on the way discussed to store variables, we will discuss two operations in detail
  - Generation of $\phi(\text{pad}(Z^{m,i}))$
  - vector $\times P_\phi^m$

Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$
5. Discussion

`im2col` in Existing Packages I

- Due to the wide use of CNN, a subroutine for $\phi(\text{pad}(Z^{m,i}))$ has been available in some packages
- For example, MATLAB has a built-in function `im2col` that can generate $\phi(\text{pad}(Z^{m,i}))$ for

$$s = 1 \text{ and } s = h \text{ (width of filter)}$$

- But this function cannot handle general s
- Can we do a reasonably efficient implementation by ourselves?

im2col in Existing Packages II

- For an easy description we consider

$$\text{pad}(Z^{m,i}) = Z^{\text{in},i} \rightarrow Z^{\text{out},i} = \phi(Z^{\text{in},i}).$$

Linear Indices and an Example I

- Consider the following column-oriented linear indices (i.e., counting elements in a column-oriented way) of $Z^{\text{in},i}$:

$$\begin{bmatrix} 1 & d^{\text{in}}+1 & \dots & (b^{\text{in}}a^{\text{in}}-1)d^{\text{in}}+1 \\ 2 & d^{\text{in}}+2 & \dots & (b^{\text{in}}a^{\text{in}}-1)d^{\text{in}}+2 \\ \vdots & \vdots & \ddots & \vdots \\ d^{\text{in}} & 2d^{\text{in}} & \dots & (b^{\text{in}}a^{\text{in}})d^{\text{in}} \end{bmatrix} \in R^{d^{\text{in}} \times a^{\text{in}} b^{\text{in}}}. \tag{6}$$

Linear Indices and an Example II

- Every element in

$$\phi(Z^{\text{in},i}) \in R^{hhd^{\text{in}} \times a^{\text{out}} b^{\text{out}}},$$

  is extracted from $Z^{\text{in},i}$
- The task is to find the mapping between each element in $\phi(Z^{\text{in},i})$ and a linear index of $Z^{\text{in},i}$.
- Consider an example with

$$a^{\text{in}} = 3, \ b^{\text{in}} = 2, \ d^{\text{in}} = 1.$$

  Because $d^{\text{in}} = 1$, we omit the channel subscript.

Linear Indices and an Example III

- In addition, we omit the instance index i, so the image is

$$\begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \\ z_{31} & z_{32} \end{bmatrix}.$$

- If

$$h = 2, s = 1,$$

two sub-images are

$$\begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix} \text{ and } \begin{bmatrix} z_{21} & z_{22} \\ z_{31} & z_{32} \end{bmatrix}$$

Linear Indices and an Example IV

- By our earlier way of representing images,

$$Z^{\text{in},i} = \begin{bmatrix} z^i_{1,1,1} & z^i_{2,1,1} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},1} \\ \vdots & \vdots & \ddots & \vdots \\ z^i_{1,1,d^{\text{in}}} & z^i_{2,1,d^{\text{in}}} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},d^{\text{in}}} \end{bmatrix}$$

the one we have is

$$Z^{\text{in}} = \begin{bmatrix} z_{11} & z_{21} & z_{31} & z_{12} & z_{22} & z_{32} \end{bmatrix}$$

- The linear indices from (6) are

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}.$$

Linear Indices and an Example V

- Recall that

$$\phi(Z^{\text{in},i}) =$$

$$\begin{bmatrix}
z^i_{1,1,1} & z^i_{1+s,1,1} & & z^i_{1+(a^{\text{out}}-1)s,1+(b^{\text{out}}-1)s,1} \\
z^i_{2,1,1} & z^i_{2+s,1,1} & & z^i_{2+(a^{\text{out}}-1)s,1+(b^{\text{out}}-1)s,1} \\
\vdots & \vdots & \ldots & \vdots \\
z^i_{h,h,1} & z^i_{h+s,h,1} & & z^i_{h+(a^{\text{out}}-1)s,h+(b^{\text{out}}-1)s,1} \\
\vdots & \vdots & & \vdots \\
z^i_{h,h,d^{\text{in}}} & z^i_{h+s,h,d^{\text{in}}} & & z^i_{h+(a^{\text{out}}-1)s,h+(b^{\text{out}}-1)s,d^{\text{in}}}
\end{bmatrix}$$

Linear Indices and an Example VI

- Therefore,

$$\phi(Z^{\text{in}}) = \begin{bmatrix} z_{11} & z_{21} \\ z_{21} & z_{31} \\ z_{12} & z_{22} \\ z_{22} & z_{32} \end{bmatrix}.$$

- Linear indices of Z^m to get elements of $\phi(Z^m)$:

$$\begin{array}{ll} Z^{m,i} & \begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6\end{bmatrix}^T \\ \phi(Z^{m,i}) & \begin{bmatrix}1 & 2 & 4 & 5 & 2 & 3 & 5 & 6\end{bmatrix}^T. \end{array}$$

- Example of using Matlab/Octave

Linear Indices and an Example VII

```
octave:8> reshape((1:6)', 3, 2)
ans =

   1   4
   2   5
   3   6

octave:9>
octave:9> im2col(reshape((1:6)', 3, 2),
                 [2,2], "sliding")
ans =
```

Linear Indices and an Example VIII

1	2
2	3
4	5
5	6

- To handle all instances together, we store

$$Z^{\text{in},1}, \ldots, Z^{\text{in},l}$$

as

$$\begin{bmatrix} \text{vec}(Z^{\text{in},1}) & \ldots & \text{vec}(Z^{\text{in},l}) \end{bmatrix}$$

Linear Indices and an Example IX

- Denote it as a MATLAB matrix

$$\texttt{Z}$$

- Then

$$\begin{bmatrix} \text{vec}(\phi(Z^{m,1})) & \dots & \text{vec}(\phi(Z^{m,l})) \end{bmatrix}$$

is simply

$$\texttt{Z(P,:)}$$

in MATLAB, where we store the mapping by

$$\texttt{P} = [\, 1 \;\; 2 \;\; 4 \;\; 5 \;\; 2 \;\; 3 \;\; 5 \;\; 6 \,]^T$$

Linear Indices and an Example X

- All instances handled in one line
- Moreover, we hope Matlab's implementation on this operation is efficient
- But how to obtain `P`?
- Note that

$$\begin{bmatrix} 1 & 2 & 4 & 5 & 2 & 3 & 5 & 6 \end{bmatrix}^T.$$

also corresponds to column indices of non-zero elements in P_ϕ^m.

Linear Indices and an Example XI

$$\begin{bmatrix} z_{11} \\ z_{21} \\ z_{12} \\ z_{22} \\ z_{21} \\ z_{31} \\ z_{22} \\ z_{32} \end{bmatrix} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & & 1 & & \\ & & & & 1 & \\ & 1 & & & & \\ & & 1 & & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \\ z_{31} \\ z_{12} \\ z_{22} \\ z_{32} \end{bmatrix} \tag{7}$$

A General Setting I

- We begin with checking how linear indices of $Z^{\text{in},i}$ can be mapped to the first column of $\phi(Z^{\text{in},i})$.
- For simplicity, we consider only channel j.
- From

$$
\begin{aligned}
& Z^{\text{in},i} \\
= & \begin{bmatrix}
z^i_{1,1,1} & z^i_{2,1,1} & \cdots & z^i_{a^{\text{in}},1,1} & z^i_{1,2,1} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},1} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
z^i_{1,1,j} & z^i_{2,1,j} & \cdots & z^i_{a^{\text{in}},1,j} & z^i_{1,2,j} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},j} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
z^i_{1,1,d^{\text{in}}} & z^i_{2,1,d^{\text{in}}} & \cdots & z^i_{a^{\text{in}},1,d^{\text{in}}} & z^i_{1,2,d^{\text{in}}} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},d^{\text{in}}}
\end{bmatrix}
\end{aligned}
$$

A General Setting II

we have

linear indices in z^{in}	values
j	$z^{\text{in}}_{1,1,j}$
$d^{\text{in}} + j$	$z^{\text{in}}_{2,1,j}$
$\vdots$	$\vdots$
$(h-1)d^{\text{in}} + j$	$z^{\text{in}}_{h,1,j}$
$a^{\text{in}}d^{\text{in}} + j$	$z^{\text{in}}_{1,2,j}$
$\vdots$	$\vdots$
$((h-1) + a^{\text{in}})d^{\text{in}} + j$	$z^{\text{in}}_{h,2,j}$
$\vdots$	$\vdots$
$((h-1) + (h-1)a^{\text{in}})d^{\text{in}} + j$	$z^{\text{in}}_{h,h,j}$

A General Setting III

- We rewrite linear indices in the earlier table as

$$\begin{bmatrix} 0 + 0a^{\text{in}} \\ \vdots \\ (h-1) + 0a^{\text{in}} \\ 0 + 1a^{\text{in}} \\ \vdots \\ (h-1) + 1a^{\text{in}} \\ \vdots \\ 0 + (h-1)a^{\text{in}} \\ \vdots \\ (h-1) + (h-1)a^{\text{in}} \end{bmatrix} d^{\text{in}} + j. \tag{8}$$

A General Setting IV

- Every linear index in (8) can be represented as

$$(p + qa^{\text{in}})d^{\text{in}} + j, \tag{9}$$

where

$$p,\ \ q \in \{0, \ldots, h-1\}$$

- Then $(p+1, q+1)$ correspond to the pixel position in the convolutional filter
- Next we consider other columns in $\phi(Z^{\text{in},i})$ by still fixing the channel to be j.

A General Setting V

- From

$$\phi(Z^{\text{in},i}) =$$

$$\begin{bmatrix}
\vdots & \vdots & & \vdots \\
z^i_{1,1,j} & z^i_{1+s,1,j} & & z^i_{1+(a^{\text{out}}-1)s,1+(b^{\text{out}}-1)s,j} \\
z^i_{2,1,j} & z^i_{2+s,1,j} & & z^i_{2+(a^{\text{out}}-1)s,1+(b^{\text{out}}-1)s,j} \\
\vdots & \vdots & \ldots & \vdots \\
z^i_{h,h,j} & z^i_{h+s,h,j} & & z^i_{h+(a^{\text{out}}-1)s,h+(b^{\text{out}}-1)s,j} \\
\vdots & \vdots & & \vdots \\
z^i_{h,h,d^{\text{in}}} & z^i_{h+s,h,d^{\text{in}}} & & z^i_{h+(a^{\text{out}}-1)s,h+(b^{\text{out}}-1)s,d^{\text{in}}}
\end{bmatrix}$$

A General Setting VI

each column contains the following elements from the jth channel of $Z^{\text{in},i}$.

$$z^{\text{in},i}_{1+p+as,1+q+bs,j}\ , \quad \begin{aligned} a &= 0, 1, \ldots, a^{\text{out}} - 1, \\ b &= 0, 1, \ldots, b^{\text{out}} - 1, \end{aligned} \tag{10}$$

where

$$(1 + as,\ 1 + bs)$$

is the top-left position of a sub-image in the channel j of $Z^{\text{in},i}$.

A General Setting VII

- From (6), the linear index of each element in (10) is

$$
\begin{aligned}
&\underbrace{\left((1+p+as-1)+(1+q+bs-1)a^{\text{in}}\right)}_{\text{column index in } Z^{\text{in},i}} d^{\text{in}} + j \\
=\ &(a+ba^{\text{in}})sd^{\text{in}} + \underbrace{(p+qa^{\text{in}})d^{\text{in}}+j}_{\text{see (9)}}. \qquad (11)
\end{aligned}
$$

- Now we have known for each element of $\phi(Z^{\text{in},i})$ what the corresponding linear index in $Z^{\text{in},i}$ is.
- Next we discuss the implementation details

A General Setting VIII

- First, we compute elements in (8) with $j = 1$ by applying Matlab's '+' operator, which has the implicit expansion behavior, to compute the outer sum of the following two arrays.

$$\begin{bmatrix} 1 \\ d^{\text{in}} + 1 \\ \vdots \\ (h-1)d^{\text{in}} + 1 \end{bmatrix}$$

and

$$\begin{bmatrix} 0 & a^{\text{in}}d^{\text{in}} & \ldots & (h-1)a^{\text{in}}d^{\text{in}} \end{bmatrix}.$$

A General Setting IX

- The result is the following matrix

$$\begin{bmatrix} 1 & a^{\text{in}}d^{\text{in}}+1 & \dots & (h-1)a^{\text{in}}d^{\text{in}}+1 \\ d^{\text{in}}+1 & (1+a^{\text{in}})d^{\text{in}}+1 & \dots & (1+(h-1)a^{\text{in}})d^{\text{in}}+1 \\ \vdots & \vdots & \dots & \vdots \\ (h-1)d^{\text{in}}+1 & ((h-1)+a^{\text{in}})d^{\text{in}}+1 & \dots & ((h-1)+(h-1)a^{\text{in}})d^{\text{in}}+1 \end{bmatrix} \tag{12}$$

- If columns are concatenated, we get (8) with $j=1$
- To get (9) for all channels $j=1,\dots,d^{\text{in}}$, we compute the outer sum:

$$\text{vec}((12)) + \begin{bmatrix} 0 & 1 & \dots & d^{\text{in}}-1 \end{bmatrix} \tag{13}$$

A General Setting X

- Then we have the first column of $\phi(Z^{\text{in},i})$
- Next, we obtain other columns in $\phi(Z^{\text{in},i})$
- In the linear indices in (11), the second term corresponds to indices of the first column, while the first term is the following column offset

$$
\begin{aligned}
(a + ba^{\text{in}})sd^{\text{in}}, \ \forall a &= 0, 1, \ldots, a^{\text{out}} - 1, \\
b &= 0, 1, \ldots, b^{\text{out}} - 1.
\end{aligned}
$$

A General Setting XI

- This is the outer sum of the following two arrays.

$$\begin{bmatrix} 0 \\ \vdots \\ a^{\text{out}} - 1 \end{bmatrix} \times sd^{\text{in}} \quad \text{and} \quad \begin{bmatrix} 0 & \dots & b^{\text{out}} - 1 \end{bmatrix} \times a^{\text{in}} sd^{\text{in}} \tag{14}$$

- Finally, we compute the outer sum of the column offset and the linear indices in the first column of $\phi(Z^{\text{in},i})$

$$\text{vec}((14))^T + \text{vec}((13)) \tag{15}$$

A General Setting XII

- In the end we store

$$\text{vec}((15)) \in R^{hhd^{\text{in}}a^{\text{out}}b^{\text{out}}\times 1}$$

  It is a vector collecting

  column index of the non-zero in each row of P_ϕ^m

- Note that each row in the $0/1$ matrix P_ϕ^m contains exactly only one non-zero element.
- See the example in (7)
- The obtained linear indices are independent of the values of $Z^{\text{in},i}$.

A General Setting XIII

- Thus the above procedure only needs to be run once in the beginning.

A Simple Code I

```
function idx = find_index_phiZ(a,b,d,h,s)

first_channel_idx = ([0:h-1]*d+1)' +
                     [0:h-1]*a*d;
first_col_idx = first_channel_idx(:) + [0:d-1];
a_out = floor((a - h)/s) + 1;
b_out = floor((b - h)/s) + 1;
column_offset = ([0:a_out-1]' +
                 [0:b_out-1]*a)*s*d;
idx = column_offset(:)' + first_col_idx(:);
idx = idx(:);
```

Discussion

- The code is simple and short
- We assume that Matlab operations used here are efficient and so is our resulting code
- But is that really the case?
- We will do experiments to check this
- Some works have tried to do similar things (e.g., `https://github.com/wiseodd/hipsternet`), though we don't see complete documents and evaluation

Outline

1 Introduction

2 Storage

3 Generation of $\phi(\text{pad}(Z^{m,i}))$

4 Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$

5 Discussion

$(\boldsymbol{v}^i)^T P_\phi^m$ I

- In the backward process, the following operation is applied.

$$(\boldsymbol{v}^i)^T P_\phi^m, \tag{16}$$

where

$$\boldsymbol{v}^i = \text{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)$$

- Consider the same example used for $\phi(Z^{\text{in},i})$

$(\boldsymbol{v}^i)^T P_\phi^m$ II

- We have

$$P_\phi^m = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & & 1 & & \\ & & & & 1 & \\ & 1 & & & & \\ & & 1 & & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

$(\boldsymbol{v}^i)^T P_\phi^m$ III

- Thus

$$(P_\phi^m)^T \boldsymbol{v}^i = \begin{bmatrix} v_1 & v_2+v_5 & v_6 & v_3 & v_4+v_7 & v_8 \end{bmatrix}^T, \quad (17)$$

which is a kind of “inverse” operation of $\phi(\text{pad}(Z^{m,i}))$

- We accumulate elements in $\phi(\text{pad}(Z^{m,i}))$ back to their original positions in $\text{pad}(Z^{m,i})$.

$(\boldsymbol{v}^i)^T P_\phi^m$ IV

- In *MATLAB*, given indices

$$\begin{bmatrix}1 & 2 & 4 & 5 & 2 & 3 & 5 & 6\end{bmatrix}^T \tag{18}$$

and the vector $\boldsymbol{v}$, a function `accumarray` can directly generate the vector (17).

- Example:

$(\mathbf{v}^i)^T P_\phi^m$ V

```
octave:18> [v a]
ans =

    1    0.406445
    2    0.067872
    4    0.036638
    5    0.279801
    2    0.490535
    3    0.369743
    5    0.429186
    6    0.054324
```

$(v^i)^T P_\phi^m$ VI

```
octave:19> accumarray(v,a)
ans =

   0.406445
   0.558407
   0.369743
   0.036638
   0.708987
   0.054324
```

$(\boldsymbol{v}^i)^T P_\phi^m$ VII

- To do the calculation over a batch of instances, we aim to have

$$\begin{bmatrix} (P_\phi^m)^T \boldsymbol{v}^1 \\ \vdots \\ (P_\phi^m)^T \boldsymbol{v}^l \end{bmatrix}^T. \tag{19}$$

- We can apply *MATLAB*'s `accumarray` on the vector

$$\begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^l \end{bmatrix}, \tag{20}$$

$(\boldsymbol{v}^i)^T P_\phi^m$ VIII

by giving the following indices as the input.

$$\begin{bmatrix} (18) \\ (18) + a^m_{\text{pad}} b^m_{\text{pad}} d^m \mathbb{1}_{h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}} \\ (18) + 2a^m_{\text{pad}} b^m_{\text{pad}} d^m \mathbb{1}_{h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}} \\ \vdots \\ (18) + (l-1)a^m_{\text{pad}} b^m_{\text{pad}} d^m \mathbb{1}_{h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}} \end{bmatrix}, \quad (21)$$

where

$$a^m_{\text{pad}} b^m_{\text{pad}} d^m \text{ is the size of } \text{pad}(Z^{m,i})$$

$(\boldsymbol{v}^i)^T P_\phi^m$ IX

and

$h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}$ is the size of $\phi(\text{pad}(Z^{m,i}))$ and $\boldsymbol{v}_i$.

- That is, by using the offset $(i-1)a^m_{\text{pad}} b^m_{\text{pad}} d^m$, `accumarray` accumulates $\boldsymbol{v}^i$ to the following positions:

$$(i-1)a^m_{\text{pad}} b^m_{\text{pad}} d^m + 1, \ \ldots, \ i a^m_{\text{pad}} b^m_{\text{pad}} d^m. \tag{22}$$

$(\boldsymbol{v}^i)^T P_\phi^m X$

- (21) can be easily obtained by the following outer sum

$$\text{vec}((18) + \begin{bmatrix} 0 & \dots & l-1 \end{bmatrix} a_{\text{pad}}^m b_{\text{pad}}^m d^m)$$

- To obtain

$$\begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^l \end{bmatrix}$$

  we note that it is the same as

$$\text{vec}\left((W^m)^T \begin{bmatrix} \frac{\partial \xi_1}{\partial S^{m,1}} & \dots & \frac{\partial \xi_l}{\partial S^{m,l}} \end{bmatrix} \right). \qquad (23)$$

$(\boldsymbol{v}^i)^T P_\phi^m$ XI

- Thus we do a matrix-matrix multiplication
- From (23), we can see why $\partial\xi_i/\partial\text{vec}(S^{m,i})^T$ over a batch of instances are stored in the form of

$$\begin{bmatrix} \frac{\partial\xi_1}{\partial S^{m,1}} & \dots & \frac{\partial\xi_l}{\partial S^{m,l}} \end{bmatrix} \in R^{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m l}.$$

A Simple Code I

```
a_prev = model.ht_pad(m);
b_prev = model.wd_pad(m);
d_prev = model.ch_input(m);

idx = net.idx_phiZm(:) +
      [0:num_v-1]*d_prev*a_prev*b_prev;
vTP = accumarray(idx(:), V(:),
      [d_prev*a_prev*b_prev*num_v 1])';
```

A Simple Code II

- Here we assume

$$V = \begin{bmatrix} \boldsymbol{v}_1 & \cdots & \boldsymbol{v}_l \end{bmatrix}$$

and `num_v` is the number of columns

Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(\boldsymbol{v}^i)^T P_\phi^m$
5. Discussion

Efficient Implementation I

- If a package provides efficient implementations of the following operations
  - matrix-matrix products
  - matrix expansion for $\phi(\mathrm{pad}(Z^{m,i}))$
  - outer sum
  - `accumarray`

  then we can easily have a good CNN implementation
- Unfortunately, the difficulty to optimize these operations may vary

Discussion I

- To work on instances together, it's difficult to decide the best storage settings
- Further, storage settings affect the implementations
- Do you think our setting is already the best?
- How could we easily check the running time of using different storage settings? Is our code flexible enough for such experiments?

References I