Implementation

Chih-Jen Lin
National Taiwan University

Last updated: May 9, 2019
Outline

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

2. Storage

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

4. Evaluation of $(\nu^i)^T P^m_\phi$

5. Discussion
After checking formulations for gradient calculation we would like to get into implementation details.

Take the following operation as an example:

\[
\frac{\partial \xi_i}{\partial W^m} = \frac{\partial \xi_i}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}^m))^T
\]

- 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.
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

\[ C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \]
Let's test the matrix multiplication

A C program:

```c
#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

\[
\begin{align*}
    a[i][j] &= 1; \quad b[i][j] = 1; \\
    \quad & \} \\
    \text{for (i=0; i<n; i++)} \\
    \quad \text{for (j=0; j<n; j++)} \\
    \quad \quad c[i][j] &= 0; \\
    \quad \quad \text{for (k=0; k<n; k++)} \\
    \quad \quad \quad c[i][j] &= a[i][k]*b[k][j]; \\
    \quad \} \}
\end{align*}
\]
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];
}

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
  
  ```matlab
  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
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
  localize operations to achieve good performance
Memory Hierarchy I

- CPU
  - Registers
  - Cache
  - Main Memory
  - Secondary storage (Disk)
Memory Hierarchy II

- ↑: increasing in speed
- ↓: 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
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, \( \frac{1024}{64} = 16 \)

Assumption: each time a continuous segment of data into one page

Approach 1: inner product
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
Total: $1024^2 \times 16$ page faults

- at least 16 million page faults

Approach 2:

```plaintext
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
```
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 $A$

$1024 \times 16$ page faults

$C, B : 16$ page faults

Approach 3: block algorithms ($nb = 256$)
for \( j = 1:nb:n \)
  for \( k=1:nb:n \)
    for \( jj=j:j+nb-1 \)
      for \( kk=k:k+nb-1 \)
        \[ c(:,jj) = a(:,kk) \times b(kk,jj) + c(:,jj); \]
      end
    end
  end
end

In MATLAB, 1:256:1025 means 1, 257, 513, 769
Note that we calculate

\[
\begin{bmatrix}
A_{11} & \cdots & A_{14} \\
\vdots & & \vdots \\
A_{41} & \cdots & A_{44}
\end{bmatrix}
\begin{bmatrix}
B_{11} & \cdots & B_{14} \\
\vdots & & \vdots \\
B_{41} & \cdots & B_{44}
\end{bmatrix}
= 
\begin{bmatrix}
A_{11}B_{11} + \cdots + A_{14}B_{41} & \cdots \\
\vdots & \ddots & \vdots
\end{bmatrix}
\]
Memory Management VIII

- Each block: 256 × 256

\[ C_{11} = A_{11}B_{11} + \cdots + A_{14}B_{41} \]
\[ C_{21} = A_{21}B_{11} + \cdots + A_{24}B_{41} \]
\[ C_{31} = A_{31}B_{11} + \cdots + A_{34}B_{41} \]
\[ C_{41} = A_{41}B_{11} + \cdots + A_{44}B_{41} \]

- 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 $A$: $16 \times 4$ 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
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”
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?
If they can also be done by calling others’ efficient implementation, then a simple and efficient CNN implementation can be done.

In the past few months my group has been building such a code.

See the repository at https://github.com/cjlin1/simplenn.

The purpose of that code is neither for industry use nor for quickly trying various architectures.

Instead, it aims to be a simple end-to-end code for education, and
Introduction

Discussion III

- experiments on optimization algorithms
- 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 $(v^i)^T P^m_{\phi}$
5. Discussion
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

We assume \( l \) is the number of data instances used to calculate the gradient (or the sub-gradient)
In our implementation, we store $Z^{m,i}$, $\forall i = 1, \ldots, l$ as the following matrix.

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

Similarly, we store

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

$$
\begin{bmatrix}
\frac{\partial \xi_1}{\partial S^{m,1}} & \ldots & \frac{\partial \xi_l}{\partial S^{m,l}}
\end{bmatrix} \in \mathbb{R}^{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m l}.
$$
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.
Recall that we conduct the following operations

\[
\frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} = \left( \frac{\partial \xi_i}{\partial \text{vec}(Z^{m+1,i})^T} \odot \text{vec}(I[Z^{m+1,i}])^T \right) P_{m,i}^{\text{pool}}
\]

(3)

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

(4)

\[
\frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} = \text{vec} \left( (W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)^T P_{\phi}^m P_{\text{pad}}^m
\]

(5)
Based on the way discussed to store variables, we will discuss two operations in detail:

- Generation of $\phi(pad(Z^{m,i}))$
- $\text{vector} \times P^{m}_\phi$
Outline

1 Introduction

2 Storage

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

4 Evaluation of $(\nu^i)^T P^m_\phi$

5 Discussion
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 \texttt{im2col} that can generate \( \phi(\text{pad}(Z_{m,i})) \) for \( s = 1 \) and \( s = h \) (width of filter).

But this function cannot handle general \( s \).

Can we do a reasonably efficient implementation by ourselves?
For an easy description we consider

$$\text{pad}(Z_{m,i}^{m,i}) = Z_{\text{in},i}^{\text{in},i} \rightarrow Z_{\text{out},i}^{\text{out},i} = \phi(Z_{\text{in},i}^{\text{in},i}).$$
Consider the following column-oriented linear indices (i.e., counting elements in a column-oriented way) of $\mathbb{Z}^{\text{in},i}$:

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

(6)
Every element in

$$\phi(Z_{in,i}^m, i) \in R^{hhd_{in} \times a_{out} b_{out}}$$

is extracted from $Z_{in,i}^m$.

The task is to find the mapping between each element in $\phi(Z_{in,i}^m)$ and a linear index of $Z_{in,i}^m$.

Consider an example with

$$a_{in} = 3, \ b_{in} = 2, \ d_{in} = 1.$$  

Because $d_{in} = 1$, we omit the channel subscript.
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

\[
\begin{bmatrix}
z_{11} & z_{12} \\
z_{21} & z_{22}
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
z_{21} & z_{22} \\
z_{31} & z_{32}
\end{bmatrix}
\]
Linear Indices and an Example IV

- By our earlier way of representing images,

\[ Z^{in,i} = \begin{bmatrix} z_{1,1,1}^i & z_{2,1,1}^i & \cdots & z_{a_{in},b_{in},1}^i \\
\vdots & \vdots & \ddots & \vdots \\
z_{1,1,d_{in}}^i & z_{2,1,d_{in}}^i & \cdots & z_{a_{in},b_{in},d_{in}}^i \end{bmatrix} \]

the one we have is

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

- The linear indices from (6) are

\[ \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \].
Recall that

$$
\phi(Z_{\text{in},i}) = 
\begin{bmatrix}
    z_{1,1,1} & z_{1}^{i} & z_{1}^{i} + (a_{\text{out}}-1)s,1,(b_{\text{out}}-1)s,1 \\
    z_{2,1,1} & z_{2}^{i} & z_{2}^{i} + (a_{\text{out}}-1)s,1,(b_{\text{out}}-1)s,1 \\
    \vdots & \vdots & \vdots \\
    z_{h,h,1} & z_{h}^{i} & z_{h}^{i} + (a_{\text{out}}-1)s,h,(b_{\text{out}}-1)s,1 \\
    \vdots & \vdots & \vdots \\
    z_{h,h,d_{\text{in}}} & z_{h}^{i} & z_{h}^{i} + (a_{\text{out}}-1)s,h,(b_{\text{out}}-1)s,d_{\text{in}} \\
\end{bmatrix}
$$
Linear Indices and an Example VI

- Therefore,

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

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

\[
Z^{m,i} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}^T
\]

\[
\phi(Z^{m,i}) = \begin{bmatrix} 1 & 2 & 4 & 5 & 2 & 3 & 5 & 6 \end{bmatrix}^T.
\]

- Example of using Matlab/Octave
Generation of $\phi(pad(Z^m_{ij}))$

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

To handle all instances together, we store

\[ Z^{in,1}, \ldots, Z^{in,l} \]

as

\[ \left[ \text{vec}(Z^{in,1}) \; \ldots \; \text{vec}(Z^{in,l}) \right] \]
Linear Indices and an Example IX

- Denote it as a MATLAB matrix
  \[ Z \]

- Then
  \[
  \begin{bmatrix}
  \text{vec}(\phi(Z_m^1)) & \ldots & \text{vec}(\phi(Z_m^l))
  \end{bmatrix}
  \]

is simply

\[ Z(P, :) \]

in MATLAB, where we store the mapping by

\[ P = [1 \ 2 \ 4 \ 5 \ 2 \ 3 \ 5 \ 6]^T \]
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

$$[1 \ 2 \ 4 \ 5 \ 2 \ 3 \ 5 \ 6]^T.$$  

also corresponds to column indices of non-zero elements in $P^m_{\phi}$. 

Chih-Jen Lin (National Taiwan Univ.)
Linear Indices and an Example XI

\[
\begin{bmatrix}
  z_{11} \\
  z_{21} \\
  z_{12} \\
  z_{22} \\
  z_{21} \\
  z_{31} \\
  z_{22} \\
  z_{32}
\end{bmatrix}
= 
\begin{bmatrix}
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  z_{11} \\
  z_{21} \\
  z_{31} \\
  z_{12} \\
  z_{22} \\
  z_{32}
\end{bmatrix}
\tag{7}
\]
A General Setting I

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

$$Z^{in,i} = \begin{bmatrix}
z_1^{i,1,1} & z_2^{i,1,1} & \cdots & z_{a^{in},b^{in},1}^{i} \\
\vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \ddots & \vdots \\
z_1^{i,1,j} & z_2^{i,1,j} & \cdots & z_{a^{in},b^{in},j}^{i} \\
\vdots & \vdots & \ddots & \vdots \\
z_1^{i,1,d^{in}} & z_2^{i,1,d^{in}} & \cdots & z_{a^{in},b^{in},d^{in}}^{i} \\
\end{bmatrix},$$
we have

<table>
<thead>
<tr>
<th>Linear indices in $z^\text{in}$</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>$z^\text{in}_{1,1,j}$</td>
</tr>
<tr>
<td>$d^\text{in} + j$</td>
<td>$z^\text{in}_{2,1,j}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$(h - 1)d^\text{in} + j$</td>
<td>$z^\text{in}_{h,1,j}$</td>
</tr>
<tr>
<td>$a^\text{in}d^\text{in} + j$</td>
<td>$z^\text{in}_{1,2,j}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$((h - 1) + a^\text{in})d^\text{in} + j$</td>
<td>$z^\text{in}_{h,2,j}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$((h - 1) + (h - 1)a^\text{in})d^\text{in} + j$</td>
<td>$z^\text{in}_{h,h,j}$</td>
</tr>
</tbody>
</table>
A General Setting III

- We rewrite linear indices in the earlier table as

\[
\begin{bmatrix}
0 + 0a^{\text{in}} \\
\vdots \\
(h - 1) + 0a^{\text{in}} \\
0 + 1a^{\text{in}} \\
\vdots \\
(h - 1) + 1a^{\text{in}} \\
\vdots \\
0 + (h - 1)a^{\text{in}} \\
\vdots \\
(h - 1) + (h - 1)a^{\text{in}}
\end{bmatrix} + d^{\text{in}} + j. 
\] (8)
A General Setting IV

Every linear index in (8) can be represented as

\[(p + qa^{in})d^{in} + j,\]  \hspace{1cm} (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 \(\phi(Z^{in,i})\) by still fixing the channel to be \(j\).
A General Setting V

From

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

$$\begin{bmatrix}
    z_{1,1,1}^i & z_{1+s,1,1}^i \\
    z_{2,1,1}^i & z_{2+s,1,1}^i \\
    \vdots & \vdots & \ddots & \vdots \\
    z_{h,h,1}^i & z_{h+s,h,1}^i \\
    \vdots & \vdots \\
    z_{h,h,d^{in}}^i & z_{h+s,h,d^{in}}^i \\
\end{bmatrix}
\begin{bmatrix}
    z_{1+(a^{out}-1)s,1+(b^{out}-1)s,1}^i \\
    z_{2+(a^{out}-1)s,1+(b^{out}-1)s,1}^i \\
    \vdots \\
    z_{h+(a^{out}-1)s,h+(b^{out}-1)s,1}^i \\
    \vdots \\
    z_{h+(a^{out}-1)s,h+(b^{out}-1)s,d^{in}}^i \\
\end{bmatrix}$$
A General Setting VI

each column contains the following elements from the $j$th channel of $Z_{in,i}$.

\[ Z_{1+p+as,1+q+bs,j}^{in,i}, \quad a = 0, 1, \ldots, a^{out} - 1, \]
\[ b = 0, 1, \ldots, b^{out} - 1, \quad (10) \]

where

\[(1 + as, 1 + bs)\]

is the top-left position of a sub-image in the channel $j$ of $Z_{in,i}$. 
From (6), the linear index of each element in (10) is
\[
((1 + p + as - 1) + (1 + q + bs - 1)a^\text{in}) d^\text{in} + j
\]
(column index in $Z^{\text{in},i}$)
\[
= (a + ba^\text{in})sd^\text{in} + (p + qa^\text{in})d^\text{in} + j.
\]
(11)

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.
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.

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

and

\[
\begin{bmatrix}
0 & a^{in}d^{in} & \ldots & (h - 1)a^{in}d^{in}
\end{bmatrix}
\]
The result is the following matrix

$$
\begin{bmatrix}
1 & a^{in}d^{in} + 1 & \ldots & (h - 1)a^{in}d^{in} + 1 \\
(d^{in} + 1) & (1 + a^{in})d^{in} + 1 & \ldots & (1 + (h - 1)a^{in})d^{in} + 1 \\
\vdots & \vdots & \ddots & \vdots \\
(h - 1)d^{in} + 1 & ((h - 1) + a^{in})d^{in} + 1 & \ldots & ((h - 1) + (h - 1)a^{in})d^{in} + 1 \\
\end{bmatrix}
$$

(12)

If columns are concatenated, we get (8) with \( j = 1 \)

To get (9) for all channels \( j = 1, \ldots, d^{in} \), we compute the outer sum:

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

(13)
A General Setting $X$

- Then we have the first column of $\phi(Z_{\text{in},i}^\text{in})$
- Next, we obtain other columns in $\phi(Z_{\text{in},i}^\text{in})$
- 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

$$(a + ba^{\text{in}})sd^{\text{in}}, \forall a = 0, 1, \ldots, a^{\text{out}} - 1,$$

$$b = 0, 1, \ldots, b^{\text{out}} - 1.$$
This is the outer sum of the following two arrays.

\[
\begin{bmatrix}
0 \\
\vdots \\
a^{\text{out}} - 1
\end{bmatrix} \times s d^{\text{in}} \quad \text{and} \quad \begin{bmatrix}
0 & \ldots & b^{\text{out}} - 1
\end{bmatrix} \times a^{\text{in}} s d^{\text{in}}
\]

(14)

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

\[
\text{vec}((14))^T + \text{vec}((13))
\]

(15)
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 \( P^m_{\phi} \).

Note that each row in the 0/1 matrix \( P^m_{\phi} \) contains exactly only one non-zero element.

See the example in (7)

The obtained linear indices are independent of the values of \( Z_{\text{in},i} \).
Thus the above procedure only needs to be run once in the beginning.
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 $(\mathbf{v}^i)^T P^m_{\phi}$
5. Discussion
In the backward process, the following operation is applied.

\[(\mathbf{v}^i)^T P^m_\phi, \tag{16}\]

where

\[\mathbf{v}^i = \text{vec} \left( (W^m)^T \frac{\partial \xi_i}{\partial S^m_i} \right)\]

Consider the same example used for \(\phi(Z^{in,i})\)
We have

\[ P^m_\phi = \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix} \]
Thus

$$(P^m_\phi)^T \mathbf{v}^i = \begin{bmatrix} v_1 & v_2 + v_5 & v_6 & v_3 & v_4 + v_7 & v_8 \end{bmatrix}^T,$$

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 $\text{pad}(Z^{m,i})$. 
In MATLAB, given indices

\[
[1 \ 2 \ 4 \ 5 \ 2 \ 3 \ 5 \ 6]^T
\]  \hspace{1cm} (18)

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

Example:
Evaluation of $(\mathbf{v}^i)^T \mathbf{P}_\phi^m \mathbf{V}$

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

1   0.406445
2   0.067872
4   0.036638
5   0.279801
2   0.490535
3   0.369743
5   0.429186
6   0.054324
```
Evaluation of $(\mathbf{v}^i)^T P^m_{\phi} \mathbf{VI}$

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

0.406445
0.558407
0.369743
0.036638
0.708987
0.054324
```
To do the calculation over a batch of instances, we aim to have

\[
\begin{bmatrix}
(P^m_\phi)^T \mathbf{v}^1 \\
\vdots \\
(P^m_\phi)^T \mathbf{v}^l
\end{bmatrix}^T.
\]  \hspace{1cm} (19)

We can apply MATLAB’s accumarray on the vector

\[
\begin{bmatrix}
\mathbf{v}^1 \\
\vdots \\
\mathbf{v}^l
\end{bmatrix},
\]  \hspace{1cm} (20)
by giving the following indices as the input.

\[
\begin{bmatrix}
(18) \\
(18) + a_{pad}^m b_{pad}^m d^m \mathbb{1} h^m h^m d^m a_{conv}^m b_{conv}^m \\
(18) + 2a_{pad}^m b_{pad}^m d^m \mathbb{1} h^m h^m d^m a_{conv}^m b_{conv}^m \\
\vdots \\
(18) + (l - 1)a_{pad}^m b_{pad}^m d^m \mathbb{1} h^m h^m d^m a_{conv}^m b_{conv}^m \\
\end{bmatrix}, \quad (21)
\]

where

\[a_{pad}^m b_{pad}^m d^m\] is the size of pad(Z^{m,i})
\[(v^i)^T P^m_\phi \] 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 \(v_i\).

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

\[(i - 1)a^m_{\text{pad}} b^m_{\text{pad}} d^m + 1, \ldots, ia^m_{\text{pad}} b^m_{\text{pad}} d^m. \] (22)
(21) can be easily obtained by the following outer product

$$\text{vec}((18) + [0 \ldots l - 1] a_{\text{pad}}^m b_{\text{pad}}^m d^m)$$

To obtain

$$\begin{bmatrix} \nu^1 \\ \vdots \\ \nu^l \end{bmatrix}$$

we note that it is the same as

$$\text{vec}((W^m)^T \left[ \frac{\partial \xi_1}{\partial S^{m,1}} \ldots \frac{\partial \xi_l}{\partial S^{m,l}} \right]) \cdot (23)$$
Thus we do a matrix-matrix multiplication

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

$$\left[ \frac{\partial \xi_1}{\partial S_{m,1}} \cdots \frac{\partial \xi_l}{\partial S_{m,l}} \right] \in \mathbb{R}^{d_{m+1} \times a_{conv} \times b_{conv} \times l}.$$
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])';
Here we assume

$$V = \begin{bmatrix} v_1 & \cdots & v_l \end{bmatrix}$$

and num_v is the number of columns
Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z_m^m, i))$
4. Evaluation of $(v^i)^T P_m^m$
5. Discussion
Efficient Implementation I

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

then we can easily have a good CNN implementation

- A comparison between MATLAB and Octave will see their respective strengths and weaknesses
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 do easily check the running time of using different storage settings? Is our code flexible enough for such experiments?
References I