Implementation

Chih-Jen Lin
National Taiwan University

Last updated: April 6, 2019
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. Computational Complexity
6. Discussion
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. Computational Complexity
6. Discussion
After checking formulations for gradient calculation we would like to get into implementation details.

Take the following operation as an example:

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

It’s a matrix-matrix product.

We all know that a three-level for loop does the job.

Does that mean we can then have an efficient implementation?

The answer is no.
To explain this, we check some details of matrix-matrix products
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:
  ```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++)
        c[i][j] = a[i][k] * b[k][j];
  }
  ```
Optimized BLAS: an Example by Using Block Algorithms II

\[
a[i][j] = 1; \quad b[i][j] = 1;
\]

\[
\text{for (i=0;i<n;i++)}
\]
\[
\quad \text{for (j=0;j<n;j++)}
\]
\[
\quad \quad c[i][j] = 0;
\]
\[
\quad \quad \text{for (k=0;k<n;k++)}
\]
\[
\quad \quad \quad c[i][j] += a[i][k] \times b[k][j];
\]

Chih-Jen Lin (National Taiwan Univ.)
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?
- C is row-oriented rather than column-oriented
- Now we sense that memory access can be an issue
- Let’s try a Matlab program
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

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:

```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)*b(kk,jj)+c(:,jj);
            end
        end
    end
end

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

\[
\begin{bmatrix}
A_{11} & \cdots & A_{14} \\
\vdots & & \vdots \\
A_{41} & \cdots & A_{44}
\end{bmatrix}
\begin{bmatrix}
B_{11} & \cdots & B_{14} \\
\vdots & & \vdots \\
B_{41} & \cdots & B_{44}
\end{bmatrix}
= \begin{bmatrix}
A_{11}B_{11} + \cdots + A_{14}B_{41} & \cdots \\
\vdots & \ddots & \ddots
\end{bmatrix}
\]
Each block: $256 \times 256$

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

For each $(j, k)$, $B_{k,j}$ is used to add $A_{:,k}B_{k,j}$ to $C_{:,j}$
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
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”
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 issues
- Parallelization can be applied for deep learning
- 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 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
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 $(\nu^i)^T P^m_\phi$
5. Computational Complexity
6. 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
In our implementation, we store $Z_{m,i}^{}, \ \forall i = 1, \ldots, l$ as the following matrix.

$$
\begin{bmatrix}
Z_{m,1}^{} & Z_{m,2}^{} & \ldots & Z_{m,l}^{}
\end{bmatrix} \in R^{d_{m} \times a_{m} b_{m} l}.
$$

(2)

Similarly, we store

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

as

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

(3)

We will explain our decision.
Note that (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 in some places we follow Section ?? 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

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

(4)

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

(5)

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

(6)
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 $\phi(pad(Z^m,i))$
4. Evaluation of $\left(v^i\right)^T P^m_\phi$
5. Computational Complexity
6. Discussion
Due to the wide use of CNN, a subrouting for $\phi(p\text{ad}(Z^{m,i}))$ has been available in some packages. For example, *MATLAB* has a built-in function `im2col` that can generate $\phi(p\text{ad}(Z^{m,i}))$ for $s = 1$ and $s = h$ (width of filter).

Further, I don’t think it can handle multiple images together? Can we do a reasonably efficient implementation by ourselves?
For an easy description we consider

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

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

(7)
Linear Indices and an Example II

- Every element in

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

is extracted from \( Z^{in,i} \).

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

- Consider an example with

\[
\begin{align*}
a^{in} & = 3, & b^{in} & = 2, & d^{in} & = 1.
\end{align*}
\]

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 $h = 2, s^m = 1$, two sub-images are

$$\begin{bmatrix}
z_{11} & z_{12} \\
z_{21} & z_{22}
\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}
z_{21} & z_{22} \\
z_{31} & z_{32}
\end{bmatrix}.$$
By our earlier way of representing images,

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

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 (7) are

\[
\begin{bmatrix}
    1 & 2 & 3 & 4 & 5 & 6
\end{bmatrix}
\]
**Linear Indices and an Example V**

- Recall that

\[
\phi(Z_{\text{in},i}) =
\begin{bmatrix}
Z_{1,1,1}^i & Z_{1+s,1,1}^i \\
Z_{2,1,1}^i & Z_{2+s,1,1}^i \\
\vdots & \vdots & \ddots & \vdots \\
Z_{h,h,1}^i & Z_{h+s,h,1}^i \\
\vdots & \vdots & & \\
Z_{h,h,d_{\text{in}}}^i & Z_{h+s,h,d_{\text{in}}}^i \\
\end{bmatrix}
\begin{bmatrix}
Z_1^{i+(a_{\text{out}}-1)s,1+(b_{\text{out}}-1)s,1} \\
Z_2^{i+(a_{\text{out}}-1)s,1+(b_{\text{out}}-1)s,1} \\
\vdots \\
Z_{h^{i+(a_{\text{out}}-1)s,h+(b_{\text{out}}-1)s,1}} \\
\vdots \\
Z_{h^{i+(a_{\text{out}}-1)s,h+(b_{\text{out}}-1)s,d_{\text{in}}}} \\
\end{bmatrix}
\]
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 $\phi(Z^m)$:

$$Z^{m,i} \quad [1 \ 2 \ 3 \ 4 \ 5 \ 6]^T$$

$$\phi(Z^{m,i}) \quad [1 \ 2 \ 4 \ 5 \ 2 \ 3 \ 5 \ 6]^T.$$
To handle all instances together, we store $Z^{in,1}, \ldots, Z^{in,l}$ as

$$[\text{vec}(Z^{in,1}) \ldots \text{vec}(Z^{in,l})]$$

Denote it as a MATLAB matrix $Z$.
Linear Indices and an Example VIII

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

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

- All instances handled in one line
- But how to obtain $P$?
Linear Indices and an Example IX

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 \( P^m_\phi \).

\[
\begin{bmatrix}
Z_{11} \\
Z_{21} \\
Z_{21} \\
Z_{31} \\
Z_{12} \\
Z_{22} \\
Z_{22} \\
Z_{32}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
Z_{11} \\
Z_{21} \\
Z_{31} \\
Z_{12} \\
Z_{22} \\
Z_{32}
\end{bmatrix}
\]

(8)
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

$$Z_{in,i} = \begin{bmatrix}
    z_{1,1,1}^i & z_{2,1,1}^i & \cdots & z_{a_{in},b_{in},1}^i \\
    \vdots      & \vdots      & \ddots & \vdots \\
    z_{1,1,d_{in}}^i & z_{2,1,d_{in}}^i & \cdots & z_{a_{in},b_{in},d_{in}}^i
\end{bmatrix},$$
A General Setting II

we have

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

- We rewrite linear indices in the earlier table as

$$
\begin{bmatrix}
0 + 0a^{in} \\
\vdots \\
(h - 1) + 0a^{in} \\
0 + 1a^{in} \\
\vdots \\
(h - 1) + 1a^{in} \\
\vdots \\
0 + (h - 1)a^{in} \\
\vdots \\
(h - 1) + (h - 1)a^{in}
\end{bmatrix}

\text{d}^{in} + j. \quad (9)
$$
Every linear index in (9) can be represented as

\[(p + qa^{in})d^{in} + j,\]  \hspace{1cm} (10)

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

Next we consider other columns in $\phi(Z^{in,i})$ by still fixing the channel to be $j$.

From

$$\phi(Z^{in,i}) =
\begin{bmatrix}
Z^i_1,1,1 & Z^i_1+s,1,1 \\
Z^i_2,1,1 & Z^i_2+s,1,1 \\
\vdots & \vdots & \vdots \\
Z^i_h,h,1 & Z^i_h+s,h,1 \\
Z^i_h,h,d^{in} & Z^i_h+s,h,d^{in}
\end{bmatrix}
\begin{bmatrix}
Z^i_1+(a^{out}-1)s,1+(b^{out}-1)s,1 \\
Z^i_2+(a^{out}-1)s,1+(b^{out}-1)s,1 \\
\vdots \\
Z^i_h+(a^{out}-1)s,h+(b^{out}-1)s,1 \\
Z^i_h+(a^{out}-1)s,h+(b^{out}-1)s,d^{in}
\end{bmatrix}$$
A General Setting VI

each column contains the following elements from the \( j \)th channel of \( Z_{\text{in},i} \).

\[
Z_{1+p+as,1+q+bs,j}^{\text{in},i}, \quad a = 0, 1, \ldots, a^{\text{out}} - 1,
\]
\[
b = 0, 1, \ldots, b^{\text{out}} - 1, \quad (11)
\]

where

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

is the top-left position of a sub-image in the channel \( j \) of \( Z_{\text{in},i} \).
From (7), the linear index of each element in (11) is

$$((1 + p + as - 1) + (1 + q + bs - 1)a^{in})d^{in} + j$$

column index in $Z^{in,i}$

$$= (a + ba^{in})sd^{in} + (p + qa^{in})d^{in} + j.$$

(12)

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

Next we discuss the implementation details
First, we compute elements in (9) 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

\[
\begin{pmatrix}
1 & a^{\text{in}}d^{\text{in}} + 1 & \ldots & (h - 1)a^{\text{in}}d^{\text{in}} + 1 \\
d^{\text{in}} + 1 & (1 + a^{\text{in}})d^{\text{in}} + 1 & \ldots & (1 + (h - 1)a^{\text{in}})d^{\text{in}} + 1 \\
\vdots & \vdots & \ddots & \vdots \\
(h - 1)d^{\text{in}} + 1 & ((h - 1) + a^{\text{in}})d^{\text{in}} + 1 & \ldots & ((h - 1) + (h - 1)a^{\text{in}})d^{\text{in}} + 1
\end{pmatrix}
\]

(13)

- If columns are concatenated, we get (9) with \( j = 1 \)

- To get (10) for all channels \( j = 1, \ldots, d^{\text{in}} \), we compute the outer sum:

\[
\text{vec}((13)) + \begin{bmatrix} 0 & 1 & \ldots & d^{\text{in}} - 1 \end{bmatrix},
\]

(14)
A General Setting X

- Next, we obtain other columns in $\phi(Z^{in,i})$
- In the linear indices in (12), the second term corresponds to indices of the first column, while the first term is the following column offset

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

$$b = 0, 1, \ldots, b^{out} - 1.$$
A General Setting XI

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

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

(15)

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

\[
\text{vec}( (15))^T + \text{vec}( (14))
\]

(16)
A General Setting XII

In the end we store

$$\text{vec((16)))} \in \mathbb{R}^{hhd^{in}a^{out}b^{out} \times 1}$$

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 (8)
- 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. Computational Complexity
6. Discussion
In the backward process, the following operation is applied.

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

(17)

where

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

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

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

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

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

\[
\begin{bmatrix}
1 & 2 & 4 & 5 & 2 & 3 & 5 & 6
\end{bmatrix}^T
\]  
(19)

and the vector \( \mathbf{v} \), a function ` accumarray` can directly generate the vector (18).

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

\[
\begin{bmatrix}
(P_m \phi)^T \mathbf{v}_1 \\
\vdots \\
(P_m \phi)^T \mathbf{v}_l
\end{bmatrix}^T.
\]  
(20)
We can apply *MATLAB*’s accumarray on the vector

\[
\begin{bmatrix}
\mathbf{v}^1 \\
\vdots \\
\mathbf{v}^l
\end{bmatrix},
\]

by giving the following indices as the input.

\[
\begin{bmatrix}
(19) \\
(19) + a_{\text{pad}}^m b_{\text{pad}}^m d_m^m 1_{h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m} \\
(19) + 2 a_{\text{pad}}^m b_{\text{pad}}^m d_m^m 1_{h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m} \\
\vdots \\
(19) + (l - 1) a_{\text{pad}}^m b_{\text{pad}}^m d_m^m 1_{h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m}
\end{bmatrix},
\]
where

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

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:

\[ (i - 1)a^m_{\text{pad}} b^m_{\text{pad}} d^m + 1, \ldots, ia^m_{\text{pad}} b^m_{\text{pad}} d^m. \]
Evaluation of \((v^i)^T P^m_\phi\) VII

- (22) can be easily obtained by the following outer product

\[
\text{vec}(\left(19\right) + \begin{bmatrix} 0 & \ldots & l - 1 \end{bmatrix} a^m_{\text{pad}} b^m_{\text{pad}} d^m)
\]

- To obtain

\[
\begin{bmatrix}
v^1 \\
\vdots \\
v^l
\end{bmatrix}
\]

we note that it is the same as

\[
\text{vec} \left( (W^m)^T \begin{bmatrix} \frac{\partial \xi_1}{\partial S^{m,1}} & \ldots & \frac{\partial \xi_l}{\partial S^{m,l}} \end{bmatrix} \right) .
\]
Thus we do a matrix-matrix multiplication.

From (24), we can see why $\frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T}$ over a batch of instances are stored in the form of

$$\left[ \frac{\partial \xi_1}{\partial S^{m,1}} \ldots \frac{\partial \xi_l}{\partial S^{m,l}} \right] \in \mathbb{R}^{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m l}.$$
A Simple Code I

```plaintext
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

\[ V = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_l \end{bmatrix} \]

\text{num}_v \text{ is the number of columns}
Outline

1. Introduction
2. Storage
3. Generation of $\phi(\text{pad}(Z^{m,i}))$
4. Evaluation of $(v^i)^T P^m_\phi$
5. Computational Complexity
6. Discussion
To see where the computational bottleneck is, it’s important to check the complexity of major operations.

Forward:

\[
W^m \text{mat}(P^m_\phi P^m_{\text{pad}} \text{vec}(Z^{m,i})) = W^m \phi(pad(Z^{m,i}))
\]

\[
\phi(pad(Z^{m,i})): \mathcal{O}(l \times h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m)
\]

\[
W^m \phi(\cdot): \mathcal{O}(l \times h^m h^m d^m d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m)
\]

\[
Z^{m+1,i} = \text{mat}(P^{m,i}_\text{pool} \text{vec}(\sigma(S^{m,i})))
\]
Backward:

\[ \Delta \leftarrow \max(\text{vec}(\Delta)^T P_{\text{pool}}^m) \]

\[ \mathcal{O}(l \times h^m h^m d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m) \]

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

\[ \mathcal{O}(l \times h^m h^m d^m d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m) \]

\[ \Delta \leftarrow \text{vec}\left((W^m)^T \Delta\right)^T P_{\phi}^m P_{\text{pad}}^m \]
We see that matrix-matrix products are the bottleneck.

If so, why check others?

The issue is that matrix-matrix products may be better optimized.

You will get first-hand experiences in doing projects.
Outline

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

- 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
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?