HW2 Recitation

10-417/617

Statistics and Probability

Conditional Probability

$$P(A|B)P(B) = P(A,B)$$

Bayes' Theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

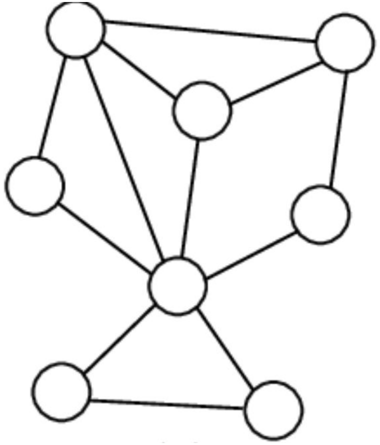
Marginalization

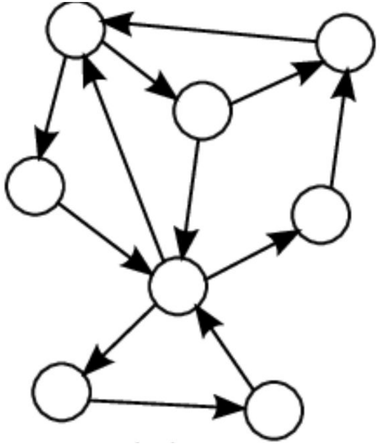
$$P(A) = \int_B P(A, B)$$

$$P(A) = \sum_B P(A, B)$$

Graphical Models

- joint p(x, y) or conditional p(y | x) probability distribution
- represented as G=(V,E)
- Enables us to encode relationships between a set of random variables
- There are two types of graphical models : Directed Graphical Model and Undirected Graphical Model
- Directed edges gives a causality relationships
- Undirected edges give correlations between variable

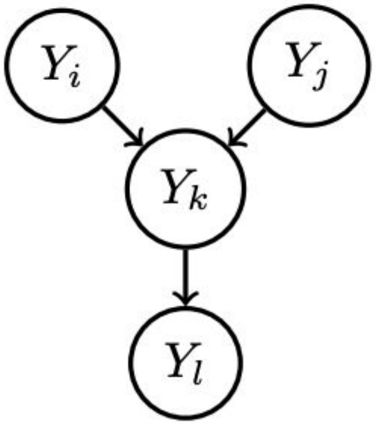




Direct Graphical Model

- Assume a directed, acyclic graphical model G=(V,E) and E ⊂ VxV

$$p(\mathbf{Y} = \mathbf{y}) = \prod_{i \in \mathcal{V}} p(y_i \mid \mathbf{y}_{\mathsf{pa}_G(i)})$$



where y_{paG}(i) is conditional probability distribution on the parents of node i

Undirected Graphical Model

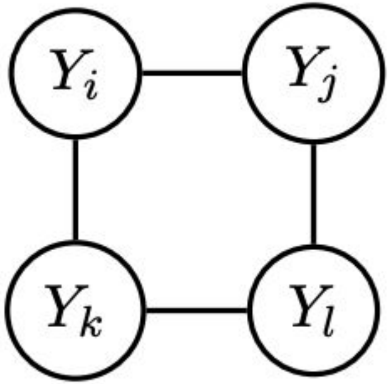
- An undirected graphical model G=(V,E) is called Markov Random Field (MRF) if two nodes are conditionally independent whenever they are not connected.

$$p(Y_i \mid Y_{\mathcal{V}\setminus\{i\}}) = p(Y_i \mid Y_{N(i)})$$

where N(i) is the neighbor of node i

$$Y_i \perp\!\!\!\perp Y_{\mathcal{V}\setminus \mathsf{cl}(i)} \mid Y_{N(i)} \ ,$$

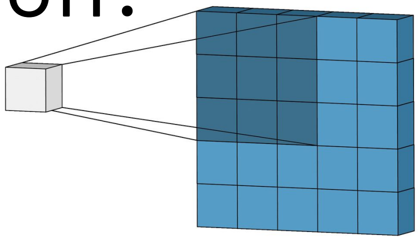
where cl(i)=N(i) U {i} is the closed neighborhood of i.



Convolutions

Some figures adapted from Simon Lucey’s 16-720b slides and Introduction to Deep Learning 11-785

What is a (Discrete) Convolution?

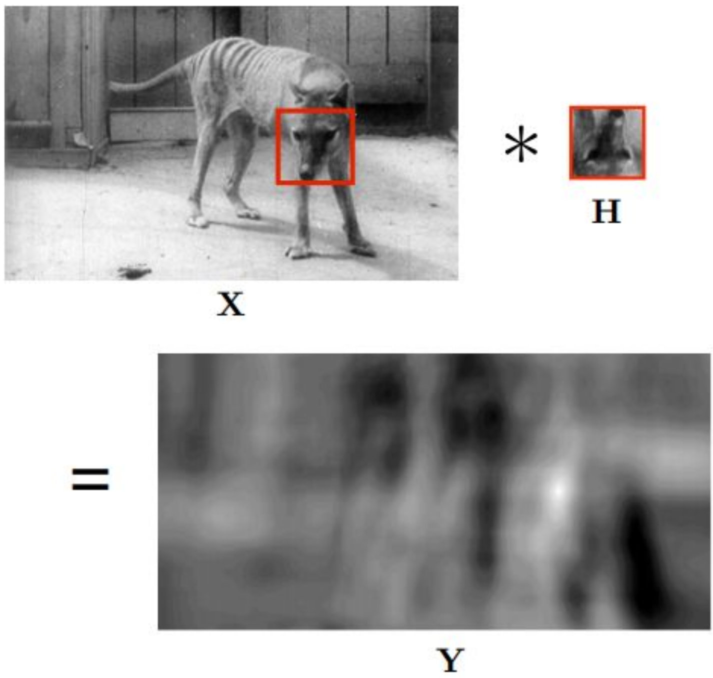


Concatenation of inner products of filter and receptive fields of signal.

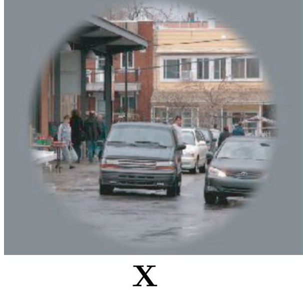
Unlike an arbitrary affine transform, convolutions preserve locality.

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

These would be + in cross-correlation



Why do Convolutions Work?

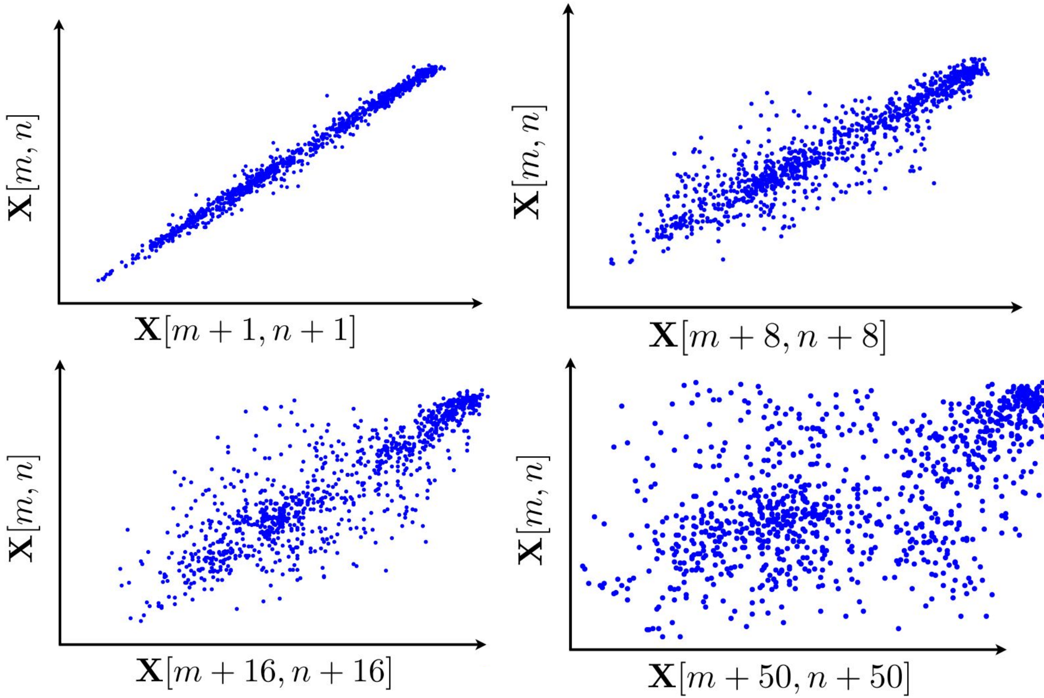


Natural signals are locally smooth!

In natural images, neighboring pixels tend to be highly correlated.

Convolutions exploit this local correlation to generalize better than fully connected networks.

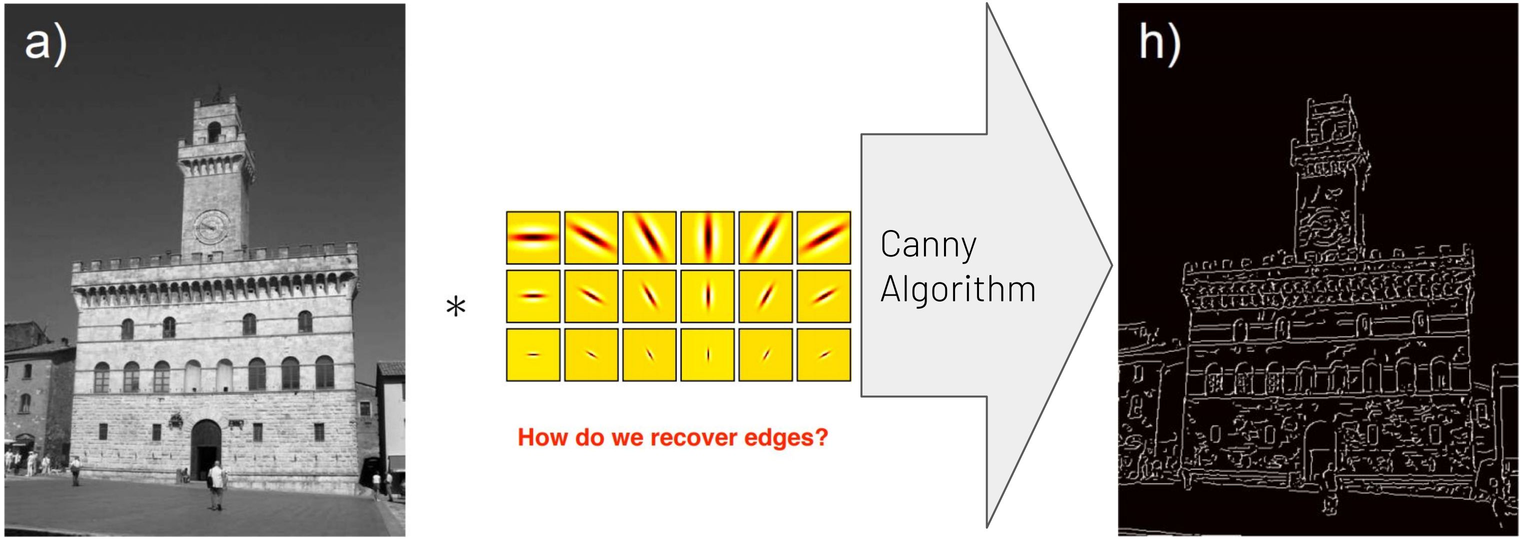
Convolutional layers DO NOT learn random noise better than fully connected layers!



Simoncelli & Olshausen 2001

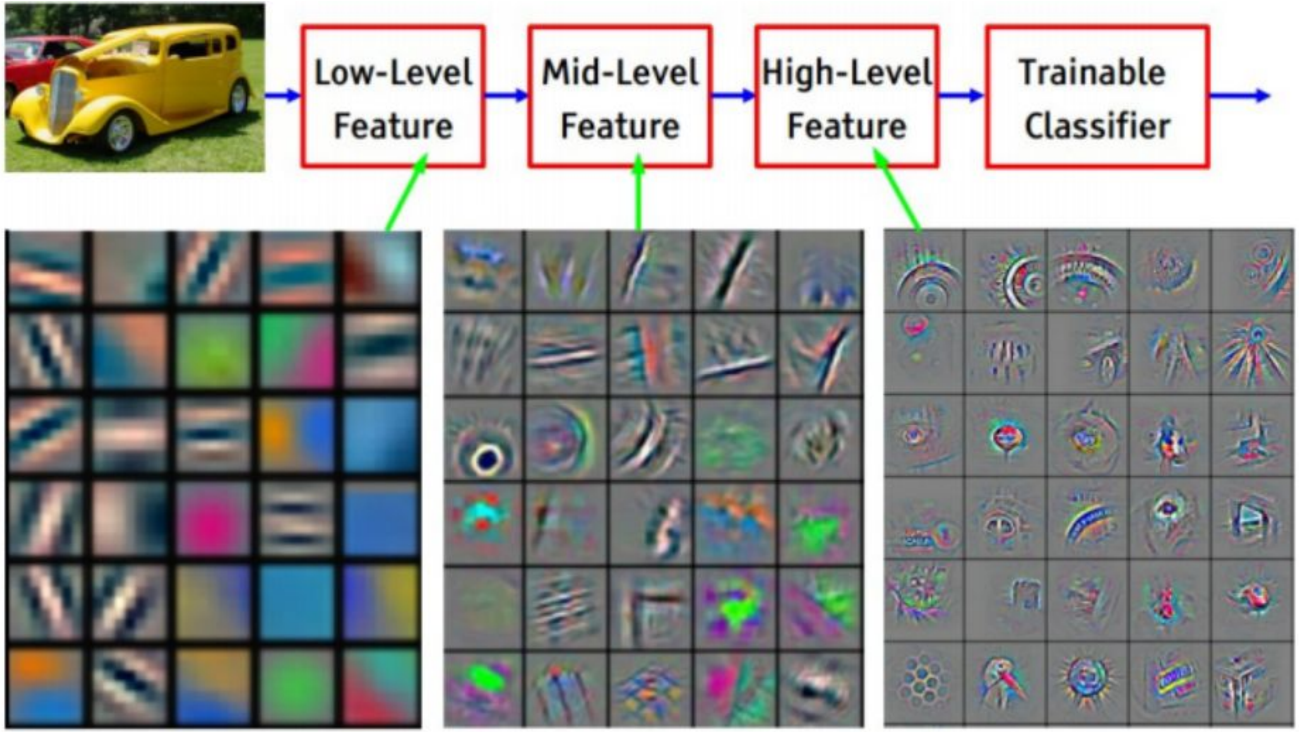
Computer Vision B.C. (Before Conv-Nets)

Filters were hand-designed to extract features for the intended task!



Computer Vision A.D. (After Deep Learning)

Now, optimal filters are *learned* through back-propagation!

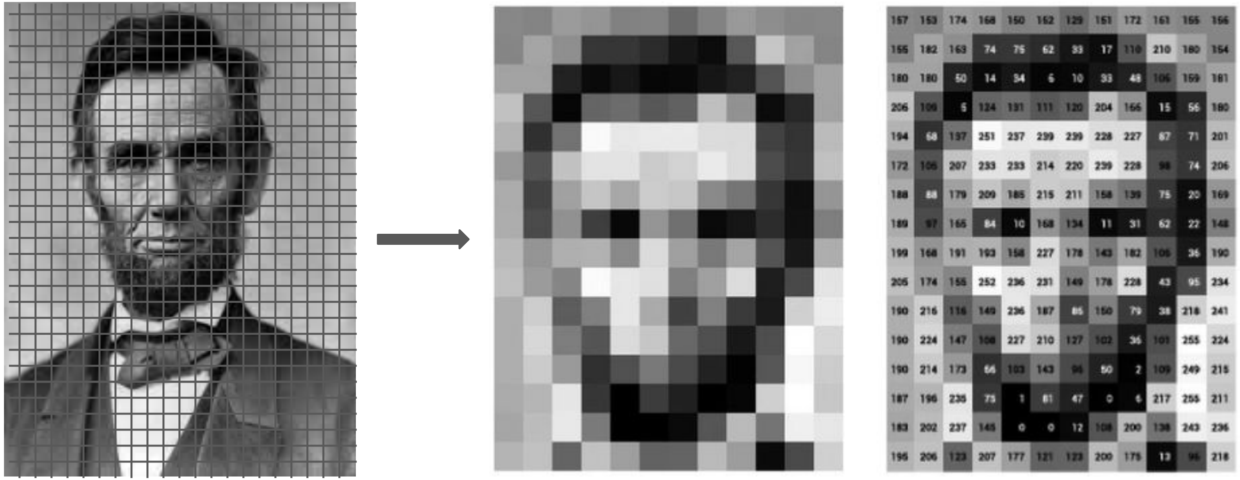


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

How do we do convolutions?

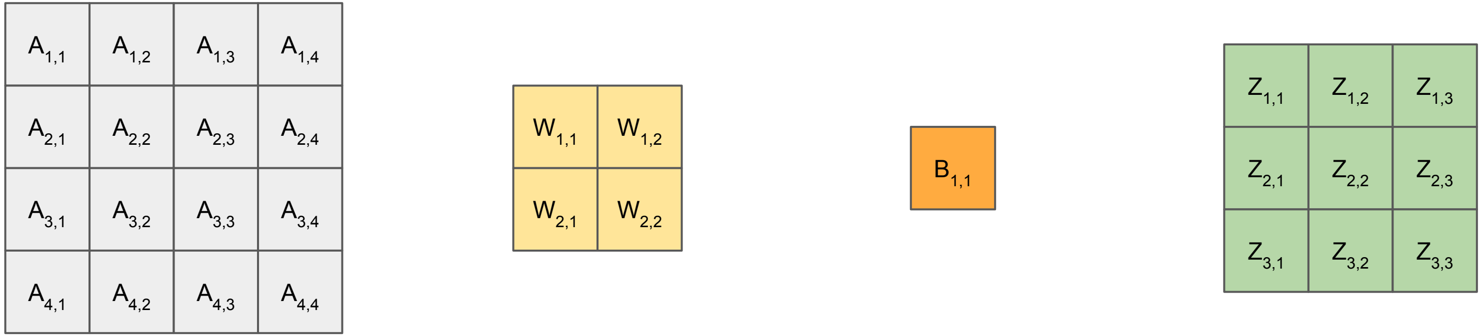
What is an image?

~~A thousand words.~~ A Matrix **`I`** of dimensions **`(M,N)`** with **`I[i][j] = intensity(pixel(i,j))`**



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Components of a CNN



Input - **A**

Kernel - **W**

Bias - **B**

Output - **Z**

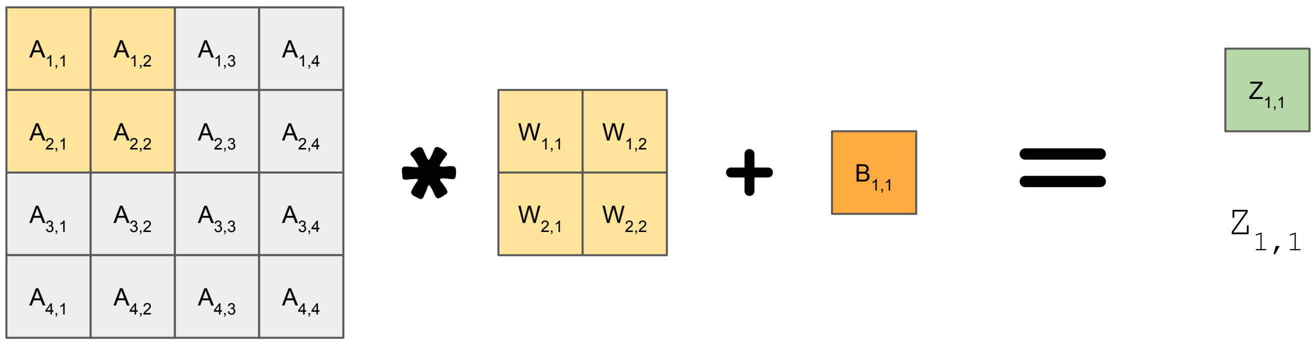
$$\mathbf{Z} = (\mathbf{A} \otimes \mathbf{W}) + \mathbf{B}$$

Convolution steps

Essentially element-wise (Hadamard) multiplications and summations

Convolution steps

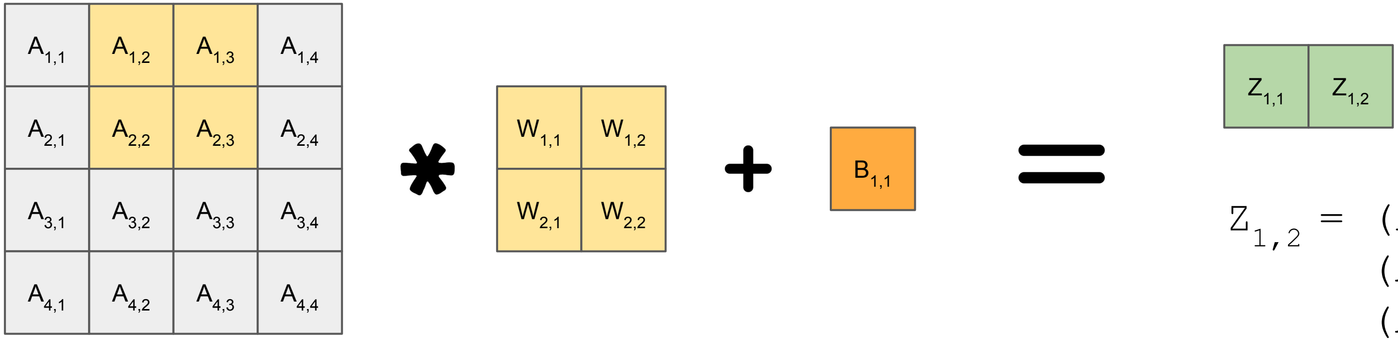
Essentially element-wise (Hadamard) multiplications and summations



$$\begin{aligned} Z_{1,1} = \; & (A_{1,1} * W_{1,1}) \; + \\ & (A_{1,2} * W_{1,2}) \; + \\ & (A_{2,1} * W_{2,1}) \; + \\ & (A_{2,2} * W_{2,2}) \; + \\ & B \end{aligned}$$

Convolution steps

Essentially element-wise (Hadamard) multiplications and summations



$$
\begin{aligned}
Z_{1,2} = \; & (A_{1,2} * W_{1,1}) \; + \\
& (A_{1,3} * W_{1,2}) \; + \\
& (A_{2,2} * W_{2,1}) \; + \\
& (A_{2,3} * W_{2,2}) \; + \\
& B
\end{aligned}
$$

Convolution steps

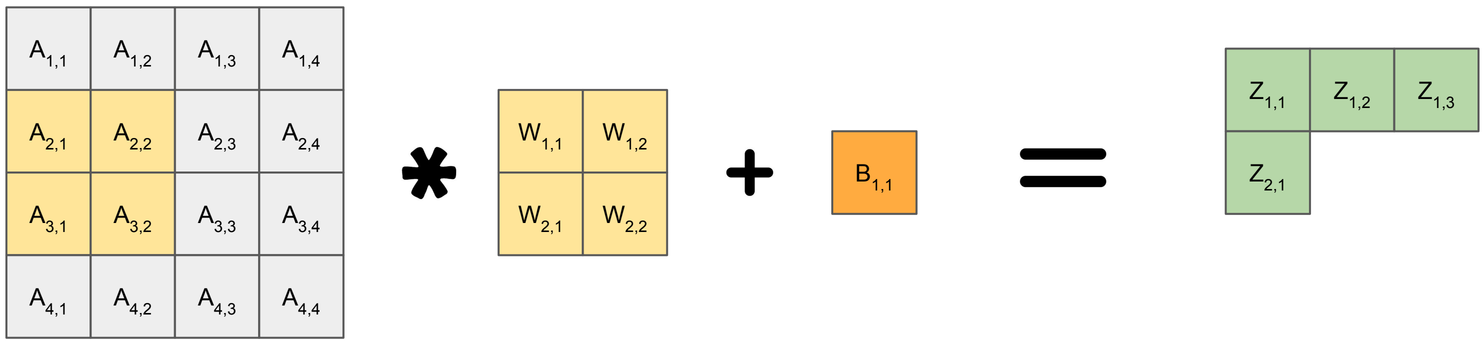
Essentially element-wise (Hadamard) multiplications and summations

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} * \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} \end{bmatrix} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \end{bmatrix}$$

$$\begin{aligned} Z_{1,3} = \; & (A_{1,3} * W_{1,1}) + \\ & (A_{1,4} * W_{1,2}) + \\ & (A_{2,3} * W_{2,1}) + \\ & (A_{2,4} * W_{2,2}) + \\ & B \end{aligned}$$

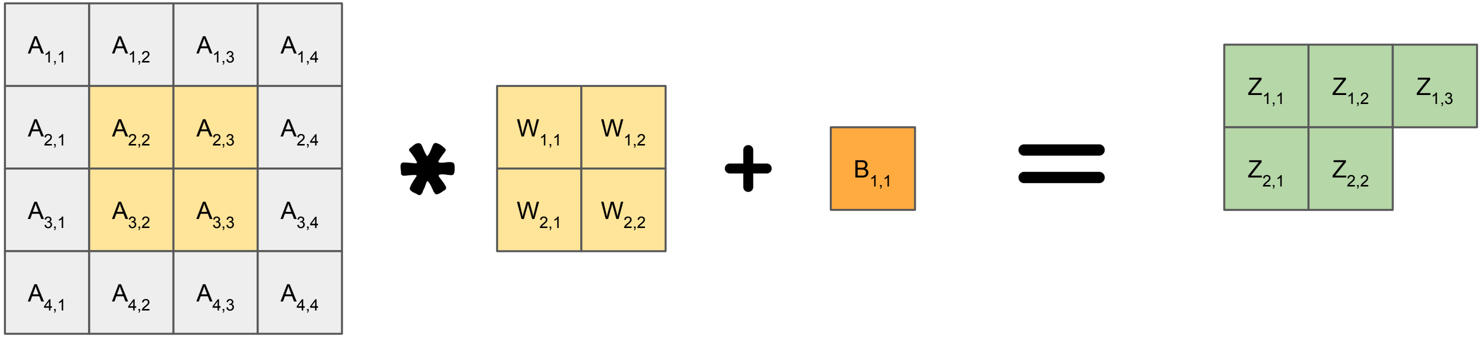
Convolution steps

Essentially element-wise (Hadamard) multiplications and summations



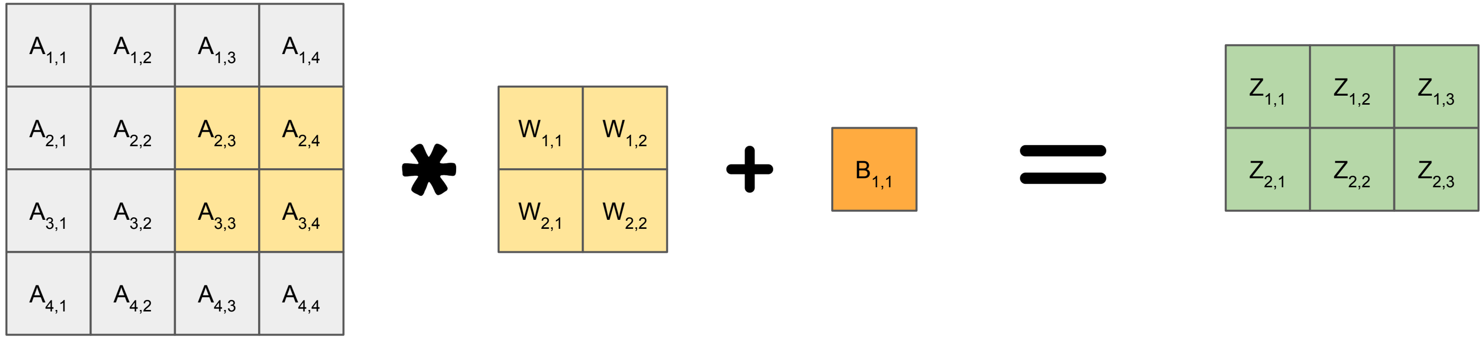
Convolution steps

Essentially element-wise (Hadamard) multiplications and summations



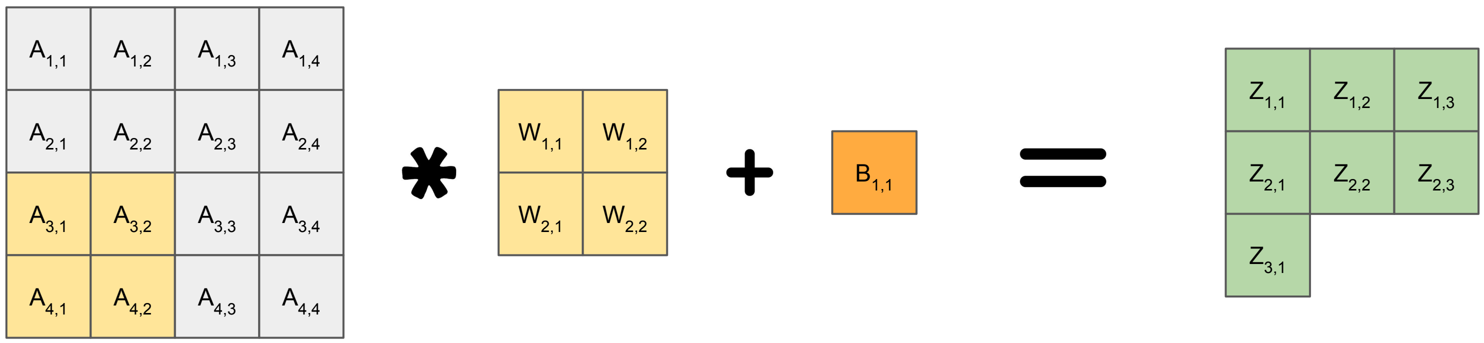
Convolution steps

Essentially element-wise (Hadamard) multiplications and summations



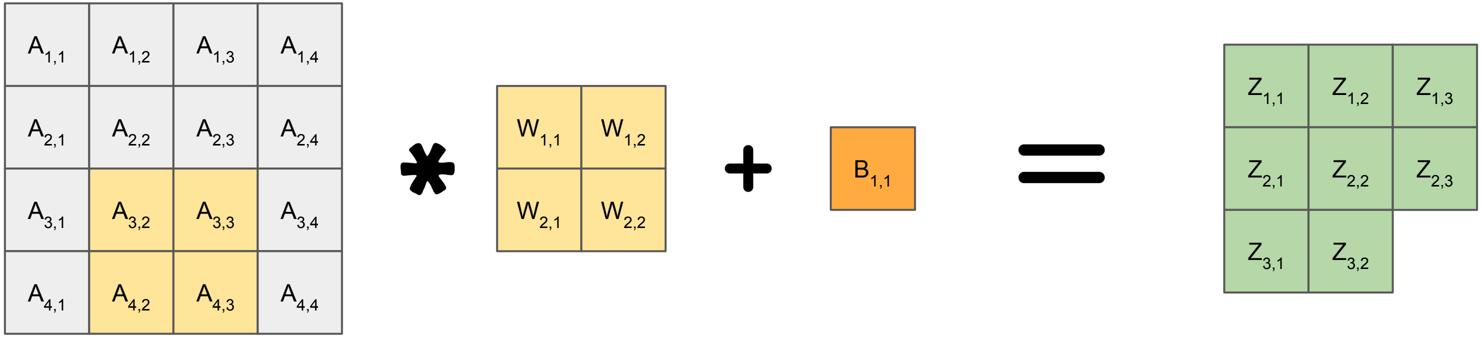
Convolution steps

Essentially element-wise (Hadamard) multiplications and summations



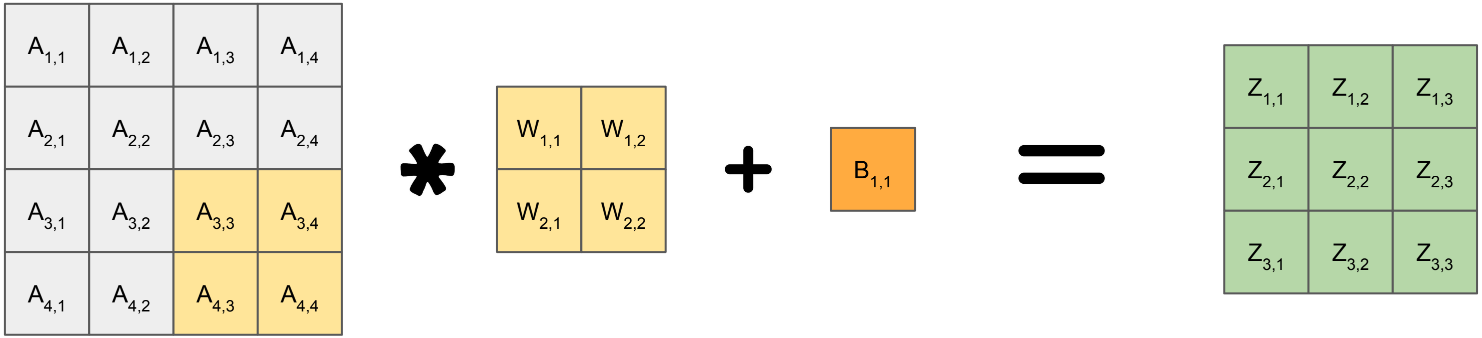
Convolution steps

Essentially element-wise (Hadamard) multiplications and summations

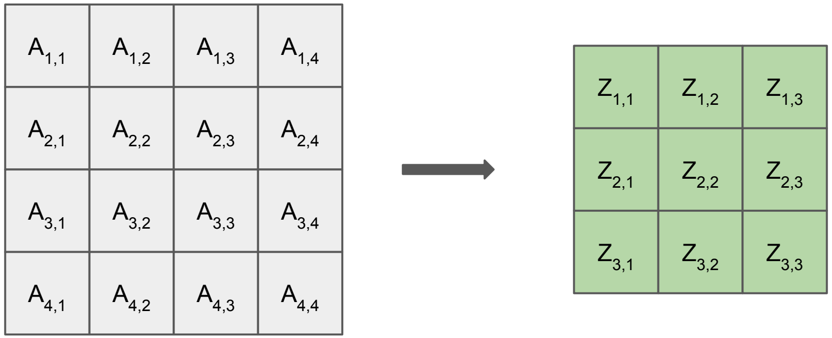


Convolution steps

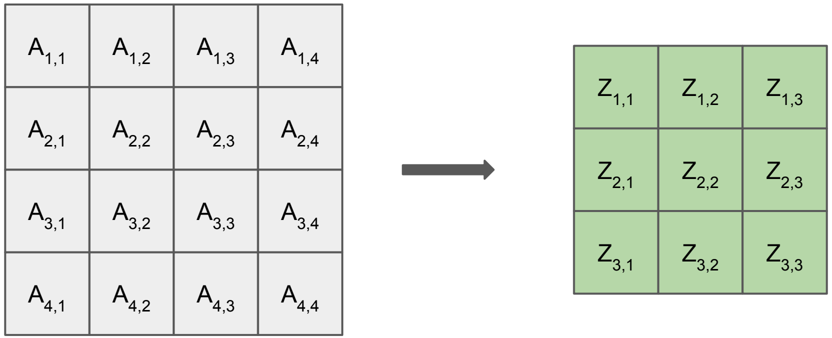
Essentially element-wise (Hadamard) multiplications and summations



Output Size


$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,4}$
$A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{2,4}$
$A_{3,1}$ $A_{3,2}$ $A_{3,3}$ $A_{3,4}$
$A_{4,1}$ $A_{4,2}$ $A_{4,3}$ $A_{4,4}$
$Z_{1,1}$ $Z_{1,2}$ $Z_{1,3}$
$Z_{2,1}$ $Z_{2,2}$ $Z_{2,3}$
$Z_{3,1}$ $Z_{3,2}$ $Z_{3,3}$

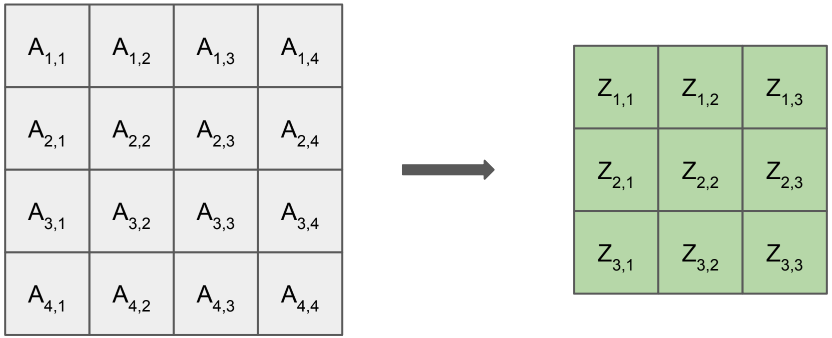
Output Size



Output Width = $[(W_{in} - W_k + 2P) \mathbin{//} (S)] + 1$

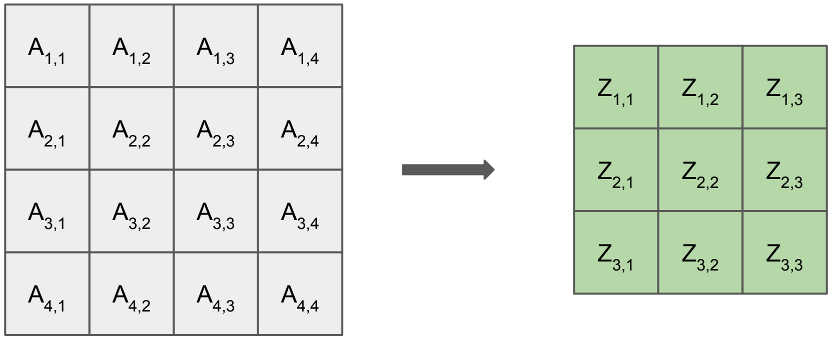
Output Height = $[(H_{in} - H_k + 2P) \mathbin{//} (S)] + 1$

Output Size



Output Width = $[(W_{in} - W_k + 2\mathbf{P}) // (\mathbf{S})] + 1$

Output Size



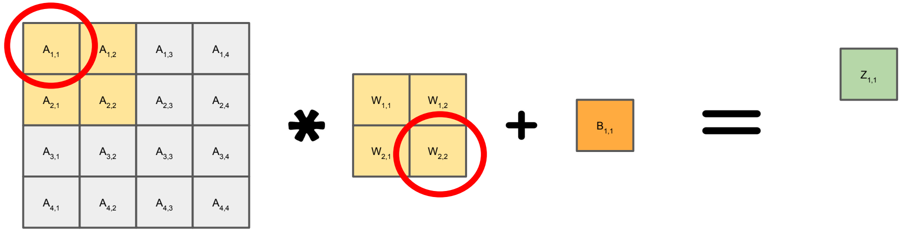
Output Width = $[(W_{in} - W_k + 2\mathbf{P}) \ // \ (\mathbf{S})] + 1$

P: Padding (here - 0)
S: Stride (here - 1)

Padding

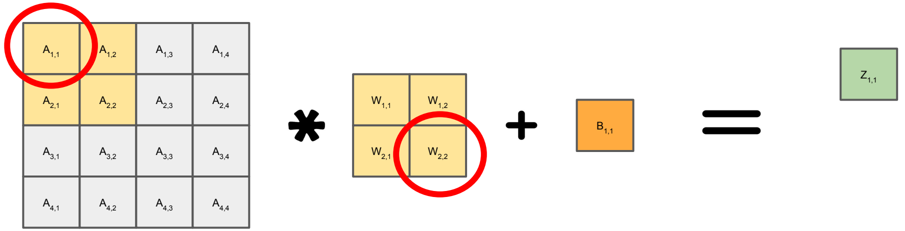
- Attaching zeros (usually) around inputs.
- Images can be padded to the left, right, top, and bottom.

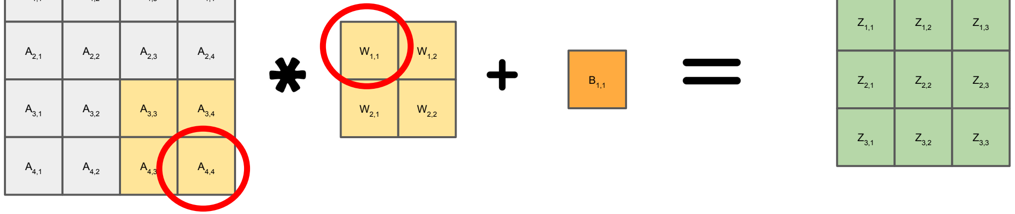
Padding



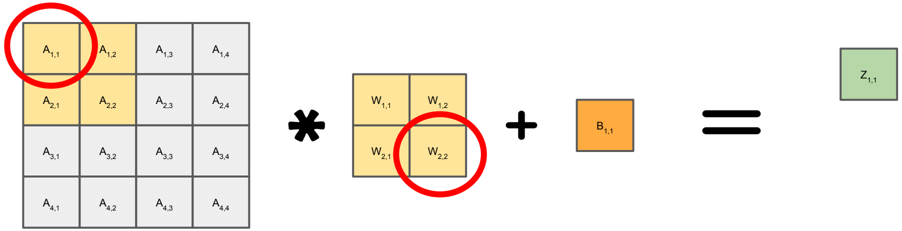
$$
\begin{aligned}
Z_{1,1} = \; & (A_{1,1} * W_{1,1}) \; + \\
& (A_{1,2} * W_{1,2}) \; + \\
& (A_{2,1} * W_{2,1}) \; + \\
& (A_{2,2} * W_{2,2}) \; + \\
& B
\end{aligned}
$$

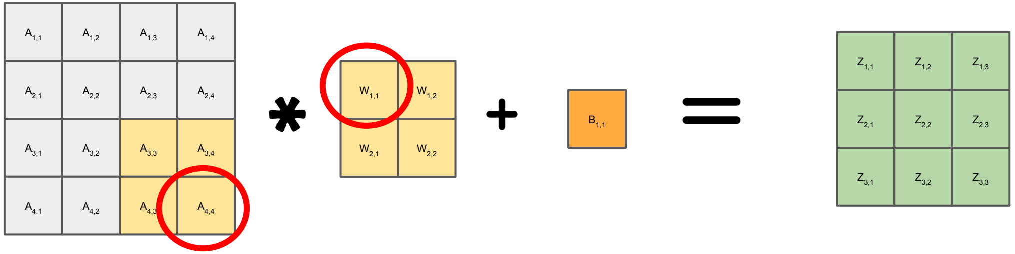
Padding


A1,1 A1,2 A1,3 A1,4
A2,1 A2,2 A2,3 A2,4
A3,1 A3,2 A3,3 A3,4
A4,1 A4,2 A4,3 A4,4
W1,1 W1,2
W2,1 W2,2
B1,1
Z1,1


A1,1 A1,2 A1,3 A1,4
A2,1 A2,2 A2,3 A2,4
A3,1 A3,2 A3,3 A3,4
A4,1 A4,2 A4,3 A4,4
W1,1 W1,2
W2,1 W2,2
B1,1
Z1,1 Z1,2 Z1,3
Z2,1 Z2,2 Z2,3
Z3,1 Z3,2 Z3,3

Padding





Never Meet…

Padding

Increase output size

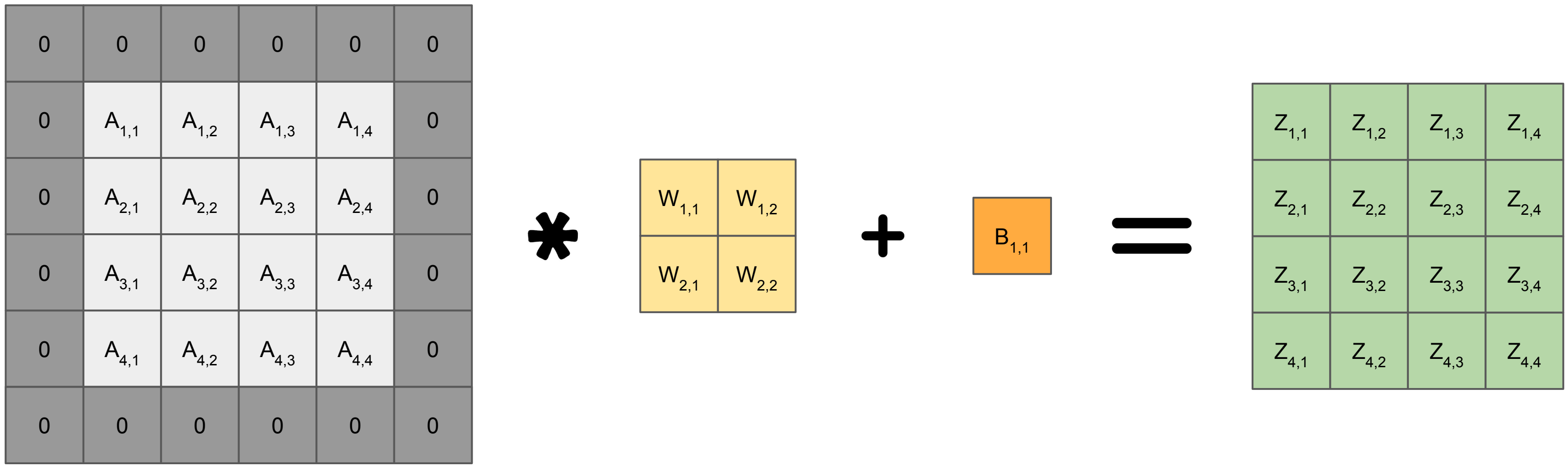
Preserve input size

More Kernel Interactions!

Padding

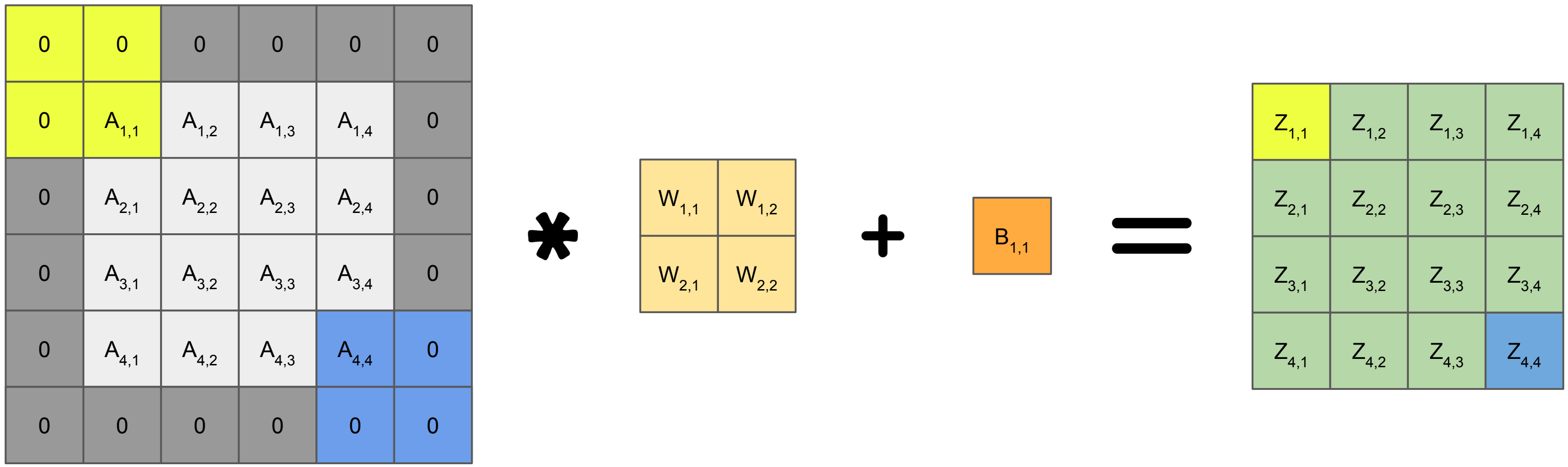
$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} * \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} \end{bmatrix} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} \end{bmatrix}$$

Padding



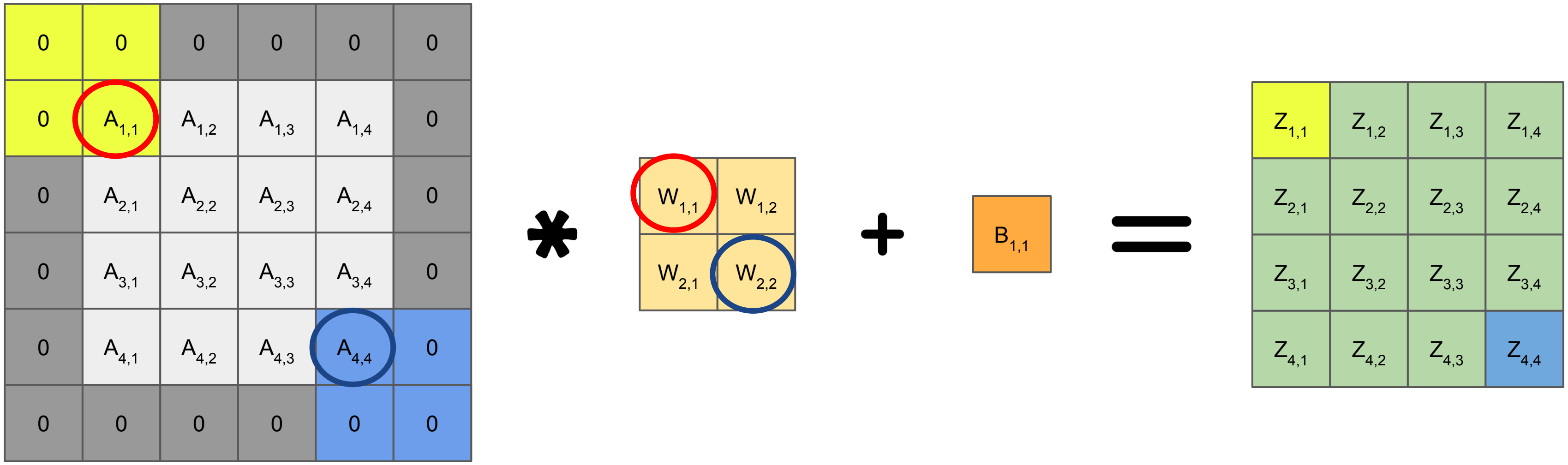
0 0 0 0 0 0
0 A1,1 A1,2 A1,3 A1,4 0
0 A2,1 A2,2 A2,3 A2,4 0
0 A3,1 A3,2 A3,3 A3,4 0
0 A4,1 A4,2 A4,3 A4,4 0
0 0 0 0 0 0
W1,1 W1,2
W2,1 W2,2
B1,1
Z1,1 Z1,2 Z1,3 Z1,4
Z2,1 Z2,2 Z2,3 Z2,4
Z3,1 Z3,2 Z3,3 Z3,4
Z4,1 Z4,2 Z4,3 Z4,4

Padding



0 0 0 0 0 0
0 A1,1 A1,2 A1,3 A1,4 0
0 A2,1 A2,2 A2,3 A2,4 0
0 A3,1 A3,2 A3,3 A3,4 0
0 A4,1 A4,2 A4,3 A4,4 0
0 0 0 0 0 0
W1,1 W1,2
W2,1 W2,2
B1,1
Z1,1 Z1,2 Z1,3 Z1,4
Z2,1 Z2,2 Z2,3 Z2,4
Z3,1 Z3,2 Z3,3 Z3,4
Z4,1 Z4,2 Z4,3 Z4,4

Padding



0 0 0 0 0 0
0 A1,1 A1,2 A1,3 A1,4 0
0 A2,1 A2,2 A2,3 A2,4 0
0 A3,1 A3,2 A3,3 A3,4 0
0 A4,1 A4,2 A4,3 A4,4 0
0 0 0 0 0 0
*
W1,1 W1,2
W2,1 W2,2
+
B1,1
=
Z1,1 Z1,2 Z1,3 Z1,4
Z2,1 Z2,2 Z2,3 Z2,4
Z3,1 Z3,2 Z3,3 Z3,4
Z4,1 Z4,2 Z4,3 Z4,4

Stride

Taking bigger steps!

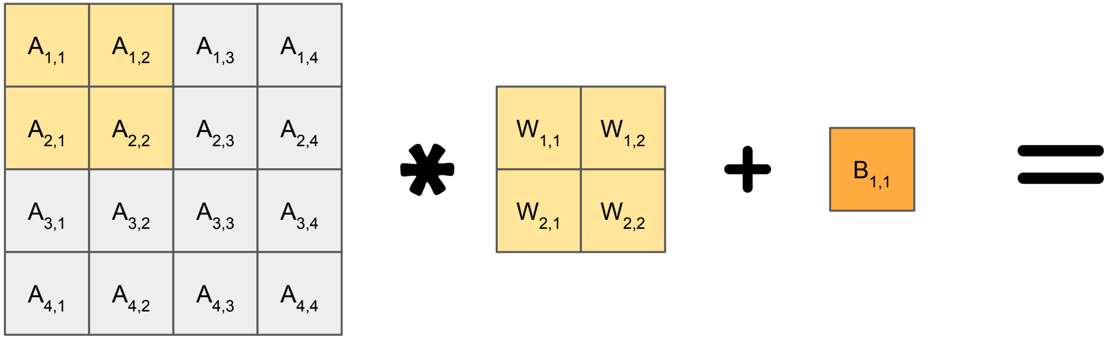
Stride = 1

What we did before - The kernel "moves" one pixel (or element) at a time.

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} * \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} \end{bmatrix} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} \end{bmatrix}$$

Stride = 2

Start at the same place



$$
\begin{aligned}
Z_{1,1} = \; & (A_{1,1} * W_{1,1}) \; + \\
& (A_{1,2} * W_{1,2}) \; + \\
& (A_{2,1} * W_{2,1}) \; + \\
& (A_{2,2} * W_{2,2}) \; + \\
& B
\end{aligned}
$$

Stride = 2

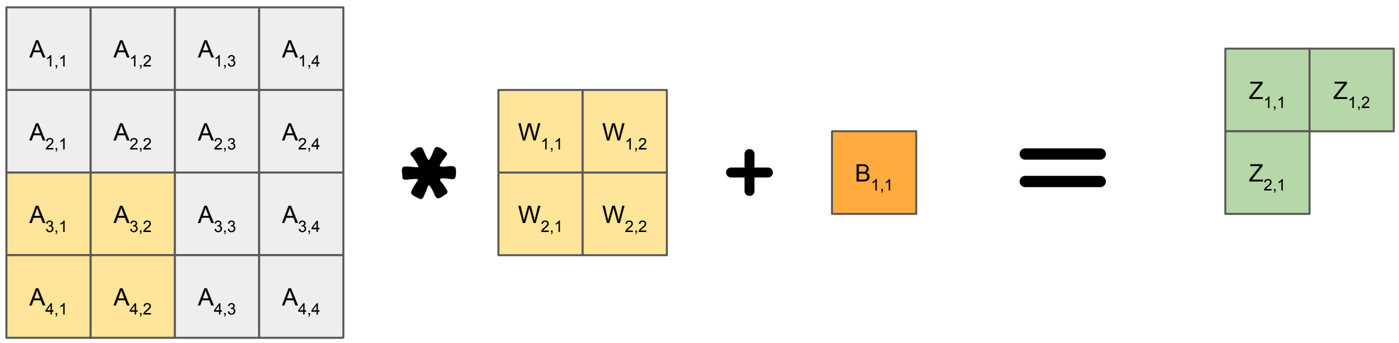
Move two elements to the right

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} * \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} \end{bmatrix} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \end{bmatrix}$$

$$\begin{aligned} Z_{1,2} = \ & (A_{1,3}*W_{1,1}) \ + \\ & (A_{1,4}*W_{1,2}) \ + \\ & (A_{2,3}*W_{2,1}) \ + \\ & (A_{2,4}*W_{2,2}) \ + \\ & B \end{aligned}$$

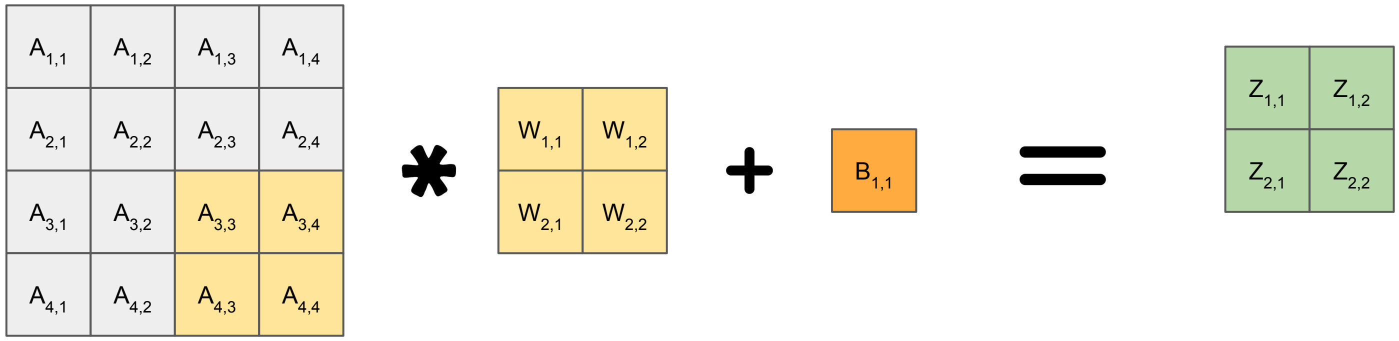
Stride = 2

Move two elements down.

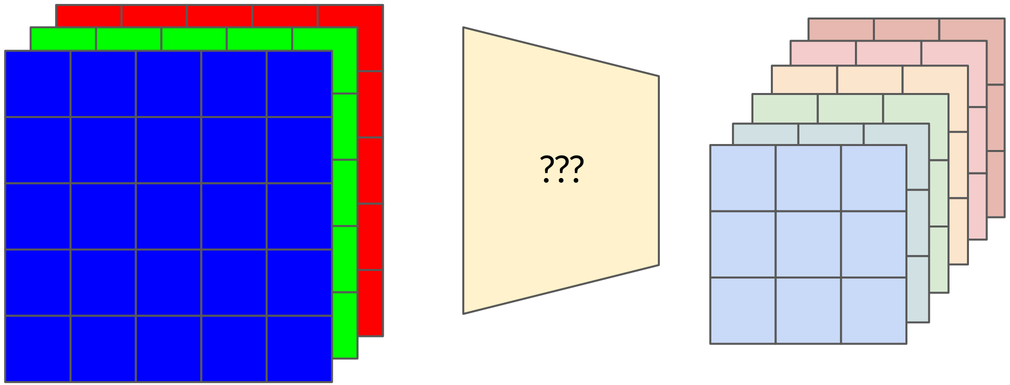


Stride = 2

Move two elements to the right.



Multi-channel CNN



Multi-channel CNN

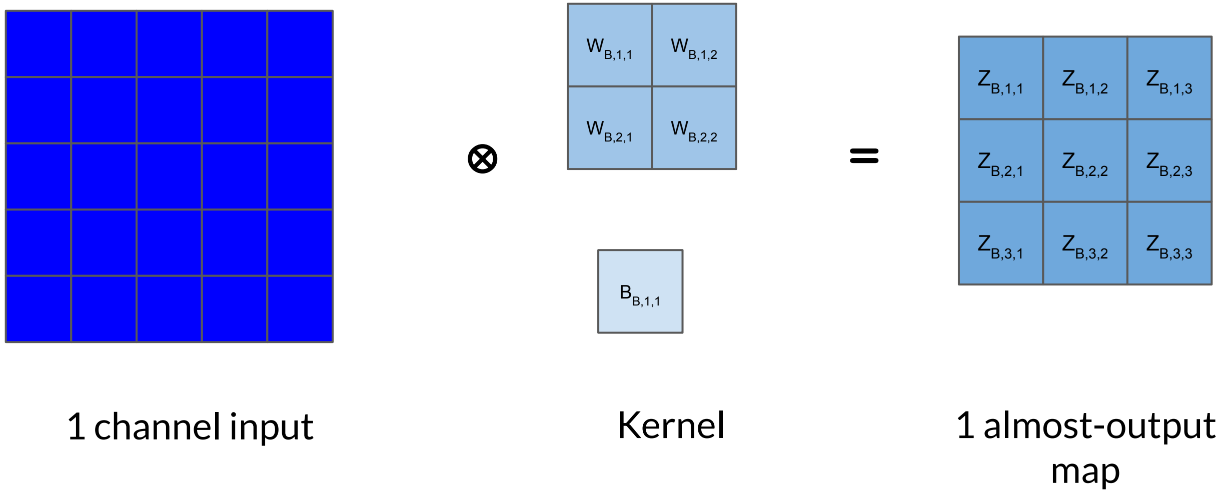
- Each kernel (or **filter**) has as many channels as the input does
- Channel `c` **of the kernel** convolves with channel `c` (corresponding) **of the input**.
- The number of output channels from the convolution = number of **filters**

C_{in} = Input channels

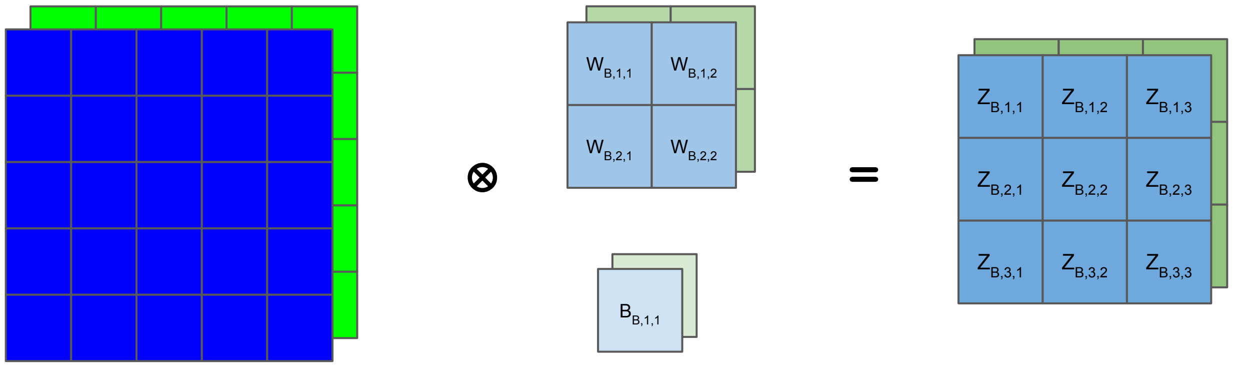
C_{kernel} = Kernel channels = C_{in}

K = Number of Kernels = C_{out} = Number of output channels

1 Filter with 3-channel input



1 Filter with 3-channel input

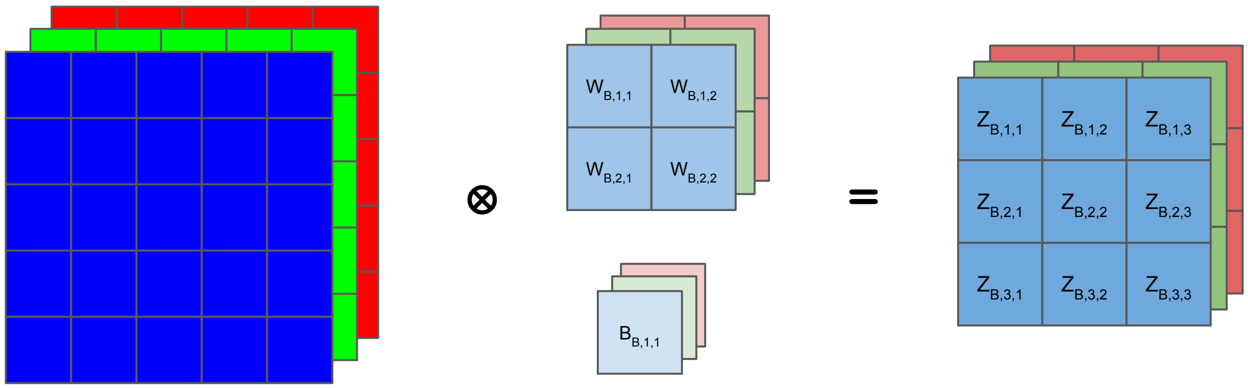


2 channel input

Kernel

2 almost-output maps

1 Filter with 3-channel input

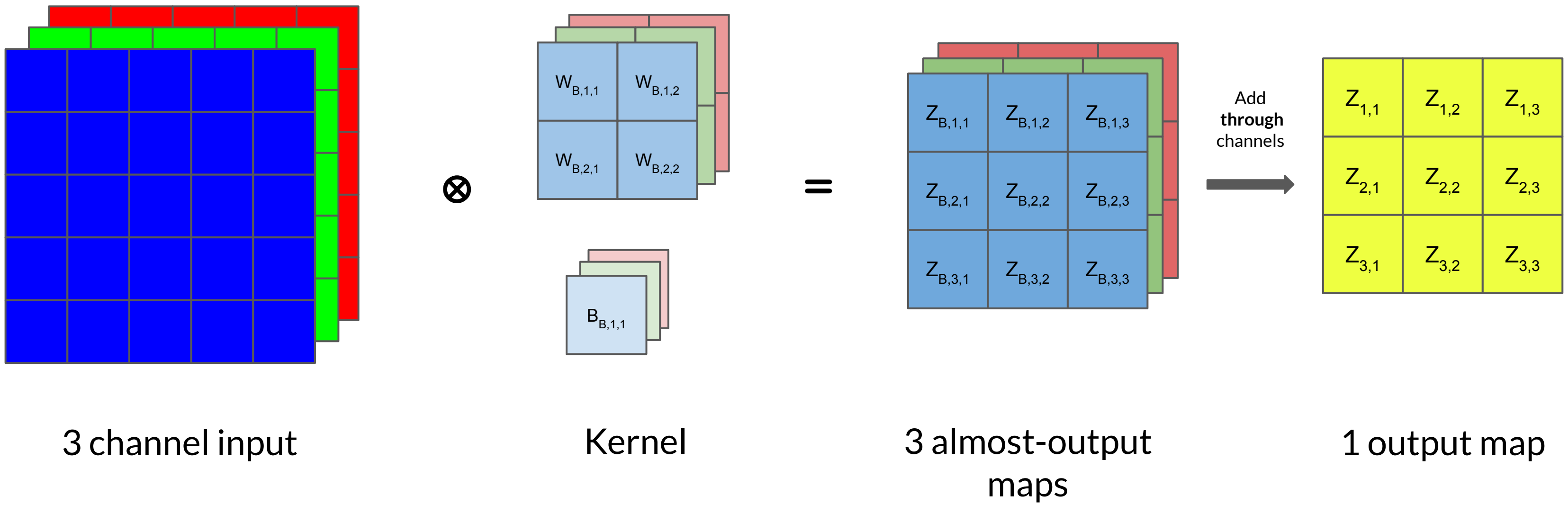


3 channel input

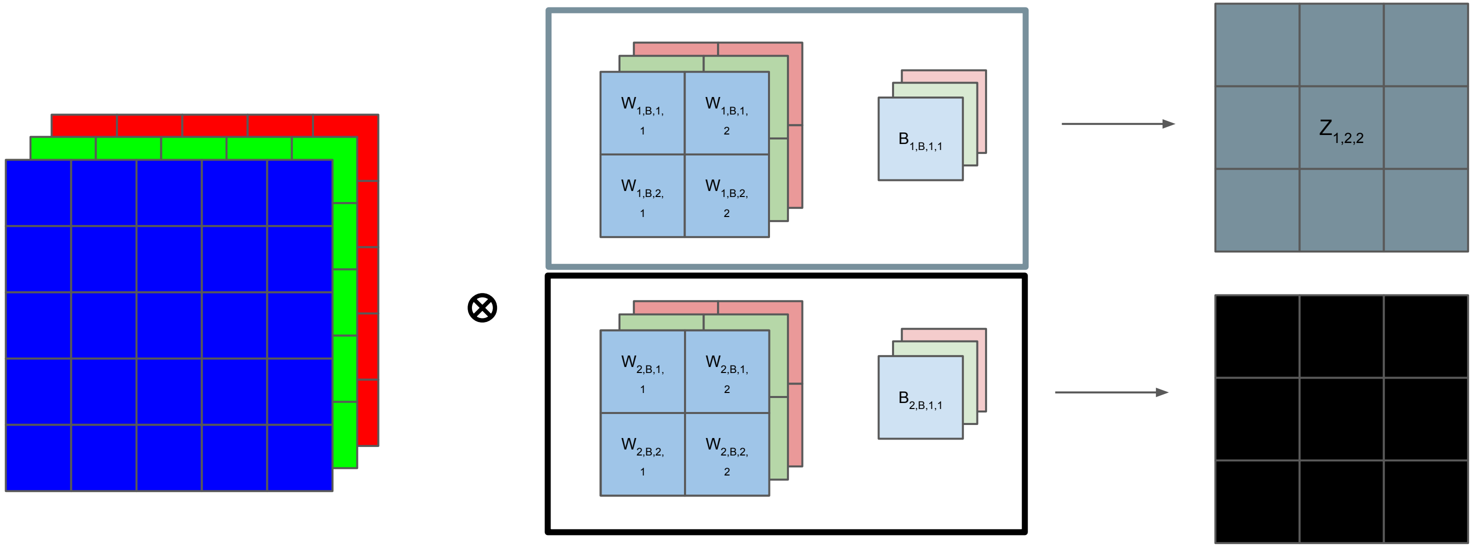
Kernel

3 almost-output maps

1 Filter with 3-channel input



2 Filters with 3-channel input

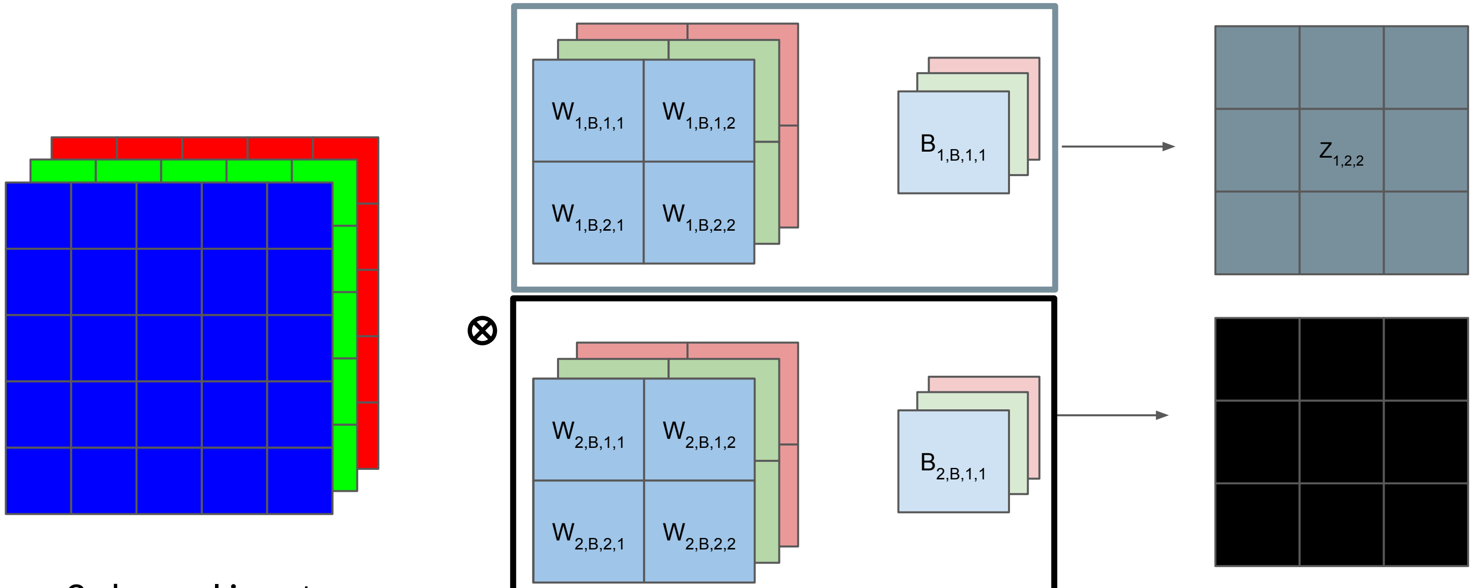


3 channel input

2 Kernels

2 output maps

2 Filters with 3-channel input

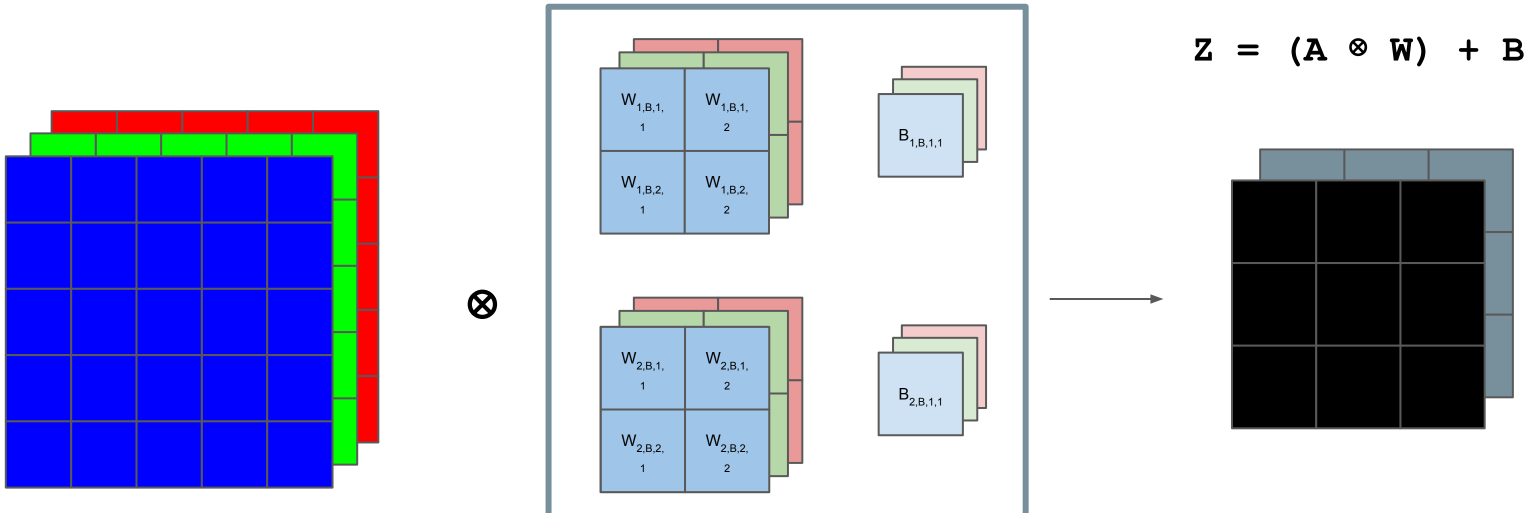


3 channel input

2 Kernels -
4 Dimensional

2 output maps

2 Filters with 3-channel input



3 channel input
(A)

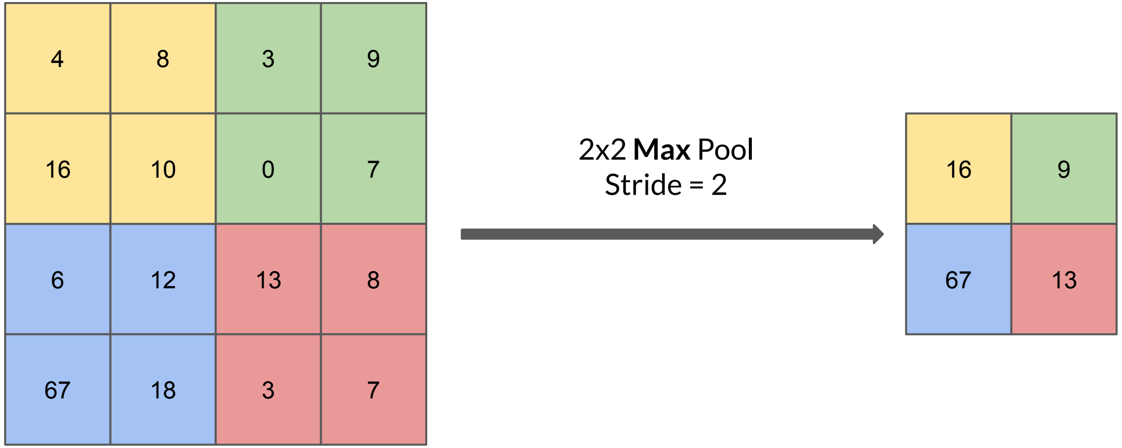
Kernels
(W, B)

2 channel Output
(Z)

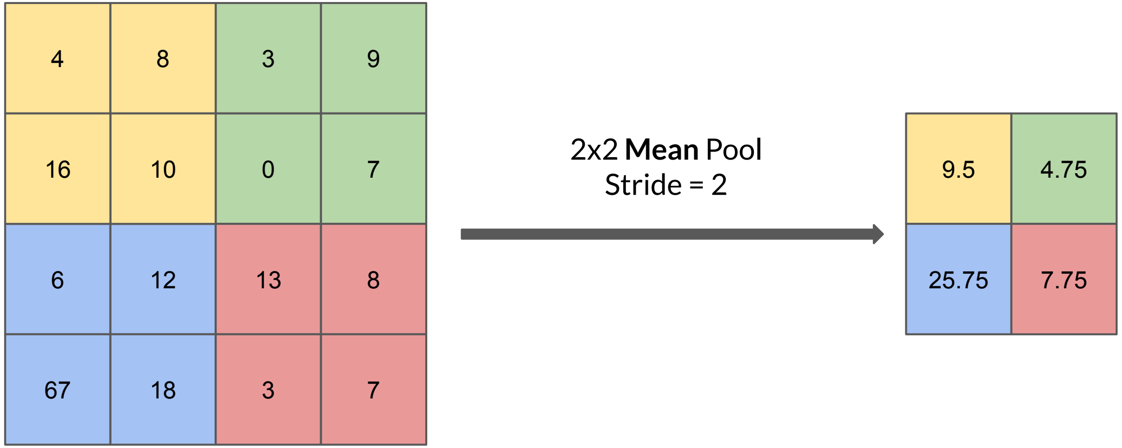
Pooling

- Usually follows convolutions
- Introduces Jitter Invariance
- Reduces feature-map size
- **Max**, **Mean**, Min

Pooling



Pooling



4
8
3
9
16
10
0
7
6
12
13
8
67
18
3
7
2x2 **Mean** Pool
Stride = 2
9.5
4.75
25.75
7.75

Convolutional Layer Implementation

- Earlier we said that a discrete convolution is a concatenation of inner products between the filter and receptive fields. This involves a lot of matrix multiplications!
- Parallelized matrix operations make this much faster than using a sliding filter.
    - `im2col`, `im2col_bw` are needed

THIS IS THE HARDEST PART OF THE PROGRAMMING. START EARLY!

Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

Step 1: Transform our input image into a matrix (im2col)

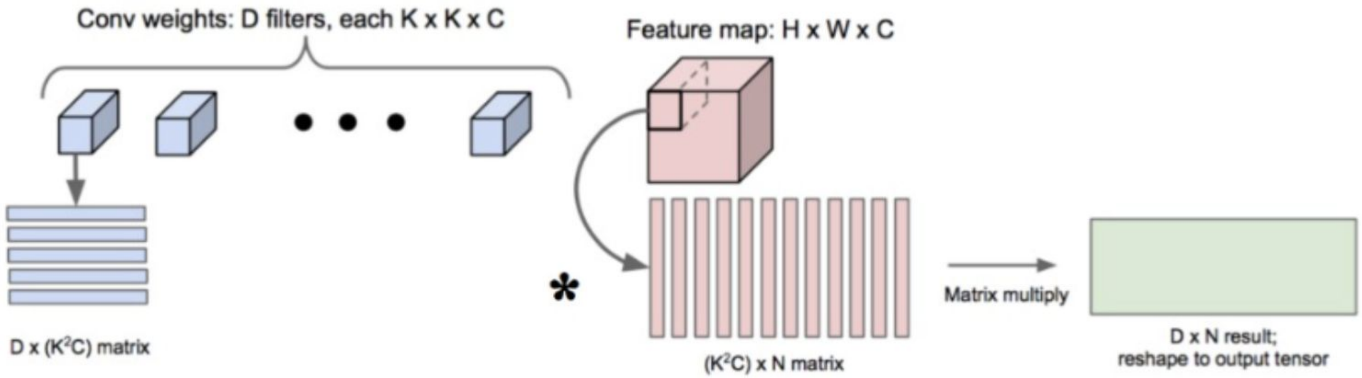
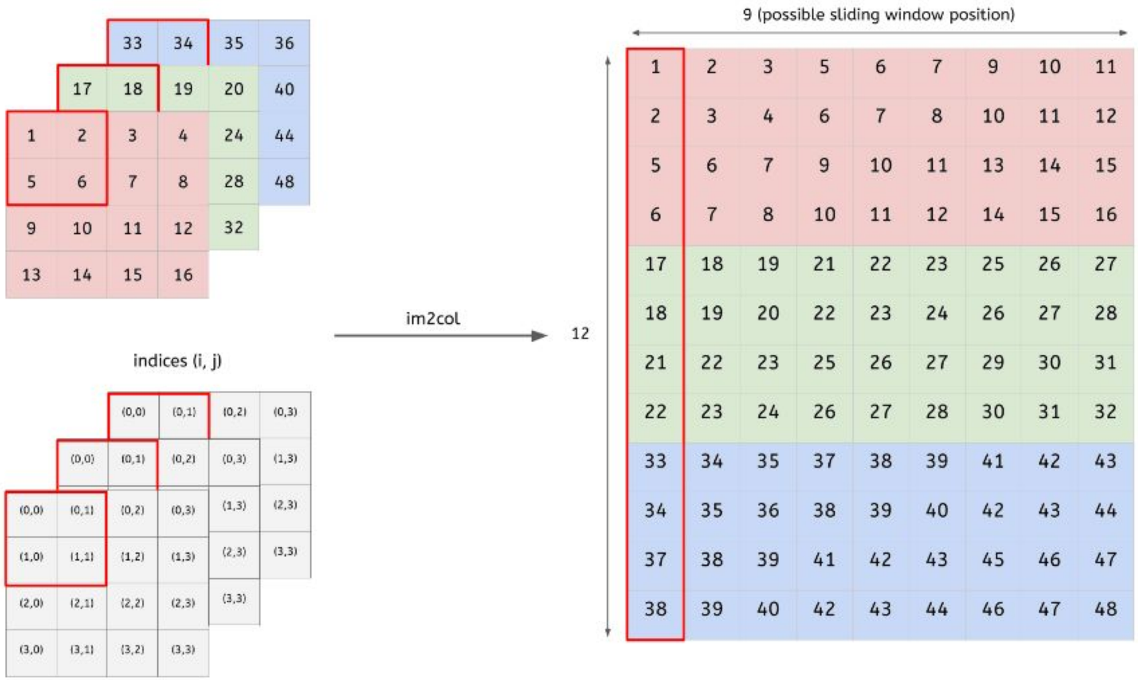


Figure 2: Illustration of im2col.

Example

We will perform a convolution between an (1,3,4,4) input image and kernels of shape (2,3,2,2)



Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

Step 2: Reshape our kernel (flatten)

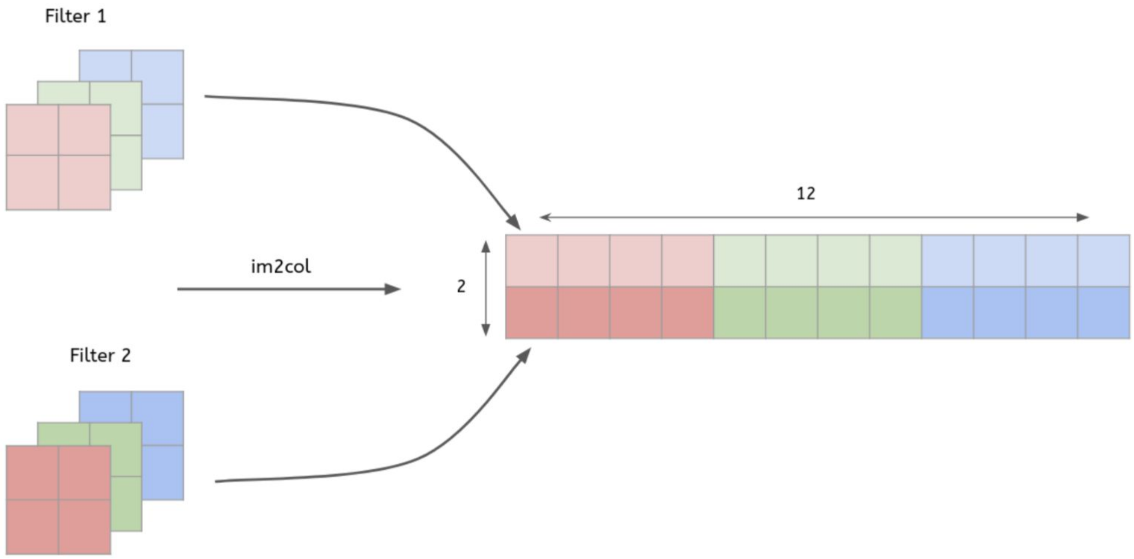


Figure 2: Reshaped version of the 2 kernels

As you can see, each filter is flattened and then stacked together. Thus, for X filter, we will flatten and stack X filters together.

Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

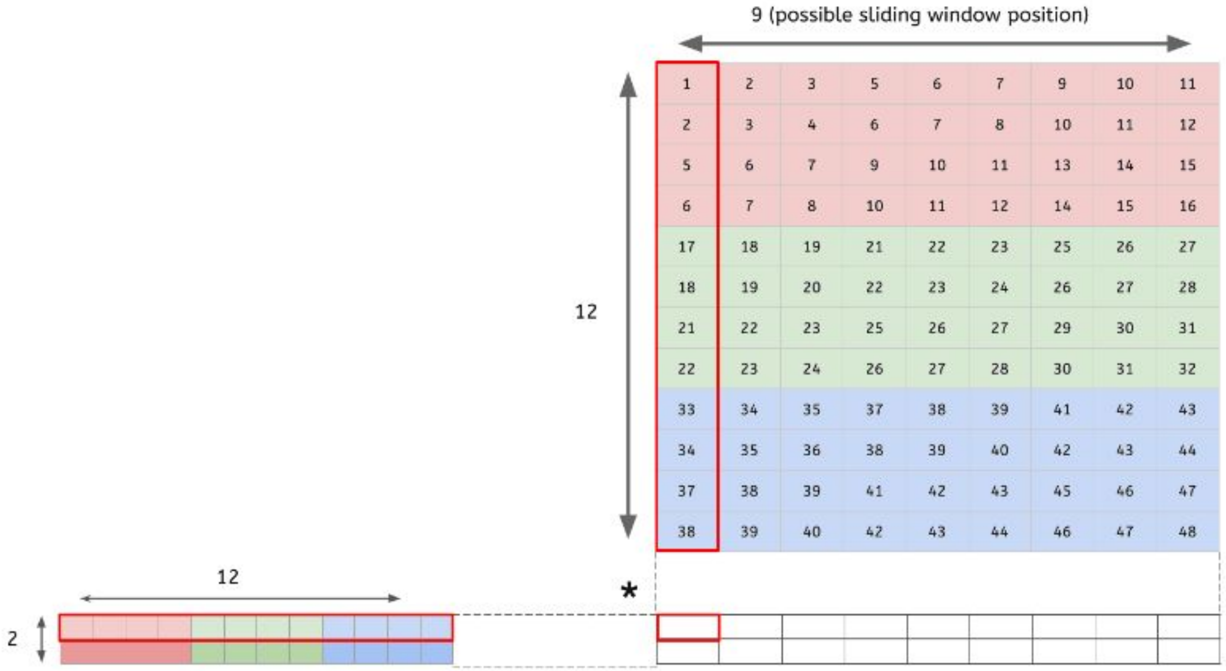
Forward propagation

Here are the following steps

1. Transform our input image into a matrix (im2col)
2. Reshape our kernel (flatten)
3. Perform matrix multiplication between reshaped input image and kernel

Step 3: Matrix multiplication between reshaped input and kernel

Now, we only need to perform a matrix multiplication.

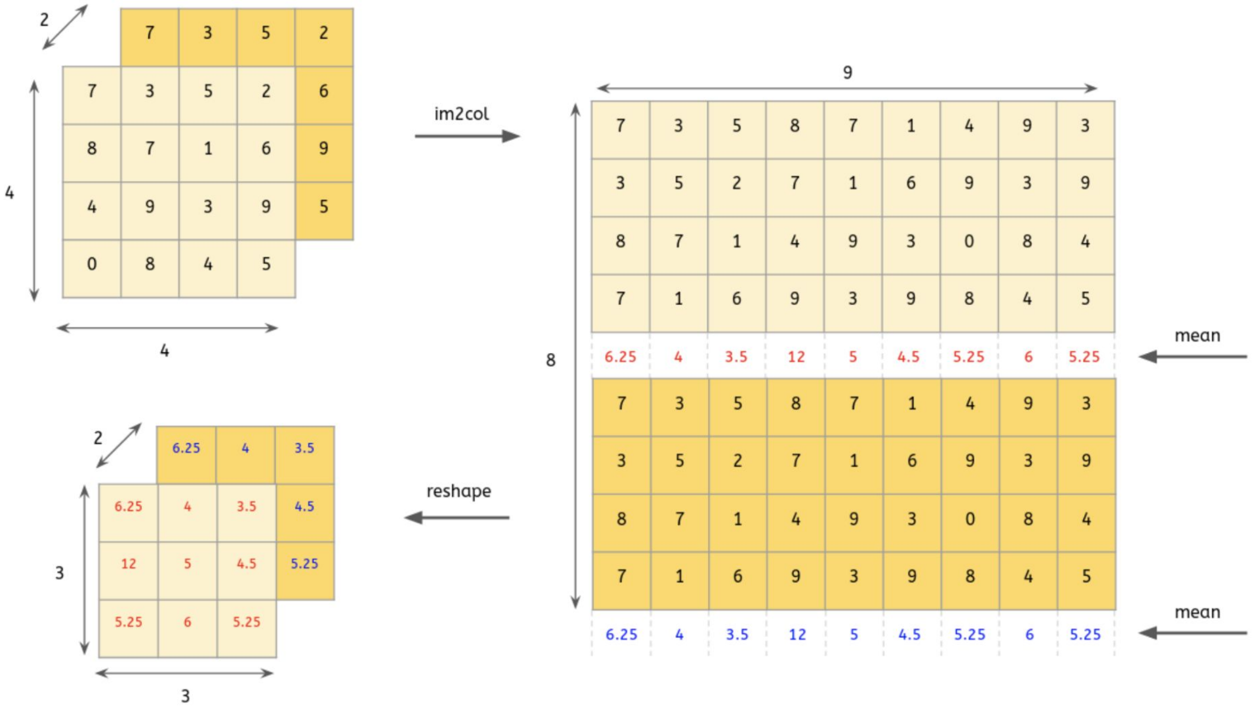


Step 3: Matrix multiplication between reshaped input and kernel

- At the end, we need to reshape our matrix back to a feature map
- Be aware the np.reshape() method doesn’t return the expected result here
  - Elements in the wrong order
  - A little bit of numpy *magic* solves the problem

Pooling Layer

- We can make the average pooling operation faster by using **im2col** method.
- Be aware that the `np.reshape()` method doesn't return the expected result here (elements in wrong order). A little bit of numpy gymnastic solves the problem.



Backward propagation

Reminder:

- We performed a convolution between (1,3,4,4) input image and kernels of shape (2,3,2,2) which output an (2,3,3) image.
- During the backward pass, the (2,3,3) image contains the error/gradient (**“dout”**) which needs to be back-propagated to the:
  - (1,3,4,4) input image (layer).
  - (2,3,2,2) kernels.

★ Layer gradient: Intuition

- The formula to compute the layer gradient is:

$$\boxed{\frac{\partial L}{\partial I} = Conv(K, \frac{\partial L}{\partial O})}$$

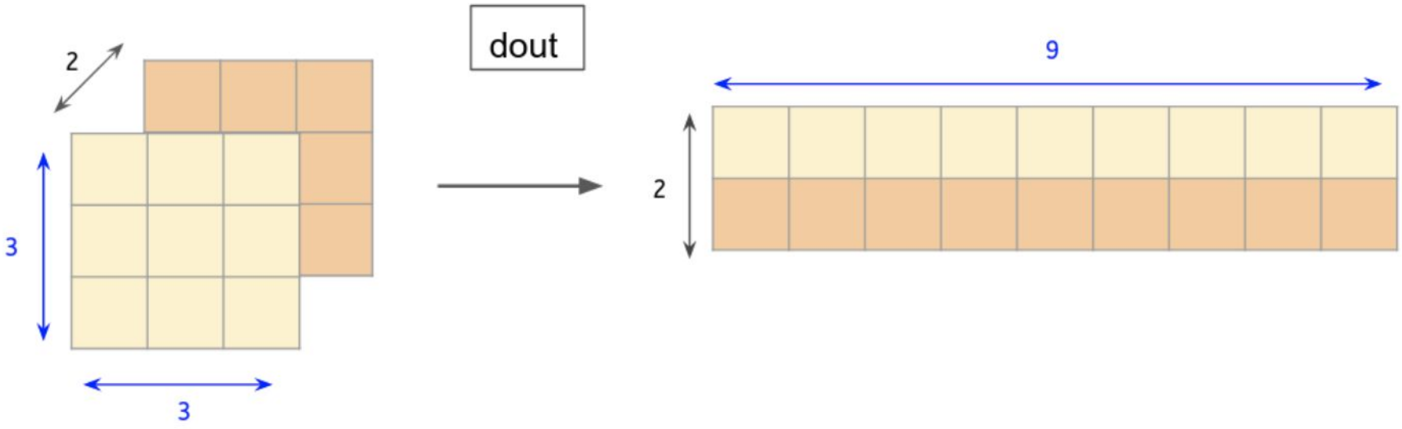
  - $\frac{\partial L}{\partial I}$: Input gradient.
  - K: Kernels.
  - $\frac{\partial L}{\partial O}$: Output gradient.
  - $Conv$: Convolution operation.

- To do so, we will proceed as follow:
  - **A.** Reshape dout ($\frac{\partial L}{\partial O}$).
  - **B.** Reshape kernels `w` into single matrix `w_col`.
  - **C.** Perform matrix multiplication between reshaped `dout` and kernel.
  - **D.** Reshape back to image (**col2im**).

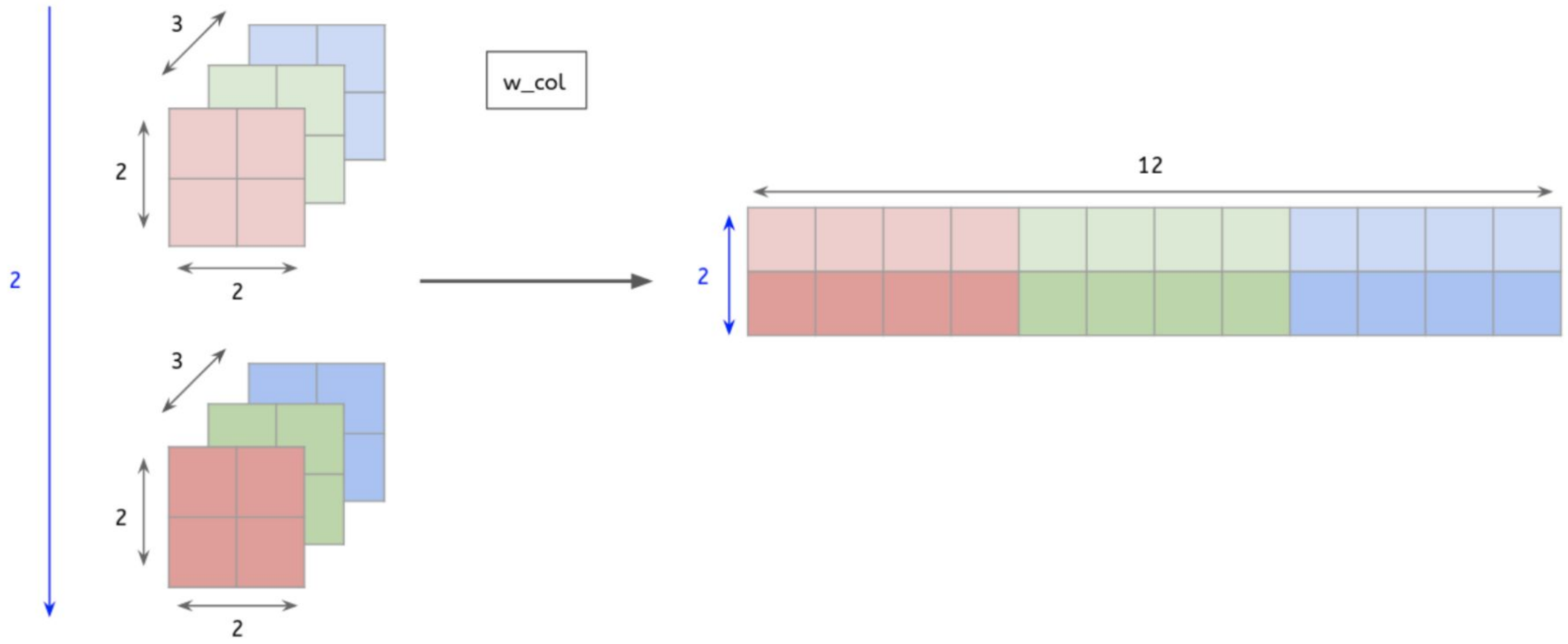
- We are going to see how it works intuitively and then how to implement it using Numpy.

A) Reshape `dout`

- During backward propagation, the output of the forward convolution contains the error that needs to be back-propagated.

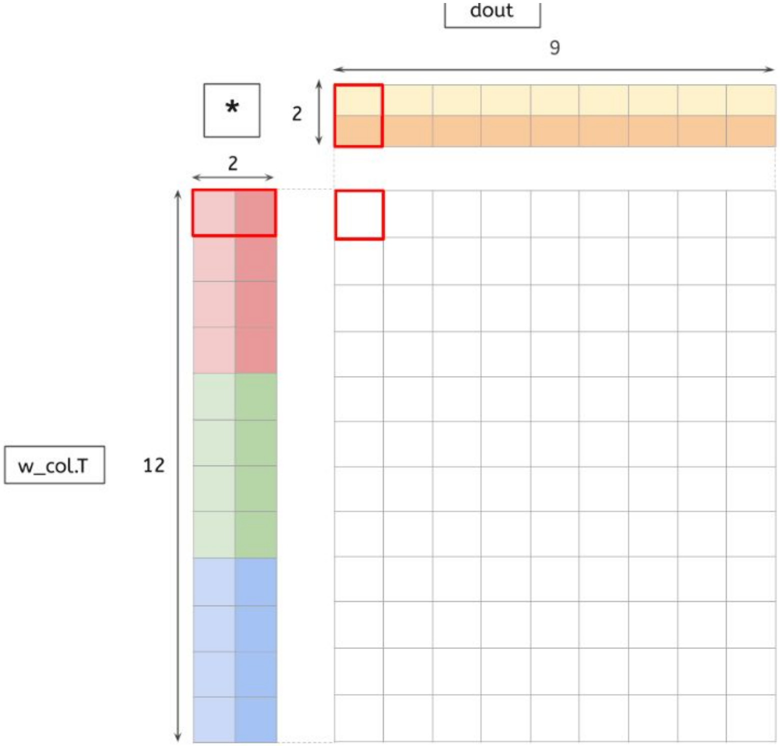


B) Reshape `w` into `w_col`



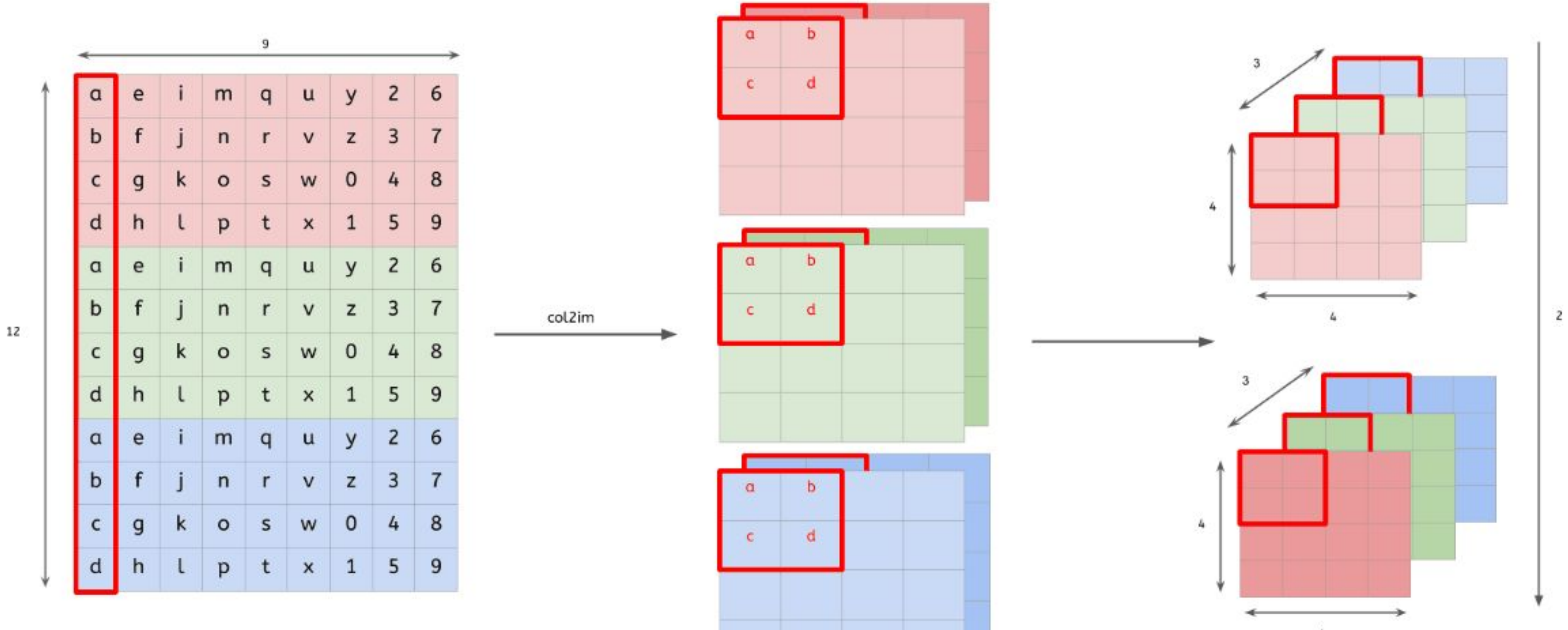
C) <u>Perform matrix multiplication between reshaped `dout` and `w_col`</u>

- In order to perform to perform the matrix multiplication, we need to transpose `w_col`.
- We will denoted the output as `dX_col`.



D) Reshape back to image (col2im)

- Here, **col2im** is more than a simple backward operation of im2col. Indeed, we have to take care of cases where errors will overlap with others.



○ Kernel gradient: Intuition

- The formula to compute the kernel gradient is:

$$\boxed{\frac{\partial L}{\partial K} = Conv(I, \frac{\partial L}{\partial O})}$$

  - $\frac{\partial L}{\partial K}$: Kernels gradient.
  - I: Input image.
  - $\frac{\partial L}{\partial O}$: Output gradient.
  - $Conv$: Convolution operation.

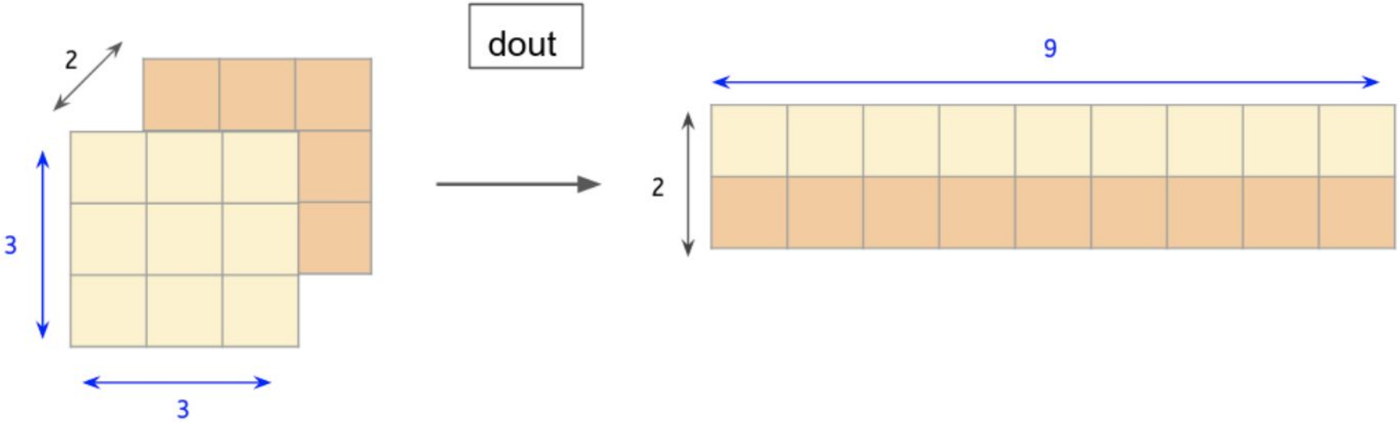
- To do so, we will:

  - **A.** Reshape dout ($\frac{\partial L}{\partial O}$).
  - **B.** Apply **im2col** on `X` to get `X_col`.
  - **C.** Perform matrix multiplication between reshaped `dout` and `X_col` to get `dw_col`.
  - **D.** Reshape `dw_col` back to `dw`.

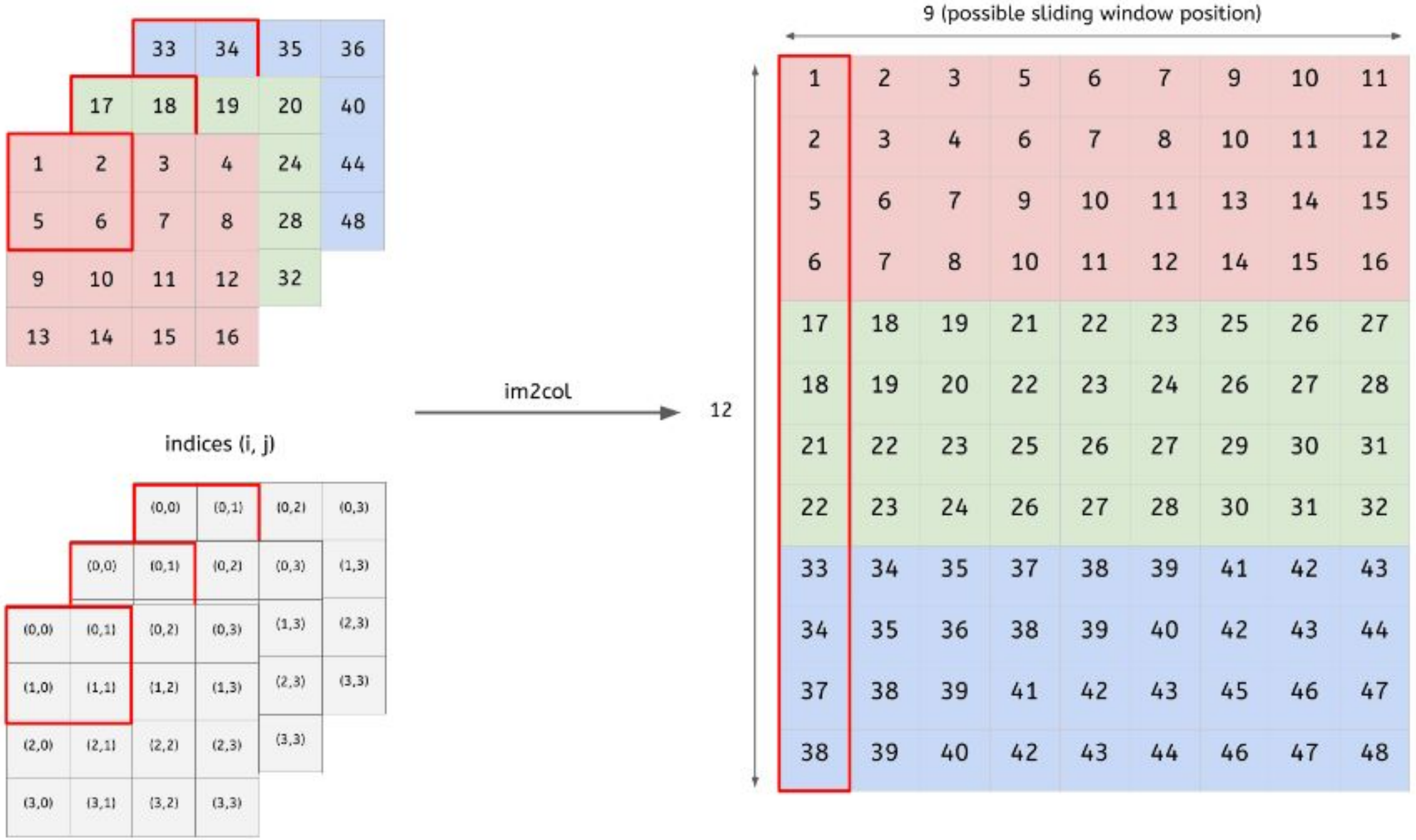
- We are going to see how it works intuitively and then how to implement it using Numpy.

A) Reshape `dout`

- Be aware that the `np.reshape()` method doesn't return the expect result here (elements in wrong order). A little bit of numpy gymnastic solves the problem.

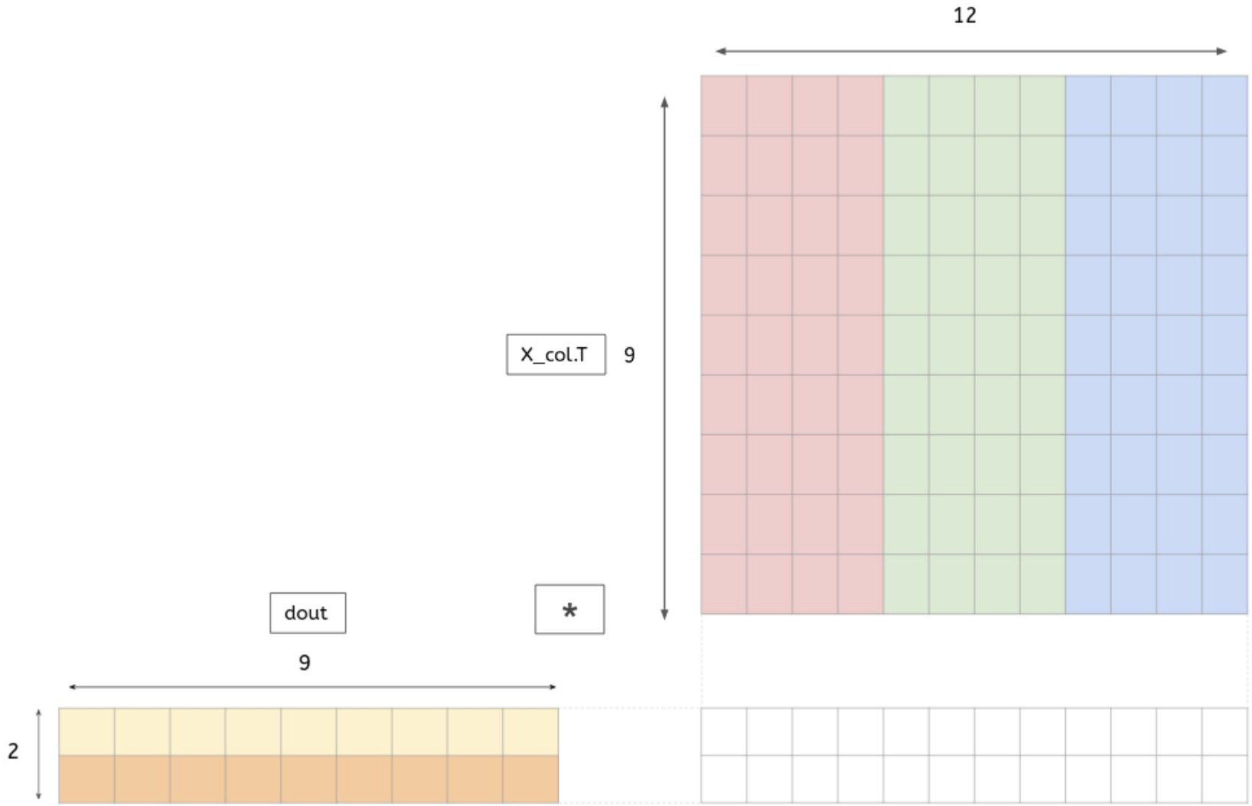


B) Apply im2col on `X` to get `X_col`



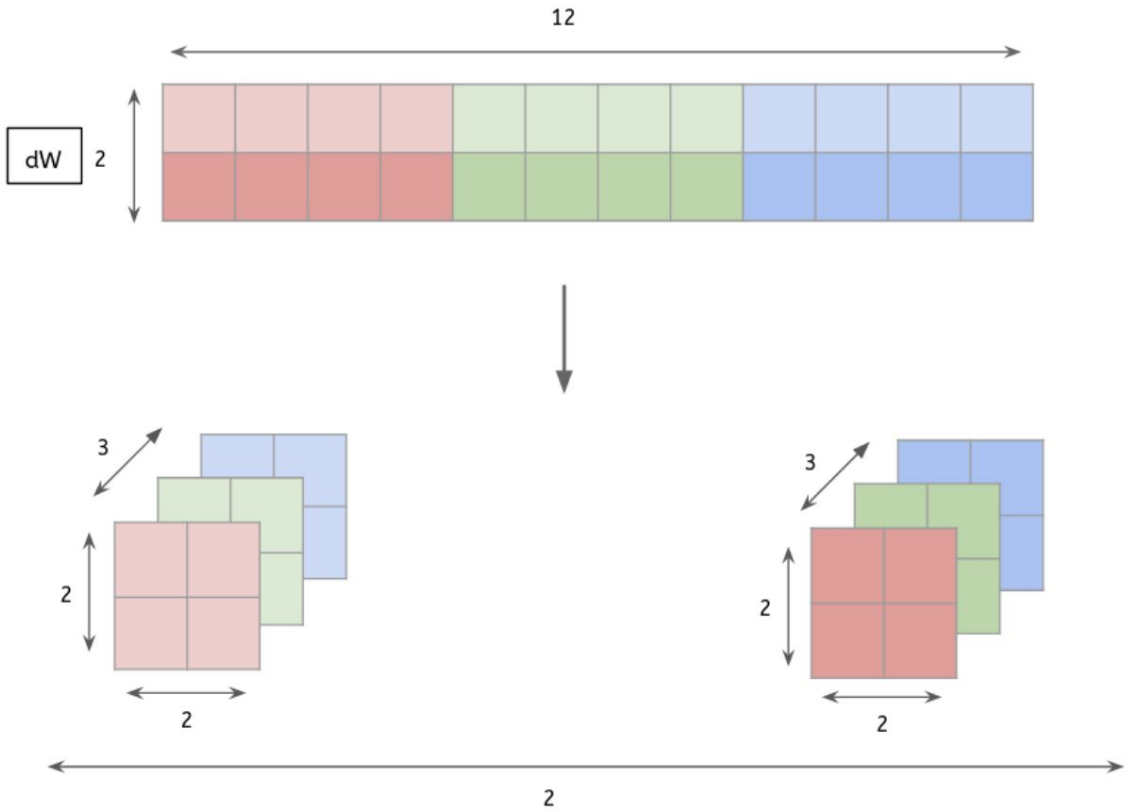
C) Perform matrix multiplication between reshaped dout and `X_col` to get `dw_col`

- In order to perform to perform the matrix multiplication, we need to transpose `X_col`.



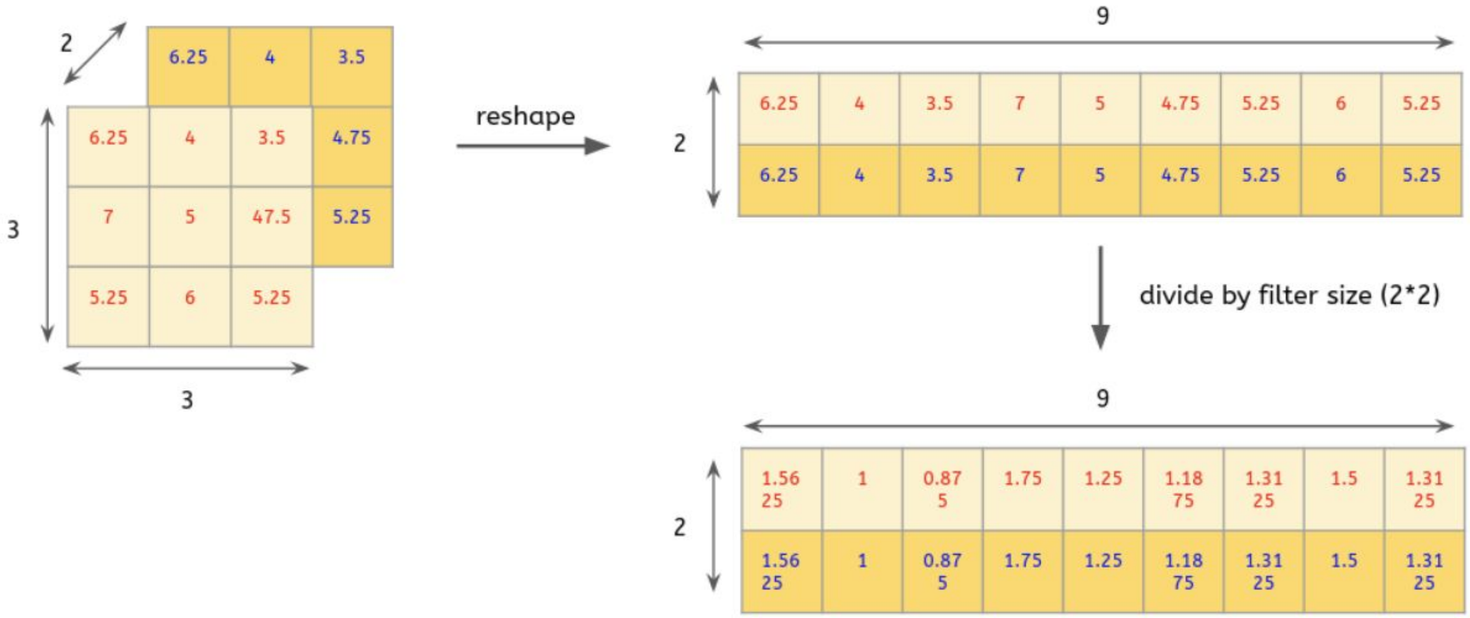
D) Reshape `dw_col` back to `dw`

- We simply need to reshape `dw_col` back to its original kernel shape.

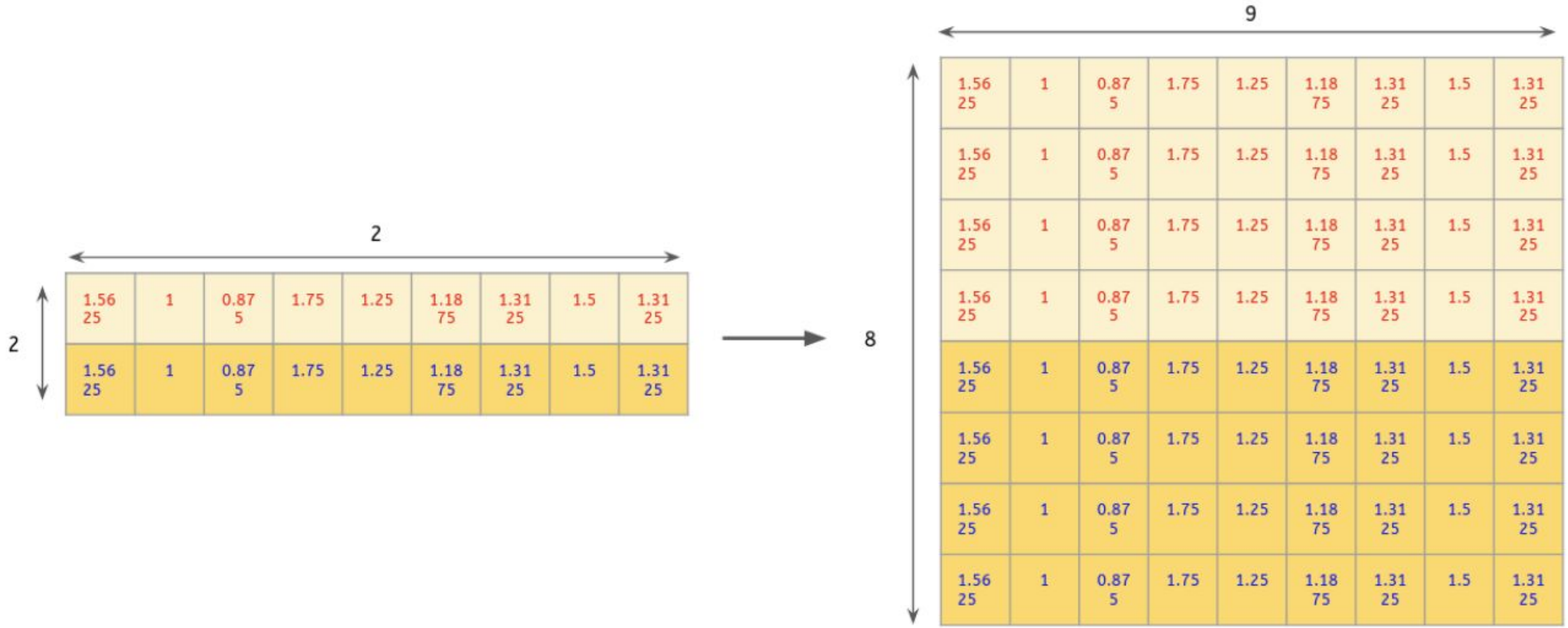


2) Pooling layer

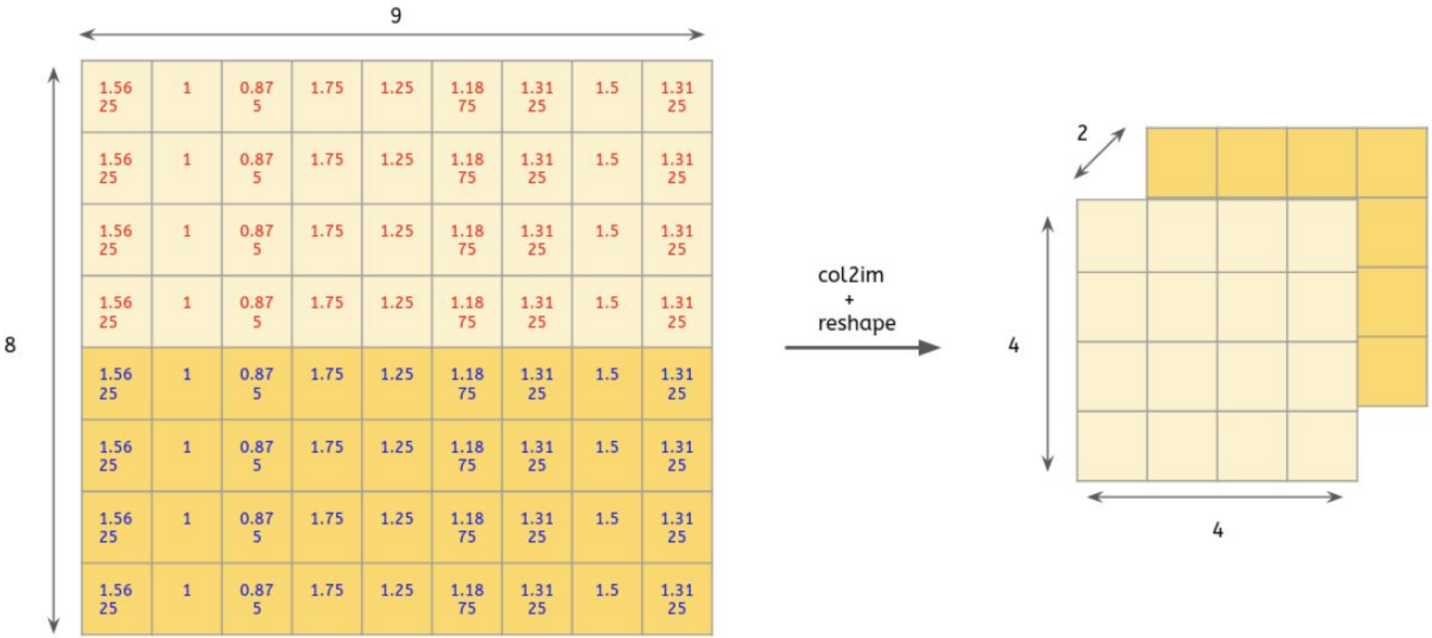
- We first have to reshape our filters and divide by the filter size.



- We then repeat each element “filter size” time.



- Finally, we apply **col2im**.
- Be aware that the `np.reshape()` method doesn't return the expected result here (elements in wrong order). A little bit of numpy gymnastic solves the problem.



Some suggestions

Please **start early**, this assignment takes time!!!

- Im2col usually takes the most effort, try derive the example we did above using your hand to better understand it
- Make sure to test it correctly with small hand-derived example.