

HW1 Recitation: **MLPs and CNNs**

10707: Advanced Deep Learning

Chris Ki, Albert Liang, Jocelyn Tseng

Administrative

Homework 1 is due **Wednesday, Feb 15, 11:59 pm**

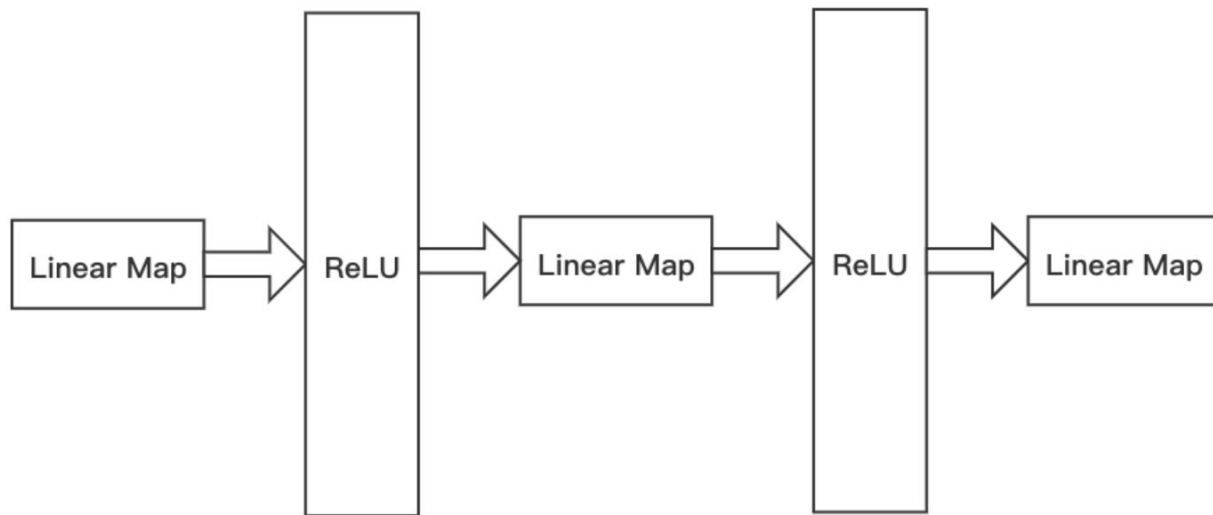
Please check Piazza for HW1 FAQs note for updates and questions!

MLPs

1. Overview
2. DropOut
3. BatchNorm

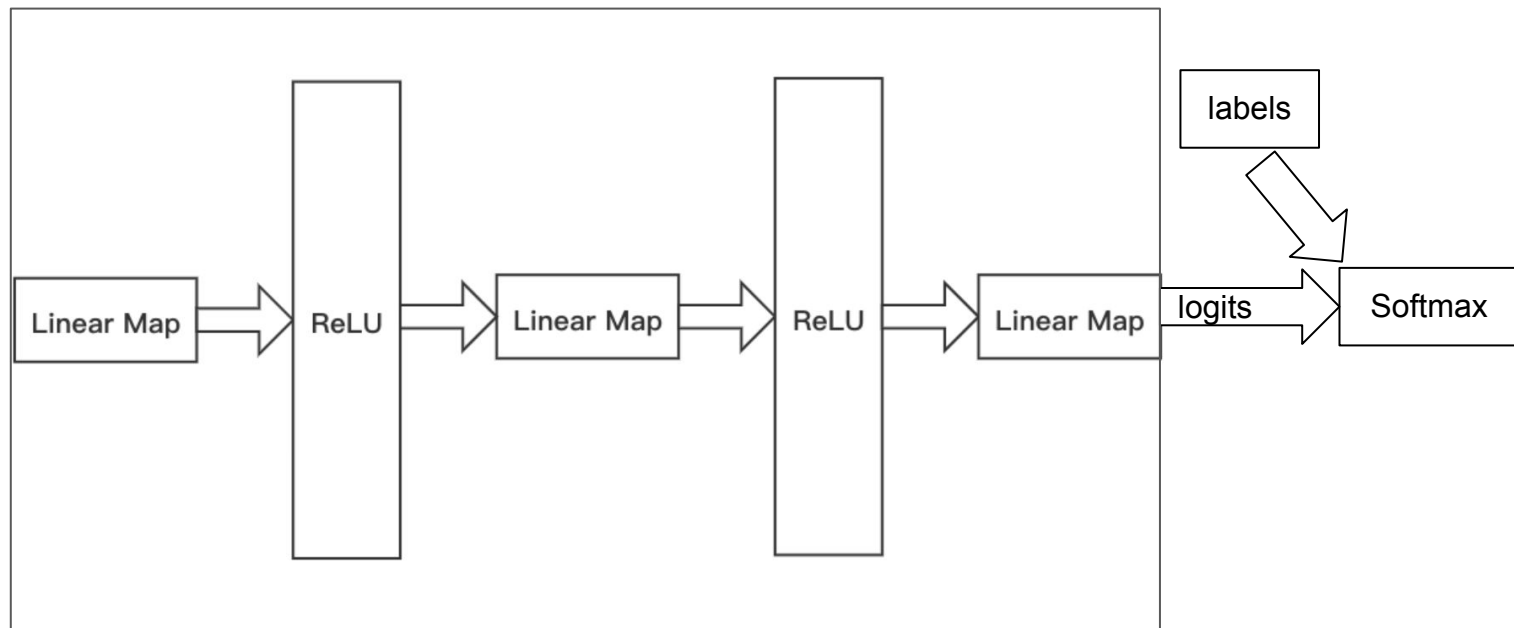
****Recommend doing problem 3 before coding MLP's and CNN's****

MLP Overview



Regularization goes before activation layer

Training: Forward

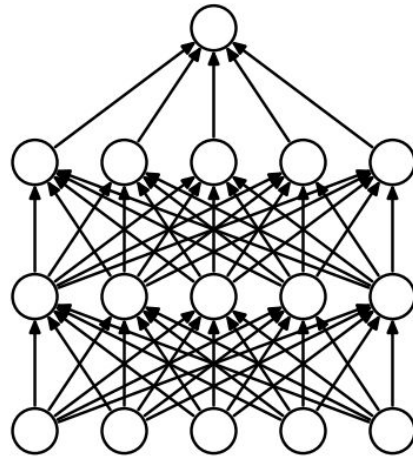


Training: Backward

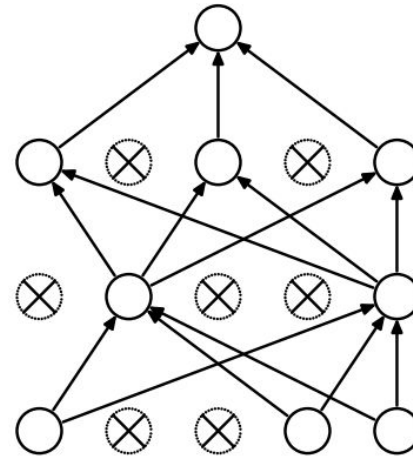
- Softmax backward outputs grads
- grads => backpropagation
- step to apply gradient updates to weights
- remember to zerograd so gradients do not accumulate across epochs

Dropout

- Only perform dropout for training
- Generate mask based on probability p , dropping out with probability p
- use `np.random.binomial`



(a) Standard Neural Net



(b) After applying dropout.

Batchnorm

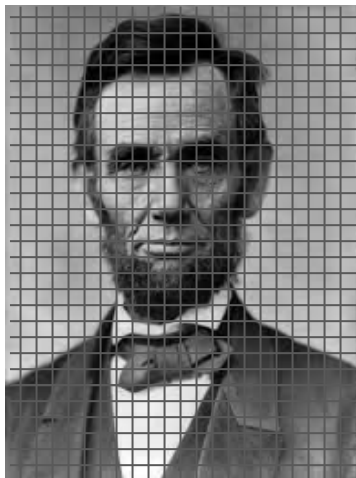
- GOAL: **prevent activation saturation** and/or exploding gradients when using sigmoid
- for each hidden layer of the currently training minibatch, you want to **normalize** the activations to a **standard normal distribution**
- then apply **linear transformation** with learned parameters

CNNs

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

What is an image?

A Matrix \mathbf{I} of dimensions (C, M, N) with $\mathbf{I}[c][i][j] = \text{intensity}(\text{pixel}(c, i, j))$



157	153	174	168	150	162	129	161	172	161	165	164
155	182	163	74	75	62	93	17	110	210	180	154
180	180	50	14	54	6	10	93	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	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	90	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

157	153	174	168	150	162	129	161	172	161	165	164
155	182	163	74	75	62	93	17	110	210	180	154
180	180	50	14	54	6	10	93	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	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	90	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

Applying a kernel to a channel of the image

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

Input - **A**

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

Kernel - **W**

$B_{1,1}$

Bias - **B**

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

Output - **Z**

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

Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations

The diagram illustrates the process of applying a kernel to a channel of an image. It shows a 4x4 grid of elements $A_{i,j}$ (representing the image channel) being multiplied element-wise by a 2x2 kernel $W_{i,j}$. The result is then summed with a bias $B_{1,1}$ to produce the output element $Z_{1,1}$.

$$Z_{1,1} = (A_{1,1} * W_{1,1}) + (A_{1,2} * W_{1,2}) + (A_{2,1} * W_{2,1}) + (A_{2,2} * W_{2,2}) + B$$

Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations

The diagram illustrates the process of applying a kernel to a channel of an image. It shows the following components and operations:

- Input Matrix A:** A 4x4 grid of elements $A_{i,j}$. The elements $A_{1,2}$, $A_{1,3}$, $A_{2,2}$, and $A_{2,3}$ are highlighted in yellow, representing the 2x2 kernel area.
- Kernel W:** A 2x2 grid of elements $W_{i,j}$.
- Bias B:** A single element $B_{1,1}$.
- Output Matrix Z:** A 2x2 grid of elements $Z_{i,j}$.

The operation is represented by the equation:

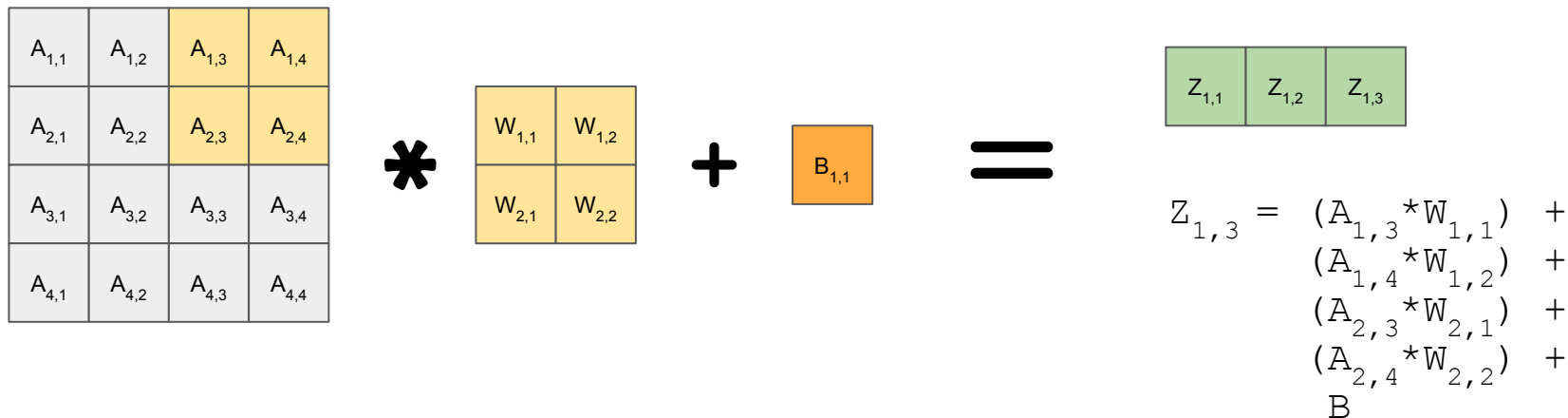
$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} * \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} \end{bmatrix}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

The output $Z_{1,2}$ is calculated as the sum of the following terms:

$$Z_{1,2} = (A_{1,2} * W_{1,1}) + (A_{1,3} * W_{1,2}) + (A_{2,2} * W_{2,1}) + (A_{2,3} * W_{2,2}) + B$$

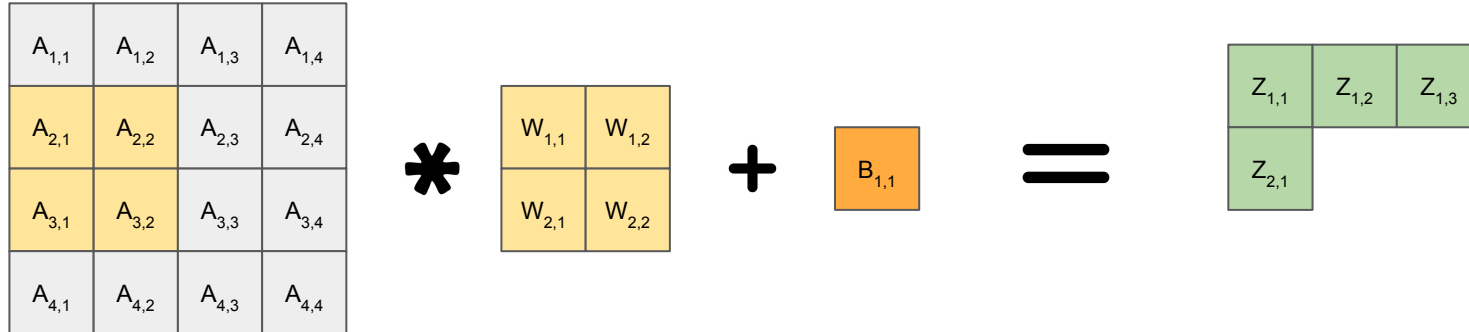
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



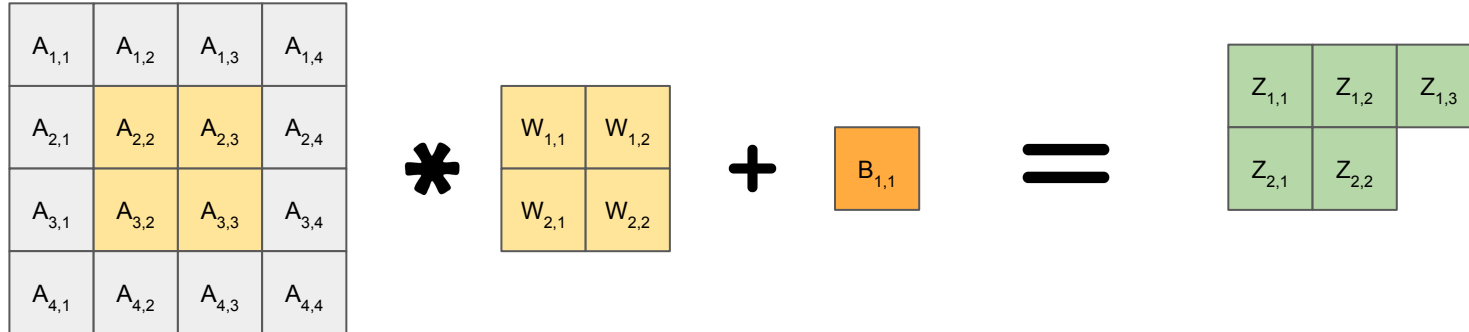
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



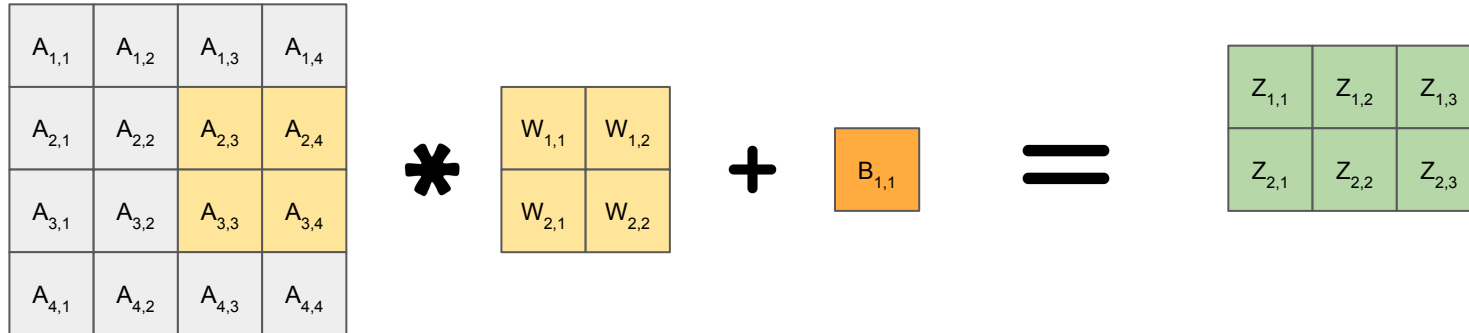
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



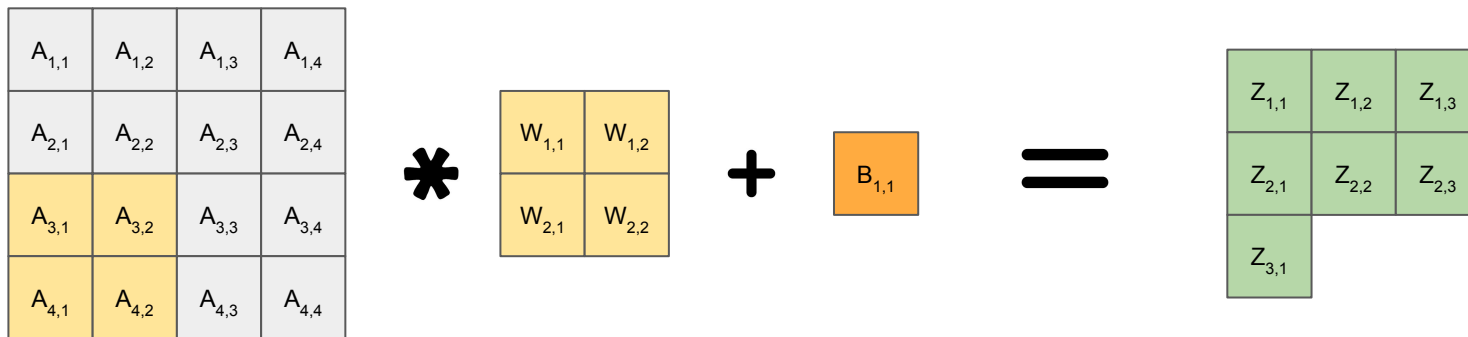
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



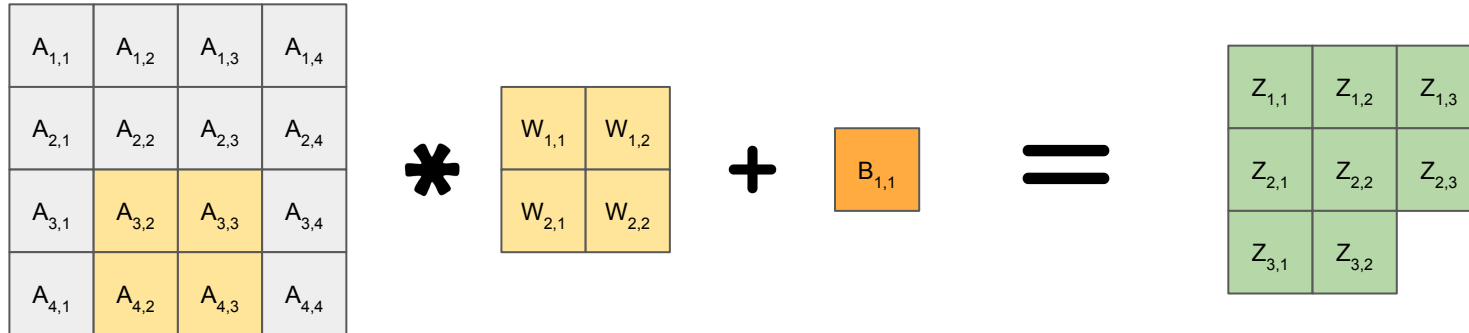
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



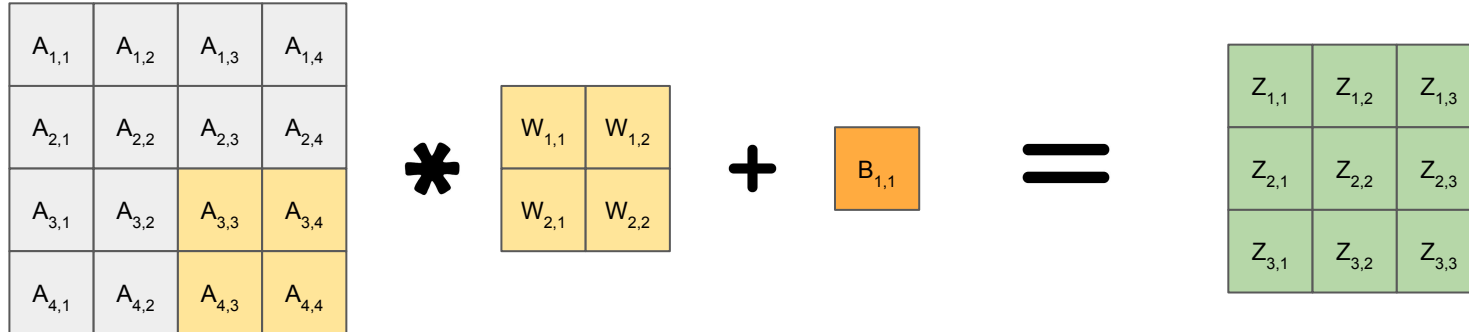
Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



Applying a kernel to a channel of the image

Essentially element-wise multiplications and summations



Formula of output size

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



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

Formula of output size

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



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

$$\text{Output Width} = \left[\frac{(W_{in} - W_k + 2P)}{(S)} \right] + 1$$

$$\text{Output Height} = \left[\frac{(H_{in} - H_k + 2P)}{(S)} \right] + 1$$

Formula of output size

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



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

$$\text{Output Width} = \left[\frac{(W_{\text{in}} - W_k + 2P)}{S} \right] + 1$$

Formula of output size

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



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

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

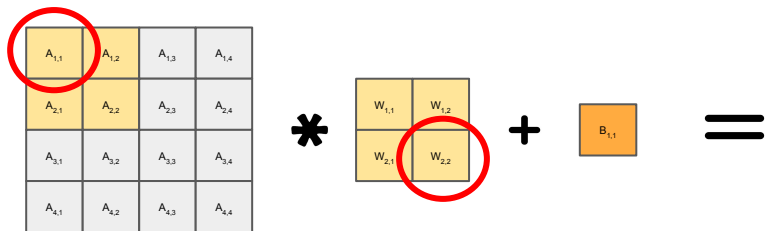
P: Padding (here - 0)

S: Stride (here - 1)

Padding

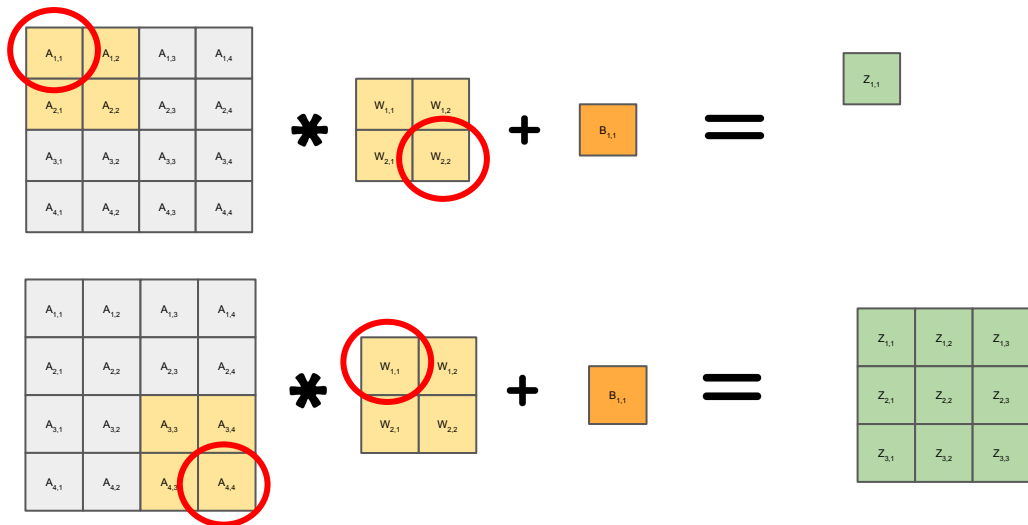
- Padding (i.e. attaching) zeros (usually) around inputs.
- Images can be padded to the left, right, top, and bottom.

One reason for padding

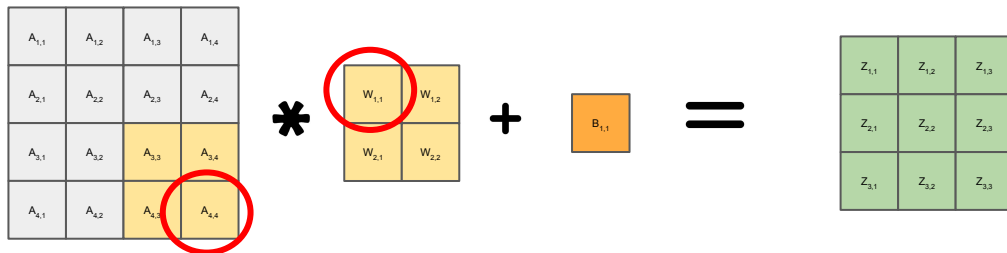
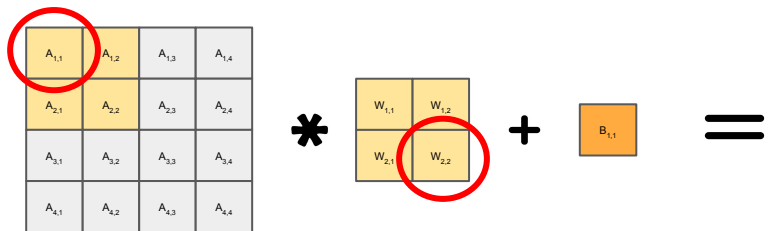


$$z_{1,1} = (A_{1,1} * W_{1,1}) + (A_{1,2} * W_{1,2}) + (A_{2,1} * W_{2,1}) + (A_{2,2} * W_{2,2}) + B$$

One reason for padding



One reason for padding



Never Meet...

After padding

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

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

=

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

After padding

0	0	0	0	0	0
0	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	0
0	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	0
0	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	0
0	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	0
0	0	0	0	0	0

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

=

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

After padding

0	0	0	0	0	0
0	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	0
0	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	0
0	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	0
0	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	0
0	0	0	0	0	0

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

=

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

After padding

0	0	0	0	0	0
0	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	0
0	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	0
0	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	0
0	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	0
0	0	0	0	0	0

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

=

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



One reason for padding

Reason 1: increases the interaction between the input and the kernel

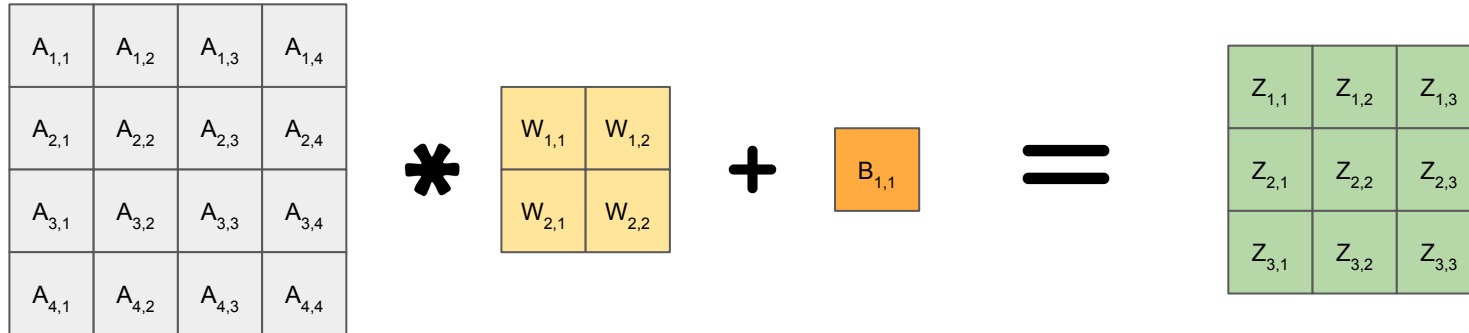
Reason 2: increase output size / preserves input size

Stride

How far the kernel moves in every iteration for each direction

Stride = 1

In the previous slides: the kernel “moves” one pixel at a time.



Stride = 2

Start at the same place

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

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

=

$Z_{1,1}$

$$Z_{1,1} = (A_{1,1} * W_{1,1}) + (A_{1,2} * W_{1,2}) + (A_{2,1} * W_{2,1}) + (A_{2,2} * W_{2,2}) + B$$

Stride = 2

Move two pixels to the right

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

*

$W_{1,1}$	$W_{1,2}$
$W_{2,1}$	$W_{2,2}$

+

$B_{1,1}$

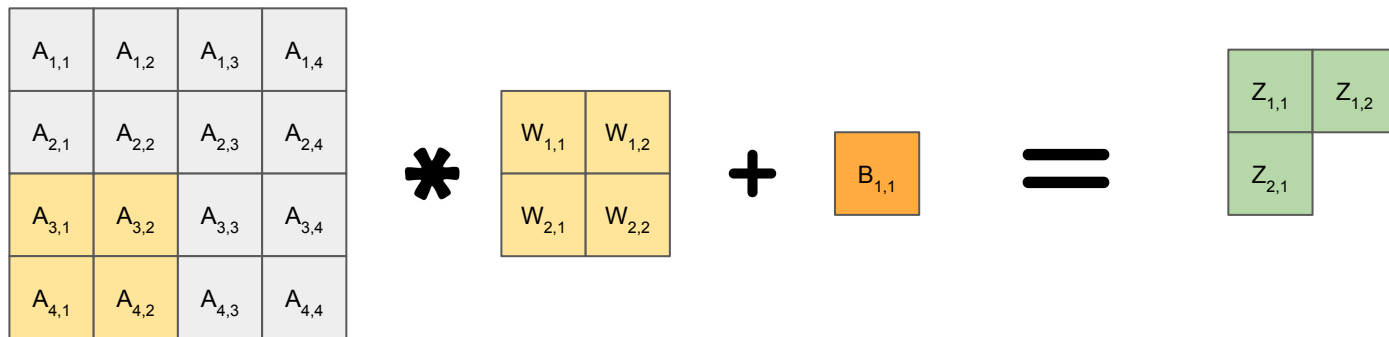
=

$Z_{1,1}$	$Z_{1,2}$
-----------	-----------

$$Z_{1,2} = (A_{1,3} * W_{1,1}) + (A_{1,4} * W_{1,2}) + (A_{2,3} * W_{2,1}) + (A_{2,4} * W_{2,2}) + B$$

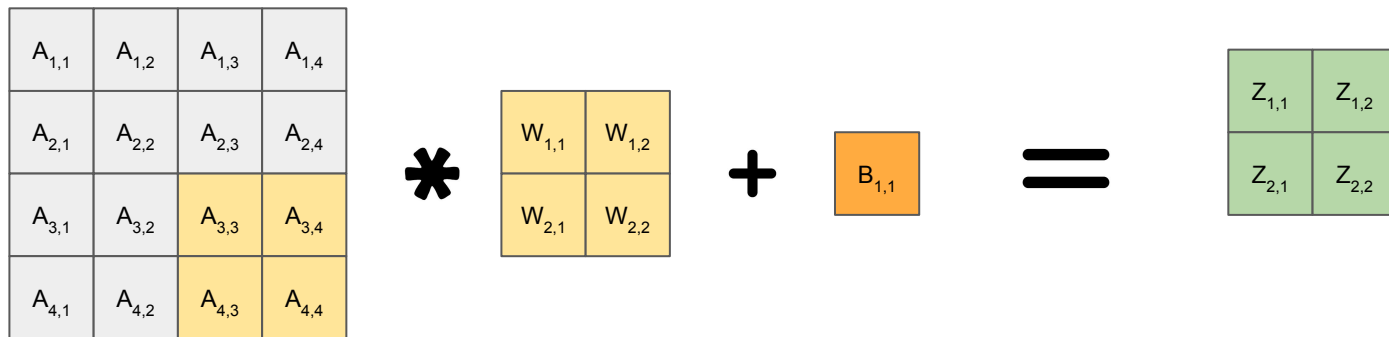
Stride = 2

Move two pixels down.

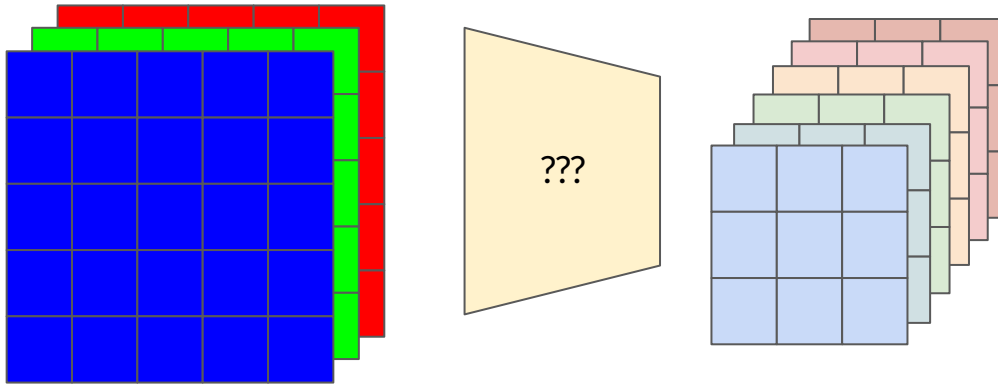


Stride = 2

Move two pixels to the right.



Multi-channel Convolution



Multi-channel Convolution

- Each filter has as many kernels as the number of channels in the input
- The c -th kernel of the filter convolves with c -th channel of the input.
- The number of output channels from the convolution = number of **filters**

C_{in} = Input channels

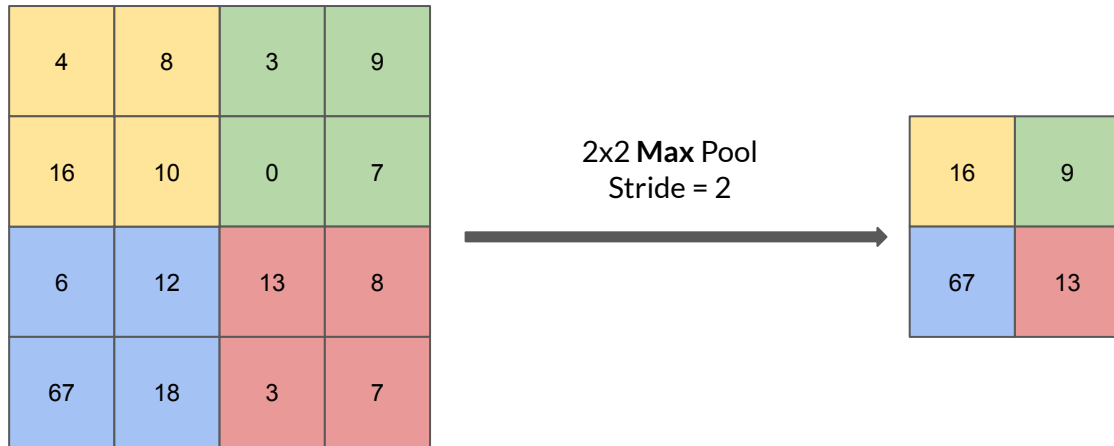
C_{filter} = Filter channels = C_{in}

F = Number of filters = C_{out} = Number of output channels

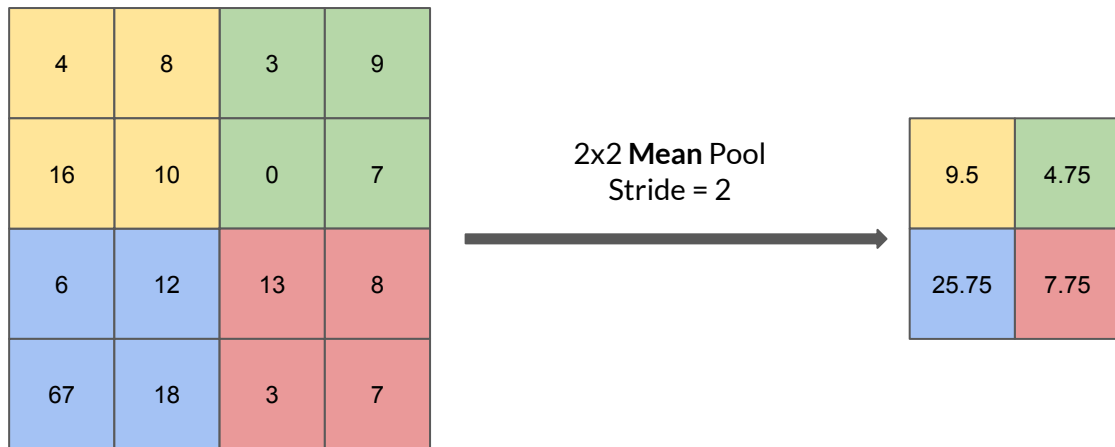
Pooling

- Usually applied after convolution
- Reduces height & width of the output
- Three common pooling operations:
 - Max Pooling
 - Mean Pooling
 - Min Pooling

Max Pooling



Mean Pooling



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 Pass for Conv Layer

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

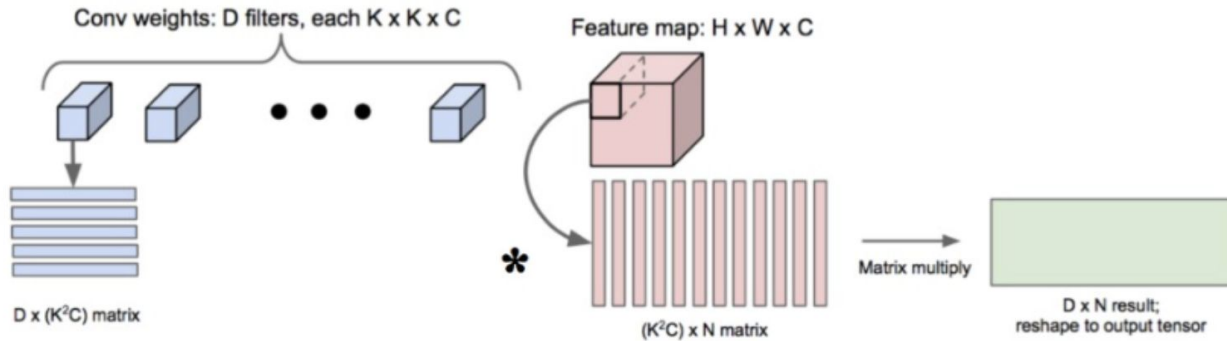


Figure 2: Illustration of im2col.

Forward Pass for Conv Layer

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.

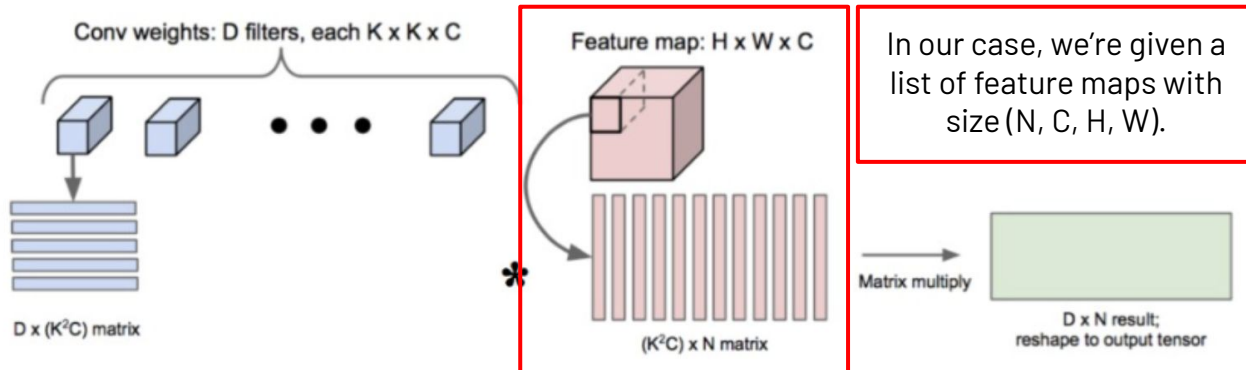


Figure 2: Illustration of im2col.

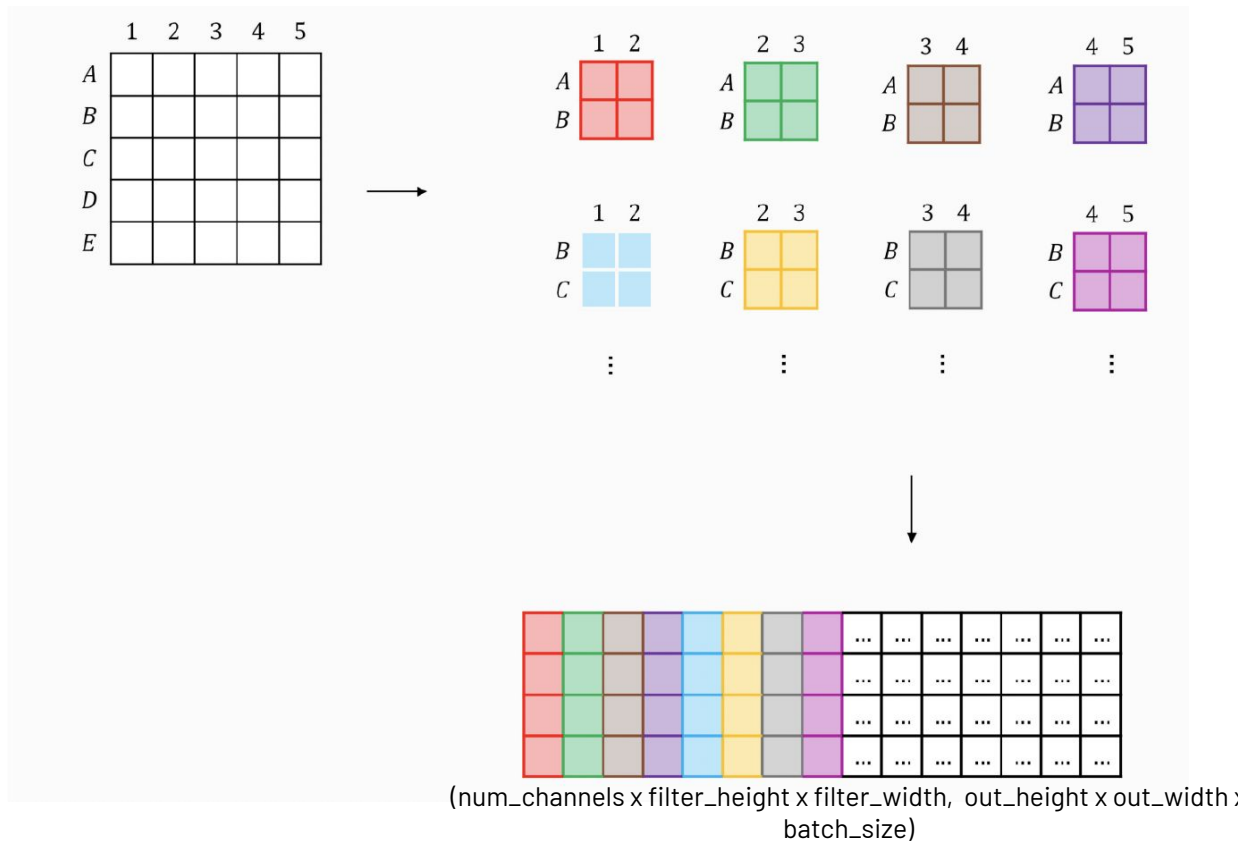
Im2col Simple Example

For this example, we have the following settings:

- Batch_size = 1
- Num_channels = 1
- Filter height and width = 2
- Feature map height and Width = 5
- Padding = 0
- Stride = 1

Key Intuition: Store all pixels of each window of feature map that will be applied to a filter in a single column.

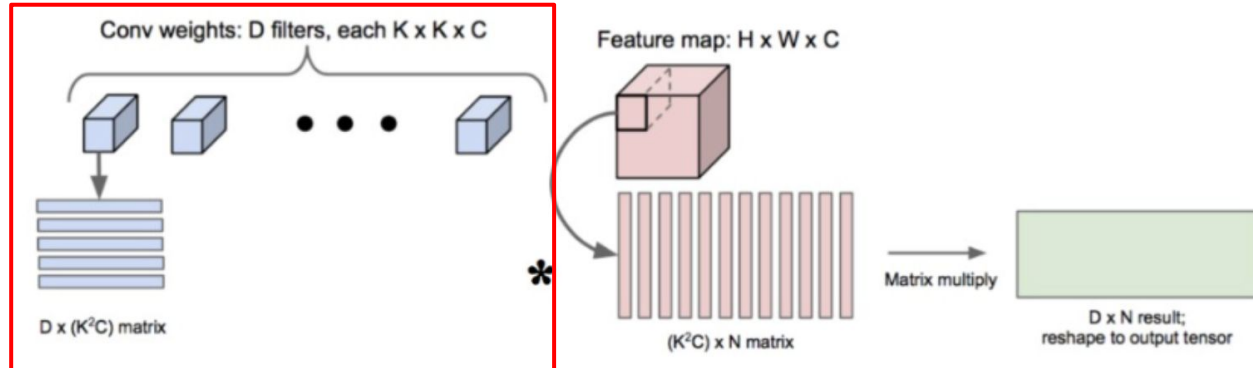
Important: Keep in mind out_height and out_width will be dependent on the padding and stride used!



Forward Pass for Conv Layer

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



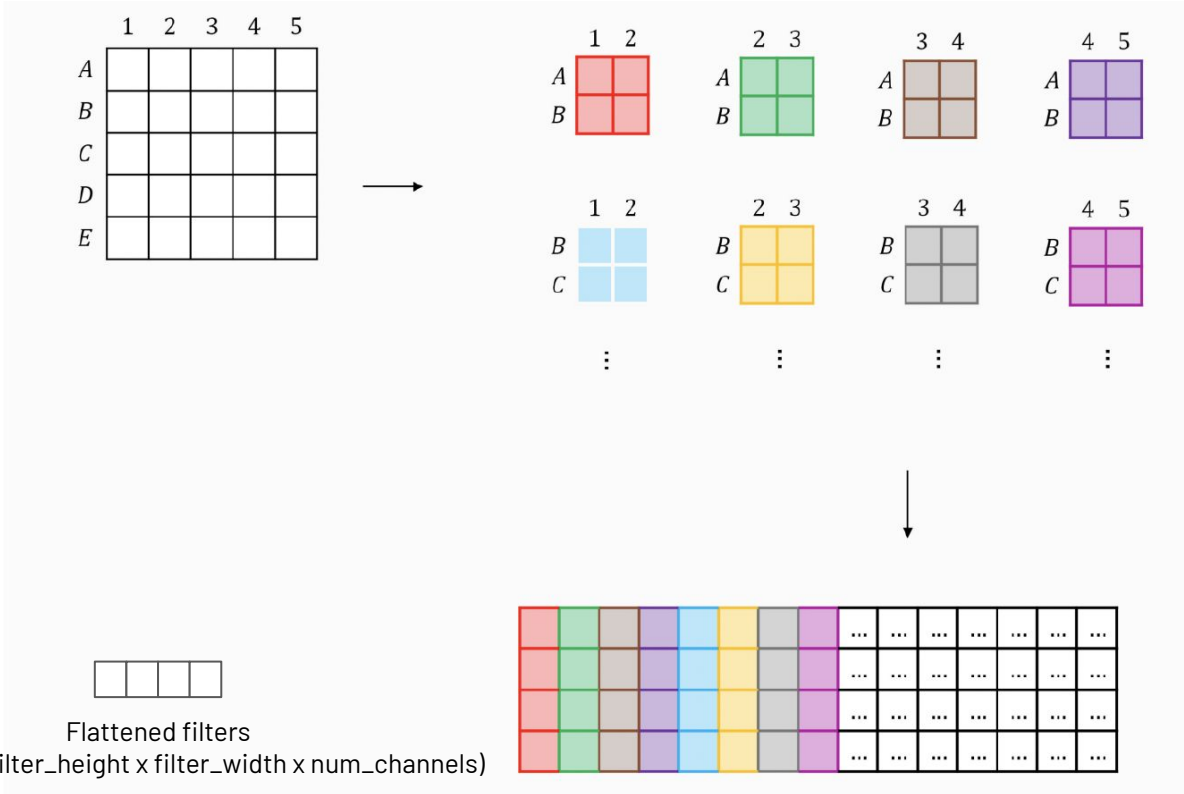
In our case, our filters have size $(D, C, K_height, K_width)$.

Figure 2: Illustration of im2col.

Reshaping our kernels

For this example, we have the following settings:

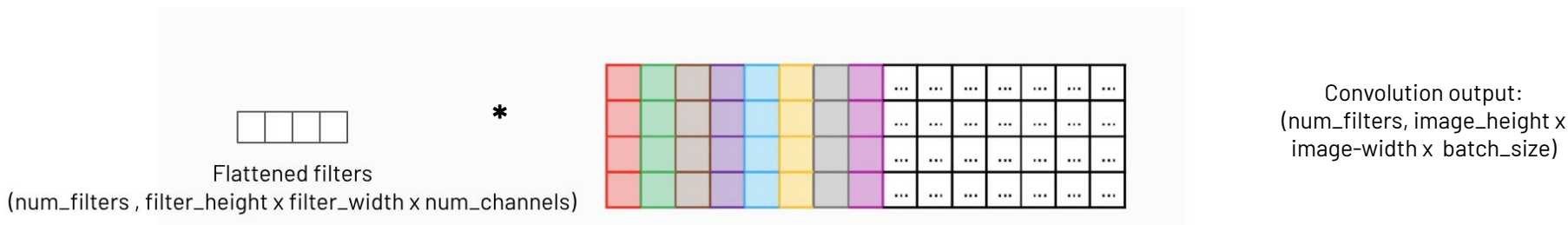
- Batch_size = 1
- Num_channels = 1
- Filter height and width = 2
- Image height and Width = 5
- Padding = 0
- Number of filters = 1



Forward Pass for Conv Layer

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



Reshape to output tensor of shape
(batch_size, num_filters, height,
width).

Backward Pass for Conv Layer

Here are the following steps given dLoss

1. Compute gradients w.r.t weights of filters and biases.
 - a. Hint: Involves taking matrix product of dLoss and im2col output.
2. Compute gradient w.r.t input image
 - a. Run im2col_bw on grad_x_col (a gradient w.r.t im2col output), which is a 2D array of size (num_filters, image_height x image-width x batch_size).

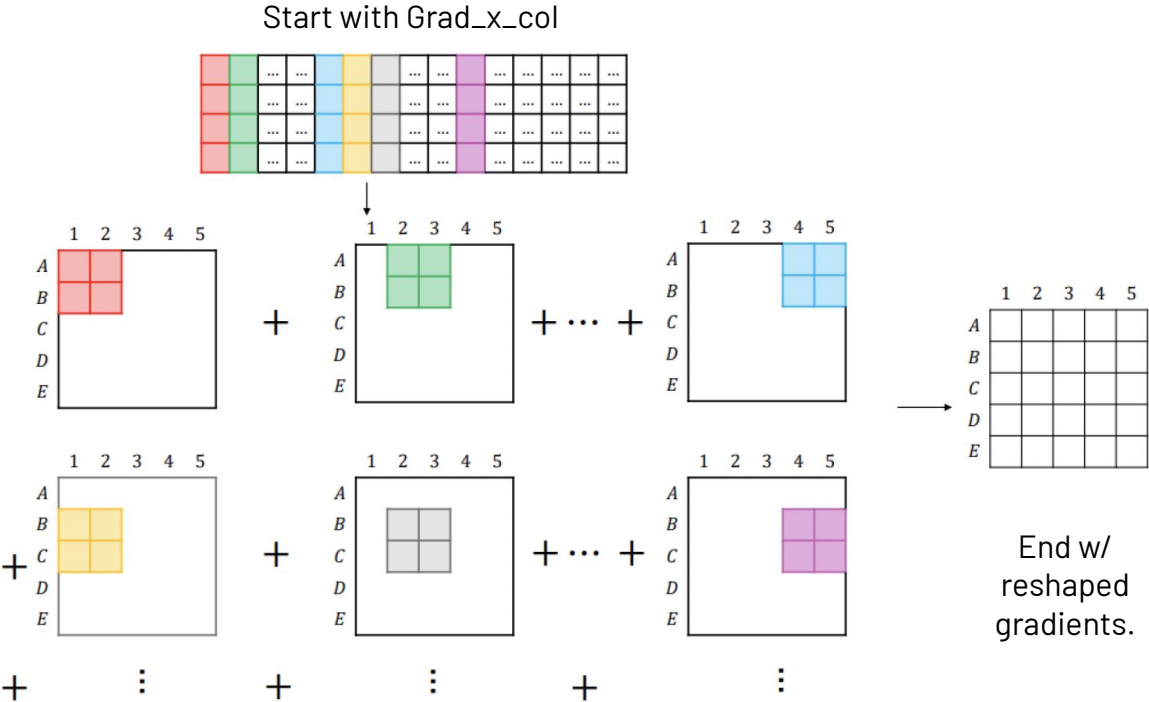
Im2col_bw Simple Example

Im2col_bw takes as an argument grad_X_col (a 2D array).

For this example, we have the following settings:

- Batch_size = 1
- Num_channels = 1
- Filter height and width = 2
- Image height and Width = 5
- Padding = 0

Return output with similar shape to input for im2col (batch_size, num_channels, height, width).



Some suggestions

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

- `lm2col` usually takes the most effort, make sure to test it correctly with small hand-derived example.
- Useful functions to keep in mind:
 - `np.transpose` can be used to shift axes (e.g., `A` is an array with shape `(1,2,3)`. `np.transpose(A, (1,0,2))` will return an array with shape `(2,1,3)`).
 - `np.reshape` is useful for “reshaping” your array. Be careful when using `np.reshape(dimension, -1)` on 3D/4D data as the order of the axes matter!