



CS540 Introduction to Artificial Intelligence

Deep Learning I: Convolutional Neural Networks

Yingyu Liang

University of Wisconsin-Madison

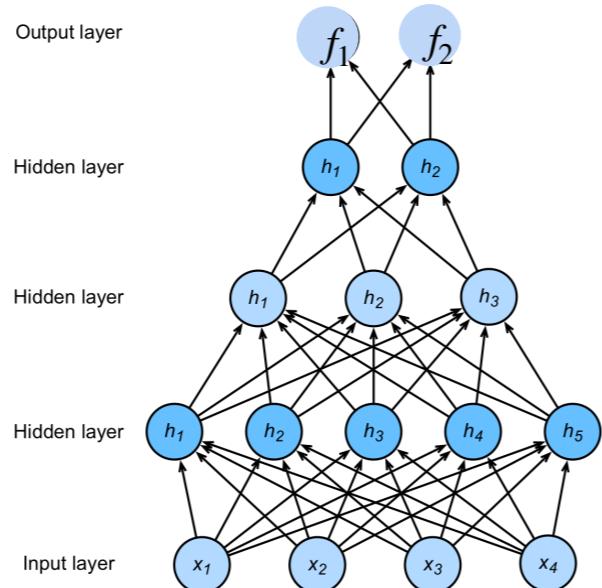
Nov 2, 2021

Slides created by Sharon Li [modified by Yingyu Liang]

Outline

- Intro of convolutional computations
 - 2D convolution
 - Padding, stride
 - Multiple input and output channels
 - Pooling

Review: Deep neural networks (DNNs)



$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}_2 = \sigma(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$$

$$\mathbf{h}_3 = \sigma(\mathbf{W}_3 \mathbf{h}_2 + \mathbf{b}_3)$$

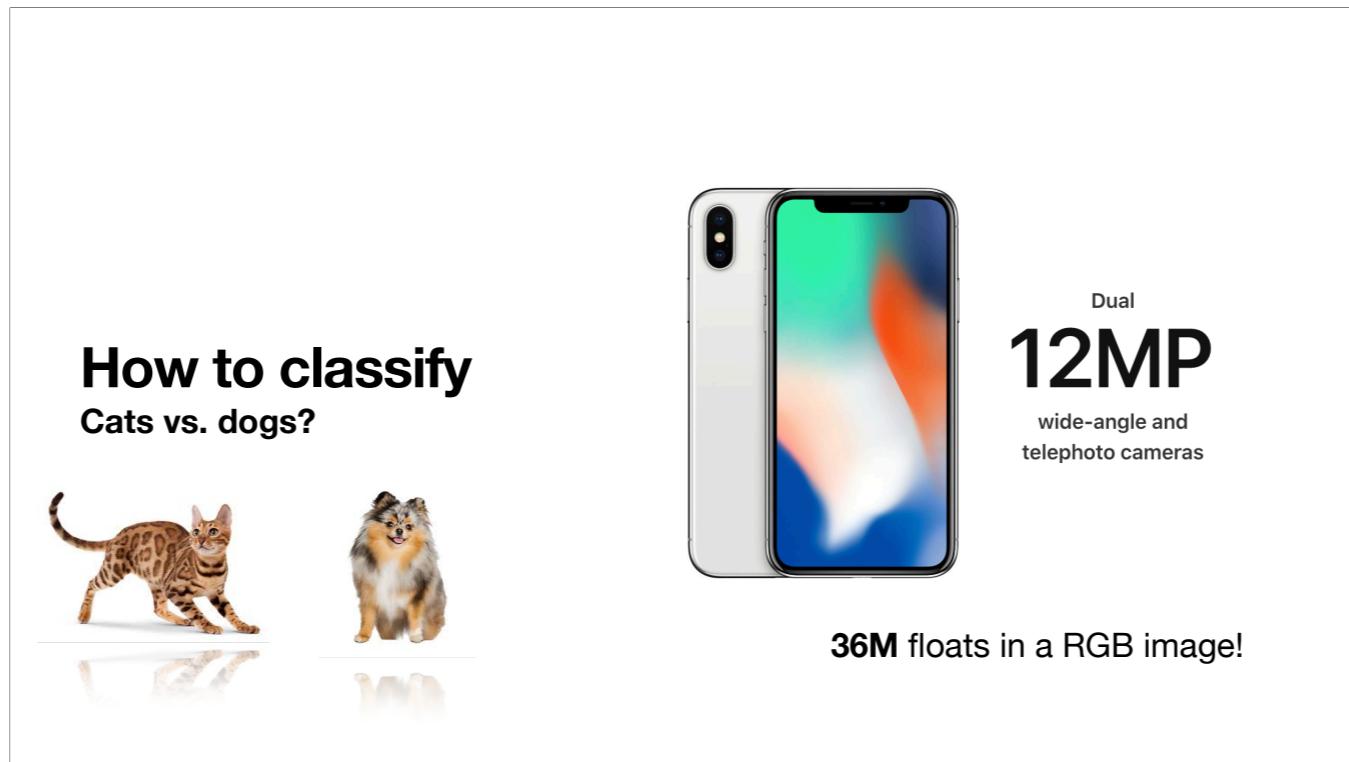
$$\mathbf{f} = \mathbf{W}_4 \mathbf{h}_3 + \mathbf{b}_4$$

$$\mathbf{y} = \text{softmax}(\mathbf{f})$$

NNs are composition
of nonlinear
functions

We first want to answer the question: why we need convolutional neural networks? Can't we just use the standard deep networks?

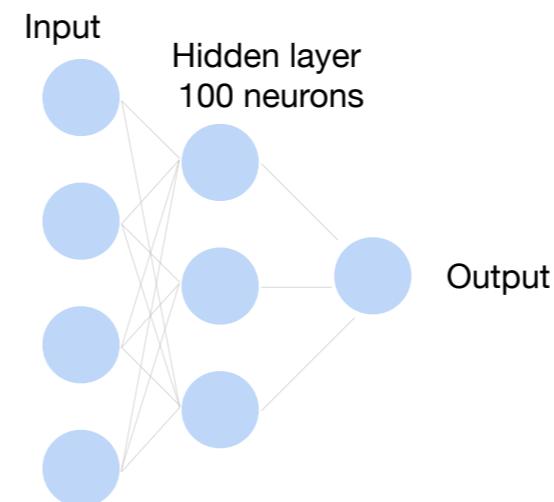
We will mention two main reasons.



To illustrate the first reason, consider the running example of image classification. A typical image can have millions of pixels.

Fully Connected Networks

Cats vs. dogs?



$\sim 36M \text{ elements} \times 100 = \sim \mathbf{3.6B} \text{ parameters!}$

Suppose we use a standard neural network (also called fully connected networks in contrast to convolutional networks). Even with only one hidden layer with only 100 hidden neurons, the number of parameters in the first layer is \sim 3.6 billion! This is too large even for modern computing resources.

Convolutions come to rescue!

Convolutional networks have much less parameters, and thus are used to replace the fully connected networks.



The second reason to use convolutions is that they can make use of properties of computer vision tasks on images. Consider identifying the face of a particular person in an image with many faces.

Why Convolution?

- Translation Invariance
- Locality



The pattern we're interested in (the face) appears in a local patch. This is called the locality property.

The pattern can also appear in different locations of the images. This is called translation invariance property.

Convolution can make use of these two properties.

2-D Convolution

Input	Kernel	Output													
<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td></tr><tr><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td><td style="padding: 2px;">5</td></tr><tr><td style="padding: 2px;">6</td><td style="padding: 2px;">7</td><td style="padding: 2px;">8</td></tr></table>	0	1	2	3	4	5	6	7	8	*	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td></tr><tr><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td></tr></table>	0	1	2	3
0	1	2													
3	4	5													
6	7	8													
0	1														
2	3														
	=	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">19</td><td style="padding: 2px;">25</td></tr><tr><td style="padding: 2px;">37</td><td style="padding: 2px;">43</td></tr></table>	19	25	37	43									
19	25														
37	43														

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$

Given 2D input matrix, convolution has a 2D kernel matrix (also called filter). We put a sliding window (of the same size as the kernel) on the input, and then multiply the corresponding entries in the window and in the kernel, and sum up the multiplications' outcomes. This gives an output entry.

Then we move the sliding window to different locations in the input matrix, and compute in the same way to get the corresponding output entries.

Note that: if we view the sliding window and kernel as vectors, then the output entry is exactly the inner product between these two vectors. Inner products measure similarities. Therefore, the output entry can be viewed as a measurement of the similarity between the sliding window and the pattern represented by the kernel. If the window contains the pattern of kernel, the output entry will be large (detecting the presence of the pattern in this window). That means, convolution can serve as pattern detector.

2-D Convolution

Input

0	1	2
3	4	5
6	7	8

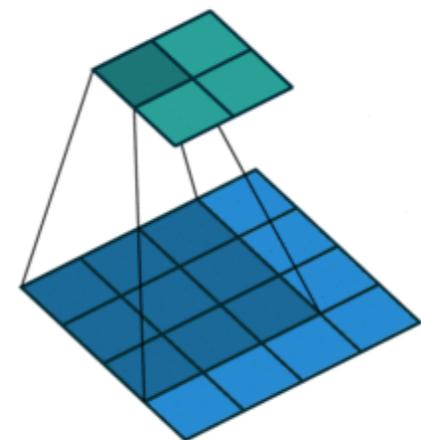
Kernel

0	1
2	3

Output

19	25
37	43

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$



(vduoulin@ Github)

2-D Convolution

Input	Kernel	Output													
<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td></tr><tr><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td><td style="padding: 2px;">5</td></tr><tr><td style="padding: 2px;">6</td><td style="padding: 2px;">7</td><td style="padding: 2px;">8</td></tr></table>	0	1	2	3	4	5	6	7	8	*	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td></tr><tr><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td></tr></table>	0	1	2	3
0	1	2													
3	4	5													
6	7	8													
0	1														
2	3														
	=	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">19</td><td style="padding: 2px;">25</td></tr><tr><td style="padding: 2px;">37</td><td style="padding: 2px;">43</td></tr></table>	19	25	37	43									
19	25														
37	43														

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$

We move the sliding window to the right, and compute the corresponding output entry 25.

2-D Convolution

Input

0	1	2
3	4	5
6	7	8

Kernel

0	1
2	3

Output

19	25
37	43

*

=

$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$

We move the sliding window to the bottom, and compute the corresponding output entry 37.

2-D Convolution

Input	Kernel	Output													
<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td></tr><tr><td style="padding: 2px;">3</td><td style="background-color: #ADD8E6; padding: 2px;">4</td><td style="padding: 2px;">5</td></tr><tr><td style="padding: 2px;">6</td><td style="background-color: #ADD8E6; padding: 2px;">7</td><td style="padding: 2px;">8</td></tr></table>	0	1	2	3	4	5	6	7	8	*	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td></tr><tr><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td></tr></table>	0	1	2	3
0	1	2													
3	4	5													
6	7	8													
0	1														
2	3														
	=	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 2px;">19</td><td style="padding: 2px;">25</td></tr><tr><td style="padding: 2px;">37</td><td style="background-color: #ADD8E6; padding: 2px;">43</td></tr></table>	19	25	37	43									
19	25														
37	43														

$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43$$

We move the sliding window to the right and bottom corner, and compute the corresponding output entry 43.

2-D Convolution Layer

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$$

- $\mathbf{X} : n_h \times n_w$ input matrix
- $\mathbf{W} : k_h \times k_w$ kernel matrix
- $\mathbf{Y} : (n_h - k_h + 1) \times (n_w - k_w + 1)$ output matrix

$$\mathbf{Y} = \mathbf{X} \star \mathbf{W}$$

In general, suppose we have input matrix \mathbf{X} with height n_h and width n_w , and we have kernel matrix \mathbf{W} with height k_h and k_w . Then the output is a matrix of a particular size $(n_h - k_h + 1)$ by $(n_w - k_w + 1)$.

To see this, consider only the width (same reasoning for the height). In the leftmost location, the sliding window's left end aligns with the column 1 of the input. In the rightmost location, the sliding window's right end align with the column n_w of the input, and thus its left end aligns with the column $(n_w - k_w + 1)$ of the input. Therefore, there are in total $(n_w - k_w + 1)$ possible locations for the sliding window. This is also the width of the output matrix.

2-D Convolution Layer

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} + 1 = \begin{array}{|c|c|} \hline 20 & 26 \\ \hline 38 & 44 \\ \hline \end{array}$$

- $\mathbf{X} : n_h \times n_w$ input matrix
- $\mathbf{W} : k_h \times k_w$ kernel matrix
- b : scalar bias
- $\mathbf{Y} : (n_h - k_h + 1) \times (n_w - k_w + 1)$ output matrix

$$\mathbf{Y} = \mathbf{X} \star \mathbf{W} + b$$

- \mathbf{W} and b are learnable parameters

The convolution can also involve a bias scalar. Note that we will add the scalar to each entry of the output matrix.

Examples



(wikipedia)

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



Edge Detection
(Viewed black and white)

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Sharpen

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



Gaussian Blur

Some practical examples of convolution's effect on an image. If we use the first kernel (which puts a weight 8 on the central pixel and subtracts all the surrounding pixels), then the effect is edge detection, ie, the output image highlights the edges.

If we use the second kernel (a smaller weight 5 for the center and subtract the immediate neighboring pixels), then the effect is sharpening the image.

If we use the third kernel which leads to a weighted average of the pixels in the window, the effect is smoothing the image (more precisely, Gaussian blur).

Different kernels can be used for different purposes.

Convolutional Neural Networks

- Strong empirical application performance
- Convolutional networks: neural networks that use convolution in place of general matrix multiplication in at least one of their layers

Advantage: sparse interaction

Fully connected layer, $m \times n$ edges

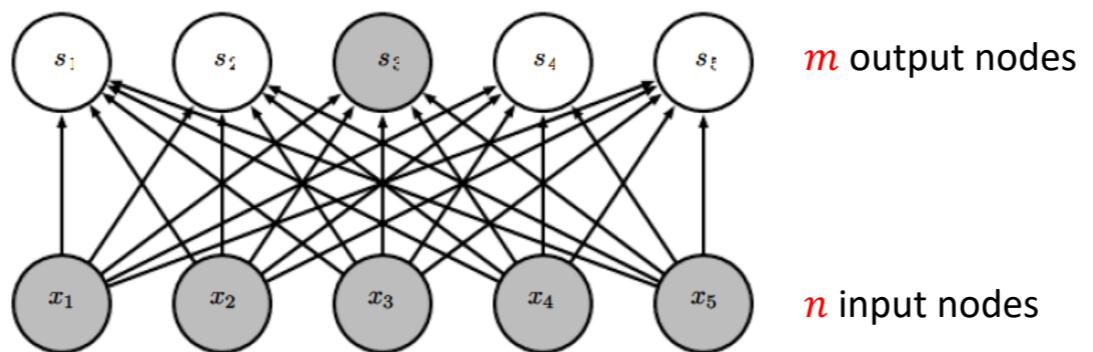


Figure from *Deep Learning*, by Goodfellow, Bengio, and Courville

Convolutions lead to fewer parameters and few computation (=connections in the network).

Fully connected layer has (input size) \times (output size) connections

Advantage: sparse interaction

Convolutional layer, $\leq m \times k$ edges

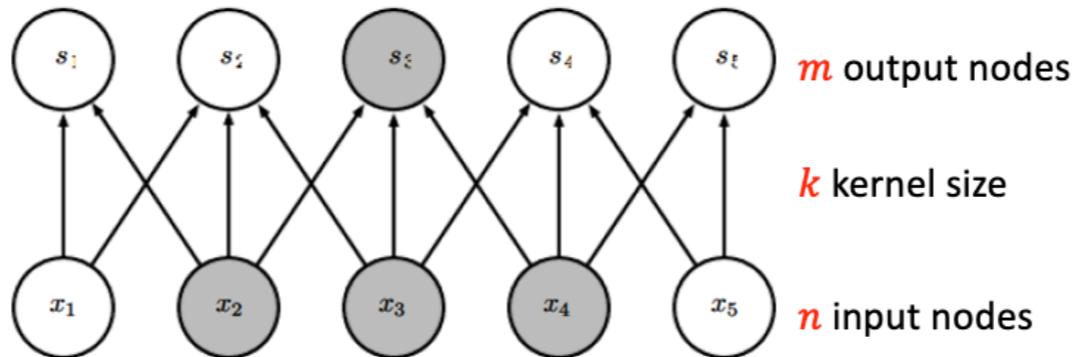
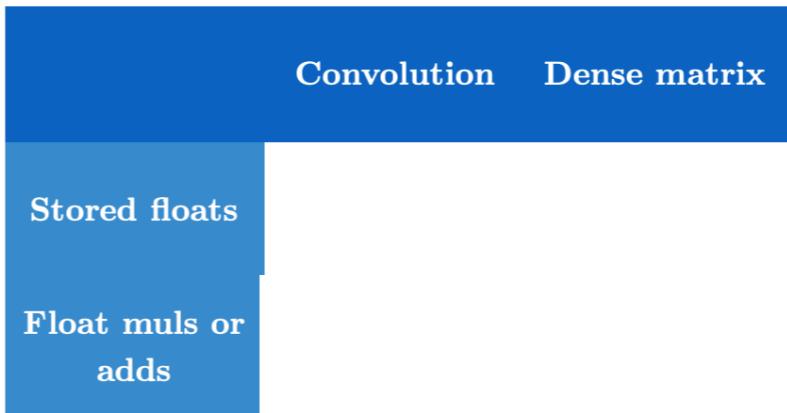


Figure from *Deep Learning*, by Goodfellow, Bengio, and Courville

Convolutional layer has at most (kernel size) x (output size) connections

Efficiency of Convolution

- Input size: 320 x 280
- Kernel Size: 2 x 1
- Output size: 319 x 280



(ignoring bias)

#parameters:

Convolution's is equal to the entries in the kernel which is 2. Fully connected layer's is equal to the number of entries in the weight matrix, which is (input size) X (output size).

#computations:

For convolution, for each output entry: we need to multiple the kernel's entries with the sliding window's and then sum up, and thus #multiplications=kernel_size, #additions=kernel_size -1.

For fully connected layer (also called dense layer), for each output entry: we need to multiple the weight vector with the input and then sum up, and thus #multiplications=input_size, #additions=input_size -1.

Q1. Suppose we want to perform convolution as follows. What's the output?

0	1	2
3	4	5
6	7	8

*

0	1
1	-1

+ 1 = ?

A.

1	2
4	5

B.

1	2
3	4

C.

1	3
3	5

D.

0	1
3	4

Q1. Suppose we want to perform convolution as follows. What's the output?

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 1 & -1 \\ \hline \end{array} + 1 = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array}$$

$$0 \times 0 + 1 \times 1 + 3 \times 1 + 4 \times (-1) + 1 = 1$$

$$1 \times 0 + 2 \times 1 + 4 \times 1 + 5 \times (-1) + 1 = 2$$

$$3 \times 0 + 4 \times 1 + 6 \times 1 + 7 \times (-1) + 1 = 4$$

$$4 \times 0 + 5 \times 1 + 7 \times 1 + 8 \times (-1) + 1 = 5$$

A. $\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array}$

B. $\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$

C. $\begin{array}{|c|c|} \hline 1 & 3 \\ \hline 3 & 5 \\ \hline \end{array}$

D. $\begin{array}{|c|c|} \hline 0 & 1 \\ \hline 3 & 4 \\ \hline \end{array}$

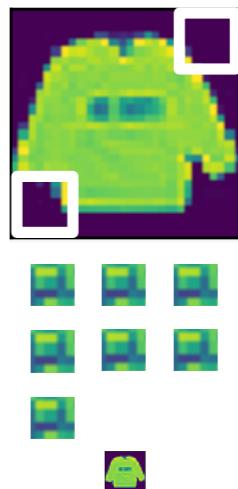
Slide the window to different locations. Don't forget to add the bias to each output entry.



Padding and Stride

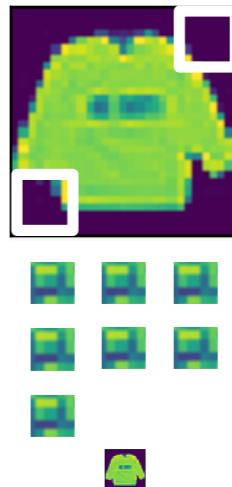
Padding

- Given a 32×32 input image
- Apply convolution with 5×5 kernel
 - 28×28 output with 1 layer
 - 4×4 output with 7 layers



Padding

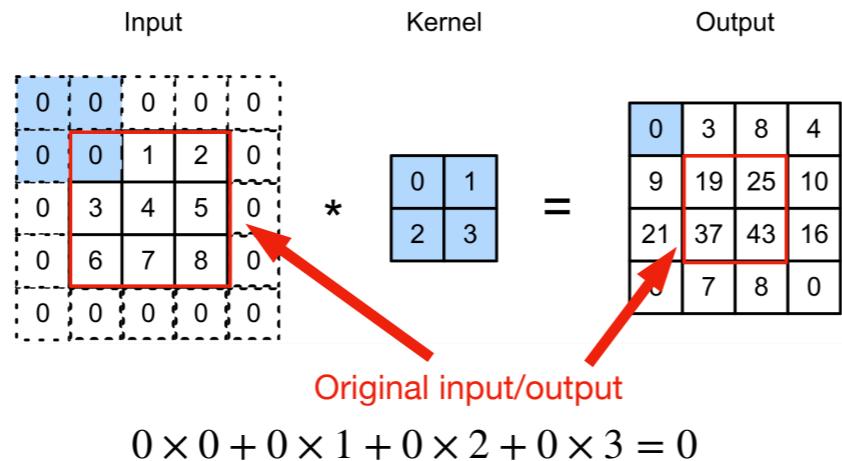
- Given a 32×32 input image
- Apply convolution with 5×5 kernel
 - 28×28 output with 1 layer
 - 4×4 output with 7 layers
- Shape decreases faster with larger kernels
- Shape reduces from $n_h \times n_w$ to
$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$



The issue with 2D conv: each time it can shrink the size of the layer.

Padding

Padding adds rows/columns around input

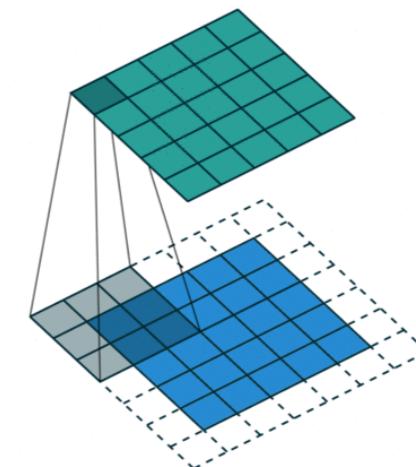


To address the issue of shrinking size, we can pad rows/columns (with only 0's) to the input matrix. Then move the sliding window on the padded input.

Padding

Padding adds rows/columns around input

Input					Kernel		Output					
0	0	0	0	0	*	0	1	=	0	3	8	4
0	0	1	2	0		2	3		9	19	25	10
0	3	4	5	0					21	37	43	16
0	6	7	8	0					6	7	8	0
0	0	0	0	0								



$$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$$

Padding

- Padding p_h rows and p_w columns, output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

If we add p_h rows, then along the height dimension, we have p_h more locations for the sliding window, and thus the output height is increased by p_h . Similar for the width dimension.

Padding

- Padding p_h rows and p_w columns, output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

- A common choice is $p_h = k_h - 1$ and $p_w = k_w - 1$
 - Odd k_h : pad $p_h/2$ on both sides
 - Even k_h : pad $\lceil p_h/2 \rceil$ on top, $\lfloor p_h/2 \rfloor$ on bottom

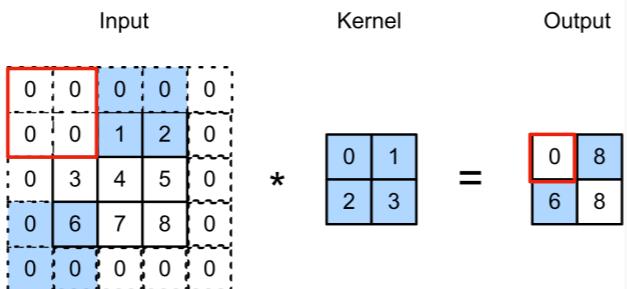
The common choice is to make sure the output shape=input shape.

If k_h is even and $p_h = k_h - 1$ is odd, we can round $p_h/2$ up to the nearest integer (called the ceil function). We can also round it down to the nearest integer (called the floor function). $\text{ceil}(4.5) = 5$ and $\text{floor}(4.5)=4$.

Stride

- Stride is the #rows/#columns per slide

Strides of 3 and 2 for height and width



$$0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$$

$$0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$$

We can move >1 positions each time we move the sliding window.

The example shows a case we have stride 3 for height and stride 2 for width.

Stride

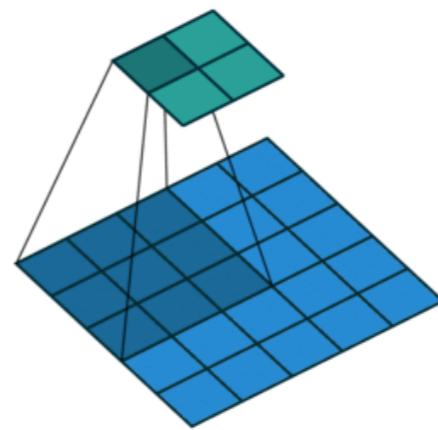
- Stride is the #rows/#columns per slide

Strides of 3 and 2 for height and width

Input					Kernel		Output	
0	0	0	0	0	*	0	1	=
0	0	1	2	0		2	3	
0	3	4	5	0				
0	6	7	8	0				
0	0	0	0	0				

$$0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$$

$$0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$$



Stride

- Given stride s_h for the height and stride s_w for the width, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

In general, suppose we have input matrix X with height n_h and width n_w , and we have kernel matrix W with height k_h and k_w , and we pad p_h rows and p_w columns, and we use stride s_h for height and s_w for width. Then the output is a matrix of a particular size in the expression.

To see this, consider only the width (same reasoning for the height). The padded input has n_w+p_w columns, indexed as column 1 to column (n_w+p_w) .

In the leftmost location, the sliding window's left end aligns with the column 1 of the input.

If we slide it to the right once, the sliding window's left end aligns with the column $1+s_w$ of the input.

If we slide it to the right 2 times, the sliding window's left end aligns with the column $1+2*s_w$ of the input.

...

Suppose the rightmost location is by sliding the original window to the right t times, then the sliding window's left end aligns with the column $1+t*s_w$ of the input.

Then the sliding window's right end aligns with the column $(1+t*s_w + k_w - 1)$ of the input.

We know that $(1+t*s_w + k_w - 1) \leq n_w+p_w$. This means

$$t \leq (n_w - k_w + p_w)/s_w.$$

Note that the total number of possible locations for the sliding window is $(t+1)$:

$$t+1 \leq (n_w - k_w + p_w + s_w)/s_w.$$

It needs to be an integer. So

$$t+1 = \text{floor}((n_w - k_w + p_w + s_w)/s_w).$$

This is just the output width.

Stride

- Given stride s_h for the height and stride s_w for the width, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

- With $p_h = k_h - 1$ and $p_w = k_w - 1$

$$\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$$

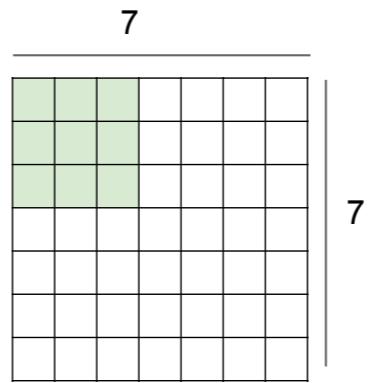
- If input height/width are divisible by strides

$$(n_h/s_h) \times (n_w/s_w)$$

If input height/width are divisible by strides: n_h/s_h is an integer, but $(s_h-1)/s_h$ is smaller than 1, so $\text{floor}(n_h+s_h-1)/s_h = n_h/s_h$. Similar for the width dimension.

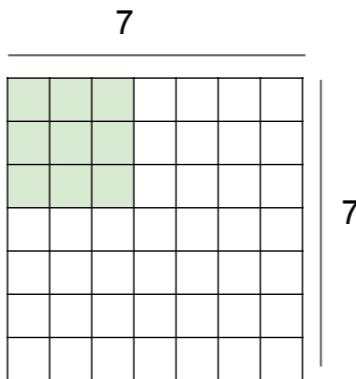
Q2. Suppose we want to perform convolution on a single channel image of size 7×7 (no padding) with a kernel of size 3×3 , and stride = 2. What is the dimension of the output?

- A. 3×3
- B. 7×7
- C. 5×5
- D. 2×2



Q2. Suppose we want to perform convolution on a single channel image of size 7x7 (no padding) with a kernel of size 3x3, and stride = 2. What is the dimension of the output?

- A. 3x3
- B. 7x7
- C. 5x5
- D. 2x2



$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

We can simply apply the math expression. Or for this simple case, we can count the possible locations for the sliding window. For the width, the left end of the window can align with the following columns of the input: column 1, 3, 5. So the output width is 3. Similar for the height.



**Multiple Input and
Output Channels**

Multiple Input Channels

- Color image may have three RGB channels
- Converting to grayscale loses information



Multiple Input Channels

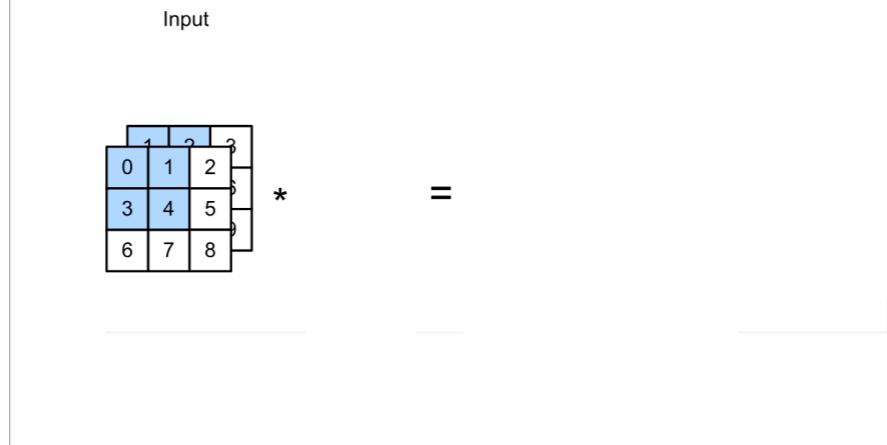
- Color image may have three RGB channels
- Converting to grayscale loses information



We would like to be able to directly handle such multiple-channel inputs.

Multiple Input Channels

- Have a kernel for each channel, and then sum results over channels



For multiple-channel inputs, we will use kernel with the same number of channels (each channel is a 2D kernel matrix, so the overall kernel is 3D).

Put a sliding window across all input channels; for each channel, do the multiplication and sum; then sum up all results of all channels to get one output entry.

Move the sliding window across all input channels simultaneously; do the computation to get different output entries.

Multiple Input Channels

- $\mathbf{X} : c_i \times n_h \times n_w$ input
- $\mathbf{W} : c_i \times k_h \times k_w$ kernel
- $\mathbf{Y} : m_h \times m_w$ output

$$\mathbf{Y} = \sum_{i=0}^{c_i} \mathbf{X}_{i,:,:} \star \mathbf{W}_{i,:,:}$$

In general, the input \mathbf{X} can have c_i input channels. The overall kernel is 3D with the same number of channels. The output shape remains the same; we can think of as: first compute the output matrix of the convolution for each channel, then sum up the output matrices of all channels.

Multiple Output Channels

- No matter how many inputs channels, so far we always get single output channel
- We can have **multiple 3-D kernels**, each one generates an output channel

If we use multiple 3D kernels, then we get multiple output channels: Each 3D kernel leads to one output channel.

Multiple Output Channels

- No matter how many inputs channels, so far we always get single output channel
- We can have **multiple 3-D kernels**, each one generates an output channel
- Input $\mathbf{X} : c_i \times n_h \times n_w$
- Kernels $\mathbf{W} : c_o \times c_i \times k_h \times k_w$
- Output $\mathbf{Y} : c_o \times m_h \times m_w$

$$\mathbf{Y}_{i,:,:} = \mathbf{X} \star \mathbf{W}_{i,:,:,:}$$

for $i = 1, \dots, c_o$

Now we have c_o so many 3D kernels (and the overall kernel is 4D). The number of output channels = the number of 3D kernels.

Multiple Input/Output Channels

- Each 3-D kernel may recognize a particular pattern



(Gabor filters)

In practical applications, when we learn convolutional networks (ie, learn the kernel matrices in the convolutions), the kernel can correspond to different interesting patterns, like the famous Gabor filters (simple geometric patterns like vertical lines, diagonal lines, color dots etc).

Recall that convolution can be viewed as detecting patterns represented by the kernels. So the convolutional networks can learn interesting patterns and store them in the kernels, and use the kernels to detect these patterns in the images to do the computer vision tasks like image classification.

Q3. Suppose we want to perform convolution on a RGB image of size 224x224 (no padding) with 64 kernels, each with height 3 and width 3. Stride = 1. Which is a reasonable estimate of the total number of scalar multiplications involved in this operation (without considering any optimization in matrix multiplication)?

- A. $64 \times 3 \times 3 \times 222 \times 222$
- B. $64 \times 3 \times 3 \times 222$
- C. $3 \times 3 \times 222 \times 222$
- D. $64 \times 3 \times 3 \times 3 \times 222 \times 222$

Q3. Suppose we want to perform convolution on a RGB image of size 224x224 (no padding) with 64 kernels, each with height 3 and width 3. Stride = 1. Which is a reasonable estimate of the total number of scalar multiplications involved in this operation (without considering any optimization in matrix multiplication)?

- A. $64 \times 3 \times 3 \times 222 \times 222$
- B. $64 \times 3 \times 3 \times 222$
- C. $3 \times 3 \times 222 \times 222$
- D. $64 \times 3 \times 3 \times 222 \times 222$

For each kernel, since stride=1, we slide the window to 222x222 different locations. For each location, the number of multiplication is equal to the number of entries in the kernel, which is $3 \times 3 \times 3$ (height x width x channels). So in total $64 \times 3 \times 3 \times 222 \times 222$.

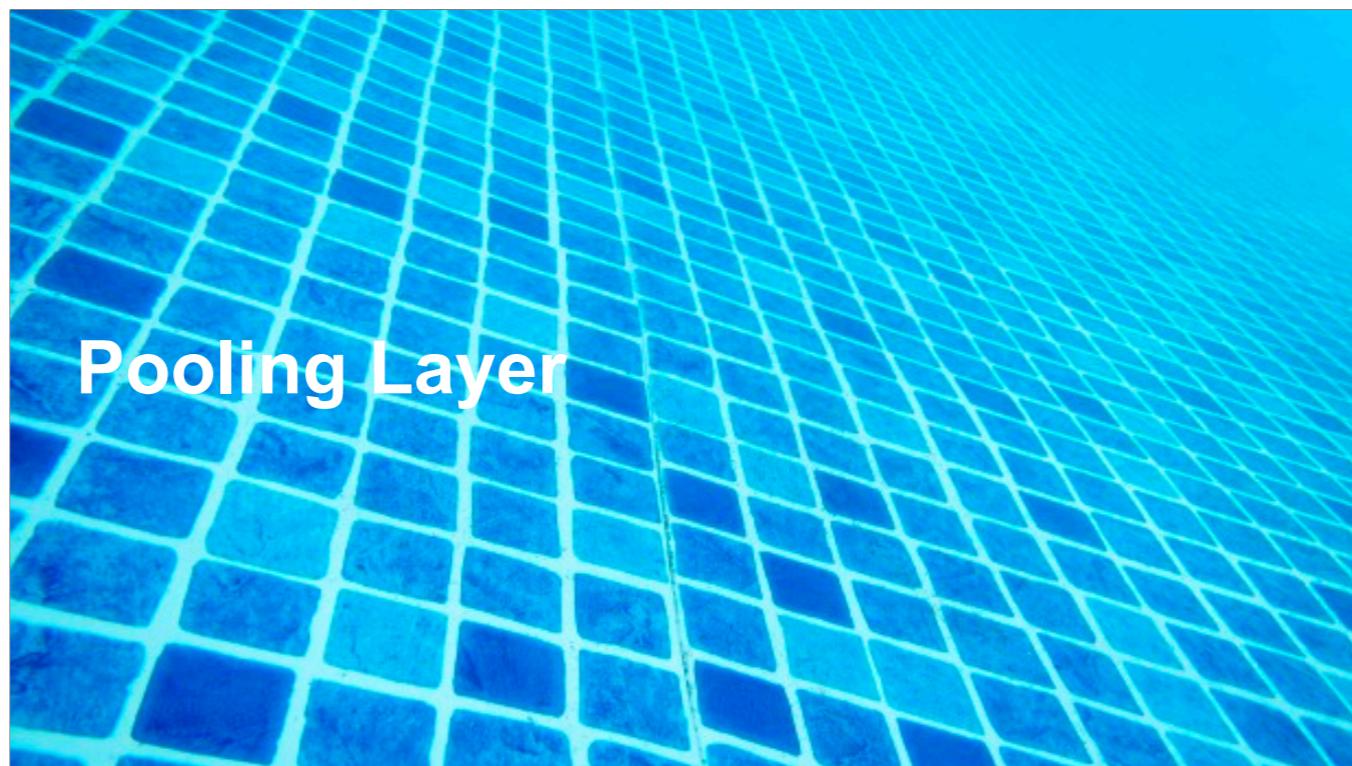
Q4. Suppose we want to perform convolution on a RGB image of size 224x224 (no padding) with 64 kernels, each with height 3 and width 3. Stride = 1. The convolution layer has bias parameters. Which is a reasonable estimate of the total number of learnable parameters?

- A. 64x222x222
- B. 64x3x3x222
- C. 3x3x3x64
- D. $(3 \times 3 \times 3 + 1) \times 64$

Q4. Suppose we want to perform convolution on a RGB image of size 224x224 (no padding) with 64 kernels, each with height 3 and width 3. Stride = 1. The convolution layer has bias parameters. Which is a reasonable estimate of the total number of learnable parameters?

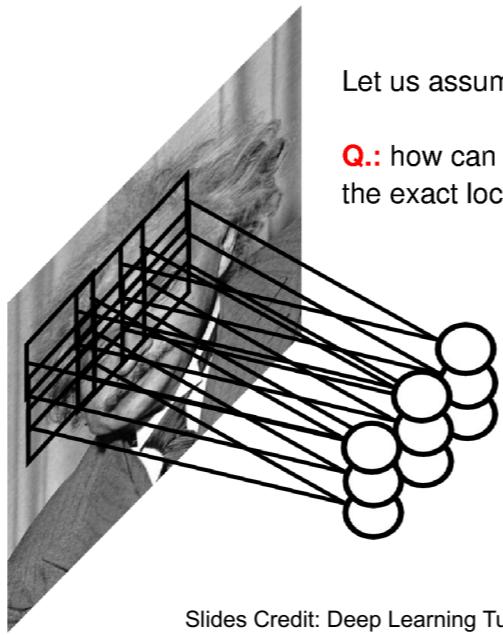
- A. 64x222x222
- B. 64x3x3x222
- C. 3x3x3x64
- D. $(3 \times 3 \times 3 + 1) \times 64$

Each kernel is 3D kernel across 3 input channels, so has $3 \times 3 \times 3$ parameters. Each kernel has 1 bias parameter. So in total $(3 \times 3 \times 3 + 1) \times 64$



Pooling Layer

Pooling



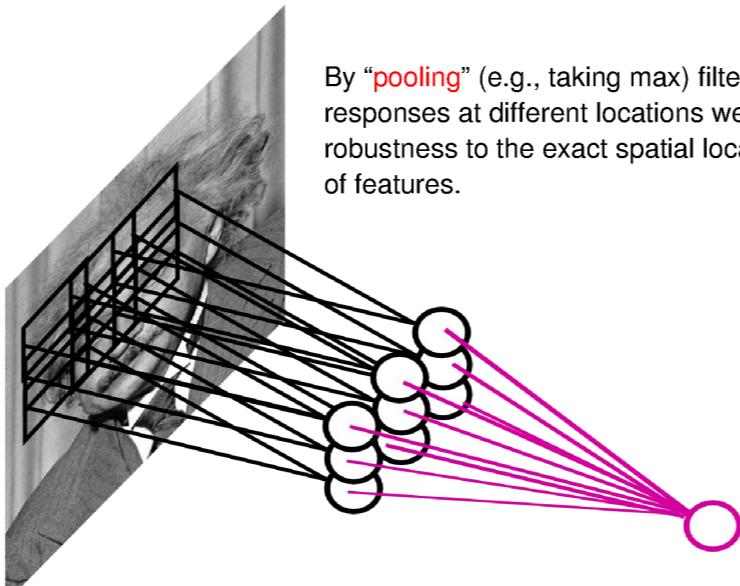
Let us assume filter is an “eye” detector.

Q.: how can we make the detection robust to the exact location of the eye?

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

Recall that the convolution is detecting the pattern represented by the filter/kernel: if the sliding window contains a similar pattern to the filter, the corresponding output entry will be large. If the pattern move to a different location in the image, the large output entry will also move to the corresponding location.

Pooling



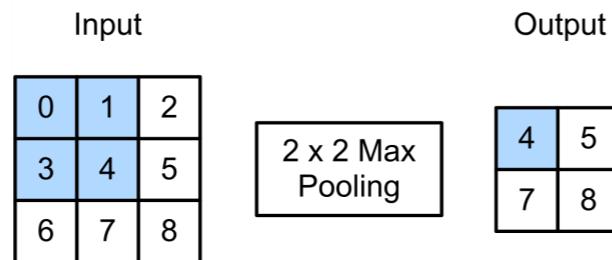
By “**pooling**” (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

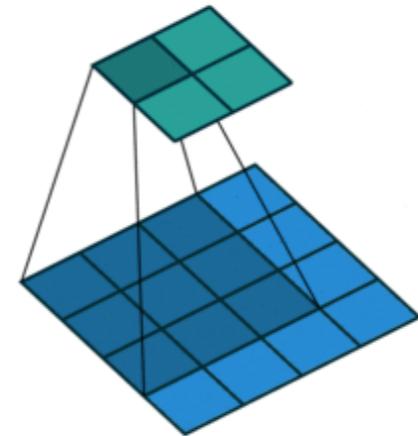
To be robust to the translation of the large output entry (due to the translation of the pattern), we can do MAX over the output entries of the convolution. This is max pooling.

2-D Max Pooling

- Returns the maximal value in the sliding window



$$\max(0,1,3,4) = 4$$



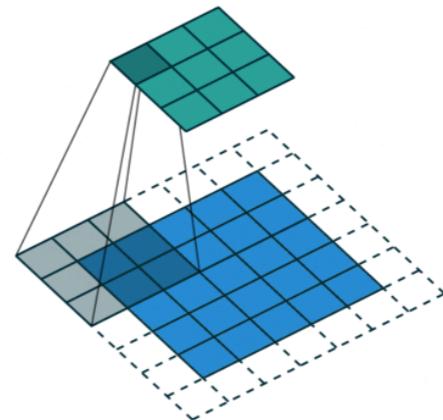
In max pooling we also have a sliding window, but no kernel. The corresponding output entry is simply the max value in the window.

Then we move the sliding window to different locations, and compute the corresponding output entries.

Padding, Stride, and Multiple Channels

- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel

#output channels = #input channels



Because sliding window, the notions of padding/stride/multiple-channels also apply to pooling.

Two key differences of pooling from convolution:

1. No learnable parameters
2. Apply pooling on the input channels individually, no sum across the inputs as in convolution.

Average Pooling

- Max pooling: the strongest pattern signal in a window
- Average pooling: replace max with mean in max pooling
 - The average signal strength in a window

Max pooling



Average pooling



Q5. Suppose we want to perform 2x2 average pooling on the following single channel feature map of size 4x4 (no padding), and stride = 2.
What is the output?

A.

20	30
70	90

B.

16	8
20	25

C.

20	30
20	25

D.

12	2
70	5

12	20	30	0
20	12	2	0
0	70	5	2
8	2	90	3

Q5. Suppose we want to perform 2x2 average pooling on the following single channel feature map of size 4x4 (no padding), and stride = 2.
What is the output?

A.

20	30
70	90

B.

16	8
20	25

C.

20	30
20	25

D.

12	2
70	5

12	20	30	0
20	12	2	0
0	70	5	2
8	2	90	3

We have the 4 possible locations of the sliding window, shown in different colors. Average pooling computes the mean in each window.

Q6. What is the output if we replace average pooling with 2 x 2 max pooling (other settings are the same)?

A.

20	30
70	90

B.

16	8
20	25

C.

20	30
20	25

D.

12	2
70	5

12	20	30	0
20	12	2	0
0	70	5	2
8	2	90	3

Q6. What is the output if we replace average pooling with 2×2 max pooling (other settings are the same)?

A.

20	30
70	90

B.

16	8
20	25

C.

20	30
20	25

D.

12	2
70	5

12	20	30	0
20	12	2	0
0	70	5	2
8	2	90	3

We have the 4 possible locations of the sliding window, shown in different colors. Max pooling computes the max value in each window.

Summary

- Intro of convolutional computations
 - 2D convolution
 - Padding, stride
 - Multiple input and output channels
 - Pooling



Acknowledgement:

Some of the slides in these lectures have been adapted from materials developed by Alex Smola and Mu Li:
<https://courses.d2l.ai/berkeley-stat-157/index.html>