APL 405: Machine Learning for Mechanics

Lecture 15: Convolutional Neural Network

by

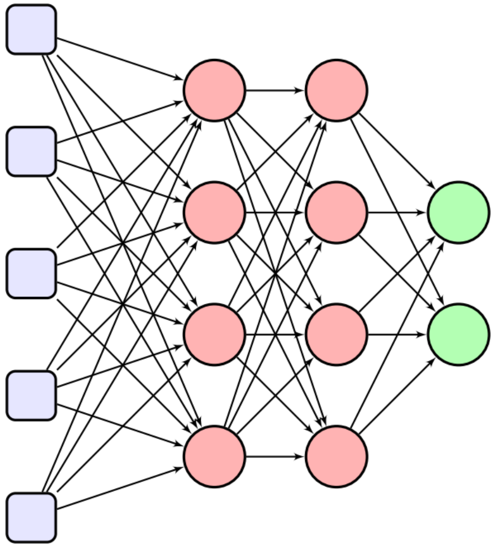
Rajdip Nayek

Assistant Professor,
Applied Mechanics Department,
IIT Delhi

Instructor email: rajdipn@am.iitd.ac.in

Introduction

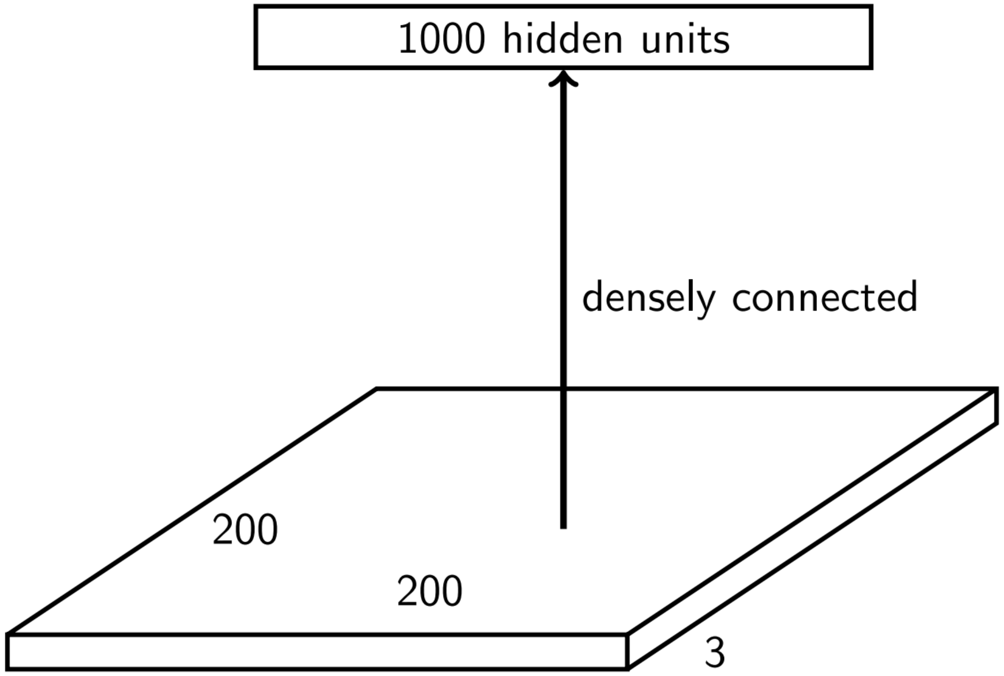
- We looked at **fully connected** neural networks which has each unit of previous layer is connected to all other units of the next layer



- **Drawbacks** of fully connected neural nets:
  - There are a lot of connections. Ex. p units in previous layer, q units in the next layer, then pq connections
  - If we are trying to classify an image, we flatten the 2D image into vectors, which discards the spatial structure/information of the image

- When dealing with images, the nearby pixels are typically related to each other, and we want to exploit this neighbourhood (or local) information to build more efficient neural networks

From fully-connected layers to Convolution layers

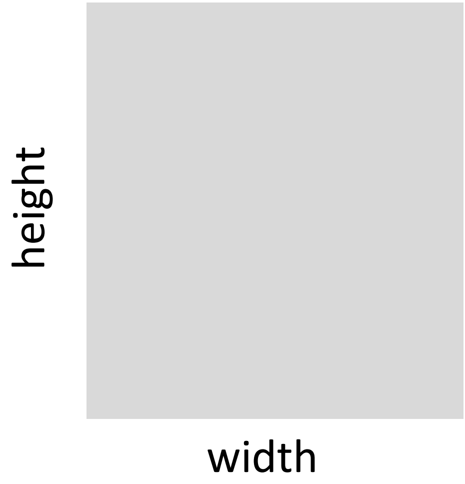


- Suppose we want to train a network (with 1000 FC hidden units) that takes a 200 × 200 colored (RGB) image as input
- What is the problem?
- **Too many parameters**! (Very complex, more chance of overfitting)

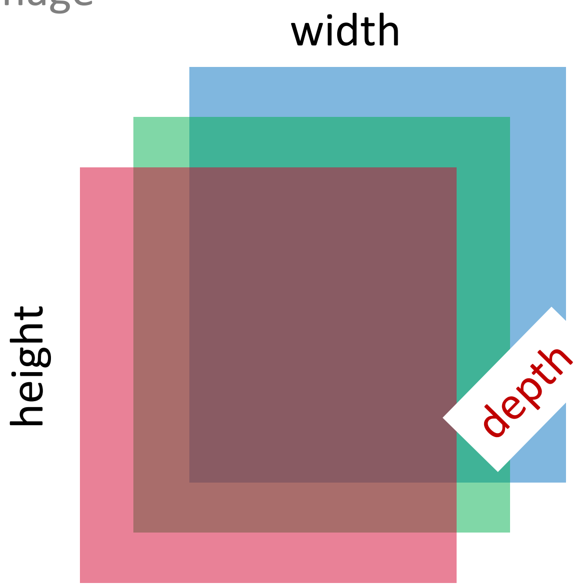
Input size = $200 \times 200 \times 3$= 1,20,000
Parameters = $1{,}20{,}000 \ \times \ 1000$ = 12×10^7

Grayscale vs Colored images

Grayscale Image



Color Image



- Grayscale images have a single channel (depth = 1)
- Colored images have more than one channel (depth > 1)
- Ex. RGB images has **3 channels**

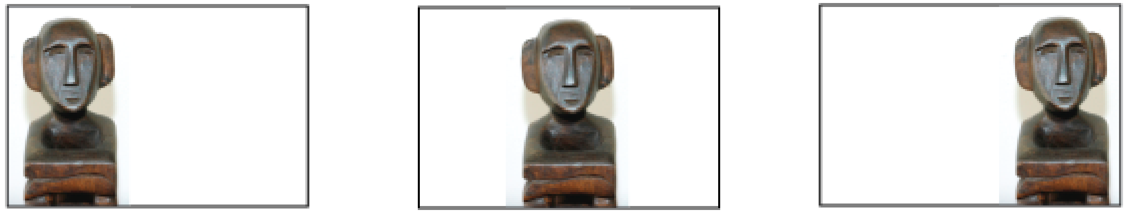
From fully-connected layers to Convolution layers

- Suppose we want to train a network (with 1000 hidden units) that takes a 200×200 RGB image as input

- What is the problem?

- **Too many parameters**! (Very complex, morechance of overfitting)

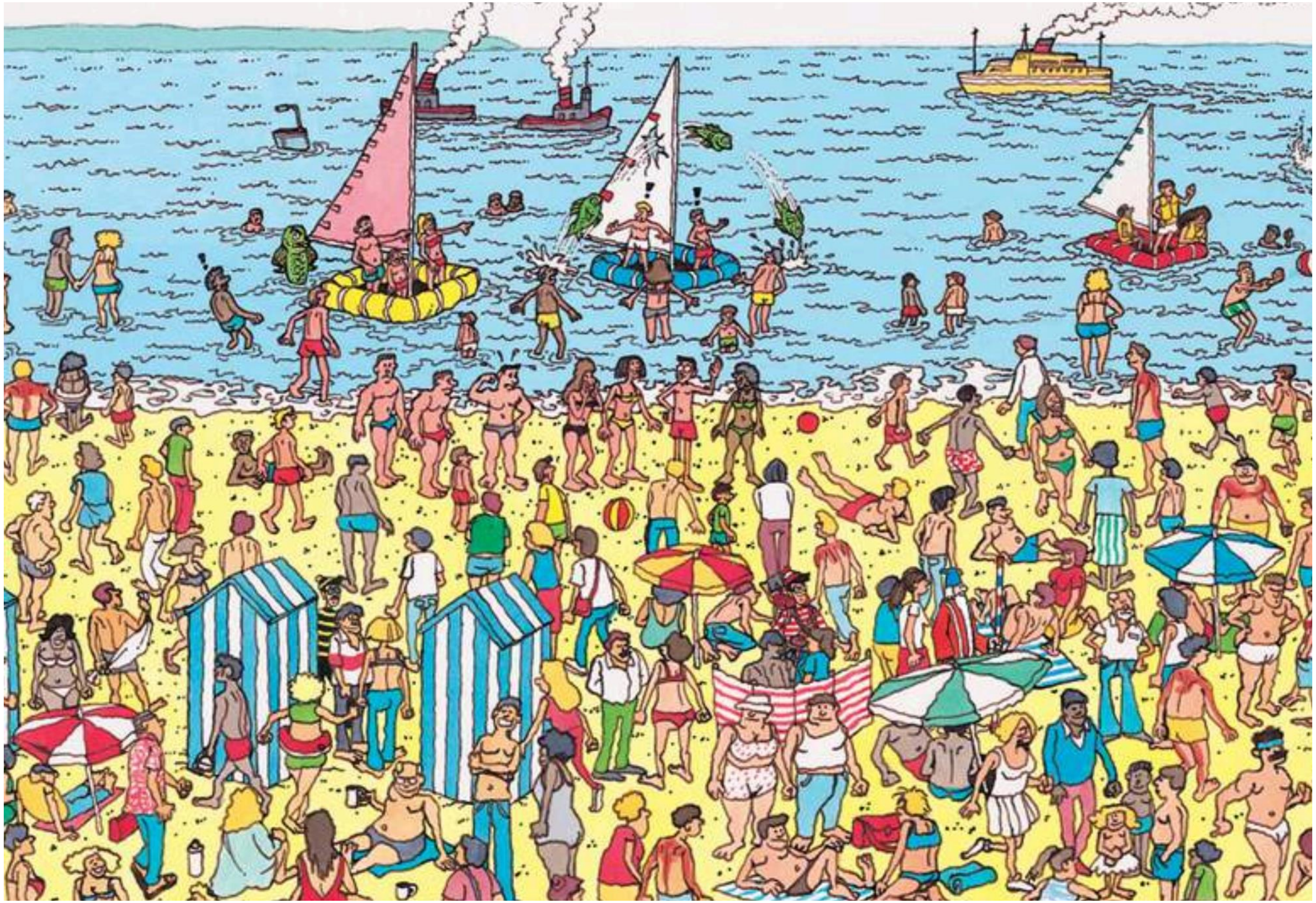
  Input size = $200 \times 200 \times 3$= 1,20,000
  Parameters = $1{,}20{,}000 \times 1000 = 12 \times 10^7$



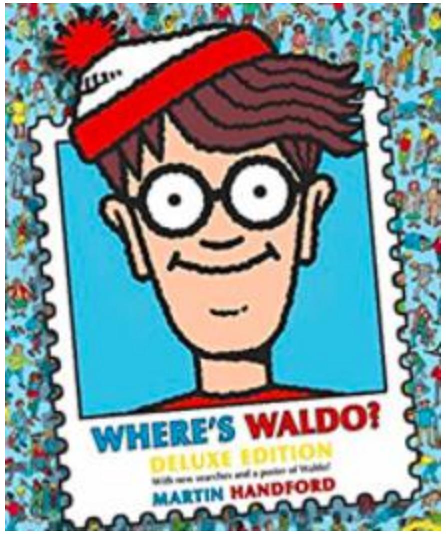
- **Too translation sensitive** – Precise locations of objects in the image matter too much
  - If you translate the objects in the image to different locations, you may have to re-train a fully-connected MLP, else it may fail to classify the outputs correctly

- We can do much better with CNN for images

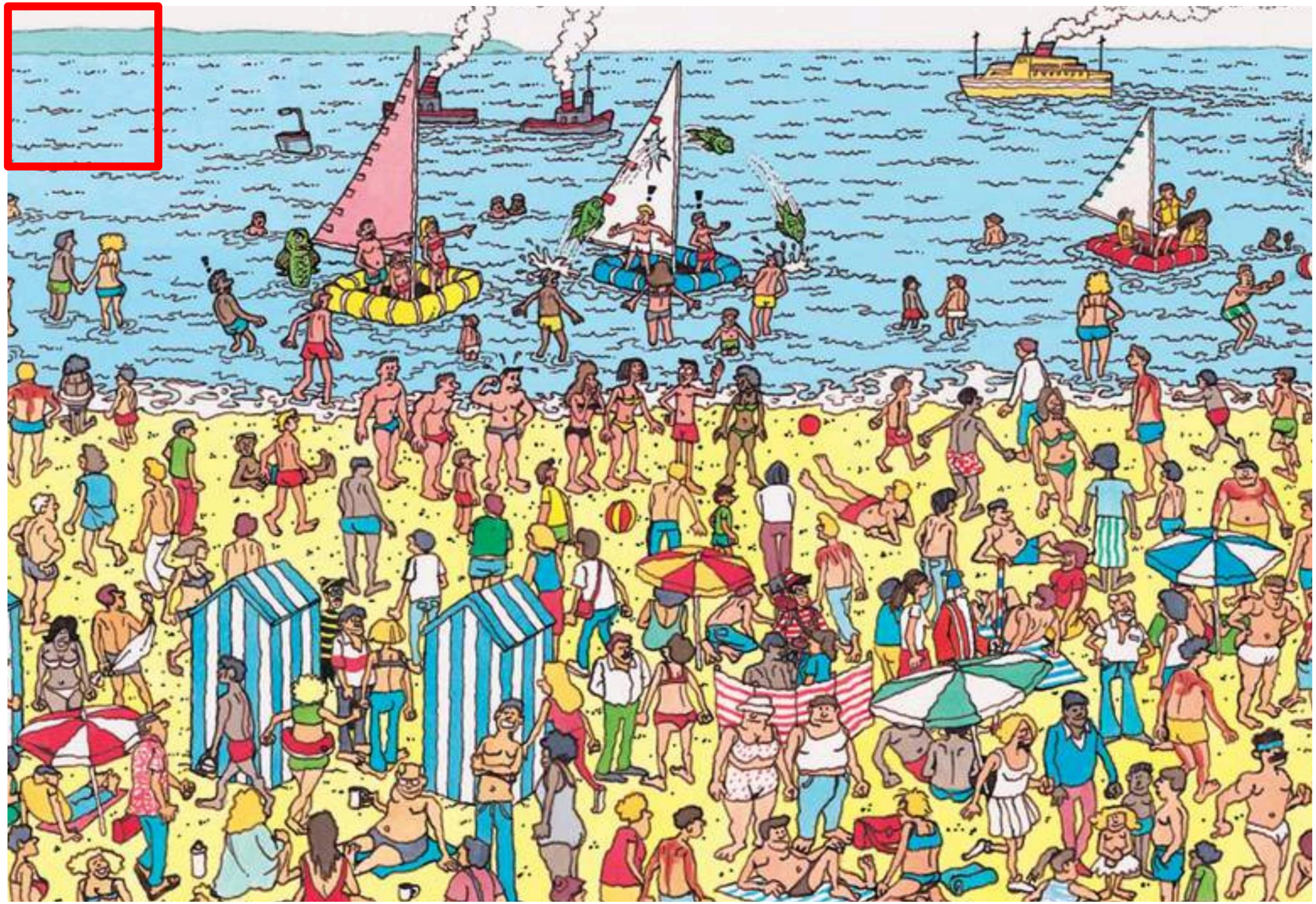
From fully-connected layers to Convolution layers



- "Where's Waldo?"
  In the game, Waldo shows up somewhere in some unlikely location. The reader's goal is to locate him

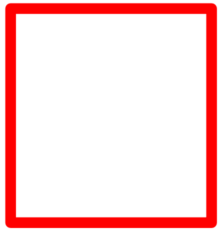


From fully-connected layers to Convolution layers

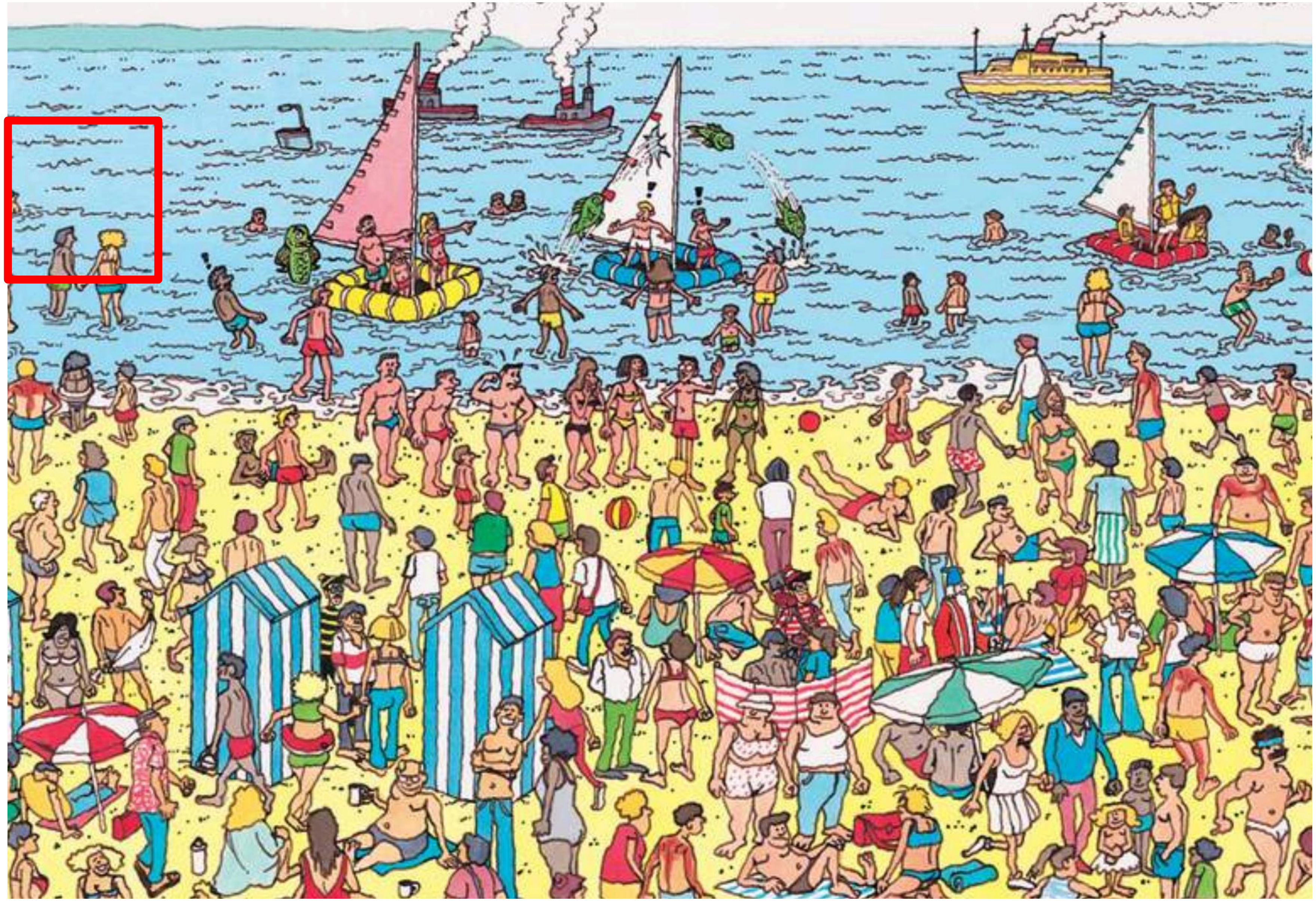


- “Where’s Waldo?”
  In the game, Waldo shows up somewhere in some unlikely location. The reader’s goal is to locate him

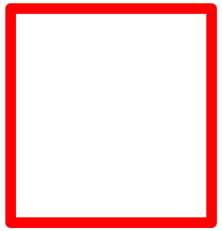
- We could sweep the image with a **Waldo detector** that could assign a score to each patch, indicating how likely the patch contains Waldo



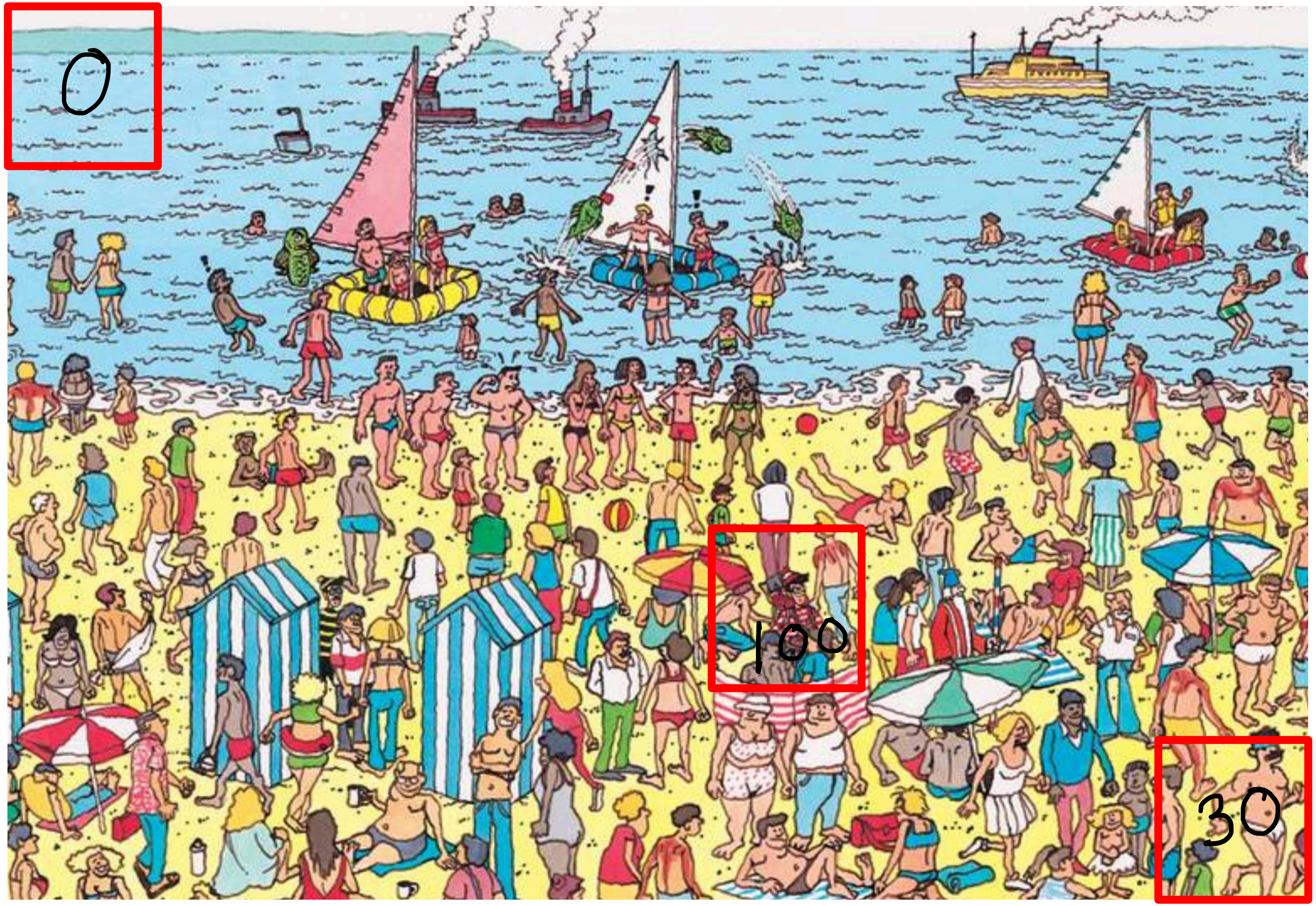
From fully-connected layers to Convolution layers



- "Where's Waldo?"
  In the game, Waldo shows up somewhere in some unlikely location. The reader's goal is to locate him
- We could sweep the image with a **Waldo detector** that could assign a score to each patch, indicating how likely the patch contains Waldo



From fully-connected layers to Convolution layers



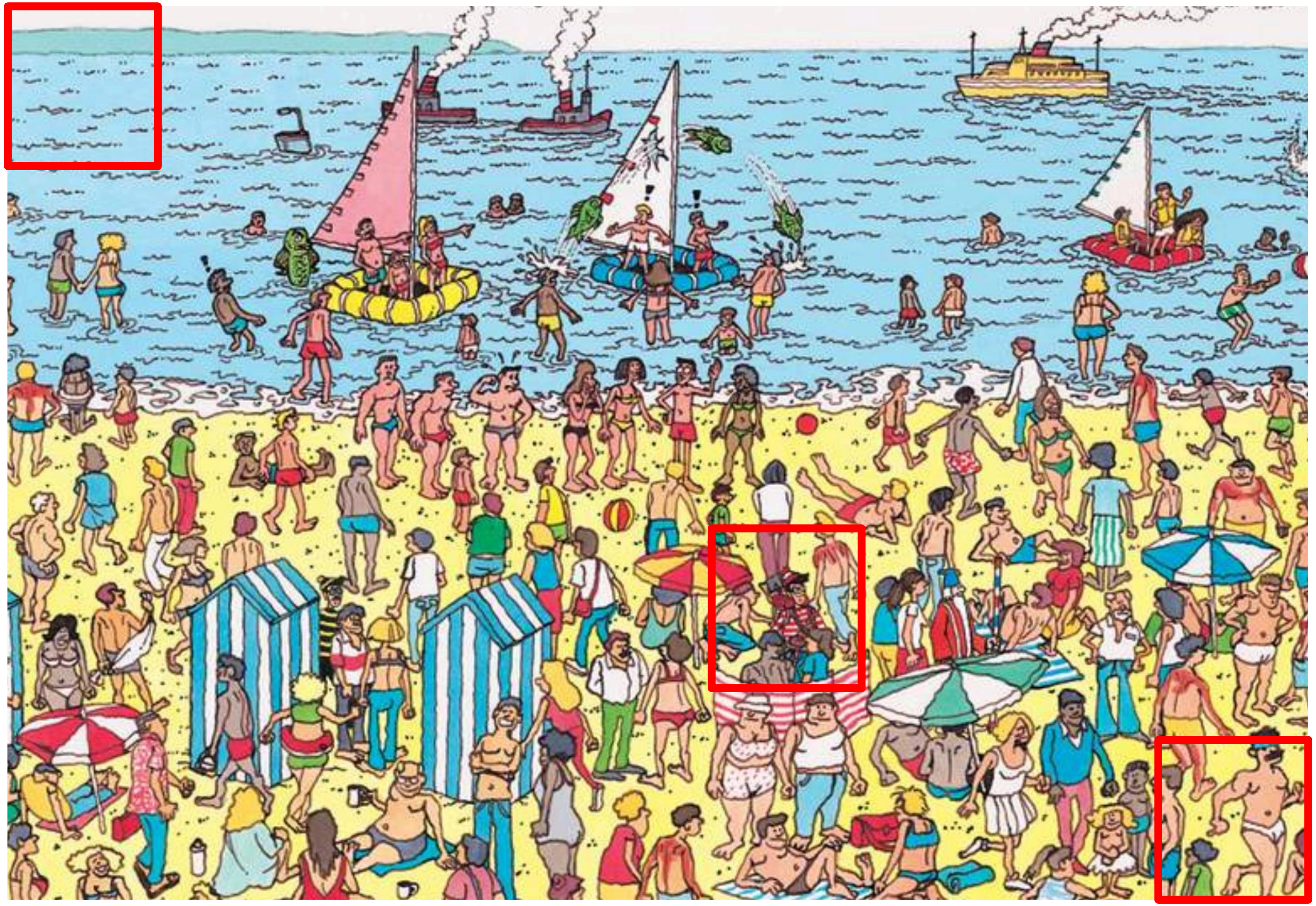
- "Where's Waldo?"
  In the game, Waldo shows up somewhere in some unlikely location. The reader's goal is to locate him

- We could sweep the image with a **Waldo detector** that could assign a score to each patch, indicating how likely the patch contains Waldo

- The patch with maximum score is where Waldo should be located

- As this local patch sweeps the entire image, it does not matter where Waldo is located

From fully-connected layers to Convolution layers

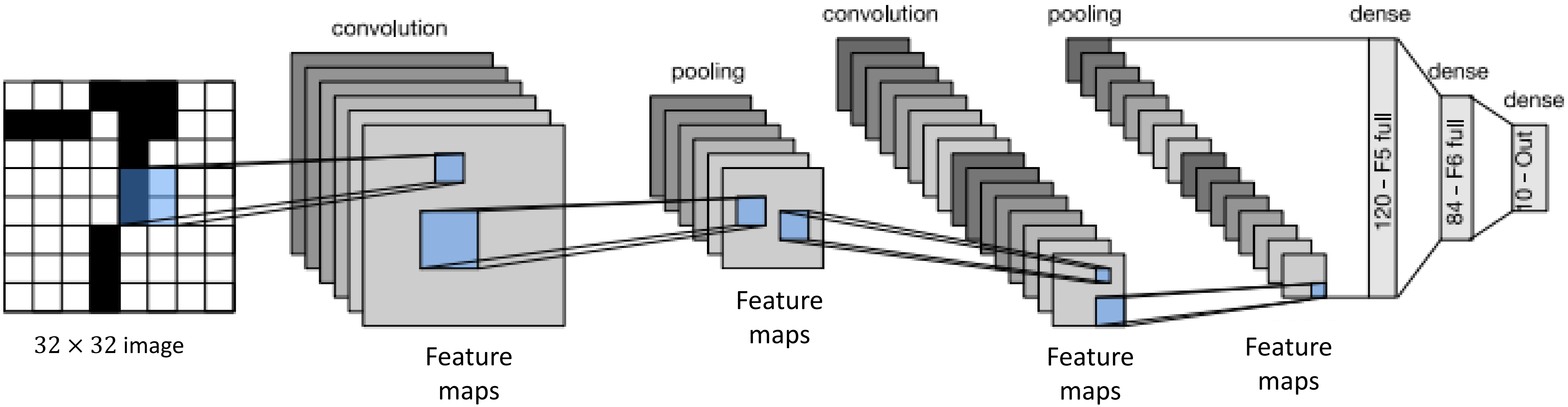


- "Where's Waldo?"
  In the game, Waldo shows up somewhere in some unlikely location. The reader's goal is to locate him

- We could sweep the image with a **Waldo detector** that could assign a score to each patch, indicating how likely the patch contains Waldo

- CNNs systematize this idea of **translation invariance** and **localised feature detection**, via convolutions and max pooling, with much less parameters

From fully-connected layers to Convolution layers



- CNNs systematize this idea of **translation invariance** and **localised feature detection**, via convolutions and max pooling, with much less parameters
- CNNs uses multiple kernels (“Waldo detectors”) that detects different features

What is convolution?

1D Convolution

- Convolution of two scalar-valued functions $w(x)$ and $g(x)$ is defined as: $s(x) = (w * g)(x) = \int w(x-a)\, g(a)\, da$

- Whenever we have discrete objects (arrays), the integral turns into a sum: $s[i] = \sum_a w[i-a]\, g[a]$

$w[0]$	$w[1]$	$w[2]$	$w[3]$		$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$		
0.1	**0.2**	**0.3**	**0.4**	$*$	**1**	**2**	**3**	**4**	**5**	**6**	=	?

- The array $\boldsymbol{g}$ is the input
- The array $\boldsymbol{w}$ is called the **filter (or kernel)**

1D Convolution

- Convolution of two scalar-valued functions $w(x)$ and $g(x)$ is defined as: $s(x) = (w * g)(x) = \int w(x-a)\, g(a)\, da$
- Whenever we have discrete objects (arrays), the integral turns into a sum: $s[i] = \sum_a w[i-a]\, g[a]$

$w[0]$	$w[1]$	$w[2]$	$w[3]$
0.1	**0.2**	**0.3**	**0.4**

$*$

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

$=$?

- The array $\boldsymbol{g}$ is the input
- The array $\boldsymbol{w}$ is called the **filter (or kernel)**
- **Flip-and-filter**
  - Slide the filter over the input and compute windowed dot product

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

$w[3]$	$w[2]$	$w[1]$	$w[0]$
0.4	**0.3**	**0.2**	**0.1**

(flipped)

$$s[3] = w[3]g[0] + w[2]g[1] + w[1]g[2] + w[0]g[3]$$

$s[3]$	$s[4]$	$s[5]$
2		

Convolution in 1D

- Convolution of two scalar-valued functions $w(x)$ and $g(x)$ is defined as: $s(x) = (w * g)(x) = \int w(x-a)\, g(a)\, da$
- Whenever we have discrete objects (arrays), the integral turns into a sum: $s[i] = \sum_a w[i-a]\, g[a]$

$w[0]$	$w[1]$	$w[2]$	$w[3]$
0.1	**0.2**	**0.3**	**0.4**

$*$

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

$=$?

- The array $\boldsymbol{g}$ is the input
- The array $\boldsymbol{w}$ is called the **filter (or kernel)**
- **Flip-and-filter**
  - Slide the filter over the input and compute windowed dot product

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

$w[3]$	$w[2]$	$w[1]$	$w[0]$
0.4	**0.3**	**0.2**	**0.1**

$s[3]$	$s[4]$	$s[5]$
2	**4**	

$s[4] = w[3]g[1] + w[2]g[2] + w[1]g[3] + w[0]g[4]$

1D Convolution

- Convolution of two scalar-valued functions $w(x)$ and $g(x)$ is defined as: $s(x) = (w * g)(x) = \int w(x-a)\, g(a)\, da$

- Whenever we have discrete objects (arrays), the integral turns into a sum: $s[i] = \sum_a w[i-a]\, g[a]$

$w[0]$	$w[1]$	$w[2]$	$w[3]$
0.1	**0.2**	**0.3**	**0.4**

$*$

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

= ?

- The array $\boldsymbol{g}$ is the input
- The array $\boldsymbol{w}$ is called the **filter (or kernel)**
- **Flip-and-filter**
  - Slide the filter over the input and compute windowed dot product
- Here the input (and the kernel) is 1D

$g[0]$	$g[1]$	$g[2]$	$g[3]$	$g[4]$	$g[5]$
1	**2**	**3**	**4**	**5**	**6**

$w[3]$	$w[2]$	$w[1]$	$w[0]$
0.4	**0.3**	**0.2**	**0.1**

$s[3]$	$s[4]$	$s[5]$
2	**4**	**5**

$$s[5] = w[3]g[2] + w[2]g[3] + w[1]g[4] + w[0]g[5]$$

1D Convolution to 2D Convolution

- Convolution is more like doing a **flipped cross-correlation** operation
- The **filters (or kernels)** will resemble the weights in CNN (as we will see soon)
- Most machine learning libraries just implement a moving window cross-correlation (and *ignore flipping*) since it does not matter much whether you learn a flipped set of weights or unflipped set of weights
- How does convolution (think more like cross-correlation) look in 2D?
- Let's now consider 2D grayscale images (has depth of 1) and 2D kernels

Input **Kernel** **Output**

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

1D Convolution to 2D Convolution

- Convolution is more like doing a **flipped cross-correlation** operation
- The **filters (or kernels)** will resemble the weights in CNN (as we will see soon)
- Most machine learning libraries just implement a moving window cross-correlation (and *ignore flipping*) since it does not matter much whether you learn a flipped set of weights or unflipped set of weights
- How does convolution (think more like cross-correlation) look in 2D?
- Let's now consider 2D grayscale images (has depth of 1) and 2D kernels

Input

0	1	2
3	4	5
6	7	8

*

Kernel

0	1
2	3

=

Output

19	

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

2D Convolution

- Convolution is more like a moving window **flipped cross-correlation** operation
- The **filters (or kernels)** will resemble the weights in CNN (as we will see soon)
- Most machine learning libraries just implement a moving window cross-correlation (and *ignore flipping*) since it does not matter much whether you learn a flipped set of weights or unflipped set of weights
- How does convolution (think cross-correlation from now on) look in 2D?
- Let’s now consider 2D grayscale images (has depth of 1) and 2D kernels

Input

0	1	2
3	4	5
6	7	8

*

Kernel

0	1
2	3

=

Output

19	25

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

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

2D Convolution

- Convolution is more like a moving window **flipped cross-correlation** operation
- The **filters (or kernels)** will resemble the weights in CNN (as we will see soon)
- Most machine learning libraries just implement a moving window cross-correlation (and *ignore flipping*) since it does not matter much whether you learn a flipped set of weights or unflipped set of weights
- How does convolution (think cross-correlation from now on) look in 2D?
- Let’s now consider 2D grayscale images (has depth of 1) and 2D kernels

Input **Kernel** **Output**

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 19 & 25 \\ 37 & \end{bmatrix}$$

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$

2D Convolution

- Convolution is more like a moving window **flipped cross-correlation** operation
- The **filters (or kernels)** will resemble the weights in CNN (as we will see soon)
- Most machine learning libraries just implement a moving window cross-correlation (and *ignore flipping*) since it does not matter much whether you learn a flipped set of weights or unflipped set of weights
- How does convolution (think cross-correlation from now on) look in 2D?
- Let's now consider 2D grayscale images (has depth of 1) and 2D kernels

Input

0	1	2
3	4	5
6	7	8

*

Kernel

0	1
2	3

=

Output

19	25
37	43

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43$$

2D Convolution

- Despite the simplicity of the operation, convolution can do some pretty interesting things

- Let’s look at some examples of convolutions with grayscale images

$*$

1	1	1
1	1	1
1	1	1

$=$

Blur kernel: Takes an average of all the neigbouring pixels

2D Convolution

- Despite the simplicity of the operation, convolution can do some pretty interesting things
- Let's look at some examples of convolutions with grayscale images

*

0	-1	0
-1	5	-1
0	-1	0

=

Sharpen kernel: Emphasizes differences in adjacent pixel values

2D Convolution

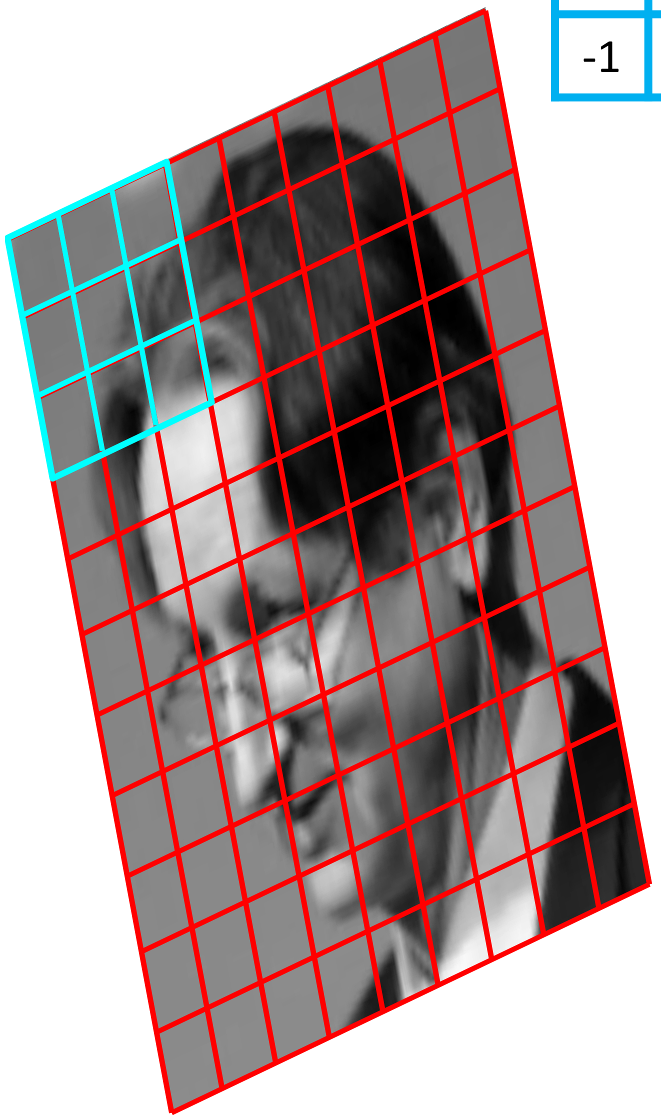
- Despite the simplicity of the operation, convolution can do some pretty interesting things
- Let's look at some examples of convolutions with grayscale images

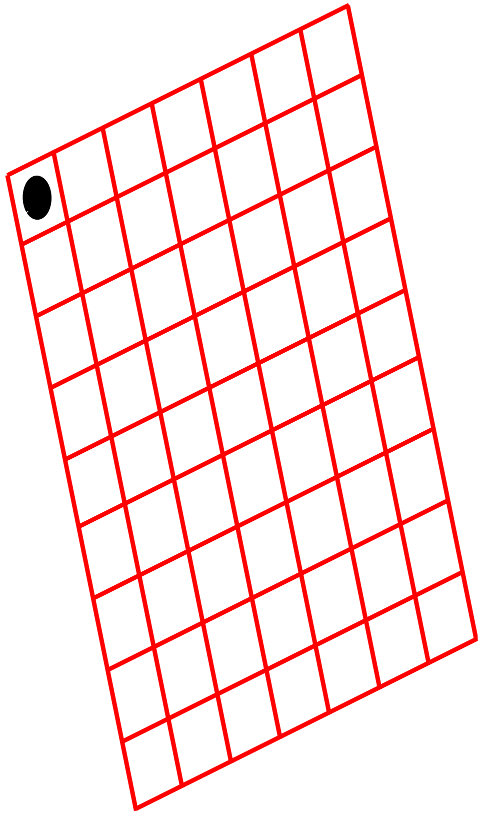
$$* \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} =$$

Vertical edge detector: Finds edges from darker to brighter intensities

2D Convolution

-1	0	1
-2	0	2
-1	0	1

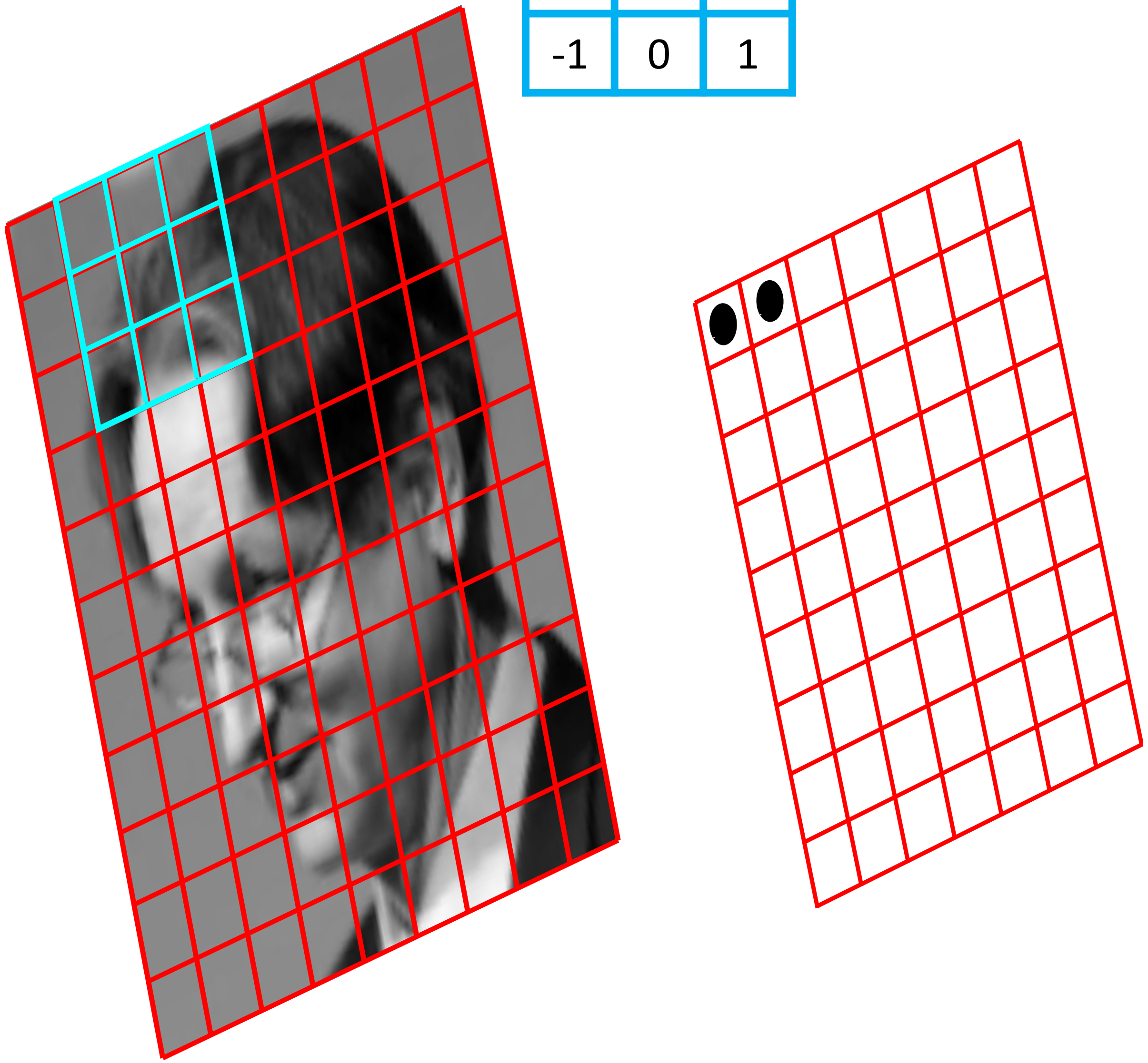




- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

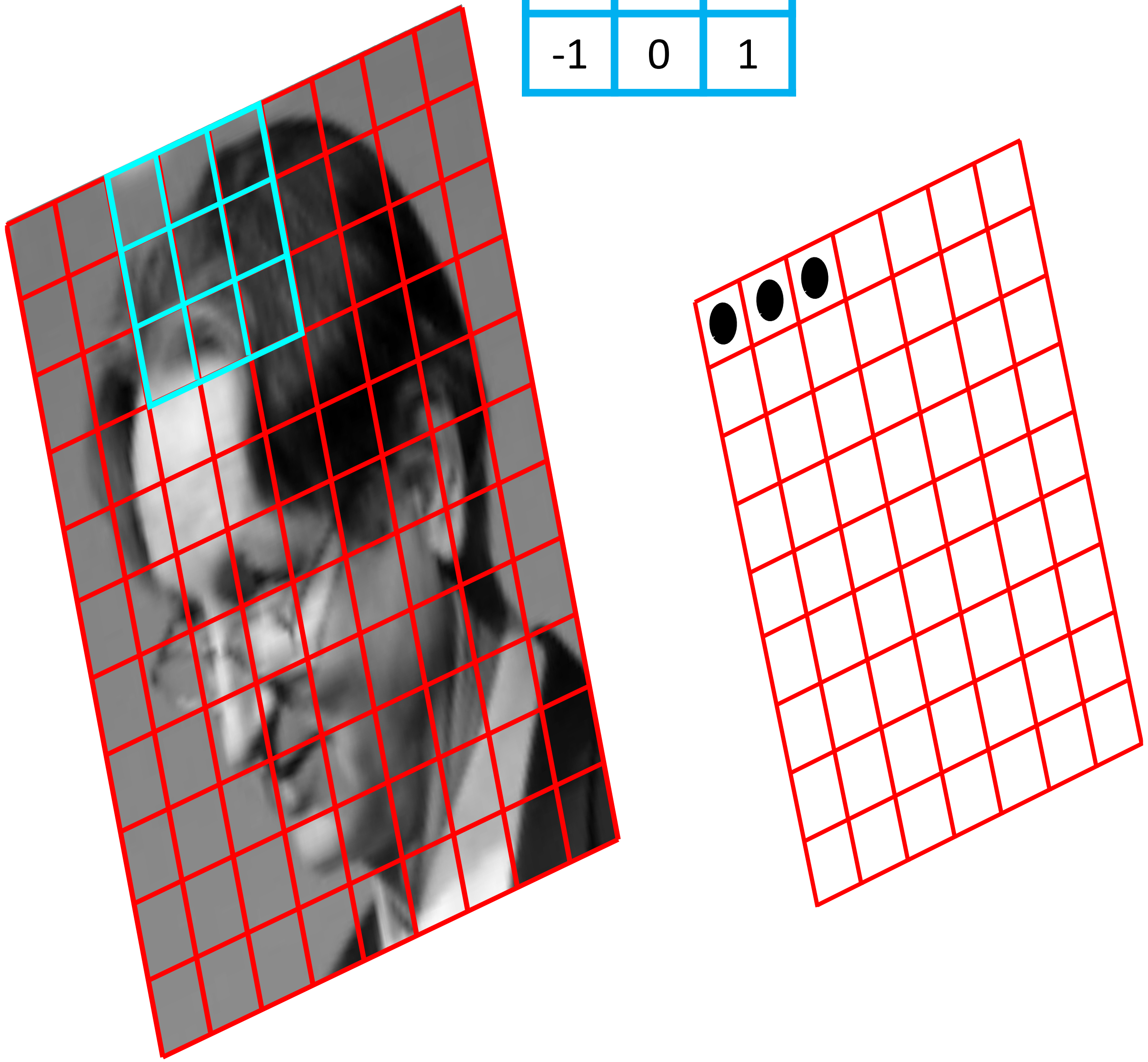
-1	0	1
-2	0	2
-1	0	1



- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

-1	0	1
-2	0	2
-1	0	1

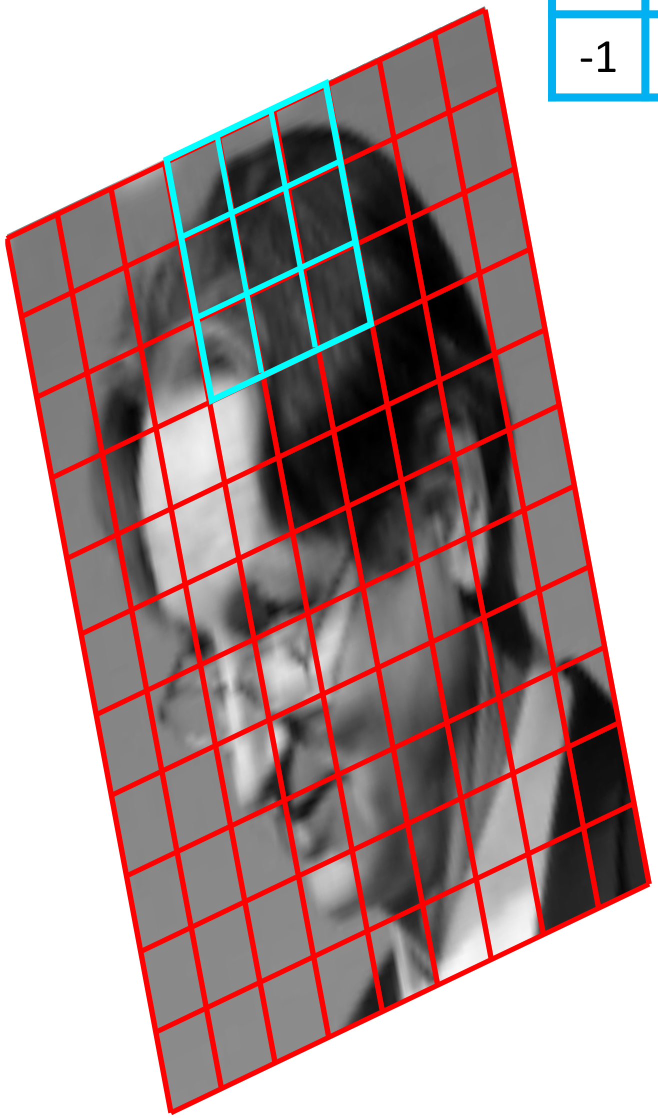


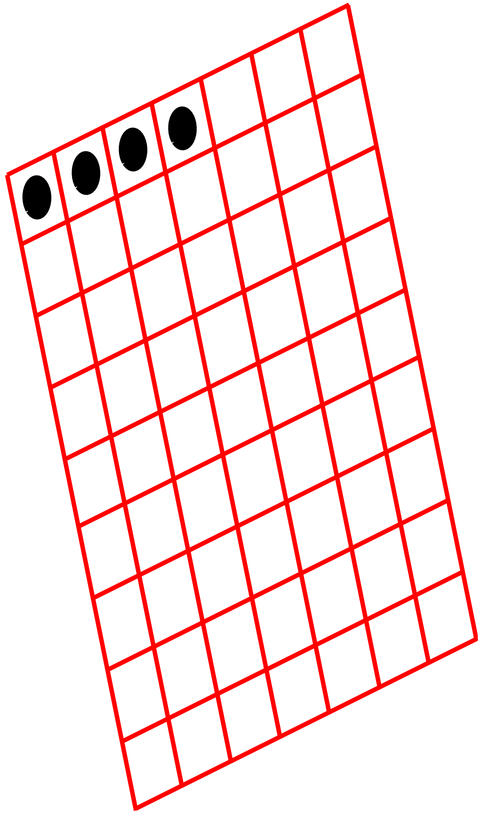
- We just slide the kernel over the input image

- Each time we slide the kernel we get one value in the output

2D Convolution

-1	0	1
-2	0	2
-1	0	1

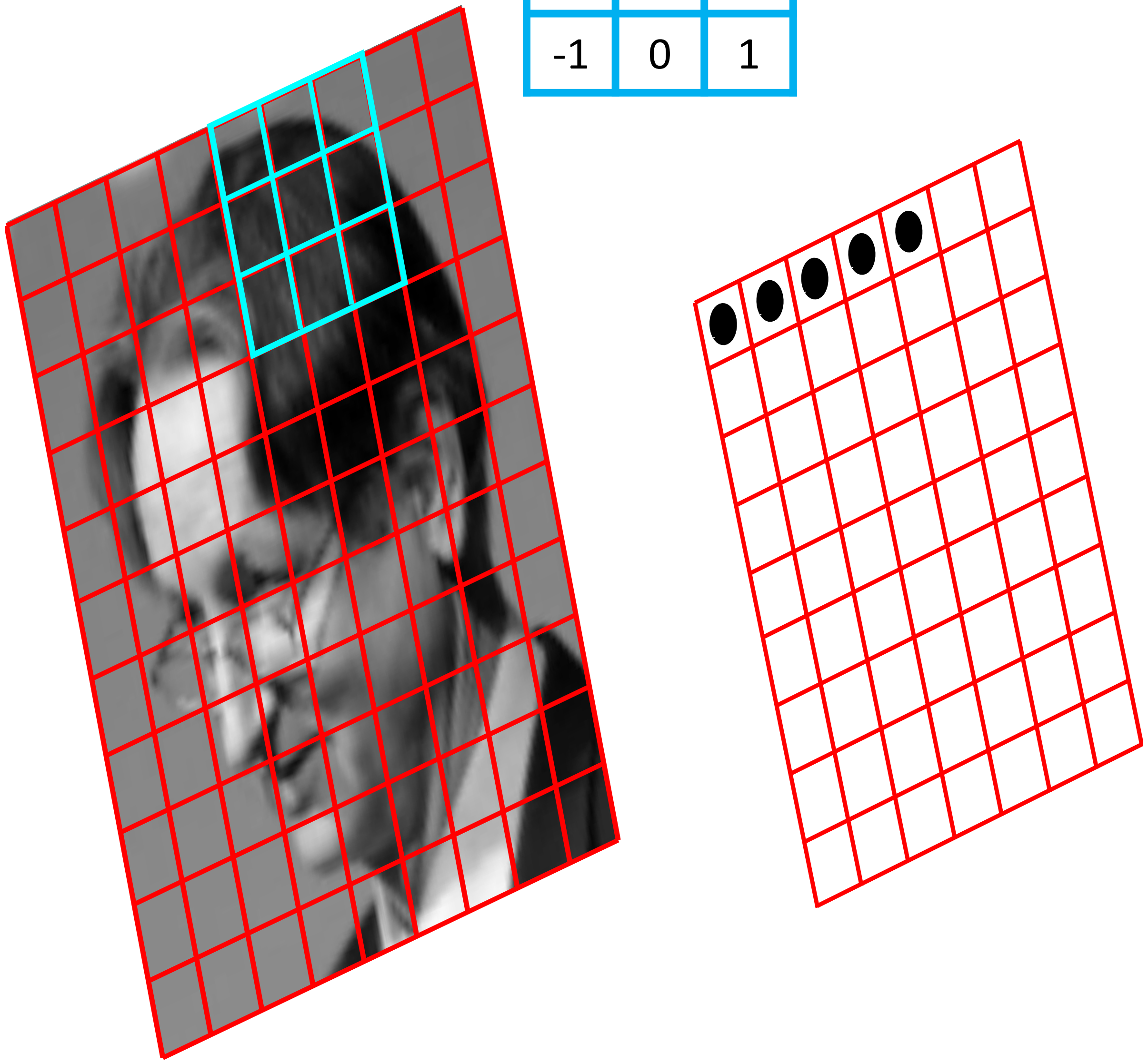




- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

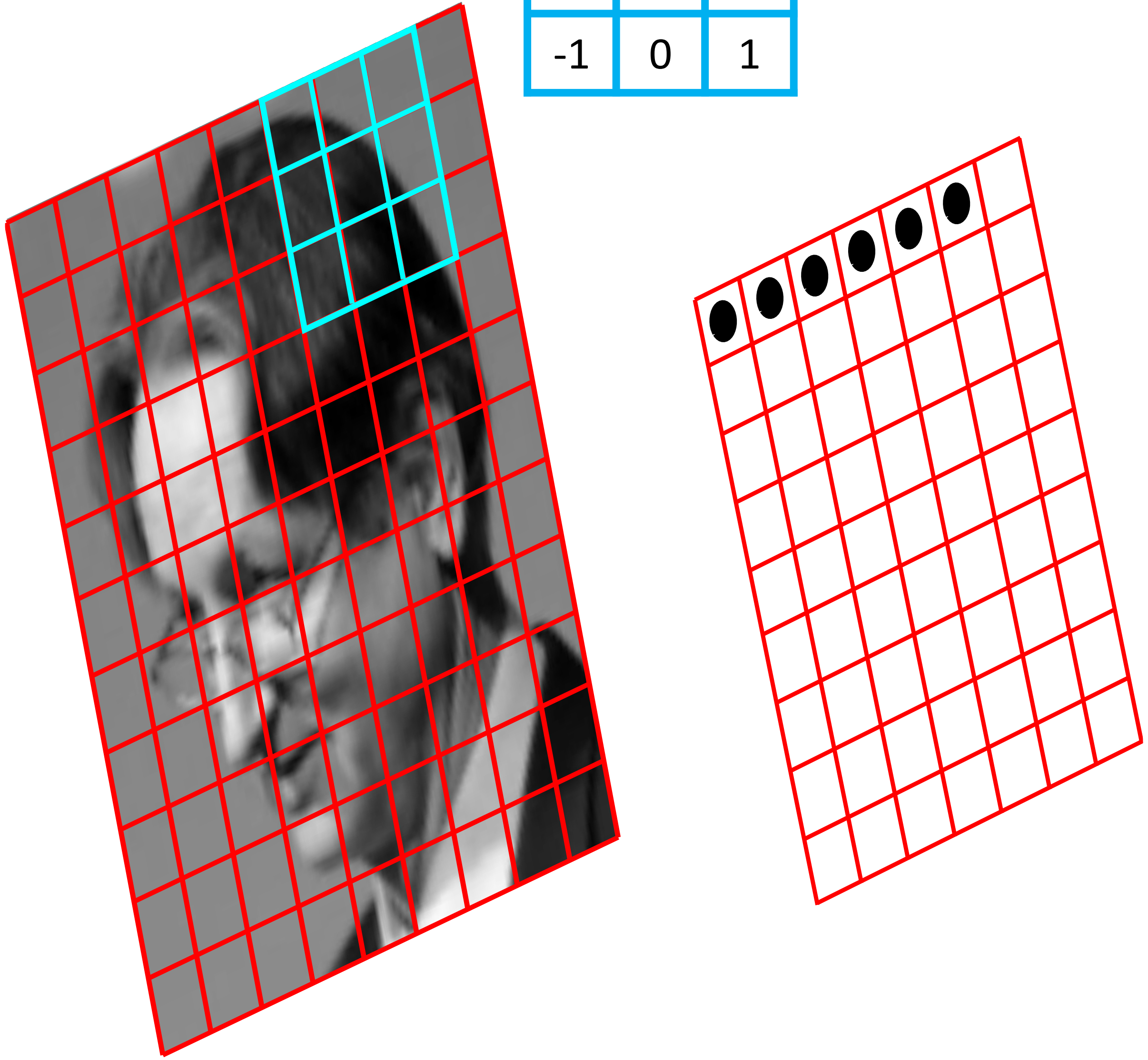
-1	0	1
-2	0	2
-1	0	1



- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

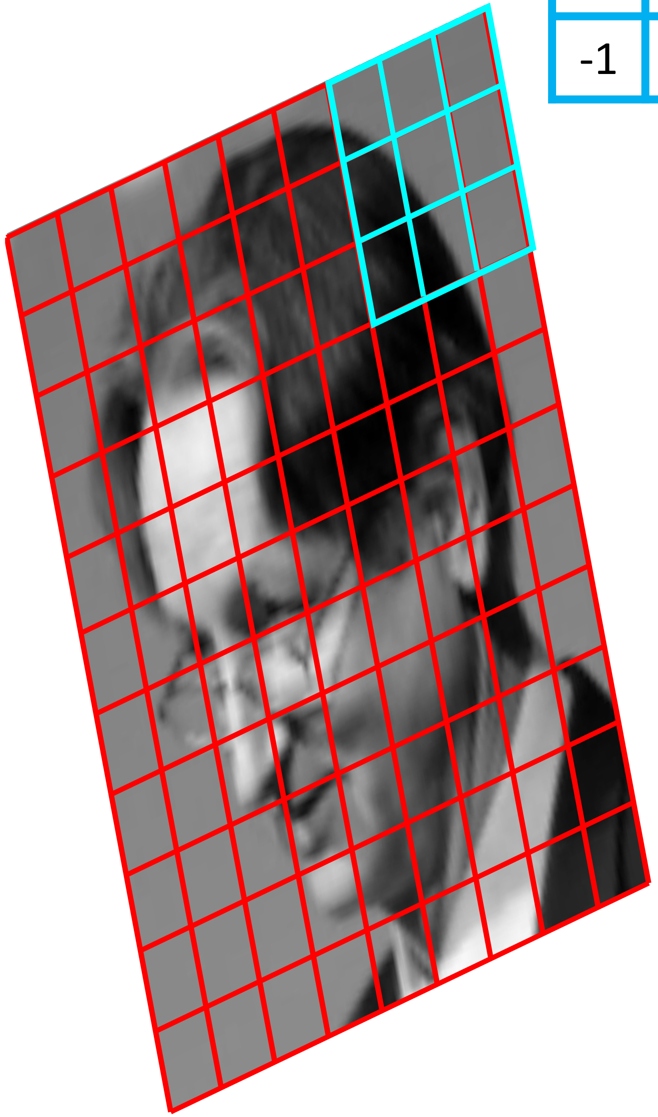
-1	0	1
-2	0	2
-1	0	1

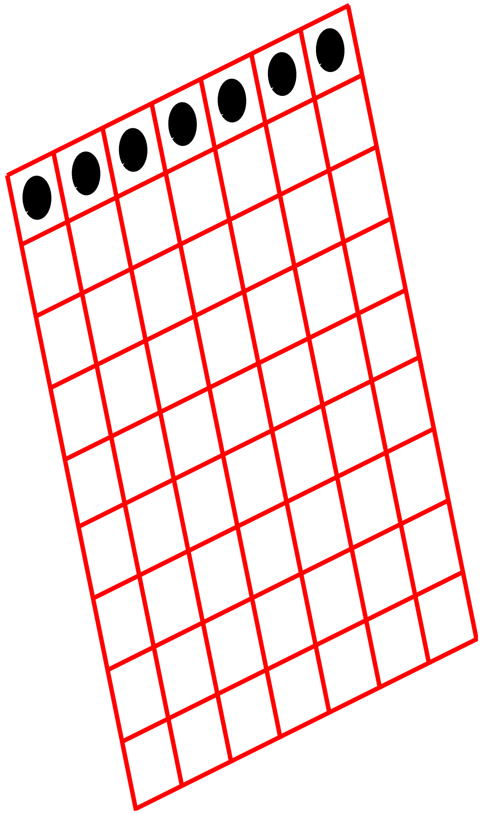


- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

-1	0	1
-2	0	2
-1	0	1

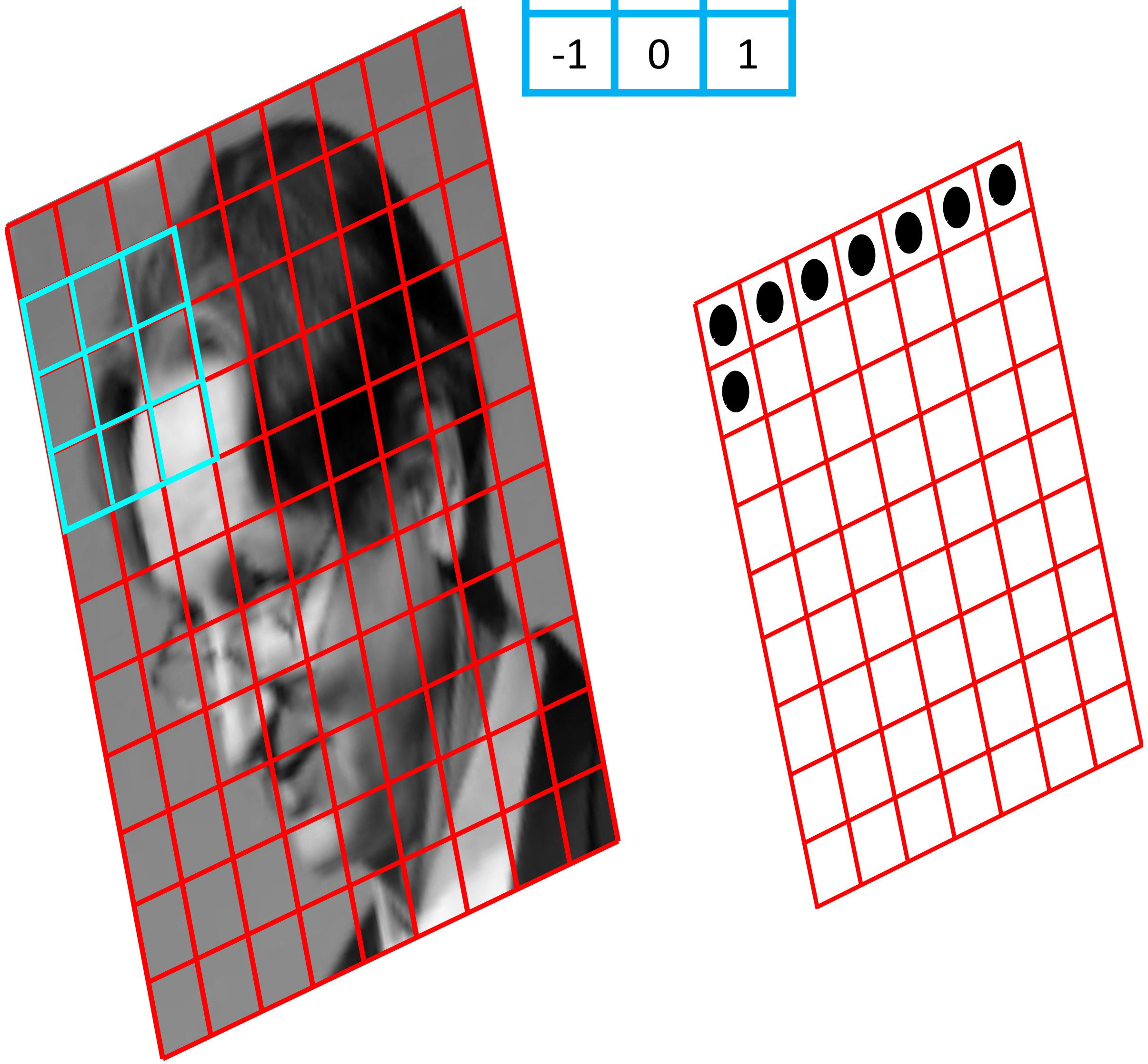




- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

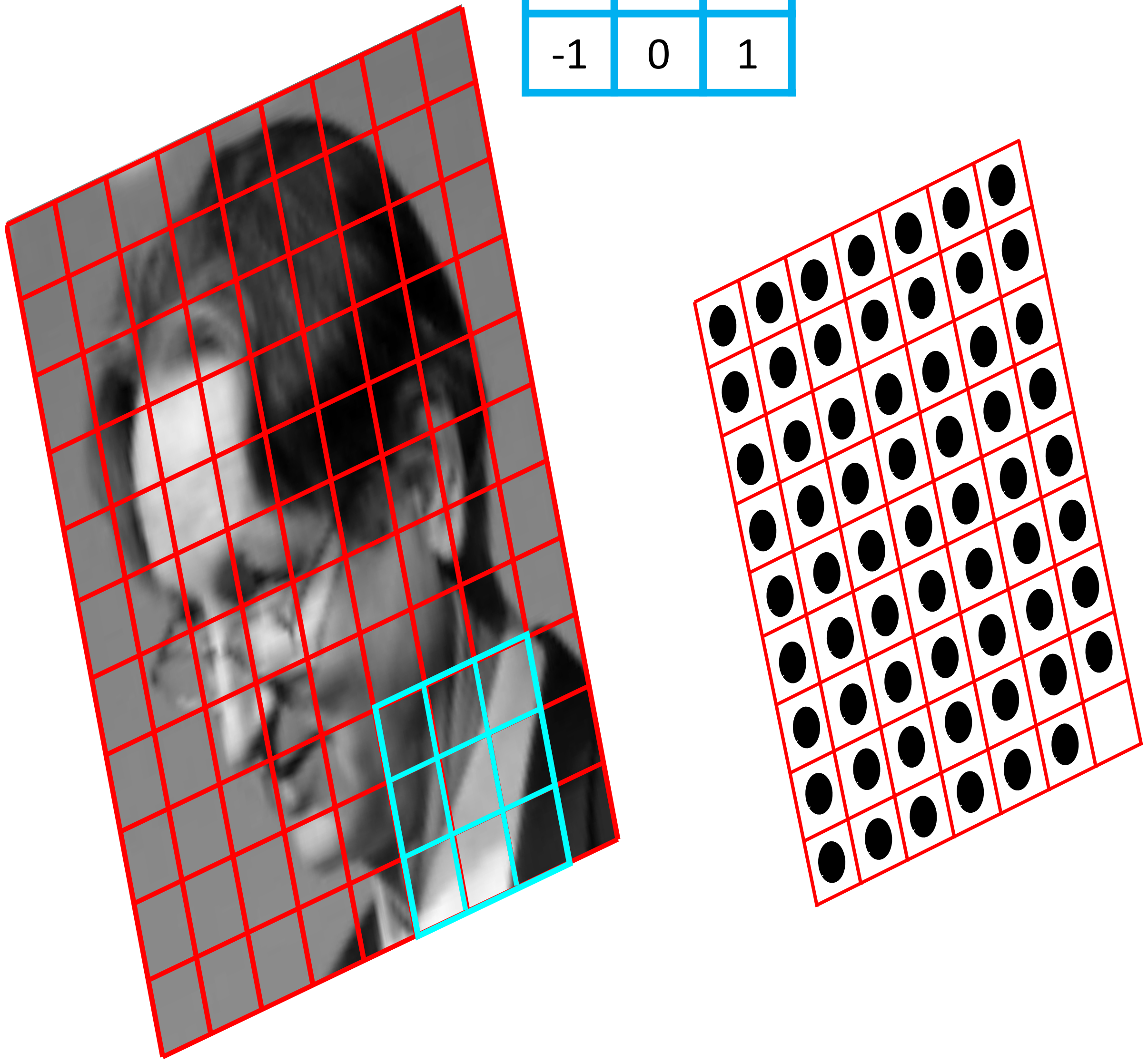
-1	0	1
-2	0	2
-1	0	1



- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

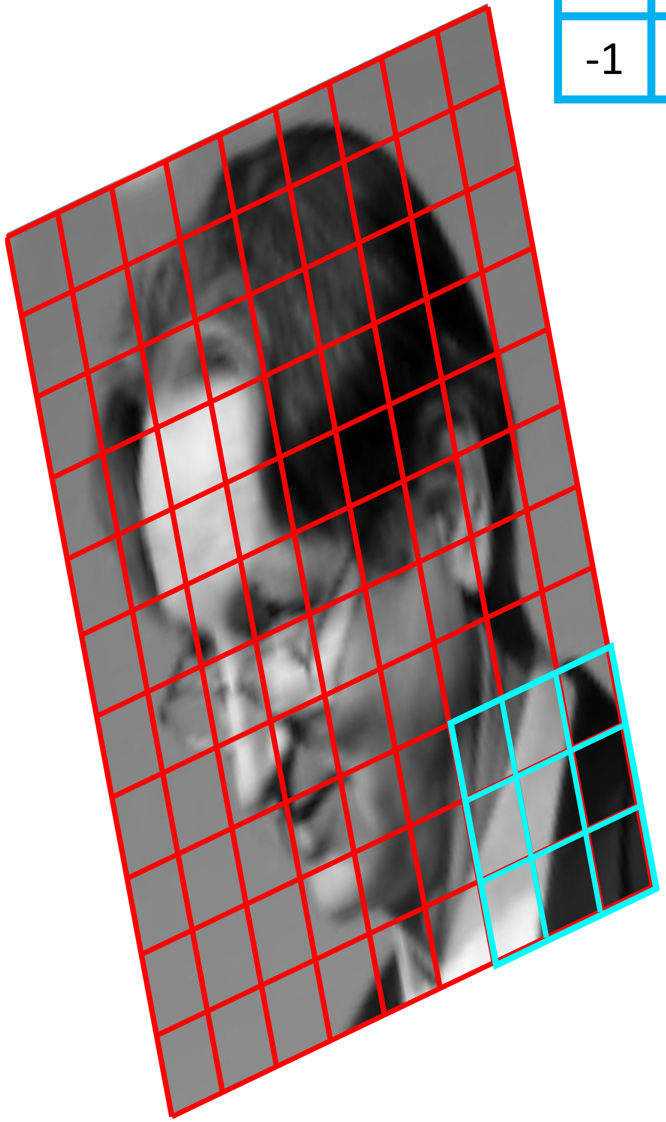
-1	0	1
-2	0	2
-1	0	1

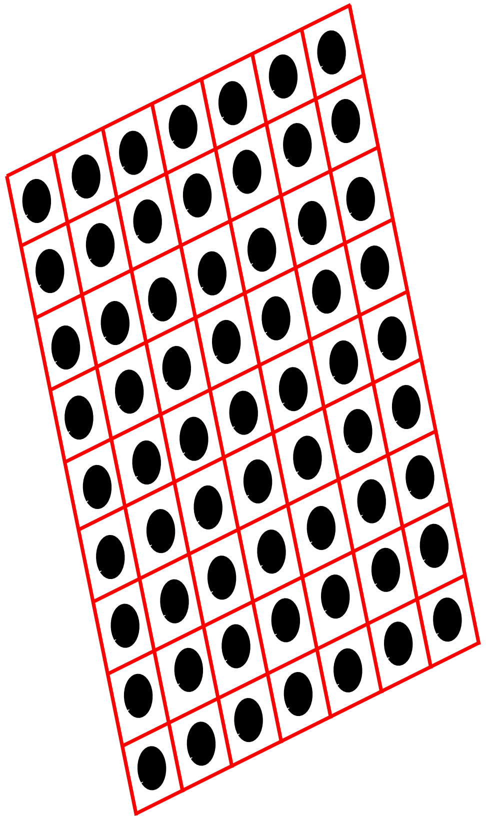


- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

2D Convolution

-1	0	1
-2	0	2
-1	0	1





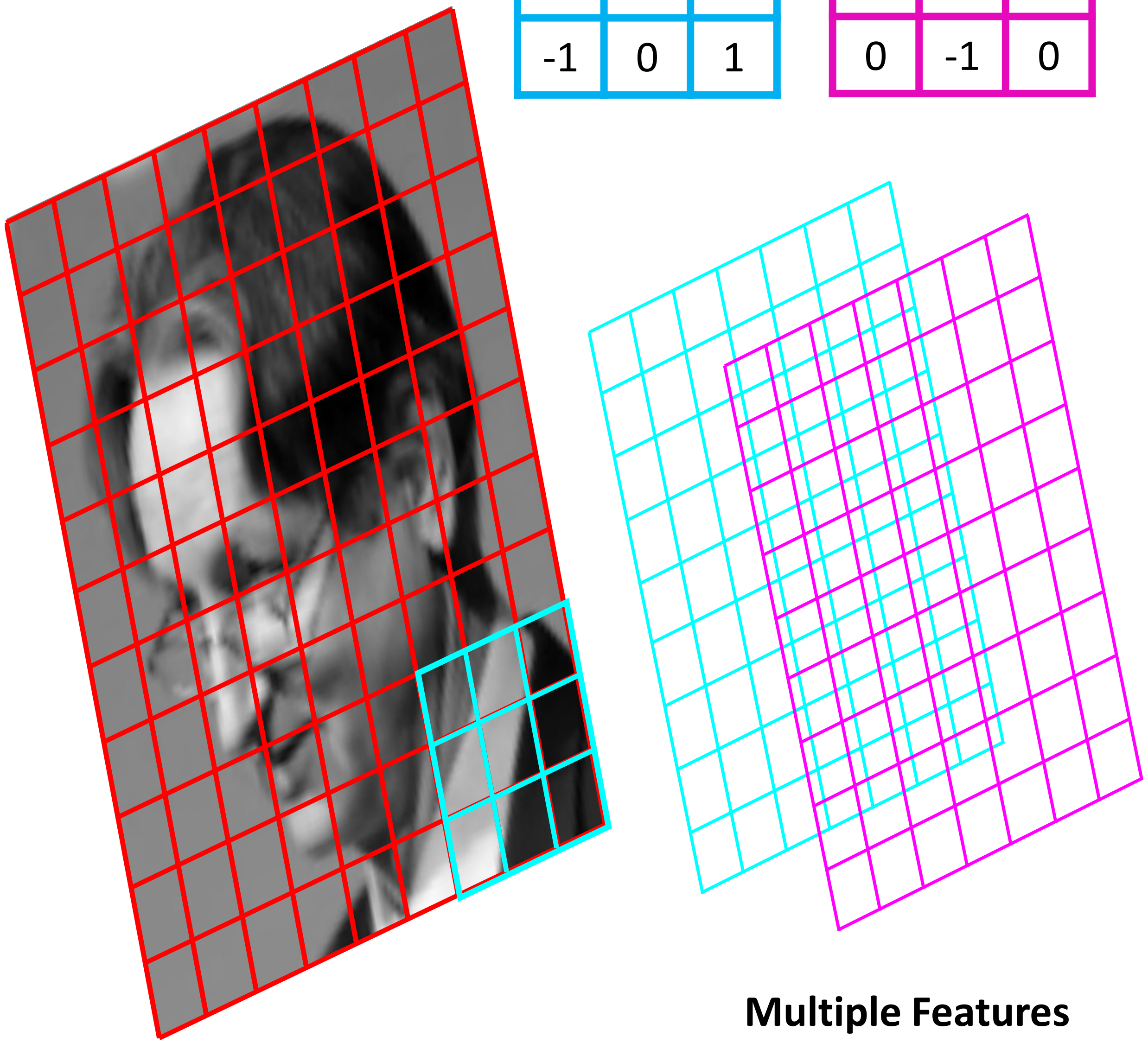
Feature map

- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output
- The resulting output is called a **feature map**

2D Convolution

-1	0	1
-2	0	2
-1	0	1

0	-1	0
-1	5	-1
0	-1	0

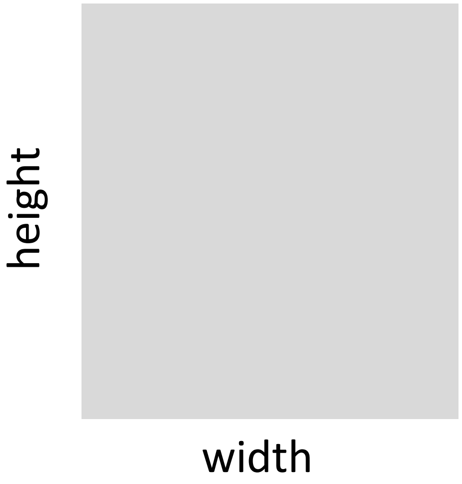


Multiple Features

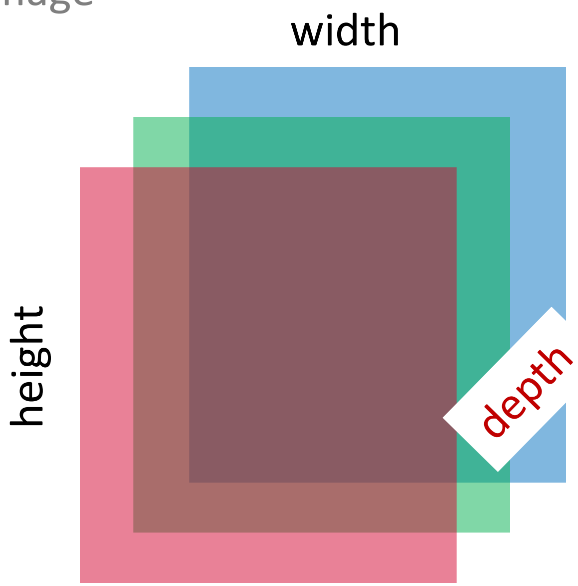
- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output
- The resulting output is called a **feature map**
- We can use **multiple filters** to get multiple feature maps
- How convolutions will happen for colored images?

What would happen in case of colored images?

Grayscale Image

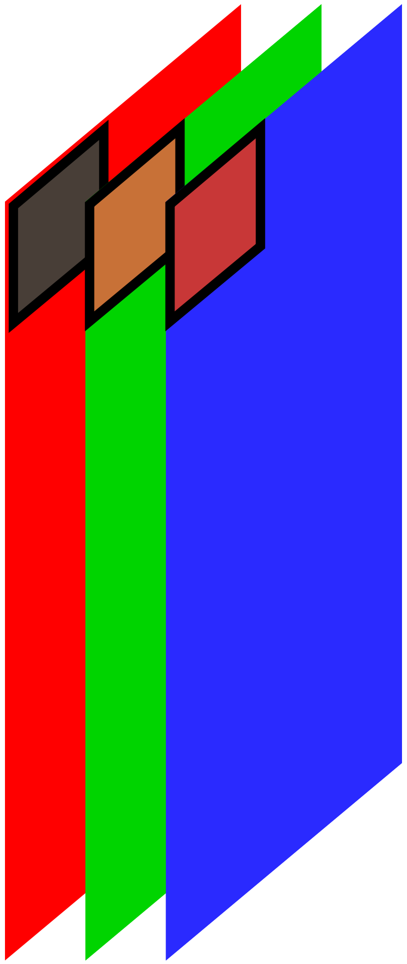


Color Image



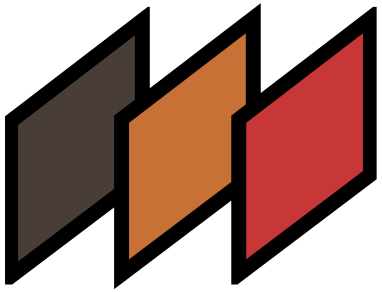
- Grayscale images have a single channel (depth = 1)
- Colored images have more than one channel (depth > 1)
- Ex. RGB images has **3 channels**
- How does convolution happen in 3D case?
- Well the kernel or filter will be 3D too (i.e. will have same number of channels as the input)

What would happen in case of colored images?



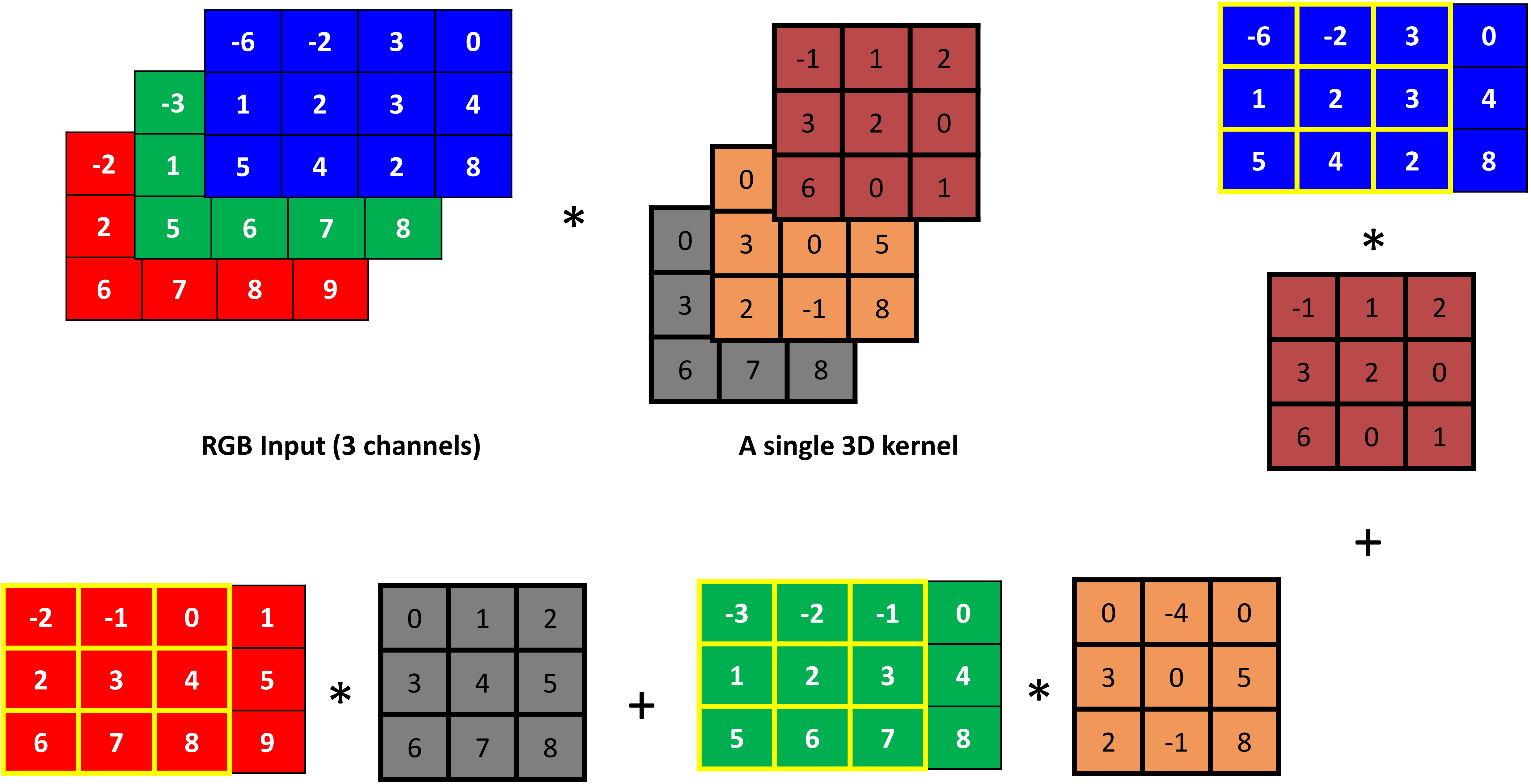
Input

- Grayscale images have a single channel (depth = 1)
- Colored images have more than one channel (depth > 1)
- Ex. RGB images has **3 channels**
- How does convolution happen in 3D case?
- Well the kernel or filter will be 3D too (i.e. will have same number of channels as the input)



A single 3D kernel

- Let’s see how the 3D convolutions happen


RGB Input (3 channels)
A single 3D kernel

$$\begin{bmatrix} -2 & -1 & 0 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} -3 & -2 & -1 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} + \begin{bmatrix} -6 & -2 & 3 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 4 & 2 & 8 \end{bmatrix}$$

$$* \qquad\qquad\qquad * \qquad\qquad\qquad *$$

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \qquad \begin{bmatrix} 0 & -4 & 0 \\ 3 & 0 & 5 \\ 2 & -1 & 8 \end{bmatrix} \qquad \begin{bmatrix} -1 & 1 & 2 \\ 3 & 2 & 0 \\ 6 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 323 & \; \end{bmatrix}$$

$$\begin{array}{l} (-2)(0) + (-1)(1) + (0)(2) \\ + (2)(3) + (3)(4) + (4)(5) \\ + (6)(6) + (7)(7) + (8)(8) \end{array} + \begin{array}{l} (-3)(0) + (-2)(-4) + (-1)(0) \\ + (1)(3) + (2)(0) + (3)(5) \\ + (5)(2) + (6)(-1) + (7)(8) \end{array} + \begin{array}{l} (-6)(-1) + (-2)(1) + (3)(2) \\ + (1)(3) + (2)(2) + (3)(0) \\ + (5)(6) + (4)(0) + (2)(1) \end{array}$$

$$\mathbf{188} \qquad\qquad\qquad \mathbf{86} \qquad\qquad\qquad \mathbf{49}$$

$$\begin{bmatrix} -2 & -1 & 0 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} -3 & -2 & -1 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} + \begin{bmatrix} -6 & -2 & 3 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 4 & 2 & 8 \end{bmatrix}$$

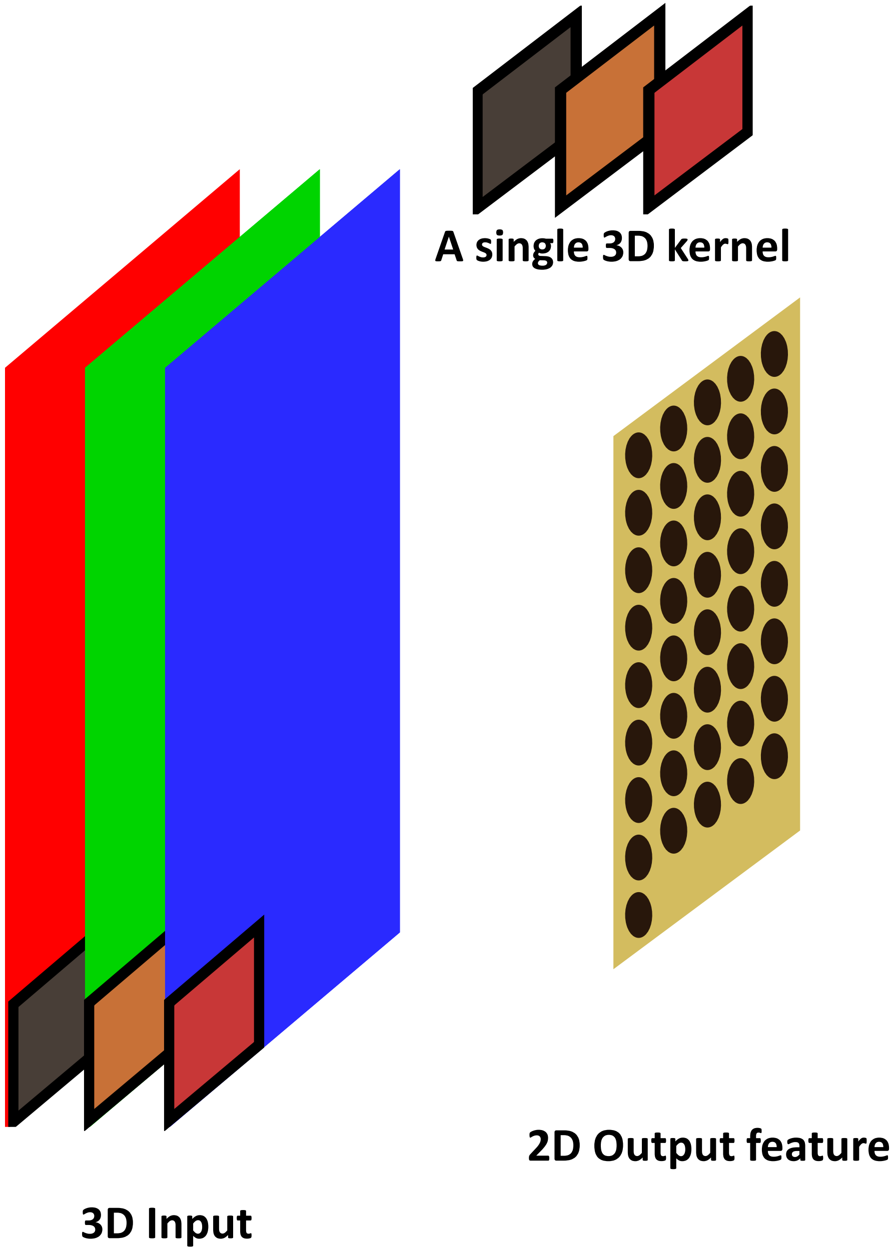
$$* \qquad\qquad * \qquad\qquad *$$

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \qquad \begin{bmatrix} 0 & -4 & 0 \\ 3 & 0 & 5 \\ 2 & -1 & 8 \end{bmatrix} \qquad \begin{bmatrix} -1 & 1 & 2 \\ 3 & 2 & 0 \\ 6 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 323 & 370 \end{bmatrix}$$

$$\begin{aligned} &(-1)(0) + (0)(1) + (1)(2) \\ &+ (3)(3) + (4)(4) + (5)(5) \\ &+ (7)(6) + (8)(7) + (9)(8) \end{aligned} \quad + \quad \begin{aligned} &(-2)(0) + (-1)(-4) + (0)(0) \\ &+ (2)(3) + (3)(0) + (4)(5) \\ &+ (6)(2) + (7)(-1) + (8)(8) \end{aligned} \quad + \quad \begin{aligned} &(-2)(-1) + (3)(1) + (0)(2) \\ &+ (2)(3) + (3)(2) + (4)(0) \\ &+ (4)(6) + (2)(0) + (8)(1) \end{aligned}$$

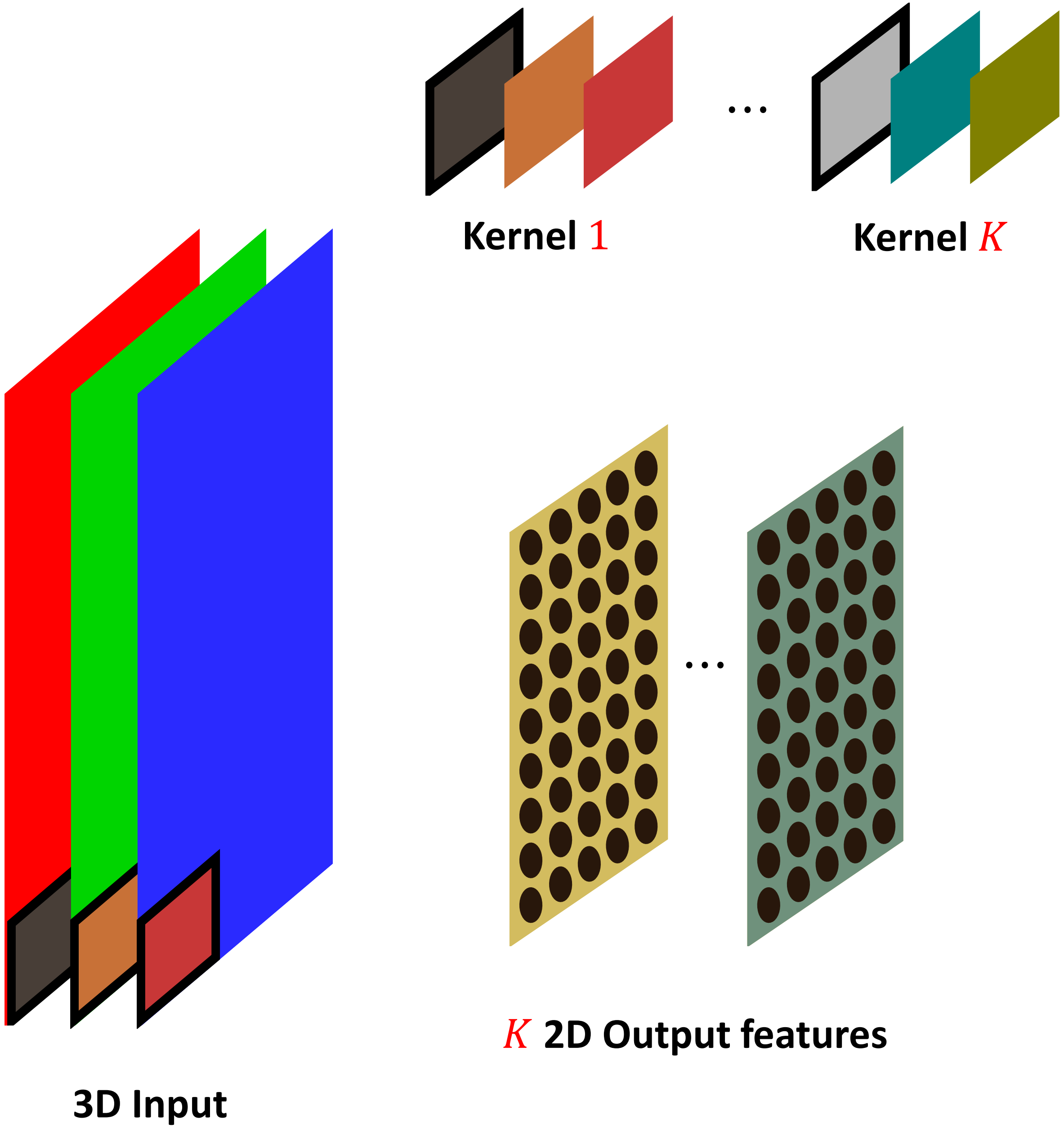
222 **99** **49**

What would happen in case of colored images?



- Grayscale images have a single channel (depth = 1)
- Colored images have more than one channel (depth > 1)
- Ex. RGB images has **3 channels**
- How does convolution happen in 3D case?
- Well the kernel or filter will be 3D too (i.e. will have same number of channels as the input)
- The kernel moves along the width and height (and not along the depth)
- **Therefore, the feature output is 2D**!

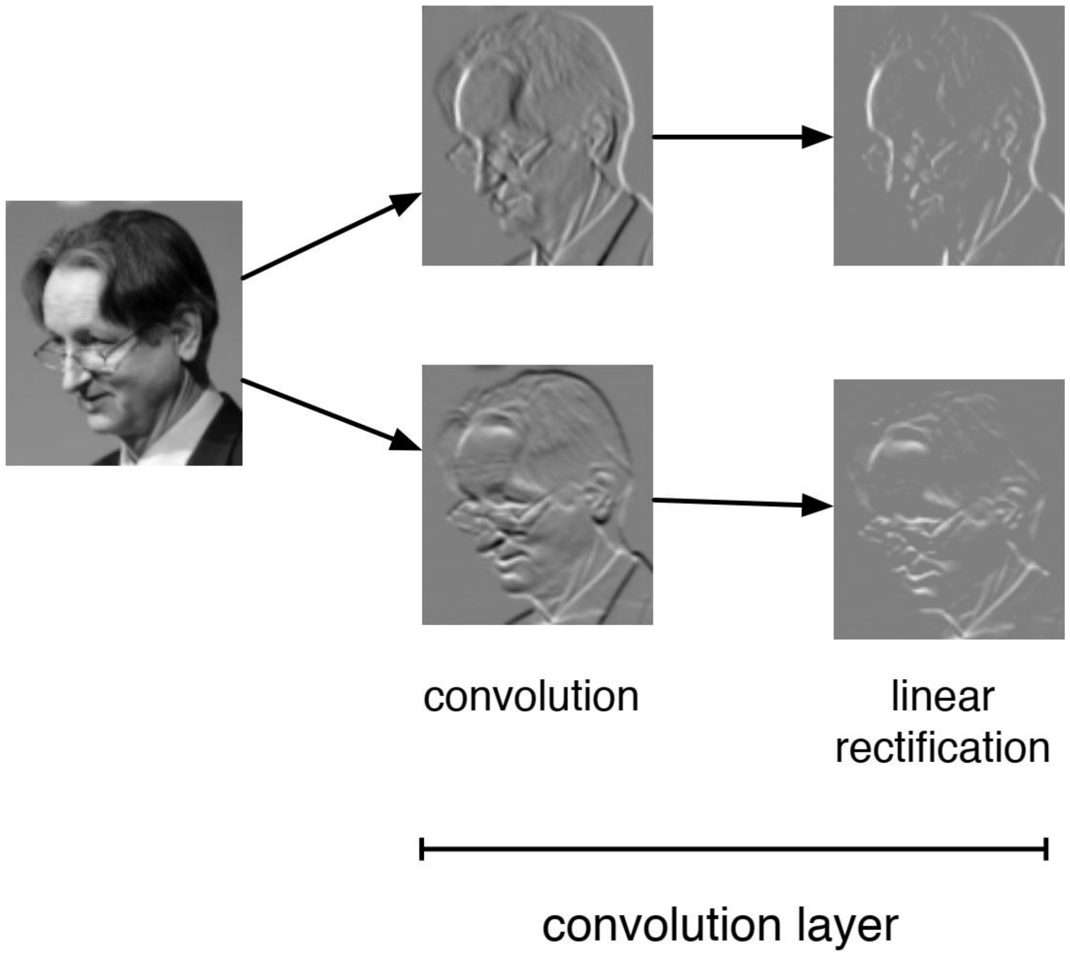
What would happen in case of colored images?



- Grayscale images have a single channel (depth = 1)
- Colored images have more than one channel (depth > 1)
- Ex. RGB images has **3 channels**
- How does convolution happen in 3D case?
- Well the kernel or filter will be 3D too (i.e. will have same number of channels as the input)
- The kernel moves along the width and height (and not along the depth)
- **Therefore, the feature output is 2D**!
- Once again, if we apply **multiple 3D filters**, we will get multiple 2D output features

Convolution followed by linear rectification

It is common to apply a ReLU nonlinear activation on the output feature following convolution: $y = \max(z, 0)$

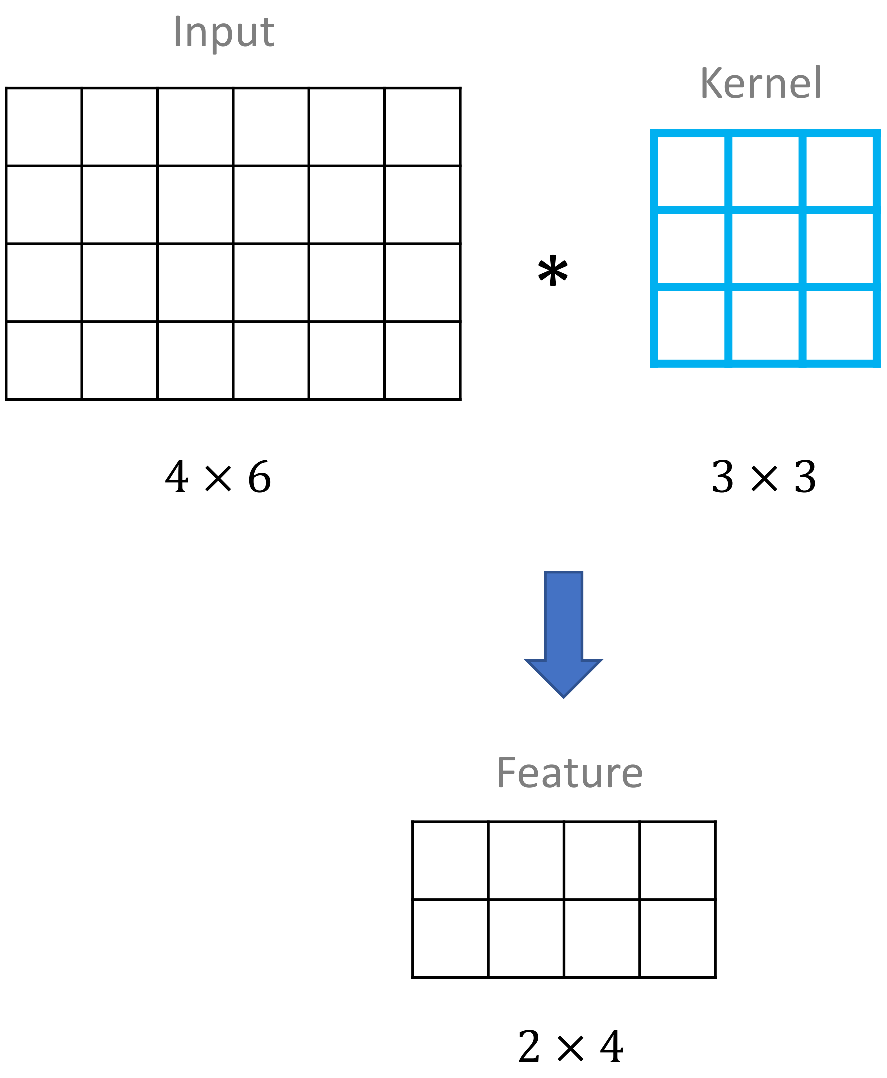


Why might we do this?

- Convolution is a **linear** operation. Passing the linear output through nonlinear activation leads to more powerful features
- While pooling the results, two edges in opposite directions shouldn't cancel
- It has been reported that nonlinear activations (like ReLU or ELU) when used after convolutional layers given better performance

What are the relations between input sizes, kernel sizes, and output sizes?

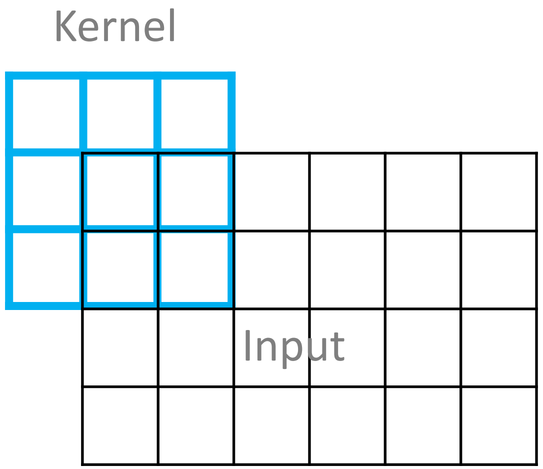
Relation between input sizes, kernel sizes, and output sizes

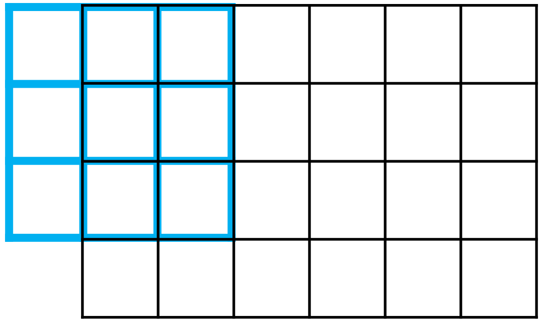


- So far we have not said anything explicit about the dimensions of the
  - Inputs
  - Kernels
  - Outputs
  - the relations between them

- We will see how they are related but before that we will discuss **zero-padding** and **stride**

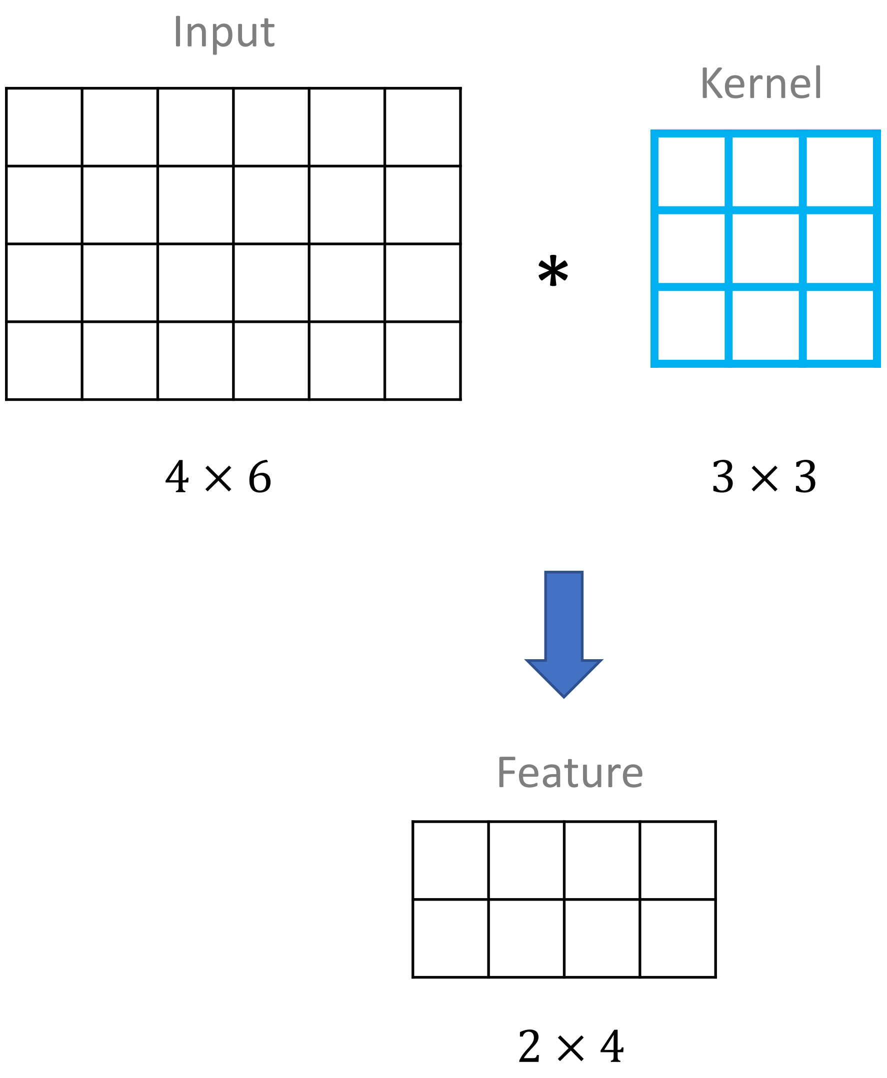
Zero padding





- Note that we can't place the kernel centred at the corners or at boundaries of our image
- Thus any interesting information on the boundaries of the original image is lost

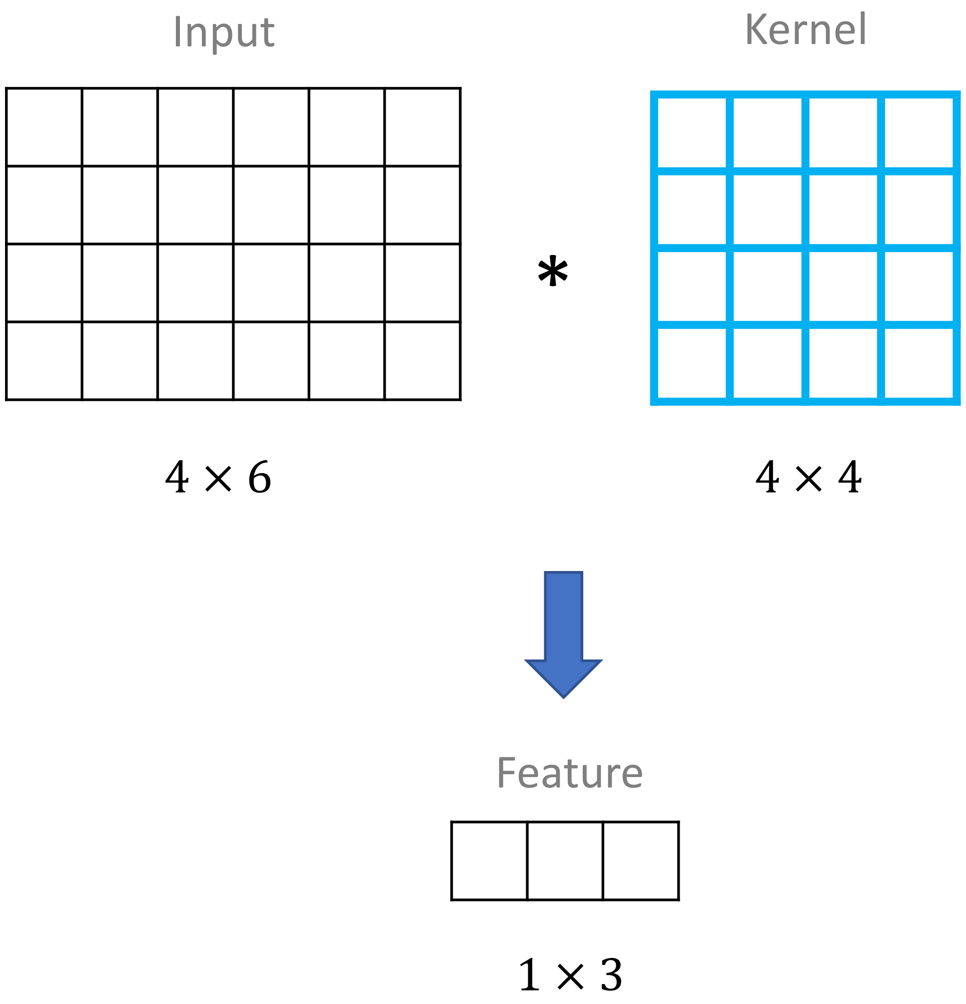
Zero padding



- Note that we can't place the kernel centred at the corners or at boundaries of our image
- Thus any interesting information on the boundaries of the original image is lost
- This loss of information results in an output feature size smaller than the input image
- If input size is $n_h \times n_w$, kernel size is $k_h \times k_w$, then output feature size $f_h \times f_w$ is related as follows:

$$f_h = n_h - k_h + 1$$
$$f_w = n_w - k_w + 1$$

Zero padding



- Note that we can't place the kernel centred at the corners or at boundaries of our image
- Thus any interesting information on the boundaries of the original image is lost
- This loss of information results in an output feature size smaller than the input image
- If input size is $n_h \times n_w$, kernel size is $k_h \times k_w$, then output feature size $f_h \times f_w$ is related as follows:

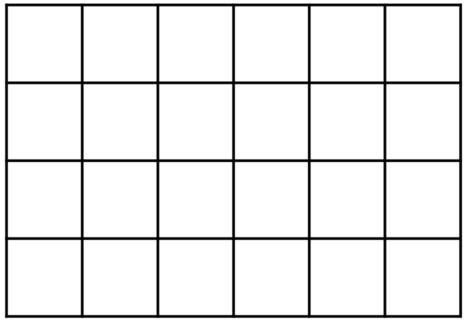
$$f_h = n_h - k_h + 1$$

$$f_w = n_w - k_w + 1$$

- As the size of the kernel increases, this output size reduces even more

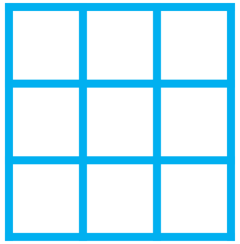
Zero padding

Input



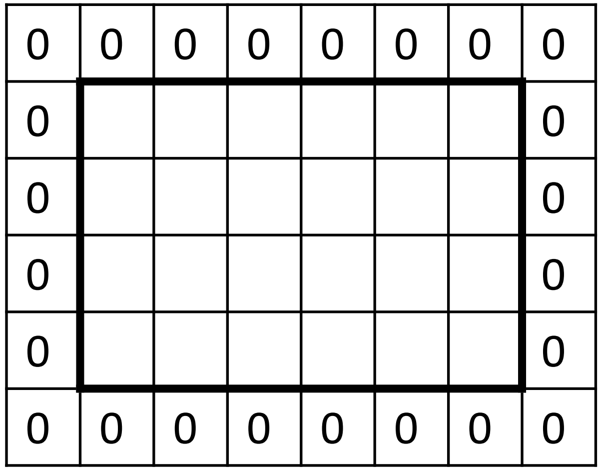
4×6

Kernel



3×3

Zero-padded Input

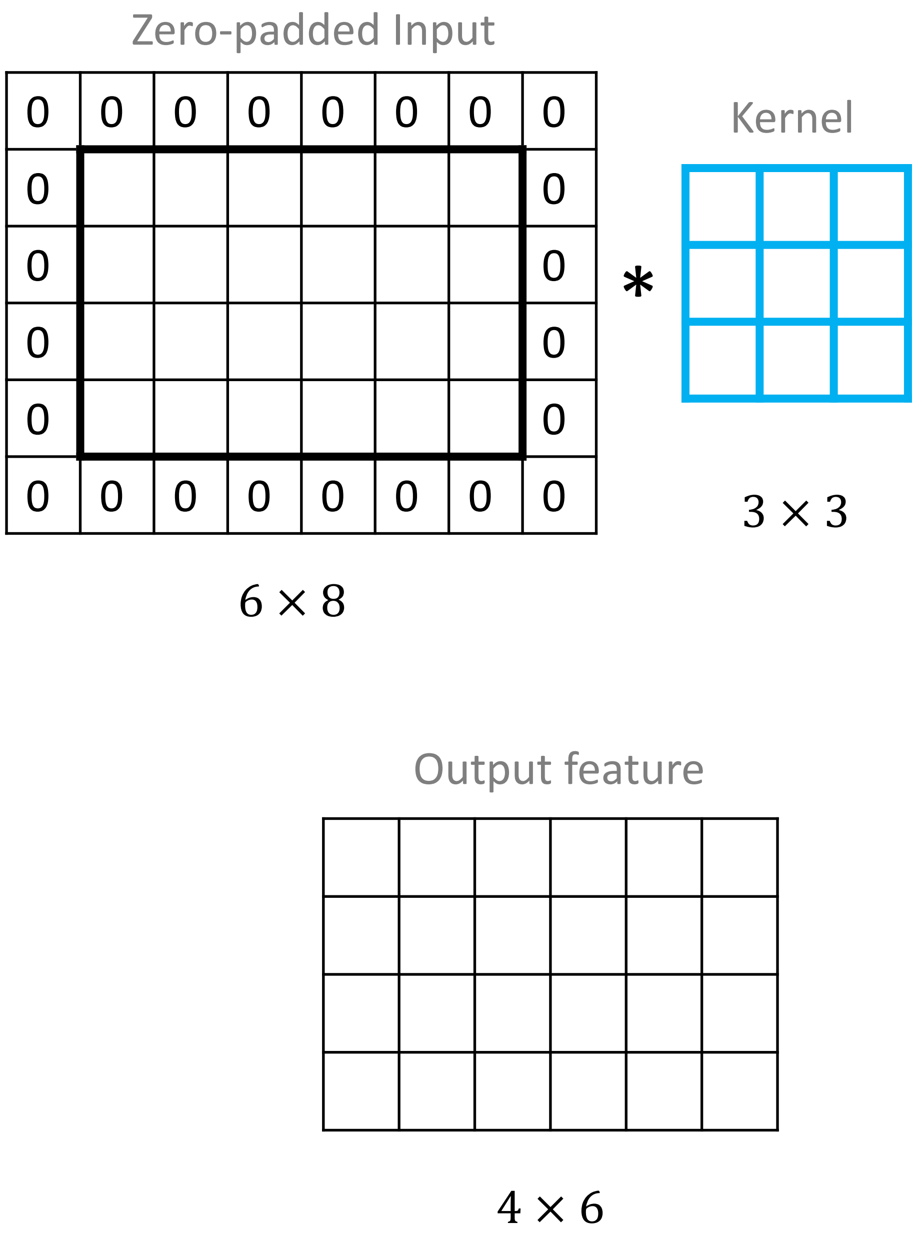


6×8

- One straightforward solution to this problem is **to add zero pixels around the boundary of our input image**, thus increasing the effective size of the image
- This means we pad zeros of width p_w on left and right, and pad zeros of height p_h on top and bottom
- The output feature shape will be $f_h \times f_w$:

$$f_h = n_h - k_h + 2p_w + 1$$
$$f_w = n_w - k_w + 2p_h + 1$$

Zero padding



- One straightforward solution to this problem is **to add zero pixels around the boundary of our input image**, thus increasing the effective size of the image

- This means we pad zeros of width p_w on left and right, and pad zeros of height p_h on top and bottom

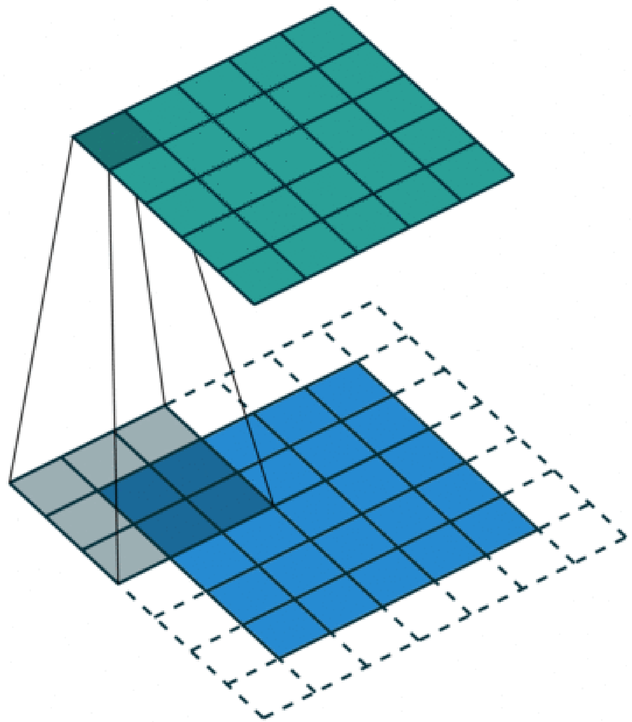
- The output feature shape will be $f_h \times f_w$:

$$f_h = n_h - k_h + 2p_w + 1$$
$$f_w = n_w - k_w + 2p_h + 1$$

- Usual values: $p_h = \frac{k_h - 1}{2}$ and $p_w = \frac{k_w - 1}{2}$

- It becomes easy to work with odd-sized kernels like 3x3, 5x5, 7x7 as zero-padding will give an integer number, else ceil the value

Stride



- When computing the cross-correlation, we typically move our kernel by one interval along right and/or downwards
- **Stride** defines the intervals at which the kernel is applied
- By default, we slide one element at a time
- However, sometimes, **either for computational efficiency** or because **we wish to downsample (reduce size of output)**, we move our window more than one element at a time, skipping the intermediate locations
- For such cases, the stride would be greater than 1

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. Let's consider an example with $S = 2$

$$\left\lfloor \frac{4-3+2}{2} + 1 \right\rfloor \qquad \left\lfloor \frac{6-3+2}{2} + 1 \right\rfloor$$

$$\lfloor 2.5 \rfloor \qquad \lfloor 3.5 \rfloor$$

$$2 \qquad 3$$

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

*

Kernel

0	1	2
3	4	5
6	7	8

3×3

=

Output feature

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

*

Kernel

0	1	2
3	4	5
6	7	8

3×3

=

Output feature

56		

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

$*$

Kernel

0	1	2
3	4	5
6	7	8

3×3

$=$

Output feature

56	151	

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

$*$

Kernel

0	1	2
3	4	5
6	7	8

3×3

$=$

Output feature

56	151	164

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

*

Kernel

0	1	2
3	4	5
6	7	8

3×3

=

Output feature

56	151	164
122		

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

$*$

Kernel

0	1	2
3	4	5
6	7	8

3×3

$=$

Output feature

56	151	164
122	159	

2×3

Stride

- When the stride along the height is s_h and the stride along the width is s_w, the output shape $f_h \times f_w$ is

$$\left\lfloor \frac{n_h - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{n_w - k_w + 2p_w}{s_w} + 1 \right\rfloor$$

- Typically, equal strides are taken, $s_h = s_w = S$. For stride $S = 2$, the following output is obtained

Input

0	0	0	0	0	0	0	0
0	1	2	3	4	4	5	0
0	6	0	7	8	9	0	0
0	4	1	7	2	9	4	0
0	9	4	9	1	4	2	0
0	0	0	0	0	0	0	0

4×6

$*$

Kernel

0	1	2
3	4	5
6	7	8

3×3

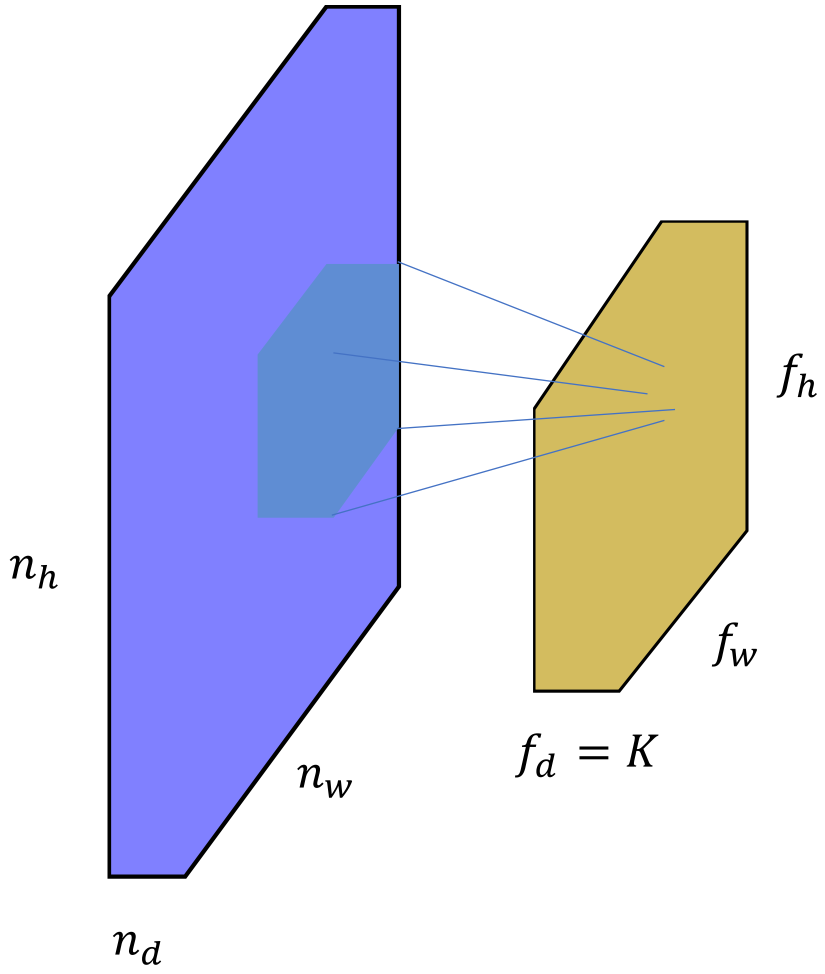
$=$

Output feature

56	151	164
122	159	121

2×3

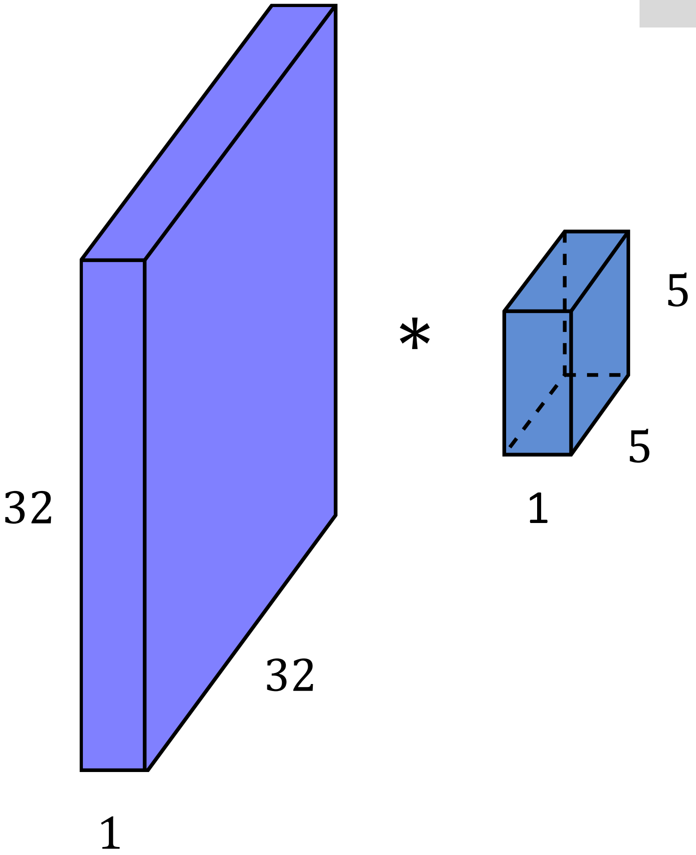
Depth of the output layer

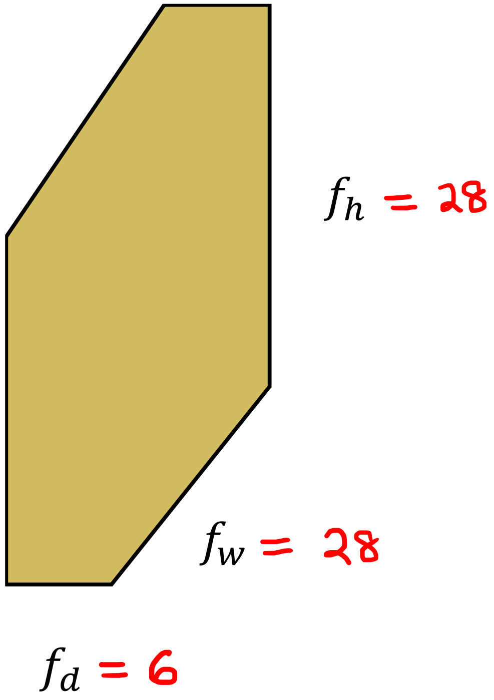


- Finally, let's come to the depth f_d of the output feature layer
- If we have multi-channel inputs, depth will be $n_d > 1$
- Each 3D kernel will give us one 2D output feature
- K kernels will give us K such 2D output features
- We can think of the resulting output feature as $f_h \times f_w \times f_d$ volume
- Thus, $f_d = K$

Example for determining sizes

- 6 kernels
- Stride, S = 1
- Padding, P = 0





- Output layer dimensions

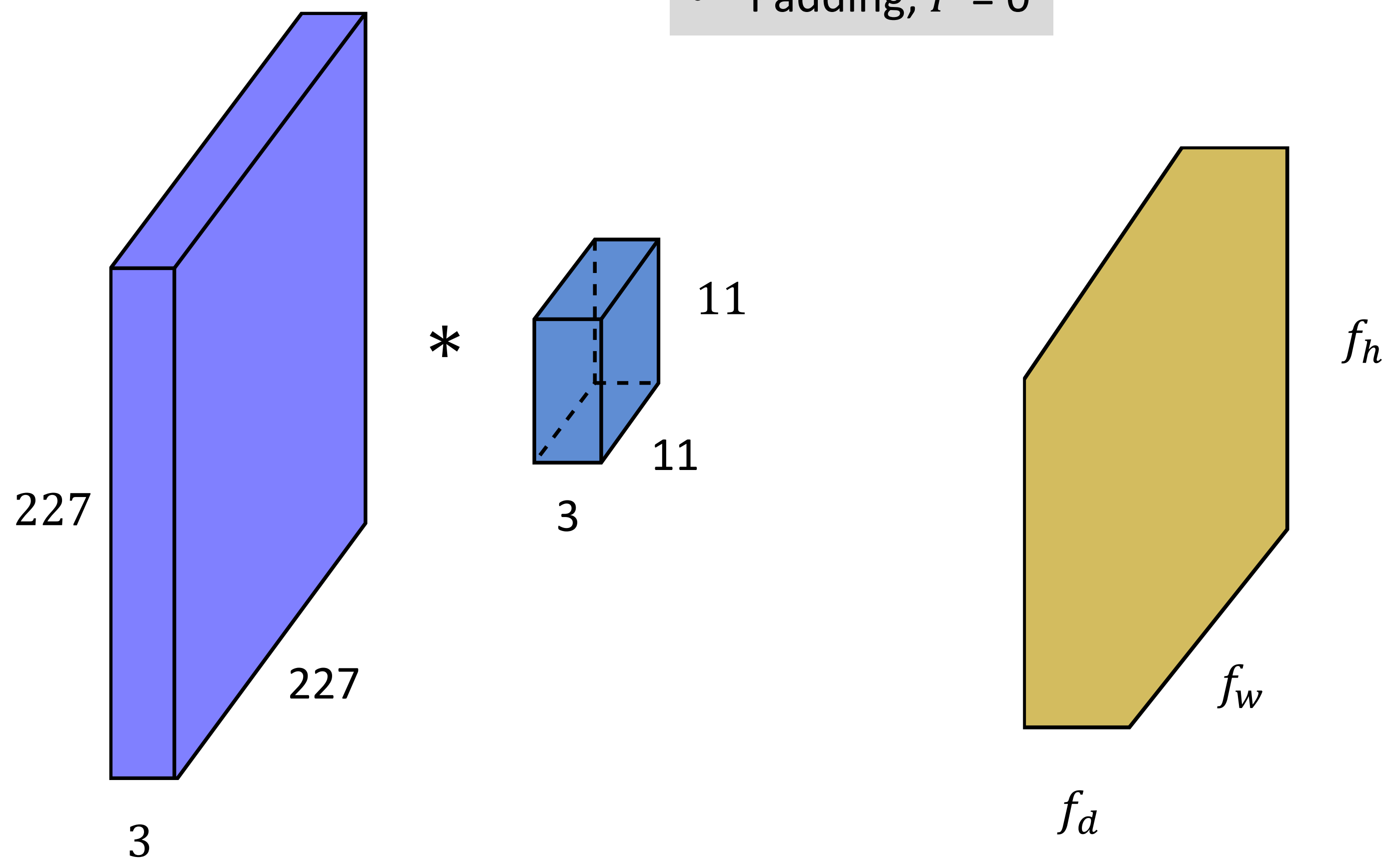
$$f_h = \left\lfloor \frac{n_h - k_h + 2P}{S} + 1 \right\rfloor$$

$$f_w = \left\lfloor \frac{n_w - k_w + 2P}{S} + 1 \right\rfloor$$

$f_d = K$ (number of kernels)

Example for determining sizes

- 96 kernels
- Stride, S = 4
- Padding, P = 0



- Output layer dimensions

$$f_h = \left\lfloor \frac{n_h - k_h + 2P}{S} + 1 \right\rfloor$$

$$f_w = \left\lfloor \frac{n_w - k_w + 2P}{S} + 1 \right\rfloor$$

$f_d = K$ (number of kernels)

$$f_w = \left\lfloor \frac{227-11}{4} + 1 \right\rfloor$$

$$= \left\lfloor \frac{216}{4} + 1 \right\rfloor$$

$$= \lfloor 54 + 1 \rfloor$$

$$= 55$$

- What is the connection between this operation (convolution) and neural networks?

- We will try to understand this by considering the task of image classification

Output features for image classification

Input

Kernel

-1	0	1
-2	0	2
-1	0	1

Edge detector

Feature map

Optimize the weights of classifier based on training data

Classifier output

Face, Hands, Legs

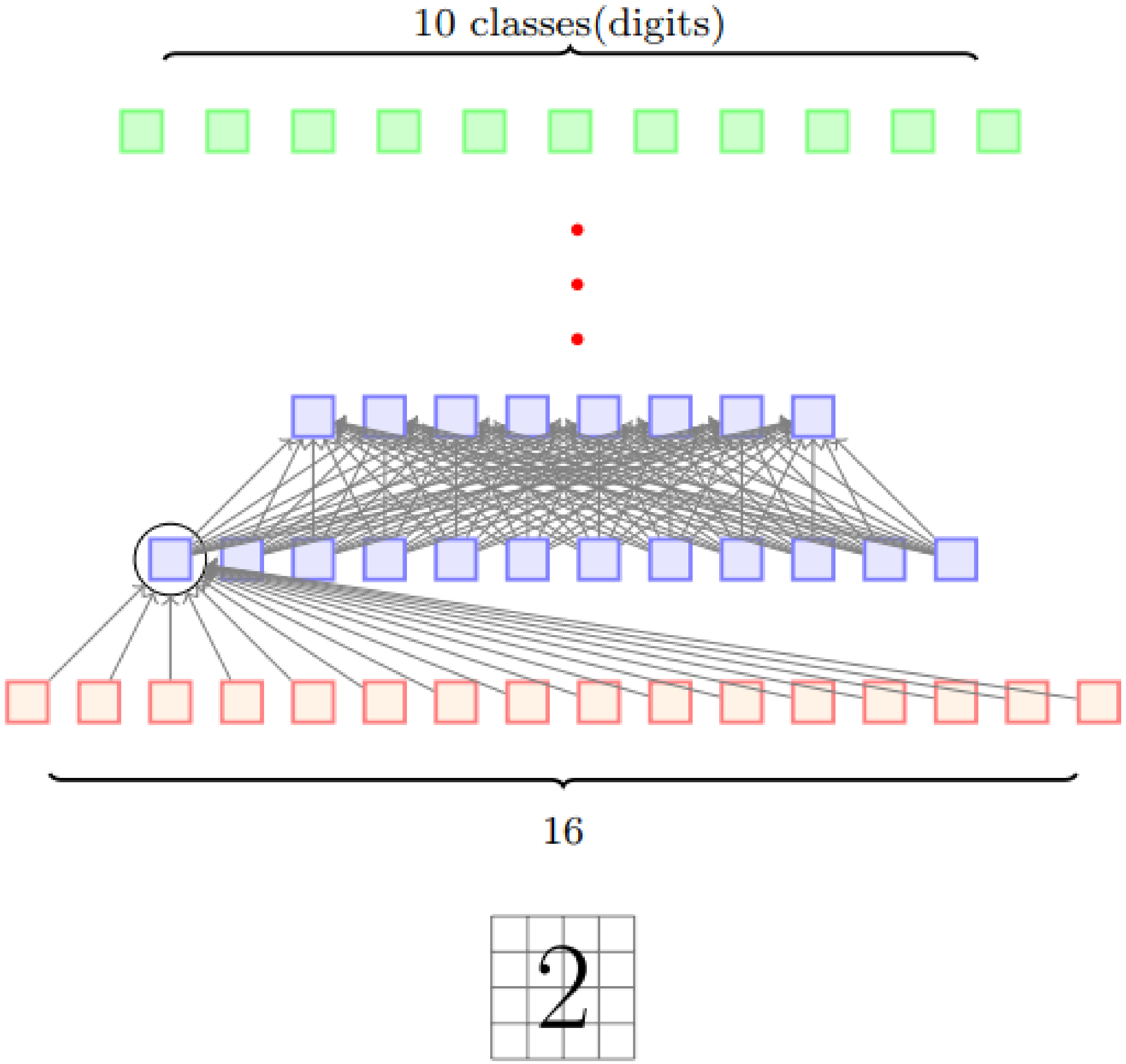
- Instead of using handcrafted kernels such as edge detectors **can we learn (or optimize) meaningful kernels/filters in addition to learning the weights of the classifier?**

- Even better, if we can learn **multiple** meaningful kernels/filters in addition to the weights of the classifier

- In CNN, we treat these kernels as parameters and learn them in addition to the weights of the classifier (using back propagation) in CNN

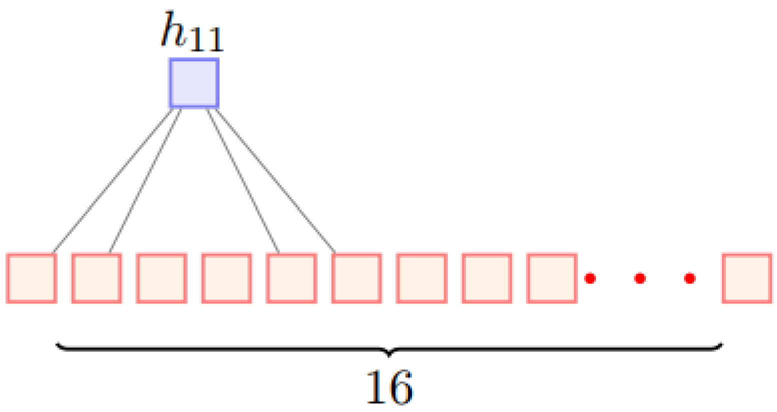
- But how is CNN different than fully-connected feedforward neural networks?

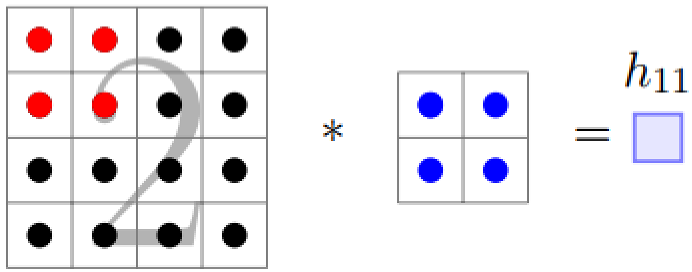
Full connected NN vs CNN



- This is what a regular fully-connected feed-forward neural network looks like
- It is **dense,** there are many connections
- For example, all the 16 input neurons are contributing to the computation of $h_{11} = h_1^{(1)}$
- Contrast this to what happens in the case of convolution

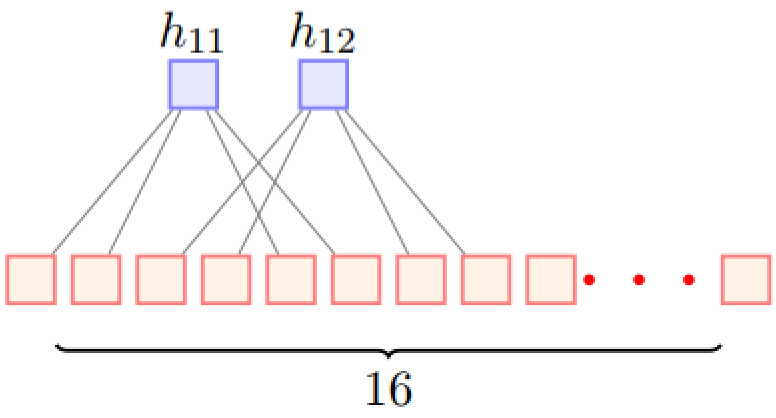
Full connected NN vs CNN

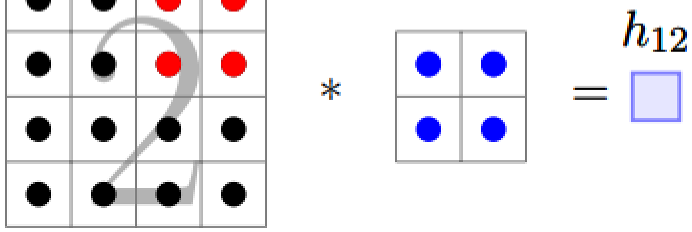




- Only a few local neurons participate in the computation of $h_{11} = h_1^{(1)}$
- For example, only pixels 1, 2, 5, 6 contribute to $h_1^{(1)}$

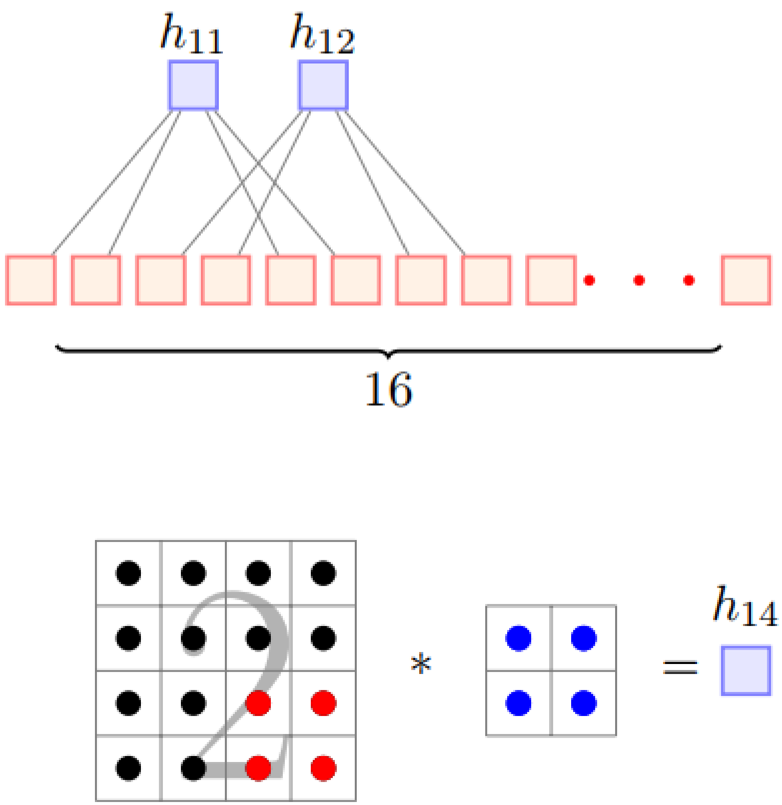
Full connected NN vs CNN





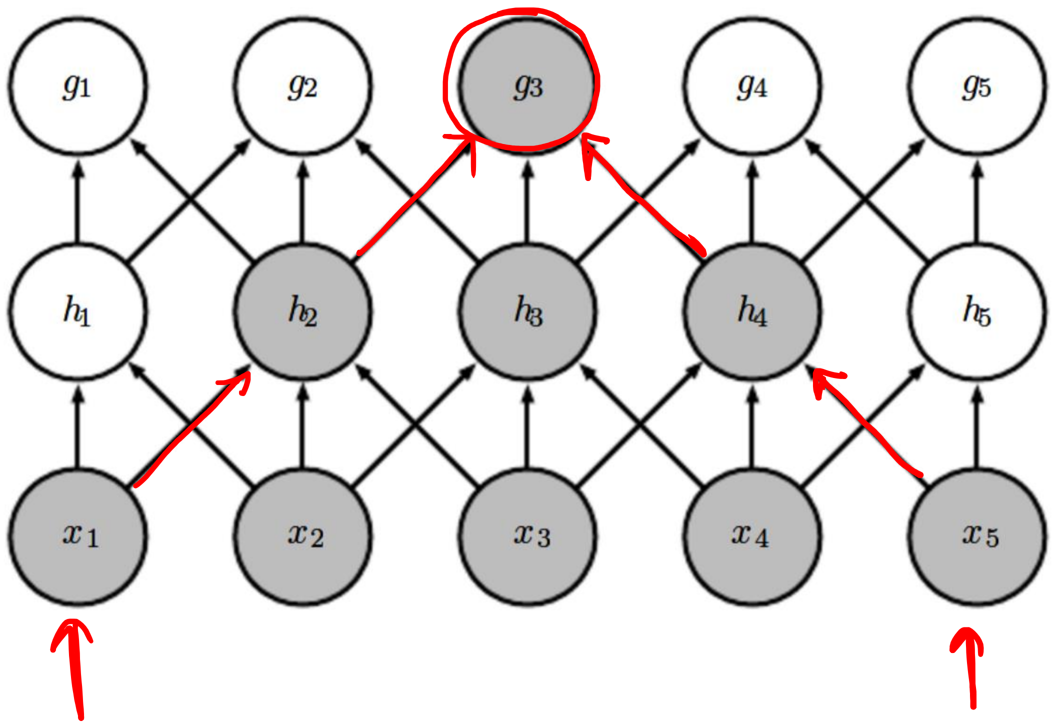
- Only a few local neurons participate in the computation of $h_{11} = h_1^{(1)}$
- For example, only pixels 1, 2, 5, 6 contribute to $h_1^{(1)}$
- For example, only pixels 3, 4, 7, 8 contribute to $h_{12} = h_2^{(1)}$

Full connected NN vs CNN



- Only a few local neurons participate in the computation of $h_{11} = h_1^{(1)}$
- For example, only pixels 1, 2, 5, 6 contribute to $h_1^{(1)}$
- The connections are much sparser
- This **sparse connectivity** reduces the number of parameters in the model
- But is sparse connectivity good? Aren't we losing information (by losing interactions between some input pixels) ?

Full connected NN vs CNN



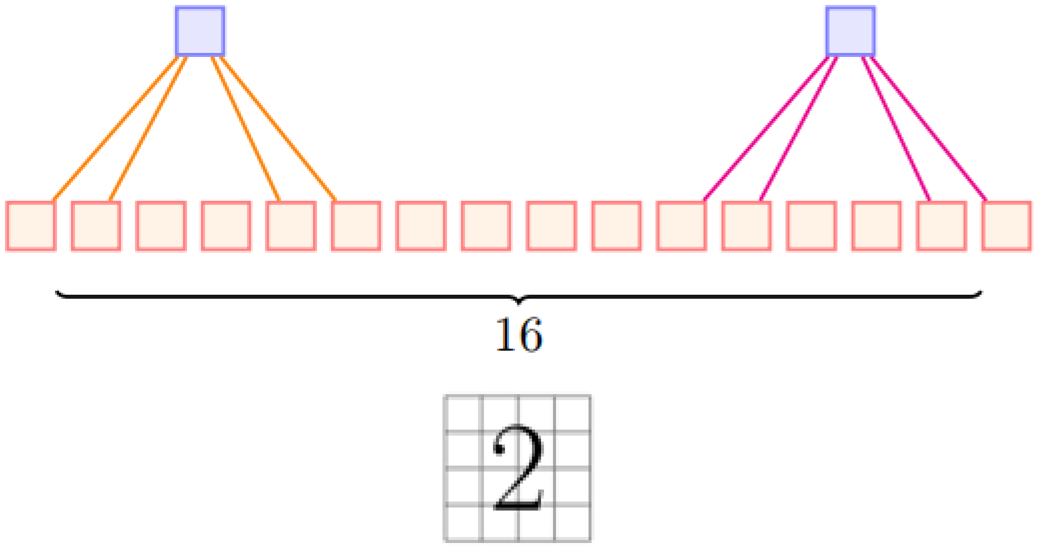
- But is sparse connectivity good? Aren't we losing information (by losing interactions between some input pixels) ?

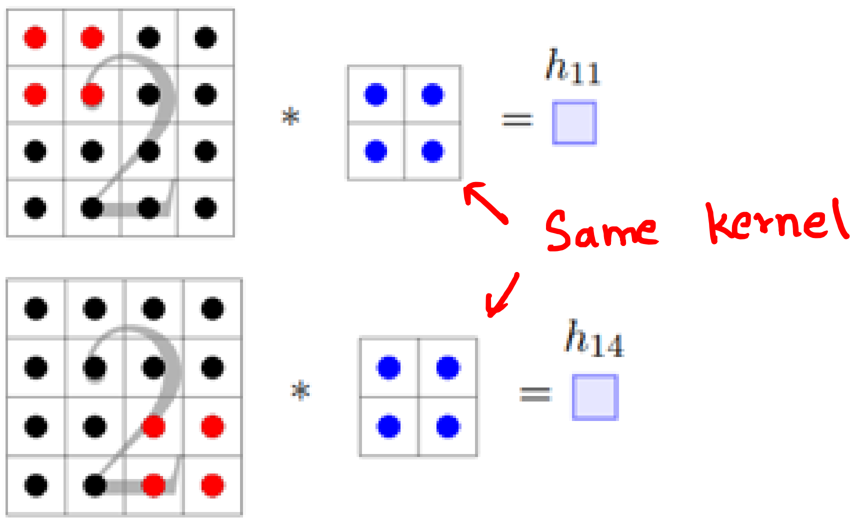
- It turns out we are not losing information/interactions

- The two highlighted neurons (x_1 and x_5) do not interact in *hidden layer* 1

- But they indirectly contribute to the computation of g_3 and hence interact indirectly

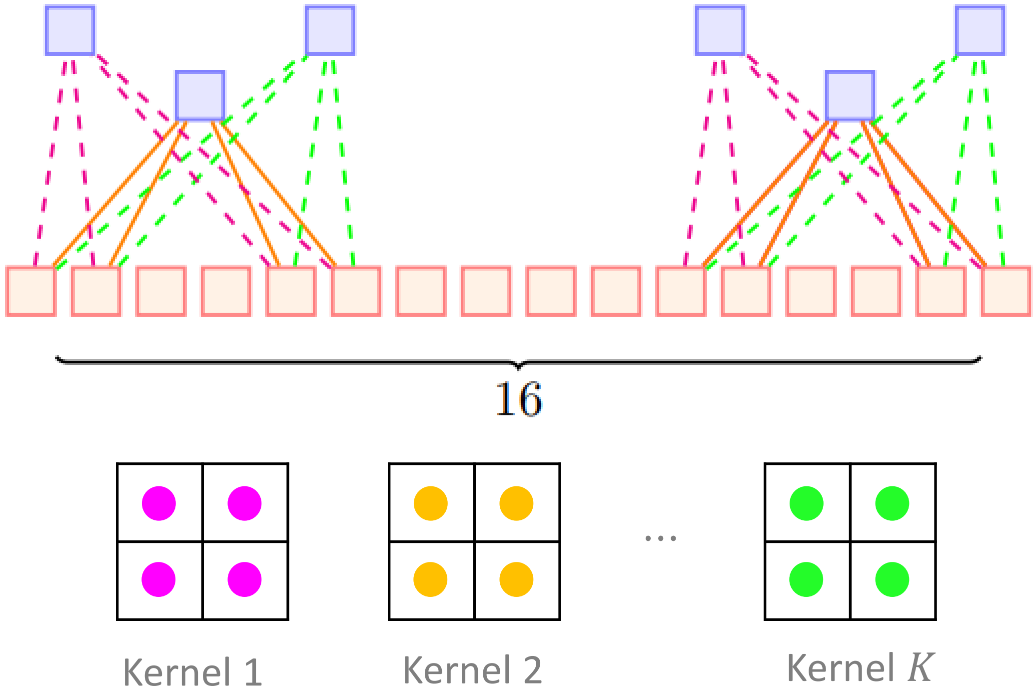
Full connected NN vs CNN





- Another characteristic of CNNs is **weight sharing**
- Imagine if we use an edge detection kernel
- Then the same kernel is passed over the all locations of the image to produce $h_1^{(1)}, h_2^{(1)}, h_3^{(1)}, h_4^{(1)}, \cdots$
- Since the kernel weights remain same as we sweep across all locations of the image, it is as if we share the weights across all locations of the image

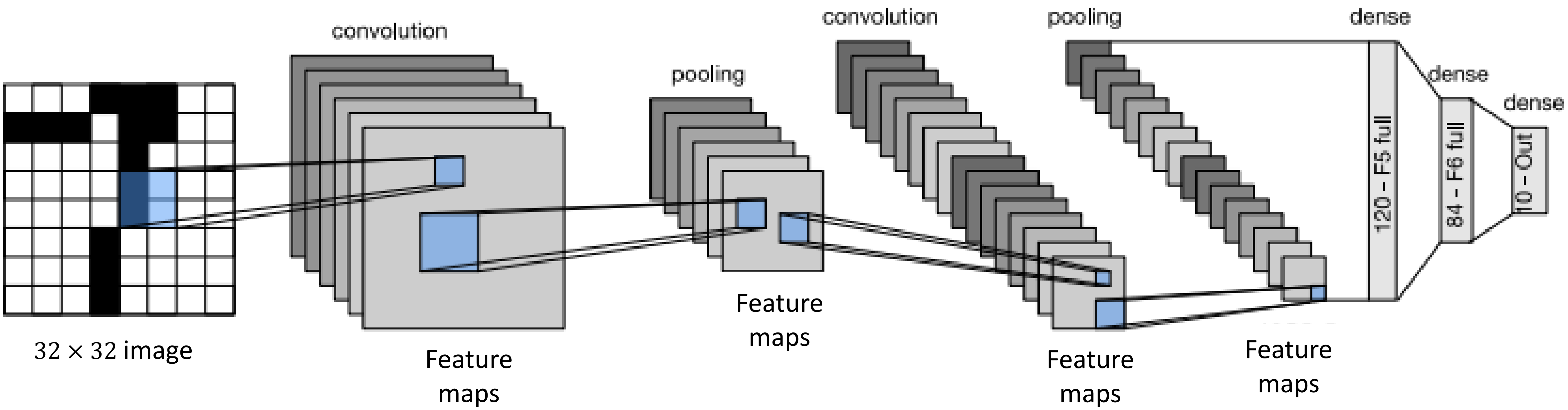
Full connected NN vs CNN



- Another characteristic of CNNs is **weight sharing**
- Imagine if we use an edge detection kernel
- Then the same kernel is passed over the all locations of the image to produce $h_1^{(1)}, h_2^{(1)}, h_3^{(1)}, h_4^{(1)}, \cdots$
- Since the kernel weights remain same as we sweep across all locations of the image, it is as if we share the weights across all locations of the image
- Note, we can have many such kernels and each kernel will be shared by all locations in the image

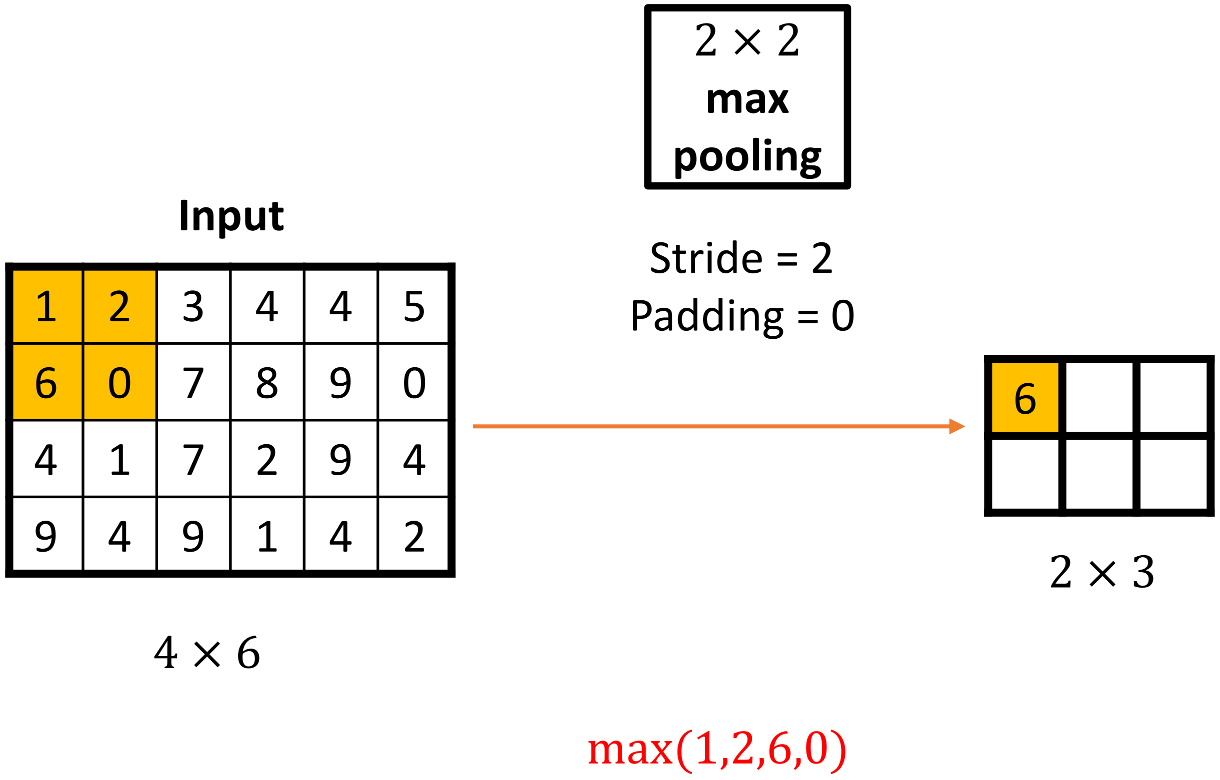
- So far we have talked a lot on convolution layers
- Saw how kernels are convolved with inputs to produce features
- Understood that kernels are to be learned (or optimized), not manually set
- Let’s look at CNN for a moment

Convolutional neural networks



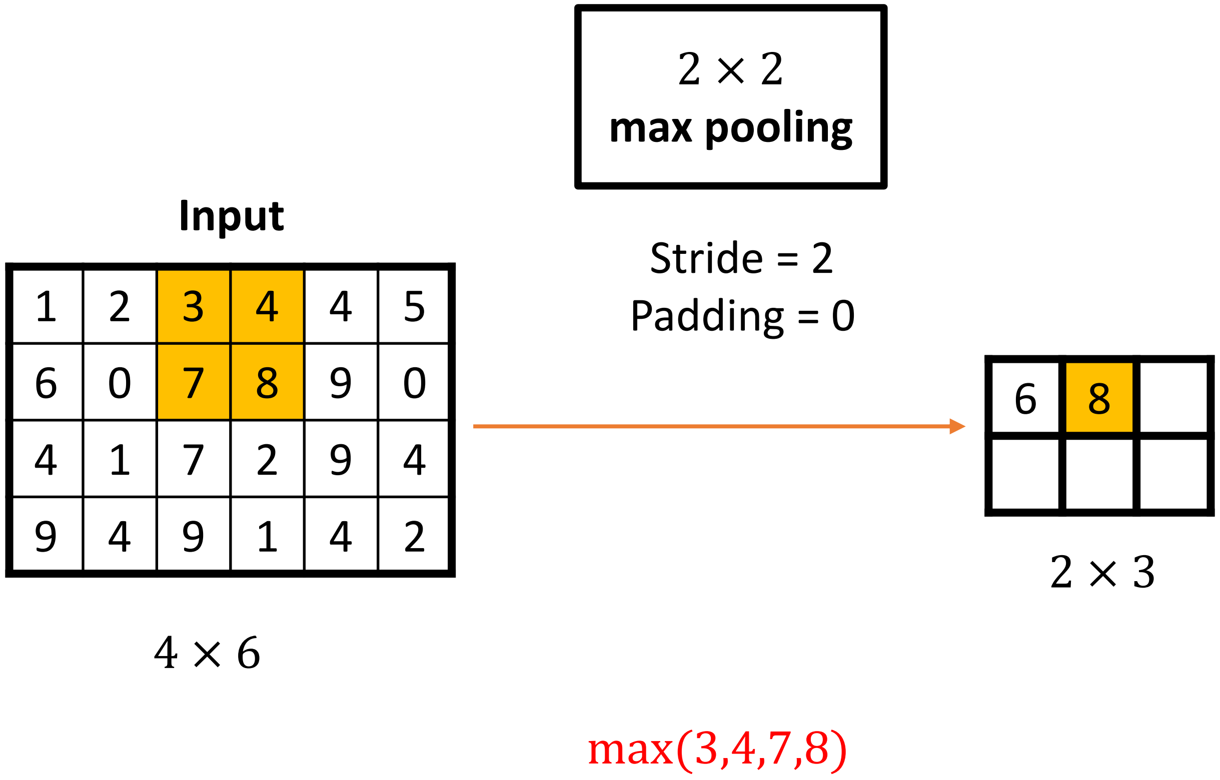
- So a CNN has alternate convolution and pooling layers
- What does a pooling layer do?

Pooling



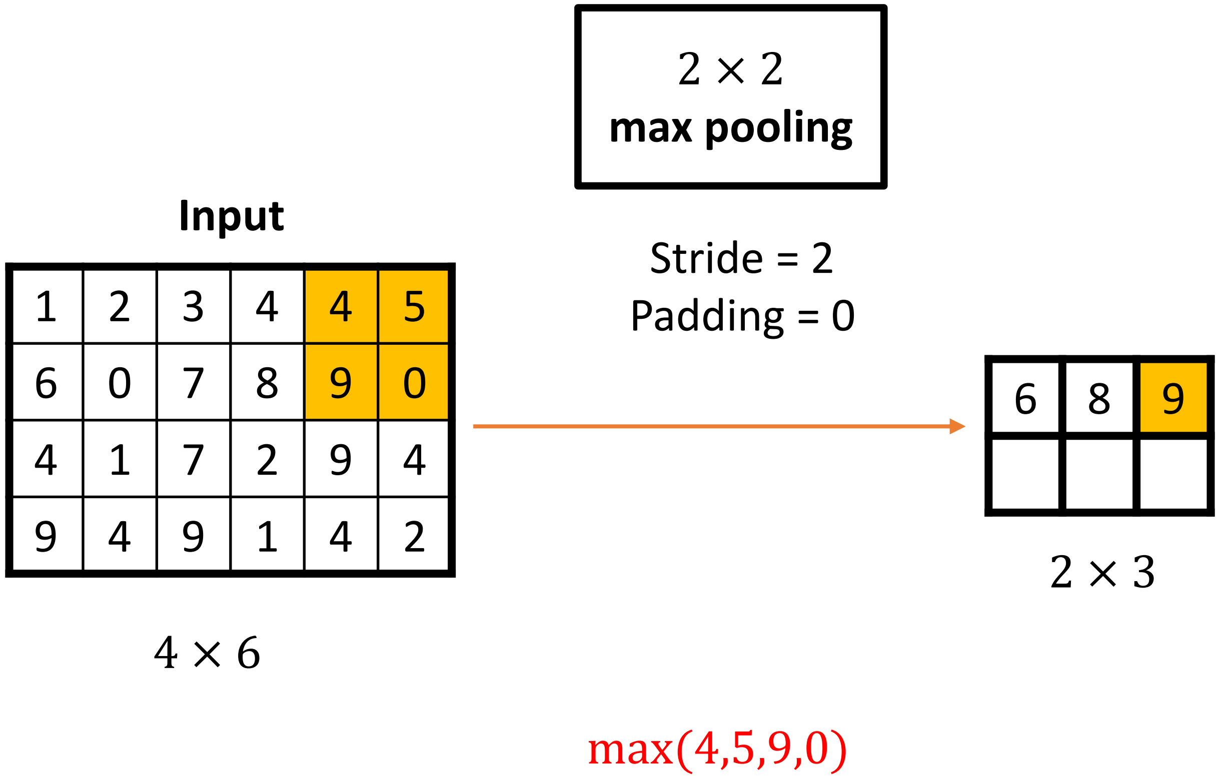
- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**

Pooling



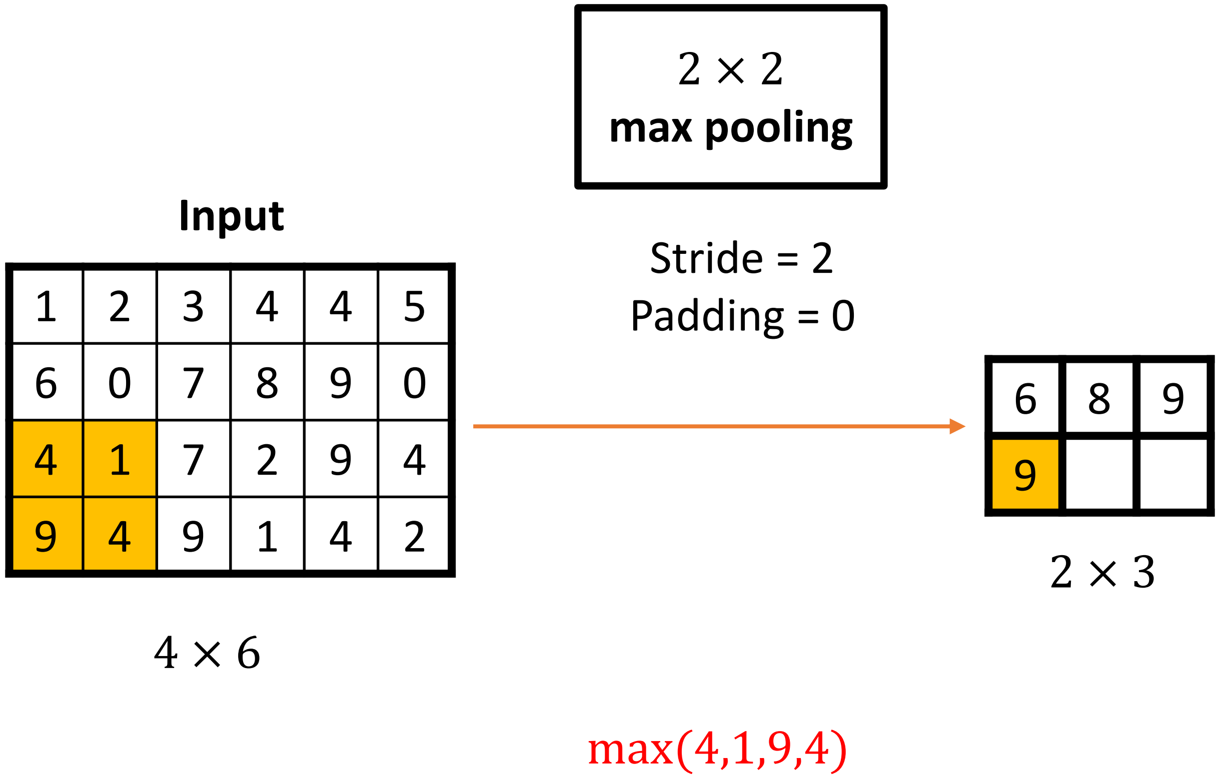
- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**

Pooling



- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**

Pooling



- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**

Pooling

Input

1	2	3	4	4	5
6	0	7	8	9	0
4	1	7	2	9	4
9	4	9	1	4	2

4×6

2×2
max pooling

Stride = 2
Padding = 0

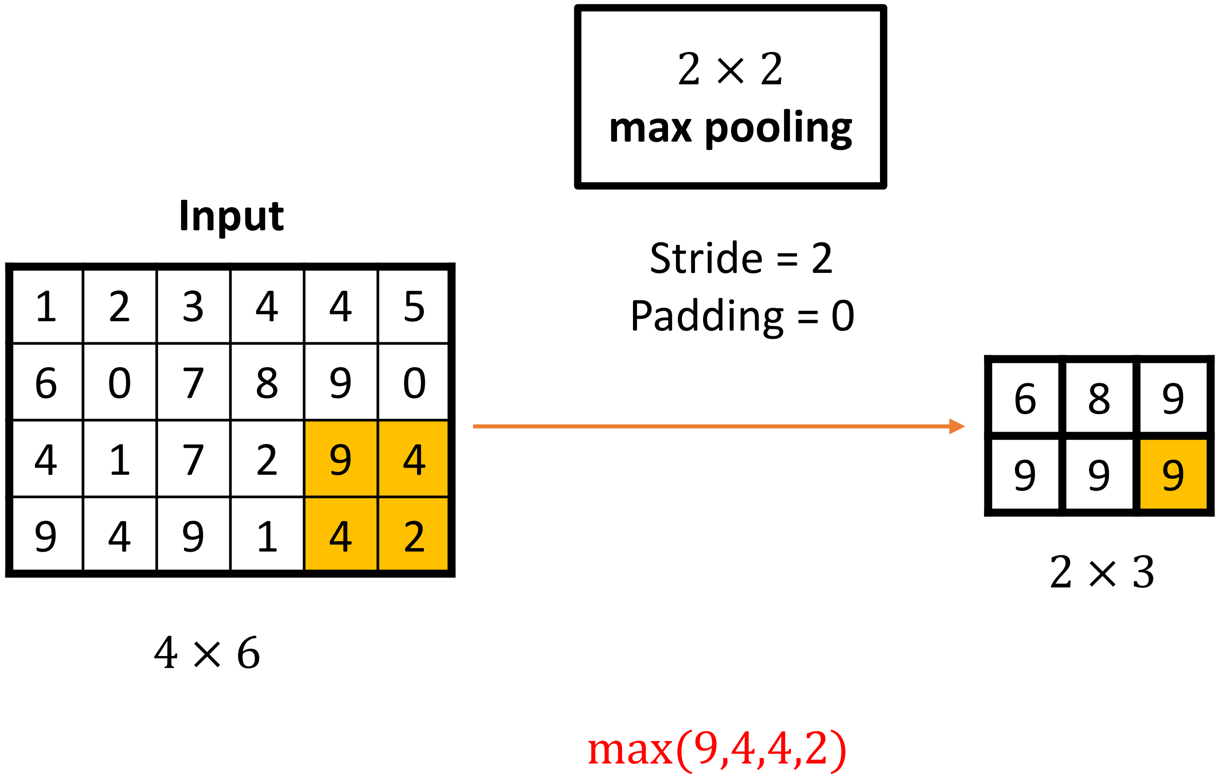
6	8	9
9	9	

2×3

$\max(7,2,9,1)$

- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**

Pooling



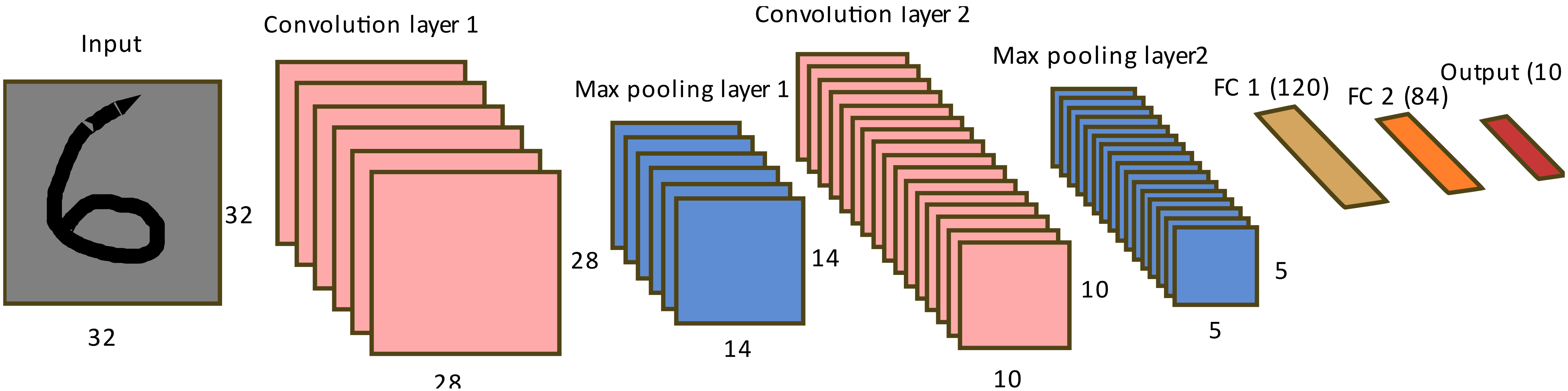
- We want to gradually reduce the spatial resolution of our hidden representations while aggregating meaningful features
- Pooling helps in reducing the spatial resolution
- Like convolutional layer, *pooling* operators consist of a fixed-shape window that is slid over all regions of the input according to its stride
- However unlike convolutional layer, the pooling layer contains no parameters (there is no *kernel*)
- Mostly, we take the maximum of the elements in the pooling window – **max pooling**
- Max pooling gets feature representation that is somewhat invariant to translation (recall we wanted to find Waldo irrespective of its location in image)
- There is also **average pooling**, taking average of the elements in the window

- Now we have all the ingredients to assemble a CNN

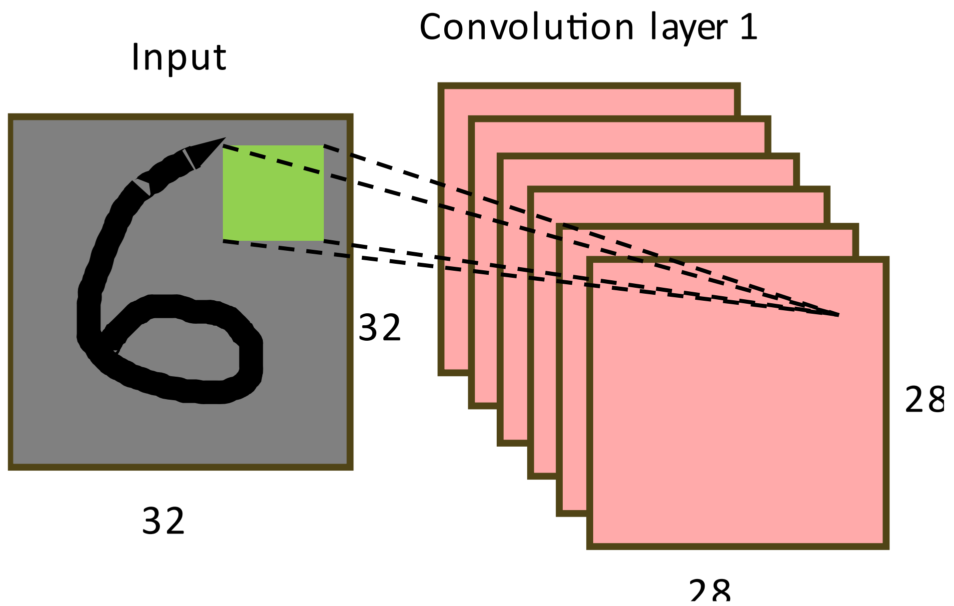
- We will now see the first CNN – *LeNet* (1998) by Yann LeCun for handwritten digit recognition

LeNet for handwritten digit recognition

- We have a **grayscale image** of an handwritten digit of size 32 x 32 with depth = 1
- This is going to be our input to LeNet



LeNet for handwritten digit recognition



$$\left\lfloor \frac{32-5-0}{1} + 1 \right\rfloor$$

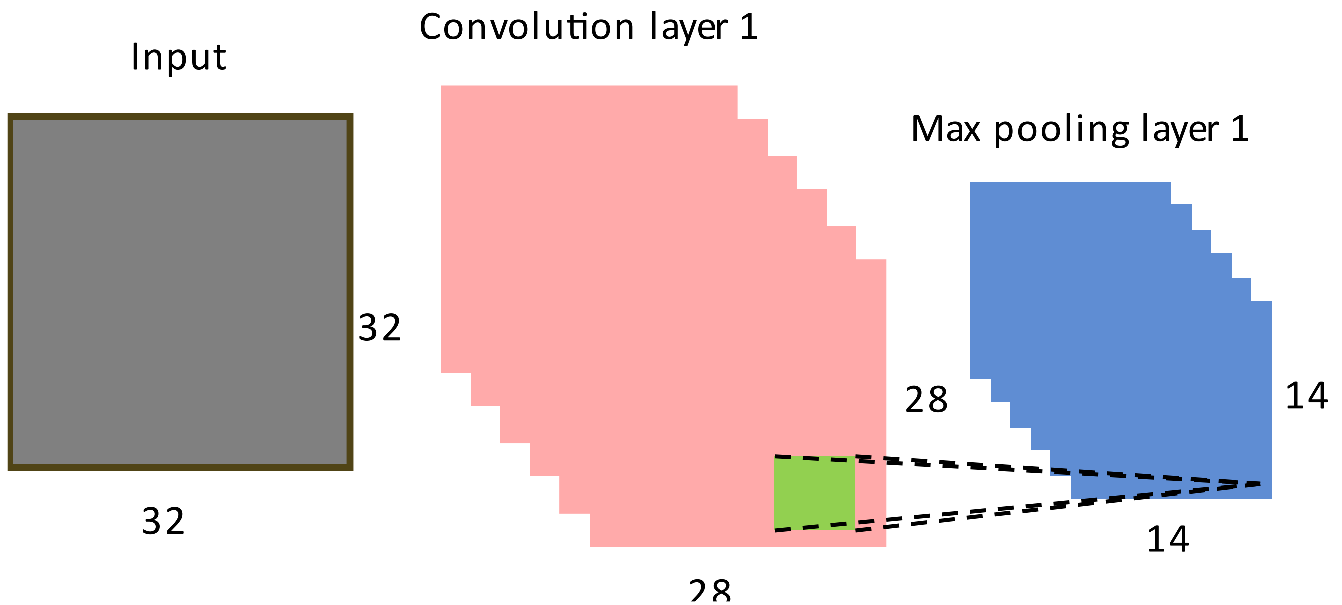
$= 28$

- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 5 \times 5$
- # kernels $\rightarrow 6$

- Parameters $\rightarrow$

- We have 6 kernels
- Each kernel has 5x5 = 25 weights
- # parameters = 25x6 = 150
- Input size = 32x32 = 1024
- Output size = 28x28 = 784
- If this was a fully-connected network, you needed 1024 x 784 weights!
- For convolution layer, we have just 150 parameters
- Great reduction in # of parameters
- A sigmoid activation was applied (ReLU was not known then)

LeNet for handwritten digit recognition



- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 1 \times 5 \times 5$
- # kernels $\rightarrow$ 6
- Parameters $\rightarrow$

- Stride $S = 2$
- Pad $P = 0$
- Kernel $\rightarrow 2 \times 2$
- Parameters $\rightarrow$

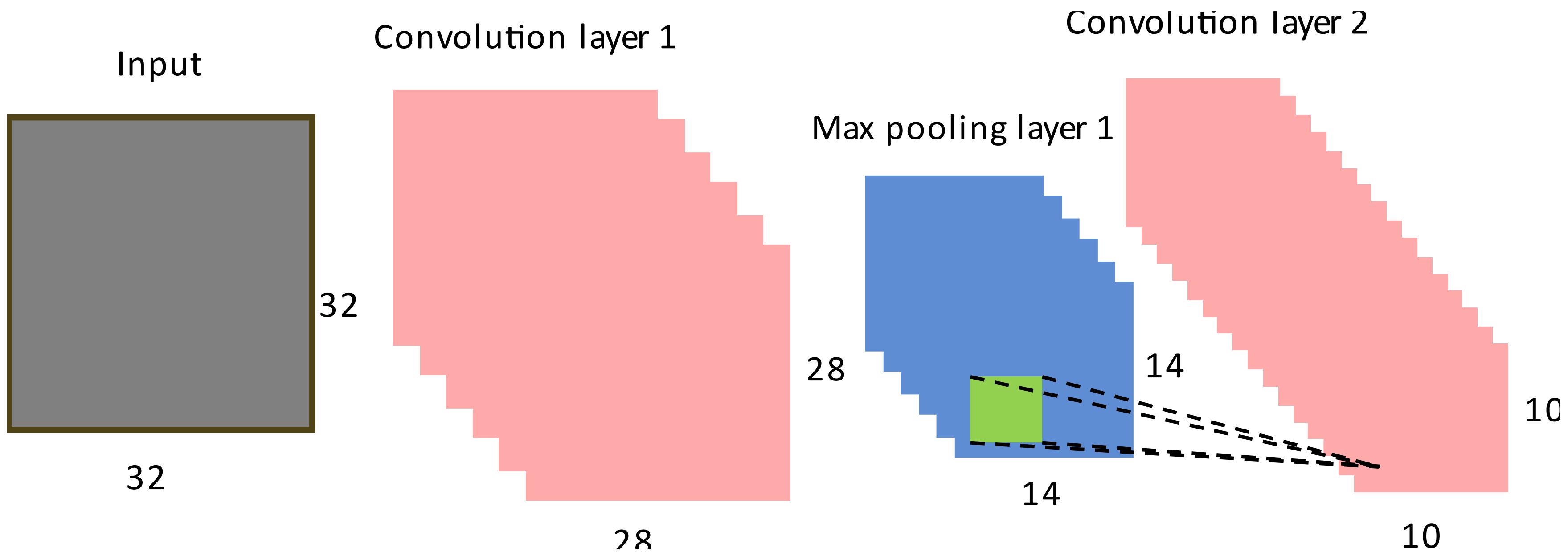
- Max pooling is a per feature map operation
- Here the kernel size is 2 x 2
- It downscales the size of feature maps

$$f_h = \left\lfloor \frac{28 - 2 + 0}{2} + 1 \right\rfloor = 14$$

$$f_w = \left\lfloor \frac{28 - 2 + 0}{2} + 1 \right\rfloor = 14$$

- The depth of max pooling layer is the same as the preceding convolution layer; here depth is 6
- There are no parameters in max pooling layers; they just take maximum of elements in a window

LeNet for handwritten digit recognition



- Feature map size

$$f_h = \left\lfloor \frac{14 - 5 + 0}{1} + 1 \right\rfloor = 10$$

$$f_w = \left\lfloor \frac{14 - 5 + 0}{1} + 1 \right\rfloor = 10$$

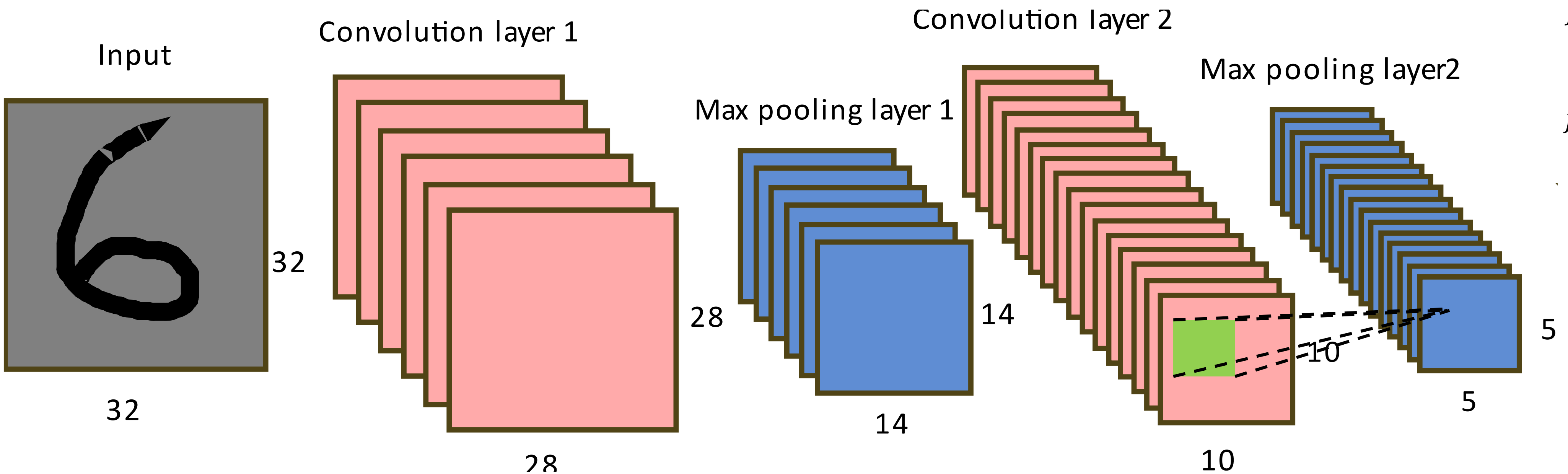
- Depth of kernels = 6
- Here kernel size is 5 x 5 x 6
- # parameters = 5 x 5 x 6 x 16
  = 2400

- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 5 \times 5$
- # kernels $\rightarrow$ 6
- Parameters $\rightarrow$

- Stride $S = 2$
- Pad $P = 0$
- Kernel $\rightarrow 2 \times 2$
- Parameters $\rightarrow$

- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 5 \times 5 \times 6$
- # kernels $\rightarrow$ 16
- Parameters $\rightarrow$

LeNet for handwritten digit recognition



$$f_h = \left\lfloor \frac{10 - 2 + 0}{2} + 1 \right\rfloor = 5$$

$$f_w = \left\lfloor \frac{10 - 2 + 0}{2} + 1 \right\rfloor = 5$$

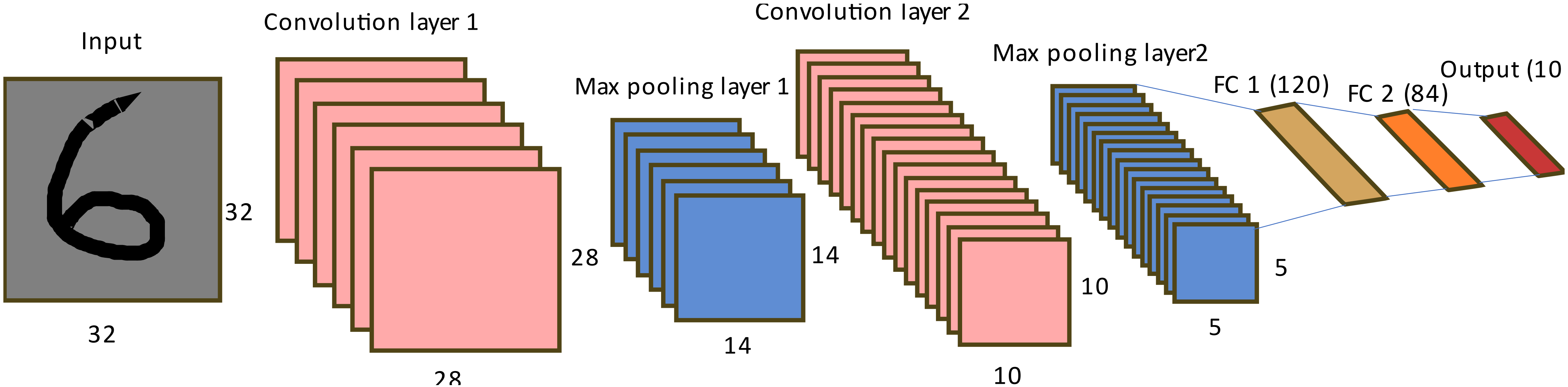
- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 5 \times 5$
- # kernels $\rightarrow$ 6
- Parameters $\rightarrow$

- Stride $S = 2$
- Pad $P = 0$
- Kernel $\rightarrow 2 \times 2$
- Parameters $\rightarrow$

- Stride $S = 1$
- Pad $P = 0$
- Kernel $\rightarrow 5 \times 5 \times 6$
- # kernels $\rightarrow 16$
- Parameters $\rightarrow$

- Stride S =2
- Pad $P = 0$
- Kernel $\rightarrow 2 \times 2$
- Parameters $\rightarrow 0$

LeNet for handwritten digit recognition



After max pooling layer 2, there are two fully connected hidden layers

- The features of the max pooling layer is flattened out into a vector of size 16x5x5 = 400 and fed to FC 1 layer as inputs
- FC 1 layer has 120 hidden units → (16x5x5) x 120 = 48000 weights + 120 biases = 48120 parameters
- FC 2 layer has 84 hidden units → 120 x 84 = 10080 weights + 84 biases = 10164 parameters
- Output layer has 10 classes → 84 x 10 = 840 weights + 10 biases = 850 parameters
- The entire network can be trained using back propagation