

ECE 1508S2: Applied Deep Learning

Chapter 4: Convolutional Neural Networks

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering
University of Toronto

Winter 2024

Computer Vision

Computer vision has been a fundamental problem in machine learning

The aim is to design a machine that can recognize patterns in visual contents

Long study on biological vision systems has been conducted

- Hubel and Wiesel studies visual cortex in late 1950s
 - ↳ They won Nobel Prize in 1981 for their discoveries
- Inspired by that study Fukushima introduced Neocognitron in 1979
 - ↳ It was a convolutional NN for unsupervised learning
- Yann LeCun proposed LeNet for supervised learning in 1989
 - ↳ This is pretty much the beginning of modern CNNs

Convolutional NNs: CNNs

Let's make an agreement: *though we all know CNN News Channel we also say*

Convolutional NN ≡ CNN

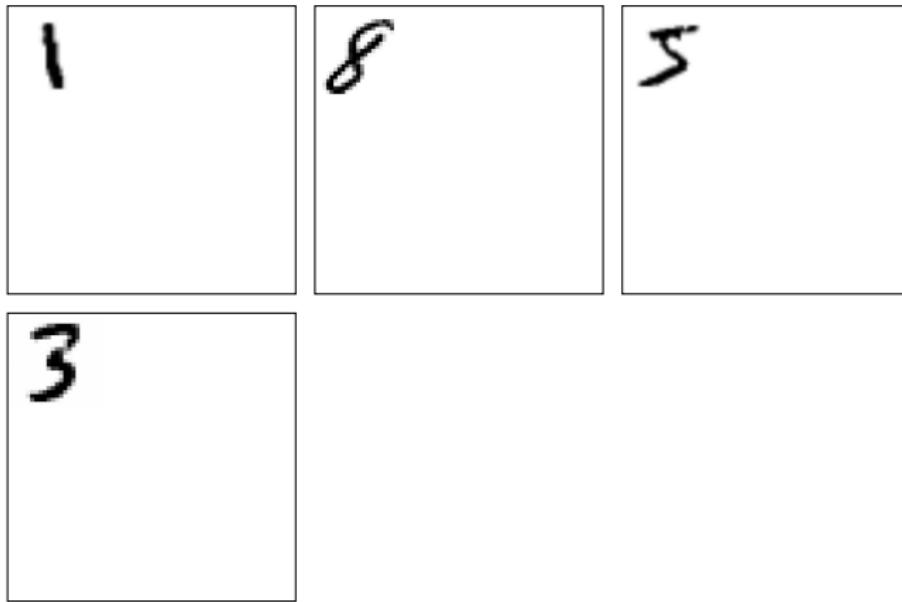
CNN

CNN is an **FNN** which in addition to standard fully-connected layers uses **convolutional** and **pooling** layers for **feature extraction**

- + But, what is a **convolutional** layer? What is a **pooling** layer? What do you mean by **feature extraction**?
- Well! We get there **soon**! But first let's see what the main **motivation** is

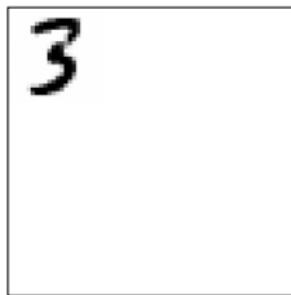
Motivational Example: Pattern Recognition

We intend to train an **NN** that classifies **MNIST** dataset with **one difference** to our earlier classification problem: *we assume that MNIST images are now included in a larger white background*



Motivational Example: Pattern Recognition

- + Can't we simply use standard **fully-connected** FNN?
- Sure! We can



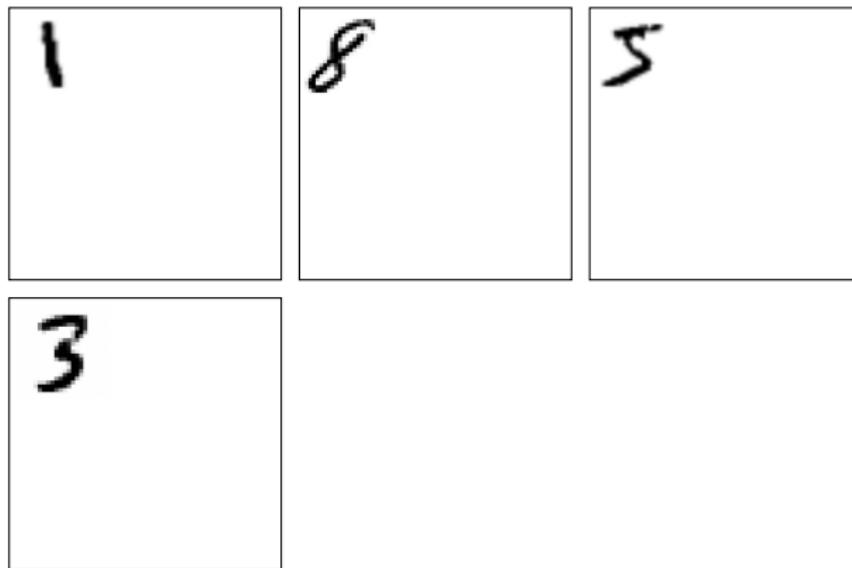
Say the photos with background are **3 times larger** in height and width

- **Images** are now $84 \times 86 \equiv$ we have **7056 pixels**
- We should have **input layer with 7056 pixels** and train the NN with **MNIST**

To do the training, we should first **convert** our **MNIST images** into new **larger-size images**

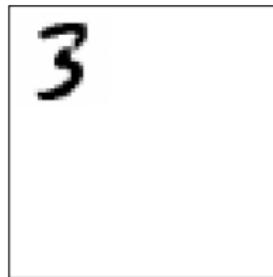
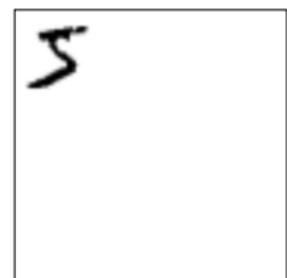
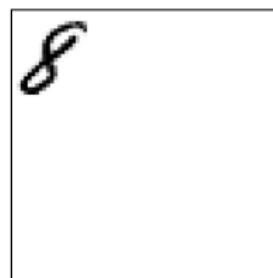
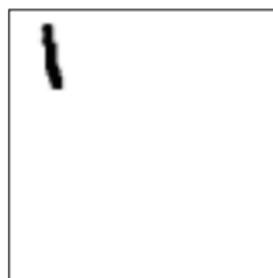
Motivational Example: Pattern Recognition

We do this by **zero-padding**: all **images after conversion lie on top left corner**



Motivational Example: Pattern Recognition

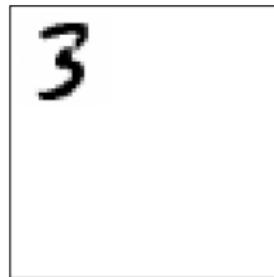
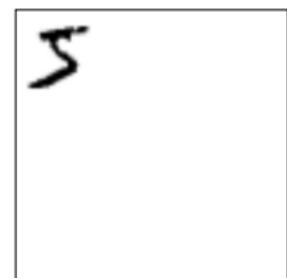
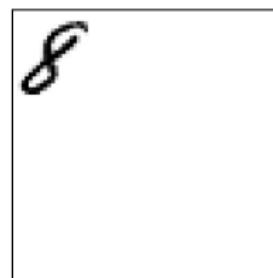
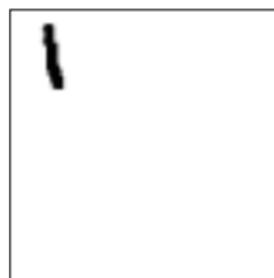
We do this by **zero-padding**: all images after conversion lie on **top left corner**



Do you think after training, NN classifies "3" in **lower right corner** correctly?

Motivational Example: Pattern Recognition

We do this by **zero-padding**: all images after conversion lie on **top left corner**



Do you think after training, NN classifies "3" in **lower right corner** correctly? **No!**

Motivational Example: Pattern Recognition

- + Why does this happen?
- Our NN is trained to only look at top left corner, it will miss information anywhere else including in lower right corner

Motivational Example: Pattern Recognition

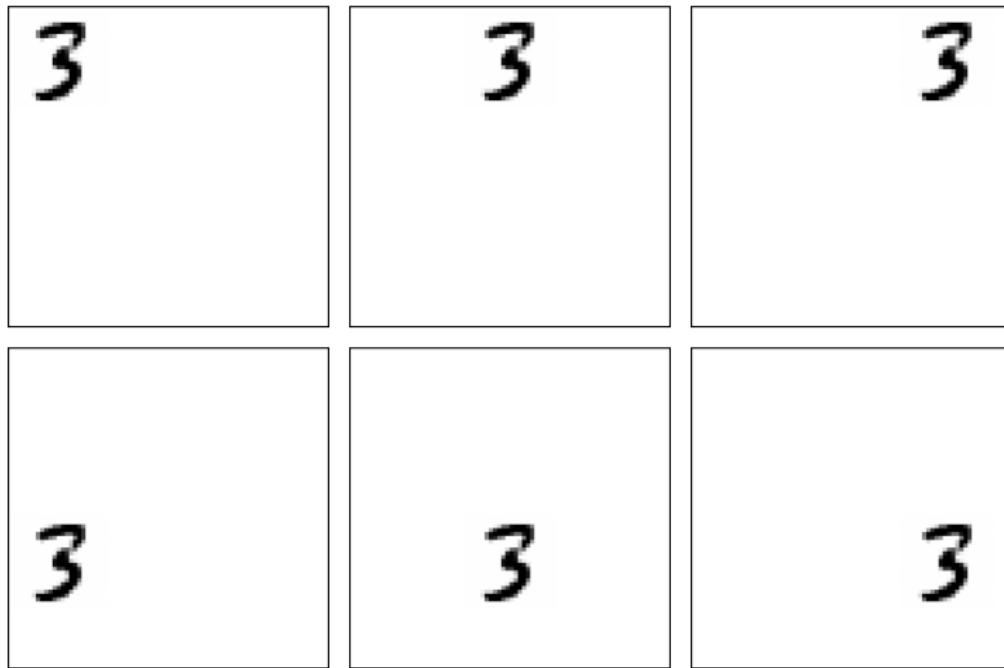
- + Why does this happen?
- Our NN is trained to only look at top left corner, it will miss information anywhere else including in lower right corner
- + Can we do anything about it?

Motivational Example: Pattern Recognition

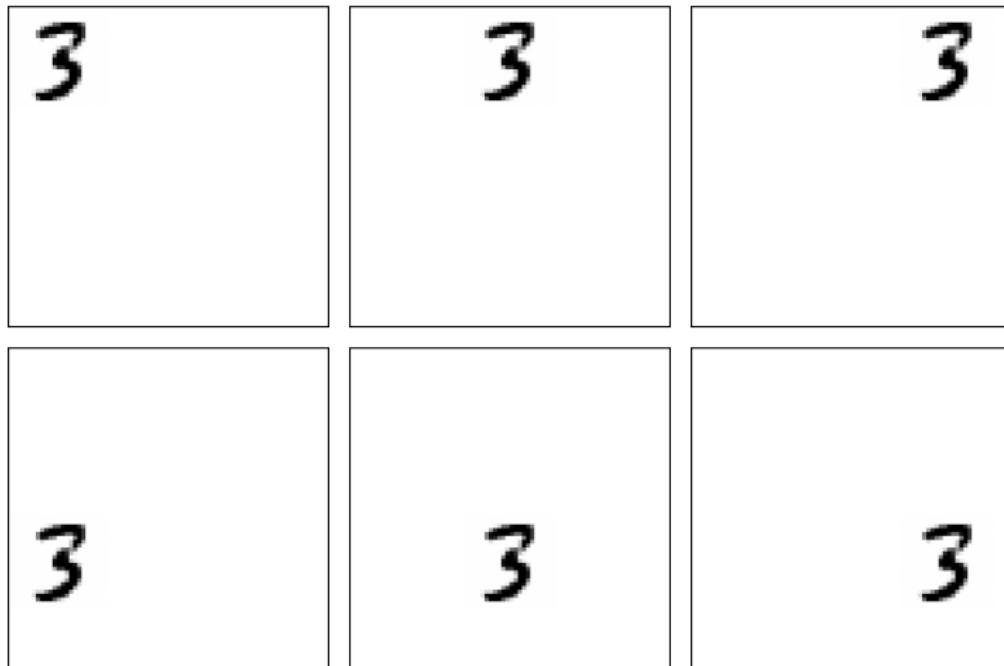
- + Why does this happen?
- Our NN is trained to only look at top left corner, it will miss information anywhere else including in lower right corner
- + Can we do anything about it?
- Yes! We learned it in the last chapter: Data Augmentation

We shift MNIST images in the large background left and right, up and down and add all those shifts with same label to the dataset

Motivational Example: Recognition with Augmented Data



Motivational Example: Recognition with Augmented Data



We should get **too many** of them!

Using a Trained FNN

- + But it sounds like **too much work** and **computation!**
- Yes! It is! and frankly speaking **it is not worth it!**

Many scientist noted that our **brain** doesn't work **like that**

once we learn "3" we can **recognize** it **anywhere** in our vision!

Using a Trained FNN

- + But it sounds like **too much work** and **computation!**
- Yes! It is! and frankly speaking **it is not worth it!**

Many scientist noted that our **brain** doesn't work **like that**

once we learn "3" we can **recognize** it **anywhere** in our vision!

We may **initially** note that

- ① Our brain **doesn't** process the **visual field as the whole**

Using a Trained FNN

- + But it sounds like **too much work** and **computation!**
- Yes! It is! and frankly speaking **it is not worth it!**

Many scientist noted that our **brain** doesn't work **like that**

once we learn "3" we can recognize it anywhere in our vision!

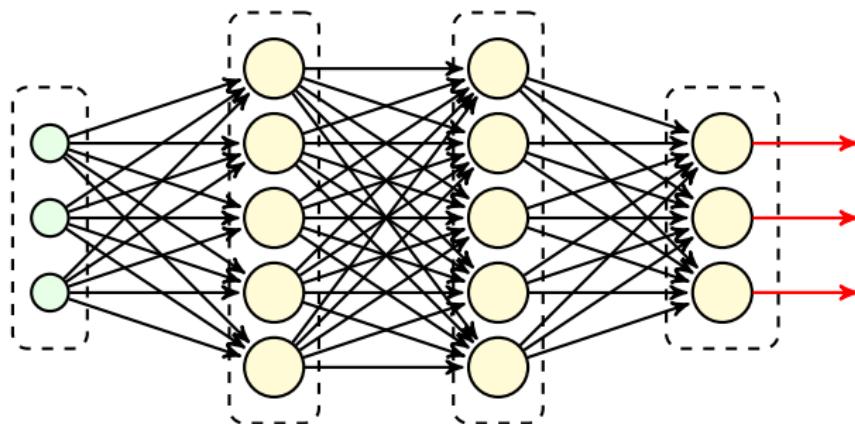
We may **initially** note that

- ① Our brain **doesn't** process the **visual field as the whole**
- ② It **searches** for patterns in **smaller fields** within our vision
 - ↳ It constructs a pattern for "3" through **training**
 - ↳ After **training**, it **scans** any visual field to see if it finds that pattern

Let's try realizing it with a NN!

Using a Trained FNN

Let's assume we have trained the following FNN on MNIST

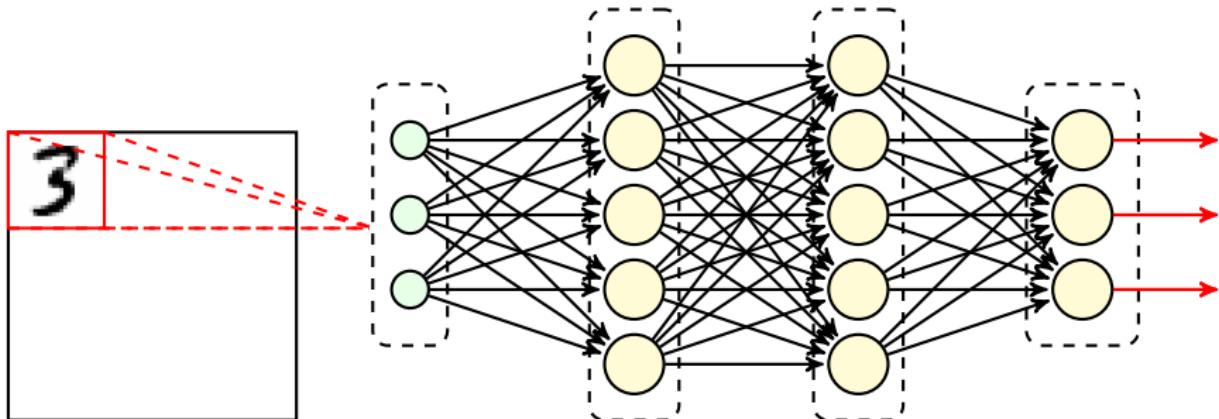


- It gets a 784-pixel image as input
- It passes it through 3 fully-connected layers
- It returns the class of the image
 - ↳ If it doesn't find a class it returns \emptyset

Scanning via MNIST Trained FNN

We can mimic what *our brain* does

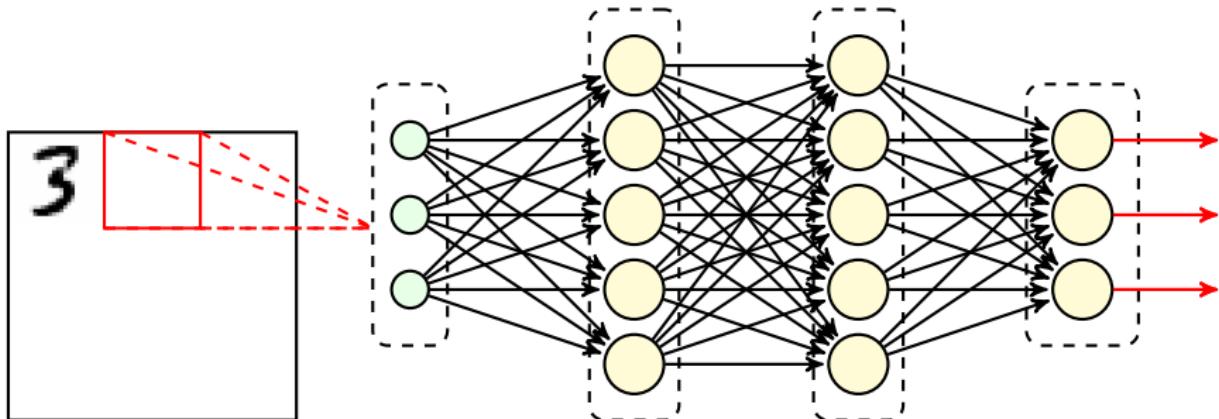
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

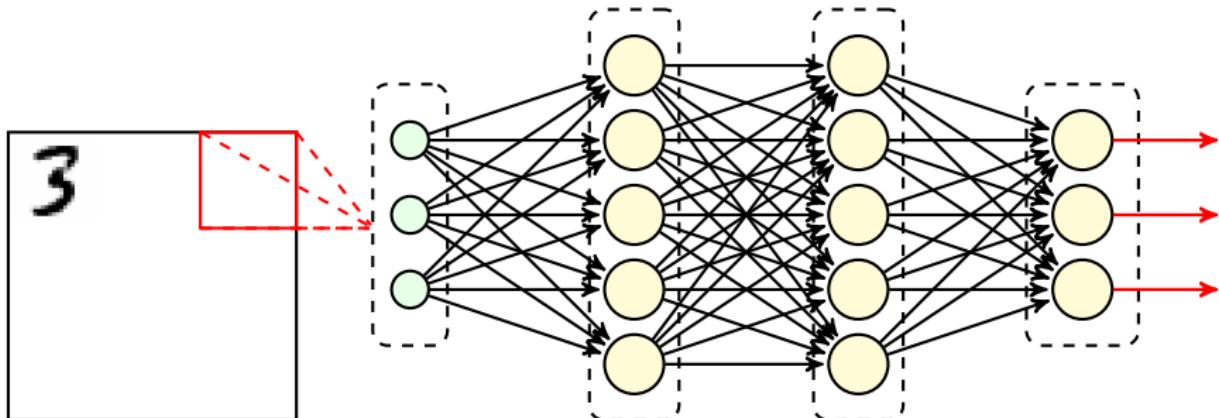
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

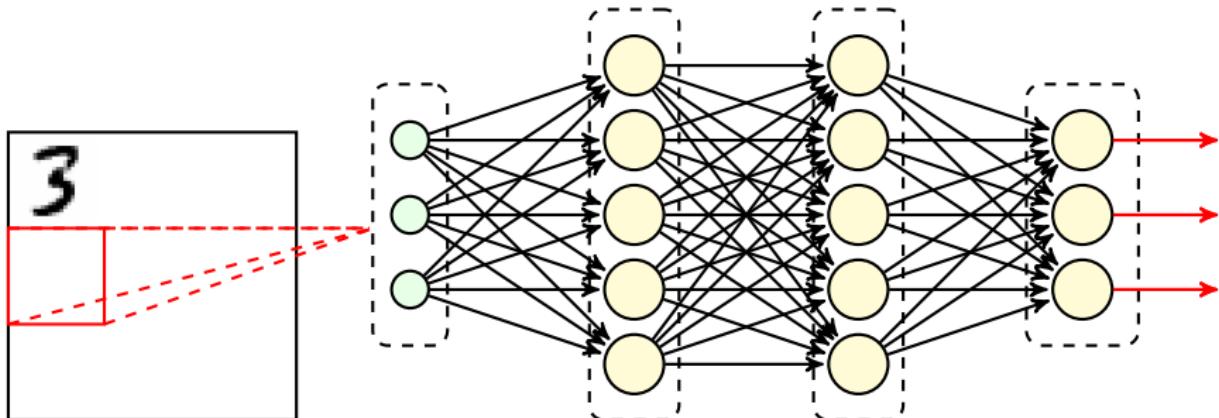
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

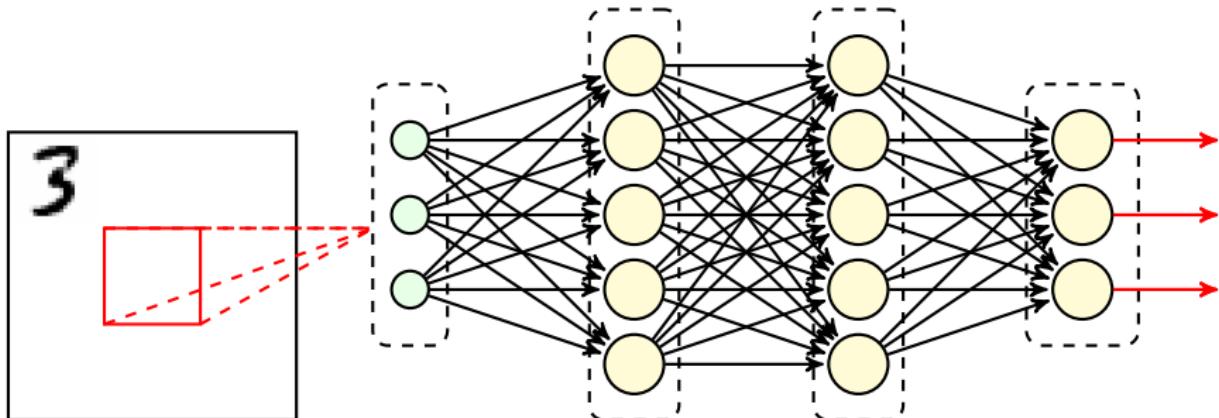
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

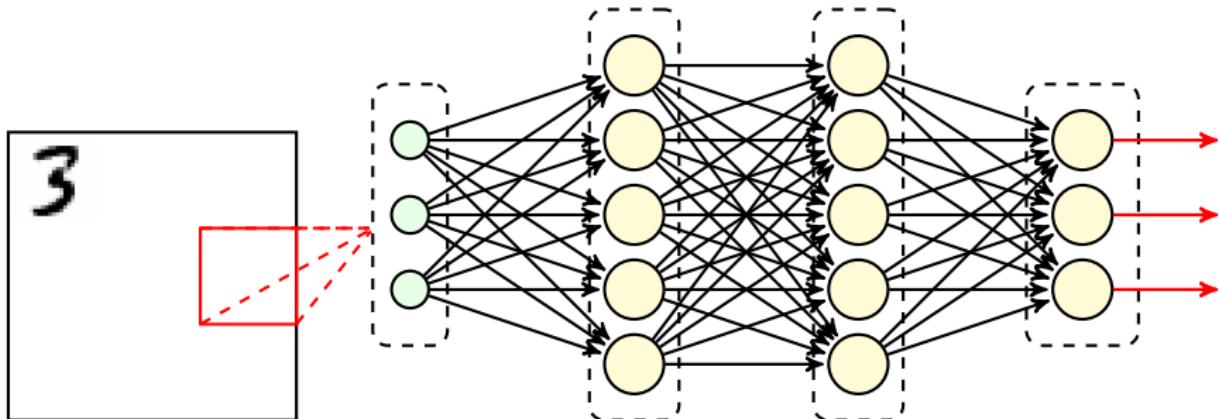
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

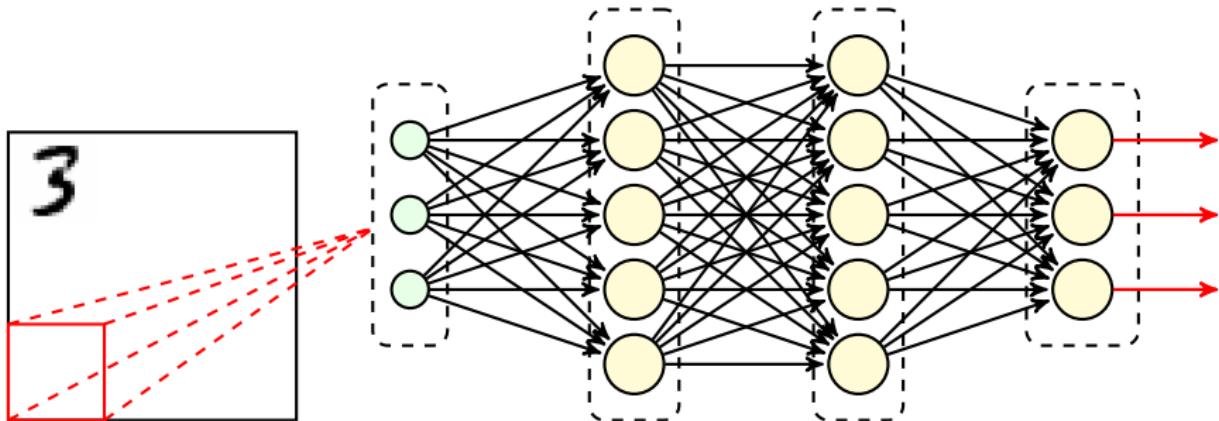
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

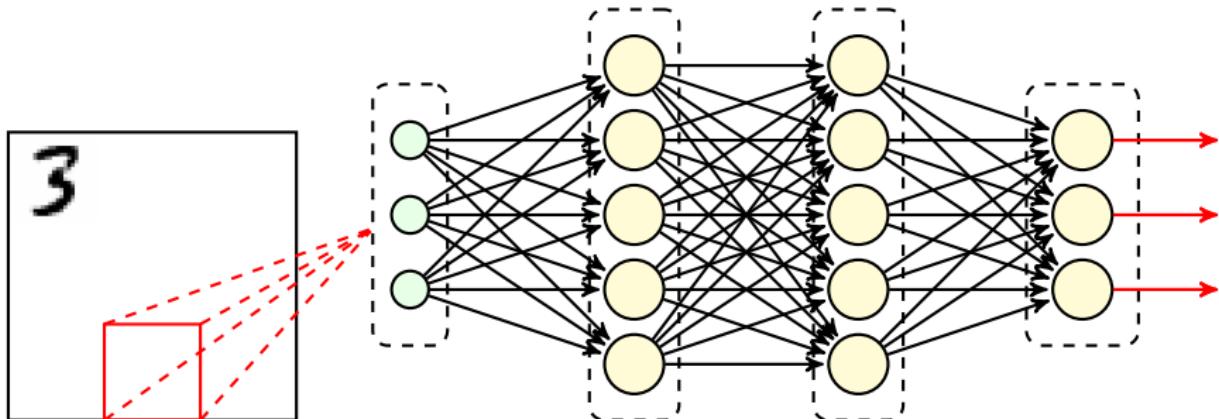
we *scan* the *larger image* with *background*



Scanning via MNIST Trained FNN

We can mimic what *our brain* does

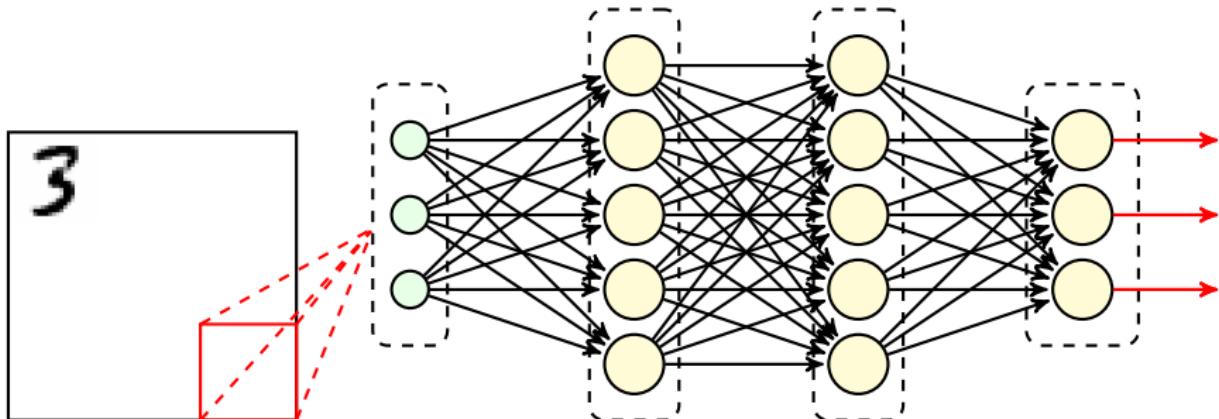
we *scan* the *larger image* with *background*



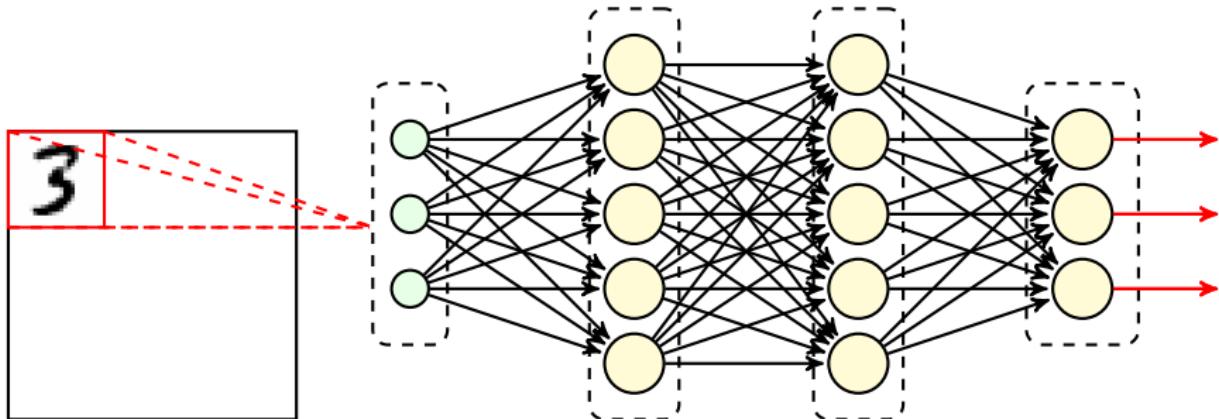
Scanning via MNIST Trained FNN

We can mimic what *our brain* does

we *scan* the *larger image* with *background*

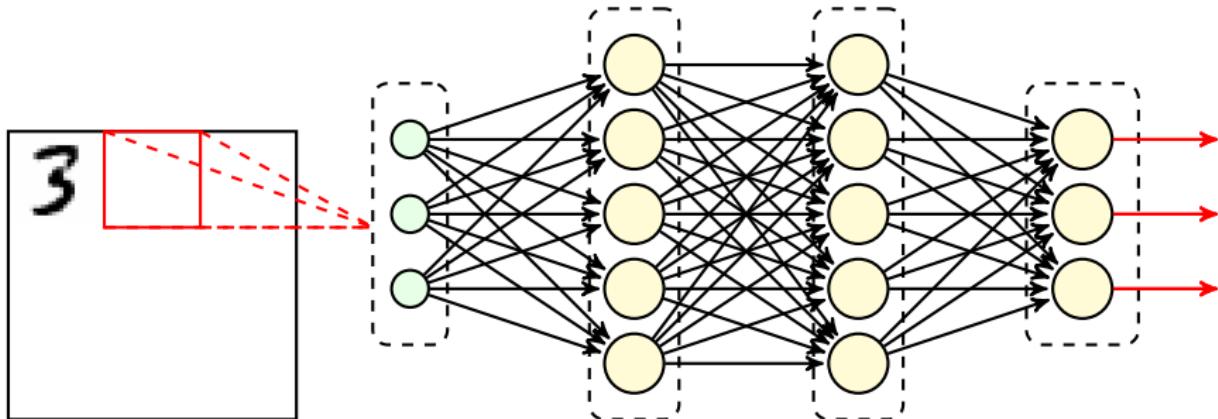


Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

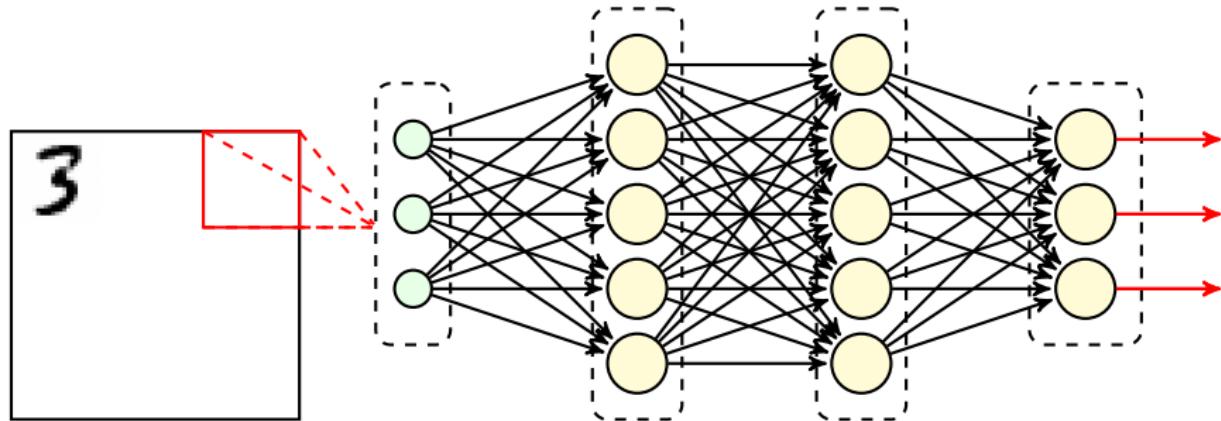
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**

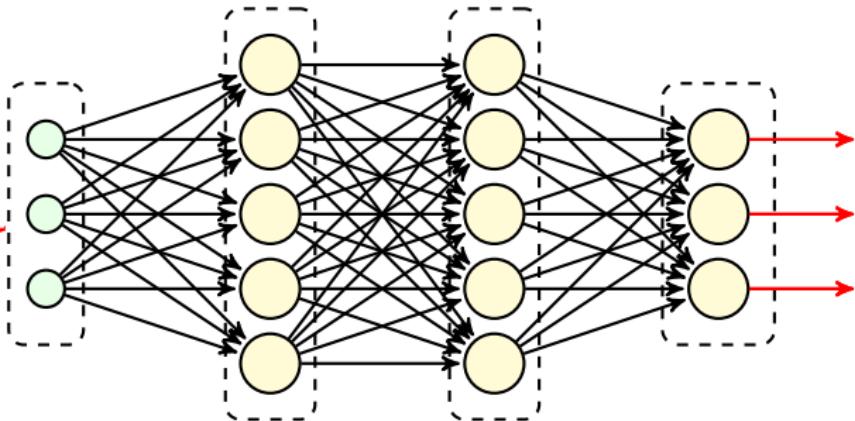
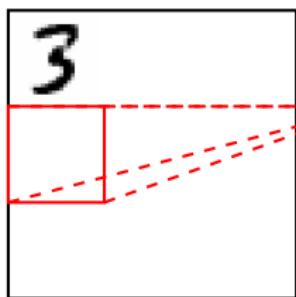
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

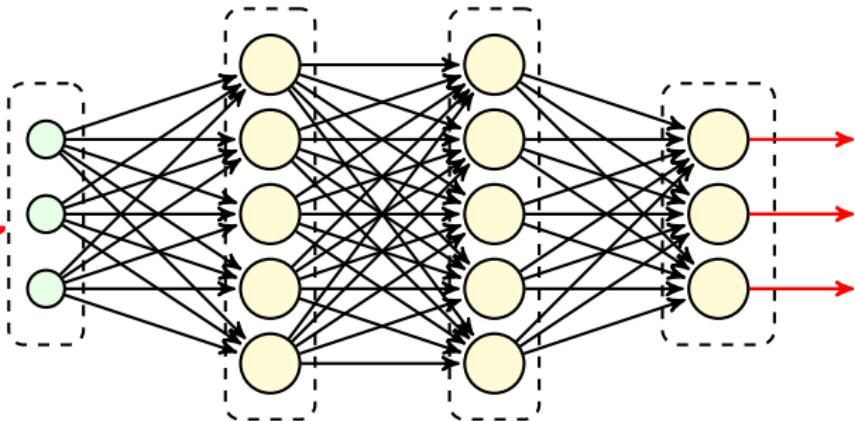
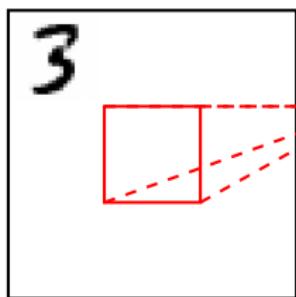
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

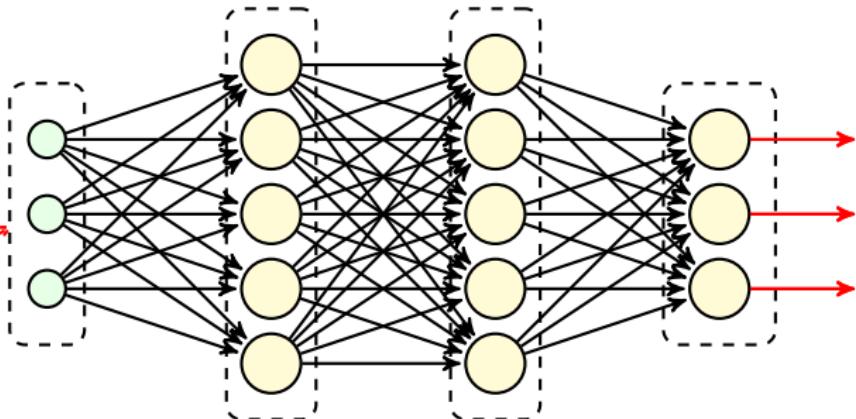
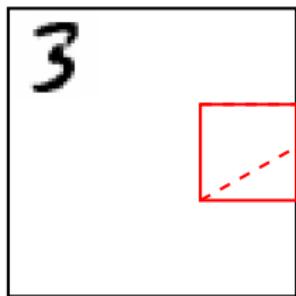
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

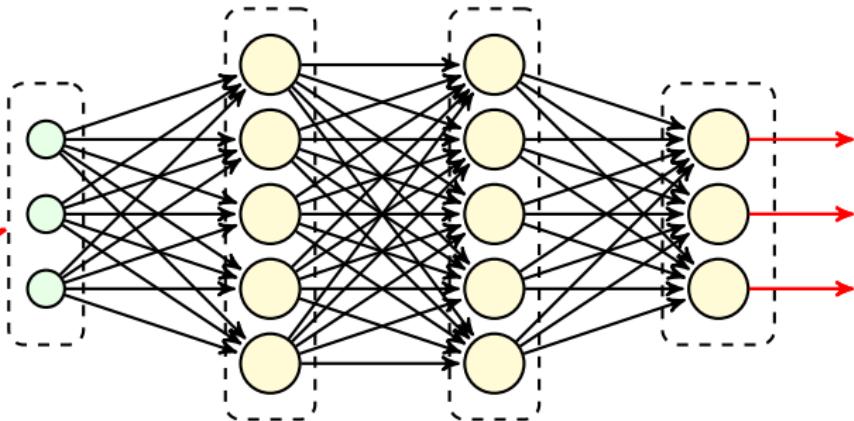
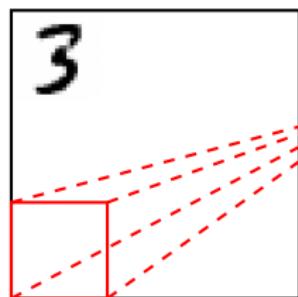
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

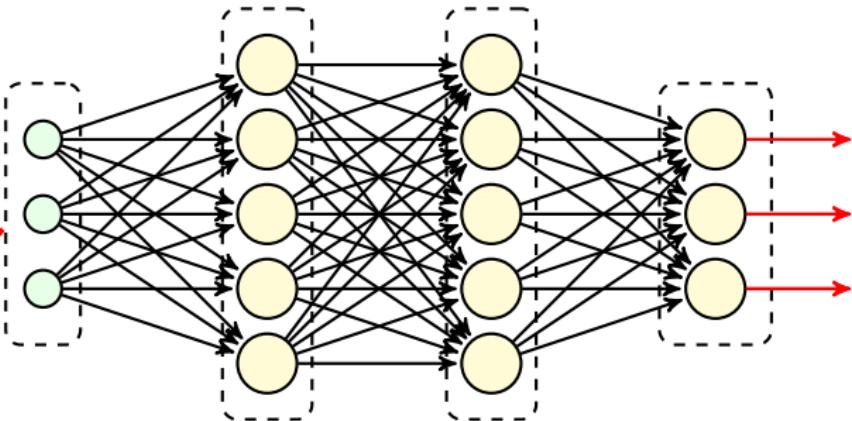
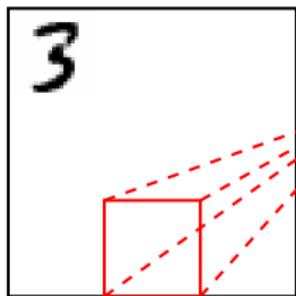
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

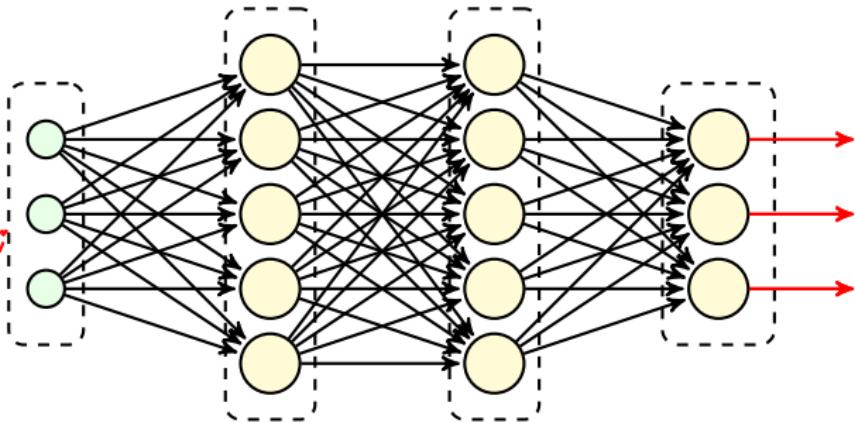
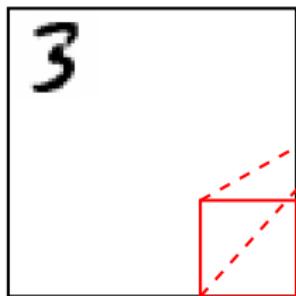
Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all windows
 - ↳ If True: we return **the saved class**
 - ↳ If False: we return **\emptyset**

Scanning via MNIST Trained FNN



We go through **windows** for size 28×28

- ① At each window, we give it **784 pixels** to the FNN to classify
 - ↳ If we find a class: we **save the class** and **return True**
 - ↳ If we don't find a class: we **return False**
- ② We compute **OR** of **outputs** for all **windows**
 - ↳ If **True**: we return **the saved class**
 - ↳ If **False**: we return **\emptyset**

CNNs: Scanning via Shared Weights

The above example is a *simple CNN*

CNNs: Scanning via Shared Weights

The above example is a *simple CNN*

- This CNN extracts features from the image using a 28×28 filter
 - ↳ the filter's weights are those given by the first layer of trained FNN
 - ↳ the features are affine values calculated in first layer of trained FNN

CNNs: Scanning via Shared Weights

The above example is a *simple CNN*

- This CNN extracts features from the image using a 28×28 filter
 - ↳ the **filter's weights** are those given by the first layer of trained FNN
 - ↳ the **features** are affine values calculated in **first layer of trained FNN**
- The scanning procedure has a specific name: convolution
 - ↳ it goes through the image by **sliding** over it via a **smaller window**
 - ↳ it determines an **affine transform** of smaller subsets of pixels

CNNs: Scanning via Shared Weights

The above example is a *simple CNN*

- This CNN extracts features from the image using a 28×28 filter
 - ↳ the **filter's weights** are those given by the first layer of trained FNN
 - ↳ the **features** are affine values calculated in **first layer of trained FNN**
- The scanning procedure has a specific name: convolution
 - ↳ it goes through the image by **sliding** over it via a **smaller window**
 - ↳ it determines an **affine transform** of smaller subsets of pixels
- We can look at it as a **giant FNN** with **shared** weights
 - ↳ each pixel is connected to the next layer via **affine transform**
 - ↳ this affine transform has the **same weights** for many pixels
 - ↳ **not every feature** depends on **every pixels**
 - ↳ the first layer is **not fully-connected**: it's **locally-connected**

CNNs: Scanning via Shared Weights

The above example is a *simple CNN*

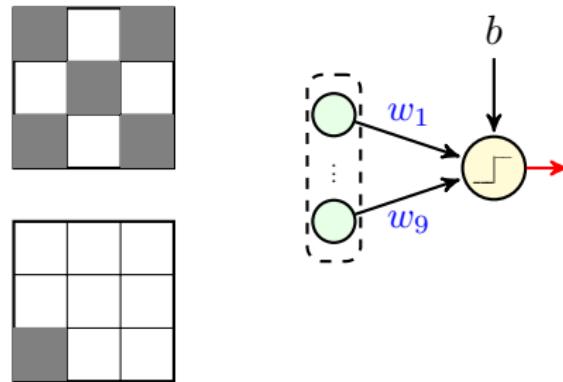
- This CNN extracts features from the image using a 28×28 filter
 - ↳ the **filter's weights** are those given by the first layer of trained FNN
 - ↳ the **features** are affine values calculated in **first layer of trained FNN**
- The scanning procedure has a specific name: convolution
 - ↳ it goes through the image by **sliding** over it via a **smaller window**
 - ↳ it determines an **affine transform** of smaller subsets of pixels
- We can look at it as a **giant FNN** with **shared** weights
 - ↳ each pixel is connected to the next layer via **affine transform**
 - ↳ this affine transform has the **same weights** for many pixels
 - ↳ **not every feature** depends on **every pixels**
 - ↳ the first layer is **not fully-connected**: it's **locally-connected**

Let's make our understanding deeper by making our first CNN!

Recognizing X

In Assignment 1, we **trained** a perceptron with 9 inputs

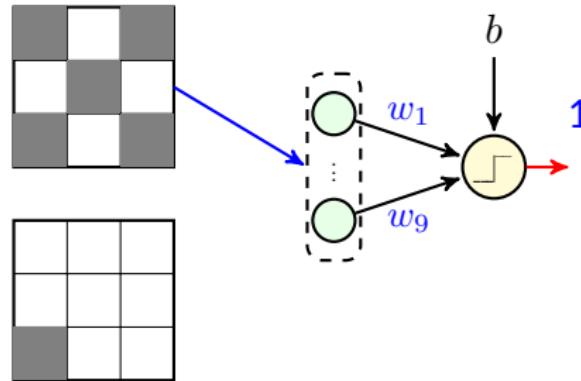
it gets a 3×3 image and says whether it is “X” or not



Recognizing X

In Assignment 1, we **trained** a perceptron with 9 inputs

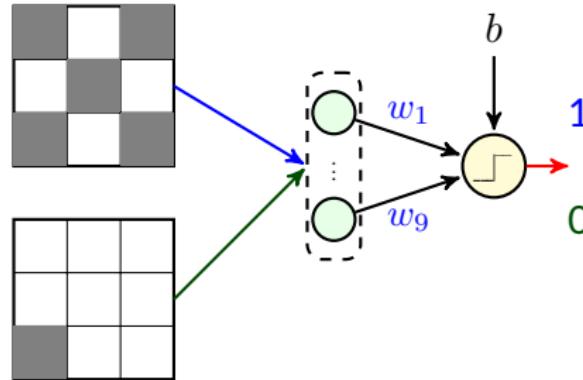
it gets a 3×3 image and says whether it is “X” or not



Recognizing X

In Assignment 1, we **trained** a perceptron with 9 inputs

it gets a 3×3 image and says whether it is “X” or not

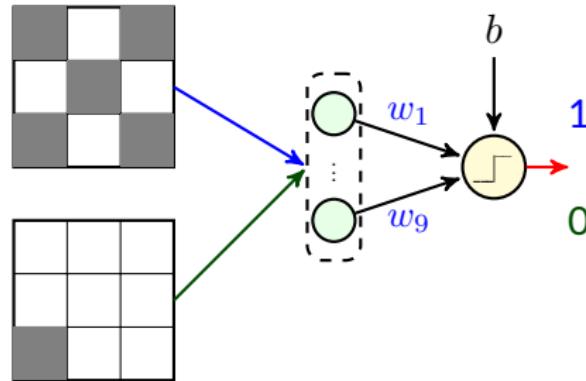


Recognizing X

In Assignment 1, we **trained** a perceptron with 9 inputs

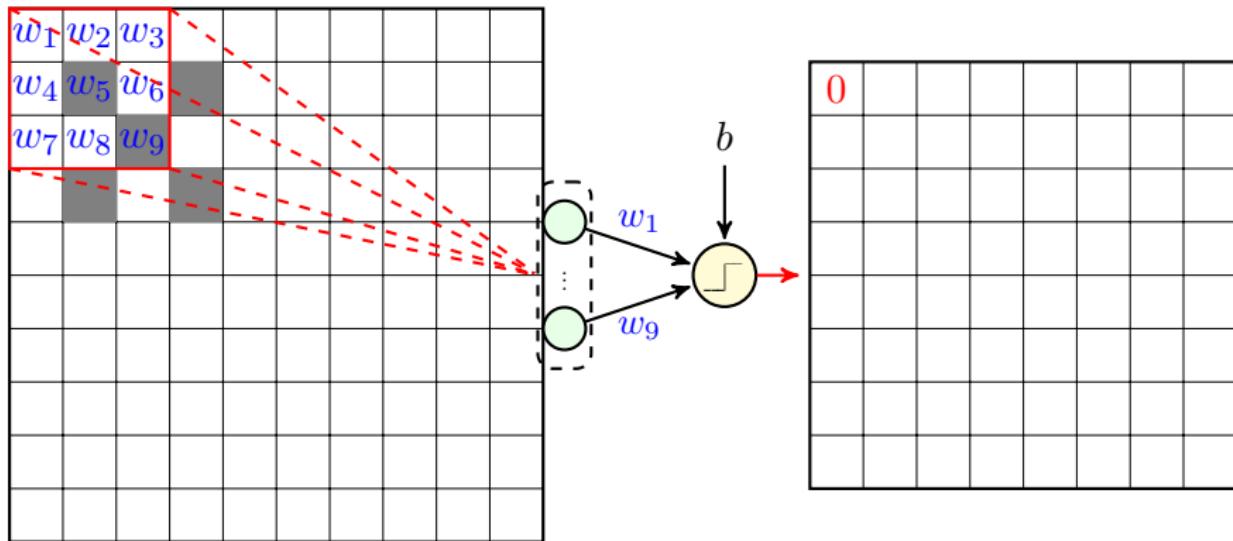
it gets a 3×3 image and says whether it is “X” or not

Assume we have **weights and bias**: we want to **recognize “X” in a larger image**



Recognizing X

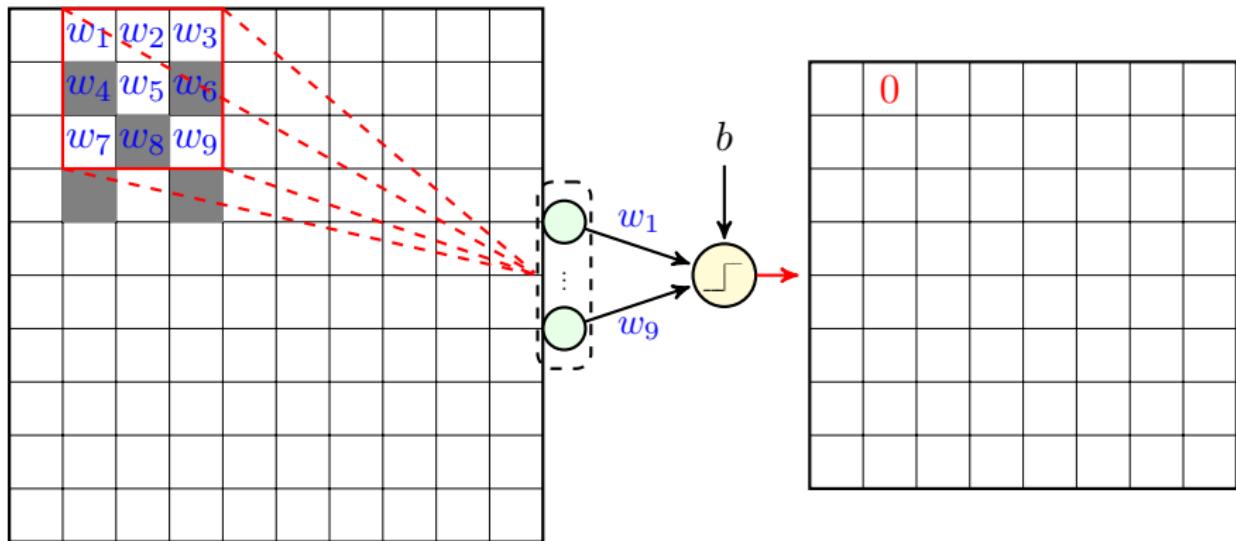
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

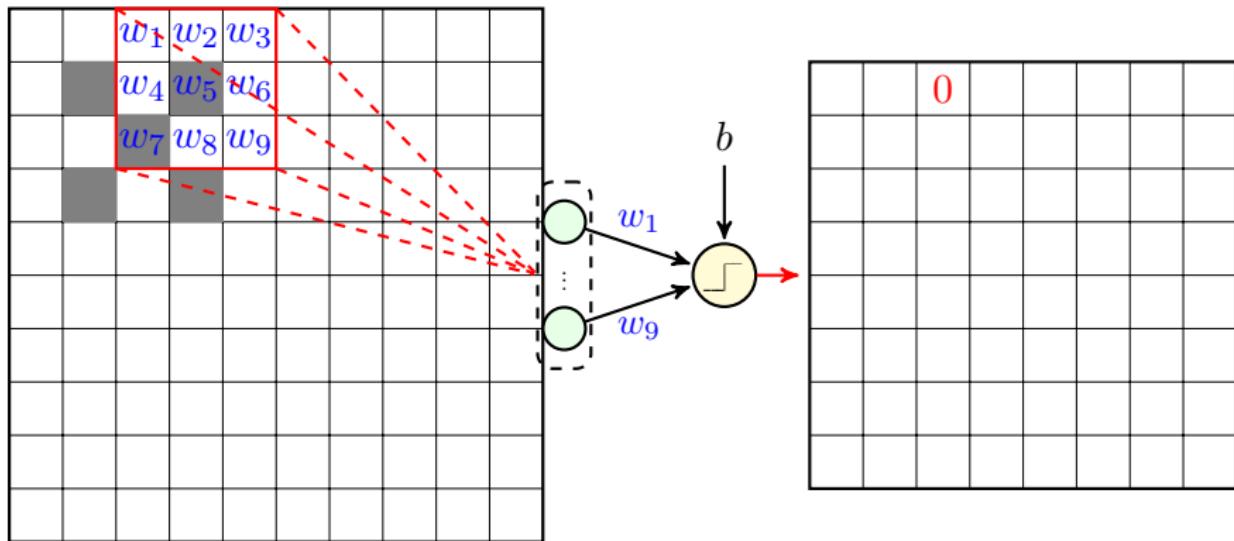
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

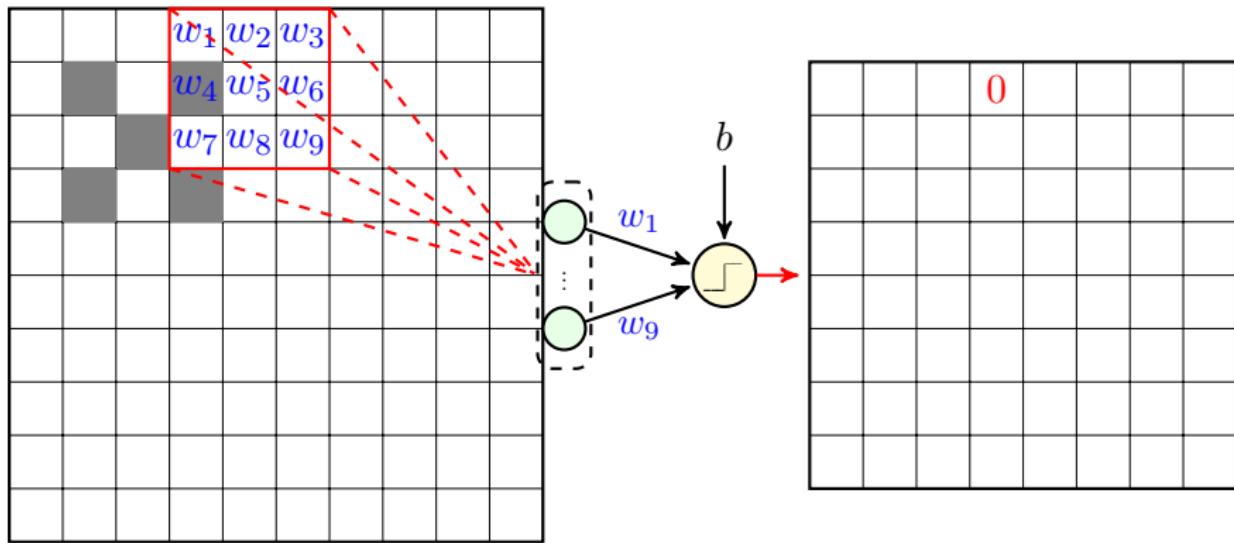
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

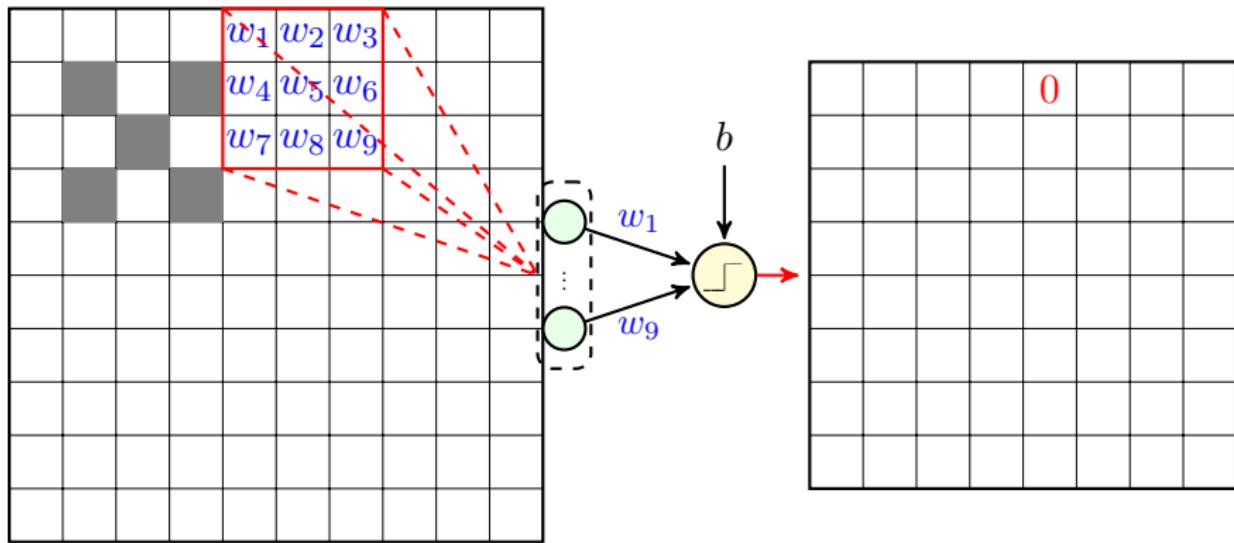
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

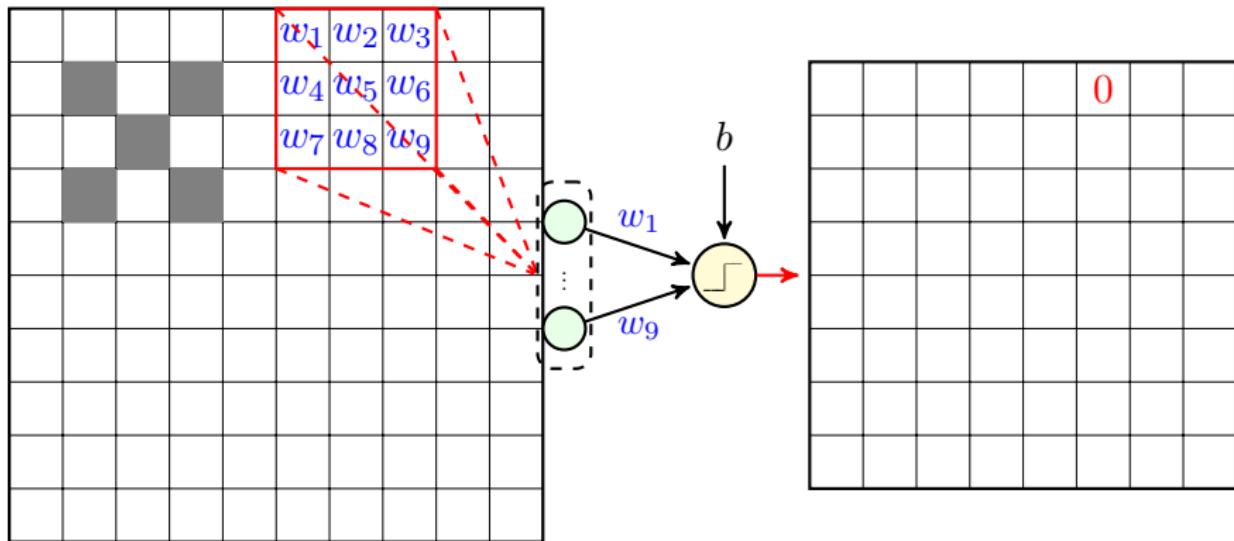
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

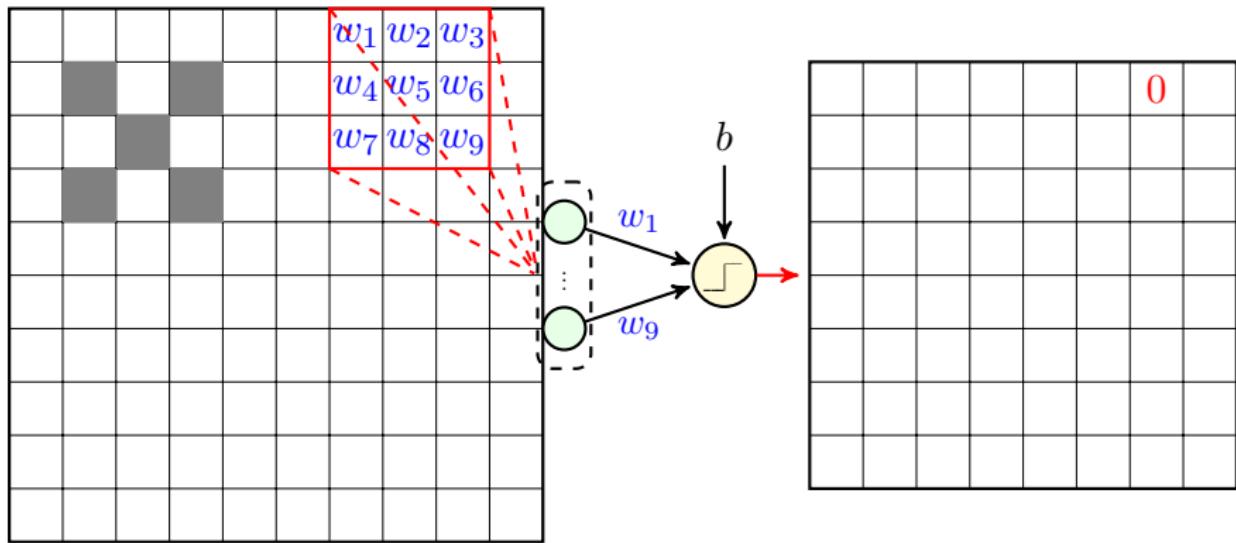
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

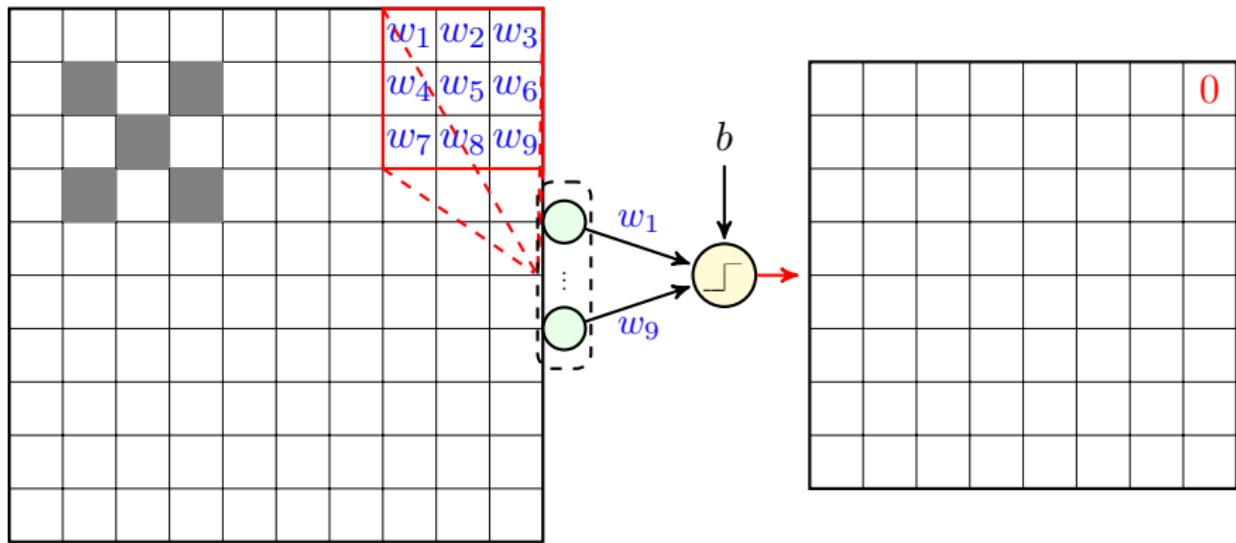
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

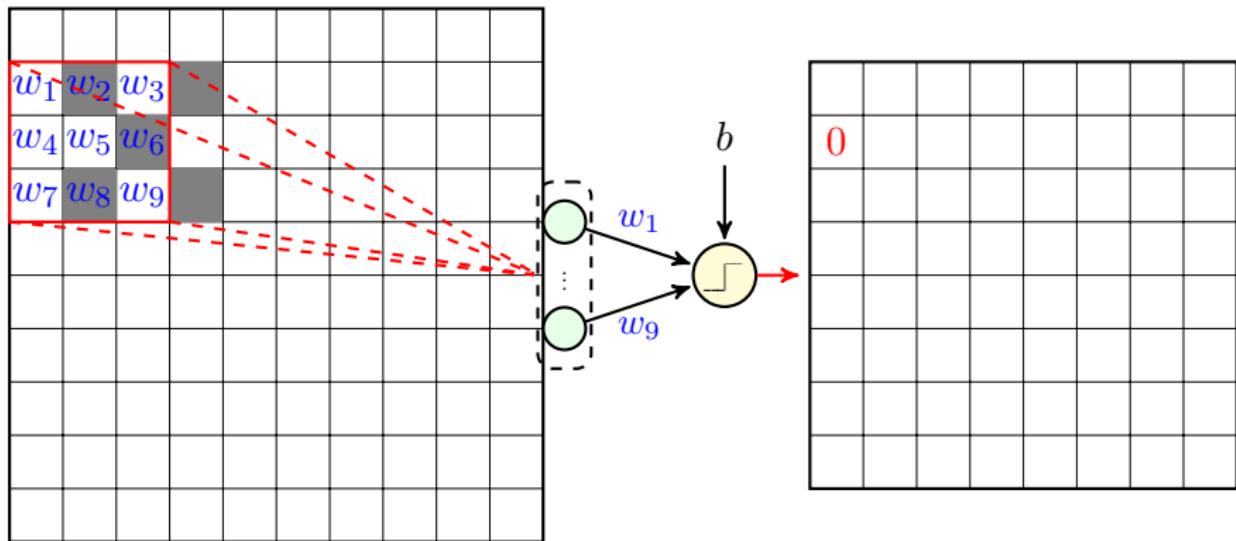
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

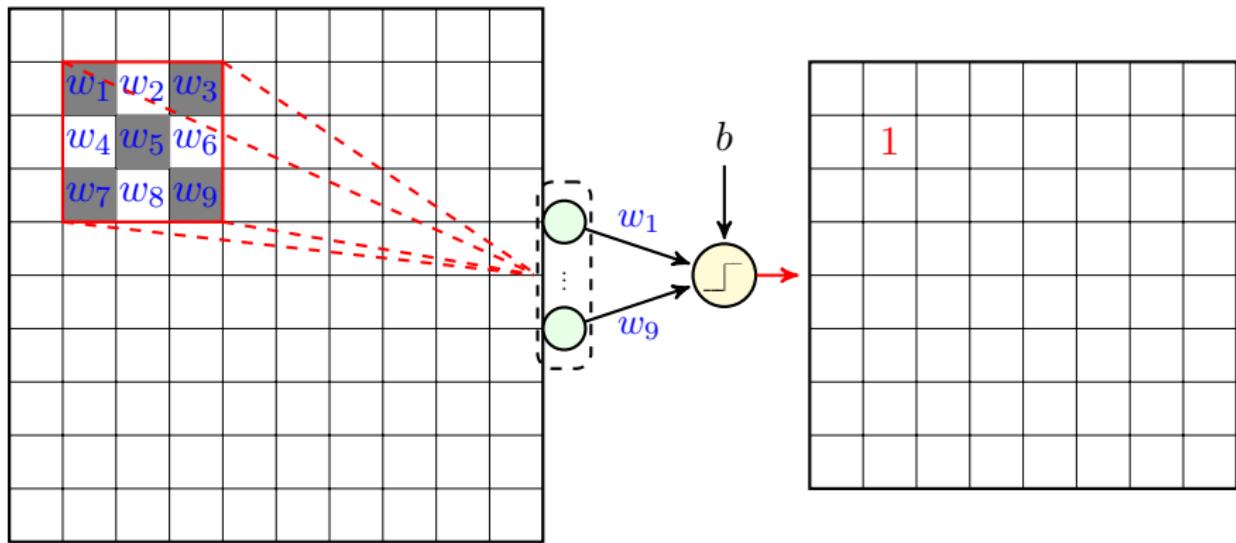
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

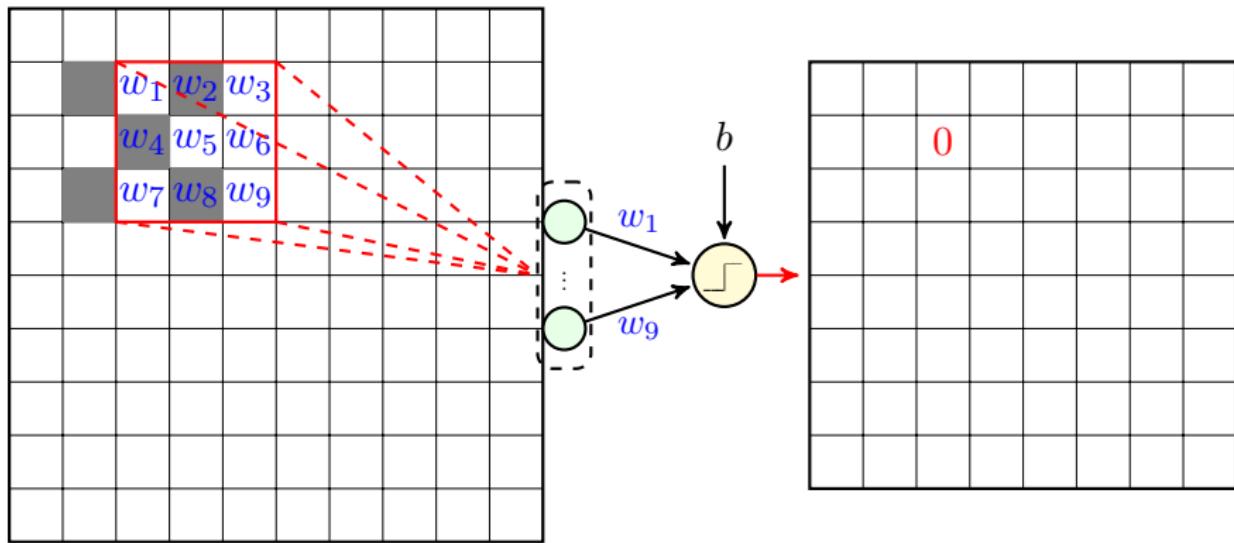
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

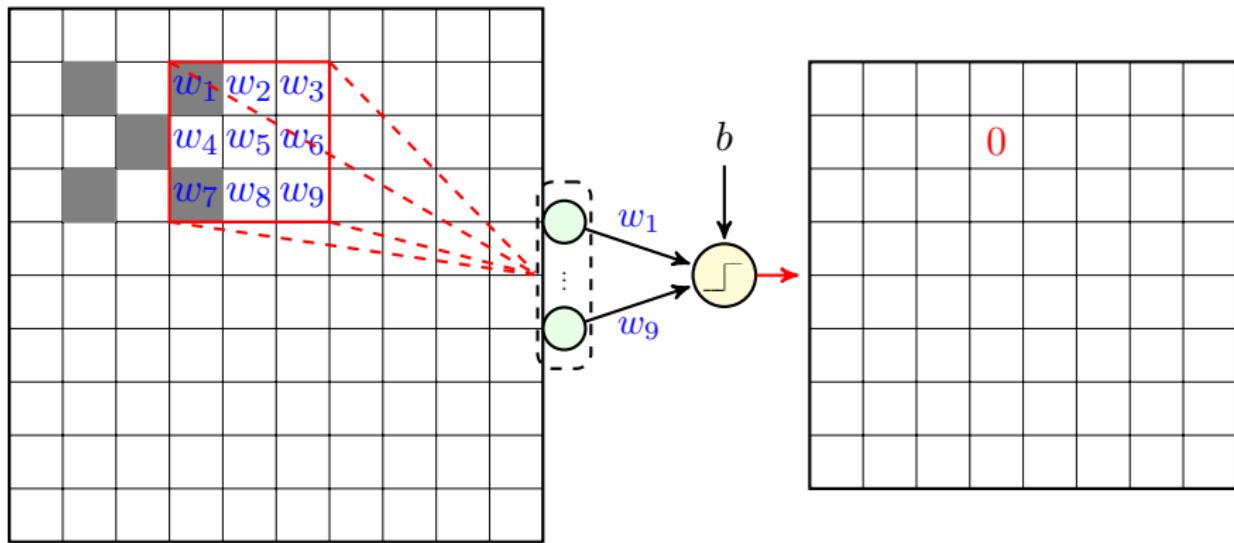
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

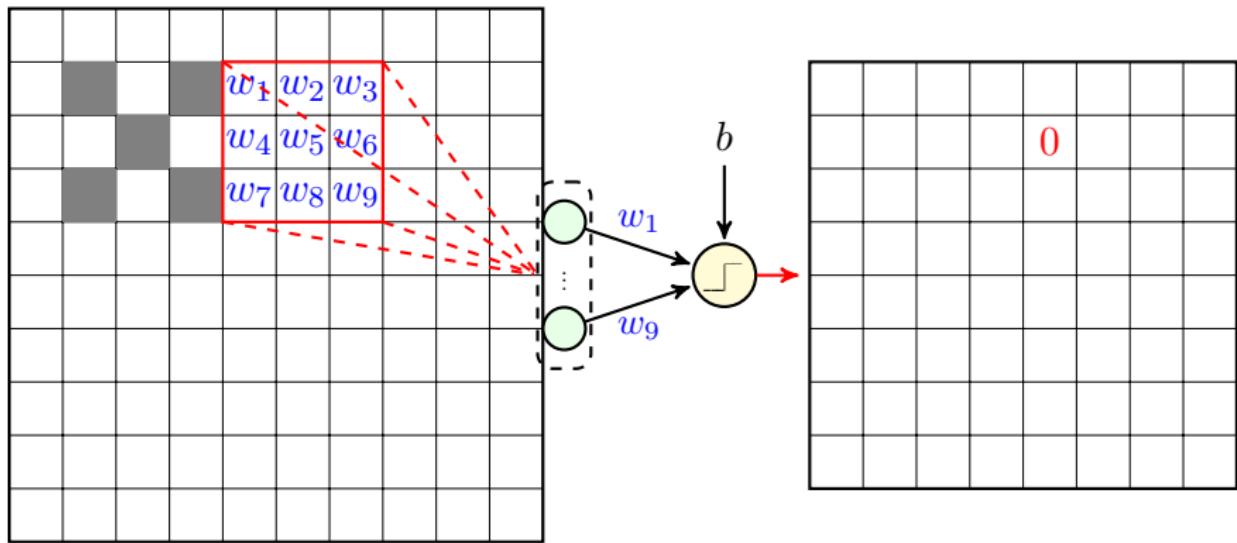
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

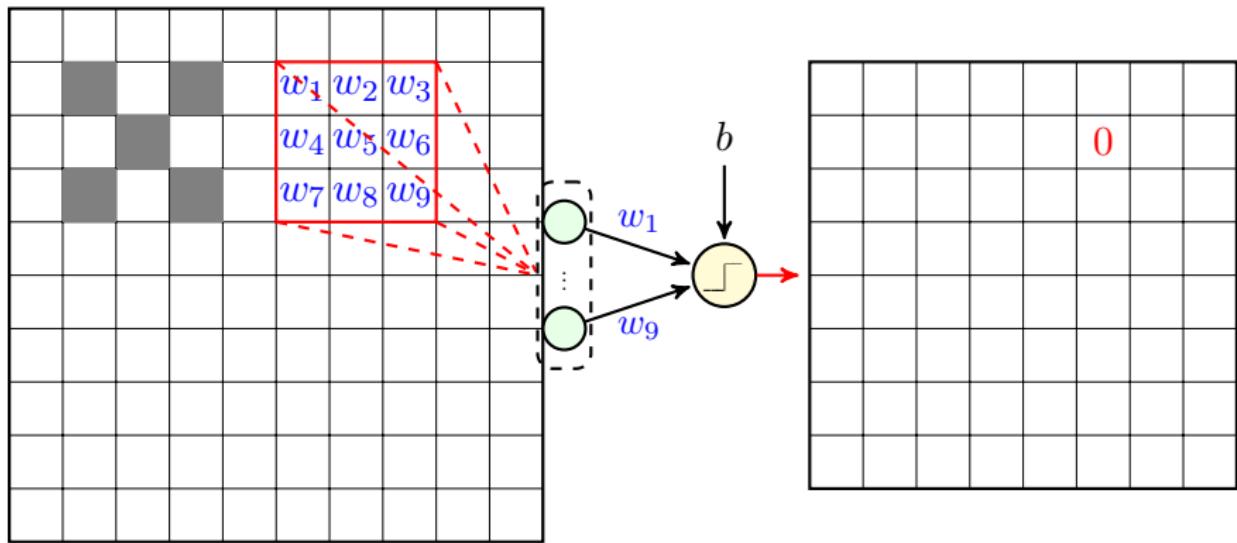
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

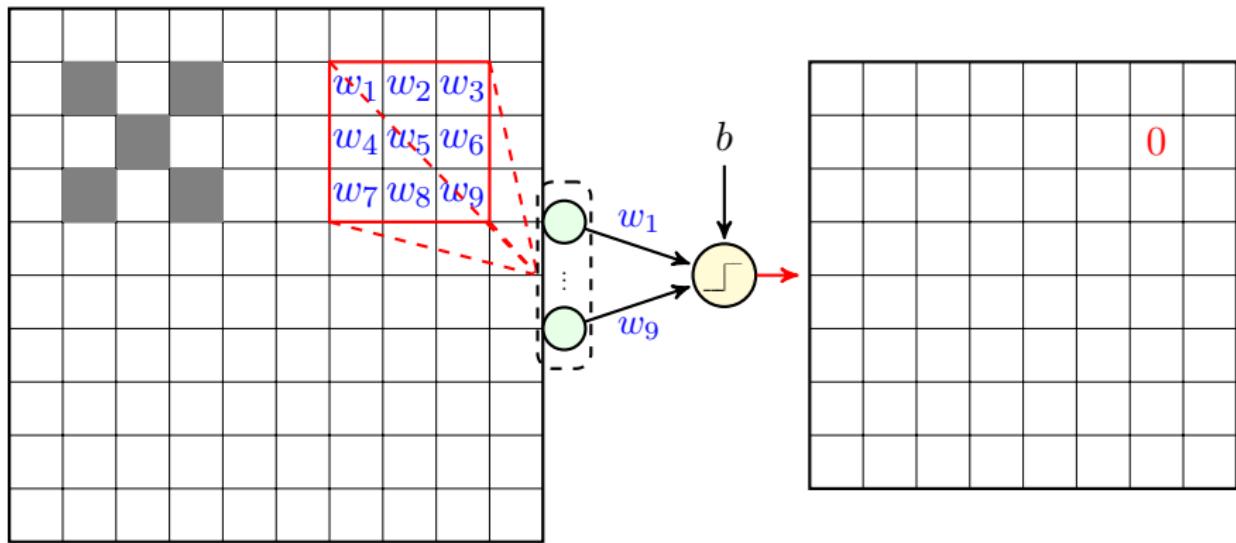
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

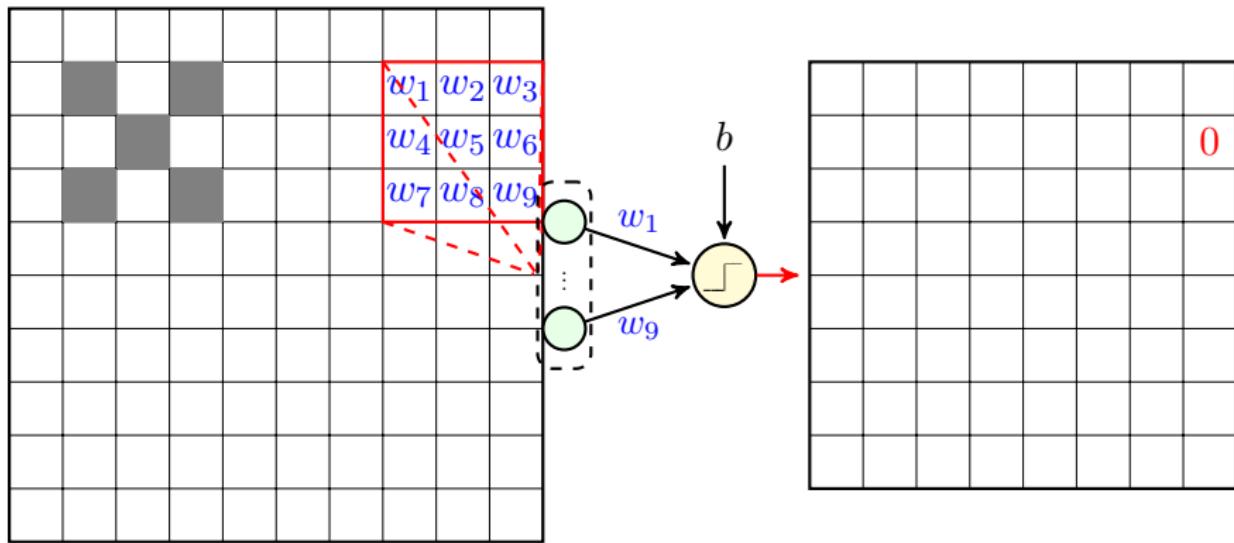
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

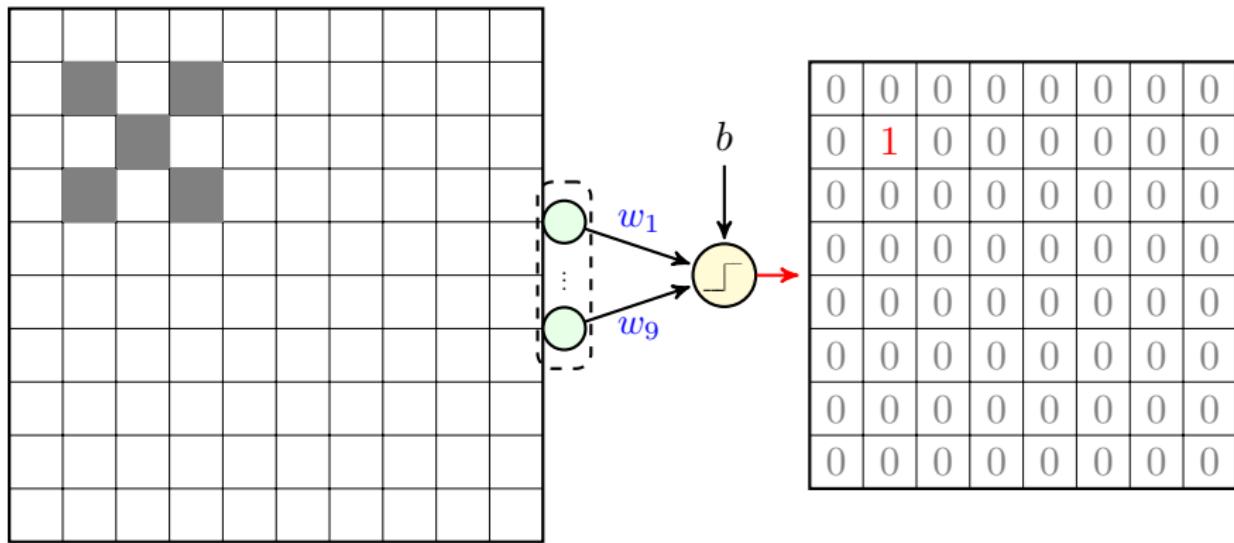
We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

We follow our scanning idea to recognize 3×3 "X" in a 10×10 image: we put the weights on a 3×3 filter and slide it over the image



We slide **the filter** with **stride 1** and save the outputs of perceptron on a **map**

Recognizing X

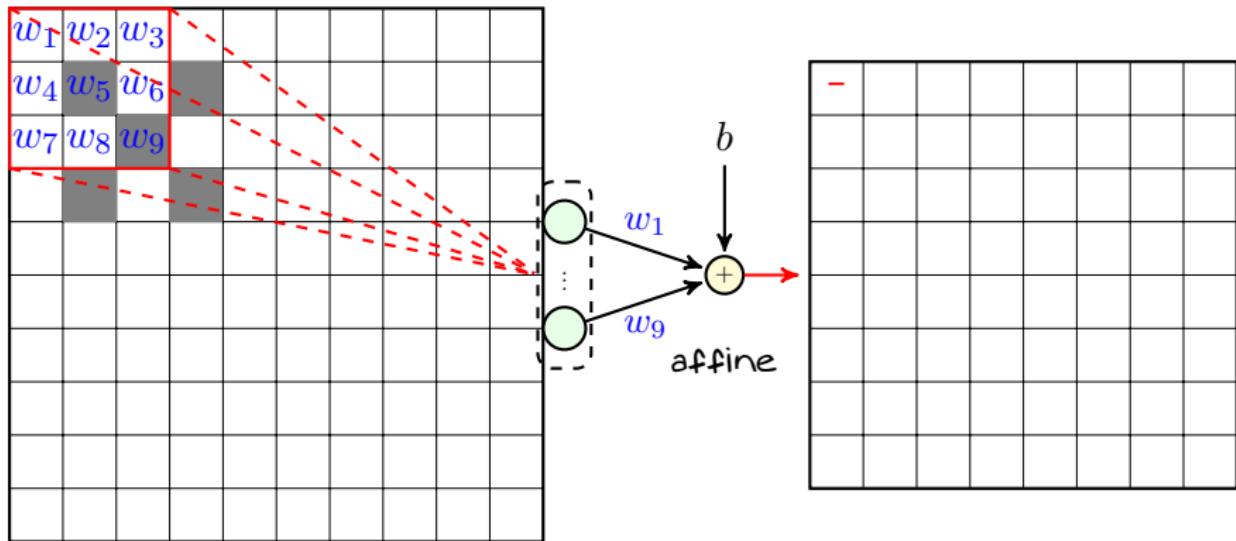
*It's enough to have **only a single 1** to recognize "X"*

0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

We therefore return the **OR** of all entries in the above **map**

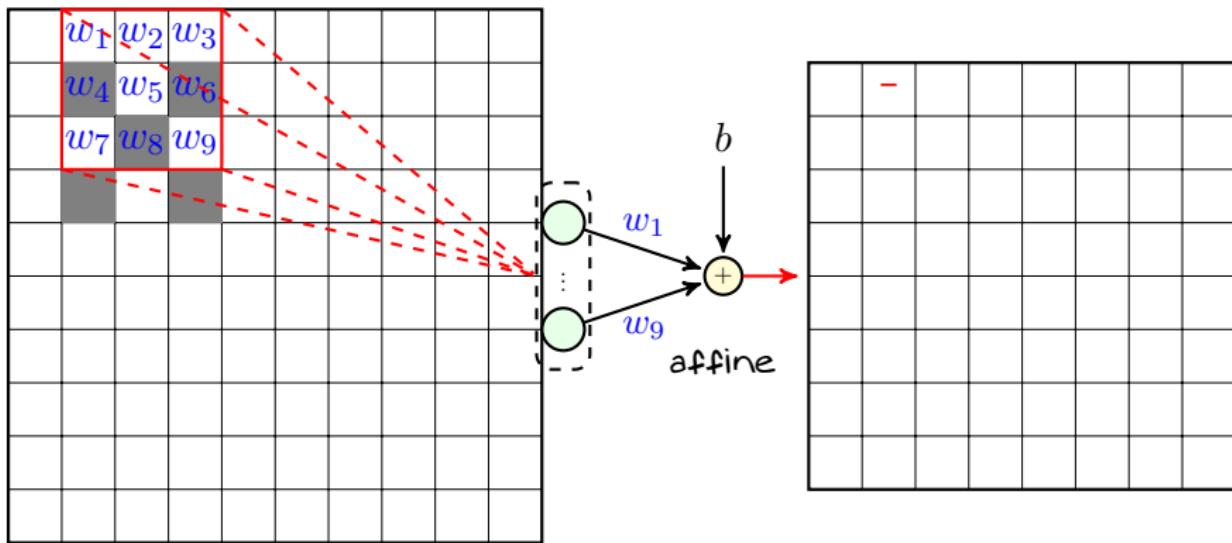
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms in the map**



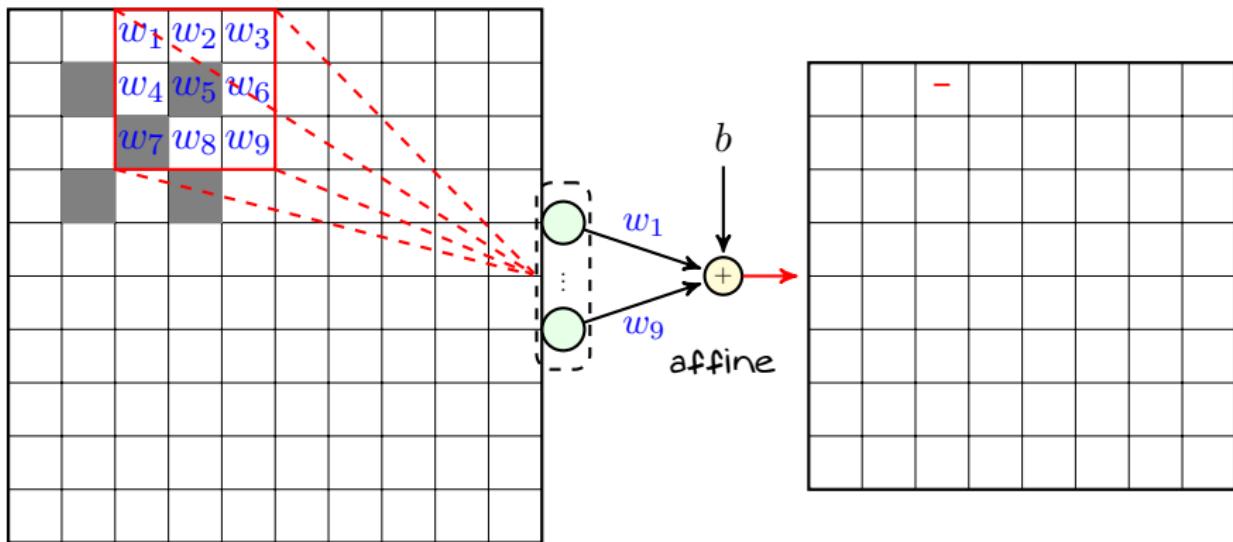
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



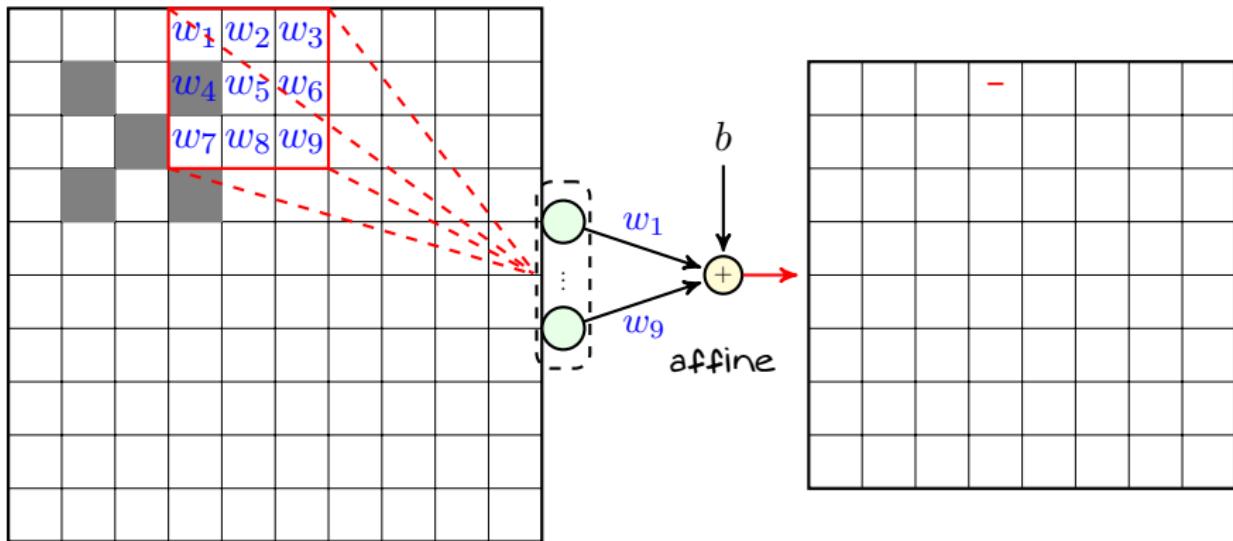
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



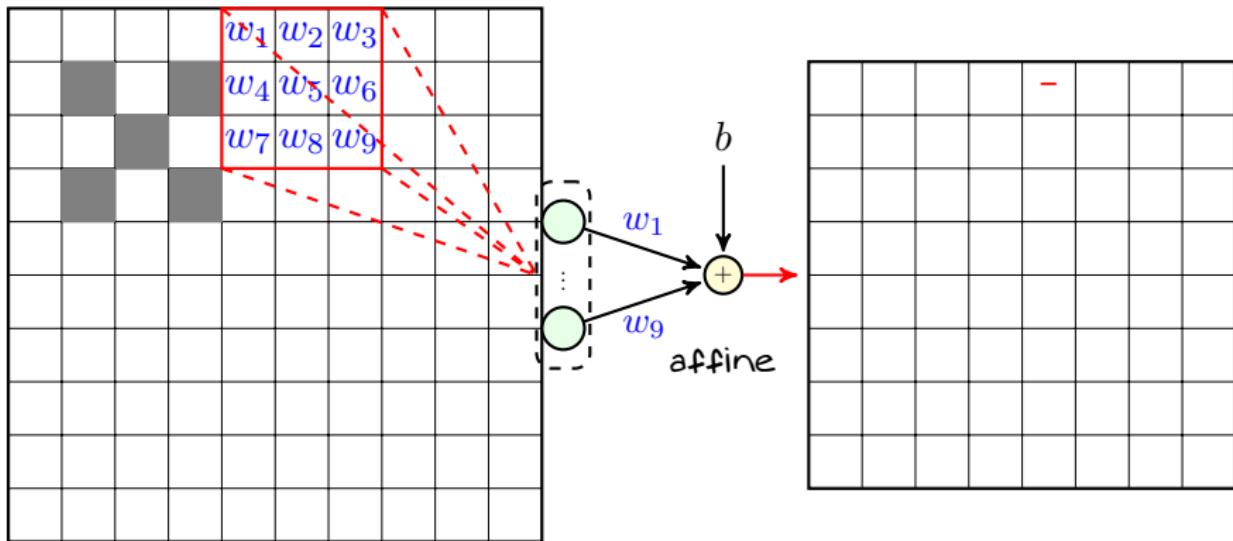
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



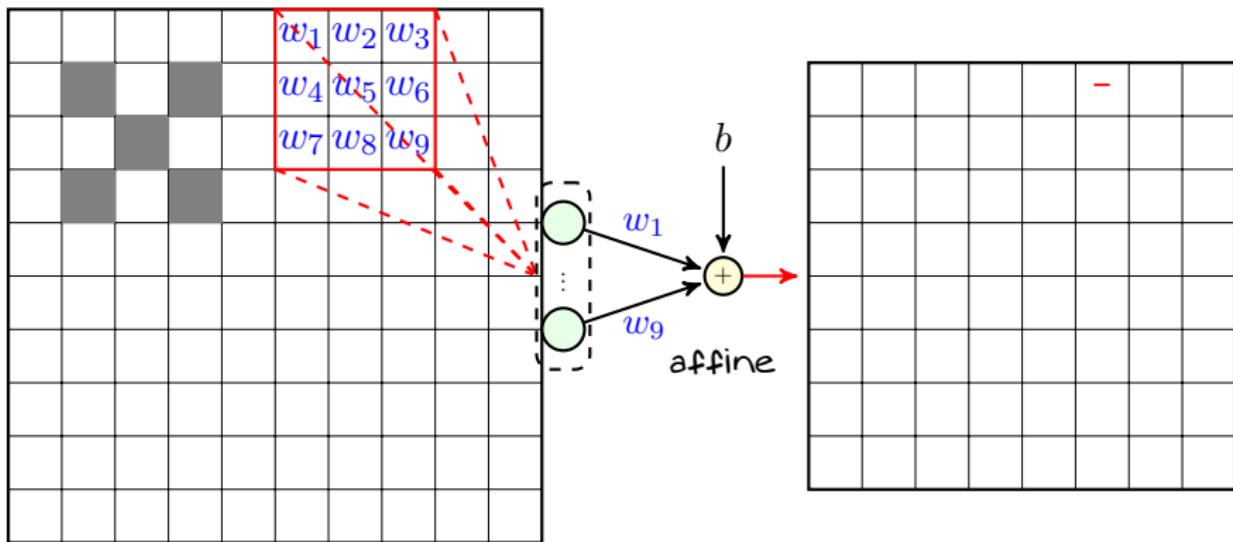
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



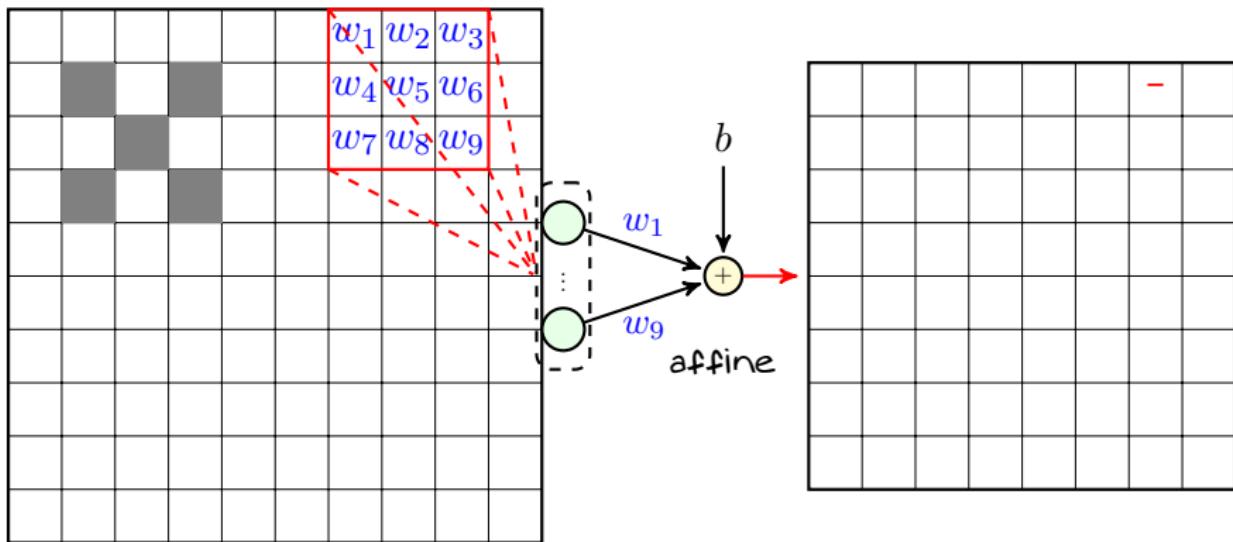
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



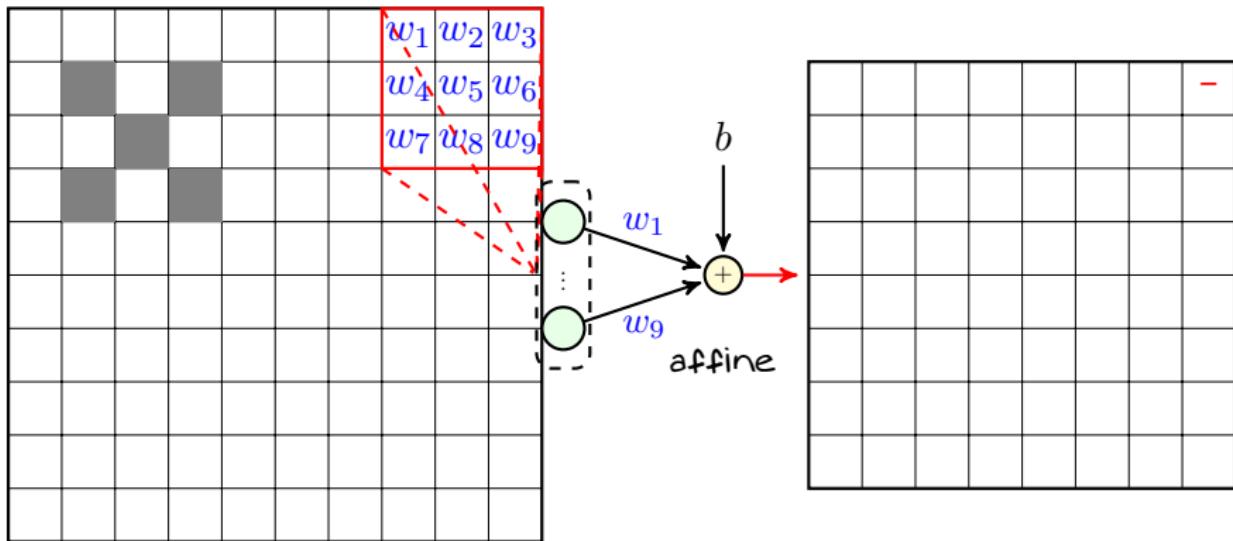
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



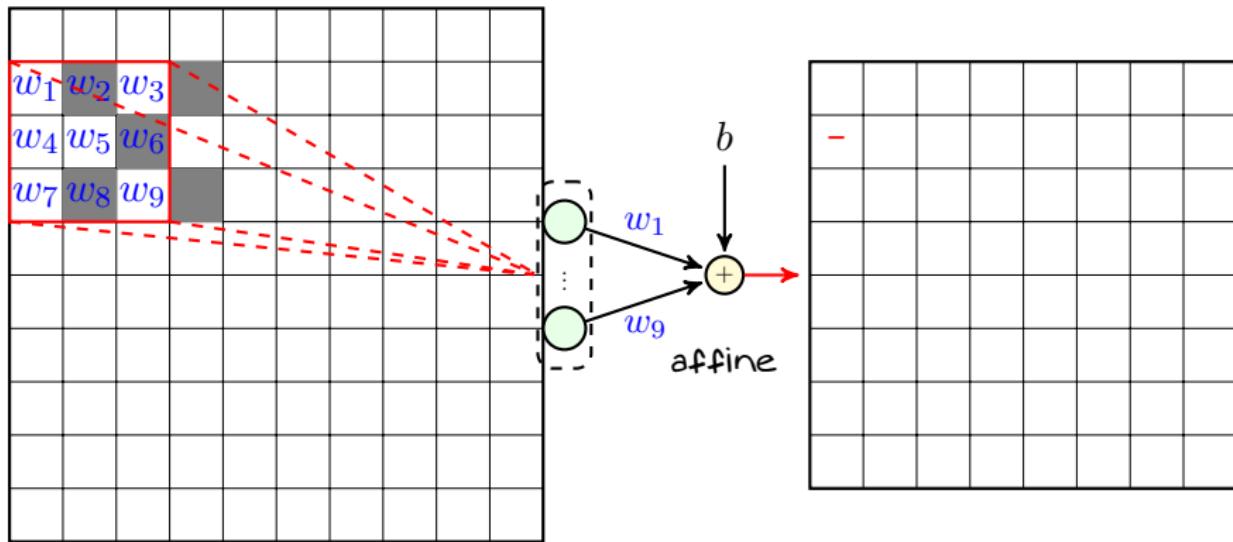
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



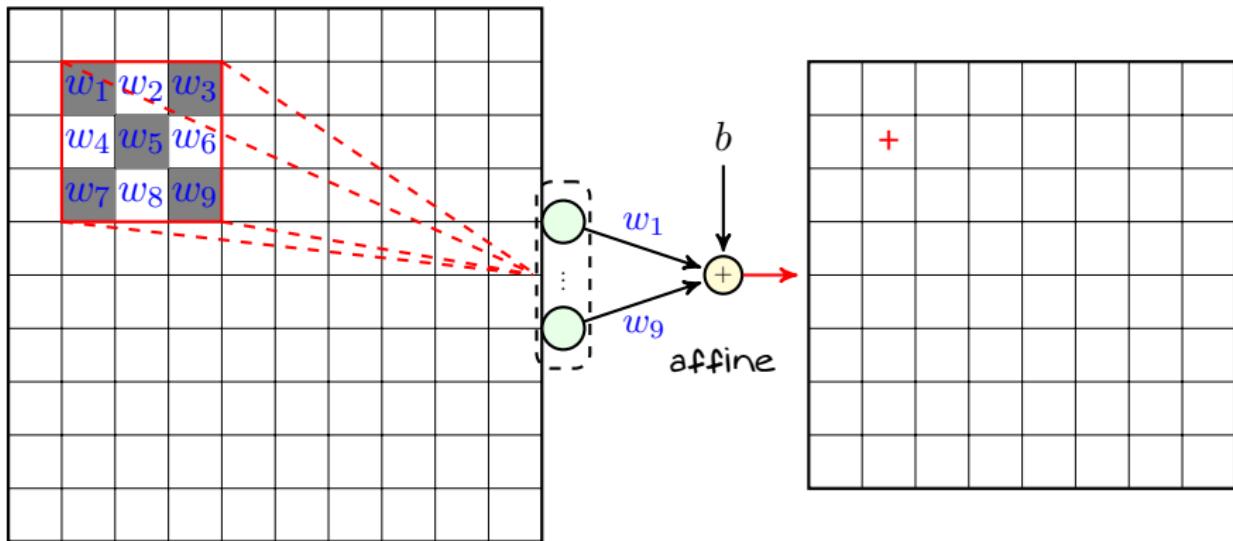
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



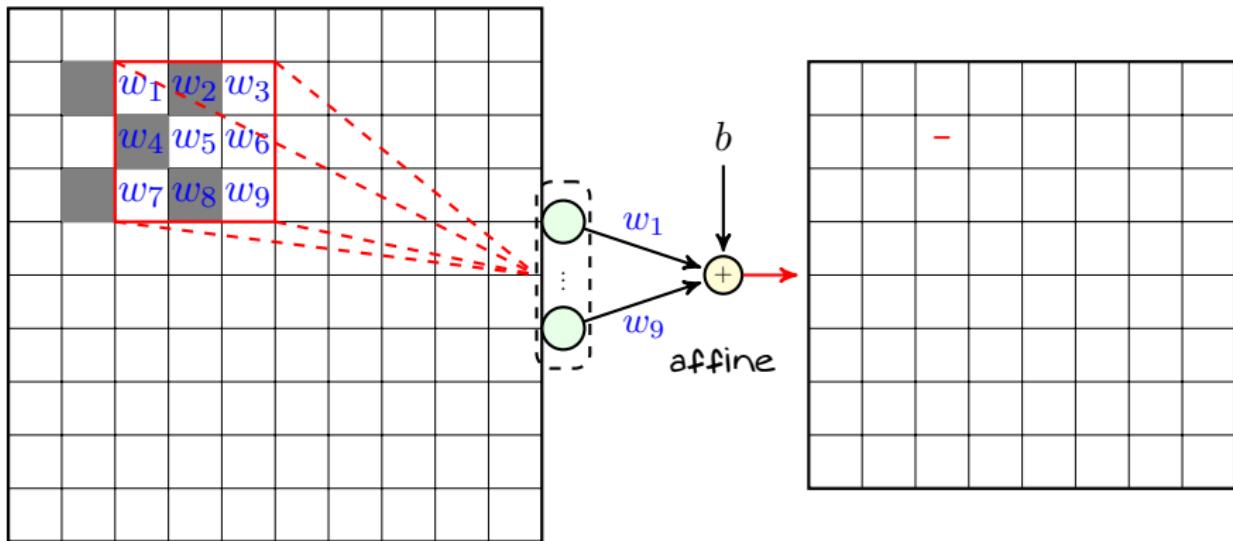
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



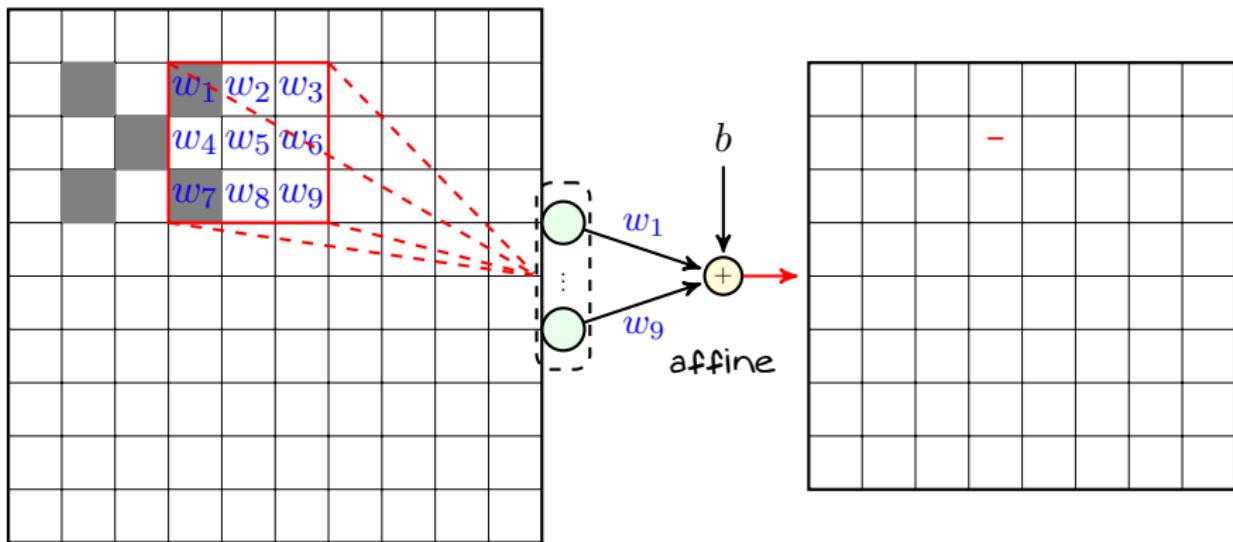
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



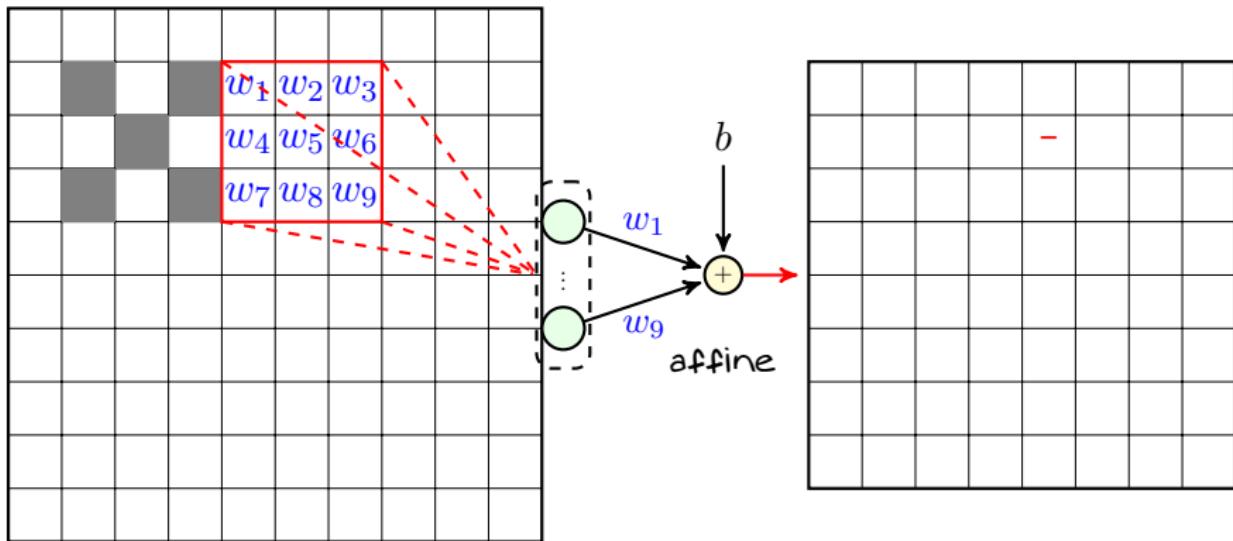
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



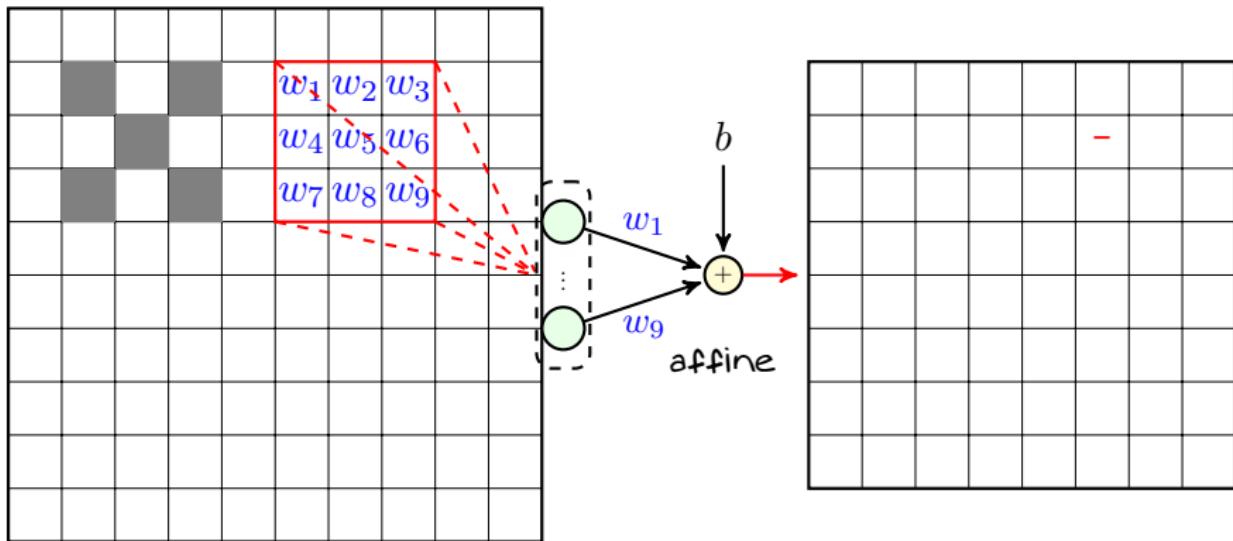
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



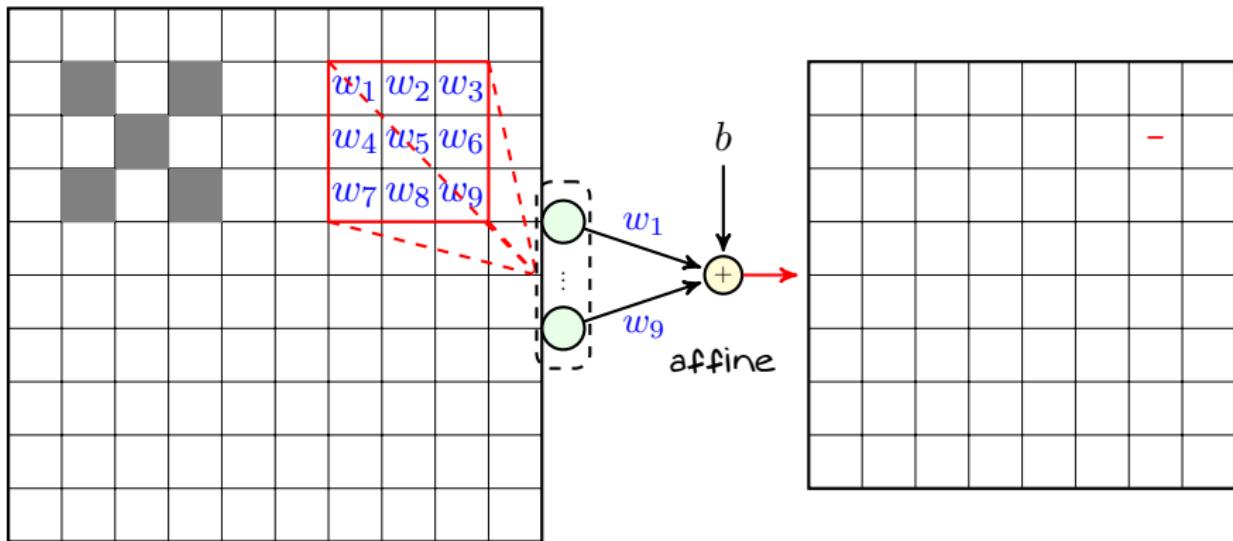
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the map



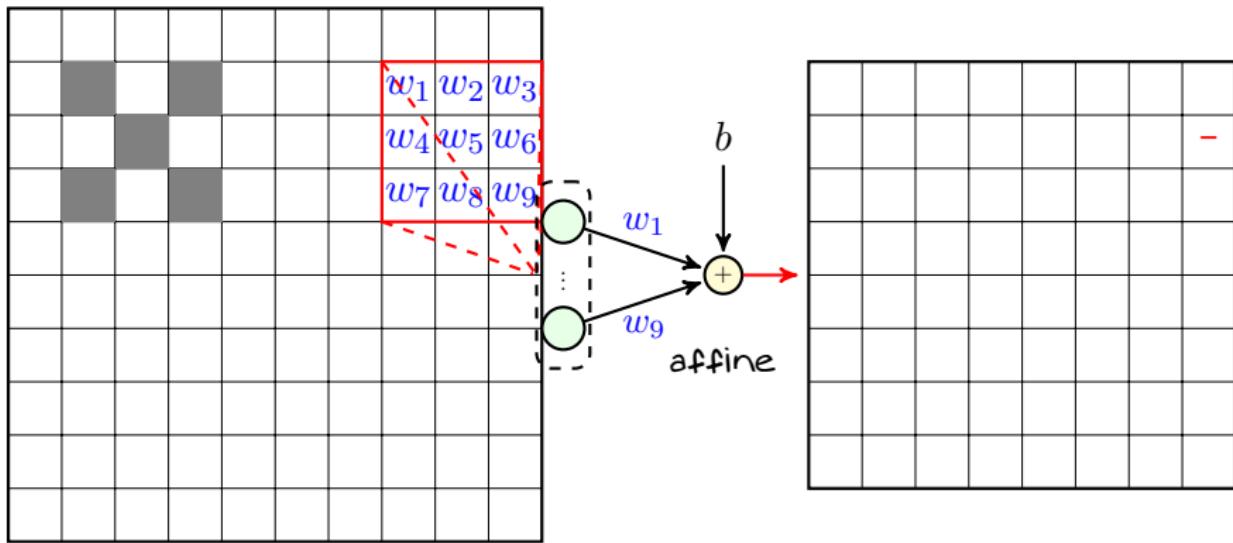
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



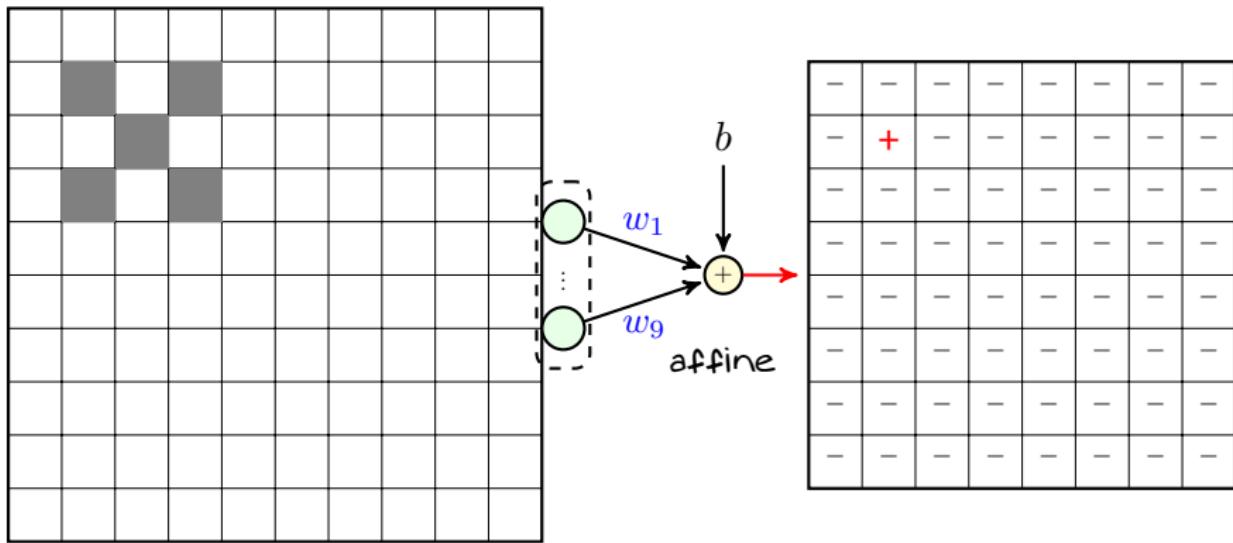
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



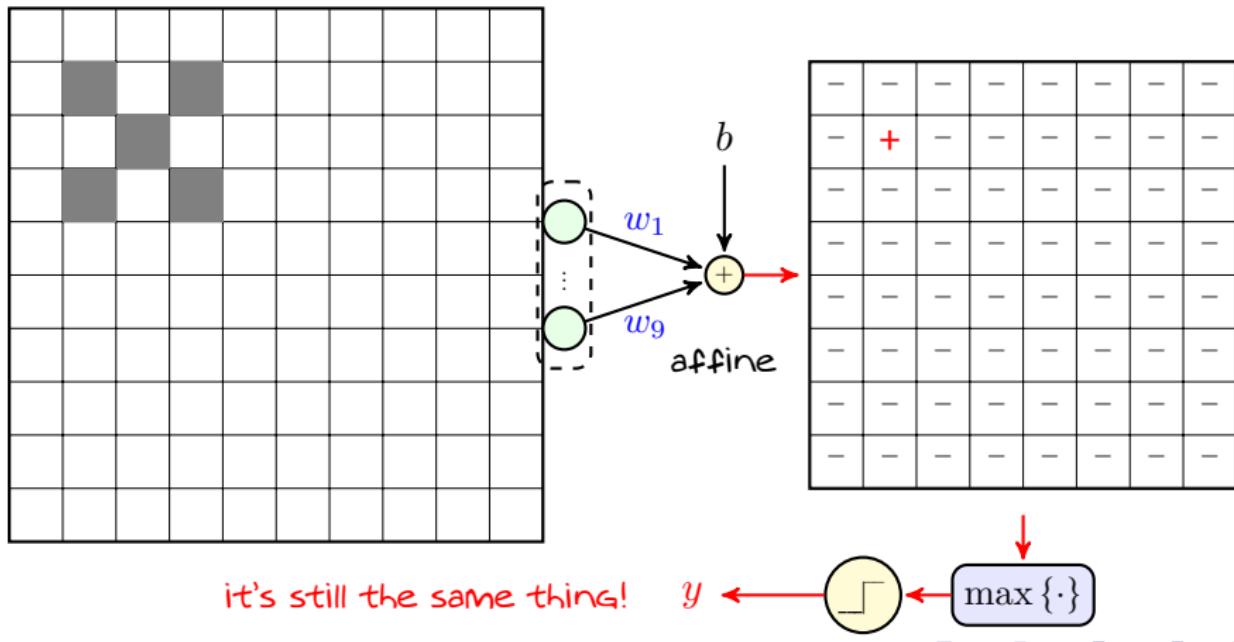
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms in the map**



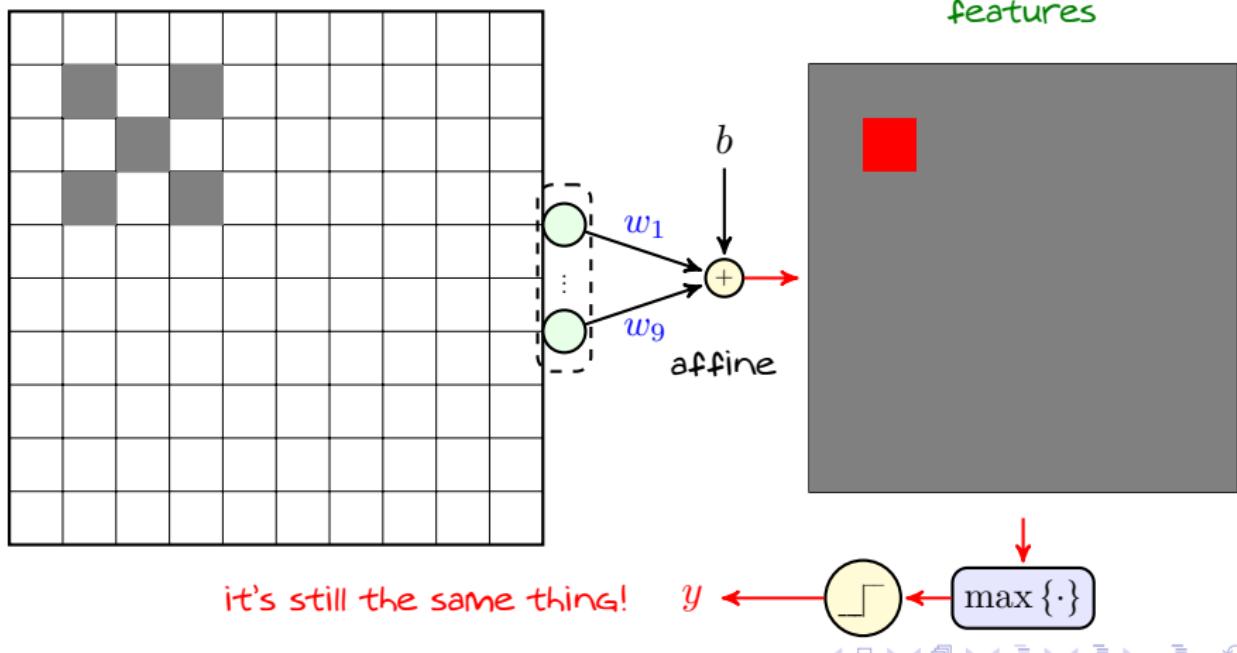
Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms** in the **map**



Recognizing X: Convolution

We really don't need to determine the **activation** after each scan: we could only save the **affine transforms in the map**



Convolution with Stride 1

image

w_1	w_2	w_3						
w_4	w_5	w_6						
w_7	w_8	w_9						

feature map

filter

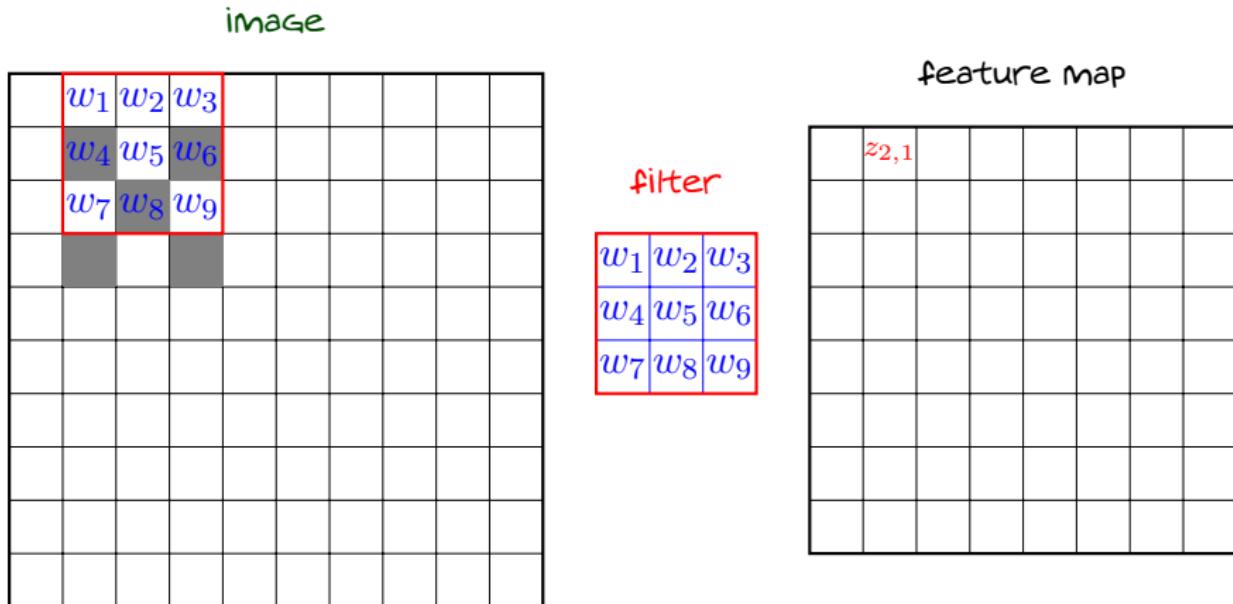
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

$z_{1,1}$								

The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

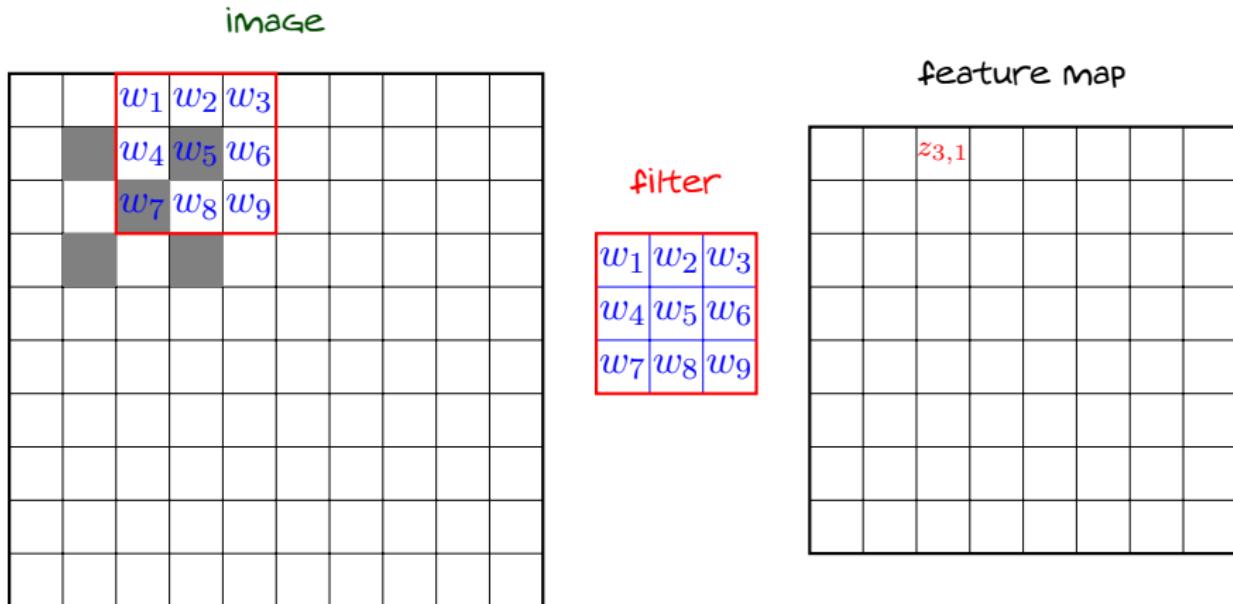
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

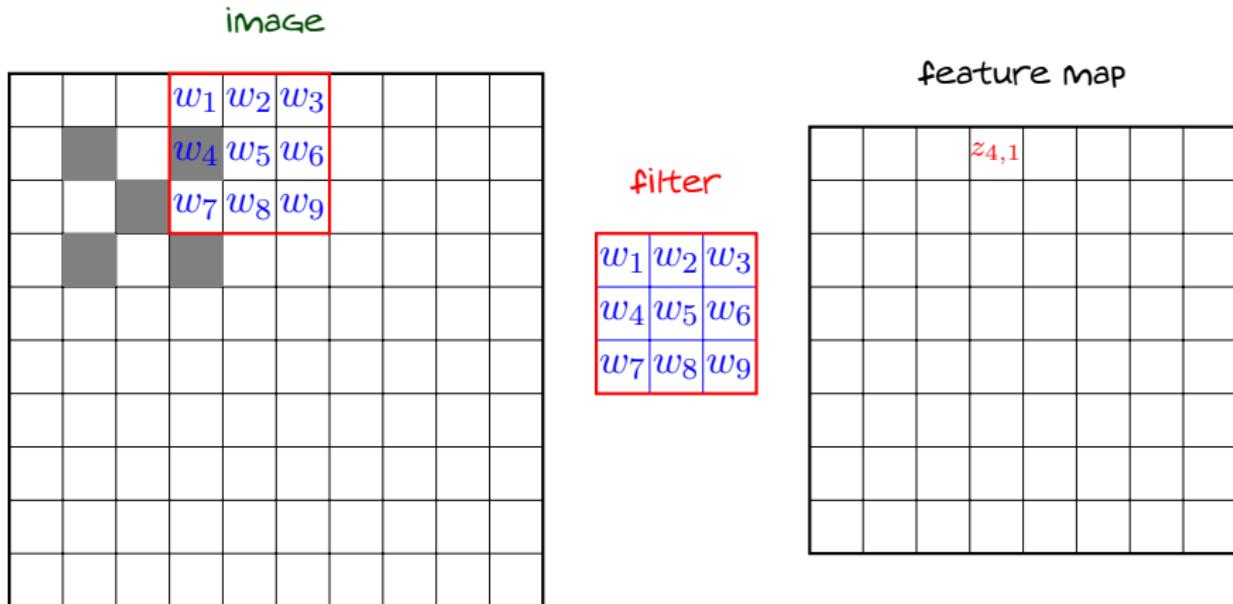
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

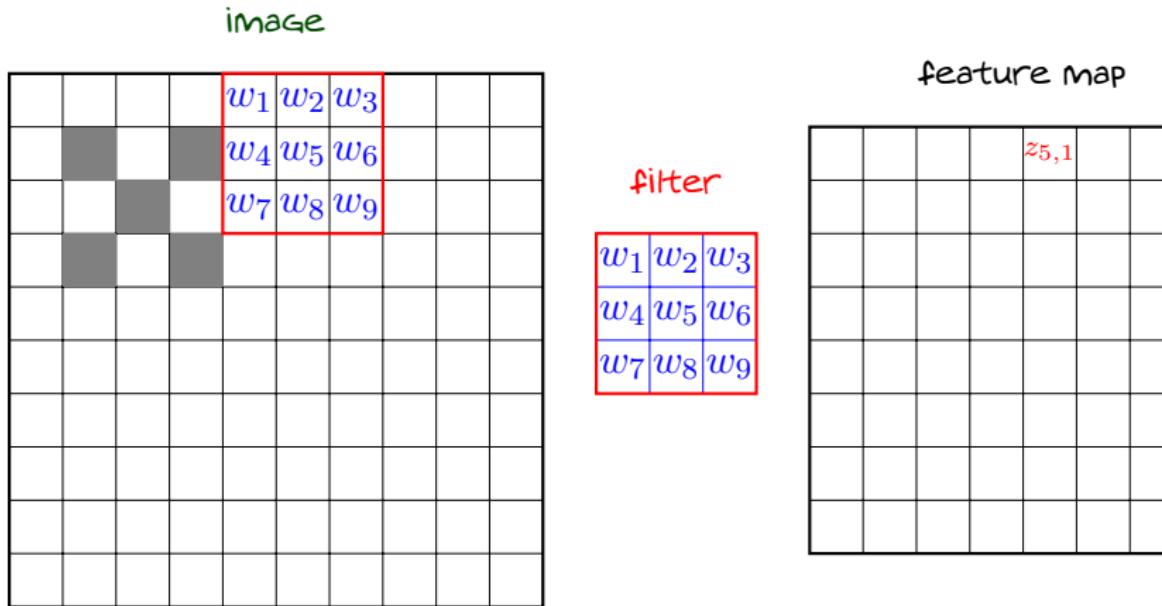
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

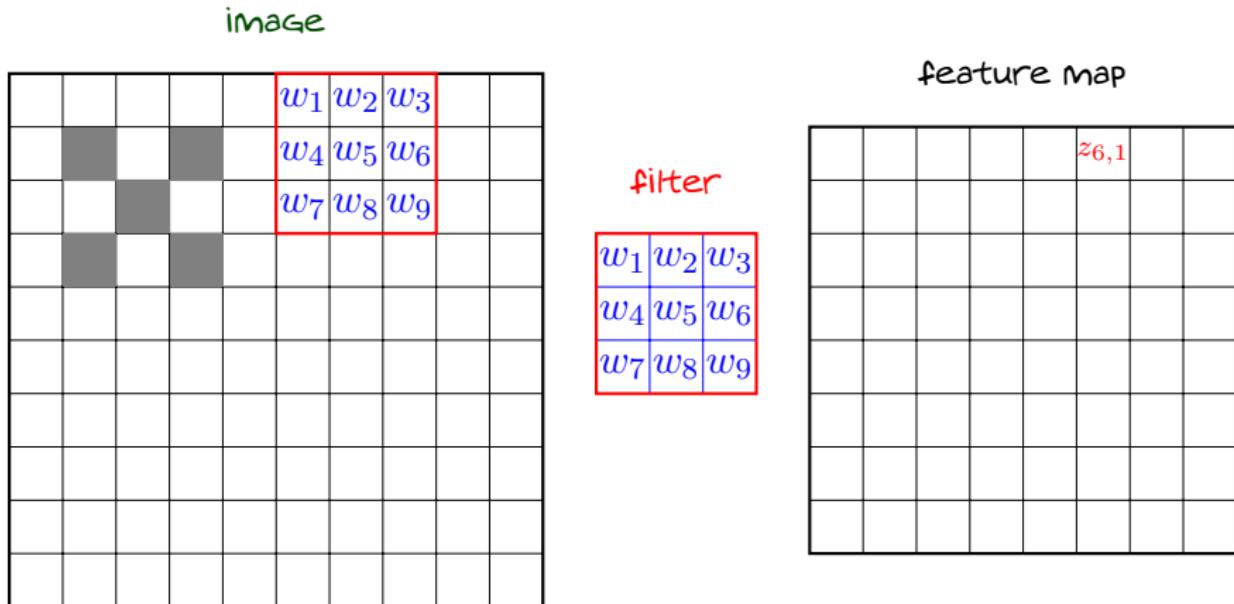
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

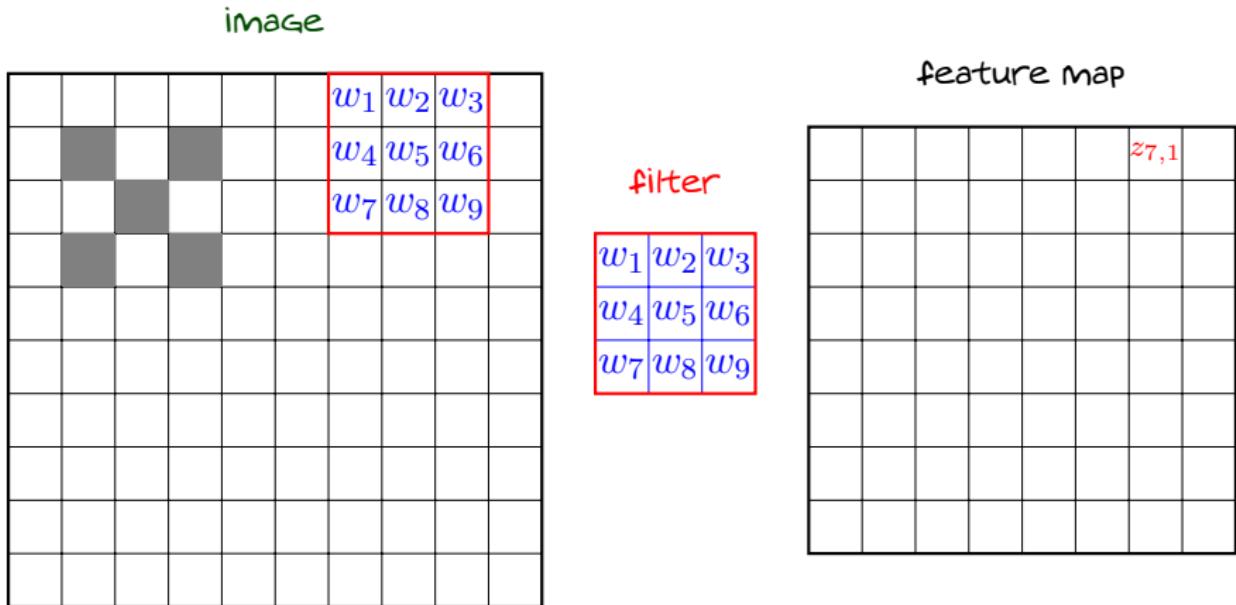
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

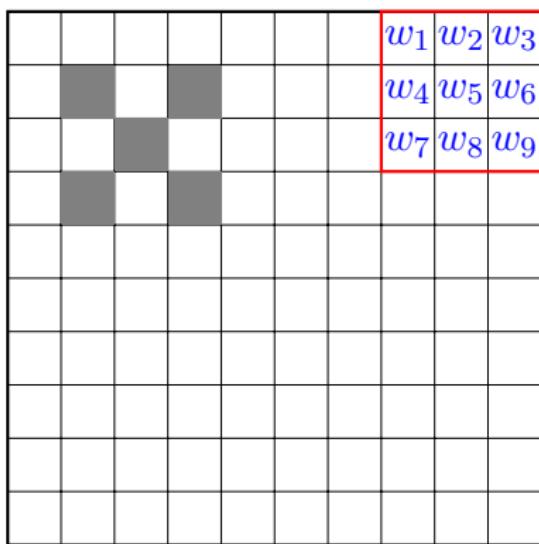


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

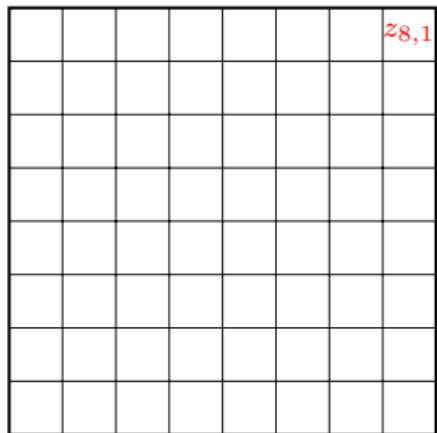
image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

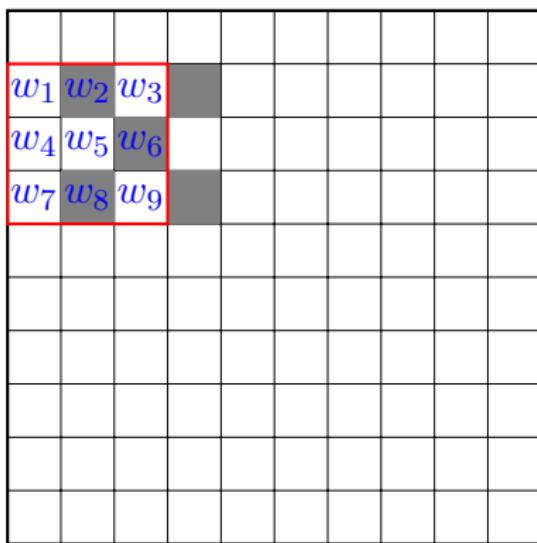


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

image

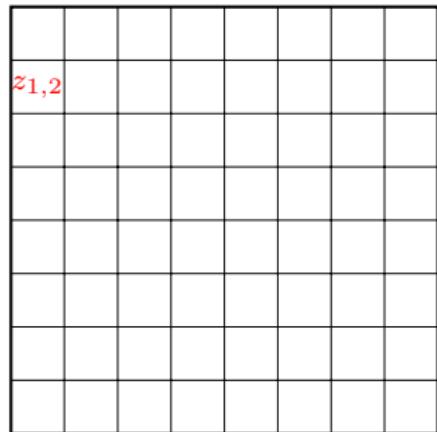


feature map

filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

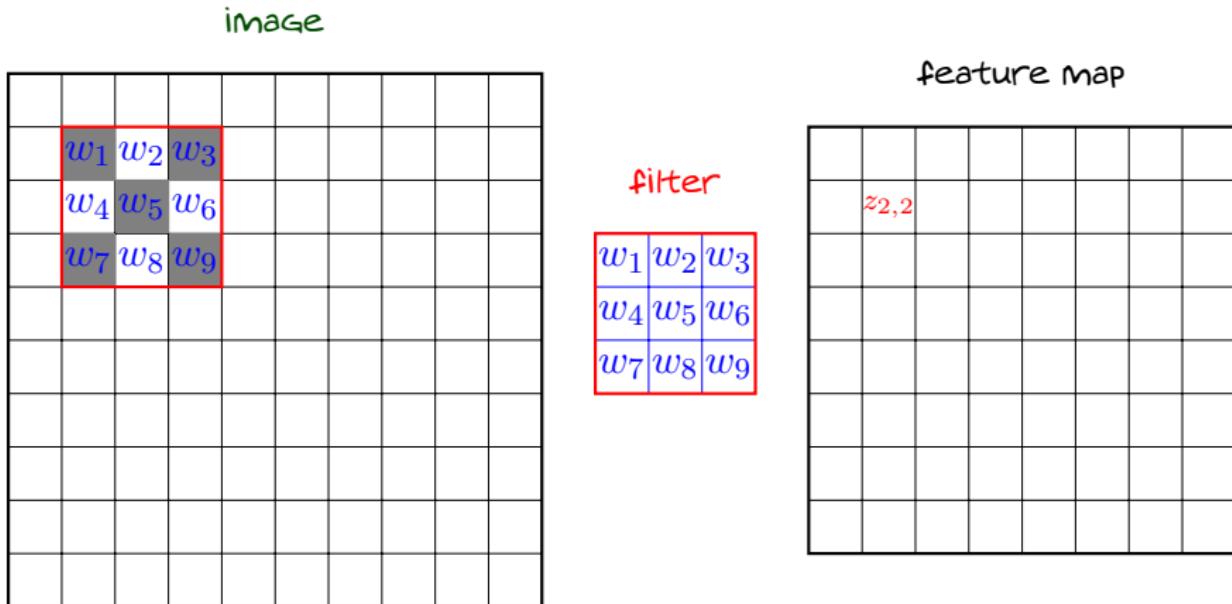
$z_{1,2}$



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

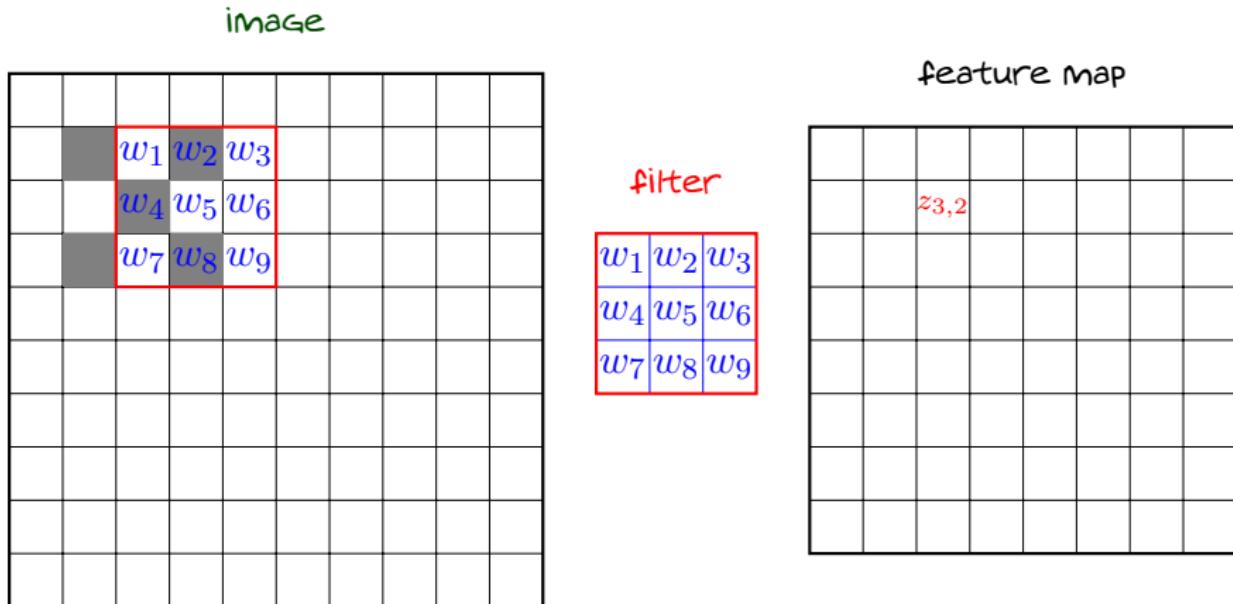
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

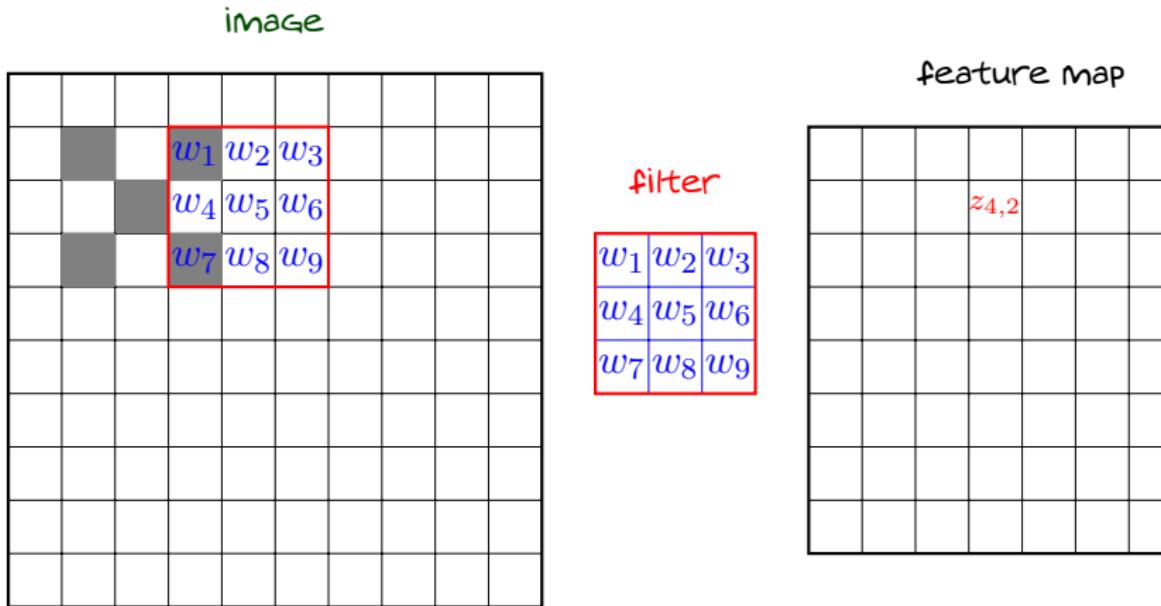
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

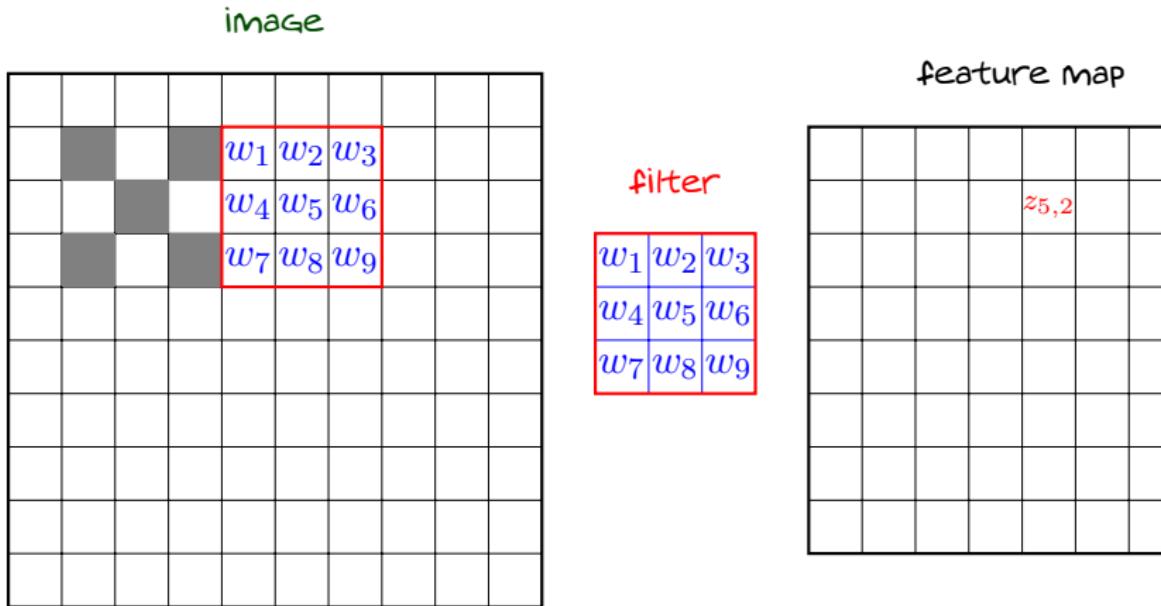
Convolution with Stride 1



The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

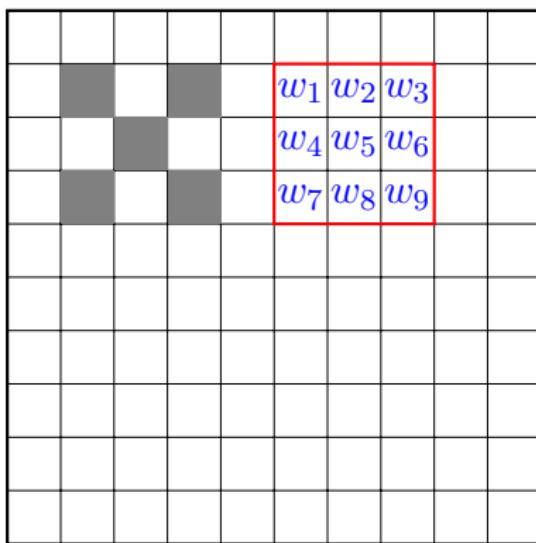


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

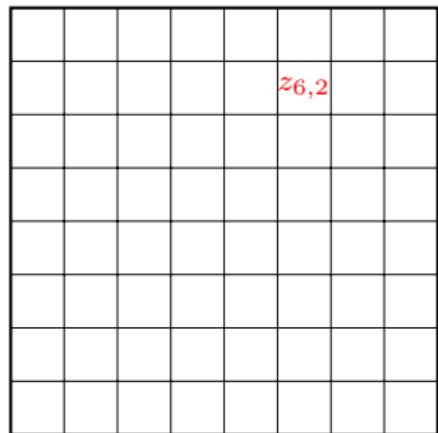
image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

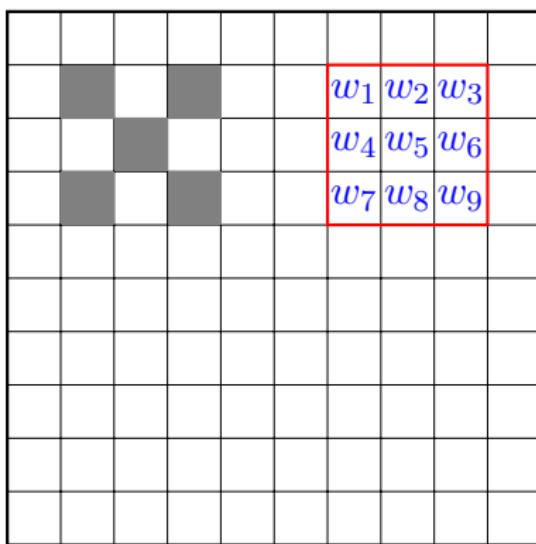


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

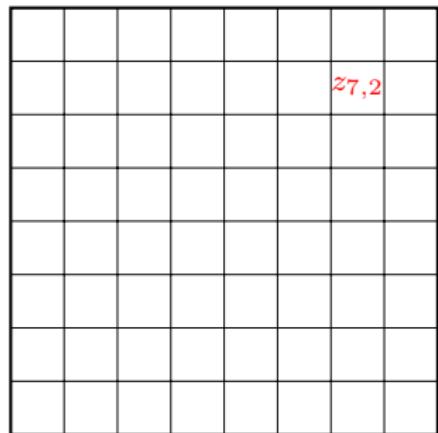
image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

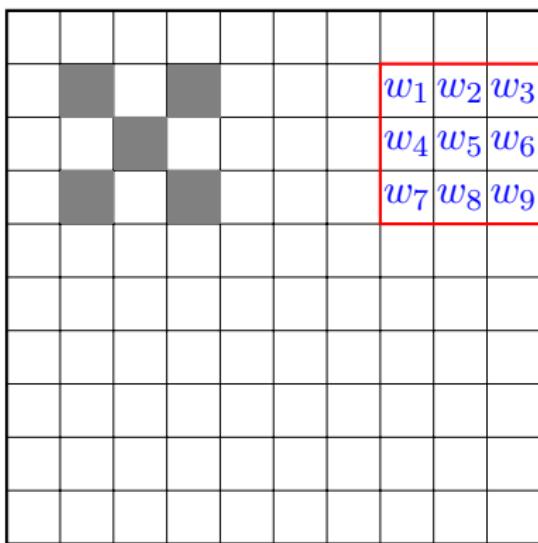


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

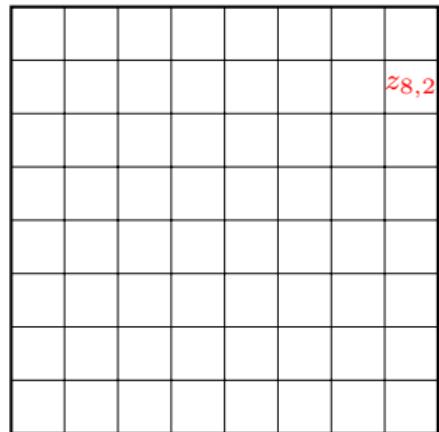
image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

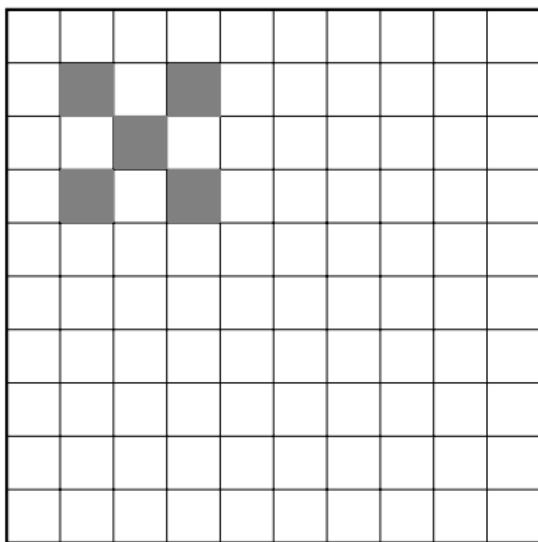


The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 1

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

$z_{1,1}$	$z_{2,1}$	$z_{3,1}$	$z_{4,1}$	$z_{5,1}$	$z_{6,1}$	$z_{7,1}$	$z_{8,1}$
$z_{1,2}$	$z_{2,2}$	$z_{3,2}$	$z_{4,2}$	$z_{5,2}$	$z_{6,2}$	$z_{7,2}$	$z_{8,2}$
$z_{1,3}$	$z_{2,3}$	$z_{3,3}$	$z_{4,3}$	$z_{5,3}$	$z_{6,3}$	$z_{7,3}$	$z_{8,3}$
$z_{1,4}$	$z_{2,4}$	$z_{3,4}$	$z_{4,4}$	$z_{5,4}$	$z_{6,4}$	$z_{7,4}$	$z_{8,4}$
$z_{1,5}$	$z_{2,5}$	$z_{3,5}$	$z_{4,5}$	$z_{5,5}$	$z_{6,5}$	$z_{7,5}$	$z_{8,5}$
$z_{1,6}$	$z_{2,6}$	$z_{3,6}$	$z_{4,6}$	$z_{5,6}$	$z_{6,6}$	$z_{7,6}$	$z_{8,6}$
$z_{1,7}$	$z_{2,7}$	$z_{3,7}$	$z_{4,7}$	$z_{5,7}$	$z_{6,7}$	$z_{7,7}$	$z_{8,7}$
$z_{1,8}$	$z_{2,8}$	$z_{3,8}$	$z_{4,8}$	$z_{5,8}$	$z_{6,8}$	$z_{7,8}$	$z_{8,8}$

The above operation is **convolution** and we show it as below

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 1)$$

Convolution with Stride 2

image

w_1	w_2	w_3						
w_4	w_5	w_6						
w_7	w_8	w_9						

filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

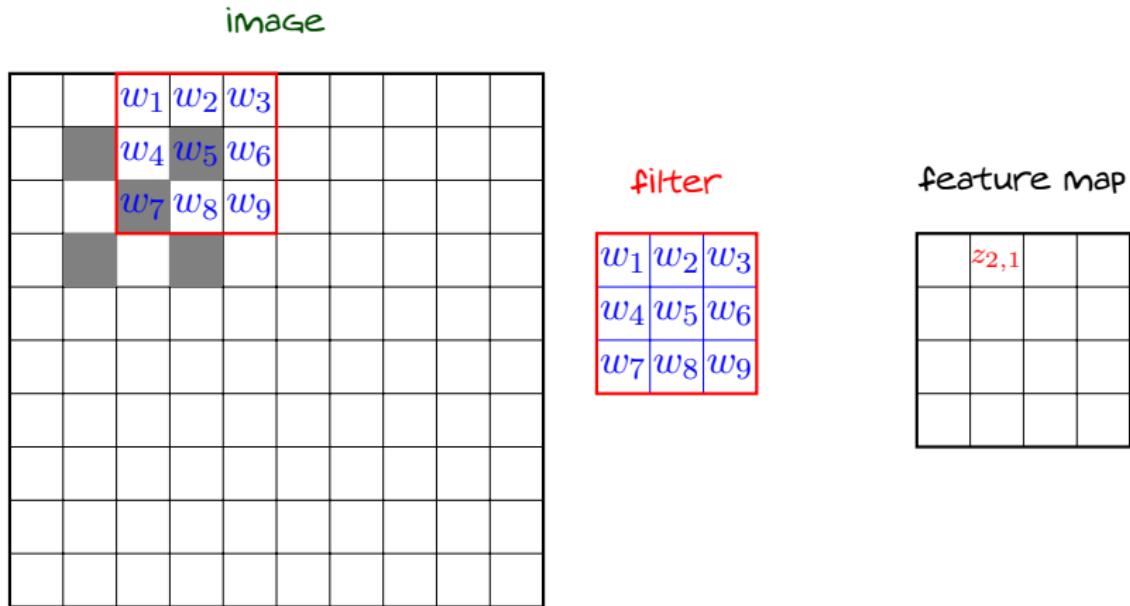
feature map

$z_{1,1}$			

We could also play with the *stride* \equiv the step-size by which we move filter

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

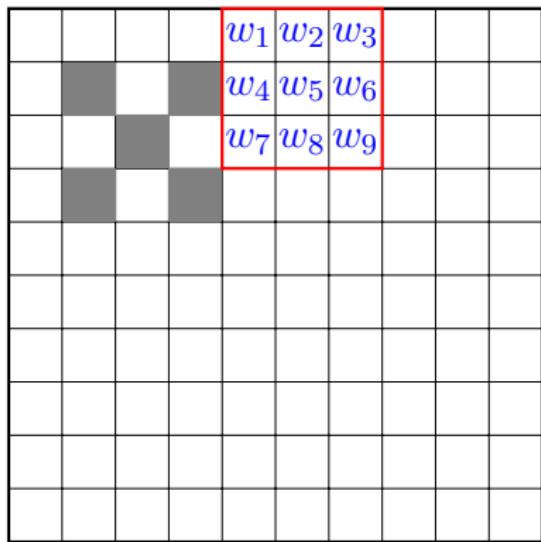


We could also play with the *stride* \equiv the step-size by which we move *filter*

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

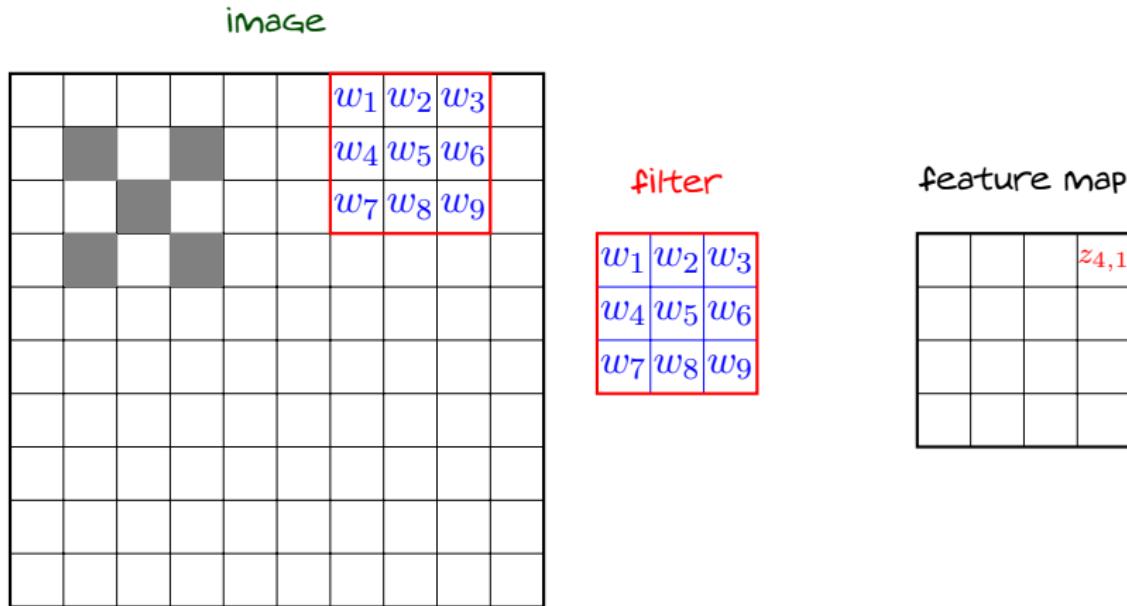
feature map

	$z_{3,1}$	

We could also play with the *stride* \equiv the step-size by which we move *filter*

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

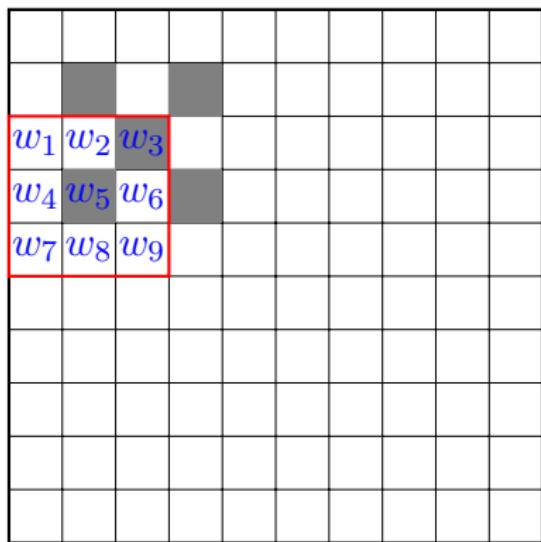


We could also play with the **stride** \equiv the step-size by which we move **filter**

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

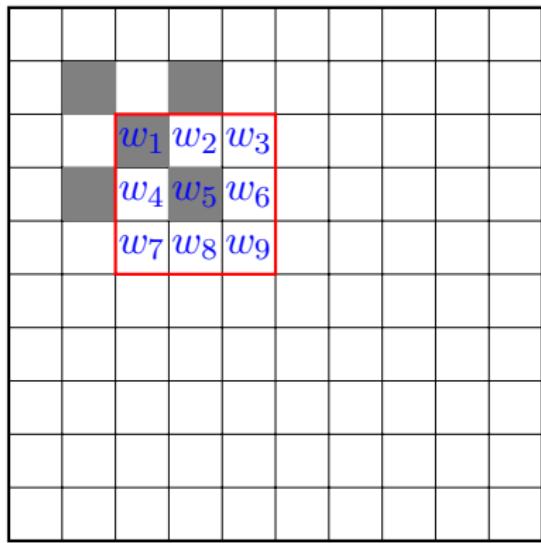
$z_{1,2}$		

We could also play with the *stride* \equiv the step-size by which we move *filter*

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

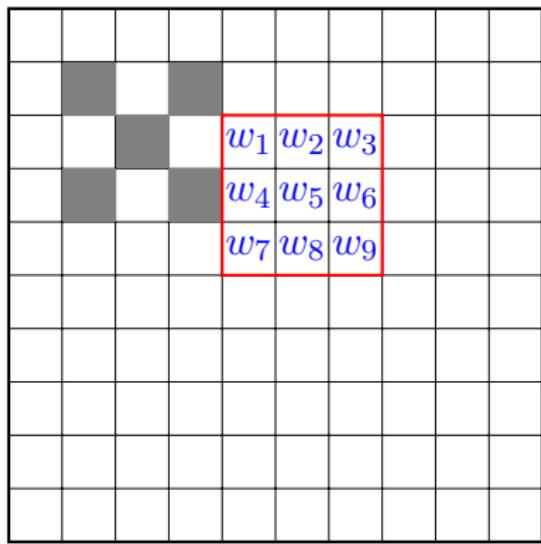
	$z_{2,2}$	

We could also play with the *stride* \equiv the step-size by which we move filter

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

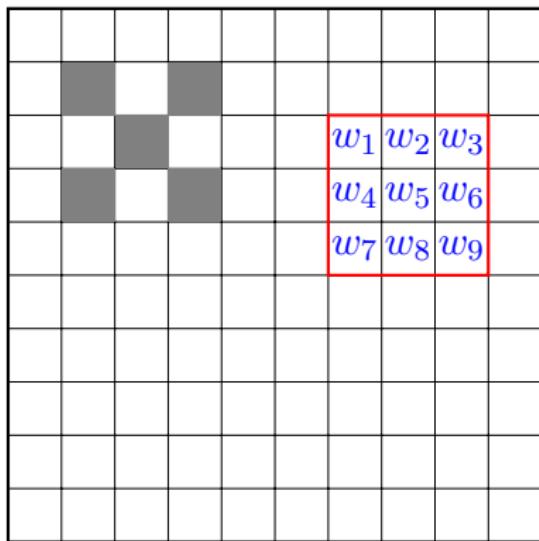
	$z_{3,2}$	

We could also play with the *stride* \equiv the step-size by which we move filter

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

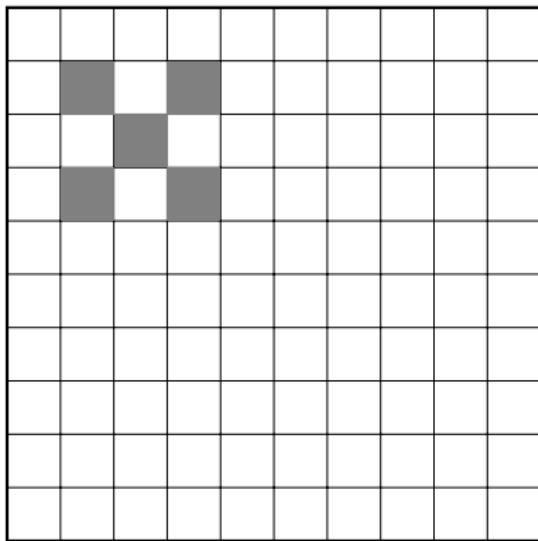
		$z_{4,2}$

We could also play with the *stride* \equiv the step-size by which we move filter

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride 2

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

feature map

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

We could also play with the *stride* \equiv the step-size by which we move filter

$$\text{feature map} = \text{Conv}(\text{image} | \text{filter}, \text{stride} = 2)$$

Convolution with Stride S

Let's formulate the convolution for a general filter: assume $\mathbf{W} \in \mathbb{R}^{F \times F}$ be a **filter**, we also call it **kernel**. Let $\mathbf{X} \in \{N \times N\}$ be **pixel matrix** of the **image**. We want to find the output **feature map**, i.e.,

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}, \text{stride} = S)$$

It's enough to find the corresponding sub-matrix for each entry of \mathbf{Z}

$$\mathbf{Z}[i, j] = \text{sum}(\mathbf{W} \odot \mathbf{X}_{i,j})$$

where $\mathbf{X}_{i,j}$ is the corresponding $F \times F$ sub-matrix, i.e.,

$$\mathbf{X}_{i,j} = \mathbf{X}[1 + (i - 1)S : F + (i - 1)S, 1 + (j - 1)S : F + (j - 1)S]$$

Recognizing X: Pooling

The next operation we did is called **pooling**

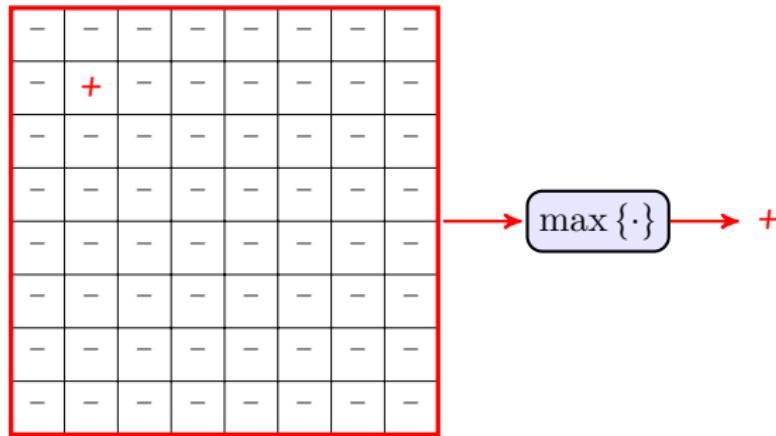
feature map

-	-	-	-	-	-	-	-
-	+	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

Recognizing X: Pooling

The next operation we did is called **pooling**

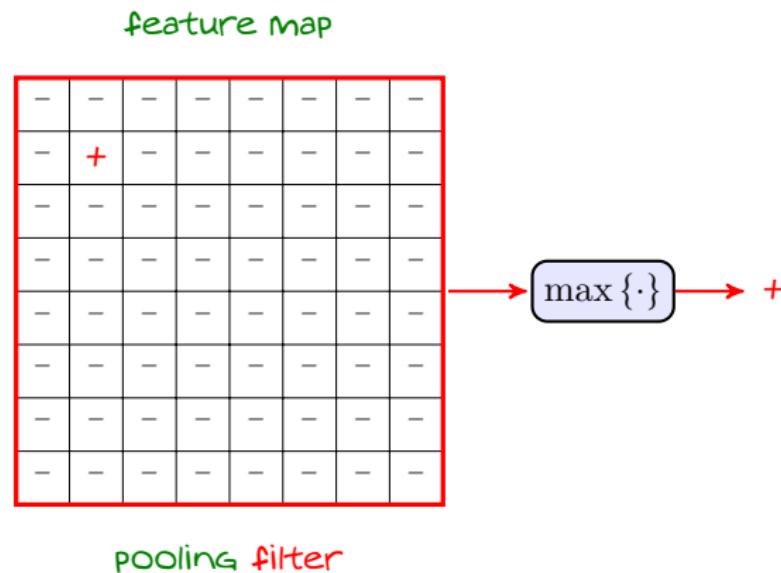
feature map



pooling filter

Recognizing X: Pooling

The next operation we did is called **pooling**

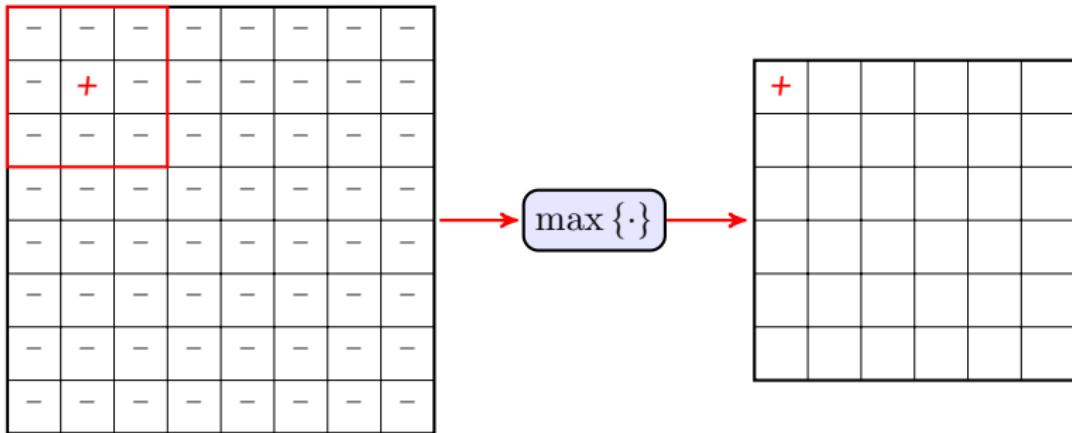


This is however **not conventional** to have a **pooling filter** of different size

Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

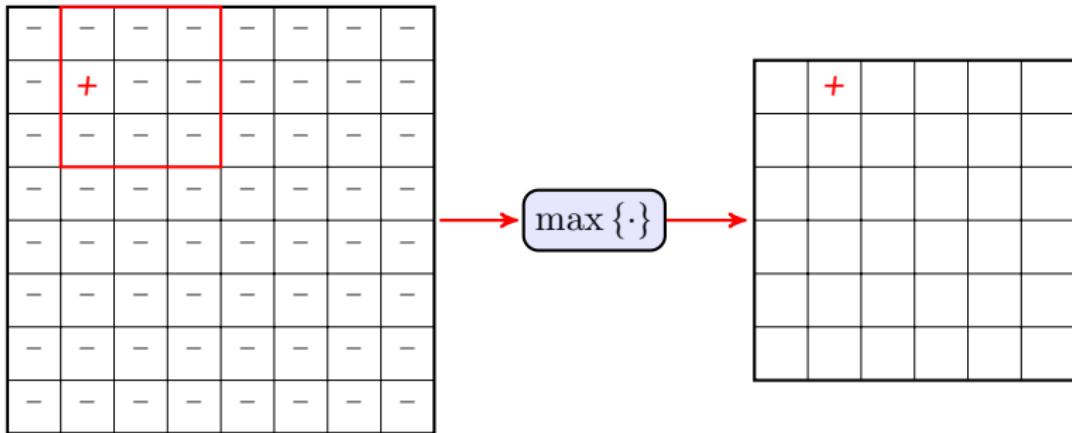
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

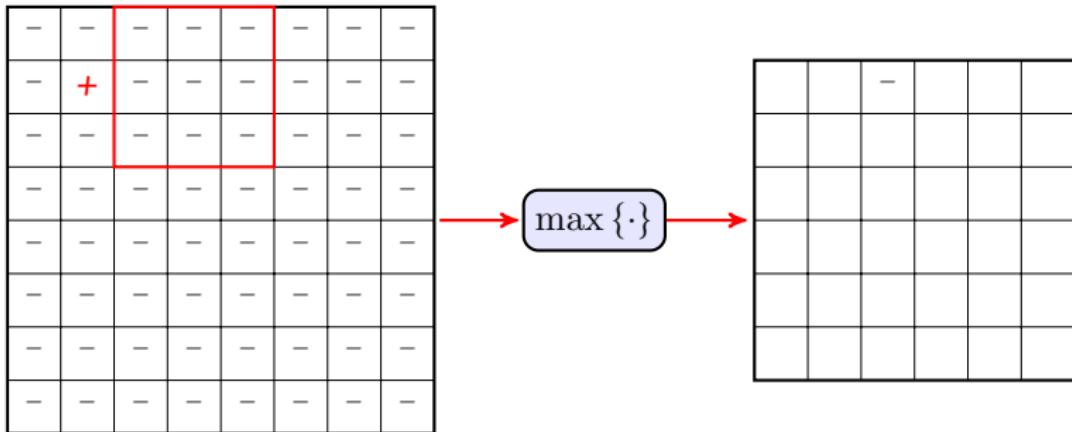
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

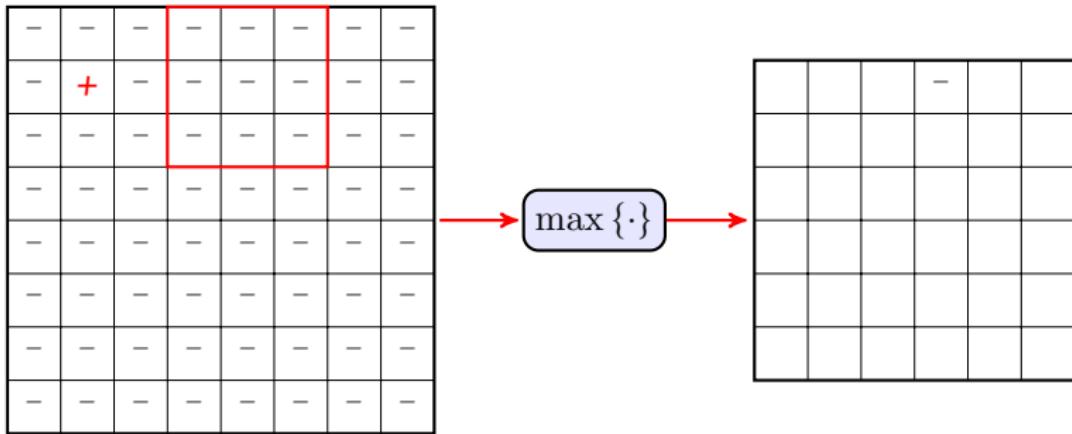
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

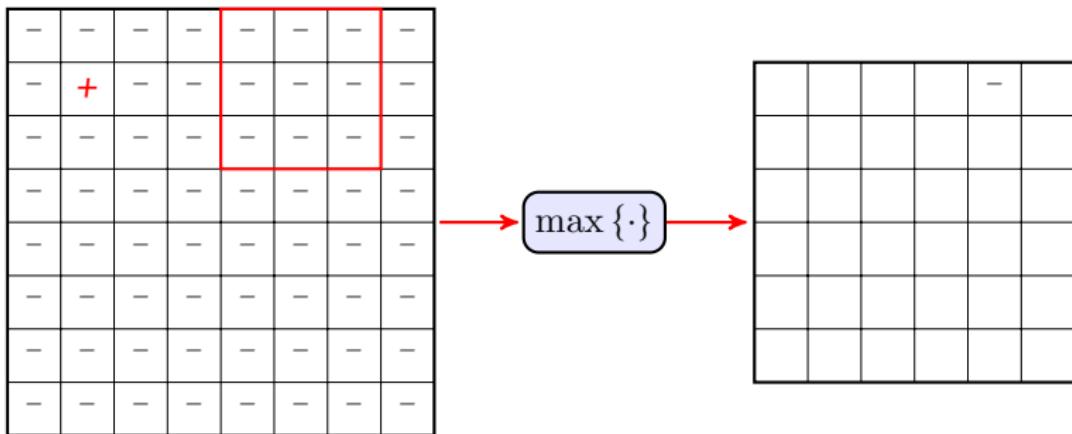
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

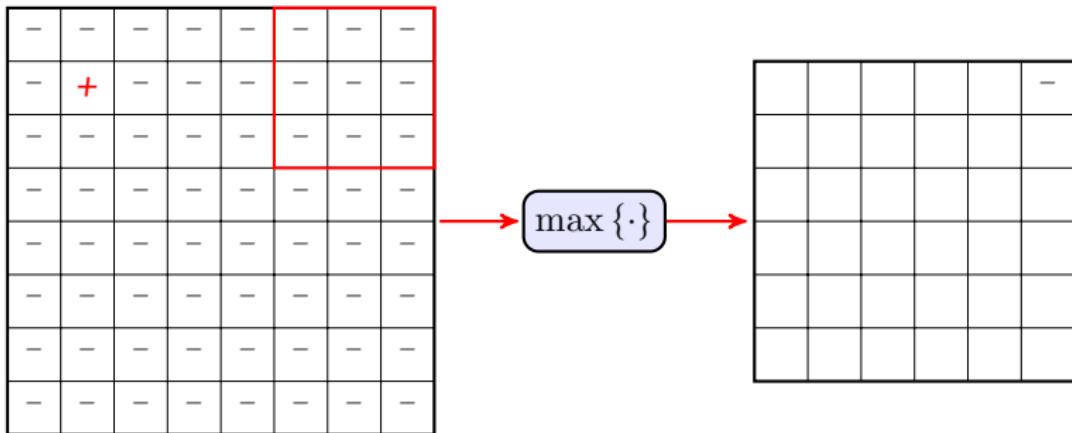
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

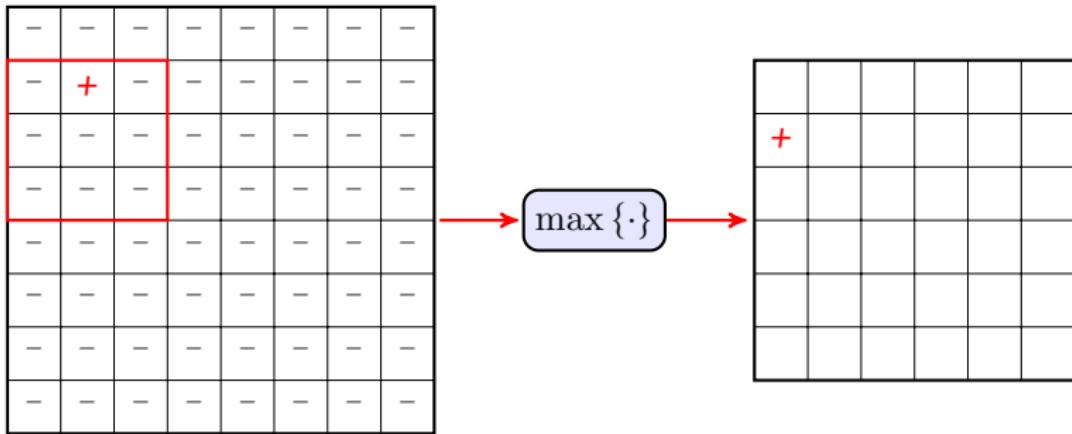
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

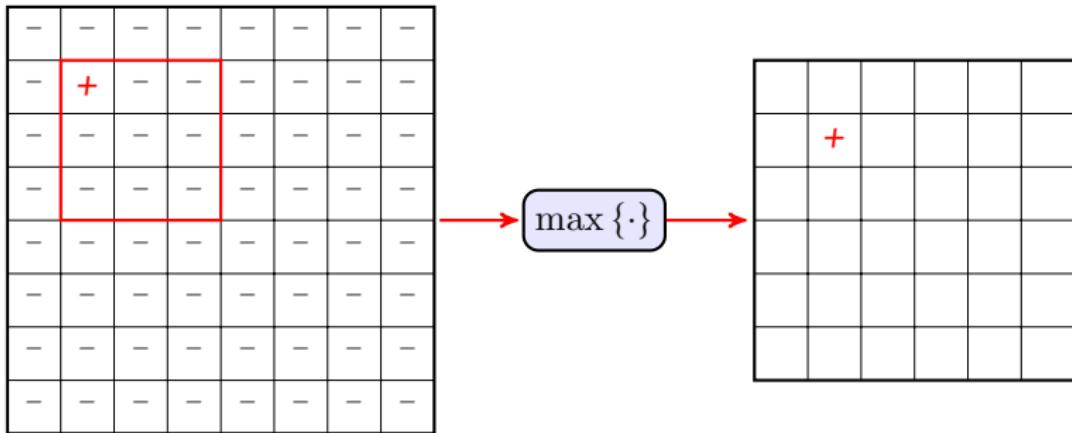
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

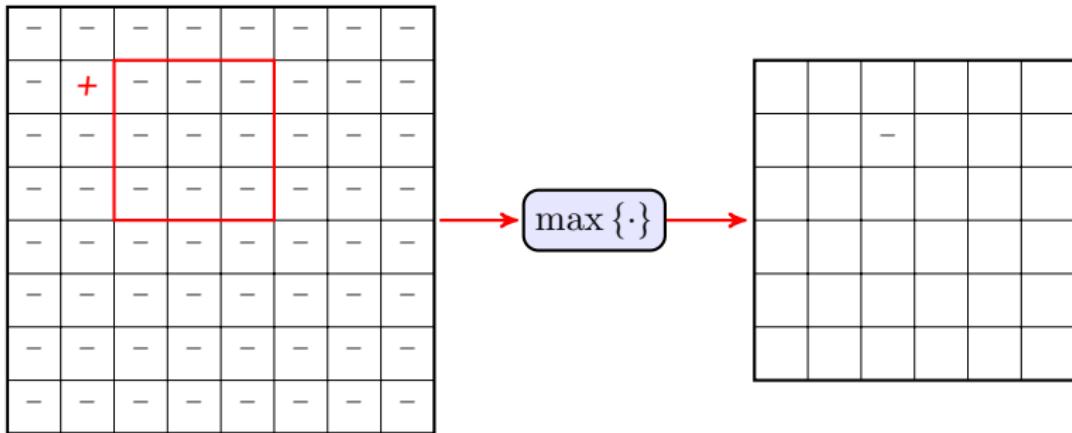
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

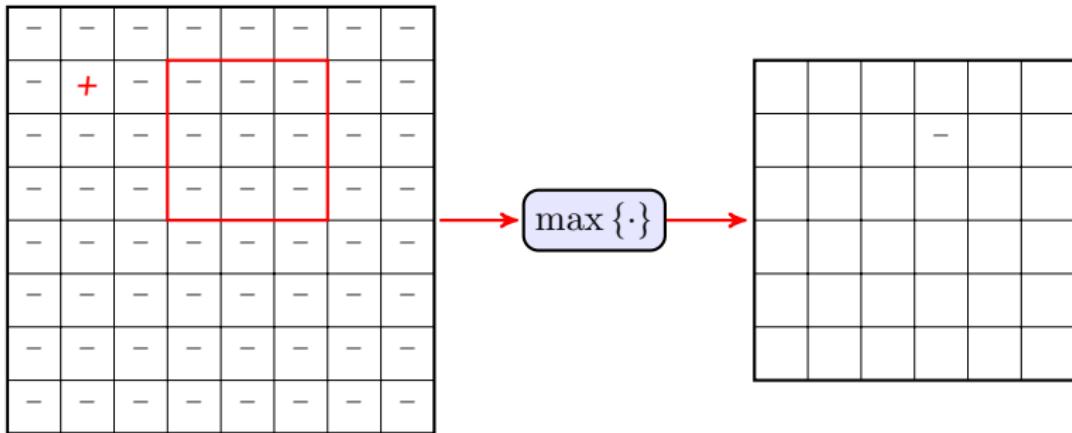
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

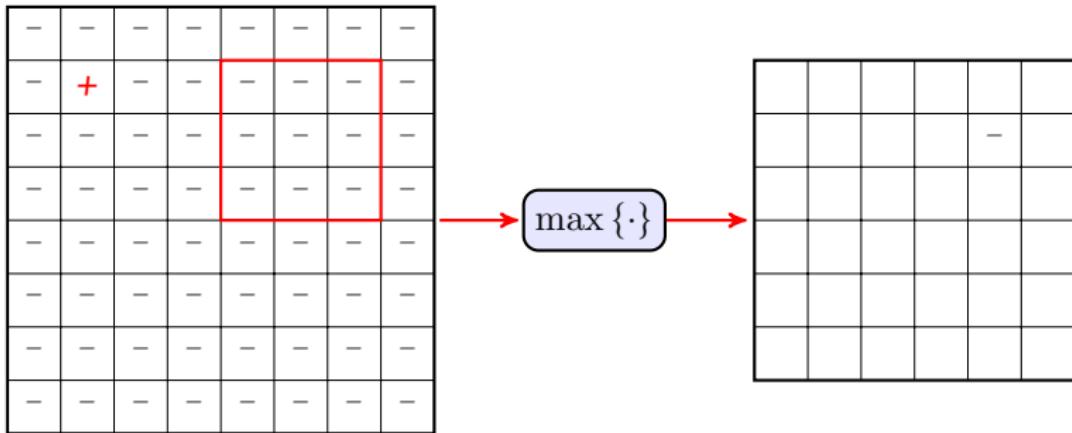
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

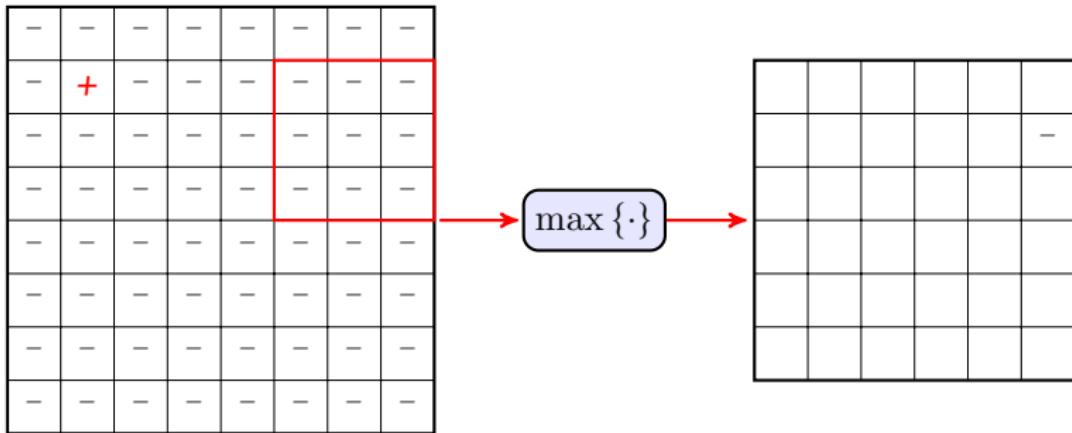
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

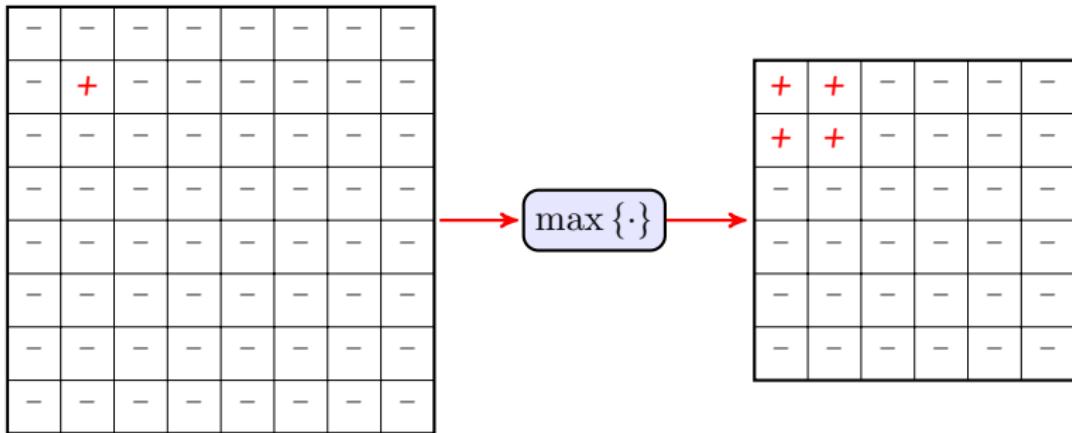
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

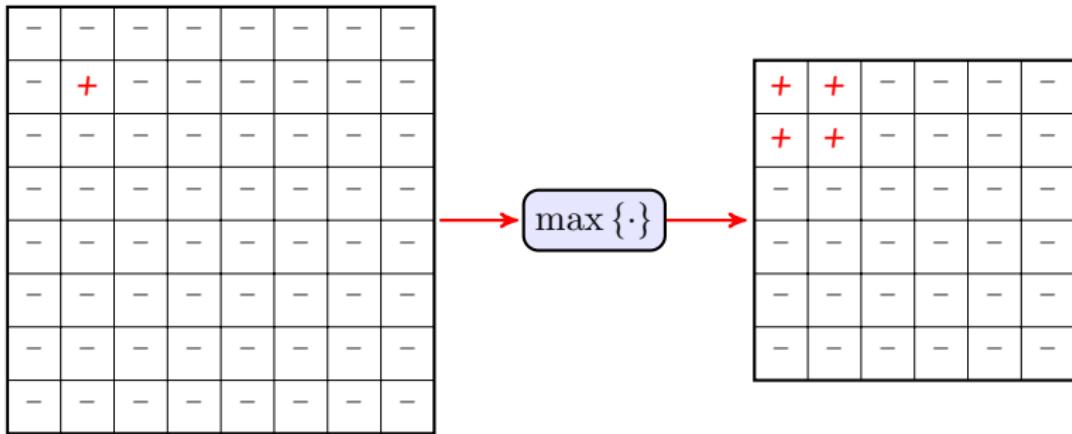
feature map



Pooling: Max Pooling with Stride 1

The convention is to use *the same filter size as used in convolution layer*

feature map

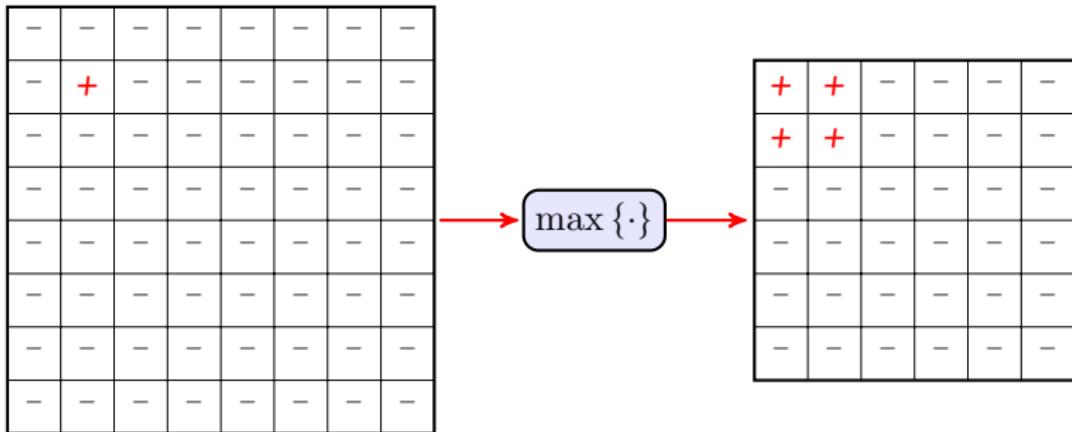


We can now give the *feature map* after pooling to a *fully-connected FNN*: this is a *feature vector of reduced size!*

Pooling: Max Pooling with Stride 1

The convention is to use **the same filter size as used in convolution layer**

feature map

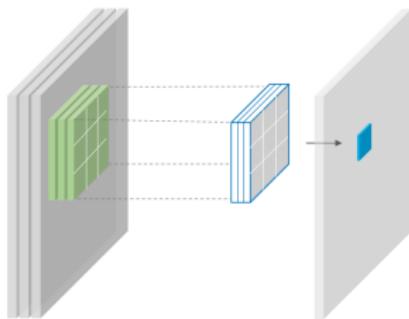


We can now give the **feature map** after pooling to a **fully-connected FNN**: this is a **feature vector of reduced size!**

We can **repeat convolution and pooling over and over**

CNN: Simple Architecture

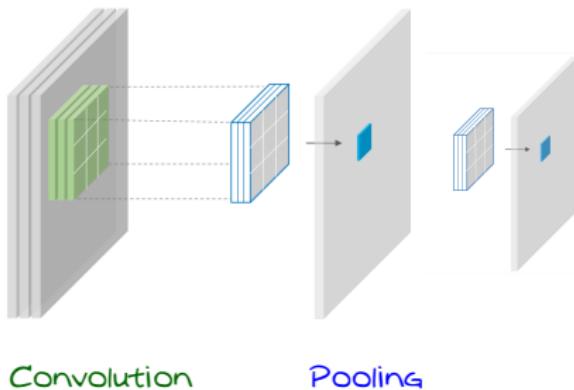
Our simple CNN looks like this



Convolution

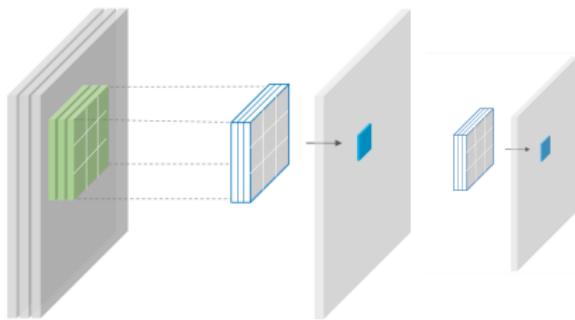
CNN: Simple Architecture

Our simple CNN looks like this



CNN: Simple Architecture

Our simple CNN looks like this



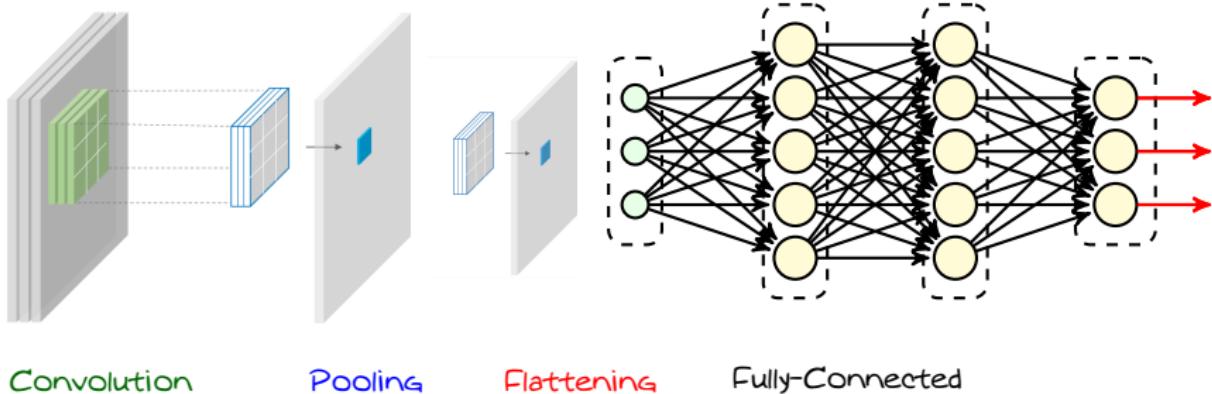
Convolution

Pooling

Flattening

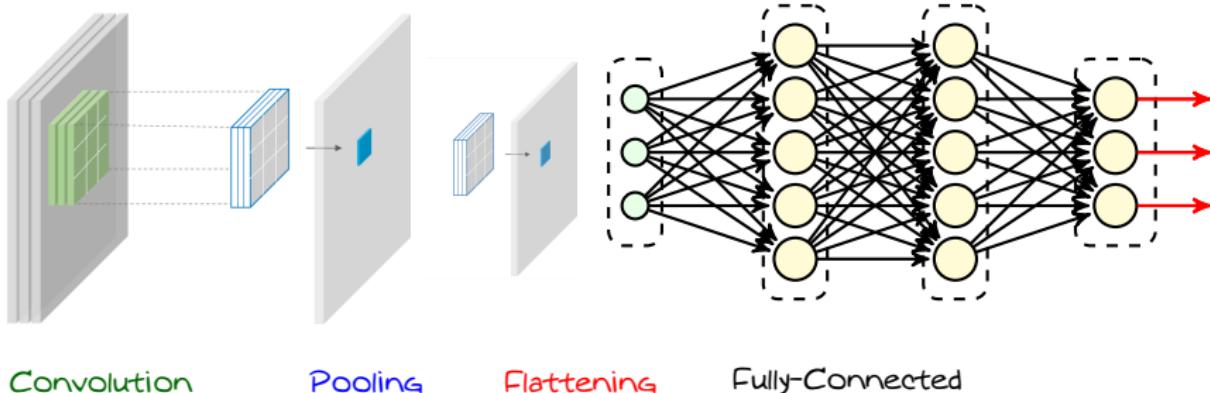
CNN: Simple Architecture

Our simple CNN looks like this



CNN: Simple Architecture

Our simple CNN looks like this



Convolution

Pooling

Flattening

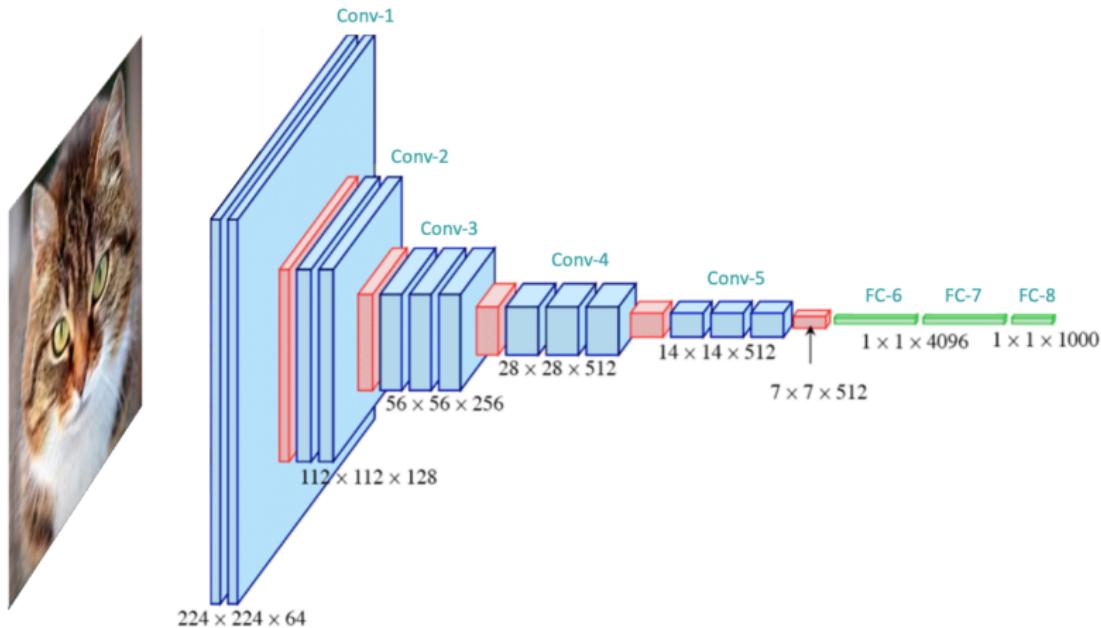
Fully-Connected

This is a **general architecture** for CNNs, but of course we **go deeper!**

- We have more **convolutional** and **pooling layers**
- We do **higher-dimensional convolutions** and **more advanced poolings**

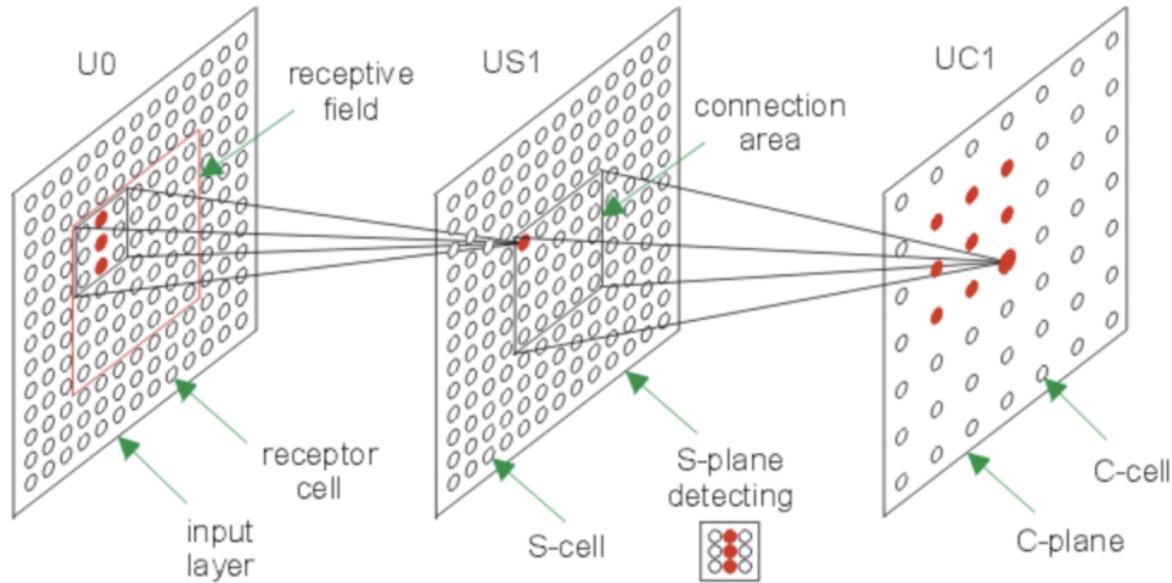
CNN: Realistic Architectures

For instance the famous VGG-16 architecture looks like below



CNN: Connections to Biological Vision

Though it's **artificially** developed as a **computation model**: it is related to the initial model developed for description of **biological vision**



Components of CNNs

CNNs consist of **three** major components

① Convolutional layers

- ↳ they are the **key** component
- ↳ they extract **features** from **input** by sweeping **filters** over **input**

② Pooling layers

- ↳ they **smooth** the extracted features

③ output FNN

- ↳ they learn **label** from **final** extracted features

We now go through each of these components

Convolution: 2D Arrays

We already know the definition of convolution for 2D arrays

2D Convolution

Let $\mathbf{X} \in \mathbb{R}^{N \times M}$ be the *input matrix*: convolution of \mathbf{X} by *filter/kernel* $\mathbf{W} \in \mathbb{R}^{F \times F}$ with *stride* S is denoted by

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}, S)$$

Convolution: 2D Arrays

We already know the definition of convolution for 2D arrays

2D Convolution

Let $\mathbf{X} \in \mathbb{R}^{N \times M}$ be the *input matrix*: convolution of \mathbf{X} by *filter/kernel* $\mathbf{W} \in \mathbb{R}^{F \times F}$ with *stride* S is denoted by

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}, S)$$

The matrix \mathbf{Z} has $\lfloor (N - F)/S \rfloor + 1$ rows and $\lfloor (M - F)/S \rfloor + 1$ columns and its entry at row i and column j is computed as

$$\mathbf{Z}[i, j] = \text{sum}(\mathbf{W} \odot \mathbf{X}_{i,j})$$

Convolution: 2D Arrays

We already know the definition of convolution for 2D arrays

2D Convolution

Let $\mathbf{X} \in \mathbb{R}^{N \times M}$ be the *input matrix*: convolution of \mathbf{X} by *filter/kernel* $\mathbf{W} \in \mathbb{R}^{F \times F}$ with *stride* S is denoted by

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}, S)$$

The matrix \mathbf{Z} has $\lfloor (N - F)/S \rfloor + 1$ rows and $\lfloor (M - F)/S \rfloor + 1$ columns and its entry at row i and column j is computed as

$$\mathbf{Z}[i, j] = \text{sum}(\mathbf{W} \odot \mathbf{X}_{i,j})$$

with $\mathbf{X}_{i,j}$ being the corresponding $F \times F$ sub-matrix of \mathbf{X} , i.e.,

$$\mathbf{X}_{i,j} = \mathbf{X}[1 + (i - 1)S : F + (i - 1)S, 1 + (j - 1)S : F + (j - 1)S]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \left[\quad \right]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad * \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

$\mathbf{X}_{1,1} \rightarrow \text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \left[\quad \right]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad * \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

$\mathbf{X}_{1,1} \rightarrow \text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \left[Z_{1,1} \right]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \boxed{X_{1,3}} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & \boxed{X_{1,3}} & X_{2,4} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \\ & & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & \boxed{X_{1,4}, X_{1,5}} \\ X_{2,1} & X_{2,2} & X_{2,3} & \boxed{X_{2,4}, X_{2,5}} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & \mathbf{X}_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & & & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & \boxed{X_{2,2}} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & \boxed{X_{3,2}} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & \boxed{X_{2,3}} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & \textcolor{blue}{X_{3,3}} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ \boxed{X_{3,1}} & \textcolor{blue}{X_{3,2}} & X_{3,3} & X_{3,4} & X_{3,5} \\ \textcolor{blue}{X_{3,1}} & \textcolor{blue}{X_{3,2}} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,1} & & & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & \boxed{X_{3,2}} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & \textcolor{blue}{X_{3,2}} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,1} & Z_{3,2} & & \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & \boxed{X_{3,3}} & X_{3,4} \\ X_{4,1} & X_{4,2} & X_{4,3} & \textcolor{blue}{X_{4,4}} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & \boxed{X_{3,4}} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & \boxed{X_{4,5}} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} & Z_{3,4} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

If we convolve with stride $S = 1$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & Z_{1,4} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & Z_{2,4} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} & Z_{3,4} \end{bmatrix}$$

Let's verify the dimensions of \mathbf{Z}

$$\# \text{ rows} = \lfloor (4 - 2)/1 \rfloor + 1 = 3 \quad \# \text{ columns} = \lfloor (5 - 2)/1 \rfloor + 1 = 4$$

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1			

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1		

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5			

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2		

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4
1.7			

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4
1.7	2		

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4
1.7	2	2.5	

1	0
1	1

Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4
1.7	2	2.5	3.4

1	0
1	1

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad * \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \left[\quad \right]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad * \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

$\mathbf{X}_{1,1} \rightarrow \text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \left[\quad \right]$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad * \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

$\mathbf{X}_{1,1} \rightarrow \text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \boxed{X_{1,3}} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & \boxed{X_{1,2}} & X_{2,4} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ \boxed{X_{3,1}} & \textcolor{blue}{X_{3,2}} & X_{3,3} & X_{3,4} & X_{3,5} \\ \textcolor{blue}{X_{2,1}} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & \boxed{X_{3,3}} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & \textcolor{blue}{X_{4,2}} & X_{4,4} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} & Z_{2,2} \end{bmatrix}$$

Convolution: 2D Arrays

Let's see an example: assume $\mathbf{X} \in \mathbb{R}^{4 \times 5}$ and $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} & X_{1,5} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} & X_{2,5} \\ X_{3,1} & X_{3,2} & X_{3,3} & X_{3,4} & X_{3,5} \\ X_{4,1} & X_{4,2} & X_{4,3} & X_{4,4} & X_{4,5} \end{bmatrix} \quad \circledast \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

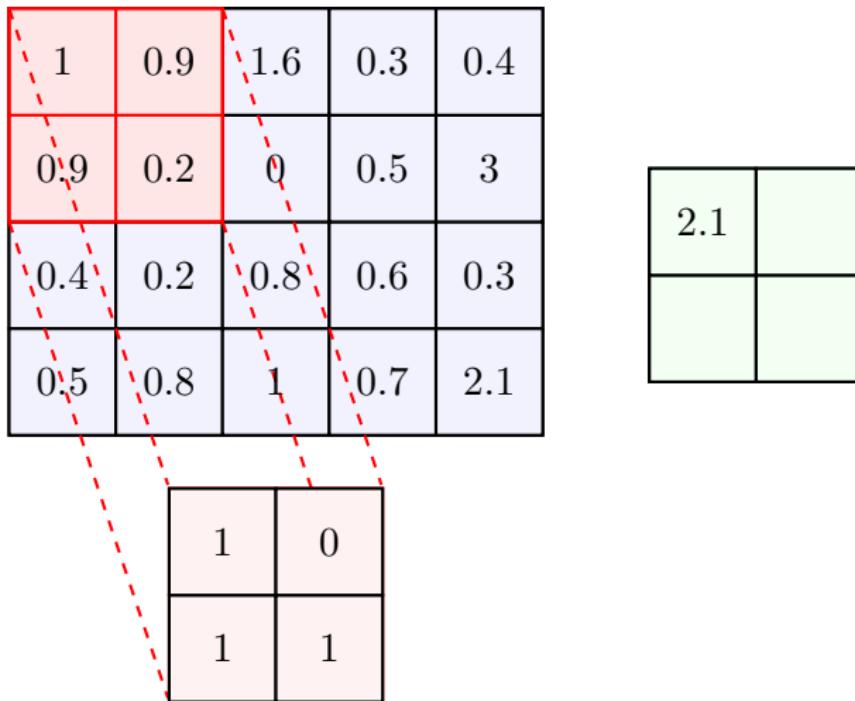
Now, we convolve with stride $S = 2$

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} & Z_{2,2} \end{bmatrix}$$

Let's verify the dimensions of \mathbf{Z}

$$\# \text{ rows} = \lfloor (4 - 2)/2 \rfloor + 1 = 2 \quad \# \text{ columns} = \lfloor (5 - 2)/2 \rfloor + 1 = 2$$

Convolution: Numerical Example



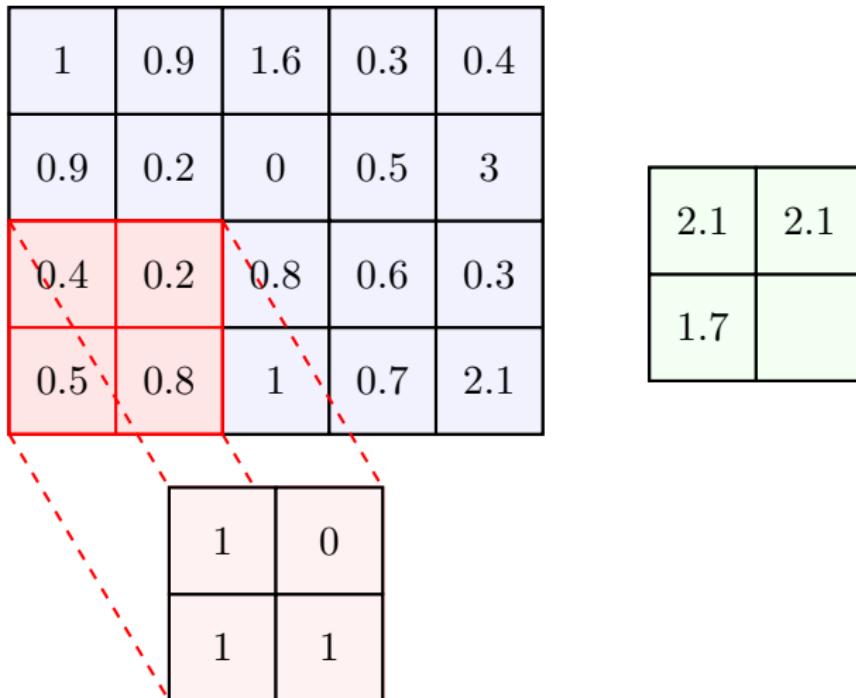
Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	2.1

1	0
1	1

Convolution: Numerical Example



Convolution: Numerical Example

1	0.9	1.6	0.3	0.4
0.9	0.2	0	0.5	3
0.4	0.2	0.8	0.6	0.3
0.5	0.8	1	0.7	2.1

2.1	2.1
1.7	2.5

1	0
1	1

Convolution: *Downsampling*

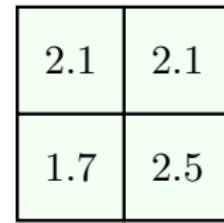
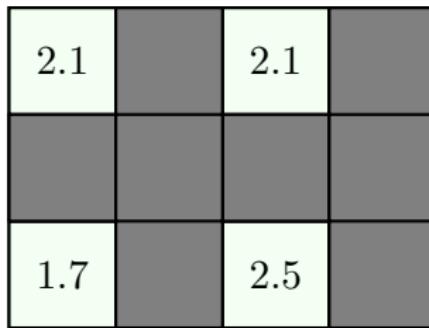
Let's compare the result with strides 1 and 2

2.1	1.1	2.1	3.8
1.5	1.2	1.4	1.4
1.7	2	2.5	3.4

2.1	2.1
1.7	2.5

Convolution: *Downsampling*

Let's compare the result with strides 1 and 2



Convolution: Downsampling

Let's compare the result with strides 1 and 2

2.1		2.1	
1.7		2.5	

2.1	2.1
1.7	2.5

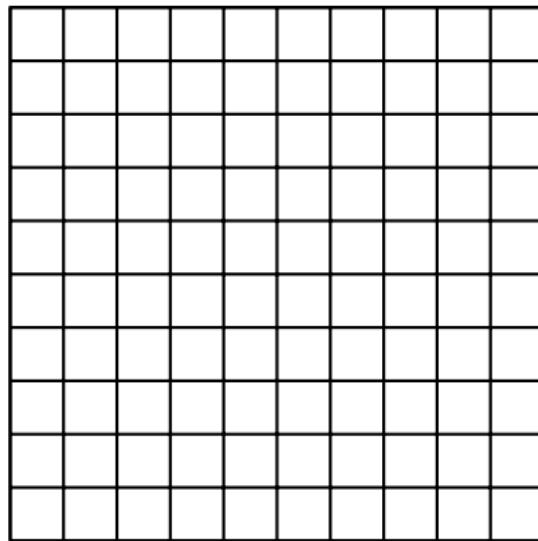
This is **downsampling** with factor 2!

Downsampling

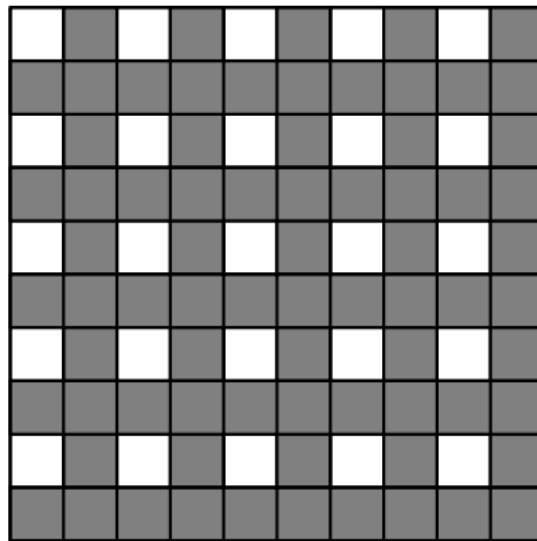
Downsampling with factor f drops the last $f - 1$ of every f columns and rows

$$\text{dSample}(\mathbf{Z}|f)$$

Convolution: *Downsampling*

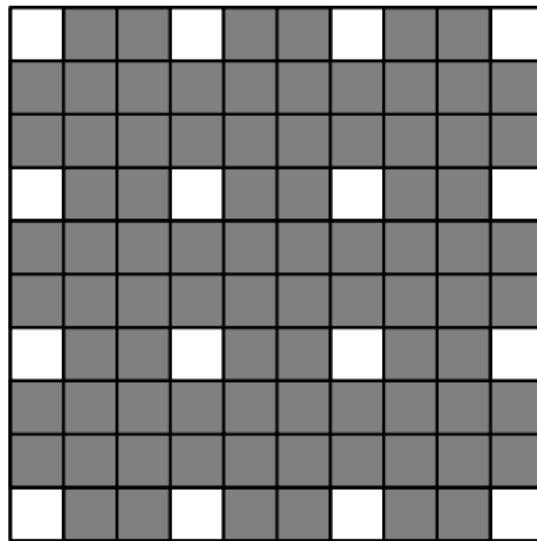


Convolution: *Downsampling*



Downsampling with factor 2 dSample ($\mathbf{Z}|2$)

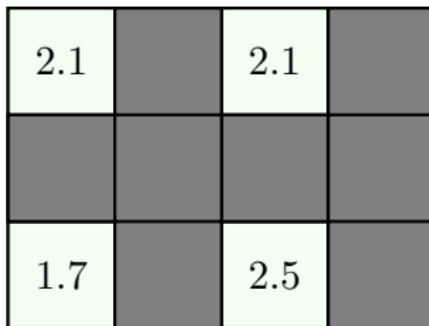
Convolution: *Downsampling*



Downsampling with factor 3 dSample ($\mathbf{Z}|3$)

Convolution: Downsampling

Let's compare the result with strides 1 and 2



Stride as Downsampling

We can look at stride S as **downsampling** with factor S , i.e.,

$$\text{Conv}(\mathbf{X}|\mathbf{W}, S) = \text{dSample}(\text{Conv}(\mathbf{X}|\mathbf{W}, 1) | S)$$

Convolution: *Downsampling*

So, we can make an agreement: *by default we consider unit stride, i.e., $S = 1$, i.e., we drop S in convolution from now on*

$$\text{Conv}(\mathbf{X}|\mathbf{W}) = \text{Conv}(\mathbf{X}|\mathbf{W}, S=1)$$

Whenever we need to convolve with $\text{stride } S > 1$

We add an *downsampling* next to the convolution

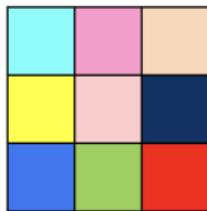
- + Why do we do this?
- We'll see how easy things get when we want to *backpropagate*; however, it also helps to easily define having a *stride smaller than one!*

Convolution: Upsampling

We can do the *sampling* in other way, i.e., add some zeros

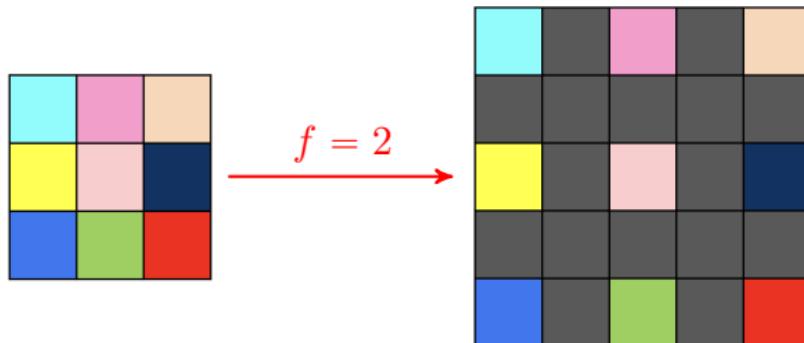
Convolution: Upsampling

We can do the *sampling* in other way, i.e., add some zeros



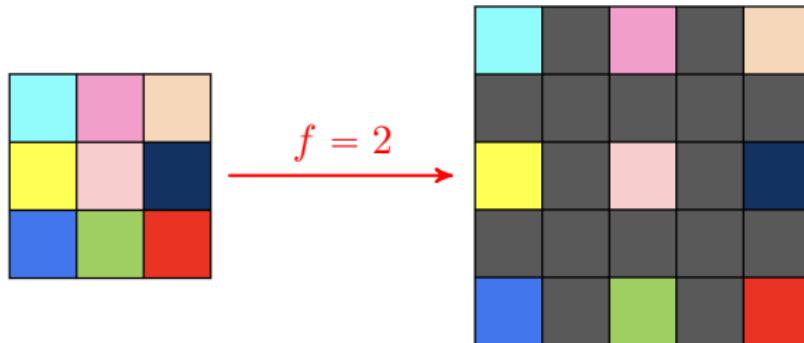
Convolution: Upsampling

We can do the *sampling* in other way, i.e., add some zeros



Convolution: Upsampling

We can do the *sampling* in other way, i.e., add some zeros



Upsampling

Upsampling with factor f adds $f - 1$ rows and columns of zeros after every f row and column

$$\text{uSample}(\mathbf{Z}|f)$$

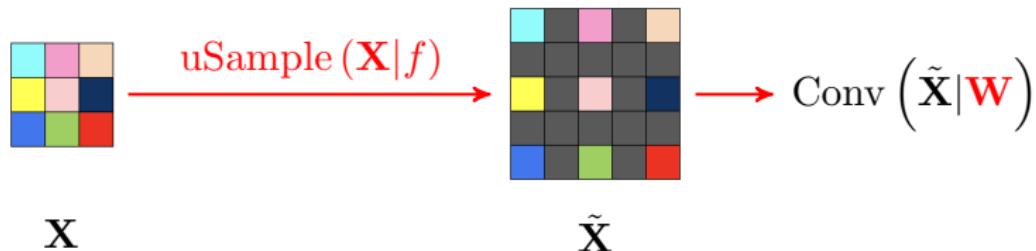
Convolution with Fractional Stride

- + Where do we use *upsampling*?
- If we want to **increase** the size of the feature map

Convolution with Fractional Stride

- + Where do we use **upsampling**?
- If we want to **increase** the size of **the feature map**

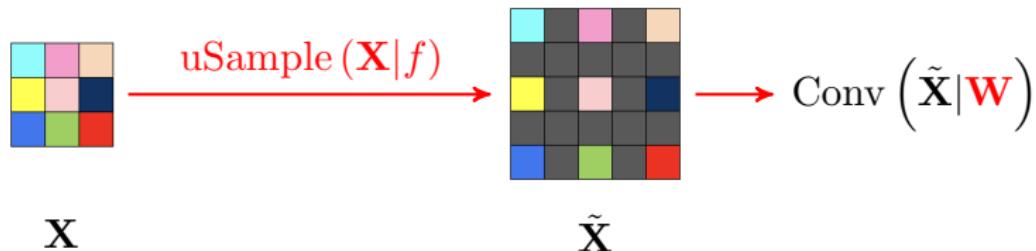
To **increase** the size of **the feature map**, we can do following



Convolution with Fractional Stride

- + Where do we use **upsampling**?
- If we want to **increase** the size of the feature map

To **increase** the size of the feature map, we can do following



This is called **fractionally-strided convolution**: for $S < 1$ with $1/S$ being integer

$$\text{Conv}(\mathbf{X}|\mathbf{W}, S) = \text{Conv}(\text{uSample}(\mathbf{X}|1/S)|\mathbf{W})$$

Convolution: Resampling

We can extend **stride** to any fraction $S = S_2/S_1$ with integer S_1 and S_2 : we first do **upsampling** with factor S_1 and then **downsampling** with factor S_2

Convolution: Resampling

We can extend **stride** to any fraction $S = S_2/S_1$ with integer S_1 and S_2 : we first do **upsampling** with factor S_1 and then **downsampling** with factor S_2

Moral of Story

Convolution with **stride** is equivalent with

convolution with resampling

Note that the **order of resampling** differs

- **Upsampling** is done always *before* convolution
- **Downsampling** is done always *after* convolution

The above interpretation of **stride** helps a lot in **backpropagation!**

Convolution: Padding

In our definition, feature map Z has *smaller* dimensions than X even with stride $S = 1$. We can play with *dimensions* of Z via *zero-padding*

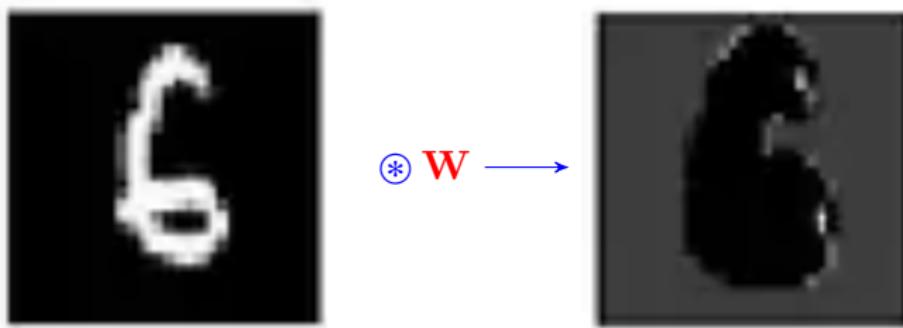
- + Why should we be interested in *changing* dimensions Z ?
- We'll see *multiple* reasons: a simple one is that we may like to have *a same-size feature map* to sketch it *as an image* and compare it to *input*

Convolution: Padding

In our definition, feature map Z has *smaller* dimensions than X even with stride $S = 1$. We can play with *dimensions* of Z via *zero-padding*

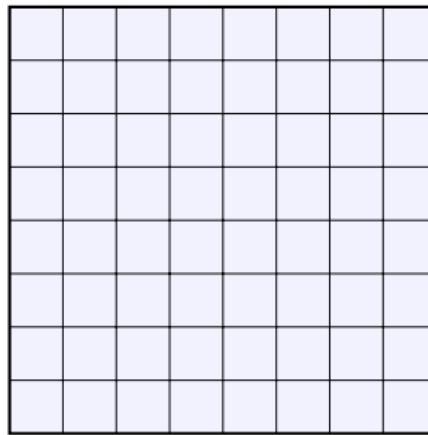
- + Why should we be interested in *changing* dimensions Z ?
- We'll see *multiple* reasons: a simple one is that we may like to have *a same-size feature map* to sketch it *as an image* and compare it to *input*

For instance in MNIST, we want to sketch the *feature map* as a 28×28 image



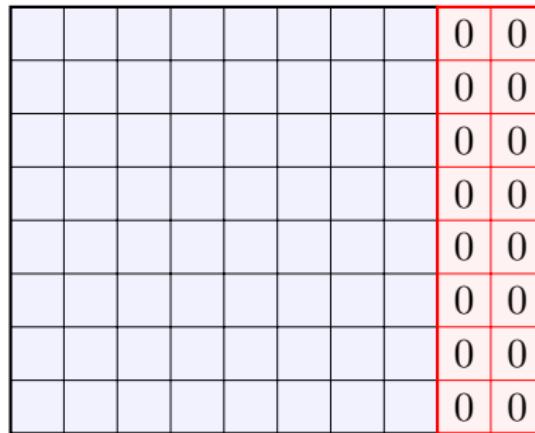
Convolution: Zero-Padding

The trick to *resize feature map* is to *pad zeros at boundaries*



Convolution: Zero-Padding

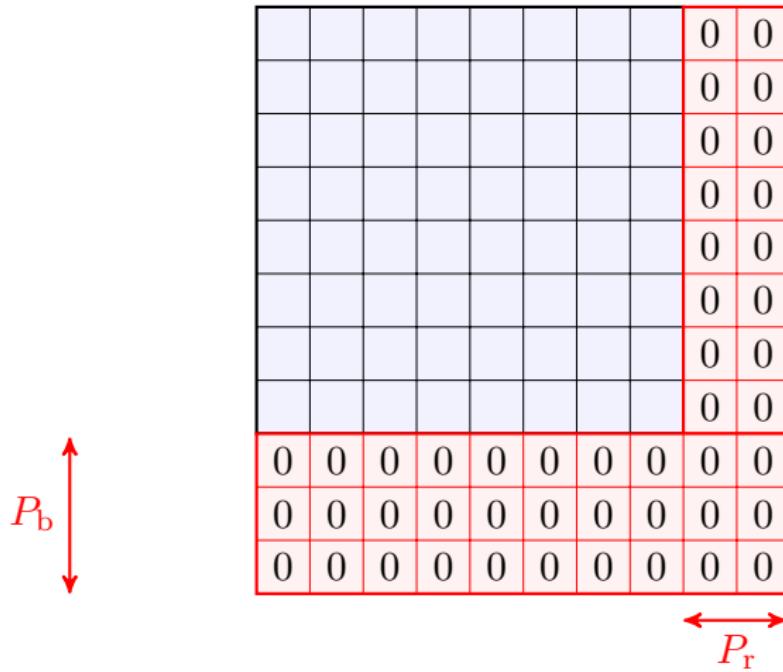
The trick to *resize feature map* is to *pad zeros at boundaries*



$$\overleftarrow{\overrightarrow{P_r}}$$

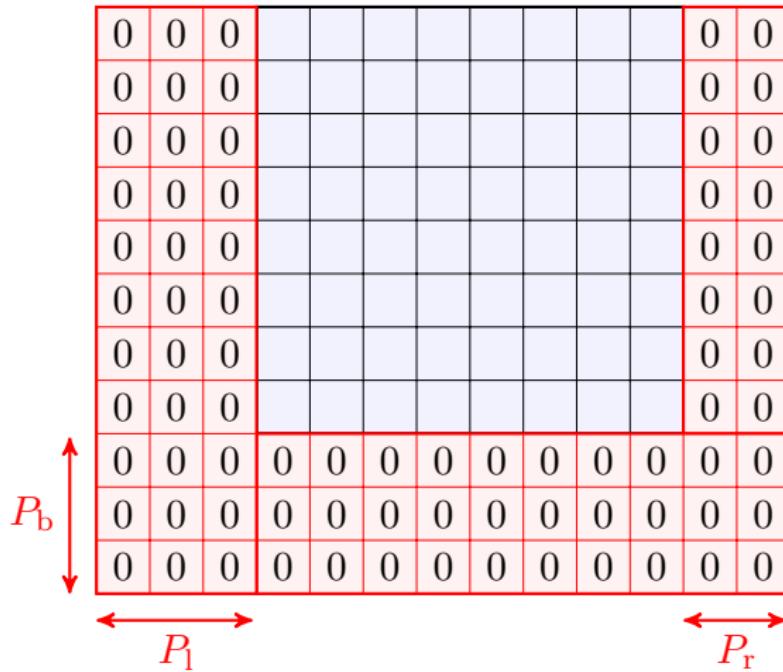
Convolution: Zero-Padding

The trick to *resize feature map* is to *pad zeros at boundaries*



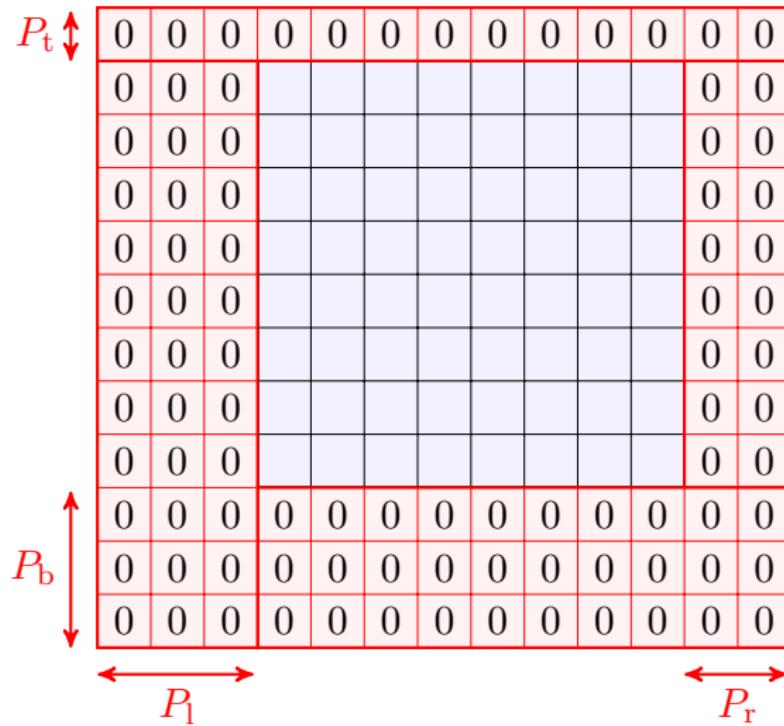
Convolution: Zero-Padding

The trick to *resize feature map* is to *pad zeros at boundaries*



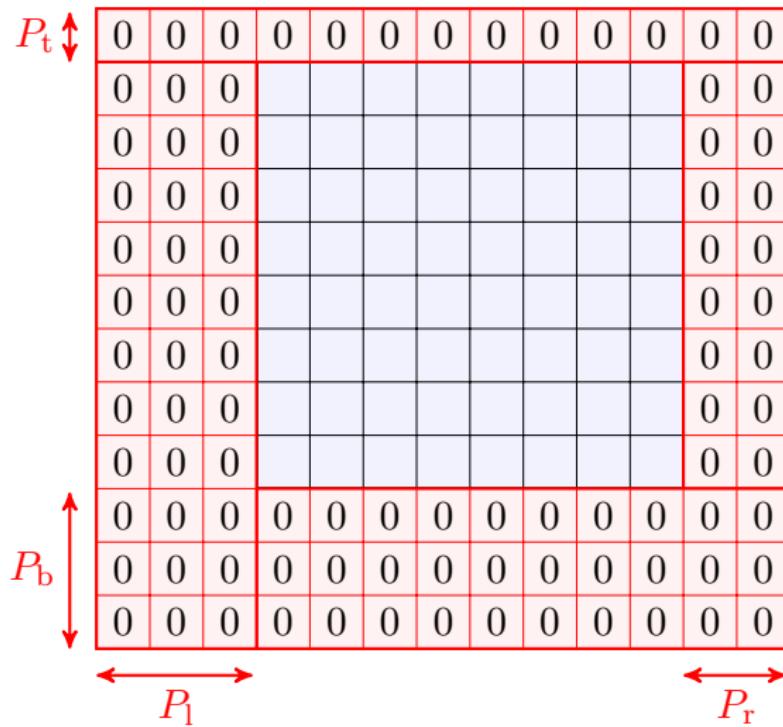
Convolution: Zero-Padding

The trick to *resize feature map* is to *pad zeros at boundaries*



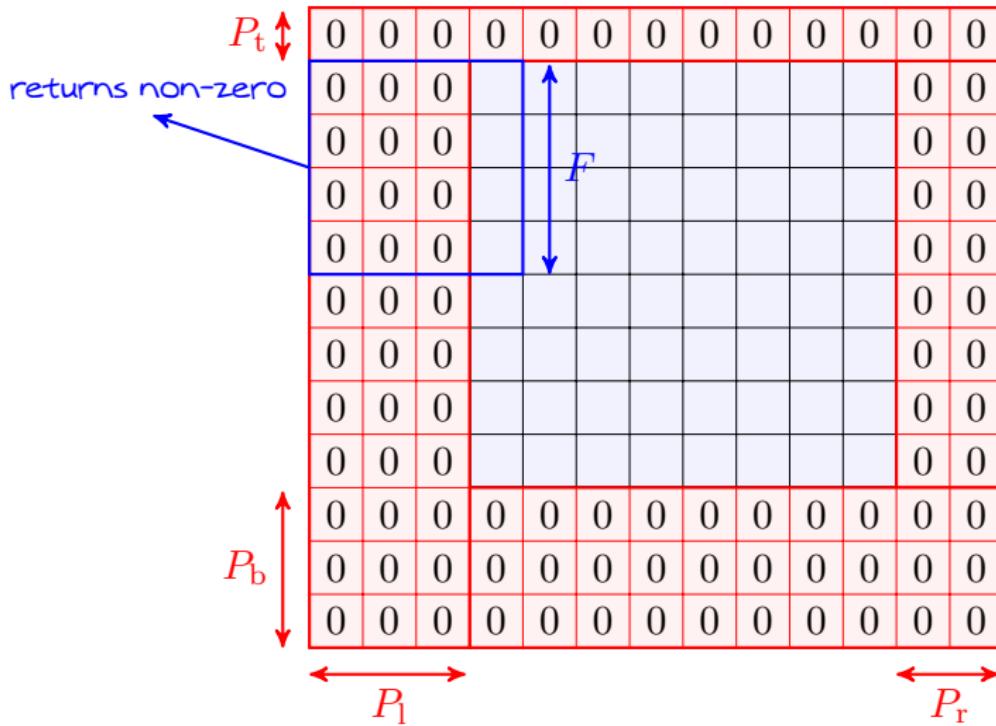
Convolution: Zero-Padding

Typically we *pad* with widths smaller than *filter dimension F*



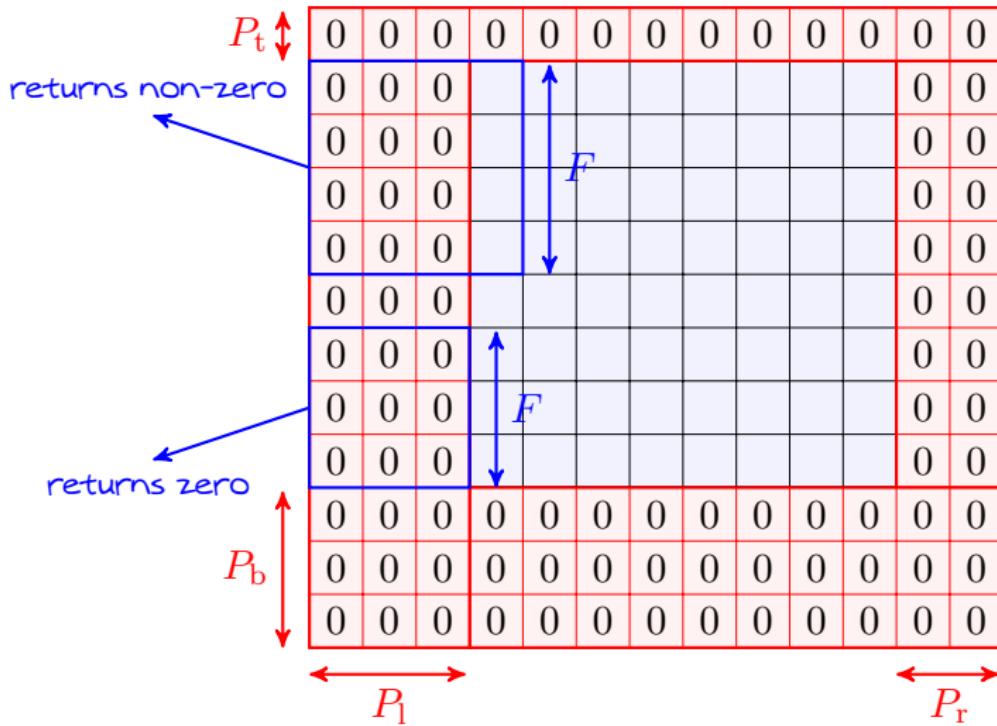
Convolution: Zero-Padding

Typically we *pad* with widths smaller than *filter dimension F*



Convolution: Zero-Padding

Typically we *pad* with widths smaller than *filter dimension F*



Convolution: Zero-Padding

With **zero-padding**, the dimensions of **feature map** can be modified: say the **input** has N rows and M columns; then, at the feature map we have

- $\lfloor (N + P_t + P_b - F)/S \rfloor + 1$ rows
- $\lfloor (M + P_l + P_r - F)/S \rfloor + 1$ columns

In practice, we **pad symmetrically**, i.e., $P_b = P_t = P_r = P_l$

Convolution: Zero-Padding

With **zero-padding**, the dimensions of **feature map** can be modified: say the **input** has N rows and M columns; then, at the feature map we have

- $\lfloor (N + P_t + P_b - F)/S \rfloor + 1$ rows
- $\lfloor (M + P_l + P_r - F)/S \rfloor + 1$ columns

In practice, we **pad symmetrically**, i.e., $P_b = P_t = P_r = P_l$

As we see the dimensions of **feature map** is a function of

input dimensions, stride, padding length and filter size

It is thus typical that we get some of this items **not specified**,

Convolution: Zero-Padding

With **zero-padding**, the dimensions of **feature map** can be modified: say the **input** has N rows and M columns; then, at the feature map we have

- $\lfloor (N + P_t + P_b - F)/S \rfloor + 1$ rows
- $\lfloor (M + P_l + P_r - F)/S \rfloor + 1$ columns

In practice, we **pad symmetrically**, i.e., $P_b = P_t = P_r = P_l$

As we see the dimensions of **feature map** is a function of

input dimensions, stride, padding length and **filter size**

It is thus typical that we get some of this items **not specified**, e.g., we might get **input and output dimensions, stride** and **filter size** but not the **padding length**

we can find the **padding length** from other specifications

Example: Stride = 2 and Padding

1	0.9	1.6	0.3	0.4	0
0.9	0.2	0	0.5	3	0
0.4	0.2	0.8	0.6	0.3	0
0.5	0.8	1	0.7	2.1	0

2.1		

1	0
1	1

Example: Stride = 2 and Padding

1	0.9	1.6	0.3	0.4	0
0.9	0.2	0	0.5	3	0
0.4	0.2	0.8	0.6	0.3	0
0.5	0.8	1	0.7	2.1	0

2.1	2.1	

1	0
1	1

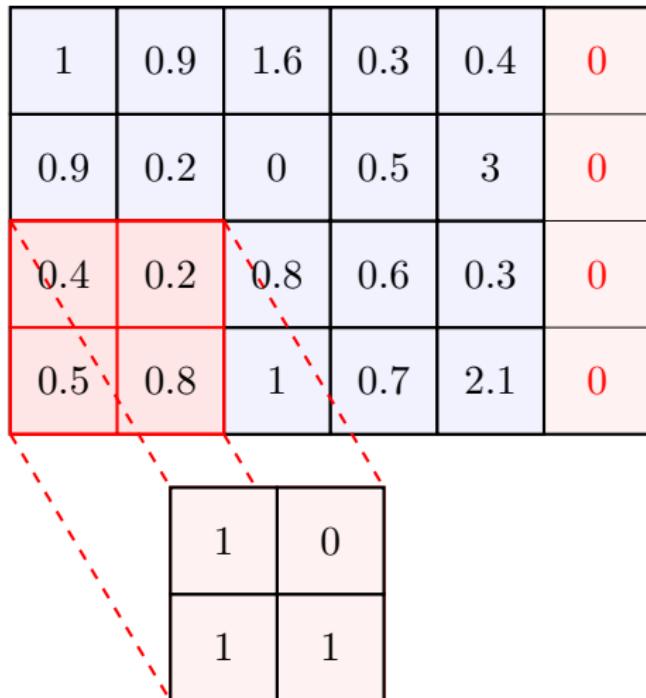
Example: Stride = 2 and Padding

1	0.9	1.6	0.3	0.4	0
0.9	0.2	0	0.5	3	0
0.4	0.2	0.8	0.6	0.3	0
0.5	0.8	1	0.7	2.1	0

2.1	2.1	3.4

1	0
1	1

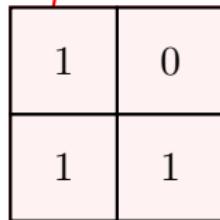
Example: Stride = 2 and Padding



Example: Stride = 2 and Padding

1	0.9	1.6	0.3	0.4	0
0.9	0.2	0	0.5	3	0
0.4	0.2	0.8	0.6	0.3	0
0.5	0.8	1	0.7	2.1	0

2.1	2.1	3.4
1.7	2.5	



Example: Stride = 2 and Padding

1	0.9	1.6	0.3	0.4	0
0.9	0.2	0	0.5	3	0
0.4	0.2	0.8	0.6	0.3	0
0.5	0.8	1	0.7	2.1	0

2.1	2.1	3.4
1.7	2.5	2.4

1	0
1	1

Convolution: Activation

Convolution is a *spatial linear transform* on the input image

we should activate this linear transform if we go deep

Convolution: Activation

Convolution is a *spatial linear transform* on the input image

we should *activate* this *linear transform* if we go *deep*

A convolutional layer can hence be formally defined as below

Convolutional Layer

Convolutional layer with *filter $\mathbf{W} \in \mathbb{R}^{F \times F}$, bias b and activation function $f(\cdot)$* transforms the *input map \mathbf{X}* to the *activated feature map \mathbf{Y}* as follows:

- ① it first applies *linear convolution* to find \mathbf{Z}

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}) + b \quad + \text{applied entry-wise}$$

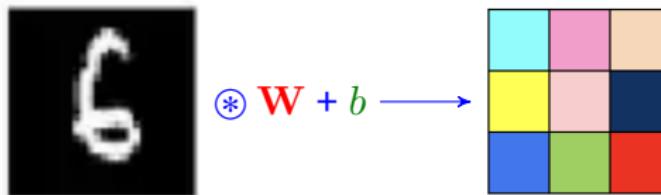
- ② it then *activates* \mathbf{Z}

$$\mathbf{Y} = f(\mathbf{Z}) \quad f(\cdot) \text{ applied entry-wise}$$

Convolution: Activation

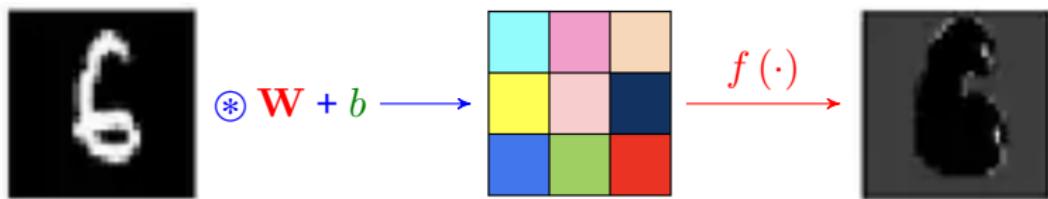


Convolution: Activation



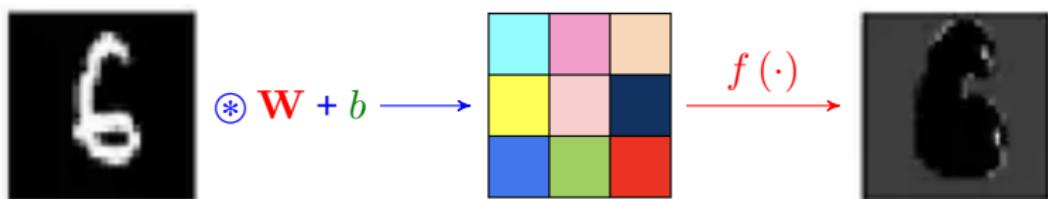
- + We did not discuss *bias* before!
- It's just a *scalar* added to *all entries*

Convolution: Activation



- + We did not discuss *bias* before!
- It's just a *scalar added to all entries*

Convolution: Activation



- + We did not discuss **bias** before!
- It's just a **scalar added to all entries**
- + What kinds of **activation** are used in CNNs?
- Similar to FNNs with **ReLU** being the **popular one**

Convolution: Multi-Channel Input

What we discussed up to now *holds for single-channel images*: these are gray images that can be represented by *2D pixel arrays*, e.g., MNIST images

Convolution: Multi-Channel Input

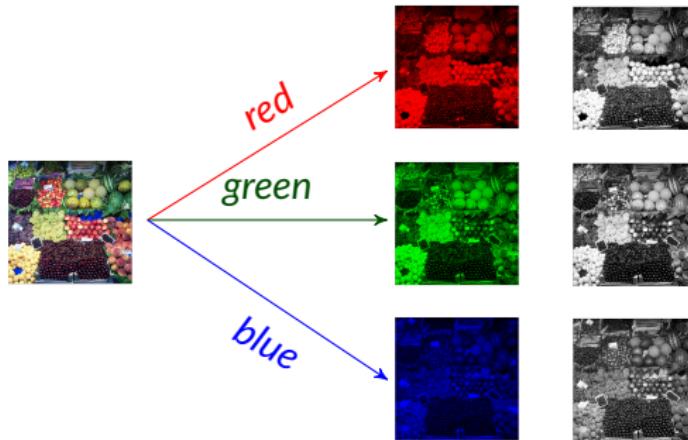
What we discussed up to now *holds for single-channel images*: these are gray images that can be represented by *2D pixel arrays*, e.g., MNIST images

In practice, we have *multi-channel inputs*; for instance, *RGB images* have *three channels*: an $N \times M$ color image is stored in the form of three $N \times M$ matrices, one storing *red map*, another *green map*, and the other *blue map*

Convolution: Multi-Channel Input

What we discussed up to now holds for *single-channel images*: these are gray images that can be represented by *2D pixel arrays*, e.g., MNIST images

In practice, we have *multi-channel inputs*; for instance, *RGB images* have *three channels*: an $N \times M$ color image is stored in the form of three $N \times M$ matrices, one storing *red map*, another *green map*, and the other *blue map*



Recap: 3D Tensors

We can think of multi-channel input as a **tensor of order 3**, i.e., a 3D array

Reminder: Tensor of Order 3

Tensor $\mathbf{X} \in \mathbb{R}^{C \times N \times M}$ is a collection of **C matrices** each of size $N \times M$

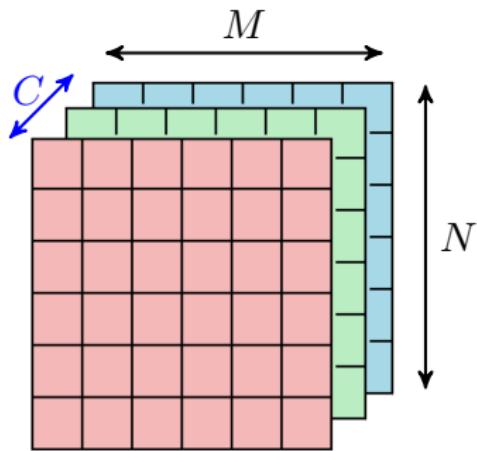
Recap: 3D Tensors

We can think of multi-channel input as a **tensor of order 3**, i.e., a 3D array

Reminder: Tensor of Order 3

Tensor $\mathbf{X} \in \mathbb{R}^{C \times N \times M}$ is a collection of **C matrices** each of size $N \times M$

For instance, an $N \times M$ pixel **RGB** image is tensor in $\mathbb{R}^{3 \times N \times M}$



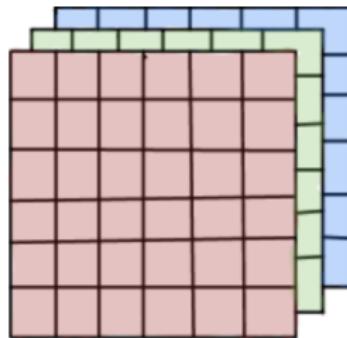
Convolution: 3D Arrays

- + *How can we extend convolution to these 3D tensors?*
- We can look at 3D tensors as **stack** of 2D arrays

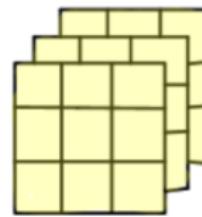
Convolution: 3D Arrays

- + How can we extend convolution to these 3D tensors?
- We can look at 3D tensors as stack of 2D arrays

Let's try a visual example: we want to convolve RGB image with filter \mathbf{W}



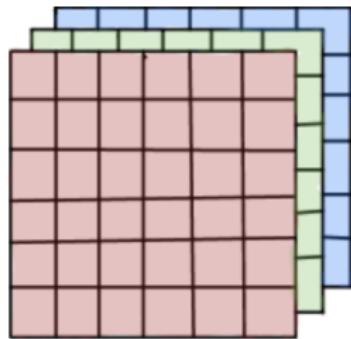
convolve with



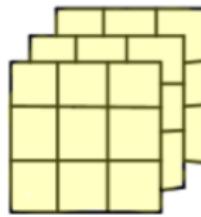
Convolution: 3D Arrays

- + How can we extend convolution to these 3D tensors?
- We can look at 3D tensors as stack of 2D arrays

Let's try a visual example: we want to convolve RGB image with filter \mathbf{W}



convolve with



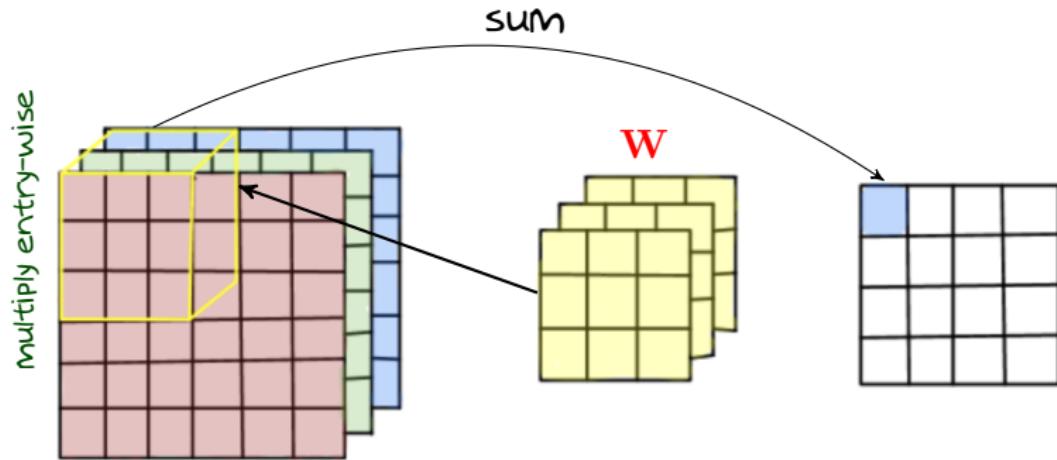
Filter should have the same number of channels, i.e., $\mathbf{W} \in \mathbb{R}^{3 \times F \times F}$

Convolution: General Form

We convolve every channel of **filter** with corresponding channel of **input**, and then *sum them up*

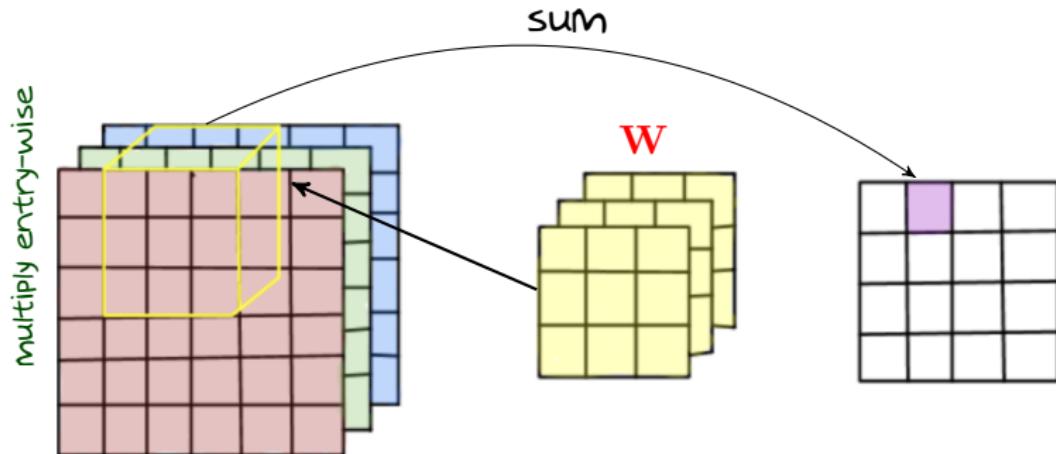
Convolution: General Form

We convolve every channel of **filter** with corresponding channel of **input**, and then **sum them up**



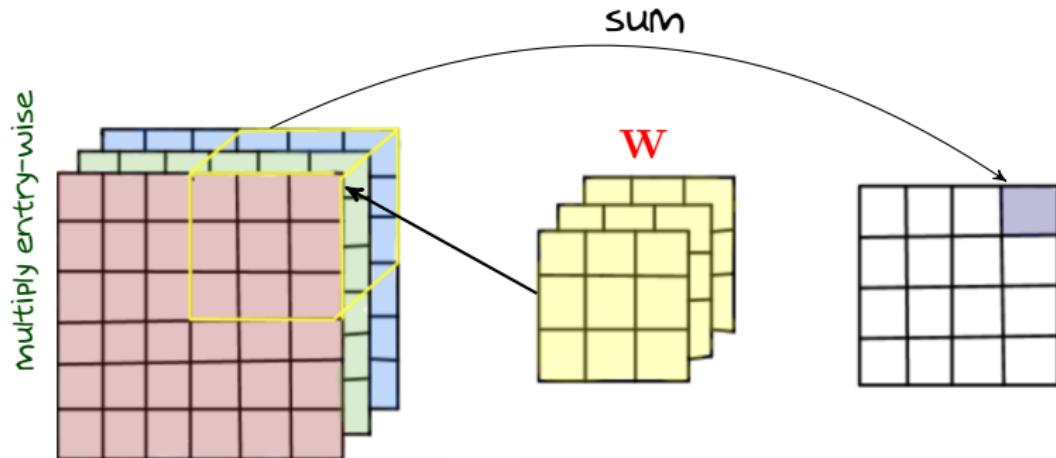
Convolution: General Form

We convolve every channel of **filter** with corresponding channel of **input**, and then *sum them up*



Convolution: General Form

We convolve every channel of **filter** with corresponding channel of **input**, and then *sum them up*



Convolution: 3D Arrays

3D Convolution

Convolution of tensor $\mathbf{X} \in \mathbb{R}^{C \times N \times M}$ by filter/kernel $\mathbf{W} \in \mathbb{R}^{C \times F \times F}$ with stride S is denoted by

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}, S)$$

The matrix \mathbf{Z} is a matrix with $\lfloor (N - F)/S \rfloor + 1$ rows and $\lfloor (M - F)/S \rfloor + 1$ columns and its entry at row i and column j is computed as

$$\mathbf{Z}[i, j] = \text{sum}(\mathbf{W} \odot \mathbf{X}_{i,j})$$

with $\mathbf{X}_{i,j}$ being the corresponding $C \times F \times F$ sub-tensor of \mathbf{X} , i.e.,

$$\mathbf{X}_{i,j} = \mathbf{X}[1:C, 1 + (i-1)S:F + (i-1)S, 1 + (j-1)S:F + (j-1)S]$$

3D Convolution: Summary

As for 2D arrays, we can *apply stride and by resampling*; thus, we write

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W})$$

from now on and keep in mind that

- \mathbf{X} is a tensor with C channels
- \mathbf{W} is a *tensor-like kernel* with C channels: some people say with depth C
- \mathbf{Z} is a matrix, i.e., it has a single channel

3D Convolution: Summary

As for 2D arrays, we can *apply stride and by resampling*; thus, we write

$$\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W})$$

from now on and keep in mind that

- \mathbf{X} is a tensor with C channels
- \mathbf{W} is a *tensor-like kernel* with C channels: some people say with depth C
- \mathbf{Z} is a matrix, i.e., it has a single channel

whenever needed we can

- apply a *fractional stride* by resampling
 - upsampling before convolution
 - downsampling after convolution
- adjust the size of \mathbf{Z} by zero-padding

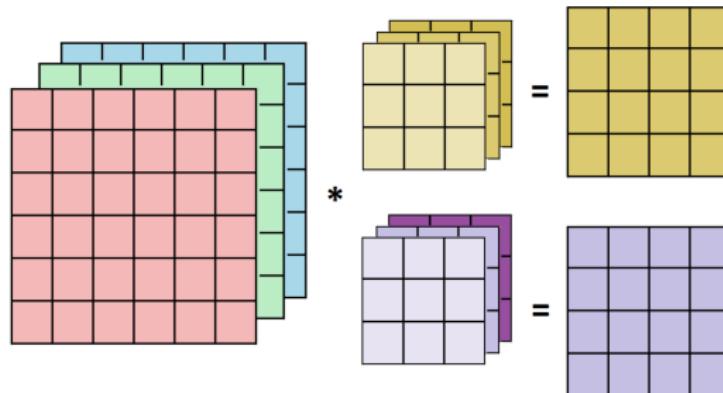
Convolution Layer: General Form

- + But, does it make sense to map a **large tensor** to a **small feature map**?
- This is a great point! This is why we compute **multiple feature maps**

Convolution Layer: General Form

- + But, does it make sense to map a **large tensor** to a **small feature map**?
- This is a great point! This is why we compute **multiple feature maps**

A general convolutional layer has **multiple kernels**: each **kernel** computes a **separate feature map**



Convolution Layer: General Form

Multi-Channel Convolutional Layer

Convolutional layer with C -channel input and K -channel output consists of K filters $\mathbf{W}_1, \dots, \mathbf{W}_K \in \mathbb{R}^{C \times F \times F}$, K biases b_1, \dots, b_K and an activation function $f(\cdot)$.

Convolution Layer: General Form

Multi-Channel Convolutional Layer

Convolutional layer with C -channel input and K -channel output consists of K filters $\mathbf{W}_1, \dots, \mathbf{W}_K \in \mathbb{R}^{C \times F \times F}$, K biases b_1, \dots, b_K and an activation function $f(\cdot)$. It transforms the C -channel input \mathbf{X} to the activated K -channel feature tensor \mathbf{Y} as follows:

- ① it finds the feature tensor \mathbf{Z}

$$\mathbf{Z} = [\text{Conv}(\mathbf{X} | \mathbf{W}_1) + b_1, \dots, \text{Conv}(\mathbf{X} | \mathbf{W}_K) + b_K]$$

Convolution Layer: General Form

Multi-Channel Convolutional Layer

Convolutional layer with C -channel input and K -channel output consists of K filters $\mathbf{W}_1, \dots, \mathbf{W}_K \in \mathbb{R}^{C \times F \times F}$, K biases b_1, \dots, b_K and an activation function $f(\cdot)$. It transforms the C -channel input \mathbf{X} to the activated K -channel feature tensor \mathbf{Y} as follows:

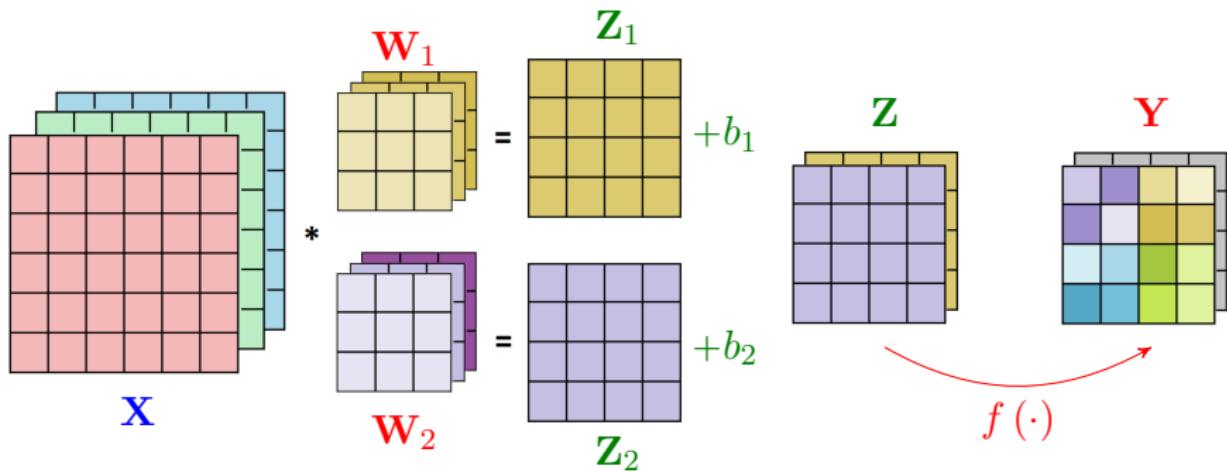
- ① it finds the feature tensor \mathbf{Z}

$$\mathbf{Z} = [\text{Conv}(\mathbf{X} | \mathbf{W}_1) + b_1, \dots, \text{Conv}(\mathbf{X} | \mathbf{W}_K) + b_K]$$

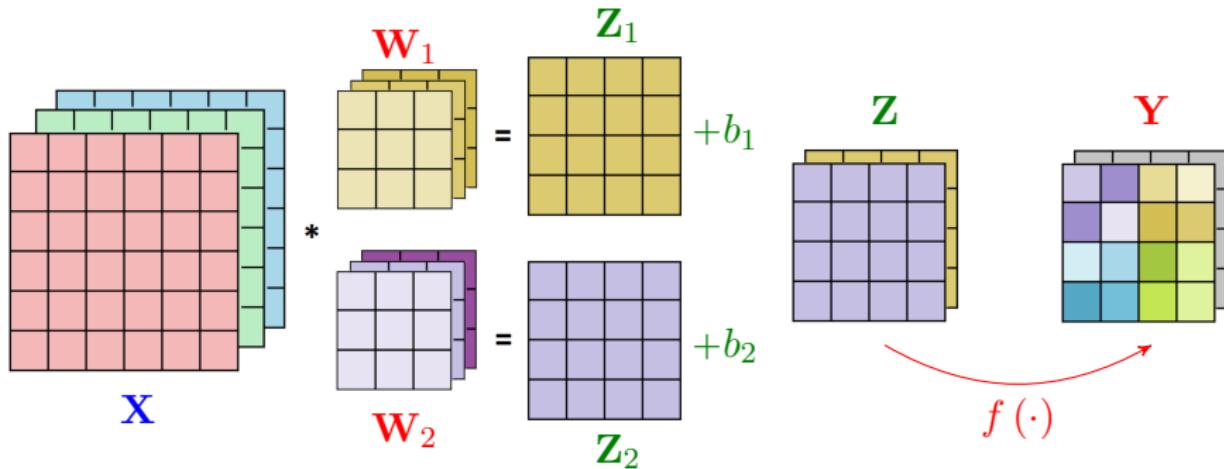
- ② it then activates \mathbf{Z}

$$\mathbf{Y} = f(\mathbf{Z}) \quad f(\cdot) \text{ applied entry-wise}$$

Multi-Channel Convolution Layer: Visualization



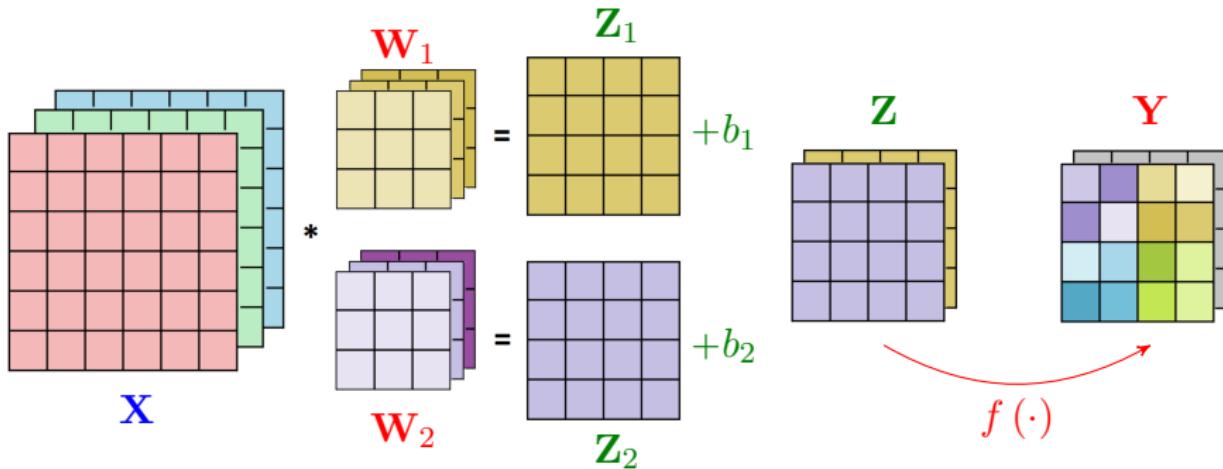
Multi-Channel Convolution Layer: Visualization



There are few points that we should keep in mind

- **Kernels have the same number of channels** (also called **depth**) as **input**

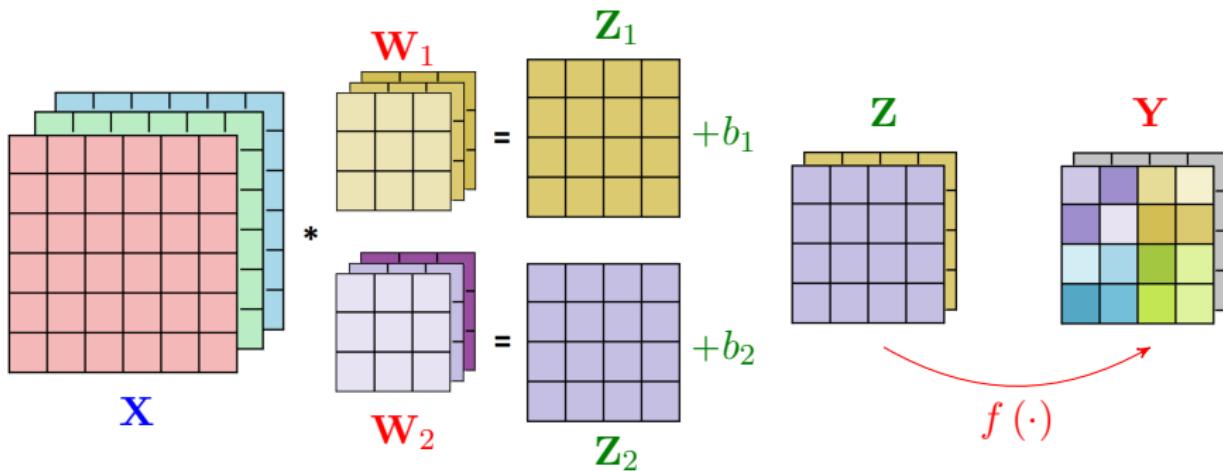
Multi-Channel Convolution Layer: Visualization



There are few points that we should keep in mind

- *Kernels have the same number of channels (also called depth) as input*
- *Number of kernels equals the number of channels in feature tensor*

Multi-Channel Convolution Layer: Visualization

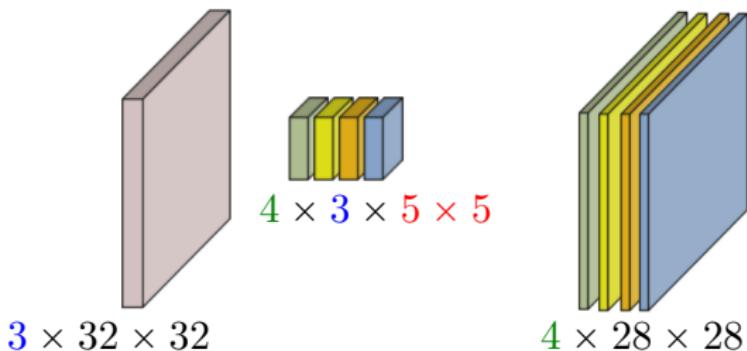


There are few points that we should keep in mind

- *Kernels have the same number of channels (also called depth) as input*
- *Number of kernels equals the number of channels in feature tensor*
- *If X is output of a convolutional layer it may have lots of channels!*
 - ↳ We **should not** think that “*input has at most 3 channels!*”

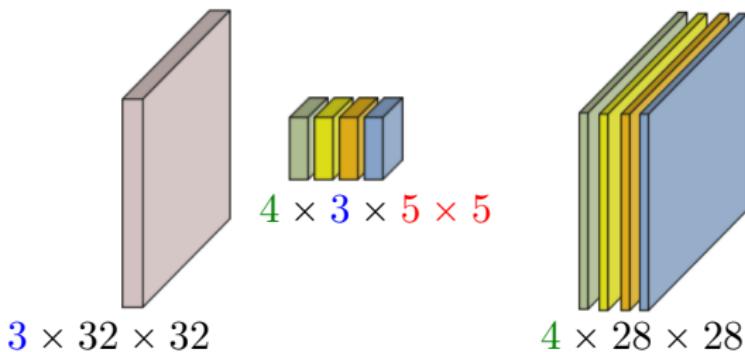
How to Read Dimensions of Convolutional Layers

Many details are dropped as they are readily inferred from architecture



How to Read Dimensions of Convolutional Layers

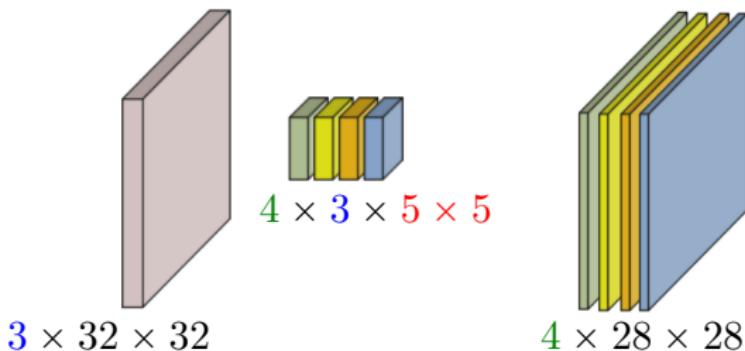
Many details are dropped as they are readily inferred from architecture



In this diagram, we see that $C = 3$ which is the same in **kernels** and **input**

How to Read Dimensions of Convolutional Layers

Many details are dropped as they are readily inferred from architecture

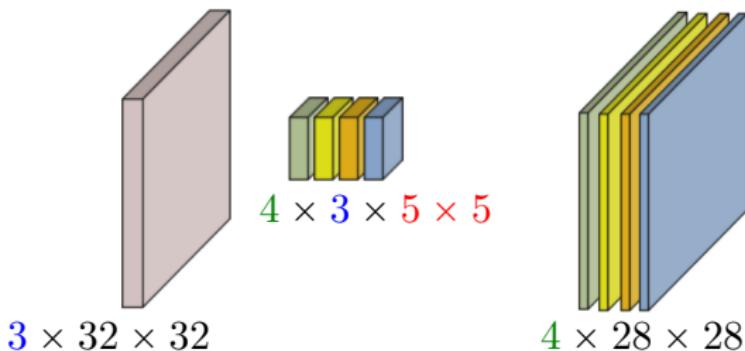


In this diagram, we see that $C = 3$ which is the same in **kernels** and **input**

- We have $K = 3$ **kernels** \leadsto **feature tensor** has 3 channels

How to Read Dimensions of Convolutional Layers

Many details are dropped as they are readily inferred from architecture

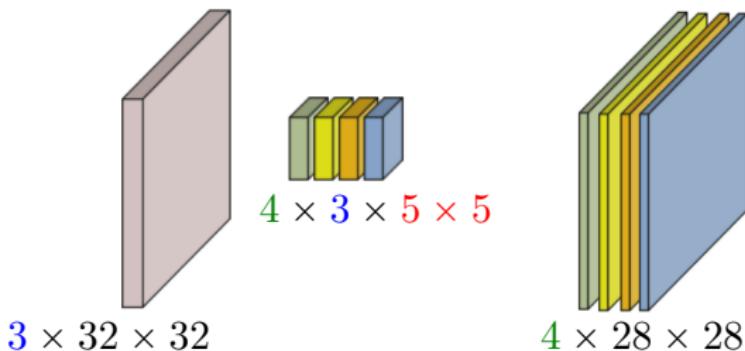


In this diagram, we see that $C = 3$ which is the same in **kernels** and **input**

- We have $K = 3$ **kernels** \leadsto **feature tensor** has 3 channels
- **Kernels** have width $F = 5 \leadsto$ we see that $32 - 5 + 1 = 28$

How to Read Dimensions of Convolutional Layers

Many details are dropped as they are readily inferred from architecture



In this diagram, we see that $C = 3$ which is the same in **kernels** and **input**

- We have $K = 3$ **kernels** \leadsto **feature tensor** has 3 channels
- **Kernels** have width $F = 5 \leadsto$ we see that $32 - 5 + 1 = 28$
 - ↳ **stride** is one, i.e., $S = 1$
 - ↳ **no zero-padding** is applied $P = 0$

Idea of Pooling: Smoothing Filters

The output of convolution can be **jittering** which can come from *spatial correlation*: *Pooling acts as a filter by sliding over the extracted features and pooling out a function of each subpart*

- *Pooling can reduce the jittering behavior*
- *It can mix extracted features and potentially improve shift-invariance*

Idea of Pooling: Smoothing Filters

The output of convolution can be **jittering** which can come from *spatial correlation*: *Pooling acts as a filter by sliding over the extracted features and pooling out a function of each subpart*

- *Pooling can reduce the jittering behavior*
- *It can mix extracted features and potentially improve shift-invariance*

Shift-Invariance

shift-invariance refers to robustness against simple geometric transform of input

Idea of Pooling: Smoothing Filters

The output of convolution can be **jittering** which can come from *spatial correlation*: *Pooling acts as a filter by sliding over the extracted features and pooling out a function of each subpart*

- *Pooling can reduce the jittering behavior*
- *It can mix extracted features and potentially improve shift-invariance*

Shift-Invariance

shift-invariance refers to robustness against simple geometric transform of input

- + How do we do the pooling?
- There are several pooling techniques but popular ones are **max-** and **mean-pooling**

Max-Pooling

For max-pooling, we use a filter of size $L \times L$ and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ \mathbf{Y}_{1,1} & Y_{2,2} & & & Y_{2,M} \\ Y_{2,1} & Y_{2,2} & & & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\text{max } \{Y_{1,1}\}}$$

In each window, we pool the **maximum**

$$\hat{\mathbf{Y}} = \left[\quad \right]$$

Max-Pooling

For max-pooling, we use a filter of size $L \times L$ and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ \mathbf{Y}_{1,1} & Y_{2,2} & & & Y_{2,M} \\ Y_{2,1} & Y_{2,2} & & & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\text{max } \{Y_{1,1}\}}$$

In each window, we pool the **maximum**

$$\hat{\mathbf{Y}} = \left[\hat{Y}_{1,1} \right]$$

Max-Pooling

For max-pooling, we use a filter of size $L \times L$ and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & Y_{N,M} \end{bmatrix} \quad \max \{Y_{1,2}\}$$

In each window, we pool the **maximum**

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} \end{bmatrix}$$

Max-Pooling

For max-pooling, we use a filter of size $L \times L$ and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \quad \max \{\mathbf{Y}_{i,j}\}$$

In each window, we pool the **maximum**

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Max-Pooling

For max-pooling, we use a filter of size $L \times L$ and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \quad \max \{\mathbf{Y}_{i,j}\}$$

In each window, we pool the **maximum**

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Attention

In **max-pooling**, we also track **index of maximizer**: we need it in **backpropagation**

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

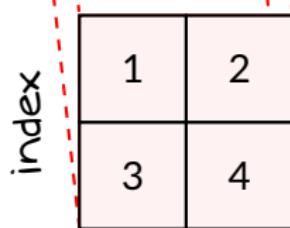
3.2			

index

4			

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



3.2	3.2		

index

A 2x2 submatrix representing the output of the max pooling operation. The top-left cell contains index 4, the top-right cell contains index 3, the bottom-left cell is empty, and the bottom-right cell is empty.

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

3.2	3.2	2.1	

index

4	3	3	

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

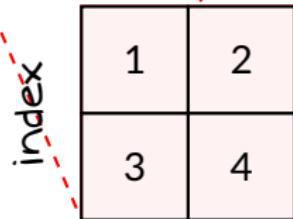
3.2	3.2	2.1	1.5

index

4	3	3	3

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



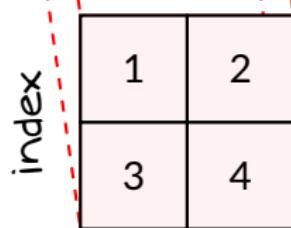
index

3.2	3.2	2.1	1.5
3.2			

4	3	3	3
2			

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



3.2	3.2	2.1	1.5
3.2	3.8		

index

4	3	3	3
2	4		

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

3.2	3.2	2.1	1.5
3.2	3.8	3.8	

index

4	3	3	3
2	4	3	

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3

index

4	3	3	3
2	4	3	4

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

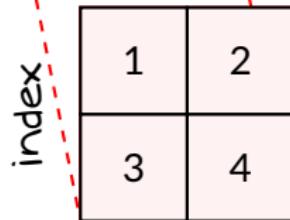
3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3			

index

4	3	3	3
2	4	3	4
1			

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8		

index

4	3	3	3
2	4	3	4
1	2		

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8	3.8	

index

4	3	3	3
2	4	3	4
1	2	1	

Max Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

index

1	2
3	4

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8	3.8	4.3

index

4	3	3	3
2	4	3	4
1	2	1	2

Mean-Pooling

In mean-pooling, we again use an $L \times L$ filter and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & & & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\text{mean } \{Y_{1,1}\}}$$

In each window, we pool the *average*

$$\hat{\mathbf{Y}} = \left[\quad \right]$$

Mean-Pooling

In mean-pooling, we again use an $L \times L$ filter and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & & & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix}$$

mean $\{\mathbf{Y}_{1,1}\}$

In each window, we pool the *average*

$$\hat{\mathbf{Y}} = \left[\hat{Y}_{1,1} \right]$$

Mean-Pooling

In mean-pooling, we again use an $L \times L$ filter and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & Y_{N,M} \end{bmatrix}$$

mean $\{\mathbf{Y}_{1,2}\}$

In each window, we pool the *average*

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} \end{bmatrix}$$

Mean-Pooling

In mean-pooling, we again use an $L \times L$ filter and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \quad \text{mean } \{\mathbf{Y}_{i,j}\}$$

In each window, we pool the *average*

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Mean-Pooling

In mean-pooling, we again use an $L \times L$ filter and slide over the feature map

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \quad \text{mean } \{\mathbf{Y}_{i,j}\}$$

In each window, we pool the *average*

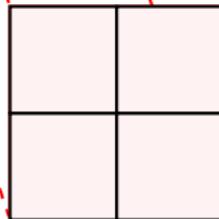
$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

We can look at mean-pooling as a *convolution* with *uniform kernel*

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

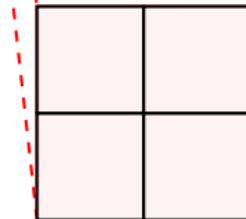
2.05			



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

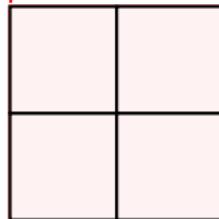
2.05	1.9		



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

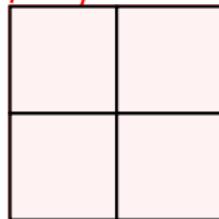
2.05	1.9	1.225	



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

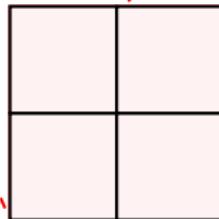
2.05	1.9	1.225	0.7



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

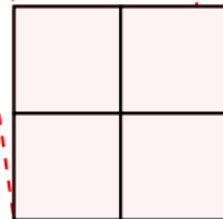
2.05	1.9	1.225	0.7
1.85			



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

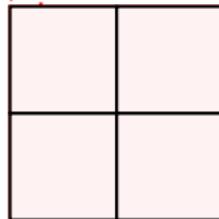
2.05	1.9	1.225	0.7
1.85	2.275		



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

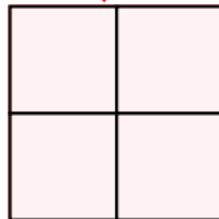
2.05	1.9	1.225	0.7
1.85	2.275	1.85	



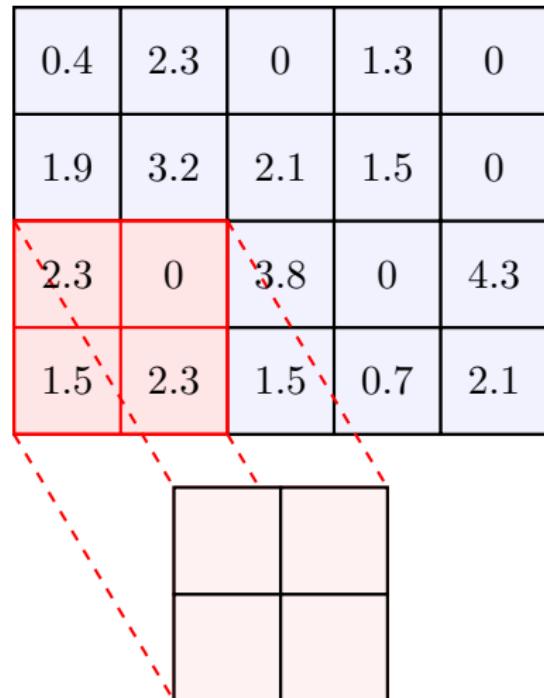
Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45



Mean-Pooling: Numerical Example

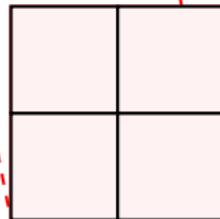


2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525			

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

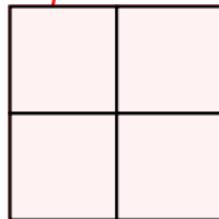
2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9		



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

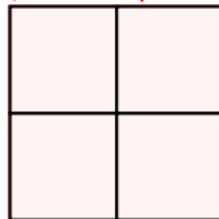
2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	1.775



Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05			

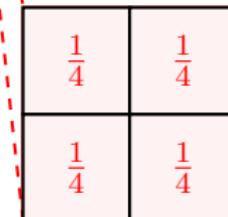


mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9		



mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	

$$\begin{array}{|c|c|} \hline \frac{1}{4} & \frac{1}{4} \\ \hline \frac{1}{4} & \frac{1}{4} \\ \hline \end{array}$$

mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7

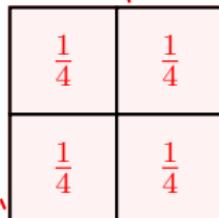
$$\begin{array}{|c|c|}\hline \frac{1}{4} & \frac{1}{4} \\ \hline \frac{1}{4} & \frac{1}{4} \\ \hline\end{array}$$

mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85			

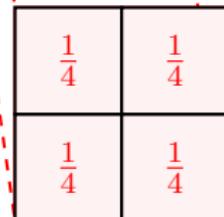


mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275		

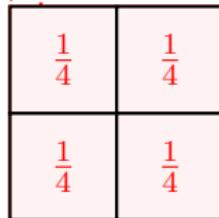


mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	



mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

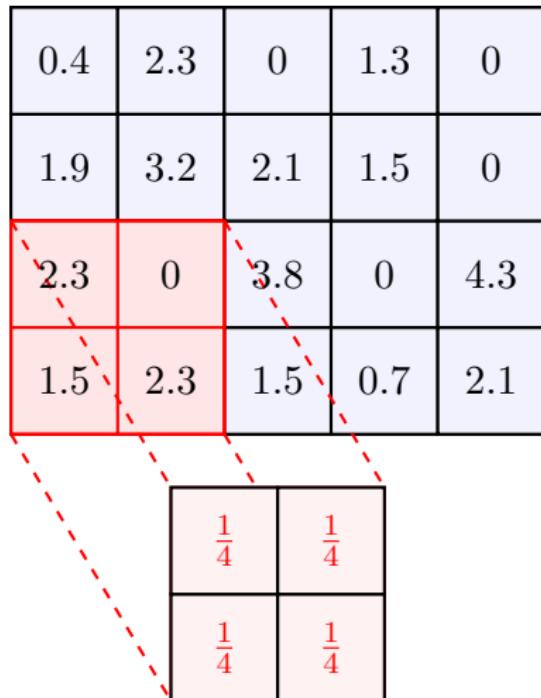
0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45

$$\begin{array}{|c|c|} \hline \frac{1}{4} & \frac{1}{4} \\ \hline \frac{1}{4} & \frac{1}{4} \\ \hline \end{array}$$

mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example



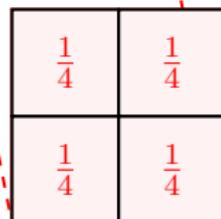
2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525			

mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9		



mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	

$$\begin{matrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{matrix}$$

mean-pooling \equiv convolution with uniform kernel

Mean-Pooling: Numerical Example

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	1.775

$$\begin{matrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{matrix}$$

mean-pooling \equiv convolution with uniform kernel

Advanced Pooling: Use a General Function

We could in general replace **max or mean operator** with a general function

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & & & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\Pi(\cdot) : \mathbb{R}^{L \times L} \mapsto \mathbb{R}}$$

Typical functions are **norms**, e.g., ℓ_2 -norm

$$\hat{\mathbf{Y}} = \left[\quad \right]$$

We can even replace it with a **small NN!**

Advanced Pooling: Use a General Function

We could in general replace **max or mean operator** with a general function

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & & & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\Pi(\cdot) : \mathbb{R}^{L \times L} \mapsto \mathbb{R}}$$

Typical functions are **norms**, e.g., ℓ_2 -norm

$$\hat{\mathbf{Y}} = \left[\hat{Y}_{1,1} \right]$$

We can even replace it with a **small NN!**

Advanced Pooling: Use a General Function

We could in general replace max or mean operator with a general function

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & Y_{N,M} \end{bmatrix} \xrightarrow{\Pi(\cdot) : \mathbb{R}^{L \times L} \mapsto \mathbb{R}}$$

Typical functions are norms, e.g., ℓ_2 -norm

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} \end{bmatrix}$$

We can even replace it with a small NN!

Advanced Pooling: Use a General Function

We could in general replace **max or mean operator** with a general function

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & \dots & Y_{1,M} \\ Y_{2,1} & Y_{2,2} & \dots & \dots & Y_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Y_{N,1} & Y_{N,2} & \dots & \dots & Y_{N,M} \end{bmatrix} \quad \Pi(\cdot) : \mathbb{R}^{L \times L} \mapsto \mathbb{R}$$

Typical functions are **norms**, e.g., ℓ_2 -norm

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{Y}_{1,1} & \hat{Y}_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

We can even replace it with a **small NN!**

Pooling with Stride

Similar to convolution, we *can apply pooling with stride*

Pooling with Stride

Similar to convolution, we can apply *pooling* with *stride*

- We usually do not use *fractional stride* with *pooling*
 - ↳ Upsampling the input, only adds zeros
 - ↳ Extra zeros usually do not make any gain: think of max-pooling for instance

Pooling with Stride

Similar to convolution, we can apply *pooling* with *stride*

- We usually do not use *fractional stride* with *pooling*
 - ↳ Upsampling the input, only adds zeros
 - ↳ Extra zeros usually do not make any gain: think of max-pooling for instance
- Integer *stride* can be seen as downsampling
 - ↳ We apply pooling with *unit stride*
 - ↳ We down-sample output with *sampling factor = stride*

Pooling with Stride

Similar to convolution, we can apply *pooling* with *stride*

- We usually do not use *fractional stride* with *pooling*
 - ↳ Upsampling the input, only adds zeros
 - ↳ Extra zeros usually do not make any gain: think of max-pooling for instance
- Integer *stride* can be seen as downsampling
 - ↳ We apply pooling with *unit stride*
 - ↳ We down-sample output with *sampling factor* = *stride*
- In practice, we often leave integer strides for *pooling* layer
 - ↳ We apply convolution with *unit stride*
 - ↳ If we need to down-sample, we do it later in the *pooling* layer

Pooling: Few Remarks

- + *Do we always pool after convolution?*
- Not really! In many architectures pooling is applied every couple of convolutional layers

Pooling: Few Remarks

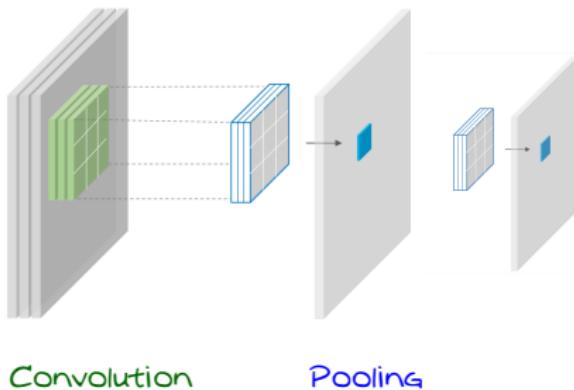
- + Do we always pool after convolution?
- Not really! In many architectures pooling is applied every couple of convolutional layers

It is worth noting that

- In many architectures **pooling** is applied **every couple of convolutional layers**
 - ↳ We apply **multiple** convolutional layers
 - ↳ We then apply a **pooling** layer **with stride**
- Most **poolings** have no **learnable parameters**; thus, **they are cheap**
 - ↳ **Max** and **mean-pooling** have **no weights**
- **Filter size for pooling** can be different from the **convolutional layers**
 - ↳ They are typically in **the same range**

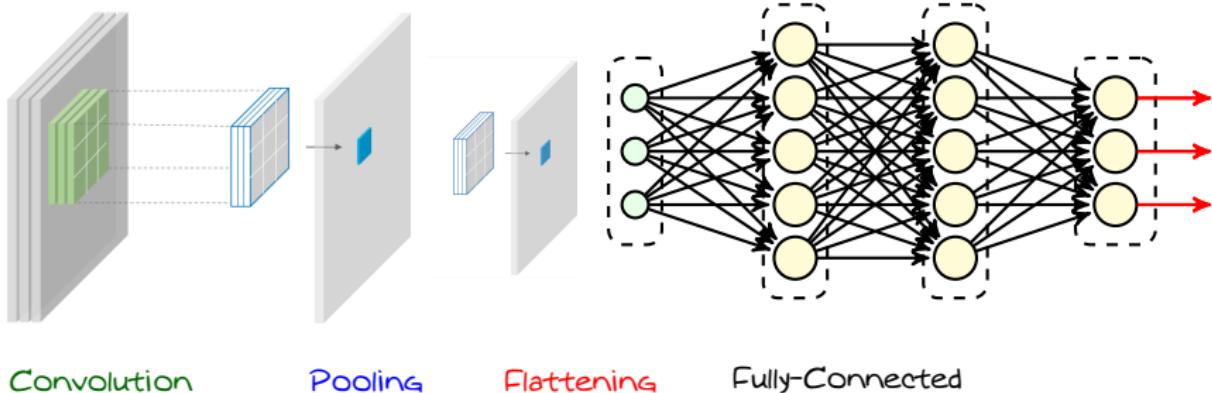
Output FNN: *Flattening*

Let's recall our simple CNN



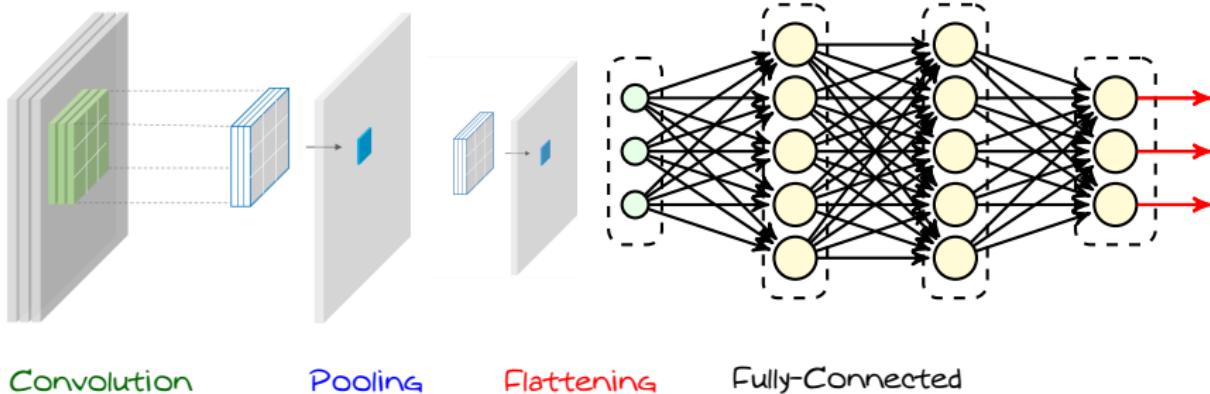
Output FNN: Flattening

Let's recall our simple CNN



Output FNN: Flattening

Let's recall our simple CNN

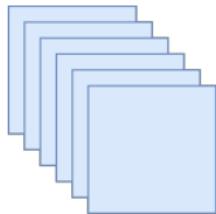


Once we are over with convolution and pooling we **flatten** final **feature tensor**

Flattening

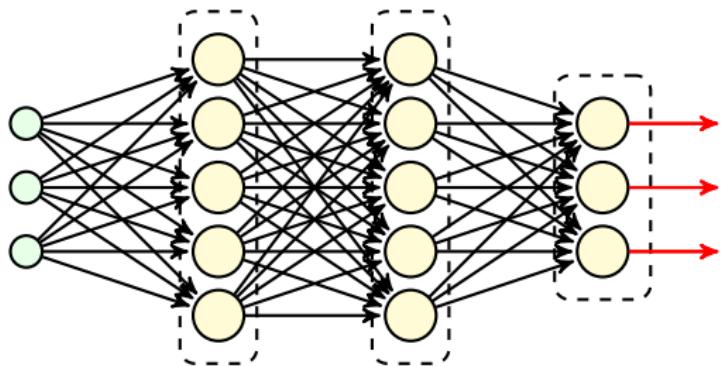
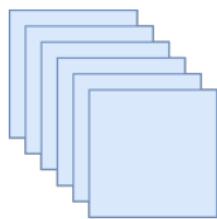
In **flattening**, we sort all entries of the **feature tensor** into a vector

Flattening



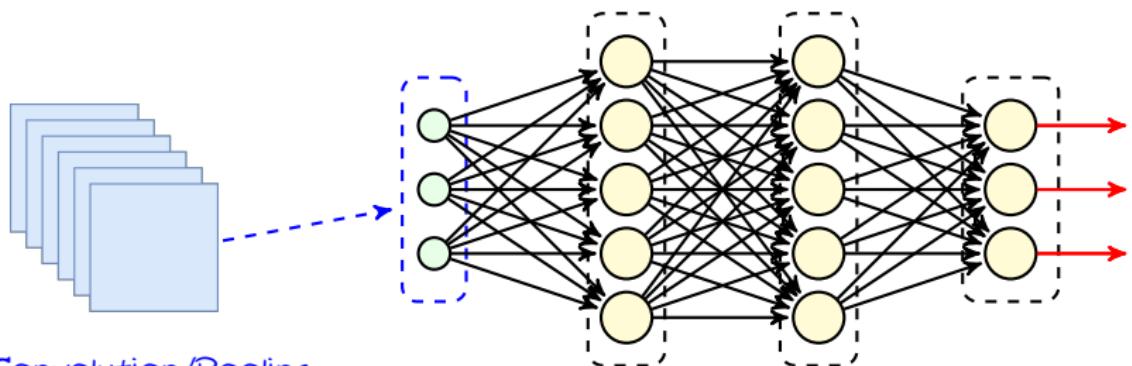
Last Convolution/Pooling

Flattening



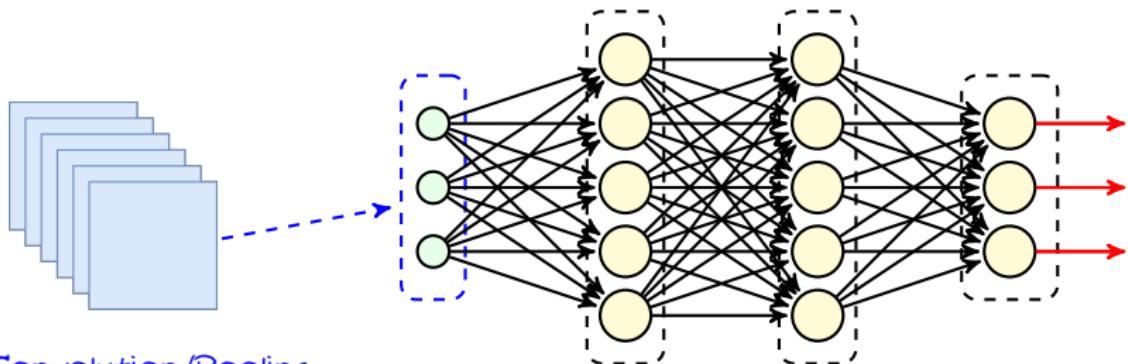
Last Convolution/Pooling

Flattening



Last Convolution/Pooling

Flattening

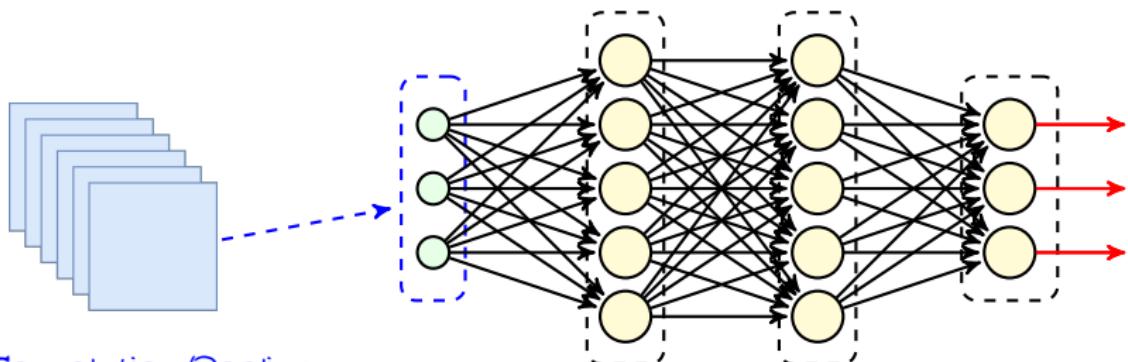


Last Convolution/Pooling

Say we have a **feature tensor** with K channels

↳ each channel is an $N \times M$ map after last **convolution** or **pooling**

Flattening



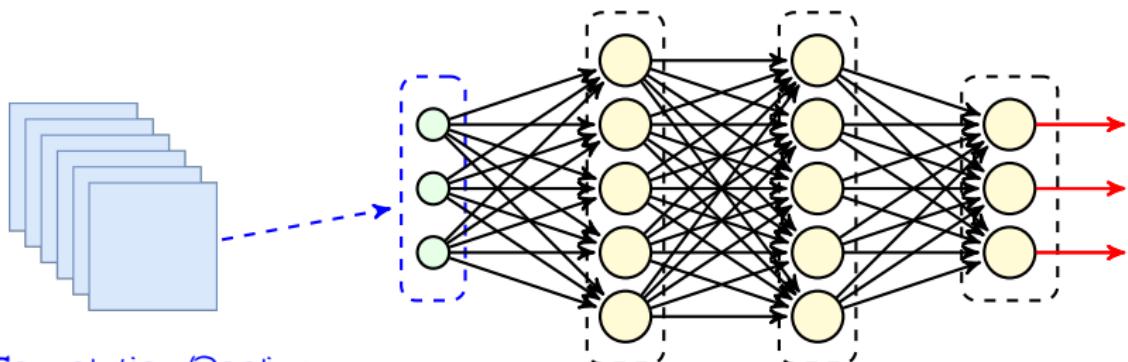
Last Convolution/Pooling

Say we have a **feature tensor** with K channels

↳ each channel is an $N \times M$ map after last **convolution** or **pooling**

We then have an input to the FNN with NMK entries

Flattening



Last Convolution/Pooling

Say we have a **feature tensor** with K channels

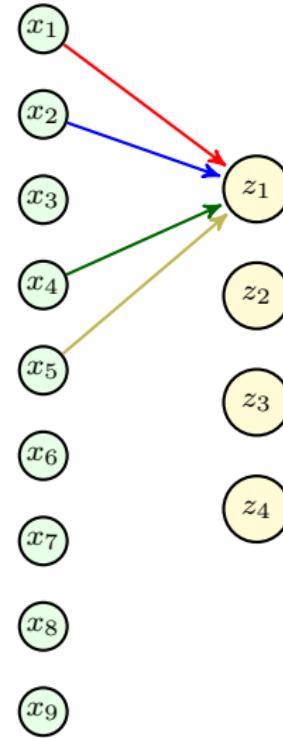
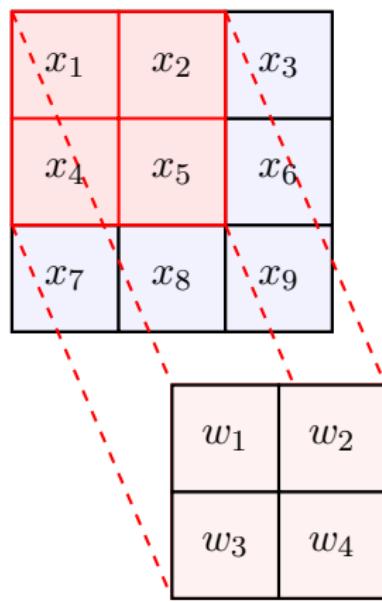
↳ each channel is an $N \times M$ map after last **convolution** or **pooling**

We then have an input to the FNN with NMK entries

After **flattening** everything goes as before through the **fully-connected FNN**

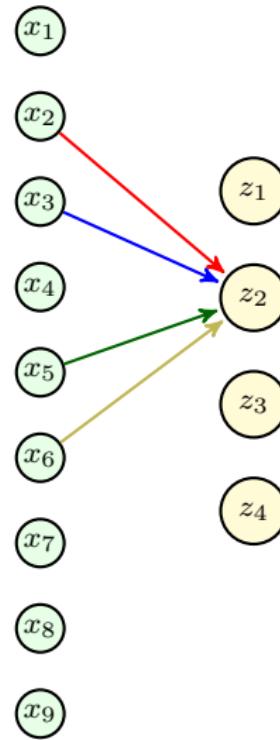
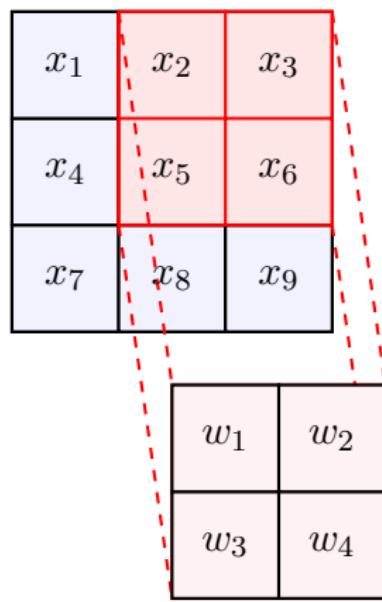
Convolutional Layer as Set of Neurons

We can look at a *convolutional* layer as a *layer of neurons*



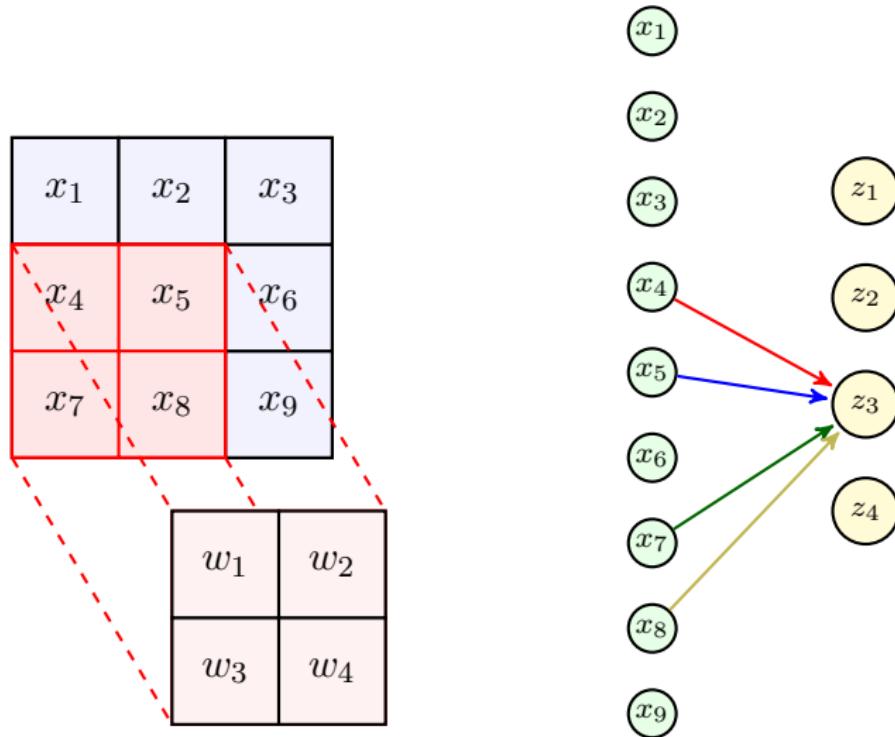
Convolutional Layer as Set of Neurons

We can look at a **convolutional** layer as a **layer of neurons**



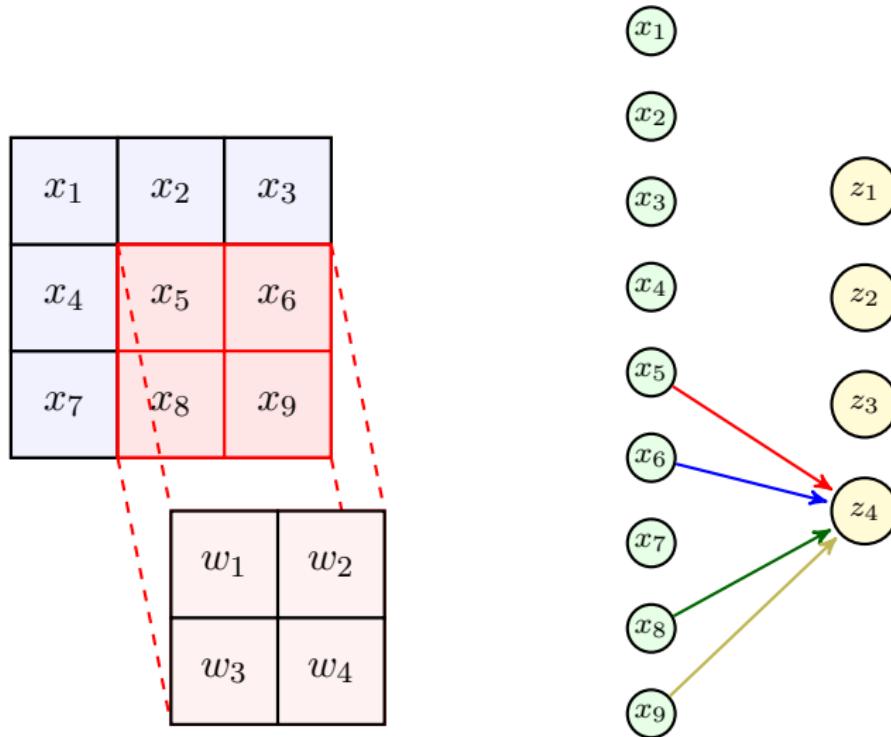
Convolutional Layer as Set of Neurons

We can look at a **convolutional** layer as a **layer of neurons**



Convolutional Layer as Set of Neurons

We can look at a *convolutional* layer as a *layer of neurons*

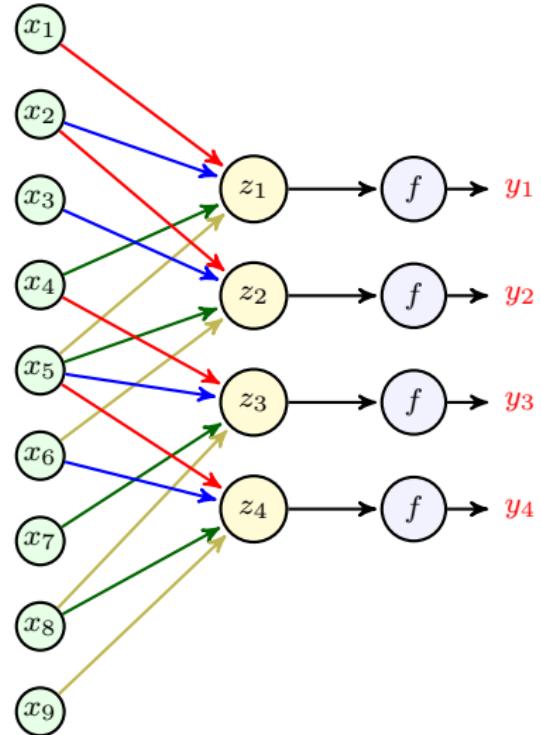


Convolutional Layer as Set of Neurons

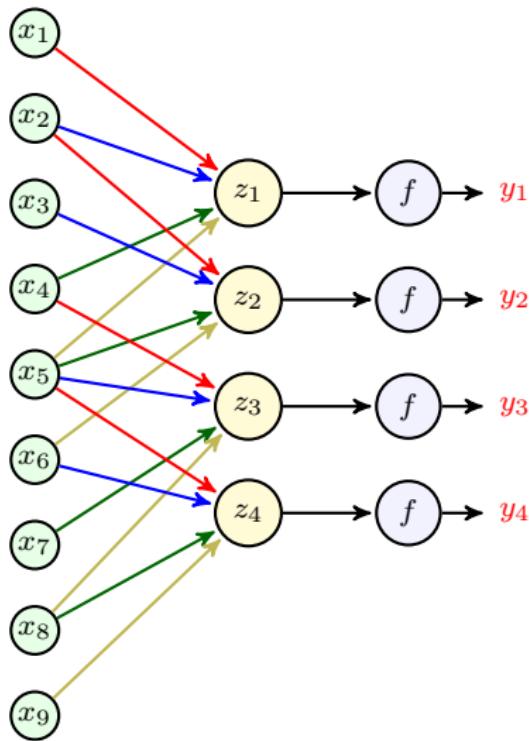
We can look at a **convolutional** layer as a **layer of neurons**

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

w_1	w_2
w_3	w_4



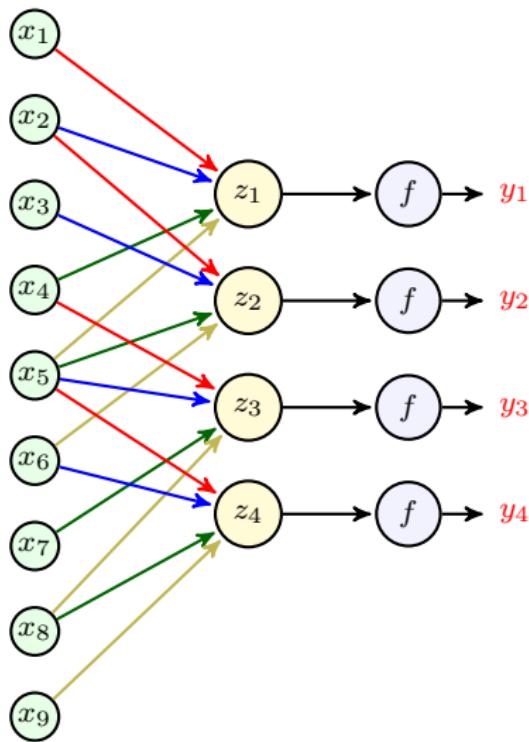
Convolutional Layer as Set of Neurons



In this viewpoint

- ↳ each **feature** is output of a neuron
- ↳ neurons **share** same weights and bias
- ↳ neurons are **activated** by $f(\cdot)$

Convolutional Layer as Set of Neurons



In this viewpoint

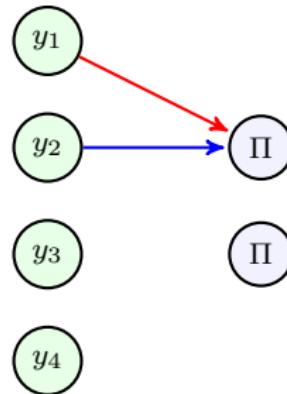
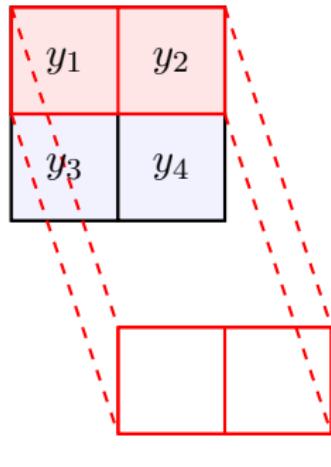
- ↳ each **feature** is output of a neuron
- ↳ neurons **share** same weights and bias
- ↳ neurons are **activated by $f(\cdot)$**

This is however not a **fully-connected** layer

- ↳ it is **locally-connected**
- ↳ neurons have **shared parameters**

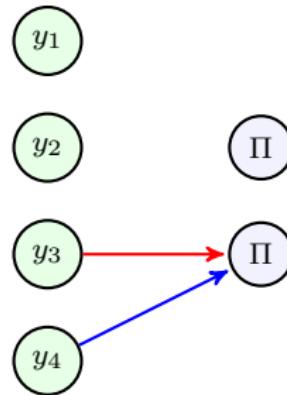
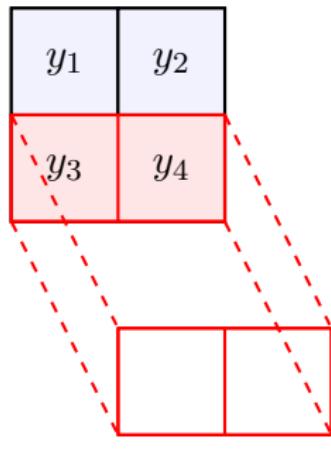
Pooling Layer as Set of Neurons

We can extend our *viewpoint* to *pooling layers*



Pooling Layer as Set of Neurons

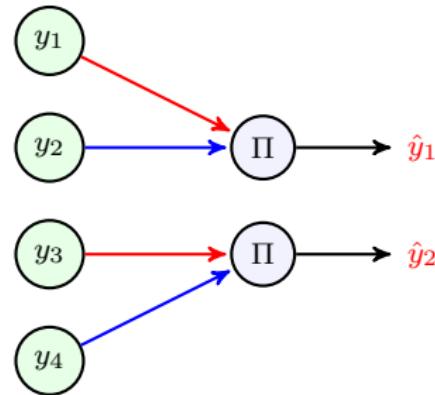
We can extend our *viewpoint* to *pooling layers*



Pooling Layer as Set of Neurons

We can extend our *viewpoint* to *pooling layers*

y_1	y_2
y_3	y_4



Pooling Layer as Set of Neurons

We can extend our **viewpoint** to **pooling layers**



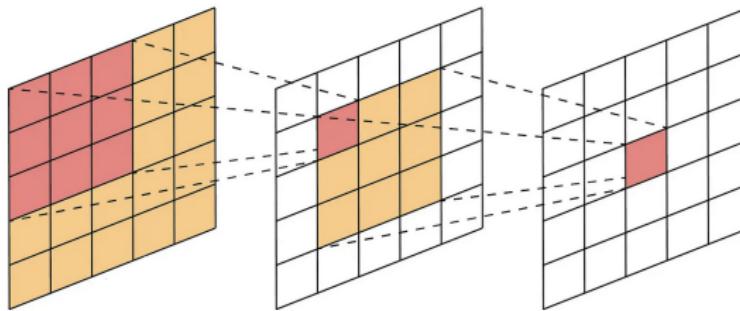
Similar to convolutional layer, pooling layer is a **feedforward layer** with
local connectivity and **shared** parameters

We usually have predefined parameters, i.e., **no weights to be learned**

Going Deep: Receptive Field

As we go **deep** in CNN, we mix **features**: *features in deeper layers depend on more pixels of input tensor. We often say*

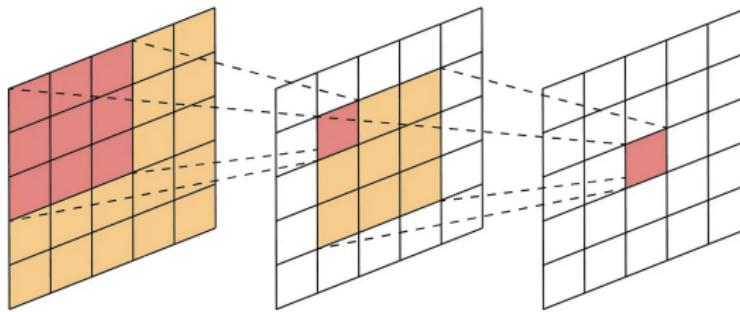
*deeper layers have larger **receptive field***



Going Deep: Receptive Field

As we go **deep** in CNN, we mix **features**: *features in deeper layers depend on more pixels of input tensor. We often say*

*deeper layers have larger **receptive field***



Receptive Field

The input region that each neuron in a given convolutional layer responds to

Receptive Field

Few points we may note regarding the **receptive field**

- ↳ For first layer the receptive field is simply *where the filter screen*
- ↳ For *deeper* layers, actual receptive field is **not** immediately obvious and must be calculated

Receptive Field

Few points we may note regarding the **receptive field**

- ↳ For first layer the receptive field is simply *where the filter screen*
- ↳ For *deeper* layers, actual receptive field is **not** immediately obvious and must be calculated

This is however obvious that

- ↳ *Receptive field increases as we go deeper*

Receptive Field

Few points we may note regarding the **receptive field**

- ↳ For first layer the receptive field is simply *where the filter screen*
- ↳ For *deeper* layers, actual receptive field is *not* immediately obvious and *must be calculated*

This is however obvious that

- ↳ *Receptive field increases* as we go *deeper*

Receptive field also depends on the filter sizes

- ↳ Large filters *increase receptive field* per layer
- ↳ Small filters *reduce receptive field* per layer

Receptive Field

Few points we may note regarding the **receptive field**

- ↳ For first layer the receptive field is simply *where the filter screen*
- ↳ For *deeper* layers, actual receptive field is *not* immediately obvious and *must be calculated*

This is however obvious that

- ↳ *Receptive field increases* as we go *deeper*

Receptive field also depends on the filter sizes

- ↳ Large filters *increase receptive field* per layer
- ↳ Small filters *reduce receptive field* per layer

-
- + Why do we define **receptive field**? Does it have any particular meaning?!
 - It helps very much building intuition, especially as we go *deep*

Intuition on Features via Receptive Field

To design a **deep** CNN, we need to first *build a bit of more intuition*

Intuition on Features via Receptive Field

To design a **deep** CNN, we need to first *build a bit of more intuition*

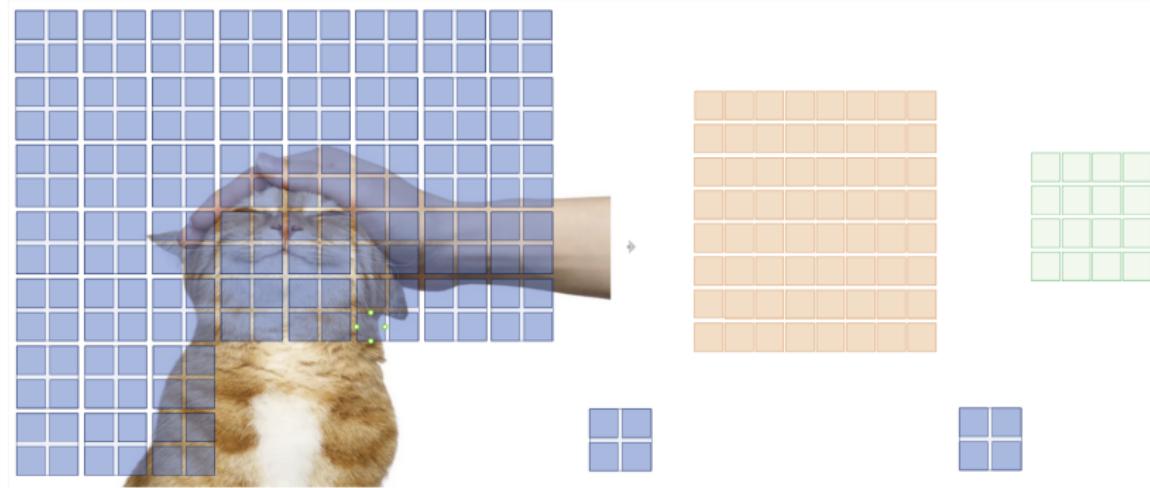
Let's make a closer look: *say we want to classify the following image*



We use two **convolutional** + **pooling** layers cascaded both with 2×2 filters

Intuition on Features via Receptive Field

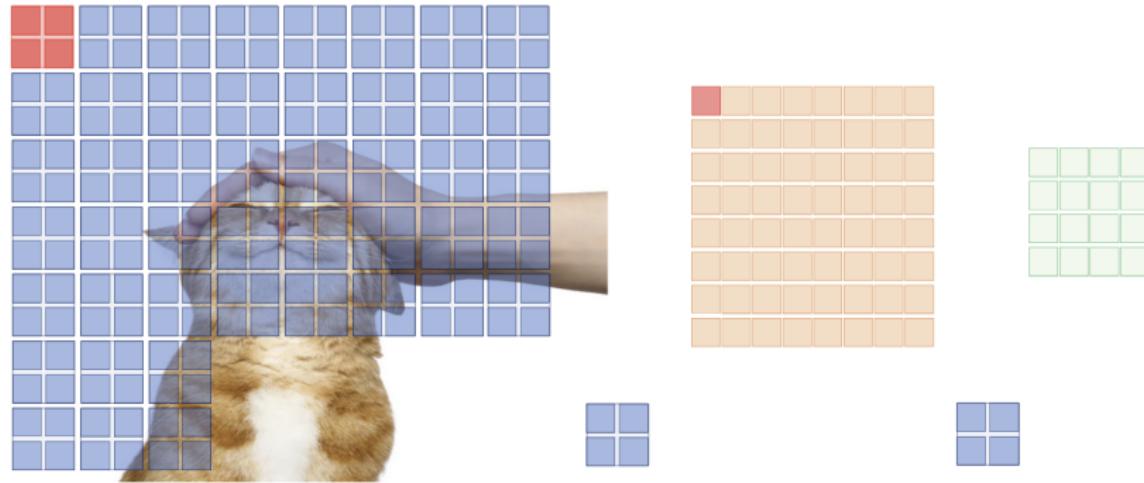
The first layer extracts features of small region of input image



Say for instance, we look at the very first feature in the orange feature map

Intuition on Features via Receptive Field

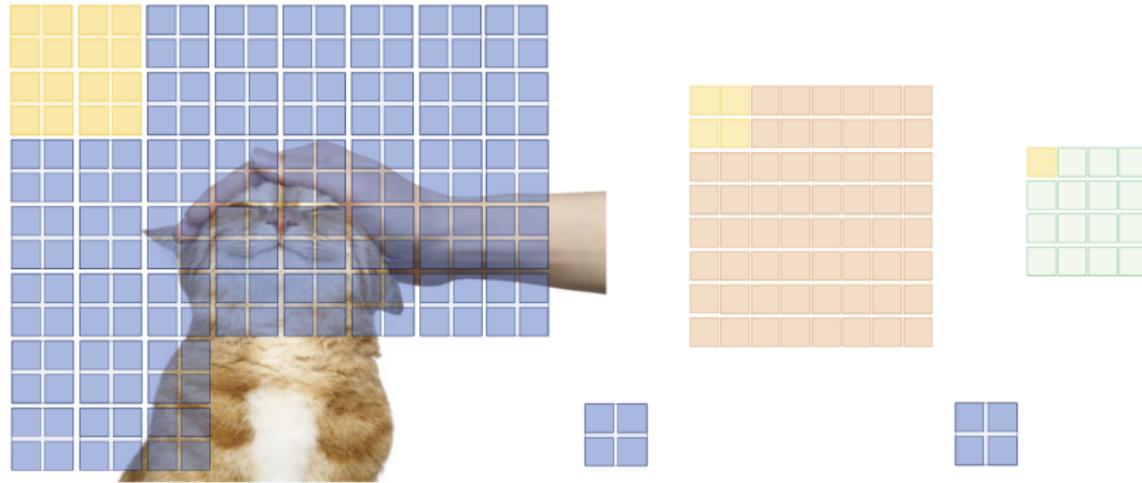
This **red feature** is calculated from input pixels at upper left corner



We could intuitively say that the **neuron** corresponding to **this feature** is looking **locally** at the upper left corner to extract the features of this region

Intuition on Features via Receptive Field

Now, let's look at the very first feature in the **green feature map**



This **neuron** is looking at a **larger region** of input image, but it extracts the features of this region through the **features** extracted via the **orange feature map**

General Perspective on Deep CNNs

The given example illustrates an *intuitive interpretation of deep* CNNs

- ↳ Deep CNNs extract *gradually* the *features of input*
- ↳ In first layer they extract features of *smaller regions* of the input
- ↳ They *gradually expand* this region as they go *deeper*
 - ↳ *deeper* layers look at *high-level features*

General Perspective on Deep CNNs

The given example illustrates an *intuitive interpretation of deep* CNNs

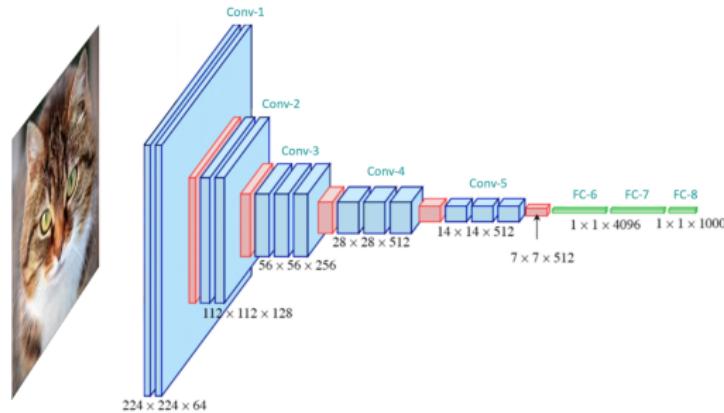
- ↳ Deep CNNs extract *gradually* the *features of input*
- ↳ In first layer they extract features of *smaller regions* of the input
- ↳ They *gradually expand* this region as they go *deeper*
 - ↳ *deeper* layers look at *high-level features*

This gives us a good idea on how a good *deep* CNN looks like

- First layer has *small* filters
 - ↳ *small* filters lead to *large feature maps*
 - ↳ the number of *output channels* is *small*
- As we go *deeper* in the CNN filters get a bit *larger*
 - ↳ *larger* filters return *smaller feature maps*
 - ↳ the number of *output channels* is *increased*

Deep CNN: Example

Recall the example of **VGG-16 architecture** we had a look on

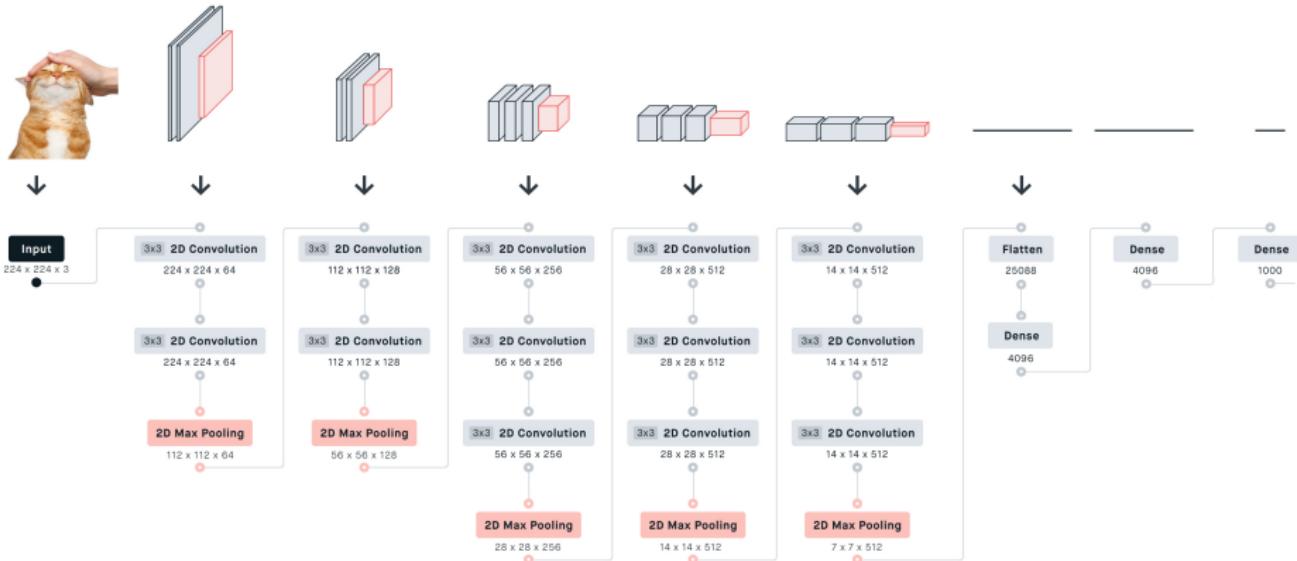


In the original proposal of this architecture, it is said

- **convolutional filters are 3×3 and similar padding is used in all layers**
- **convolution performed with unit stride and max-pooling with stride 2**

Deep CNN: Example

We can now break it down, as we know all the components



Deep CNN: Example

Let's count the number of layers with *learnable parameters*

$$\begin{aligned}\# \text{ layers with weights} &= 2 && \text{first two convolutions} \\ &+ 2 && \text{second two convolutions} \\ &+ 3 && \text{third round of convolutions} \\ &+ 3 && \text{fourth round of convolutions} \\ &+ 3 && \text{fifth round of convolutions} \\ &+ 3 && \text{fully-connected network} \\ &= 16\end{aligned}$$

Deep CNN: Example

Let's count the number of layers with *learnable parameters*

$$\begin{aligned}\# \text{ layers with weights} &= 2 && \text{first two convolutions} \\ &+ 2 && \text{second two convolutions} \\ &+ 3 && \text{third round of convolutions} \\ &+ 3 && \text{fourth round of convolutions} \\ &+ 3 && \text{fifth round of convolutions} \\ &+ 3 && \text{fully-connected network} \\ &= 16\end{aligned}$$

This is actually why it's called VGG-16

Big Picture: What to Train

In CNNs we need to train *all learnable parameters*, i.e.,

- *weights and biases of output FNN*
- *weights in the filters of convolutional layers*
- *if we use advanced pooling with weights; then, we should find them as well*

Big Picture: What to Train

In CNNs we need to train *all learnable parameters*, i.e.,

- *weights and biases* of output FNN
- *weights* in the *filters* of convolutional layers
- if we use *advanced pooling* with *weights*; then, we should find them as well

To train, we follow the same approach

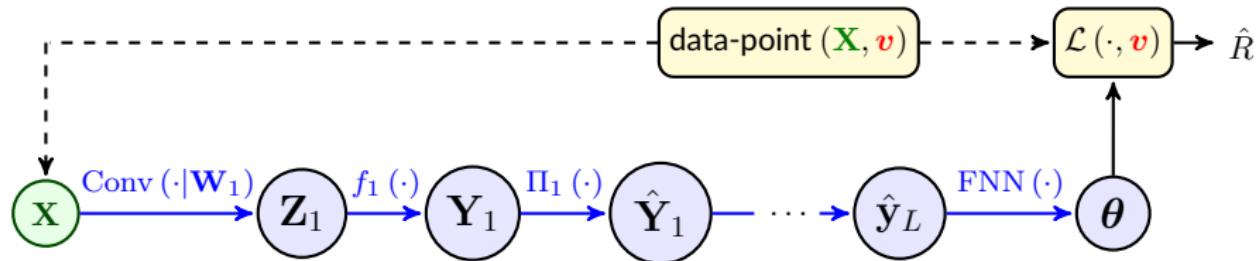
$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \hat{R}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{B} \sum_{b=1}^B \mathcal{L}(\text{CNN}(\mathbf{X}_b | \mathbf{w}), \mathbf{v}_b) \quad (\text{Training})$$

where \mathbf{X}_b is the tensor-type data-point and \mathbf{v}_b is its label

- ↳ We solve this optimization via a *gradient-based method*
- ↳ This means we should be able to pass *forward* and *backward*

Computation Graph

Computation graph for single data-point \mathbf{X} and its true label \mathbf{v} is as follows



For **simplicity**, let's assume that we are doing **basic SGD**

- ↳ We need to pass **forward** \mathbf{X} over this graph to get **loss**
- ↳ We should then pass **backward** to compute **gradient**

Let's try both directions

Forward Pass over CNN

The forward pass is what we learned in the last section

- We pass \mathbf{X} through first convolution to get \mathbf{Z}_1
- We activate \mathbf{Z}_1 to get \mathbf{Y}_1
- We pool entries of \mathbf{Y}_1 and get $\hat{\mathbf{Y}}_1$
- \vdots
- We flatten and pass **forward** through the **output FNN**

Forward Pass over CNN

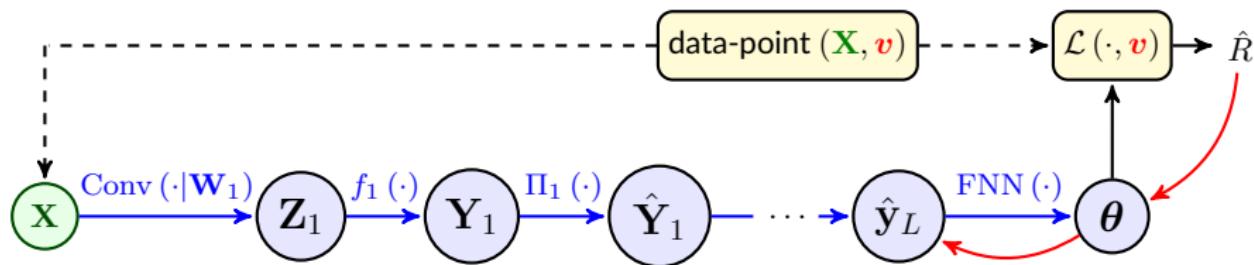
The forward pass is what we learned in the last section

- We pass \mathbf{X} through first convolution to get \mathbf{Z}_1
- We activate \mathbf{Z}_1 to get \mathbf{Y}_1
- We pool entries of \mathbf{Y}_1 and get $\hat{\mathbf{Y}}_1$
- \vdots
- We flatten and pass **forward** through the **output FNN**

Once we are over, we have

- ↳ *all convolution, activated and pooled values in convolutional layers*
- ↳ *all affine and activated values in the FNN*

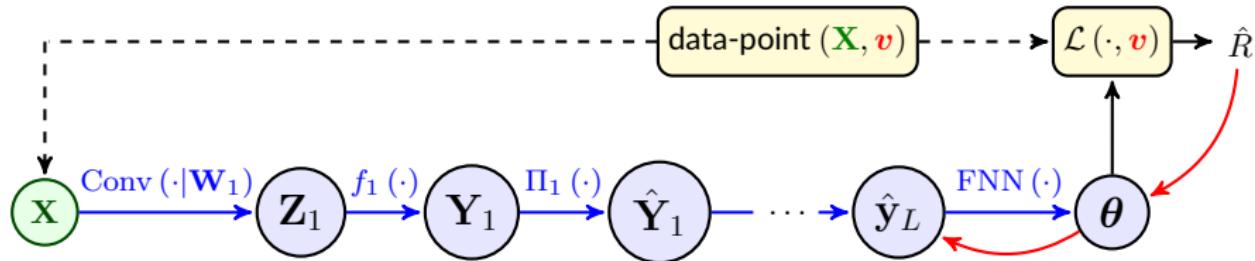
Backward Pass



We now want to **backpropagate**

- ↳ We first compute $\nabla_{\theta} \hat{R}$
- ↳ We then **backpropagate** over the FNN **till** we get to its input

Backward Pass



We now want to **backpropagate**

- ↳ We first compute $\nabla_{\theta} \hat{R}$
- ↳ We then **backpropagate** over the FNN till we get to its input

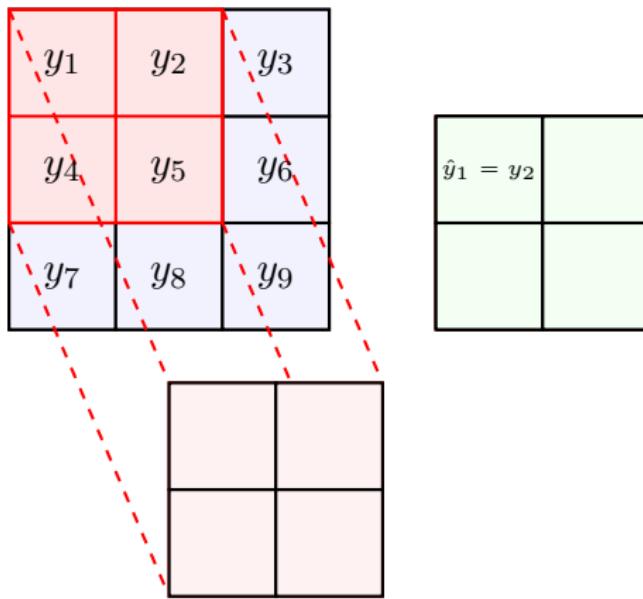
At this point, we sort $\nabla_{\hat{Y}_L} \hat{R}$ in tensor by **reversing the flattening**

we now have $\nabla_{\hat{Y}_L} \hat{R}$

we only need to learn how to **backpropagate** through **pooling** and **convolutional layers** and then we can **complete** the backward pass!

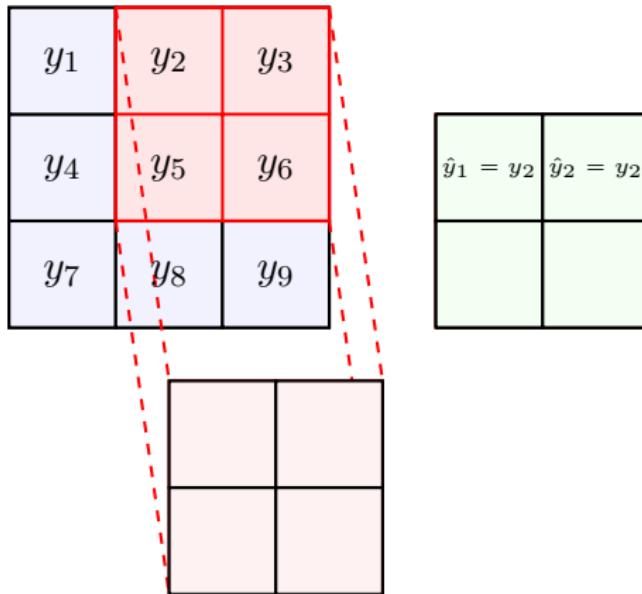
Backpropagate through Pooling: Max-Pooling

Lets start with a **simple example**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



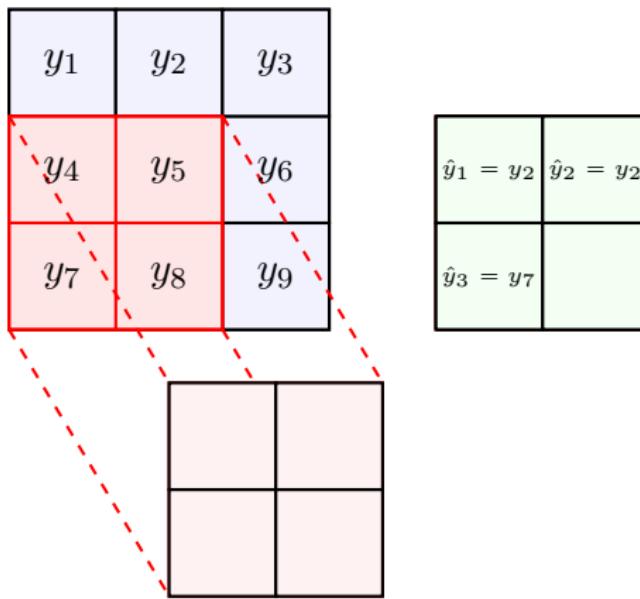
Backpropagate through Pooling: Max-Pooling

Lets start with a **simple example**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



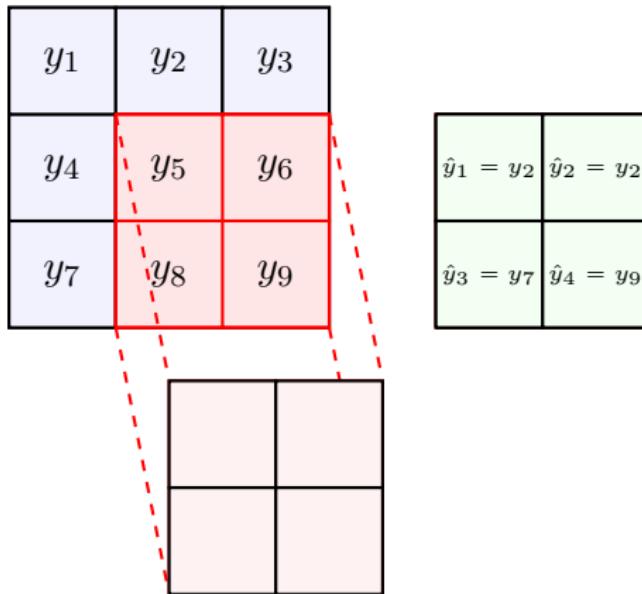
Backpropagate through Pooling: Max-Pooling

Lets start with a **simple example**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



Backpropagate through Pooling: Max-Pooling

Lets start with a **simple example**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



Backpropagation: Max-Pooling

We can use **chain rule**

Backpropagation: Max-Pooling

We can use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = 0$$

Backpropagation: Max-Pooling

We can use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = 0$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

Backpropagation: Max-Pooling

We can use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = 0$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

$$\frac{\partial \hat{R}}{\partial y_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_3} = 0$$

⋮

Backpropagation: Max-Pooling

We can use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = 0$$

$$\frac{\partial \hat{R}}{\partial y_7} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_7} = \frac{\partial \hat{R}}{\partial \hat{y}_3}$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

$$\frac{\partial \hat{R}}{\partial y_8} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_8} = 0$$

$$\frac{\partial \hat{R}}{\partial y_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_3} = 0$$

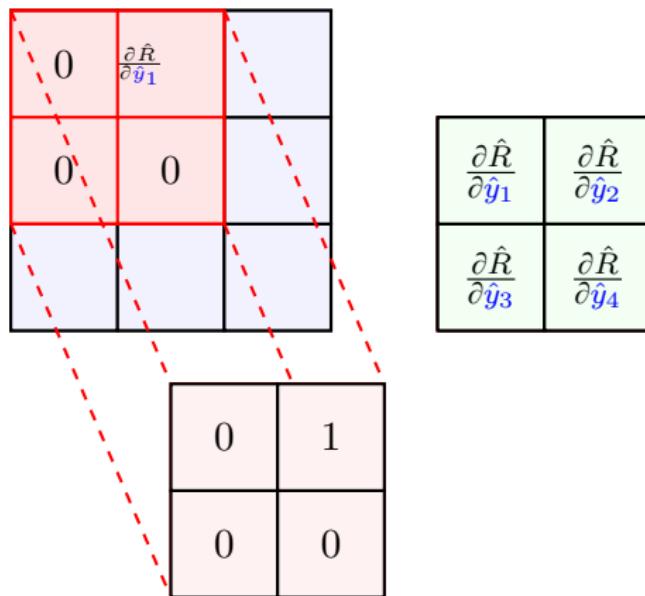
$$\frac{\partial \hat{R}}{\partial y_9} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_9} = \frac{\partial \hat{R}}{\partial \hat{y}_4}$$

⋮

Let's see its visualization

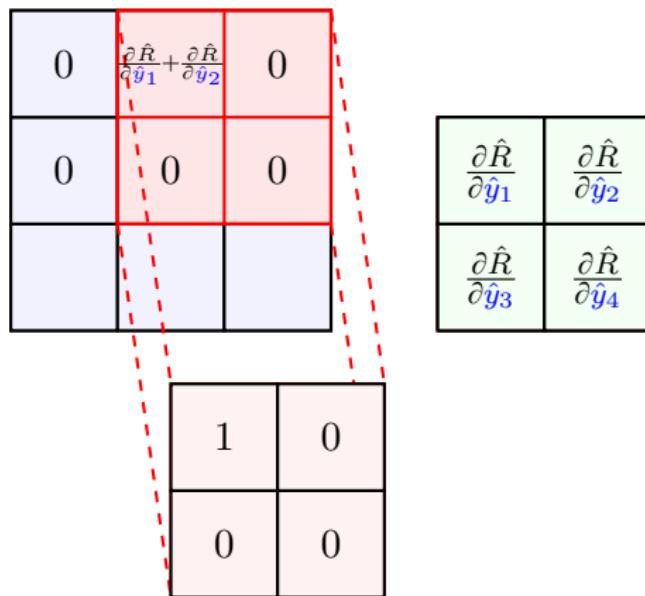
Backpropagation: Max-Pooling

Each derivative in *green map* gets multiplied with its filter and added to corresponding entries on *blue map*



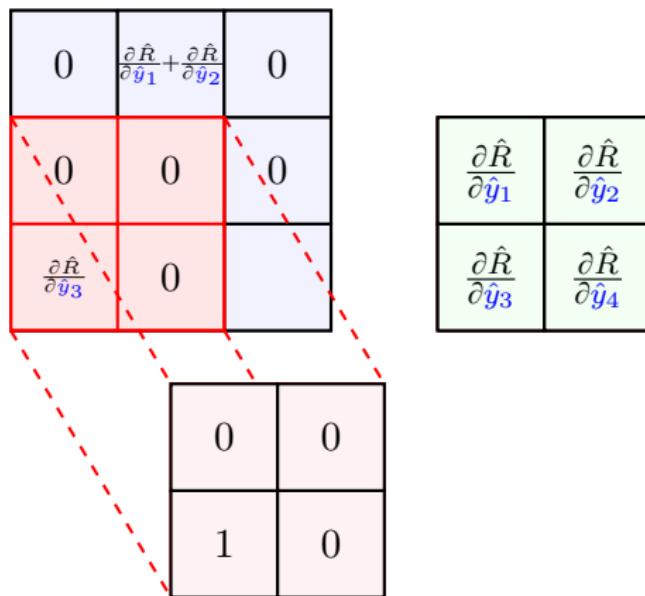
Backpropagation: Max-Pooling

Each derivative in *green map* gets multiplied with its filter and added to corresponding entries on *blue map*



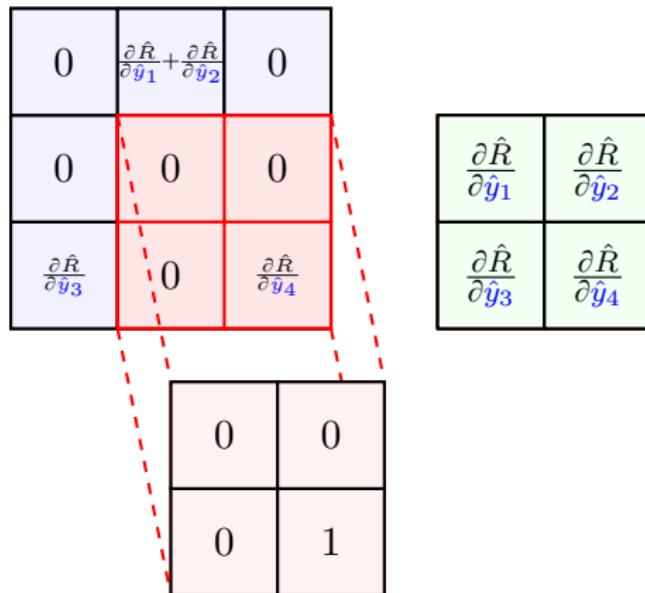
Backpropagation: Max-Pooling

Each derivative in *green map* gets multiplied with its filter and added to corresponding entries on *blue map*



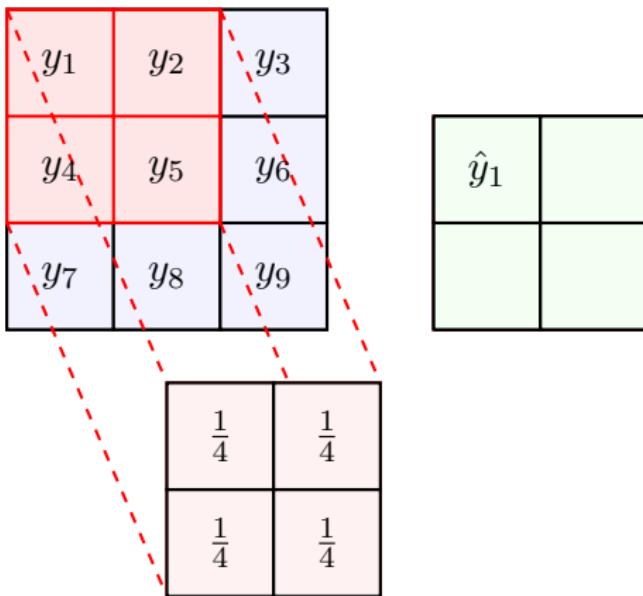
Backpropagation: Max-Pooling

Each derivative in *green map* gets multiplied with its filter and added to corresponding entries on *blue map*



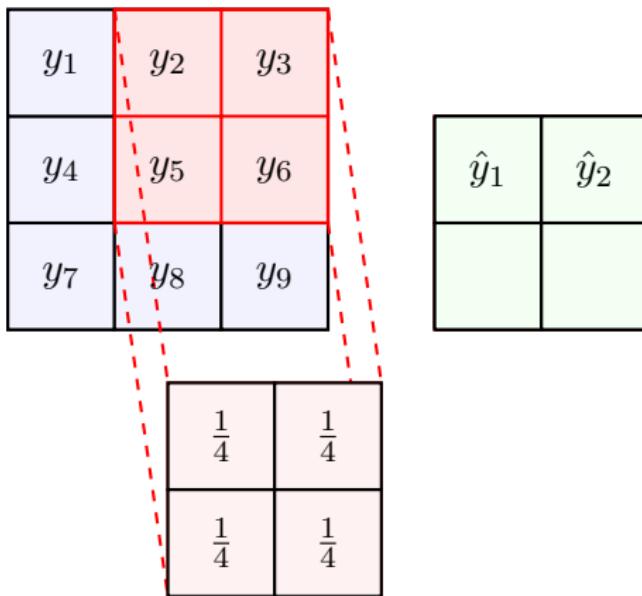
Backpropagate through Pooling: Mean-Pooling

Lets now consider **mean-pooling**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



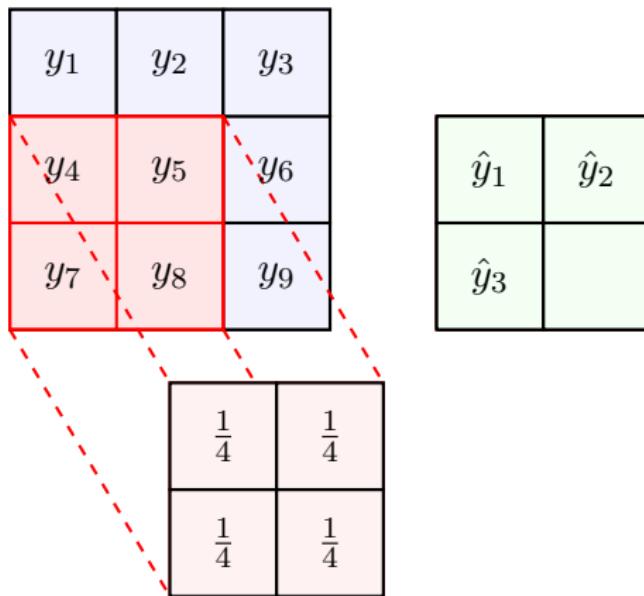
Backpropagate through Pooling: Mean-Pooling

Lets now consider **mean-pooling**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



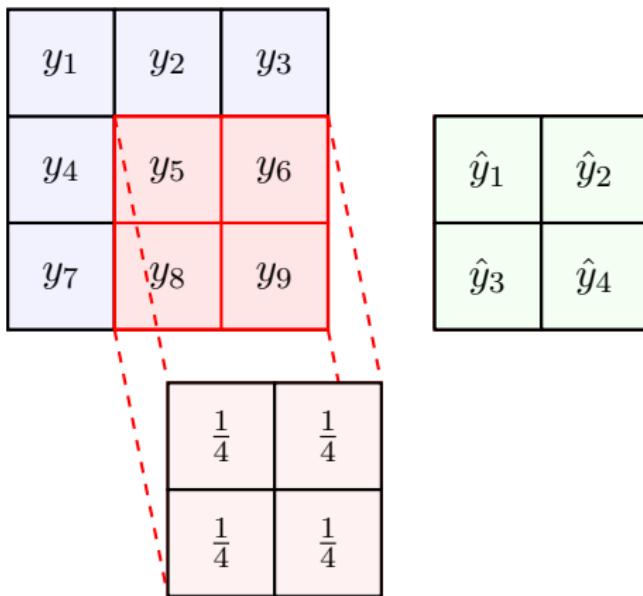
Backpropagate through Pooling: Mean-Pooling

Lets now consider **mean-pooling**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



Backpropagate through Pooling: Mean-Pooling

Lets now consider **mean-pooling**: we have partial derivatives with respect to **pooled variables** and want to compute the partial derivatives with respect to **input of pooling layer**



Backward Pass: Mean-Pooling

We again use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1}$$

Backward Pass: Mean-Pooling

We again use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1}$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

Backward Pass: Mean-Pooling

We again use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1}$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

⋮

$$\frac{\partial \hat{R}}{\partial y_5} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_5} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_2} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_3} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_4}$$

Backward Pass: Mean-Pooling

We again use **chain rule**

$$\frac{\partial \hat{R}}{\partial y_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_1} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1}$$

$$\frac{\partial \hat{R}}{\partial y_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_2} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_2}$$

⋮

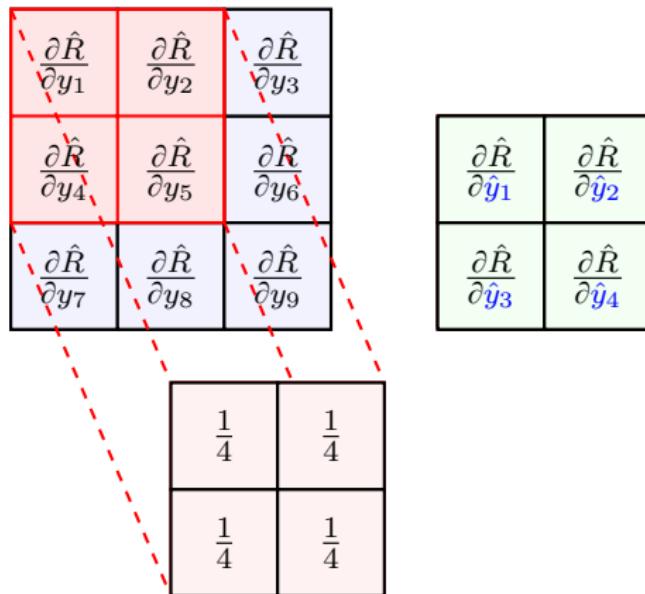
$$\frac{\partial \hat{R}}{\partial y_5} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_5} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_1} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_2} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_3} + \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_4}$$

⋮

$$\frac{\partial \hat{R}}{\partial y_9} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_9} = \frac{1}{4} \frac{\partial \hat{R}}{\partial \hat{y}_4}$$

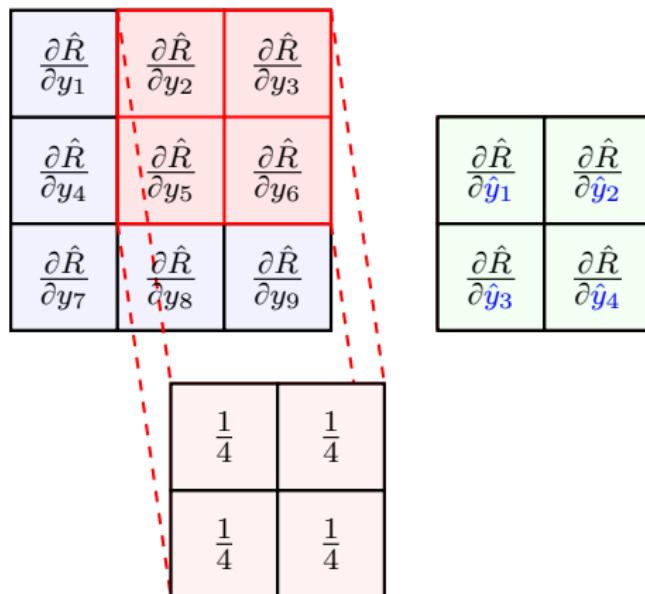
Backward Pass: Mean-Pooling

*It has same visualization: the **filter** is however this time fixed*



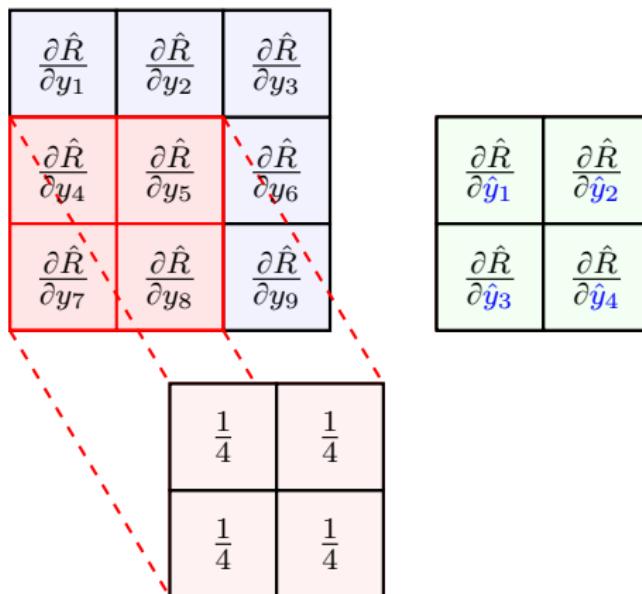
Backward Pass: Mean-Pooling

*It has same visualization: the **filter** is however this time fixed*



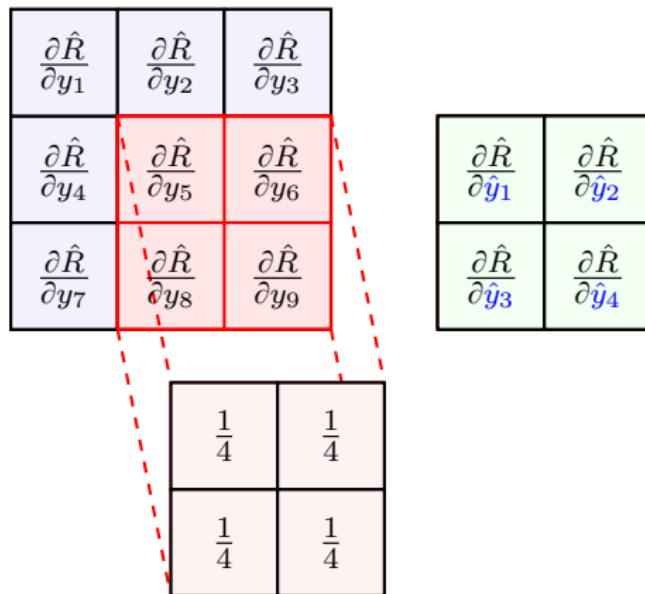
Backward Pass: Mean-Pooling

*It has same visualization: the **filter** is however this time fixed*



Backward Pass: Mean-Pooling

*It has same visualization: the **filter** is however this time fixed*



Backward Pass: Mean-Pooling

*It has same visualization: the **filter** is however this time fixed*

$\frac{\partial \hat{R}}{\partial y_1}$	$\frac{\partial \hat{R}}{\partial y_2}$	$\frac{\partial \hat{R}}{\partial y_3}$
$\frac{\partial \hat{R}}{\partial y_4}$	$\frac{\partial \hat{R}}{\partial y_5}$	$\frac{\partial \hat{R}}{\partial y_6}$
$\frac{\partial \hat{R}}{\partial y_7}$	$\frac{\partial \hat{R}}{\partial y_8}$	$\frac{\partial \hat{R}}{\partial y_9}$

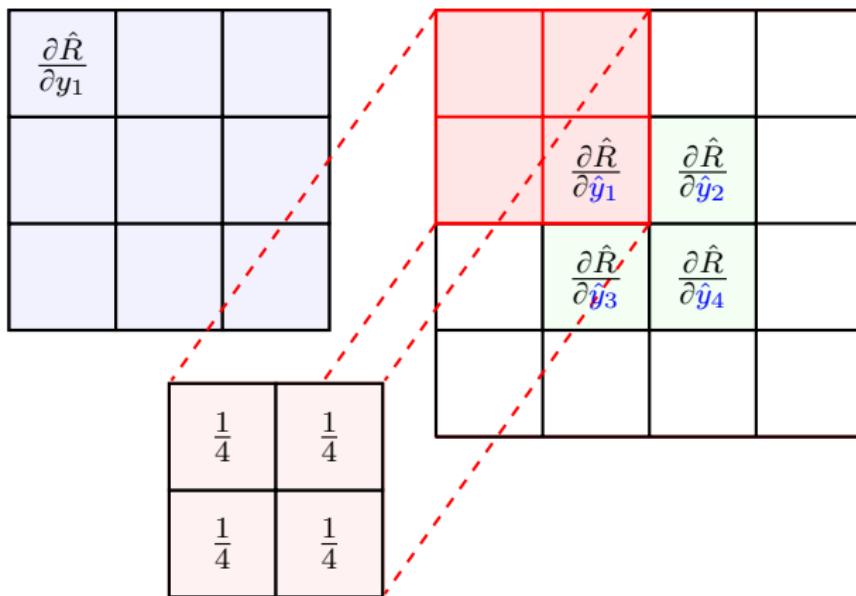
$\frac{\partial \hat{R}}{\partial \hat{y}_1}$	$\frac{\partial \hat{R}}{\partial \hat{y}_2}$
$\frac{\partial \hat{R}}{\partial \hat{y}_3}$	$\frac{\partial \hat{R}}{\partial \hat{y}_4}$

$\frac{1}{4}$	$\frac{1}{4}$
$\frac{1}{4}$	$\frac{1}{4}$

But we can visualize it even better!

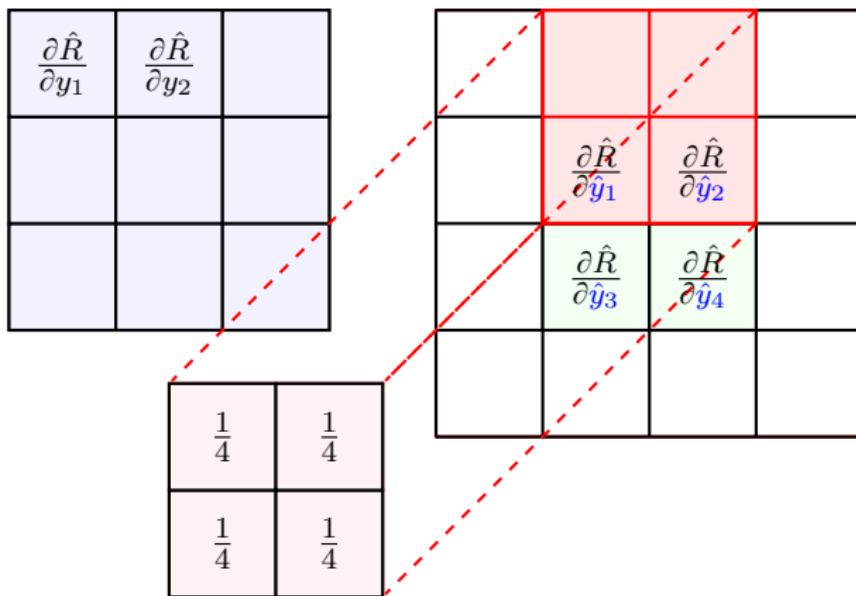
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



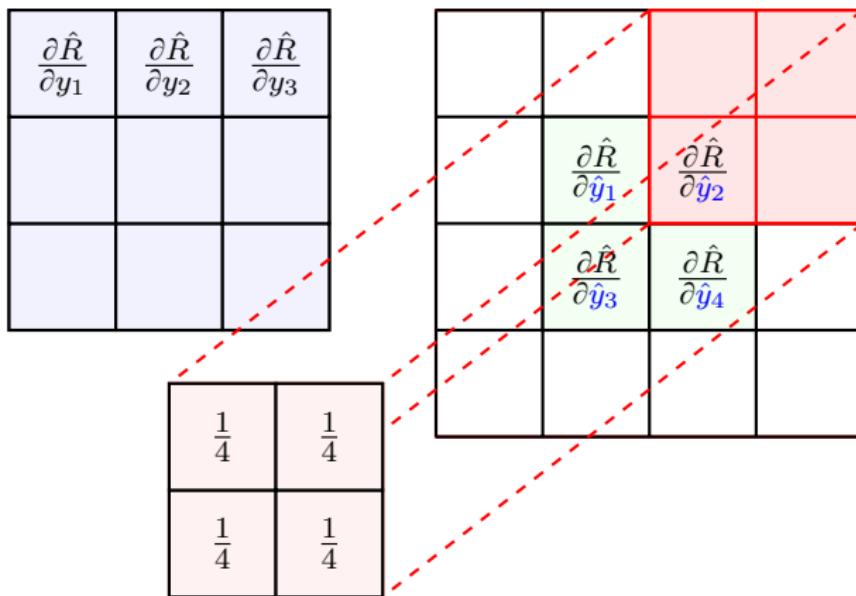
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



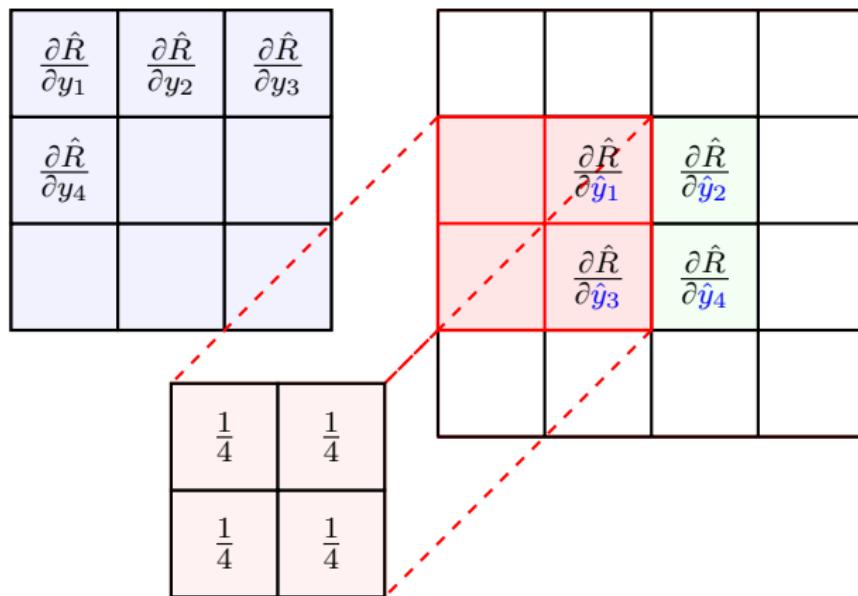
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



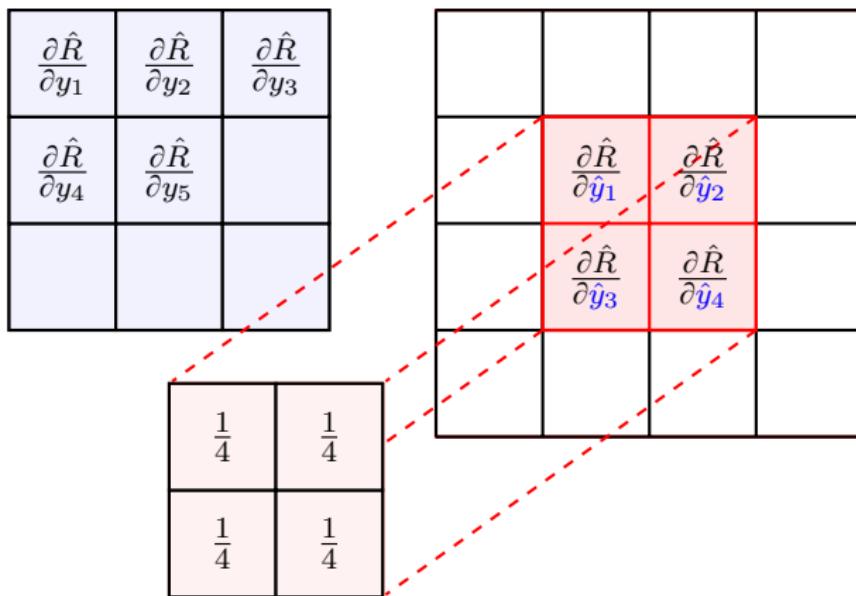
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



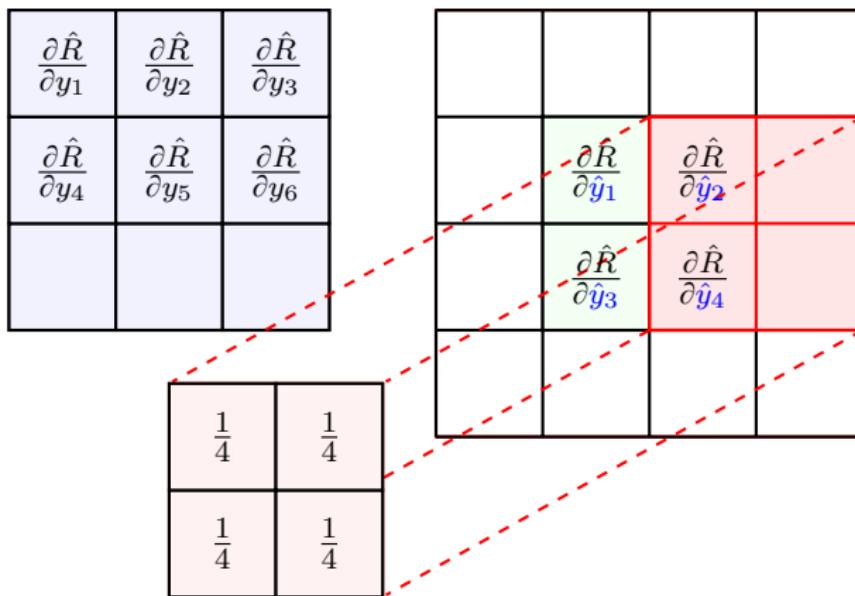
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



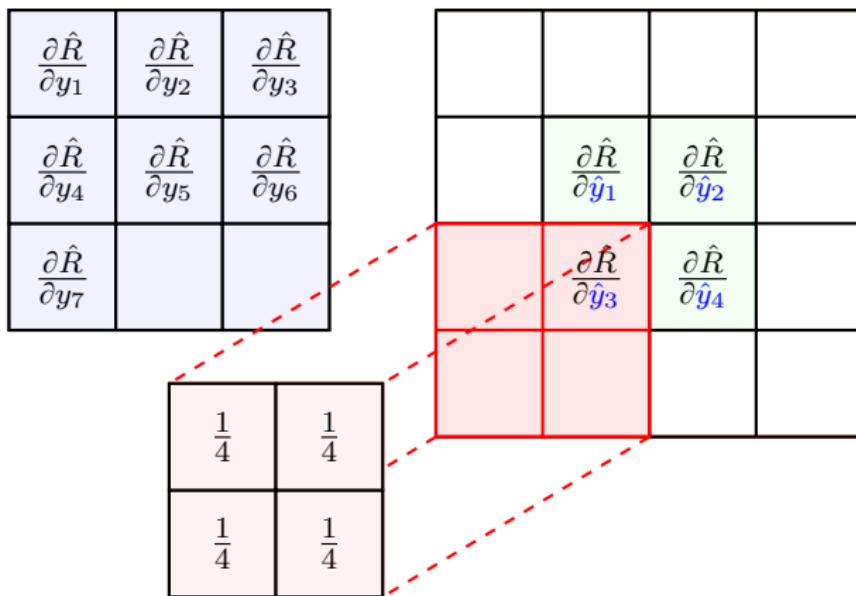
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



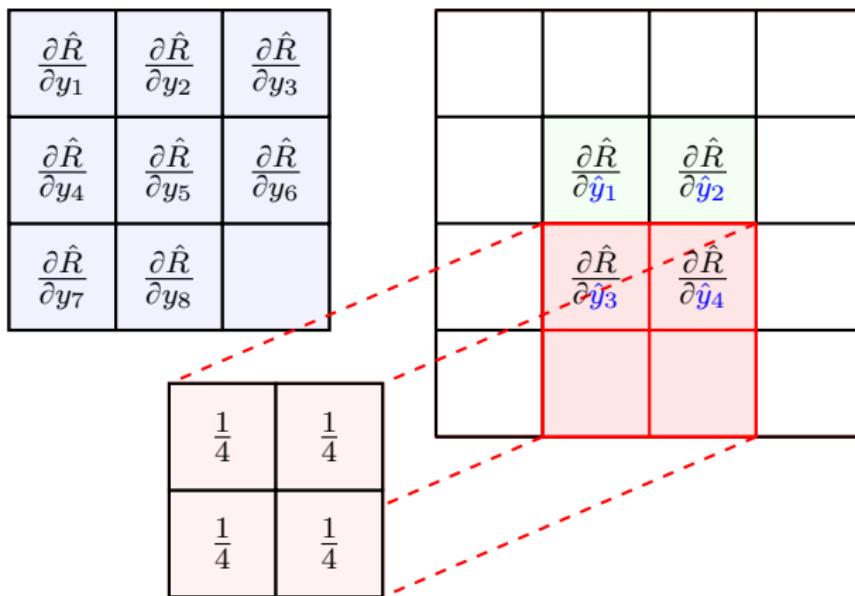
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



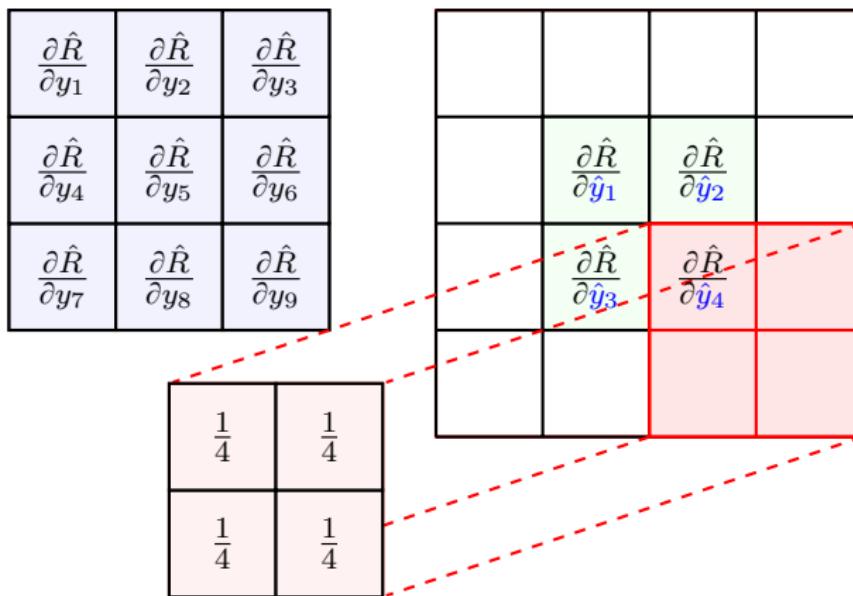
Backward Pass: Mean-Pooling

In fact, this is simply a convolution



Backward Pass: Mean-Pooling

In fact, this is simply a convolution



Backpropagation through Pooling: Summary

Depending on **pooling** function, backpropagation can be different

↳ *It's however usually described by a convolution*

Backpropagation through Pooling: Summary

Depending on **pooling** function, backpropagation can be different

↳ *It's however usually described by a convolution*

Moral of Story

Backpropagation through **pooling** can be done by a **convolution-type operation**

Backpropagation through Convolution: Activation

Convolutional layer has two operations

- ↳ linear convolution
 - ↳ entry-wise activation

Backpropagation through Convolution: Activation

Convolutional layer has two operations

- ↳ *linear convolution*
- ↳ *entry-wise activation*

The *entry-wise activation* is readily backpropagate: say we have

$$\mathbf{Y} = f(\mathbf{Z})$$

Then, we can *backpropagate* as in the *fully-connected* FNNs

$$\nabla_{\mathbf{Z}} \hat{R} = \nabla_{\mathbf{Y}} \hat{R} \odot \dot{f}(\mathbf{Z})$$

Backpropagation through Convolution: Activation

Convolutional layer has two operations

- ↳ *linear convolution*
- ↳ *entry-wise activation*

The *entry-wise activation* is readily backpropagate: say we have

$$\mathbf{Y} = f(\mathbf{Z})$$

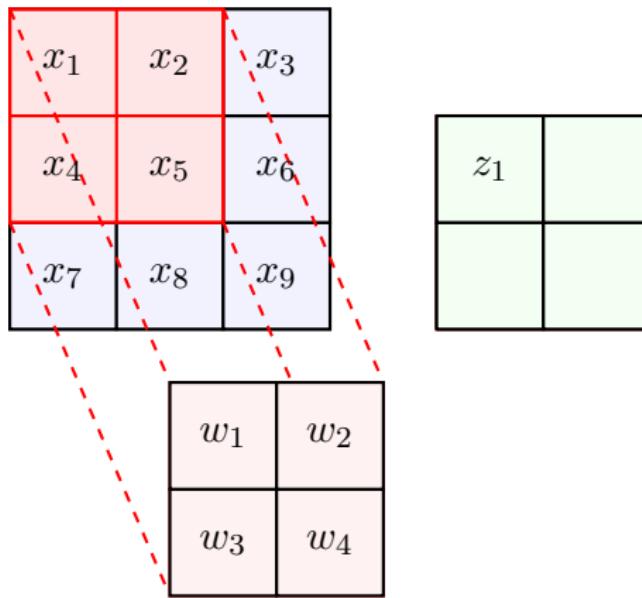
Then, we can *backpropagate* as in the *fully-connected* FNNs

$$\nabla_{\mathbf{Z}} \hat{R} = \nabla_{\mathbf{Y}} \hat{R} \odot \dot{f}(\mathbf{Z})$$

Now, let's look at *linear convolution!*

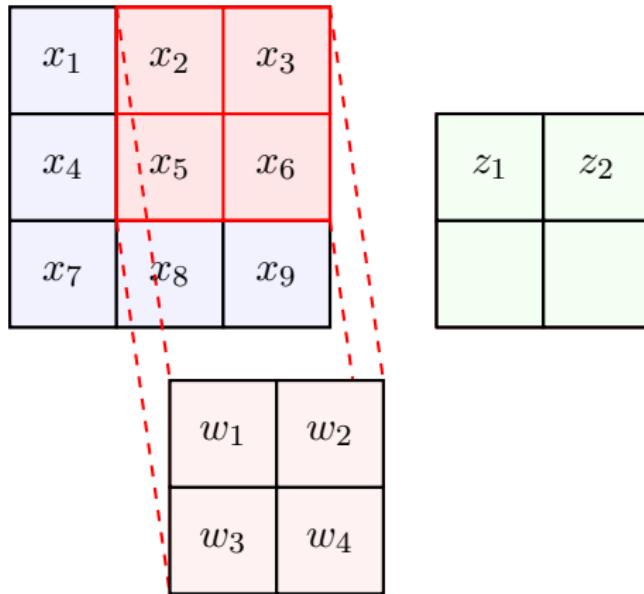
Backpropagation: 2D Convolution

Let's find it out through a **simple example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **input of convolutional layer**



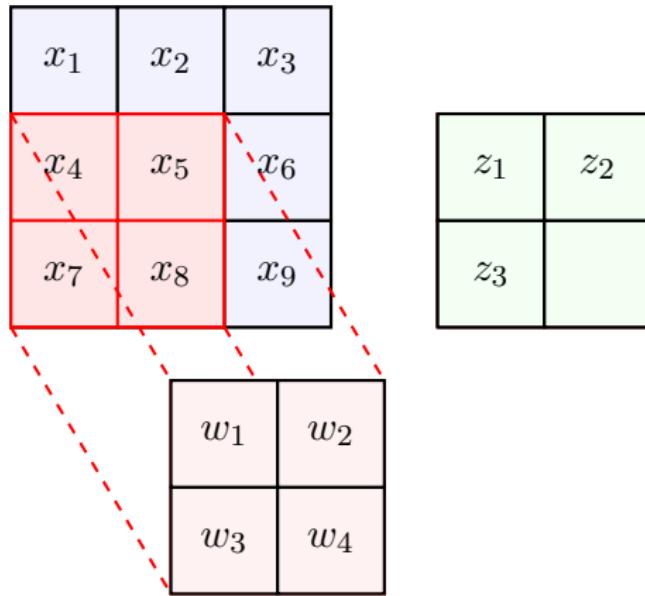
Backpropagation: 2D Convolution

Let's find it out through a **simple example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **input of convolutional layer**



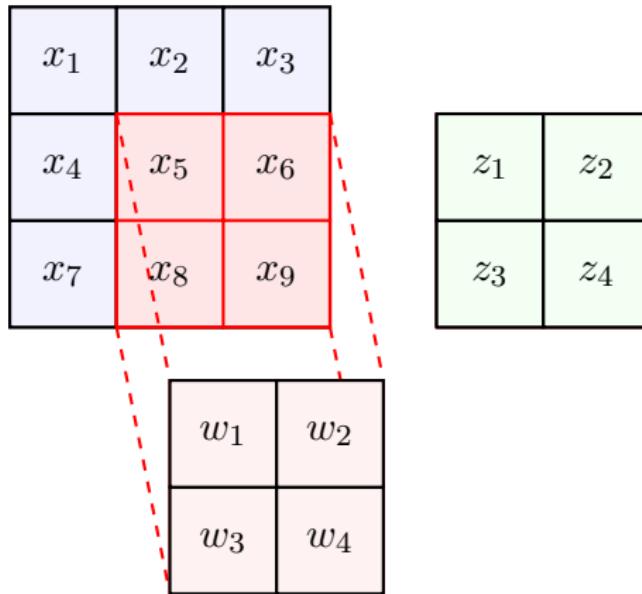
Backpropagation: 2D Convolution

Let's find it out through a **simple example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **input of convolutional layer**



Backpropagation: 2D Convolution

Let's find it out through a **simple example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **input of convolutional layer**



Backpropagation: 2D Convolution

Let's start with [chain rule](#)

Backpropagation: 2D Convolution

Let's start with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = w_1 \frac{\partial \hat{R}}{\partial z_1}$$

Backpropagation: 2D Convolution

Let's start with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = w_1 \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = w_2 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_2}$$

Backpropagation: 2D Convolution

Let's start with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = w_1 \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = w_2 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = w_2 \frac{\partial \hat{R}}{\partial z_2}$$

Backpropagation: 2D Convolution

Let's start with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = w_1 \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = w_2 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = w_2 \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_4} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_4} = w_3 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_3}$$

Backpropagation: 2D Convolution

Let's start with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = w_1 \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = w_2 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = w_2 \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_4} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_4} = w_3 \frac{\partial \hat{R}}{\partial z_1} + w_1 \frac{\partial \hat{R}}{\partial z_3}$$

$$\frac{\partial \hat{R}}{\partial x_5} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_5} = w_4 \frac{\partial \hat{R}}{\partial z_1} + w_3 \frac{\partial \hat{R}}{\partial z_2} + w_2 \frac{\partial \hat{R}}{\partial z_3} + w_1 \frac{\partial \hat{R}}{\partial z_4}$$

Backpropagation: 2D Convolution

We can keep on till finish with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_6} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_6} = w_4 \frac{\partial \hat{R}}{\partial z_2} + w_2 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial x_7} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_7} = w_3 \frac{\partial \hat{R}}{\partial z_3}$$

$$\frac{\partial \hat{R}}{\partial x_8} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_8} = w_4 \frac{\partial \hat{R}}{\partial z_3} + w_3 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial x_9} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_9} = w_4 \frac{\partial \hat{R}}{\partial z_4}$$

Backpropagation: 2D Convolution

We can keep on till finish with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_6} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_6} = w_4 \frac{\partial \hat{R}}{\partial z_2} + w_2 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial x_7} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_7} = w_3 \frac{\partial \hat{R}}{\partial z_3}$$

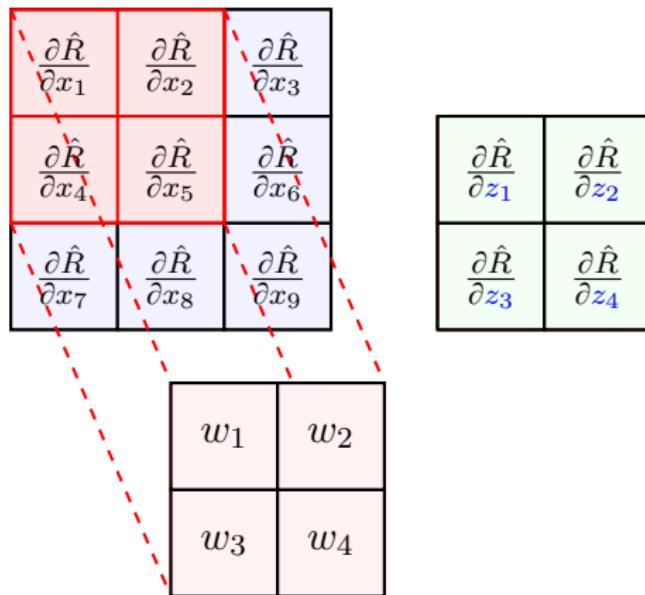
$$\frac{\partial \hat{R}}{\partial x_8} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_8} = w_4 \frac{\partial \hat{R}}{\partial z_3} + w_3 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial x_9} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_9} = w_4 \frac{\partial \hat{R}}{\partial z_4}$$

Let's look at the visualization

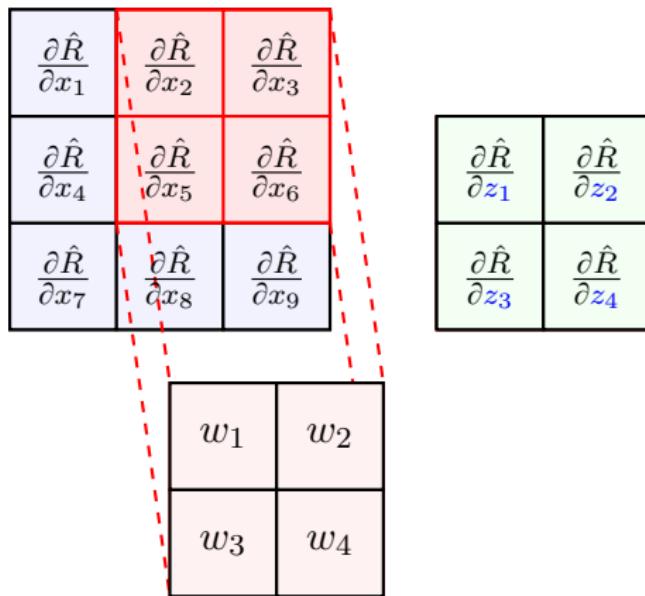
Backpropagation: 2D Convolution

We can use the same visualization as in pooling



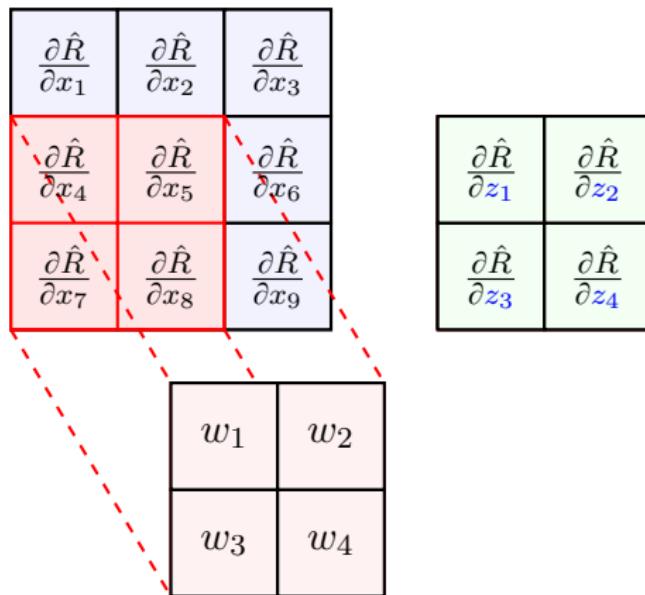
Backpropagation: 2D Convolution

We can use the same visualization as in pooling



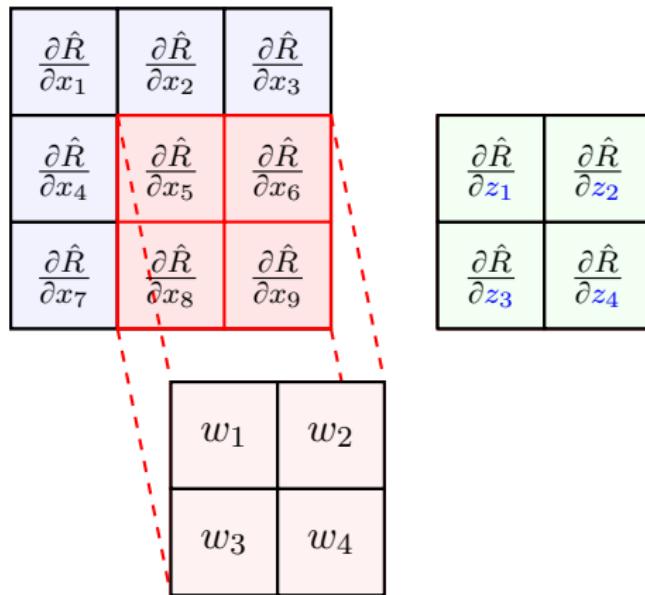
Backpropagation: 2D Convolution

We can use the same visualization as in pooling



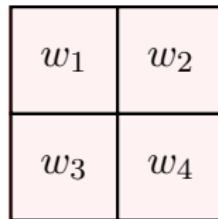
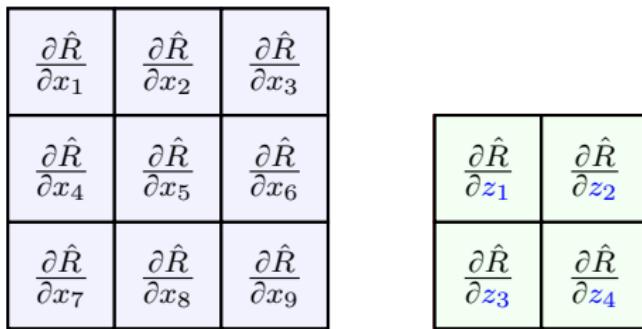
Backpropagation: 2D Convolution

We can use the same visualization as in pooling



Backpropagation: 2D Convolution

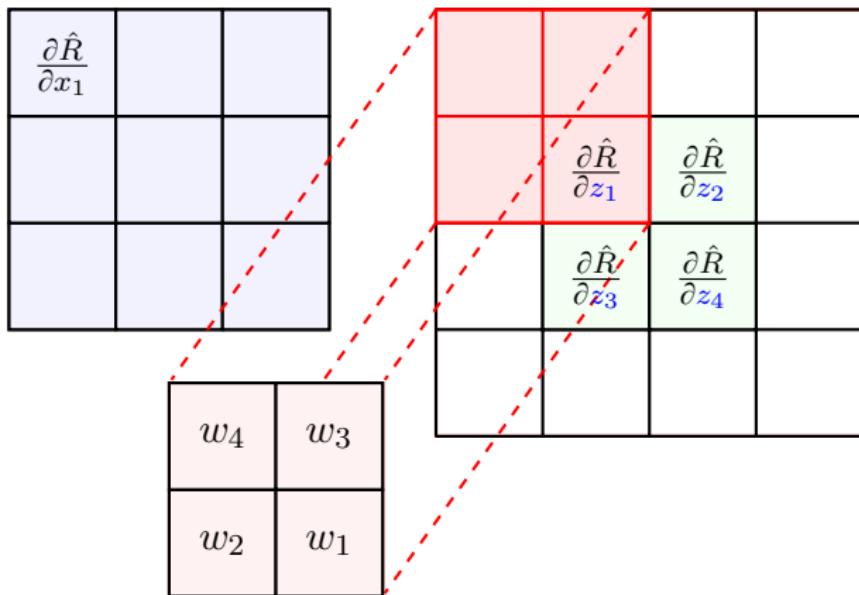
We can use the same visualization as in pooling



Or even a **better** visualization!

Backpropagation: 2D Convolution

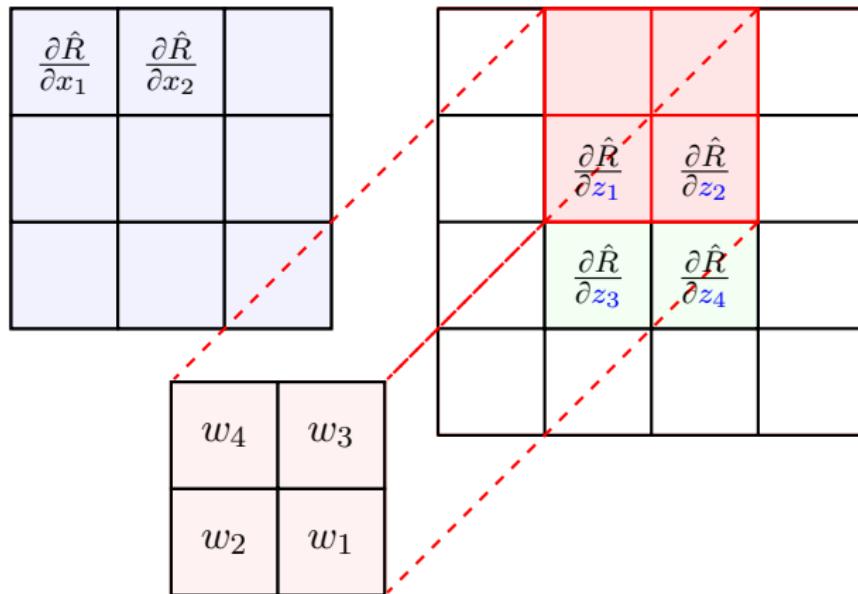
It's again simply a convolution



with **filter** being **reversed up-to-down** and **right-to-left**

Backpropagation: 2D Convolution

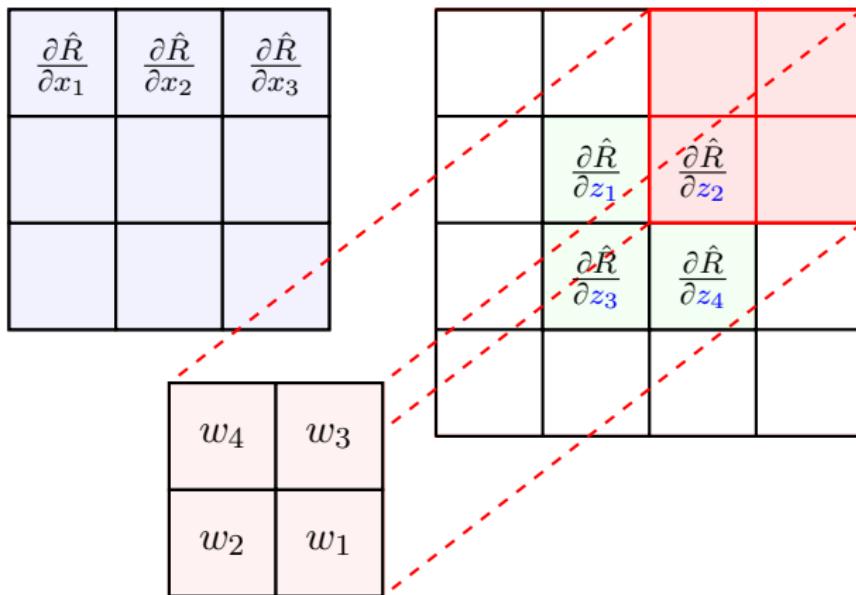
It's again simply a convolution



with **filter** being **reversed up-to-down** and **right-to-left**

Backpropagation: 2D Convolution

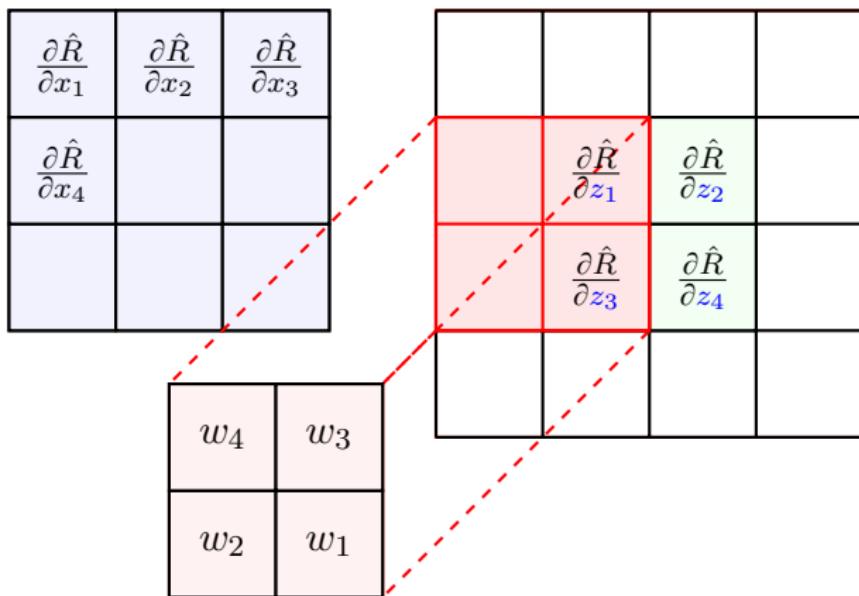
It's again simply a convolution



with **filter** being **reversed up-to-down** and **right-to-left**

Backpropagation: 2D Convolution

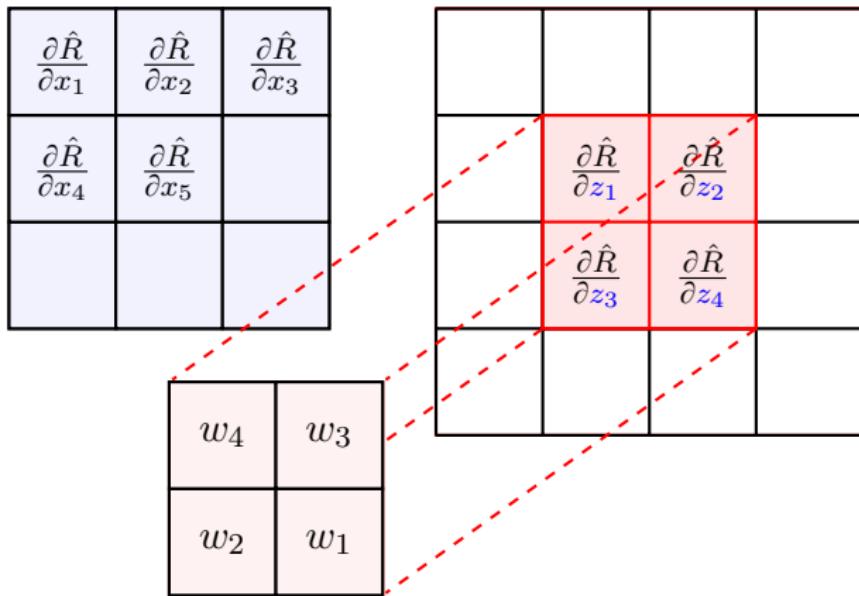
It's again simply a convolution



with **filter** being **reversed up-to-down and right-to-left**

Backpropagation: 2D Convolution

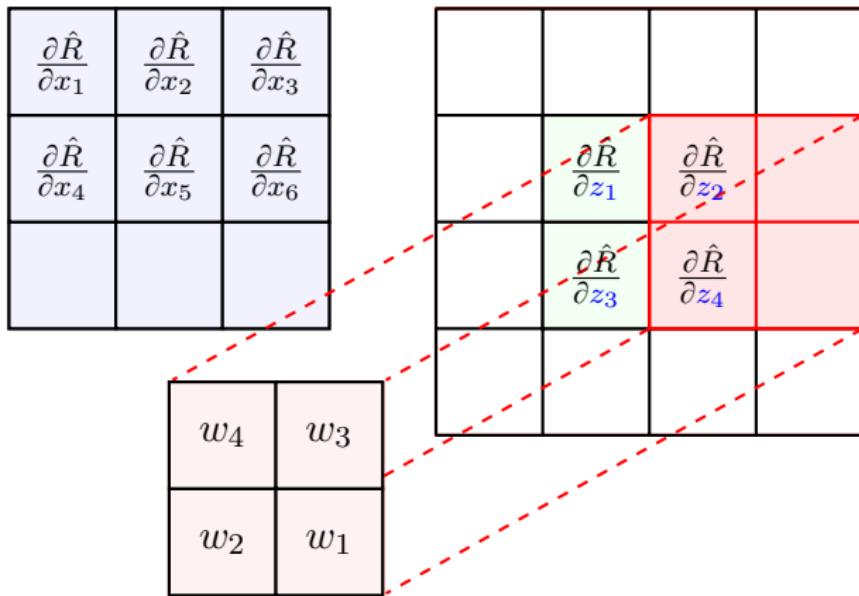
It's again simply a convolution



with **filter** being **reversed up-to-down and right-to-left**

Backpropagation: 2D Convolution

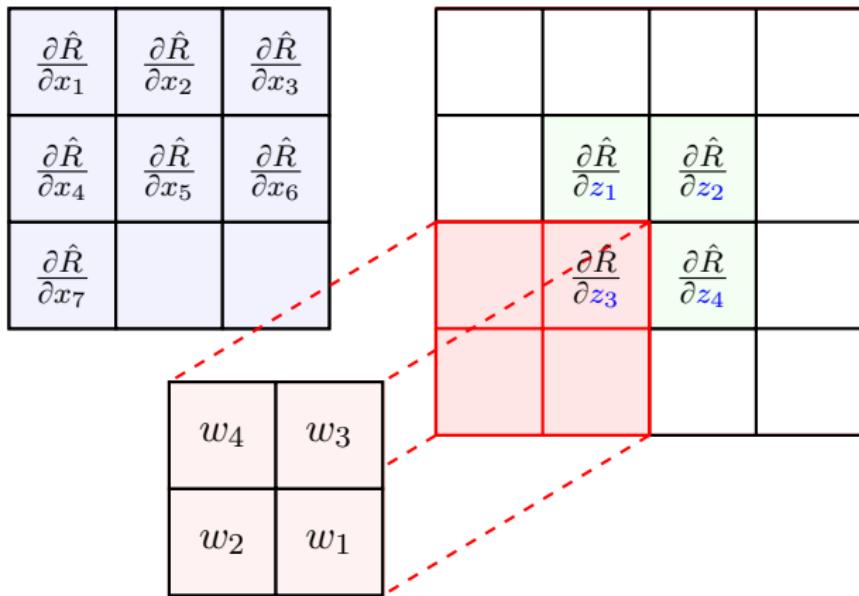
It's again simply a convolution



with **filter** being **reversed up-to-down** and **right-to-left**

Backpropagation: 2D Convolution

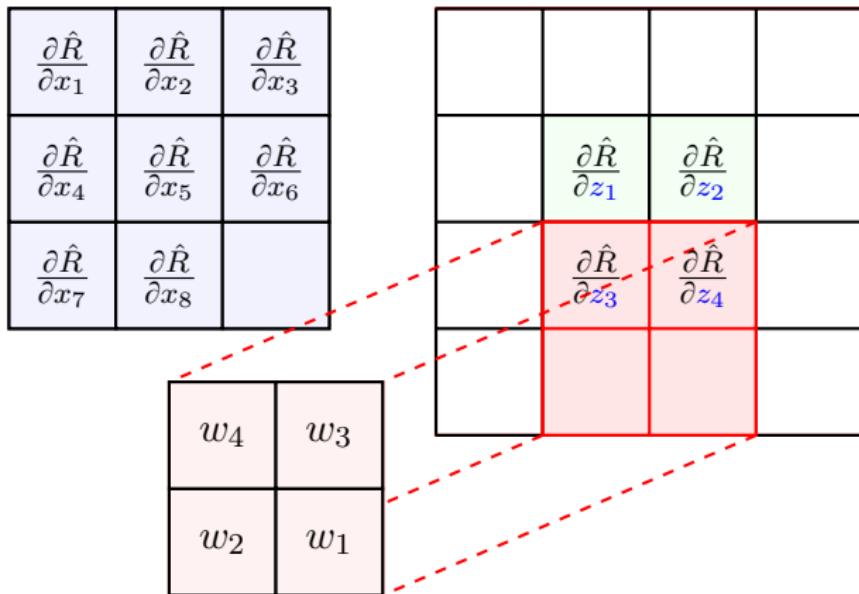
It's again simply a convolution



with **filter** being **reversed up-to-down and right-to-left**

Backpropagation: 2D Convolution

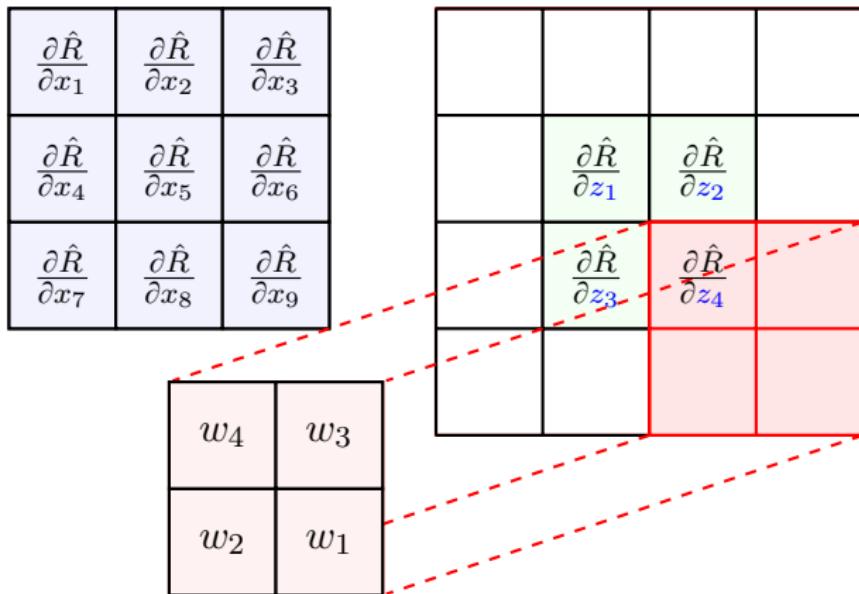
It's again simply a convolution



with **filter** being **reversed up-to-down and right-to-left**

Backpropagation: 2D Convolution

It's again simply a convolution



with **filter** being **reversed up-to-down and right-to-left**

Backpropagation: 2D Convolution

*To backpropagate a convolutional layer, we convolve the output gradient with the **filter** being **reversed up-to-down** and **right-to-left**. To match the dimension, we simply apply **zero-padding***

Backpropagation: 2D Convolution

To backpropagate a convolutional layer, we convolve the output gradient with the **filter** being **reversed up-to-down** and **right-to-left**. To match the dimension, we simply apply **zero-padding**

Backpropagation through 2D Convolution

Let $\mathbf{Z} = \text{Conv}(\mathbf{X}|\mathbf{W})$, where \mathbf{X} is the **input map**, i.e., a matrix, \mathbf{W} is **filter** and \mathbf{Z} is the **output map**. Assume that we know the gradient of loss \hat{R} with respect to the **output map**, i.e., $\nabla_{\mathbf{Z}} \hat{R}$. Then, the gradient of loss \hat{R} with respect to the **input map** is

$$\nabla_{\mathbf{X}} \hat{R} = \text{Conv}\left(\nabla_{\mathbf{Z}} \hat{R} | \check{\mathbf{W}}\right)$$

where $\check{\mathbf{W}}$ is the **up-to-down** and **left-to-right** reverse of **filter** \mathbf{W}

Backpropagation: 2D Convolution

- + What if the dimensions do **not** match?
 - The dimensions can **only** not match if

Backpropagation: 2D Convolution

- + *What if the dimensions do **not** match?*
- The dimensions can **only** not match if
 - ① we do the forward convolution with a **stride different from one**
 - ↳ We said that this is simply **resampling**
 - ↳ We are going to **discuss it later**
 - ↳ For now assume that we do the convolution with **stride one**

Backpropagation: 2D Convolution

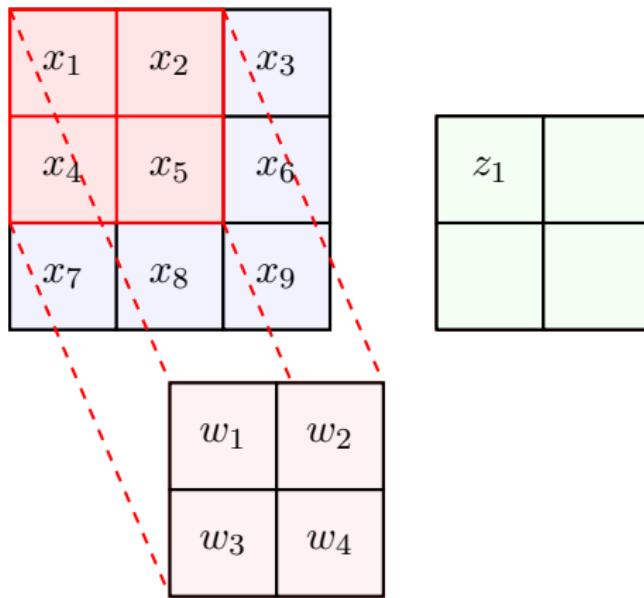
- + What if the dimensions do **not** match?
- The dimensions can **only** not match if
 - ① we do the forward convolution with a **stride different from one**
 - ↳ We said that this is simply **resampling**
 - ↳ We are going to **discuss it later**
 - ↳ For now assume that we do the convolution with **stride one**
 - ② we did **no zero-padding** in the forward convolution
 - ↳ We simply **do zero-padding** in backward convolution to match the dimensions

Backpropagation: 2D Convolution

- + What if the dimensions do **not** match?
- The dimensions can **only** not match if
 - ① we do the forward convolution with a **stride different from one**
 - ↳ We said that this is simply **resampling**
 - ↳ We are going to **discuss it later**
 - ↳ For now assume that we do the convolution with **stride one**
 - ② we did **no zero-padding** in the forward convolution
 - ↳ We simply **do zero-padding** in backward convolution to match the dimensions
- + What about the **gradient** with respect to **filter itself**? Don't we need it?
- Let's check it out

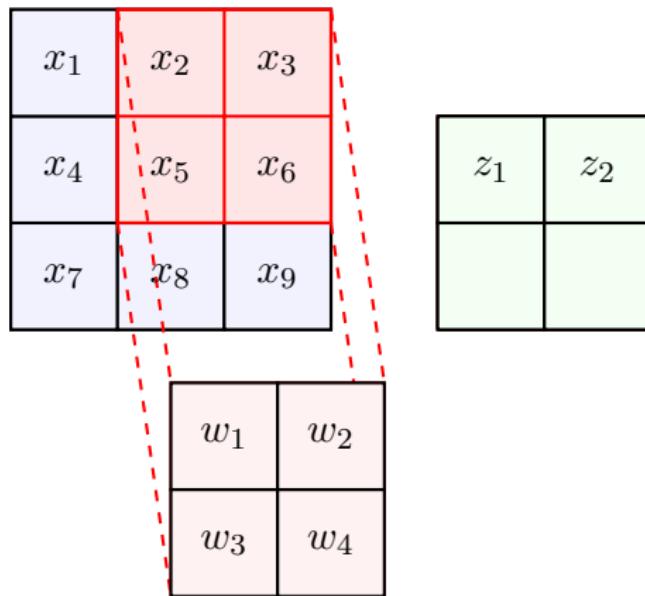
Backpropagation: Convolution Filter

We try it for **our example**: we have *partial derivatives with respect to convolved variables* and want to compute the *partial derivatives with respect to weights in the filter*



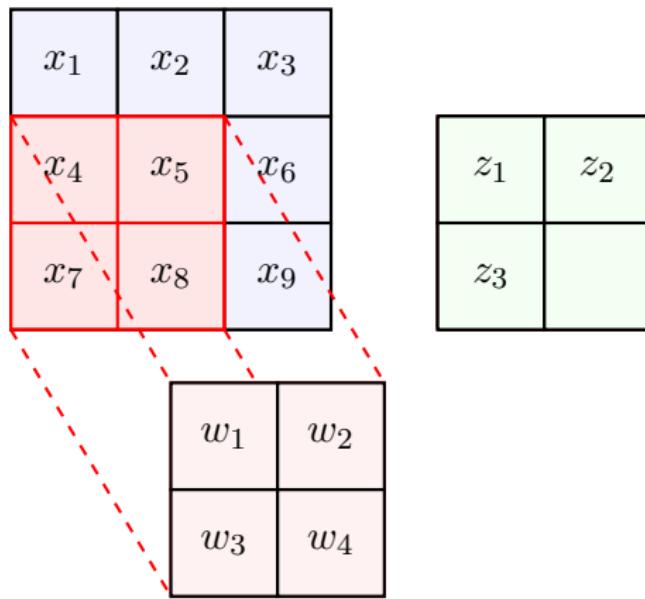
Backpropagation: Convolution Filter

We try it for **our example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **weights in the filter**



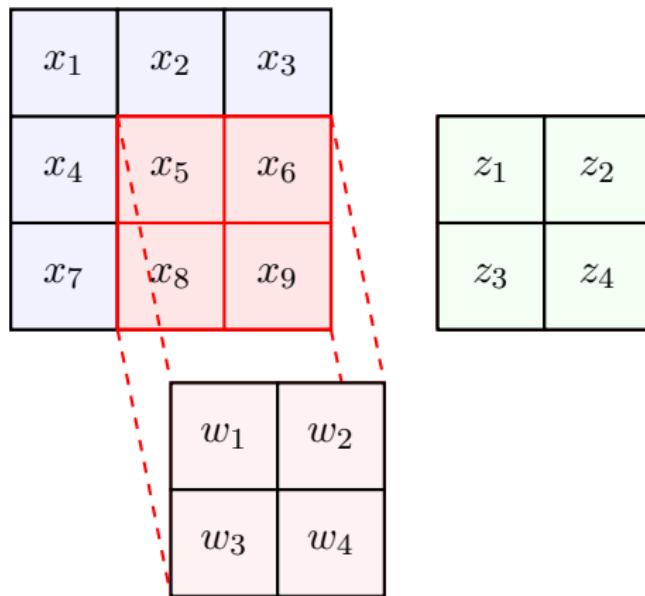
Backpropagation: Convolution Filter

We try it for **our example**: we have *partial derivatives with respect to convolved variables* and want to compute the *partial derivatives with respect to weights in the filter*



Backpropagation: Convolution Filter

We try it for **our example**: we have partial derivatives with respect to **convolved variables** and want to compute the partial derivatives with respect to **weights in the filter**



Backpropagation: Convolution Filter

As always, we use *chain rule*

$$\frac{\partial \hat{R}}{\partial w_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial w_1} = x_1 \frac{\partial \hat{R}}{\partial z_1} + x_2 \frac{\partial \hat{R}}{\partial z_2} + x_4 \frac{\partial \hat{R}}{\partial z_3} + x_5 \frac{\partial \hat{R}}{\partial z_4}$$

Backpropagation: Convolution Filter

As always, we use *chain rule*

$$\frac{\partial \hat{R}}{\partial w_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial w_1} = x_1 \frac{\partial \hat{R}}{\partial z_1} + x_2 \frac{\partial \hat{R}}{\partial z_2} + x_4 \frac{\partial \hat{R}}{\partial z_3} + x_5 \frac{\partial \hat{R}}{\partial z_4}$$

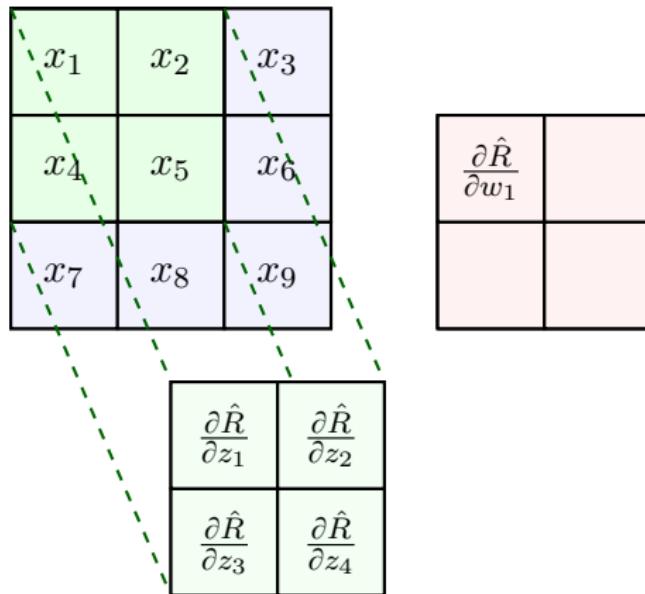
$$\frac{\partial \hat{R}}{\partial w_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial w_2} = x_2 \frac{\partial \hat{R}}{\partial z_1} + x_3 \frac{\partial \hat{R}}{\partial z_2} + x_5 \frac{\partial \hat{R}}{\partial z_3} + x_6 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial w_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial w_3} = x_4 \frac{\partial \hat{R}}{\partial z_1} + x_5 \frac{\partial \hat{R}}{\partial z_2} + x_7 \frac{\partial \hat{R}}{\partial z_3} + x_8 \frac{\partial \hat{R}}{\partial z_4}$$

$$\frac{\partial \hat{R}}{\partial w_4} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial w_4} = x_5 \frac{\partial \hat{R}}{\partial z_1} + x_6 \frac{\partial \hat{R}}{\partial z_2} + x_8 \frac{\partial \hat{R}}{\partial z_3} + x_9 \frac{\partial \hat{R}}{\partial z_4}$$

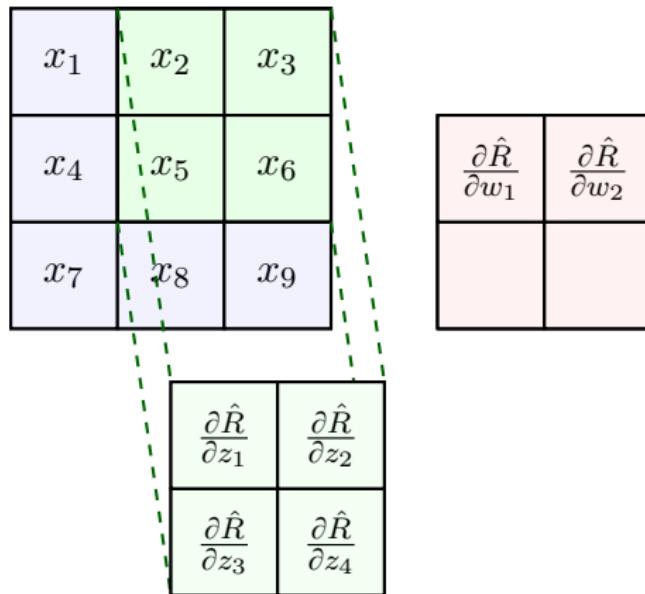
Backpropagation: Convolution Filter

It's another convolution with $\nabla_{\mathbf{z}} \hat{R}$ being the filter!



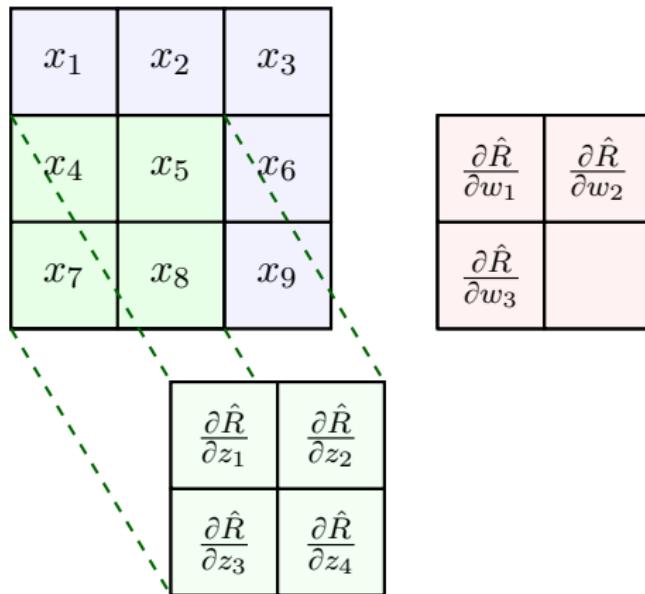
Backpropagation: Convolution Filter

It's another convolution with $\nabla_{\mathbf{z}} \hat{R}$ being the filter!



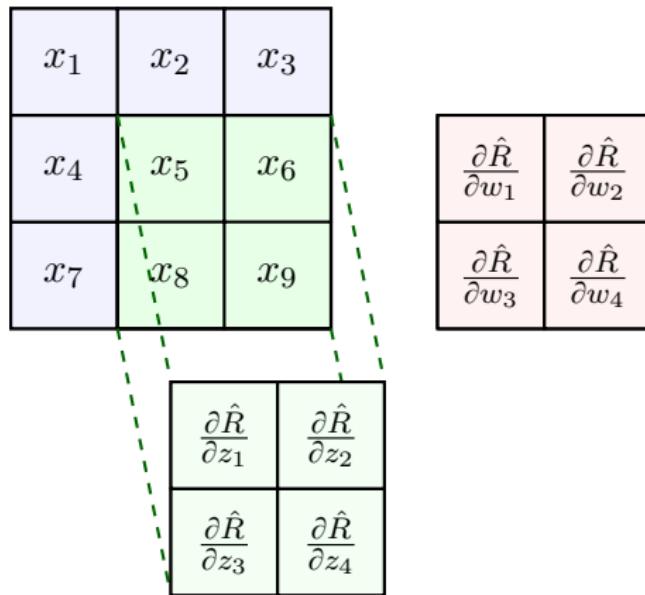
Backpropagation: Convolution Filter

It's another convolution with $\nabla_{\mathbf{z}} \hat{R}$ being the filter!



Backpropagation: Convolution Filter

It's another convolution with $\nabla_{\mathbf{z}} \hat{R}$ being the filter!



Backpropagation: Convolution Filter

Once we backpropagate to the **output** of a **convolutional layer**, then we can find the gradient with respect to convolution filter by **convolving output gradient with the input map**

Gradient w.r.t. Filters

Let $\mathbf{Z} = \text{Conv}(\mathbf{X}|\mathbf{W})$, where \mathbf{X} is the **input map**, i.e., a matrix, \mathbf{W} is **filter** and \mathbf{Z} is the **output map**. Assume that we know gradient of loss \hat{R} with respect to the **output map**, i.e., $\nabla_{\mathbf{Z}} \hat{R}$. Then, gradient of loss \hat{R} with respect to the **filter** is

$$\nabla_{\mathbf{W}} \hat{R} = \text{Conv}(\mathbf{X} | \nabla_{\mathbf{Z}} \hat{R})$$

Backpropagation: Convolution Filter

Once we backpropagate to the **output** of a **convolutional layer**, then we can find the gradient with respect to convolution filter by **convolving output gradient with the input map**

Gradient w.r.t. Filters

Let $\mathbf{Z} = \text{Conv}(\mathbf{X}|\mathbf{W})$, where \mathbf{X} is the **input map**, i.e., a matrix, \mathbf{W} is **filter** and \mathbf{Z} is the **output map**. Assume that we know gradient of loss \hat{R} with respect to the **output map**, i.e., $\nabla_{\mathbf{Z}} \hat{R}$. Then, gradient of loss \hat{R} with respect to the **filter** is

$$\nabla_{\mathbf{W}} \hat{R} = \text{Conv} \left(\mathbf{X} | \nabla_{\mathbf{Z}} \hat{R} \right)$$

- + But we checked only **2D case!** What about **multi-channel case??**
- Well, we can simply do it by **chain rule**

Backpropagation: Multi-Channel Convolution

Lets start with a simple case: we have a *C-channel input*, i.e., a *tensor* input, \mathbf{X} and we compute a *single feature map*, i.e., a *matrix*, \mathbf{Z}

↳ *Filter \mathbf{W} has also C channels*

Let's denote *channel c of input* and *filter* by $\mathbf{X}[c]$ and $\mathbf{W}[c]$; then, we can write

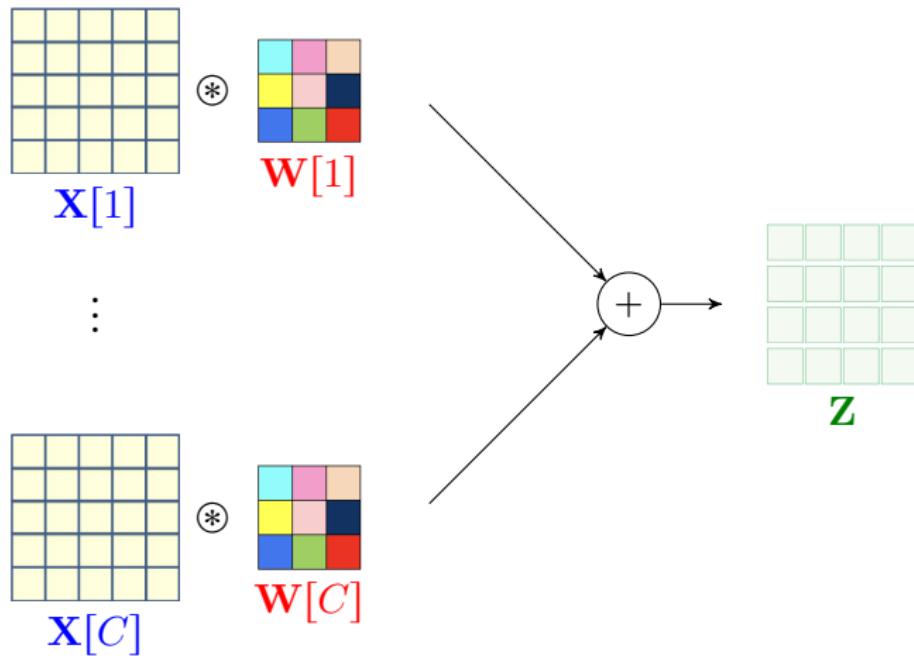
$$\mathbf{Z} = \sum_{c=1}^C \text{Conv}(\mathbf{X}[c] | \mathbf{W}[c])$$

Backpropagation: Multi-Channel Convolution

$$\begin{matrix} \text{X}[1] \\ \text{---} \\ \text{X}[C] \end{matrix} \quad * \quad \begin{matrix} \text{W}[1] \\ \text{---} \\ \text{W}[C] \end{matrix}$$

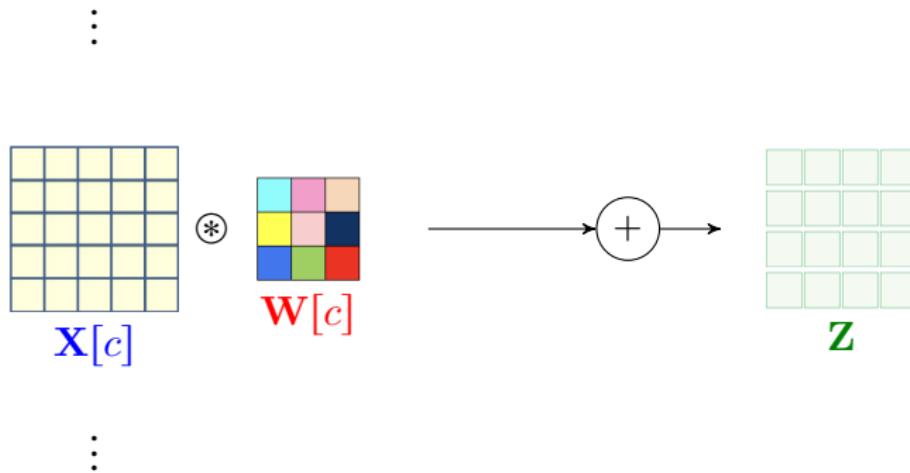
⋮

Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-Channel Convolution

Channel c of *input* is connected to the *output map* by a 2D convolution



So, we could say

$$\nabla_{\mathbf{X}[c]} \hat{R} = \text{Conv} \left(\nabla_{\mathbf{Z}} \hat{R} | \check{\mathbf{W}} \right)$$

Backpropagation: Multi-Channel Convolution

*To backpropagate a convolutional layer with **tensor input** and **matrix output**, we convolve the output gradient with the **filter of each channel** being **reversed up-to-down** and **right-to-left***

Backpropagation: Multi-Channel Convolution

To backpropagate a convolutional layer with **tensor input** and **matrix output**, we convolve the output gradient with the **filter of each channel** being **reversed up-to-down and right-to-left**

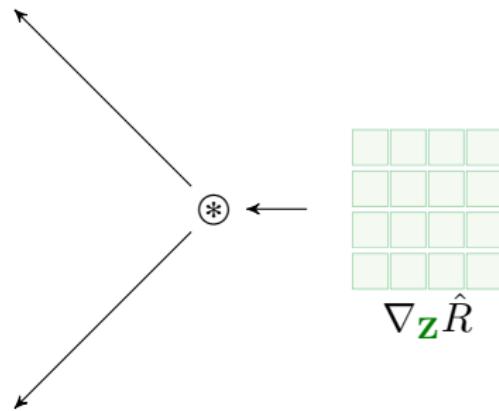
Backpropagation through 2D Convolution

Let $\mathbf{Z} = \text{Conv}(\mathbf{X}|\mathbf{W})$, where \mathbf{X} is a C -channel **input tensor**, \mathbf{W} is C -channel **filter** and \mathbf{Z} is a single-channel **output map**, i.e., a **matrix**. Assume that we know the gradient of loss \hat{R} with respect to the **output map**, i.e., $\nabla_{\mathbf{Z}} \hat{R}$. Then, the gradient of loss \hat{R} with respect to **input tensor** is

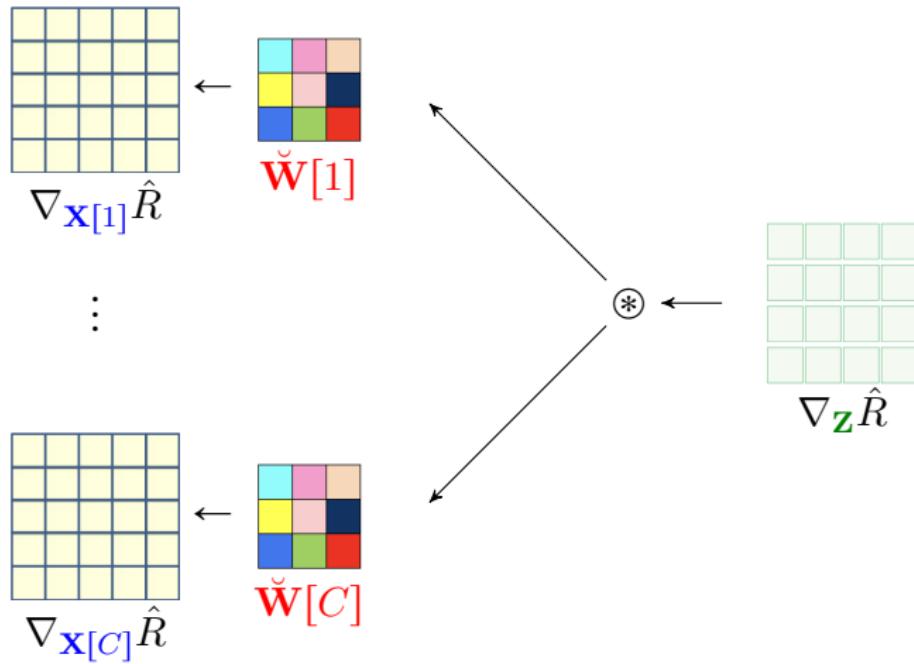
$$\nabla_{\mathbf{X}} \hat{R} = \left[\text{Conv} \left(\nabla_{\mathbf{Z}} \hat{R} | \check{\mathbf{W}}[1] \right) \dots \text{Conv} \left(\nabla_{\mathbf{Z}} \hat{R} | \check{\mathbf{W}}[C] \right) \right]$$

Note that $\nabla_{\mathbf{X}} \hat{R}$ is also a C -channel tensor

Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-channel Convolution

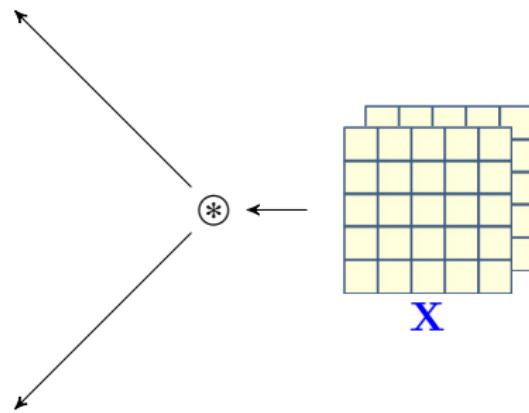
We now go for the general case: we have a C -channel input tensor \mathbf{X} and we calculate K -channel feature tensor \mathbf{Z}

- ↳ We have K filters $\mathbf{W}_1, \dots, \mathbf{W}_K$
- ↳ Each filter has C channels

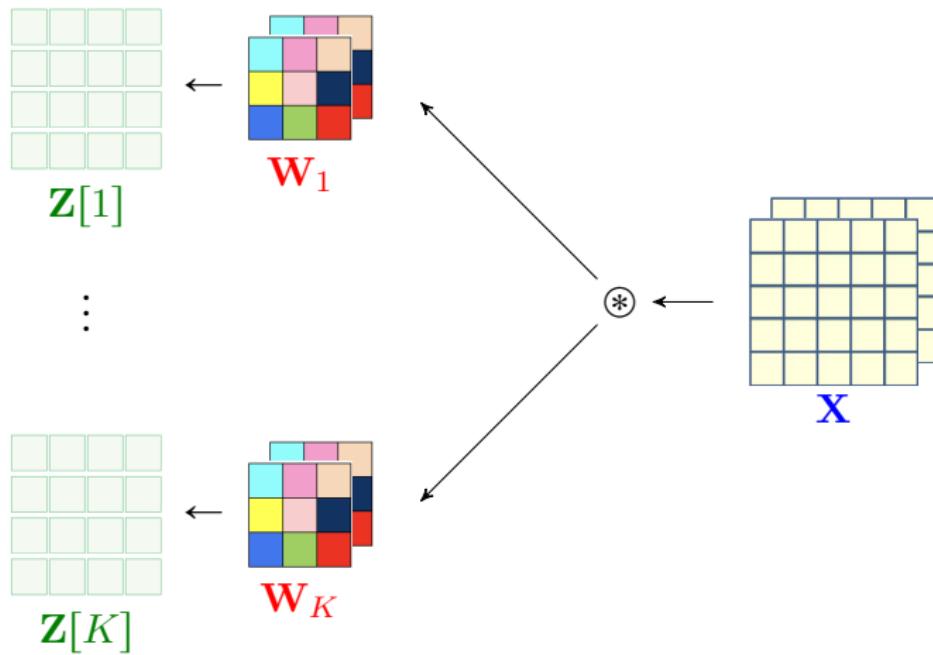
Let's denote channel k of the output by \mathbf{Z} ; then, we can write

$$\mathbf{Z}[k] = \text{Conv}(\mathbf{X} | \mathbf{W}_k)$$

Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-channel Convolution

We write the **chain rule** with a bit **cheating**: let's denote the **derivative** of an object A with respect to object B as $\nabla_B A$

- ↳ if A and B are both scalars then its simple **derivative**
- ↳ if A is a scalar and B is a vector it's **gradient**
- ↳ if A and B are both vectors then its **Jacobian**
- ↳ ...

Backpropagation: Multi-channel Convolution

We write the **chain rule** with a bit **cheating**: let's denote the **derivative** of an object A with respect to object B as $\nabla_B A$

- ↳ if A and B are both scalars then its simple **derivative**
- ↳ if A is a scalar and B is a vector it's **gradient**
- ↳ if A and B are both vectors then its **Jacobian**
- ↳ ...

When $\mathbf{Z} = f(\mathbf{X})$ and $\hat{R} = g(\mathbf{Z})$: **chain rule** says

$$\nabla_{\mathbf{X}} \hat{R} = \nabla_{\mathbf{Z}} \hat{R} \circ \nabla_{\mathbf{X}} \mathbf{Z}$$

where \circ is a kind of product

Backpropagation: Multi-channel Convolution

We write the **chain rule** with a bit **cheating**: let's denote the **derivative** of an object A with respect to object B as $\nabla_B A$

- ↳ if A and B are both scalars then its simple **derivative**
- ↳ if A is a scalar and B is a vector it's **gradient**
- ↳ if A and B are both vectors then its **Jacobian**
- ↳ ...

When $\mathbf{Z} = f(\mathbf{X})$ and $\hat{R} = g(\mathbf{Z})$: **chain rule** says

$$\nabla_{\mathbf{X}} \hat{R} = \nabla_{\mathbf{Z}} \hat{R} \circ \nabla_{\mathbf{X}} \mathbf{Z}$$

where \circ is a kind of product

We now write the **chain rule** with this simplified notation

Backpropagation: Multi-channel Convolution

Recall that all output channels are functions of the **input tensor**

$$\mathbf{Z}[1] = f_1(\mathbf{X}) \quad \dots \quad \mathbf{Z}[K] = f_K(\mathbf{X})$$

So, the **chain rule** can be written as

$$\begin{aligned}
 \nabla_{\mathbf{X}} \hat{R} &= \sum_{k=1}^K \underbrace{\nabla_{\mathbf{Z}[k]} \hat{R} \circ \nabla_{\mathbf{X}} \mathbf{Z}[k]}_{\text{what we calculated for single output map}} \\
 &= \sum_{k=1}^K \left[\text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[1] \right) \dots \text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[C] \right) \right] \\
 &= \left[\sum_{k=1}^K \text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[1] \right) \dots \sum_{k=1}^K \text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[C] \right) \right]
 \end{aligned}$$

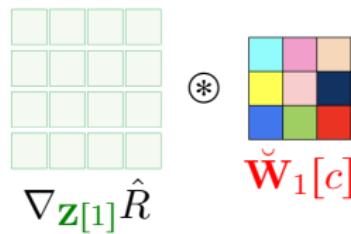
Backpropagation: Multi-Channel Convolution

The backward pass is therefore

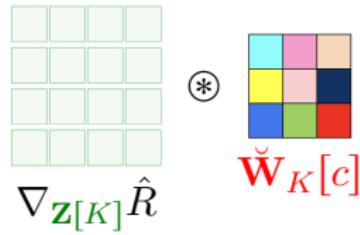
$$\nabla_{\mathbf{X}} \hat{R} = \left[\sum_{k=1}^K \text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[1] \right) \dots \sum_{k=1}^K \text{Conv} \left(\nabla_{\mathbf{Z}[k]} \hat{R} | \check{\mathbf{W}}_k[C] \right) \right]$$

Let's look at a particular input channel c

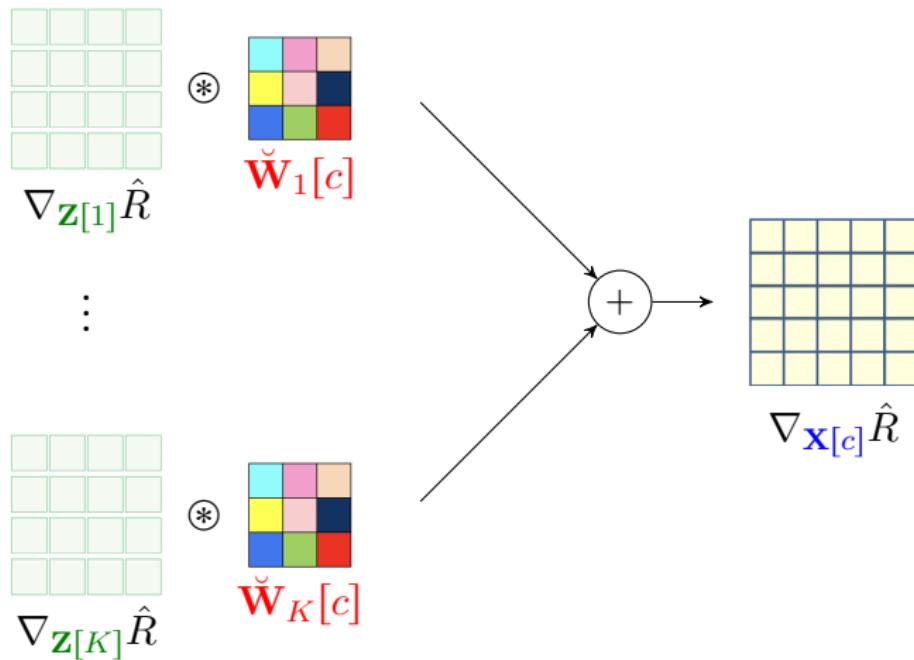
Backpropagation: Multi-Channel Convolution

$$\nabla_{\mathbf{Z}[1]} \hat{R} \quad \otimes \quad \check{\mathbf{W}}_1[c]$$


⋮

$$\nabla_{\mathbf{Z}[K]} \hat{R} \quad \otimes \quad \check{\mathbf{W}}_K[c]$$


Backpropagation: Multi-Channel Convolution



Backpropagation: Multi-Channel Convolution

This is a new multi-channel convolution: let's define K -channel filter \mathbf{W}_c^\dagger for $c = 1, \dots, C$ as follows

$$\mathbf{W}_c^\dagger = [\check{\mathbf{W}}_1[c] \dots \check{\mathbf{W}}_K[c]]$$

Backpropagation: Multi-Channel Convolution

This is a new multi-channel convolution: let's define *K-channel filter* \mathbf{W}_c^\dagger for $c = 1, \dots, C$ as follows

$$\mathbf{W}_c^\dagger = [\check{\mathbf{W}}_1[c] \dots \check{\mathbf{W}}_K[c]]$$

Then, channel c of $\nabla_{\mathbf{X}} \hat{R}$ is the convolution of $\nabla_{\mathbf{Z}} \hat{R}$ with $\mathbf{W}_1^\dagger, \dots, \mathbf{W}_C^\dagger$

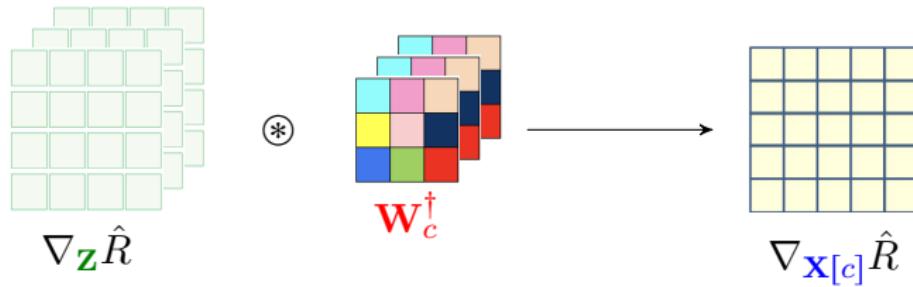
$$\nabla_{\mathbf{X}[c]} \hat{R} = \text{Conv}(\nabla_{\mathbf{Z}} \hat{R} | \mathbf{W}_c^\dagger)$$

Or shortly, we can write

$$\nabla_{\mathbf{X}} \hat{R} = \text{Conv}(\nabla_{\mathbf{Z}} \hat{R} | \mathbf{W}_1^\dagger, \dots, \mathbf{W}_C^\dagger)$$

Backpropagation: Multi-Channel Convolution

This means that we can look at channel c of $\nabla_{\mathbf{X}} \hat{R}$ as



Backpropagation: Multi-Channel Convolution

*To backpropagate to channel c of **tensor input**, we convolve the output gradient **tensor** with the K -channel **filter tensor** that is constructed by collecting the channel c of all K forward filters, each being **reversed up-to-down and right-to-left***

Backpropagation: Multi-Channel Convolution

To backpropagate to channel c of **tensor input**, we convolve the output gradient **tensor** with the K -channel **filter tensor** that is constructed by collecting the channel c of all K forward filters, each being **reversed up-to-down and right-to-left**

Backpropagation through Multi-Channel Convolution

Let $\mathbf{Z} = \text{Conv}(\mathbf{X} | \mathbf{W}_1, \dots, \mathbf{W}_K)$, where \mathbf{X} is a C -channel **input tensor**, \mathbf{W}_k is C -channel **filter** and \mathbf{Z} is a K -channel **output tensor**. Assume that we know the gradient of loss \hat{R} with respect to the **output tensor**, i.e., $\nabla_{\mathbf{Z}} \hat{R}$. Then, the gradient of loss \hat{R} with respect to **input tensor** is

$$\nabla_{\mathbf{X}} \hat{R} = \text{Conv}\left(\nabla_{\mathbf{Z}} \hat{R} | \mathbf{W}_1^\dagger, \dots, \mathbf{W}_C^\dagger\right)$$

where $\mathbf{W}_c^\dagger = [\check{\mathbf{W}}_1[c] \dots \check{\mathbf{W}}_K[c]]$ is a K -channel filter

Backpropagation through Convolution: Summary

Moral of Story

We can always backpropagate through a *convolutional layer* with C input channels and K output channels by

- ➊ multiply output gradient entry-wise with derivative of *activation function*
- ➋ convolve *output gradient* with C different *K-channel filters* computed by rearranging of the K forward filters

Backpropagation through Convolution: Summary

Moral of Story

We can always backpropagate through a **convolutional layer** with C input channels and K output channels by

- ① multiply output gradient entry-wise with derivative of **activation function**
- ② convolve **output gradient** with C different **K -channel filters** computed by **rearranging of the K forward filters**

To compute the gradient with respect to **weights of channel c in filter k**

- ↳ **convolve** channel c of input tensor with channel k of output gradient

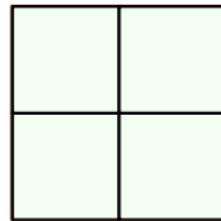
- + Sounds great! But what about cases with $\text{stride} \neq 1$?
- Well! we said we can decompose them as convolution/pooling with $\text{stride} 1 + a \text{ resampling unit}$. We just need to learn how to backpropagate through a resampling unit

Backpropagation: *Downsampling*

Let's again try an **example**: we have *partial derivatives with respect to down-sampled variables* and want to compute the *partial derivatives with respect to input variables*

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

$$\mathbf{Z} = \text{dSample}(\mathbf{X}|2)$$



Backpropagation: *Downsampling*

Let's again try an **example**: we have *partial derivatives with respect to down-sampled variables* and want to compute the *partial derivatives with respect to input variables*

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

$$\mathbf{Z} = \text{dSample}(\mathbf{X}|2)$$

~~~→

|           |  |
|-----------|--|
| $z_1=x_1$ |  |
|           |  |

## Backpropagation: *Downsampling*

Let's again try an **example**: we have *partial derivatives with respect to down-sampled variables* and want to compute the *partial derivatives with respect to input variables*

|       |       |       |
|-------|-------|-------|
| $x_1$ | $x_2$ | $x_3$ |
| $x_4$ | $x_5$ | $x_6$ |
| $x_7$ | $x_8$ | $x_9$ |

$$\mathbf{Z} = \text{dSample}(\mathbf{X}|2)$$



|           |           |
|-----------|-----------|
| $z_1=x_1$ | $z_2=x_3$ |
|           |           |

## Backpropagation: *Downsampling*

Let's again try an **example**: we have *partial derivatives with respect to down-sampled variables* and want to compute the *partial derivatives with respect to input variables*

|       |       |       |
|-------|-------|-------|
| $x_1$ | $x_2$ | $x_3$ |
| $x_4$ | $x_5$ | $x_6$ |
| $x_7$ | $x_8$ | $x_9$ |

$$\mathbf{Z} = \text{dSample}(\mathbf{X}|2)$$



|           |           |
|-----------|-----------|
| $z_1=x_1$ | $z_2=x_3$ |
| $z_3=x_7$ |           |

## Backpropagation: *Downsampling*

Let's again try an **example**: we have *partial derivatives with respect to down-sampled variables* and want to compute the *partial derivatives with respect to input variables*

|       |       |       |
|-------|-------|-------|
| $x_1$ | $x_2$ | $x_3$ |
| $x_4$ | $x_5$ | $x_6$ |
| $x_7$ | $x_8$ | $x_9$ |

$$\mathbf{Z} = \text{dSample}(\mathbf{X}|2)$$



|           |           |
|-----------|-----------|
| $z_1=x_1$ | $z_2=x_3$ |
| $z_3=x_7$ | $z_4=x_9$ |

# Backpropagation: *Downsampling*

Let's write with **chain rule**

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = 0$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = 0$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = \frac{\partial \hat{R}}{\partial z_2}$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = 0$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_4} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_4} = 0$$

⋮

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = 0$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = \frac{\partial \hat{R}}{\partial z_2}$$

$$\frac{\partial \hat{R}}{\partial x_4} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_4} = 0$$

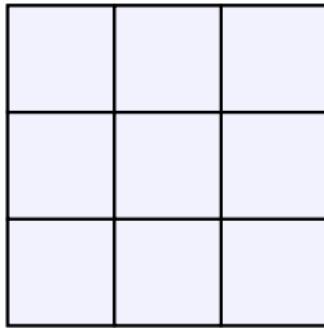
⋮

$$\frac{\partial \hat{R}}{\partial x_7} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_7} = \frac{\partial \hat{R}}{\partial z_3}$$

$$\frac{\partial \hat{R}}{\partial x_8} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_8} = 0$$

$$\frac{\partial \hat{R}}{\partial x_9} = \sum_{i=1}^4 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_9} = \frac{\partial \hat{R}}{\partial z_4}$$

# Backpropagation: *Downsampling*



|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ |
| $\frac{\partial \hat{R}}{\partial z_3}$ | $\frac{\partial \hat{R}}{\partial z_4}$ |

# Backpropagation: *Downsampling*

|                                         |   |                                         |
|-----------------------------------------|---|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | 0 | $\frac{\partial \hat{R}}{\partial z_2}$ |
| 0                                       | 0 | 0                                       |
| $\frac{\partial \hat{R}}{\partial z_3}$ | 0 | $\frac{\partial \hat{R}}{\partial z_4}$ |

|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ |
| $\frac{\partial \hat{R}}{\partial z_3}$ | $\frac{\partial \hat{R}}{\partial z_4}$ |

# Backpropagation: *Downsampling*

This is simply an *upsampling* with the same factor

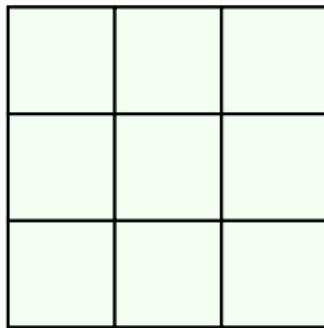
|                                         |   |                                         |
|-----------------------------------------|---|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | 0 | $\frac{\partial \hat{R}}{\partial z_2}$ |
| 0                                       | 0 | 0                                       |
| $\frac{\partial \hat{R}}{\partial z_3}$ | 0 | $\frac{\partial \hat{R}}{\partial z_4}$ |

uSample  $(\nabla_{\mathbf{z}} \hat{R} | 2)$

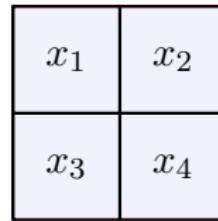
|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ |
| $\frac{\partial \hat{R}}{\partial z_3}$ | $\frac{\partial \hat{R}}{\partial z_4}$ |

## Backpropagation: *Upsampling*

Now, let's look into **upsampling**: we have *partial derivatives with respect to up-sampled variables* and want to compute the partial derivatives with respect to *input variables*



$$\mathbf{Z} = \text{uSample}(\mathbf{X}|2)$$



## Backpropagation: Upsampling

Now, let's look into **upsampling**: we have partial derivatives with respect to **up-sampled variables** and want to compute the partial derivatives with respect to **input variables**

|           |         |           |
|-----------|---------|-----------|
| $z_7=x_3$ | $z_8=0$ | $z_9=x_4$ |
| $z_4=0$   | $z_5=0$ | $z_6=0$   |
| $z_1=x_1$ | $z_2=0$ | $z_3=x_2$ |

$$\mathbf{Z} = \text{uSample}(\mathbf{X}|2)$$



|       |       |
|-------|-------|
| $x_1$ | $x_2$ |
| $x_3$ | $x_4$ |

# Backpropagation: *Downsampling*

Let's write with **chain rule**

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = \frac{\partial \hat{R}}{\partial z_3}$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = \frac{\partial \hat{R}}{\partial z_3}$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = \frac{\partial \hat{R}}{\partial z_7}$$

# Backpropagation: *Downsampling*

Let's write with **chain rule**

$$\frac{\partial \hat{R}}{\partial x_1} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_1} = \frac{\partial \hat{R}}{\partial z_1}$$

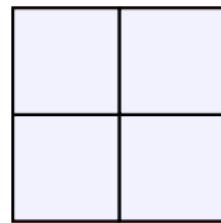
$$\frac{\partial \hat{R}}{\partial x_2} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_2} = \frac{\partial \hat{R}}{\partial z_3}$$

$$\frac{\partial \hat{R}}{\partial x_3} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_3} = \frac{\partial \hat{R}}{\partial z_7}$$

$$\frac{\partial \hat{R}}{\partial x_4} = \sum_{i=1}^9 \frac{\partial \hat{R}}{\partial z_i} \frac{\partial z_i}{\partial x_4} = \frac{\partial \hat{R}}{\partial z_9}$$

# Backpropagation: *Upsampling*

|                                         |                                         |                                         |
|-----------------------------------------|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ | $\frac{\partial \hat{R}}{\partial z_3}$ |
| $\frac{\partial \hat{R}}{\partial z_4}$ | $\frac{\partial \hat{R}}{\partial z_5}$ | $\frac{\partial \hat{R}}{\partial z_6}$ |
| $\frac{\partial \hat{R}}{\partial z_7}$ | $\frac{\partial \hat{R}}{\partial z_8}$ | $\frac{\partial \hat{R}}{\partial z_9}$ |



# Backpropagation: *Upsampling*

|                                         |                                         |                                         |
|-----------------------------------------|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ | $\frac{\partial \hat{R}}{\partial z_3}$ |
| $\frac{\partial \hat{R}}{\partial z_4}$ | $\frac{\partial \hat{R}}{\partial z_5}$ | $\frac{\partial \hat{R}}{\partial z_6}$ |
| $\frac{\partial \hat{R}}{\partial z_7}$ | $\frac{\partial \hat{R}}{\partial z_8}$ | $\frac{\partial \hat{R}}{\partial z_9}$ |

|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_3}$ |
| $\frac{\partial \hat{R}}{\partial z_7}$ | $\frac{\partial \hat{R}}{\partial z_9}$ |

# Backpropagation: *Upsampling*

This is *downsampling* with the same factor

|                                         |                                         |                                         |
|-----------------------------------------|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_2}$ | $\frac{\partial \hat{R}}{\partial z_3}$ |
| $\frac{\partial \hat{R}}{\partial z_4}$ | $\frac{\partial \hat{R}}{\partial z_5}$ | $\frac{\partial \hat{R}}{\partial z_6}$ |
| $\frac{\partial \hat{R}}{\partial z_7}$ | $\frac{\partial \hat{R}}{\partial z_8}$ | $\frac{\partial \hat{R}}{\partial z_9}$ |

dSample  $\left( \nabla_{\mathbf{z}} \hat{R} | 2 \right)$

|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| $\frac{\partial \hat{R}}{\partial z_1}$ | $\frac{\partial \hat{R}}{\partial z_3}$ |
| $\frac{\partial \hat{R}}{\partial z_7}$ | $\frac{\partial \hat{R}}{\partial z_9}$ |

# Backpropagation: Resampling

## Backpropagation through Downsampling

To backpropagate through a downsampling unit with factor (stride)  $S$  we up-sample the output gradient with factor  $S$

## Backpropagation through Upsampling

To backpropagate through an upsampling unit with factor (stride)  $S$  we down-sample the output gradient with factor  $S$

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

*In forward pass we do*

- **forward** FNN

*In backward pass we do*

- **backward** FNN

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

*In forward pass we do*

- **forward FNN**
- **pooling**

*In backward pass we do*

- **backward FNN**
- **backward pooling** ~ convolution

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

*In forward pass we do*

- **forward FNN**
- **pooling**
- **convolution**

*In backward pass we do*

- **backward FNN**
- **backward pooling ~ convolution**
- **convolution with reversed filters**

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

*In forward pass we do*

- **forward FNN**
- **pooling**
- **convolution**
- **upsampling**

*In backward pass we do*

- **backward FNN**
- **backward pooling ~ convolution**
- **convolution with reversed filters**
- **downsampling**

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

*In forward pass we do*

- *forward FNN*
- *pooling*
- *convolution*
- *upsampling*
- *downsampling*

*In backward pass we do*

- *backward FNN*
- *backward pooling ~ convolution*
- *convolution with reversed filters*
- *downsampling*
- *upsampling*

# Forward and Backpropagation in CNNs: Summary

To each **forward** action, there is a **backward** counterpart

In **forward** pass we do

- **forward FNN**
- **pooling**
- **convolution**
- **upsampling**
- **downsampling**

In **backward** pass we do

- **backward FNN**
- **backward pooling** ~ convolution
- convolution with **reversed filters**
- **downsampling**
- **upsampling**

Once we over with **backward** pass, we compute **all required gradients** from **forward** and **backward** variables

# We Use Advanced Techniques

For FNN, we looked into advanced techniques such as

- *Dropout and Regularization*
  - ↳ to reduce the impact of *overfitting*
- *Input Normalization*
  - ↳ to improve training with *unbalanced* datasets
- *Batch Normalization*
  - ↳ to make the training more stable against *feature variations*
- *Data Preprocessing*
  - ↳ to *clean* our training dataset

---

Same goes with CNNs

*we should use all **these methods** for same purposes in CNNs*

# Other Forms of Convolution

The convolution operation we considered in this chapter is often called

**2D Convolution**

*because it screens **input** in a **2D** fashion, i.e., **left-to-right** and **up-to-down***

## Other Forms of Convolution

The convolution operation we considered in this chapter is often called

**2D Convolution**

*because it screens **input** in a **2D** fashion, i.e., **left-to-right** and **up-to-down***

---

**2D convolution** is the **most popular form** of convolution; however,

# Other Forms of Convolution

The convolution operation we considered in this chapter is often called

## 2D Convolution

because it screens *input* in a 2D fashion, i.e., *left-to-right* and *up-to-down*

---

2D convolution is the *most popular form* of convolution; however,

- we could have *1D convolutions*
  - ↳ The filter *only slides in one direction*, i.e., *left-to-right* or *up-to-down*
  - ↳ This is useful with *audio data* where we slide the filter over *time*

# Other Forms of Convolution

The convolution operation we considered in this chapter is often called

## 2D Convolution

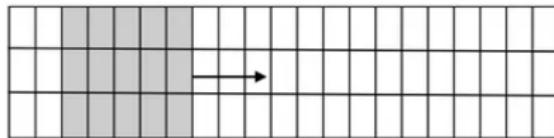
because it screens *input* in a 2D fashion, i.e., *left-to-right* and *up-to-down*

---

2D convolution is the *most popular form* of convolution; however,

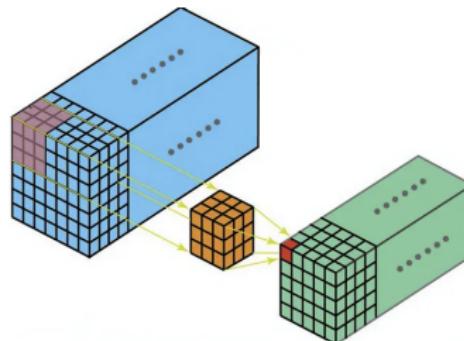
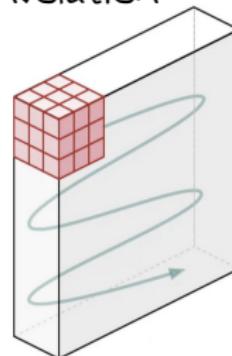
- we could have *1D convolutions*
  - ↳ The filter *only slides in one direction*, i.e., *left-to-right* or *up-to-down*
  - ↳ This is useful with *audio data* where we slide the filter over *time*
- we could also have *3D convolutions*
  - ↳ The filter slides in *all three directions*, i.e., *left-to-right*, *up-to-down* and *front-to-back*
  - ↳ This is useful with *3D images*, e.g., *3D medical image of brain*

# Other Forms of Convolution



1D Convolution

2D Convolution



3D Convolution

There are also **other forms**, e.g., depth-wise convolution

at the end of day, they all **slide** over **input** with **some filter**