

ECE 1508: Applied Deep Learning

Chapter 4: Convolutional Neural Networks

Ali Bereyhi

`ali.bereyhi@utoronto.ca`

Department of Electrical and Computer Engineering
University of Toronto

Winter 2025

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

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

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 $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

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

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

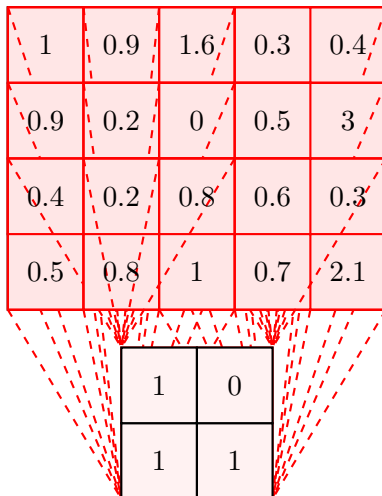
If we convolve with stride $S = 1$

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

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



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

Convolution: 2D Arrays

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

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

Diagram illustrating the convolution operation. The input matrix \mathbf{X} is a 4x5 grid. A 2x2 kernel is applied to a 2x2 region of \mathbf{X} (indicated by blue boxes and dashed lines). The elements $X_{1,1}$ and $X_{2,1}$ are highlighted in blue. The kernel \mathbf{W} is a 2x2 matrix. The operation is represented by \odot (element-wise multiplication) and $\rightarrow \text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$.

Now, we convolve with stride $S = 2$

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

Diagram illustrating the resulting output matrix \mathbf{Z} (green) after convolution with stride $S = 2$. An arrow points from the $\text{sum}(\mathbf{X}_{1,1} \odot \mathbf{W})$ result to the $Z_{1,1}$ element in \mathbf{Z} .

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*

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

1	0
1	1

2.1	2.1
1.7	2.5

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

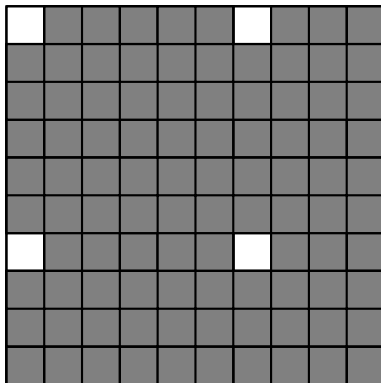
This is *downsampling* with factor 2!

Downsampling

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

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

Convolution: *Downsampling*



Downsampling with **factor 2** $\text{dSample}(\mathbf{Z}|2)$ *Downsampling* with **factor 3**
 $\text{dSample}(\mathbf{Z}|3)$

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

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*

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

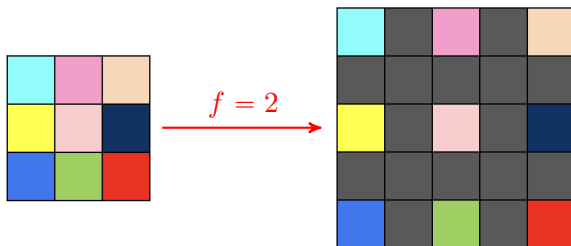
Whenever we need to convolve with *stride $S > 1$*

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



Upsampling

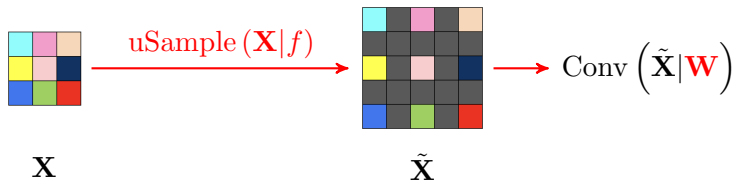
Upsampling with factor f adds $f - 1$ rows and columns of **zeros** after *every row and column*

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

Convolution with Fractional Stride

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

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

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

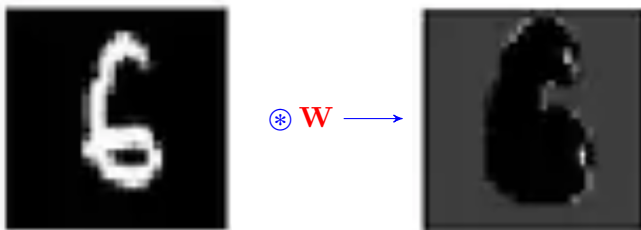
The above interpretation of **stride** helps a lot in **backpropagation!**

Convolution: *Padding*

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

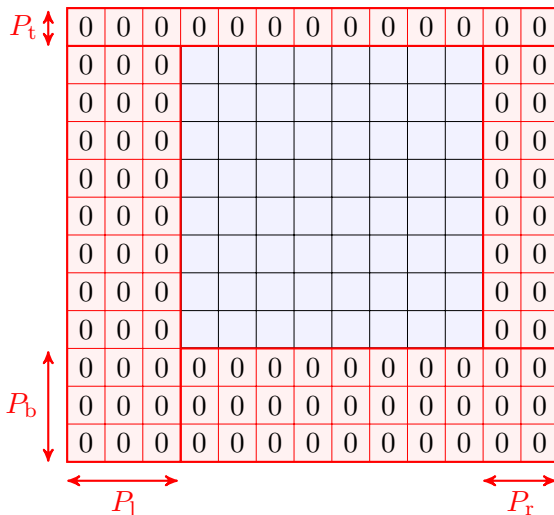
- + Why should we be interested in **changing dimensions of \mathbf{Z}** ?
- We'll see **multiple** reasons: a simple one is that we may like to have **a same-size feature map** to sketch it **as an image** and compare it to **input**

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



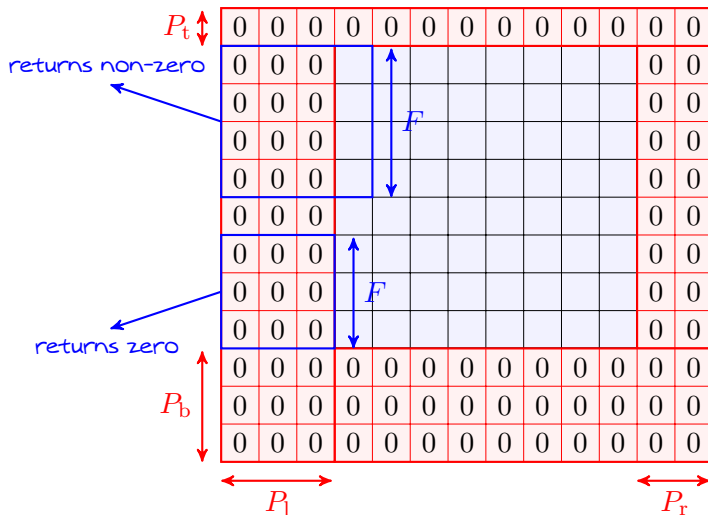
Convolution: Zero-Padding

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



Convolution: Zero-Padding

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., $P_b = P_t = P_r = P_l$

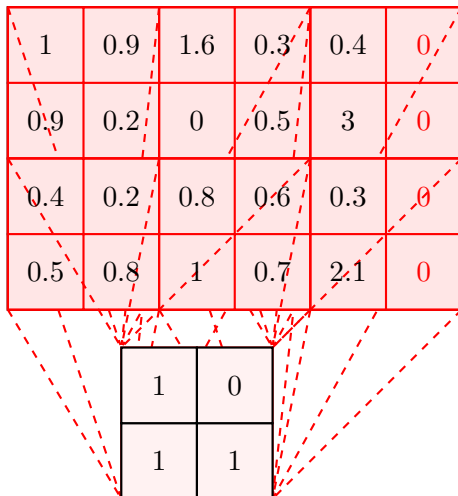
As we see the dimensions of **feature map** is a function of

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

It is thus typical that we get some of these 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*



2.1	2.1	3.4
1.7	2.5	2.4

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:

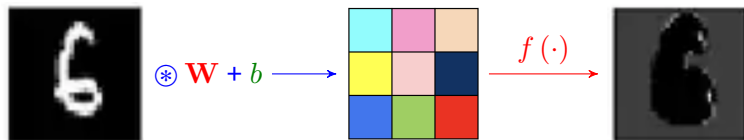
- 1 it first applies *linear convolution* to find \mathbf{Z}

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

- 2 it then *activates* \mathbf{Z}

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

Convolution: Activation

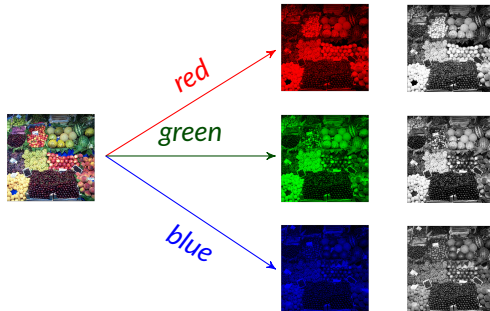


- + We did not discuss *bias* before!
- It's just a *scalar* added to *all entries*
- + What kinds of *activation* are used in CNNs?
- Similar to FNNs with *ReLU* being the *popular one*

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

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*



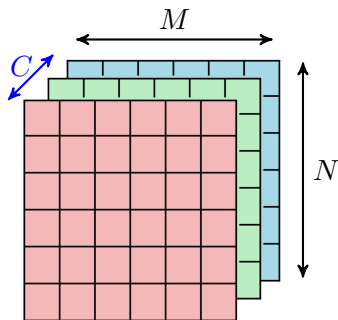
Recap: 3D Tensors

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 $N \times M$

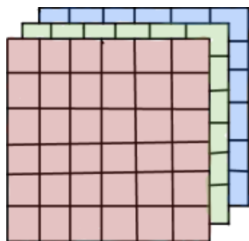
For instance, an $N \times M$ pixel RGB image is tensor in $\mathbb{R}^{3 \times N \times M}$



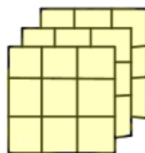
Convolution: 3D Arrays

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

Let's try a visual example: we want to convolve RGB image with filter **W**



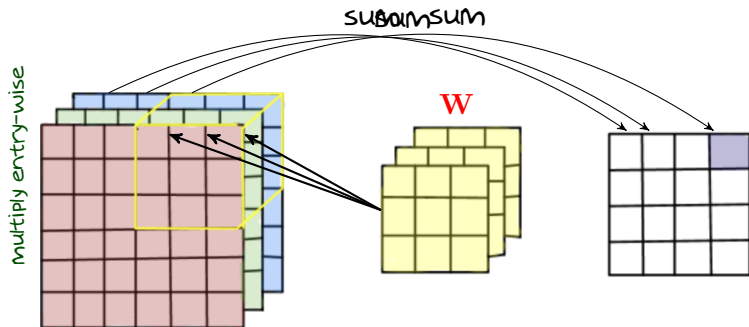
convolve with



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: 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*

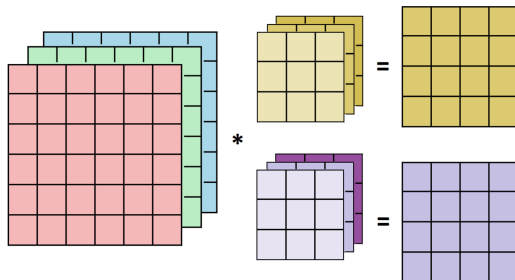
whenever needed we can

- 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**

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 $f(\cdot)$. It transforms the C -channel input \mathbf{X} to the activated K -channel feature tensor \mathbf{Y} as follows:

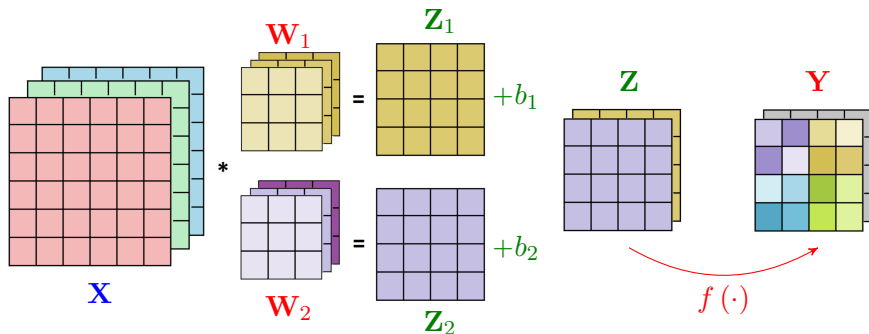
- 1 it finds the feature tensor \mathbf{Z}

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

- 2 it then activates \mathbf{Z}

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

Multi-Channel Convolution Layer: Visualization

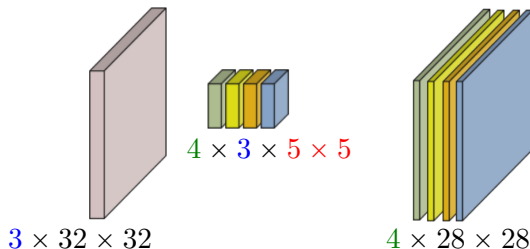


There are few points that we should keep in mind

- **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



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

- We have $K = 4$ **kernels** \rightsquigarrow **feature tensor** has 4 channels
- **Kernels** have width $F = 5$ \rightsquigarrow we see that $32 - 5 + 1 = 28$
 - \hookrightarrow **stride** is one, i.e., $S = 1$
 - \hookrightarrow **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**

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**

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

$\max \{ \mathbf{Y}_{1,1} \mathbf{Y}_{1,2} \mathbf{Y}_{i,j} \}$

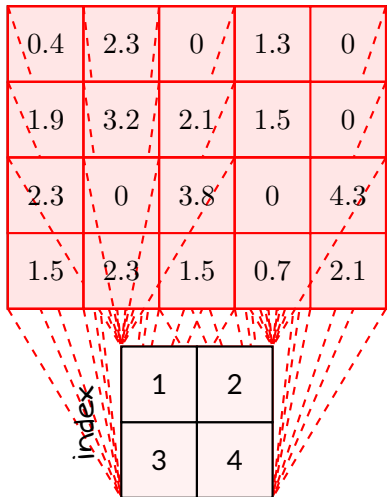
In each window, we pool the **maximum**

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

Attention

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

Max Pooling: Numerical Example



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

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

index

Mean-Pooling

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

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

A 2×2 window is highlighted in blue, containing elements $Y_{1,1}$, $Y_{1,2}$, $Y_{2,1}$, and $Y_{2,2}$. A dashed blue arrow points from this window to the expression $\text{mean} \{ \mathbf{Y}_{1,1} \mathbf{Y}_{1,2} \mathbf{Y}_{i,j} \}$.

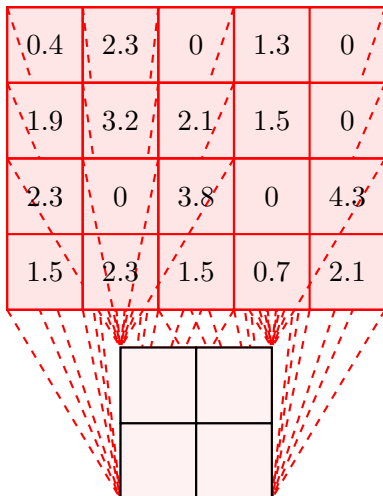
In each window, we pool the **average**

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

Two arrows point from the 2×2 window in the feature map \mathbf{Y} to the first two elements of the first row in the pooled feature map $\hat{\mathbf{Y}}$.

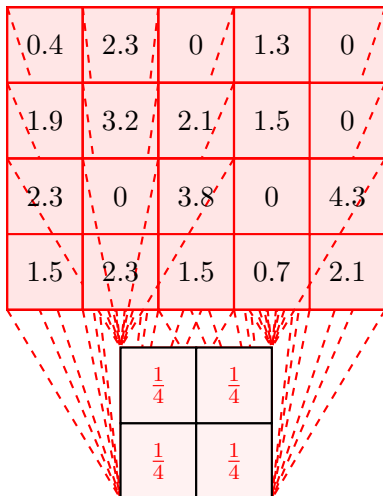
We can look at mean-pooling as a **convolution** with **uniform kernel**

Mean-Pooling: Numerical Example



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

Mean-Pooling: Numerical Example



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 \equiv convolution with uniform kernel

Advanced Pooling: Use a General Function

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

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

Note: In the original image, a blue box highlights the top-left 2x2 submatrix of Y, with blue labels Y_{1,1} and Y_{1,2} next to the first two columns. Dashed lines extend from this box to the right, indicating the general function Π(·) applied to the entire matrix.

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

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

Note: Two arrows point from the text 'Typical functions are norms...' to the first two columns of the matrix Y-hat, indicating the result of applying a norm function to the columns of Y.

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

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

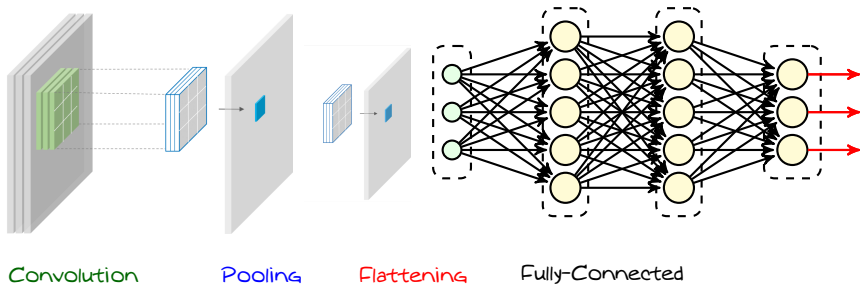
- + *Do we always pool after convolution?*
- Not really! In many architectures pooling is applied every couple of convolutional layers

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

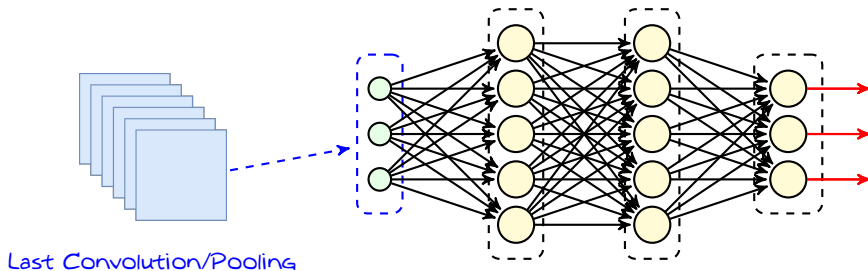


Once we are over with convolution and pooling we *flatten* final *feature tensor*

Flattening

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

Flattening



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*