Machine Learning and Optimization

Lecture 15: Convolutional Neural Networks (CNNs)

CSCI 4140/6960 — Spring 2026

Course website: `cs.rpi.edu/~gittea/.../mlandopt.html`

Learning Objectives

By the end of this lecture, you should be able to:

1. **State** the inductive biases encoded by CNNs (locality, translation equivariance, parameter efficiency) and explain why MLPs fail on images
2. **Compute** the output of a 2D convolution given an image and a kernel
3. **Apply** the output-size formula to determine spatial dimensions after convolution with given padding and stride
4. **Describe** multichannel convolution and explain how stacked filters produce feature hierarchies
5. **Distinguish** max pooling from strided convolution and state the tradeoffs
6. **Trace** spatial dimensions and parameter counts through a CNN architecture (LeNet-5)
7. **Explain** why a CNN requires fixed-size inputs, and how fully convolutional networks remove this constraint via 1×1 convolutions and GAP
8. **Express** a convolutional layer as a structured matrix multiplication (im2col) and explain why MLP backprop theory transfers directly
9. **Derive** $\nabla_K L$ and $\nabla_I L$ through a convolutional layer using the chain rule (Appendix A; CSCI 6960 students are expected to work through these)

Lecture Roadmap

The Problem with MLPs on Images

The Big Picture: CNNs

Convolution: Mechanics

Convolution: Output Geometry

Pooling

Convolution as a Structured Linear Map

CNN Architectures

Case Studies

Recap: Feature Maps for Image Classification

Key to good performance: choice of **feature map**

$$\phi\colon \mathbb{R}^{d_1 \times d_2} \to \mathbb{R}^D \qquad \text{(implemented as a NN)}$$

Then classify using logistic regression on $\phi(x)$.

So far, we have used:

1. ϕ from **flattening** the image into a vector
2. ϕ from an **autoencoder** (use encoder as feature map)
3. ϕ from an **MLP**

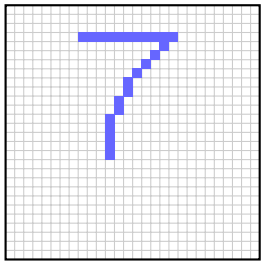
These feature maps can be *learned* on one dataset and reused on others.

Problem: none of these exploit the *spatial structure* of images.

The Problem with MLPs on Images

Running example: classifying 28×28 grayscale digit images (MNIST).

An MLP treats the image as a flat vector $x \in \mathbb{R}^{784}$.



28×28 image

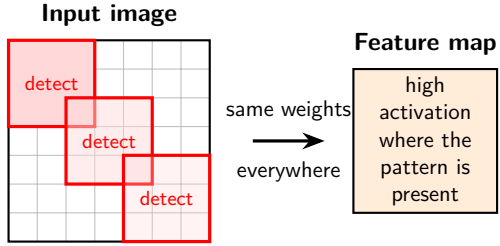
Problems with the MLP approach:

- A single FC layer from 784 inputs to 300 hidden units needs $784 \times 300 + 300 = 235{,}500$ parameters
- **No spatial structure**: pixel $(3, 4)$ and pixel $(27, 1)$ are treated identically
- An edge pattern learned in one location must be re-learned in every other location

Observation: images have **local spatial structure**—nearby pixels are strongly correlated, and the same patterns (edges, strokes) appear at many locations.

The Idea: Local Shared Feature Detectors

Key insight: instead of connecting every pixel to every hidden unit, use small *local* detectors that slide across the image.



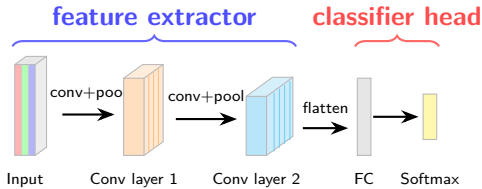
Two principles:

1. **Locality:** each detector looks at a small $k \times k$ patch, not the whole image.
2. **Weight sharing:** the *same* detector is applied at every spatial position.

The mathematical operation implementing this idea is **convolution**.

The Big Picture: Convolutional Neural Networks

A CNN stacks local feature detectors to build increasingly abstract representations.



Channels:

- Input: $C = 3$ (R, G, B)
- After each conv layer: C = number of filters
- Each channel = one **feature map**: a spatial pattern of activations

As depth increases:

- Spatial resolution $\downarrow$, channel count $\uparrow$
- Edges $\rightarrow$ textures $\rightarrow$ parts $\rightarrow$ objects
- Abstraction $\uparrow$, spatial detail $\downarrow$

Images as Tensors: Channels and Feature Maps

An image is a **tensor** of shape $H \times W \times C$:

- **Grayscale:** $C = 1$ (one intensity map)
- **RGB:** $C = 3$ (red, green, blue channels stacked)
- **After a conv layer:** $C =$ number of filters applied

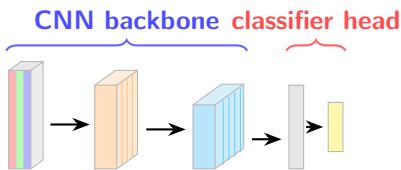
Mental model for CNN layers:

- Each **channel** in the input is one spatial map of values.
- A convolutional layer produces **new channels**, one per filter.
- Each output channel is a **feature map**: high values indicate where that filter's pattern is present.

Progression through a CNN: spatial resolution $\downarrow$, channel count $\uparrow$, abstraction $\uparrow$.
Early channels detect edges; middle channels detect textures and parts; late channels detect objects.

Transfer Learning

Early conv layers learn universal features—edges, textures, color gradients—useful for *any* vision task. Rather than training from scratch (GPU-weeks on ImageNet), reuse a pretrained **CNN backbone** and adapt only what is task-specific.



	Backbone	Head
Feat. extraction	all frozen	new
Fine-tuning	adapted	new

Feat. extraction: few labels; fast.

Fine-tuning: better accuracy when target task differs substantially from pretraining.

Why it works: low-level detectors are task-universal; only high-level abstractions need re-learning.

Inductive Biases of Convolution

CNNs are a **structured parameterization** of feature maps that encode strong priors about spatial data.

- **Locality:** each output depends only on a small $k_1 \times k_2$ patch of the input.

Inductive Biases of Convolution

CNNs are a **structured parameterization** of feature maps that encode strong priors about spatial data.

- **Locality:** each output depends only on a small $k_1 \times k_2$ patch of the input.
- **Translation equivariance:** the same filter is applied everywhere, so shifting the input shifts the output by the same amount: $(T_s I) * K = T_s(I * K)$. (Pooling adds approximate *invariance*.)

Inductive Biases of Convolution

CNNs are a **structured parameterization** of feature maps that encode strong priors about spatial data.

- **Locality:** each output depends only on a small $k_1 \times k_2$ patch of the input.
- **Translation equivariance:** the same filter is applied everywhere, so shifting the input shifts the output by the same amount: $(T_s I) * K = T_s(I * K)$. (Pooling adds approximate *invariance*.)
- **Parameter efficiency:** $k_1 k_2$ parameters per filter vs. $d_1 d_2 \times d_1' d_2'$ for a fully-connected layer.

Inductive Biases of Convolution

CNNs are a **structured parameterization** of feature maps that encode strong priors about spatial data.

- **Locality:** each output depends only on a small $k_1 \times k_2$ patch of the input.
- **Translation equivariance:** the same filter is applied everywhere, so shifting the input shifts the output by the same amount: $(T_s I) * K = T_s(I * K)$. (Pooling adds approximate *invariance*.)
- **Parameter efficiency:** $k_1 k_2$ parameters per filter vs. $d_1 d_2 \times d_1' d_2'$ for a fully-connected layer.
- **Easier optimization:** parameter sharing means each kernel entry receives gradients from *all* spatial positions $\Rightarrow$ better signal-to-noise ratio per parameter.

Biological motivation: cells in the primary visual cortex (V1) respond to oriented edges in small local receptive fields (Hubel & Wiesel, 1962).

Bottom line: CNNs dramatically reduce parameters while exploiting the spatial structure that MLPs ignore.

What Do Filters Detect?

Before defining convolution formally, consider what a local detector does:

A filter slides across the image; at each position, it computes a weighted sum of the patch.

Sobel G_x

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

detects vertical edges

Sobel G_y

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

detects horizontal edges

Laplacian

-1	0	-1
0	4	0
-1	0	-1

detects blobs/corners

Gaussian

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

blurs (smooths noise)

CNNs learn the filter entries from data via backprop—the network discovers which local patterns are task-relevant, rather than using hand-crafted kernels like those above.

Why these specific kernels detect the stated features: see the appendix.

Convolution (Cross-Correlation) — Definition

Intuition: extract every $k_1 \times k_2$ patch of the image, compute a weighted sum using the kernel, and record the result.

Convention. We index the kernel with its center at $(0, 0)$, so offsets $p \in \{-m, \ldots, m\}$ and $q \in \{-n, \ldots, n\}$ where $m = \lfloor k_1/2 \rfloor$, $n = \lfloor k_2/2 \rfloor$:

$$(I * K)_{i,j} = \sum_{p=-m}^{m} \sum_{q=-n}^{n} K_{p,q}\, I_{i+p,\, j+q}$$

Output pixel (i, j) is the weighted sum of the patch *centered* on $I_{i,j}$.

Note: true convolution flips K (i.e. uses $K_{-p,-q}$), but since K is learned, flipping is irrelevant. All major frameworks (PyTorch, TensorFlow) implement cross-correlation and call it `Conv2d`. (See Appendix B for the equivalent 0-indexed implementation form.)

A basic CNN layer returns $\sigma(I * K + b)$ for activation σ and bias $b \in \mathbb{R}$.

Convolution — Computing One Output Entry

Running example: apply the Laplacian kernel to a 5×5 image.

Image I (5×5)

1	2	1	0	0
2	1	1	2	0
0	0	1	2	1
2	1	1	1	0
0	1	2	0	2

$\odot$

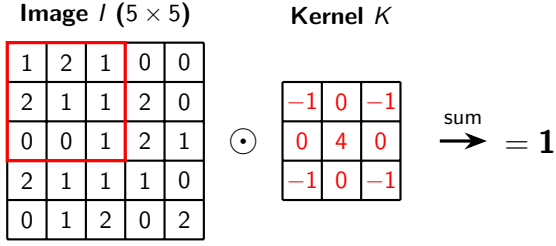
Kernel K

-1	0	-1
0	4	0
-1	0	-1

$\xrightarrow{\text{sum}} = \mathbf{1}$

Convolution — Computing One Output Entry

Running example: apply the Laplacian kernel to a 5×5 image.

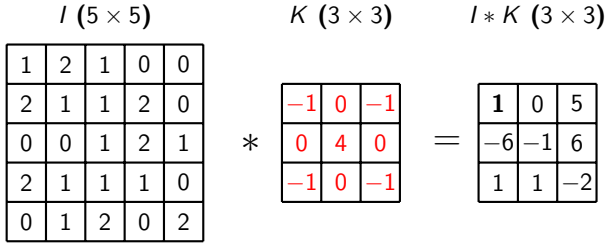


Computation of $(I * K)_{1,1}$ (output centered on input position $(1,1)$, the top-left valid position):

$$(I * K)_{1,1} = \underbrace{(-1)(1) + (0)(2) + (-1)(1)}_{-2} + \underbrace{(0)(2) + (4)(1) + (0)(1)}_{4}$$
$$+ \underbrace{(-1)(0) + (0)(0) + (-1)(1)}_{-1} = \mathbf{1}$$

Convolution — The Full Feature Map

Slide the kernel over *every* valid position to produce the output feature map.

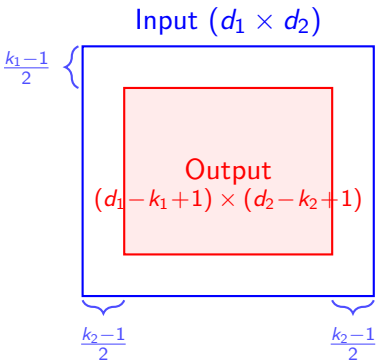


Output size (stride 1, no padding): valid positions $i \in \{m, \ldots, d_1 - 1 - m\}$, $j \in \{n, \ldots, d_2 - 1 - n\}$, giving

$$I * K \in \mathbb{R}^{(d_1 - k_1 + 1) \times (d_2 - k_2 + 1)}$$

Interpretation: the output is a **feature map** indicating where the kernel's pattern is present. Large positive values $\Rightarrow$ strong match between the patch and the kernel.

Output Size — Geometric View



The kernel cannot be centered on boundary pixels, so the output shrinks by $\frac{k_i-1}{2}$ on each side.

Example: 5×5 input, 3×3 kernel:

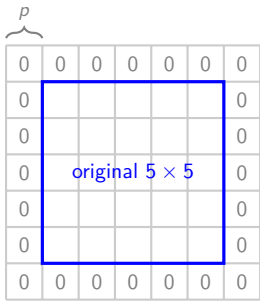
$$(5-3+1) \times (5-3+1) = 3 \times 3$$

Problem: repeated convolutions rapidly shrink the spatial dimensions.

$\Longrightarrow$ **Padding** (next slide).

Padding

Idea: pad the input with zeros around the border before convolving.
Effect: controls the spatial size of the output.



With padding p on each side, the input becomes $(d_1 + 2p) \times (d_2 + 2p)$.

Output size with padding p, stride 1:

$$(d_1 + 2p - k_1 + 1) \times (d_2 + 2p - k_2 + 1)$$

"Same" padding: choose $p = \frac{k-1}{2}$ so output size = input size.

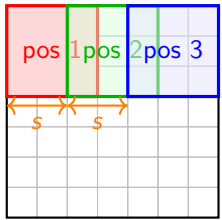
"Valid" padding: $p = 0$ (no padding).

Example: 5×5 input, 3×3 kernel, $p = 1$:

$$(5 + 2 - 3 + 1) \times (5 + 2 - 3 + 1) = 5 \times 5$$

Stride

Stride s: evaluate the filter at every s-th position instead of every position.
Effect: downsamples the spatial dimensions.



stride $s = 2$, kernel 3×3

General output size (padding p, stride s):

$$\left\lfloor \frac{d_1 + 2p - k_1}{s} + 1 \right\rfloor \times \left\lfloor \frac{d_2 + 2p - k_2}{s} + 1 \right\rfloor$$

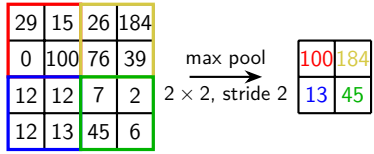
Summary:

padding	controls spatial size
stride	controls downsampling

Note: stride > 1 is an alternative to pooling for spatial downsampling. Modern architectures sometimes prefer strided convolutions over pooling.

Pooling

Pooling reduces spatial dimensions by aggregating locally. Typical types: **max pooling**, **average pooling**.



Why pool?

- Decreases computation downstream
- **Approximate translation invariance:** small shifts in input ⇒ same pooled activation
- No learnable parameters

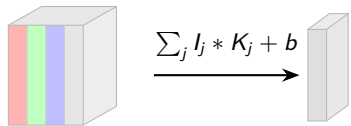
Pooling vs. strided convolution:

- Pooling has no parameters
- Strided conv is learnable
- Modern trend: strided conv often replaces pooling

Special case — global average pooling (GAP): pool each $H \times W$ channel to its spatial mean, collapsing $H \times W \times C \to C$ with no parameters. Revisited in FCNs as a replacement for flatten + FC.

Multichannel Convolution

Multichannel inputs arise from RGB images or from multiple filters in the previous layer.



Given an m-channel input $\{I_1, \ldots, I_m\}$, one convolutional unit computes:

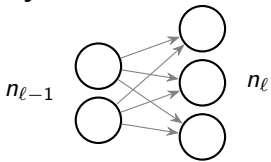
$$\sigma\Big(\sum_{j=1}^{m} I_j * K_j + b\Big)$$

Each K_j is $k_1 \times k_2$. To produce *multiple* output channels, use multiple sets of kernels.

Tensor shapes: input $H \times W \times m$, kernel $k_1 \times k_2 \times m$, one output channel $H' \times W'$.

MLP vs. CNN: Layer Formulation

MLP layer ℓ

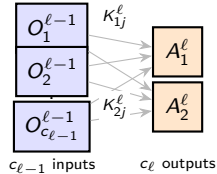


$$a^\ell = W^\ell o^{\ell-1} + b^\ell, \quad o^\ell = \sigma(a^\ell)$$

Parameters: $n_{\ell-1} n_\ell + n_\ell$

Scaling: $\mathcal{O}(HW \cdot HW)$ for images

Conv layer ℓ



Pre-activation and output for channel i:

$$A_i^\ell = \sum_{j=1}^{c_{\ell-1}} O_j^{\ell-1} * K_{ij}^\ell + b_i^\ell, \qquad O_i^\ell = \sigma(A_i^\ell)$$

- $c_{\ell-1}$ input channels, c_ℓ output channels
- Each K_{ij}^ℓ is $k_1 \times k_2$; each $b_i^\ell \in \mathbb{R}$
- **Parameters:** $c_{\ell-1}\, c_\ell\, k_1 k_2 + c_\ell$
- **Scaling:** $\mathcal{O}(k^2)$ — *independent of spatial dimensions*

Convolution as a Linear Map

Key structural observation: convolution is a *linear* operation — it can be written as multiplication by a structured matrix.

1D example: input $x \in \mathbb{R}^4$, kernel (k_0, k_1, k_2), stride 1, no padding.

$$x * k \;=\; \begin{pmatrix} k_0 & k_1 & k_2 & 0 \\ 0 & k_0 & k_1 & k_2 \end{pmatrix} x \;=:\; T\,x, \qquad T \in \mathbb{R}^{2\times 4}$$

T is a **Toeplitz matrix**: the same kernel entries appear in each row, shifted by one position.

Convolution as a Linear Map

Key structural observation: convolution is a *linear* operation — it can be written as multiplication by a structured matrix.

1D example: input $x \in \mathbb{R}^4$, kernel (k_0, k_1, k_2), stride 1, no padding.

$$x * k \;=\; \begin{pmatrix} k_0 & k_1 & k_2 & 0 \\ 0 & k_0 & k_1 & k_2 \end{pmatrix} x \;=:\; T\,x, \qquad T \in \mathbb{R}^{2\times 4}$$

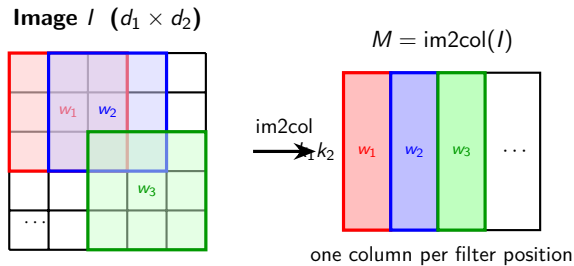
T is a **Toeplitz matrix**: the same kernel entries appear in each row, shifted by one position.

What this means:

- **Weight sharing** in matrix language: T has 3 free parameters, not $3 \times 5 = 15$ (a dense layer).
- **Backprop through a conv layer** $\equiv$ multiply by $T^\top$: no new theory beyond MLPs.
- The 2D multichannel generalization — the **im2col** operation — works the same way.

The im2col Operation

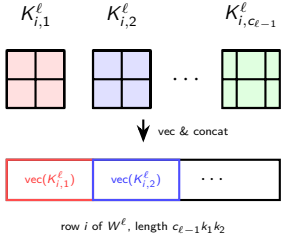
im2col maps an image to a matrix by unrolling every $k_1 \times k_2$ patch into a column.



- Each column: one patch of I unrolled into a vector of length $k_1 k_2$.
- Convolution output for one kernel K: $\quad \text{vec}(I * K)^\top = \text{vec}(K)^\top \cdot M.$

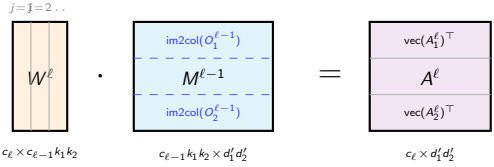
Convolution as Matrix Multiplication: W^ℓ and $M^{\ell-1}$

Forming W^ℓ: vectorize each kernel; stack into rows.



Stack c_ℓ rows: $W^\ell \in \mathbb{R}^{c_\ell \times c_{\ell-1}k_1k_2}$

Forming $M^{\ell-1}$: stack im2col of each input channel vertically.



$O^\ell = \sigma(A^\ell)$; row i of A^ℓ reshaped to $d_1' \times d_2'$ is channel i's pre-activation map.

A CNN Layer Is an MLP Layer on im2col-Transformed Input

MLP layer ℓ:

$$a^\ell = W^\ell \, o^{\ell-1} + b^\ell, \qquad o^\ell = \sigma(a^\ell)$$

CNN layer ℓ:

$$A^\ell = W^\ell \, M^{\ell-1} + b^\ell, \qquad O^\ell = \sigma(A^\ell)$$

The *only* structural difference: activation vector $o^{\ell-1}$ is replaced by $M^{\ell-1} = \text{im2col}(O^{\ell-1})$.

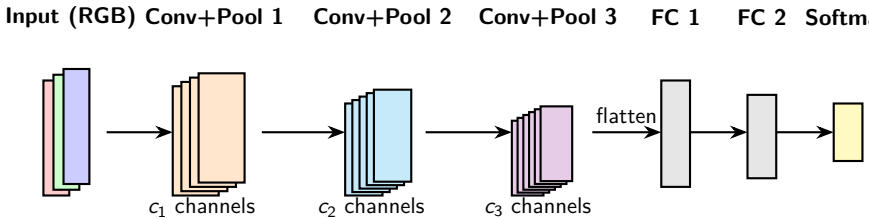
Consequences:

- **Backprop formulas** for MLPs transfer directly (with $M^{\ell-1}$ in place of $o^{\ell-1}$).
- **Implementation** reduces to BLAS GEMM — the same highly optimized code path as dense layers.
- **Theory transfers:** initialization strategies, optimizer convergence analysis, gradient norm bounds — all derived for MLPs apply to CNNs.

See appendix for the explicit chain-rule derivations of $\nabla_K L$ and $\nabla_I L$.

CNN Architecture

CNNs combine convolutional layers with fully-connected layers.



Design pattern: spatial resolution ↓, channel count ↑.

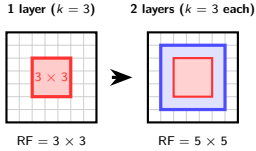
Feature hierarchy: edges $\longrightarrow$ textures, parts $\longrightarrow$ objects, scenes
(edges: early; textures, parts: middle; objects, scenes: later)

Conv layers act as a learned feature extractor; FC layers act as a classifier head.

Standard CNN + FC architectures require a fixed input resolution: changing input size changes the flattened vector dimension and breaks the FC layer.

Depth, Receptive Fields, and Hierarchical Feature Composition

The **receptive field** (RF) of a unit is the region of the original input that influences its activation.

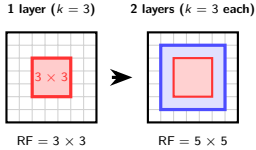


RF side after L layers ($k \times k$, stride 1):

$$\text{RF} = L(k-1) + 1$$

Depth, Receptive Fields, and Hierarchical Feature Composition

The **receptive field** (RF) of a unit is the region of the original input that influences its activation.



RF side after L layers ($k \times k$, stride 1):

$$\text{RF} = L(k-1) + 1$$

L	$k = 3$	$k = 5$
1	3×3	5×5
2	5×5	9×9
4	9×9	17×17
8	17×17	33×33

Two 3×3 layers: RF $= 5 \times 5$, 18 params vs. 25 for one 5×5.

Hierarchical feature composition: each layer operates on features of the layer below. edges $\rightarrow$ corners $\rightarrow$ parts $\rightarrow$ objects.

Empirical evidence (ImageNet top-5 error):

Architecture	Layers	Error
AlexNet	8	16.4%
VGG-16	16	7.3%
ResNet-50	50	5.3%

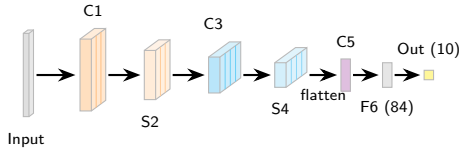
Training very deep networks fails without architectural safeguards $\rightarrow$ next lecture.

Key point: depth grows spatial context and representational power simultaneously, at constant parameter cost per layer.

LeNet-5 (LeCun et al., 1998)

A 5-layer CNN for 10-class digit classification (32×32 input).

Name	Type	Ch.	Filter	Stride	Act.	Output Size
—	Input	1	—	—	—	$32 \times 32 \times 1$
C1	Conv2D	6	5×5	1	tanh	$28 \times 28 \times 6$
S2	AvgPool	6	2×2	2	—	$14 \times 14 \times 6$
C3	Conv2D	16	5×5	1	tanh	$10 \times 10 \times 16$
S4	AvgPool	16	2×2	2	—	$5 \times 5 \times 16$
C5	Conv2D	120	5×5	1	tanh	$1 \times 1 \times 120$
F6	FC	84	—	—	tanh	84
Out	FC	10	—	—	softmax	10



Why LeNet worked: digit classification involves local stroke patterns; 5×5 filters are well-matched to the scale of strokes in 32×32 images.

LeNet-5: Tracing Dimensions and Parameters

Spatial dimension trace:

$$32 \xrightarrow[\text{conv } 5\times5]{-4} 28 \xrightarrow[\text{pool } 2\times2]{\div 2} 14 \xrightarrow[\text{conv } 5\times5]{-4} 10 \xrightarrow[\text{pool } 2\times2]{\div 2} 5 \xrightarrow[\text{conv } 5\times5]{-4} 1$$

Parameter count using $c_{\ell-1} c_\ell k_1 k_2 + c_\ell$:

Layer	**Kernel params**	**Bias**	**Total**
Conv1: $1 \times 6 \times 5 \times 5$	150	6	156
Conv2: $6 \times 16 \times 5 \times 5$	2,400	16	2,416
Conv3: $16 \times 120 \times 5 \times 5$	48,000	120	48,120
FC1: 120×84	10,080	84	10,164
FC2: 84×10	840	10	850
	Grand total:		**61,706**

LeNet-5: Tracing Dimensions and Parameters

Spatial dimension trace:

$$32 \xrightarrow[\text{conv } 5\times5]{-4} 28 \xrightarrow[\text{pool } 2\times2]{\div 2} 14 \xrightarrow[\text{conv } 5\times5]{-4} 10 \xrightarrow[\text{pool } 2\times2]{\div 2} 5 \xrightarrow[\text{conv } 5\times5]{-4} 1$$

Parameter count using $c_{\ell-1} c_\ell k_1 k_2 + c_\ell$:

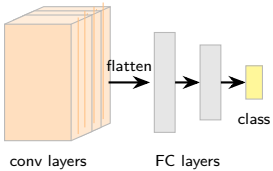
Layer	**Kernel params**	**Bias**	**Total**
Conv1: $1 \times 6 \times 5 \times 5$	150	6	156
Conv2: $6 \times 16 \times 5 \times 5$	2,400	16	2,416
Conv3: $16 \times 120 \times 5 \times 5$	48,000	120	48,120
FC1: 120×84	10,080	84	10,164
FC2: 84×10	840	10	850
	Grand total:		**61,706**

Compare: a single FC layer from $32 \times 32 = 1{,}024$ inputs to 120 outputs would need $1{,}024 \times 120 + 120 = 122{,}880$ parameters — roughly $2\times$ the *entire* LeNet-5.

Fully Convolutional Networks

The CNN + FC pattern (LeNet-style) flattens spatial feature maps before classifying. This is not the only option.

CNN + FC (classification)

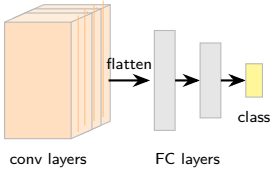


- Requires **fixed input size**
- Spatial info **destroyed** at flatten
- Output: single class label

Fully Convolutional Networks

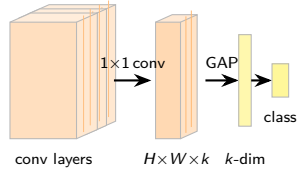
The CNN + FC pattern (LeNet-style) flattens spatial feature maps before classifying. This is not the only option.

CNN + FC (classification)



- Requires **fixed input size**
- Spatial info **destroyed** at flatten
- Output: single class label

Fully Convolutional Network (FCN)

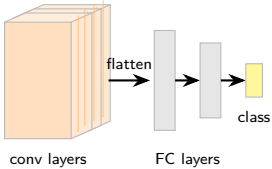


- Accepts any input size
- Spatial info preserved throughout
- Output: spatial map *or* class label via GAP

Fully Convolutional Networks

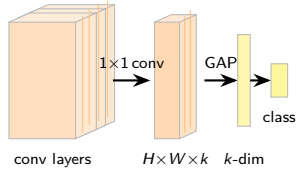
The CNN + FC pattern (LeNet-style) flattens spatial feature maps before classifying. This is not the only option.

CNN + FC (classification)



- Requires **fixed input size**
- Spatial info **destroyed** at flatten
- Output: single class label

Fully Convolutional Network (FCN)



- Accepts any input size
- Spatial info preserved throughout
- Output: spatial map *or* class label via GAP

1×1 **conv:** applies a $k \times c_{\text{in}}$ linear map independently at every spatial location — equivalent to a dense layer without flattening. **GAP:** collapses each $H \times W$ channel to its spatial mean (k-vector, no parameters). Together they enable variable input sizes and dense prediction (segmentation, detection).

A Concrete FCN: LeNet-FCN for FashionMNIST

Replace LeNet-5's FC layers with 1×1 convolutions + GAP. Input: $28 \times 28 \times 1$.

Name	**Type**	**Ch.**	**Filter**	**Pad**	**Act.**	**Output Size**
—	Input	1	—	—	—	$28 \times 28 \times 1$
C1	Conv2D	6	5×5	same	tanh	$28 \times 28 \times 6$
S2	AvgPool	6	2×2	—	—	$14 \times 14 \times 6$
C3	Conv2D	16	5×5	same	tanh	$14 \times 14 \times 16$
S4	AvgPool	16	2×2	—	—	$7 \times 7 \times 16$
C5	Conv2D	120	5×5	same	tanh	$7 \times 7 \times 120$

A Concrete FCN: LeNet-FCN for FashionMNIST

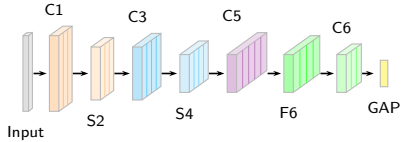
Replace LeNet-5's FC layers with 1×1 convolutions + GAP. Input: $28 \times 28 \times 1$.

Name	**Type**	**Ch.**	**Filter**	**Pad**	**Act.**	**Output Size**
—	Input	1	—	—	—	$28 \times 28 \times 1$
C1	Conv2D	6	5×5	same	tanh	$28 \times 28 \times 6$
S2	AvgPool	6	2×2	—	—	$14 \times 14 \times 6$
C3	Conv2D	16	5×5	same	tanh	$14 \times 14 \times 16$
S4	AvgPool	16	2×2	—	—	$7 \times 7 \times 16$
C5	Conv2D	120	5×5	same	tanh	$7 \times 7 \times 120$
F6	1×1 Conv	84	1×1	—	tanh	$7 \times 7 \times 84$
C6	1×1 Conv	10	1×1	—	linear	$7 \times 7 \times 10$
GAP	GlobalAvgPool	10	7×7	—	—	10

A Concrete FCN: LeNet-FCN for FashionMNIST

Replace LeNet-5's FC layers with 1×1 convolutions + GAP. Input: $28 \times 28 \times 1$.

Name	Type	Ch.	Filter	Pad	Act.	Output Size
—	Input	1	—	—	—	$28 \times 28 \times 1$
C1	Conv2D	6	5×5	same	tanh	$28 \times 28 \times 6$
S2	AvgPool	6	2×2	—	—	$14 \times 14 \times 6$
C3	Conv2D	16	5×5	same	tanh	$14 \times 14 \times 16$
S4	AvgPool	16	2×2	—	—	$7 \times 7 \times 16$
C5	Conv2D	120	5×5	same	tanh	$7 \times 7 \times 120$
F6	1×1 Conv	84	1×1	—	tanh	$7 \times 7 \times 84$
C6	1×1 Conv	10	1×1	—	linear	$7 \times 7 \times 10$
GAP	GlobalAvgPool	10	7×7	—	—	10



LeNet-FCN: Dimensions, Parameters, and Comparison with LeNet-5

Spatial dimension trace:

$$28 \xrightarrow[\text{conv same}]{} 28 \xrightarrow[\text{pool}]{\div 2} 14 \xrightarrow[\text{conv same}]{} 14 \xrightarrow[\text{pool}]{\div 2} 7 \xrightarrow[\text{conv same}]{} 7 \xrightarrow[1\times1]{} 7 \xrightarrow[1\times1]{} 7 \xrightarrow[\text{GAP}]{} 1$$

Parameter count (using $c_{\text{in}}\, c_{\text{out}}\, k_1 k_2 + c_{\text{out}}$):

Layer	Kernel	Bias	Total
Conv1: $1 \times 6 \times 5 \times 5$	150	6	156
Conv2: $6 \times 16 \times 5 \times 5$	2,400	16	2,416
Conv3: $16 \times 120 \times 5 \times 5$	48,000	120	48,120
1×1: 120×84	10,080	84	10,164
1×1: 84×10	840	10	850
	Grand total:		**61,706**

Same total as LeNet-5!

The 1×1 layer replacing FC1 has $120 \times 84 + 84 = 10{,}164$ parameters — *identical* to LeNet's FC1.

A 1×1 convolution is a dense layer applied at every spatial location independently.

Key advantage over LeNet-5: this network accepts any input $\geq 7 \times 7$ and produces a $C \times H' \times W'$ spatial output before GAP.

Summary

1. **CNNs encode spatial inductive biases as a structured parameterization.** Locality and translation equivariance are not hand-engineered features — they emerge from the convolutional weight-sharing constraint.
2. **Depth grows the receptive field and composes features hierarchically.** Two 3×3 layers cover a 5×5 region with 18 parameters; one 5×5 layer uses 25. Each layer operates on the features produced by the layer below: edges $\rightarrow$ corners $\rightarrow$ parts $\rightarrow$ objects.
3. **A conv layer is a structured matrix multiply.** $A^{\ell} = W^{\ell} M^{\ell-1} + b^{\ell}$; backprop, initialization, and optimizer analysis for MLPs transfer directly.
4. **Architecture reflects what the task requires.** Fixed-input classification: CNN + FC (or GAP). Variable-input dense prediction: FCN with 1×1 conv + GAP.

Looking Ahead

The obstacle: vanishing and exploding gradients.

LeNet has 5 layers; modern CNNs have 50–150. As depth grows, gradients propagate through one Jacobian per layer. Products of many Jacobians either shrink to zero (vanishing) or blow up (exploding), leaving early layers unable to learn — no matter how long you train.

Next lecture:

- **Batch normalization:** keep layer activations well-scaled throughout training
- **Residual connections (ResNets):** create direct gradient pathways to early layers, bypassing the Jacobian chain
- **Transfer learning and fine-tuning:** reuse representations learned on large datasets

Questions?

Appendix A — Backpropagation Through a CNN

These slides derive the gradient formulas needed to train the convolutional backbone of a CNN. **CSCI 6960 students** are expected to be able to reproduce these calculations.

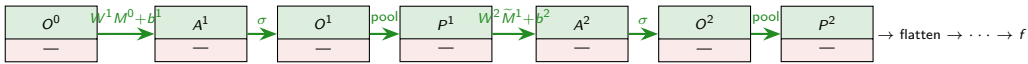
The im2col unification on the main slides already tells us the answer *must* be equivalent to MLP backprop. These slides show it explicitly.

Key difference from MLPs: pooling layers sit between conv layers, so the *input* to each conv layer is the *pooled* output of the previous one. This changes the forward equations and introduces an extra backward step.

Assumptions: stride 1, no padding. **Exercise:** generalize to arbitrary stride and padding.

A1: Forward Pass Through the CNN Backbone

Architecture: image $\to$ conv$_1$ $\to$ pool$_1$ $\to$ conv$_2$ $\to$ pool$_2$ $\to$ flatten $\to \cdots \to \ell$.



Notation. Layer ℓ has c_ℓ output channels, kernels K_{ij}^ℓ $(k_1 \times k_2)$.

Unlike MLPs, pooling sits between conv layers. The input to conv layer ℓ is the *pooled* output $P^{\ell-1}$ (not $O^{\ell-1}$ directly). For the first layer, $P^0 := O^0$ (the input image).

$$A_i^\ell = \sum_{j=1}^{c_{\ell-1}} P_j^{\ell-1} * K_{ij}^\ell + b_i^\ell, \qquad O_i^\ell = \sigma(A_i^\ell)$$

Im2col form: $A^\ell = W^\ell \widetilde{M}^{\ell-1} + b^\ell$, $O^\ell = \sigma(A^\ell)$, where $W^\ell \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2}$ and $\widetilde{M}^{\ell-1} = \text{im2col}(P^{\ell-1})$.

A2: What Gradients Do We Need?

Assume we have $\nabla_{\mathrm{vec}(P^2)} f$ (gradient w.r.t. the flattened output of the last pooling layer, supplied by backprop through the layers above).

A2: What Gradients Do We Need?

Assume we have $\nabla_{\mathrm{vec}(P^2)} f$ (gradient w.r.t. the flattened output of the last pooling layer, supplied by backprop through the layers above).

The backward pass must compute, for each conv layer $\ell = 2, 1$:

1. $\nabla_{W^\ell} f$ — gradient w.r.t. the kernel weight matrix (to update W^ℓ)
2. $\nabla_{b^\ell} f$ — gradient w.r.t. the bias (to update b^ℓ)
3. $\nabla_{\widetilde{M}^{\ell-1}} f$ — gradient w.r.t. the im2col input (to propagate further back)

A2: What Gradients Do We Need?

Assume we have $\nabla_{\mathrm{vec}(P^2)} f$ (gradient w.r.t. the flattened output of the last pooling layer, supplied by backprop through the layers above).

The backward pass must compute, for each conv layer $\ell = 2, 1$:

1. $\nabla_{W^\ell} f$ — gradient w.r.t. the kernel weight matrix (to update W^ℓ)
2. $\nabla_{b^\ell} f$ — gradient w.r.t. the bias (to update b^ℓ)
3. $\nabla_{\widetilde{M}^{\ell-1}} f$ — gradient w.r.t. the im2col input (to propagate further back)

Plus, between conv layers, we must propagate through:

4. The **activation** σ (elementwise $\Rightarrow$ Hadamard product, as in MLP backprop)
5. The **pooling layer** (routes gradients to the selected positions)

A2: What Gradients Do We Need?

Assume we have $\nabla_{\mathrm{vec}(P^2)} f$ (gradient w.r.t. the flattened output of the last pooling layer, supplied by backprop through the layers above).

The backward pass must compute, for each conv layer $\ell = 2, 1$:

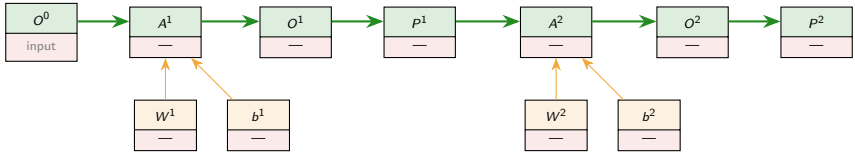
1. $\nabla_{W^\ell} f$ — gradient w.r.t. the kernel weight matrix (to update W^ℓ)
2. $\nabla_{b^\ell} f$ — gradient w.r.t. the bias (to update b^ℓ)
3. $\nabla_{\widetilde{M}^{\ell-1}} f$ — gradient w.r.t. the im2col input (to propagate further back)

Plus, between conv layers, we must propagate through:

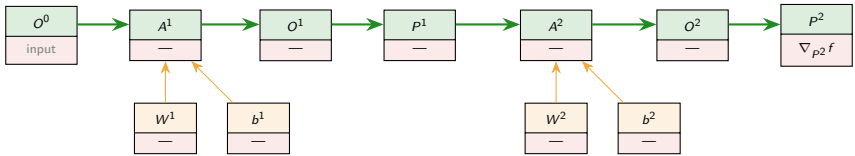
4. The **activation** σ (elementwise $\Rightarrow$ Hadamard product, as in MLP backprop)
5. The **pooling layer** (routes gradients to the selected positions)

Key insight: in im2col form, (1)–(3) are *identical* to MLP backprop with $\widetilde{M}^{\ell-1}$ replacing $o^{\ell-1}$. Steps (4)–(5) are the only new ingredients.

A3: The Backward Pass (DAG)

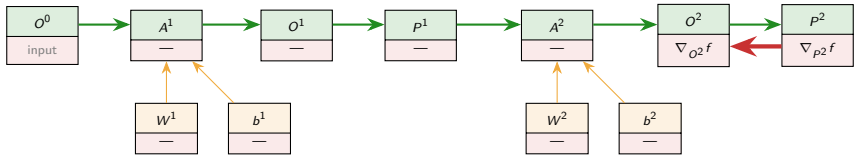


A3: The Backward Pass (DAG)



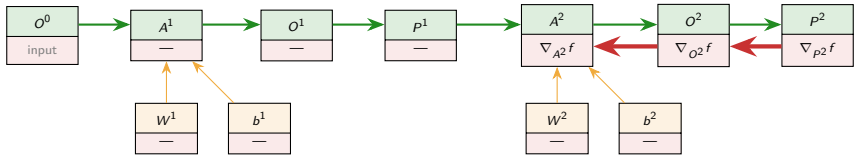
Given: $\nabla_{P^2} f$ from the layers above (flatten $\rightarrow$ FC $\rightarrow$ loss).

A3: The Backward Pass (DAG)



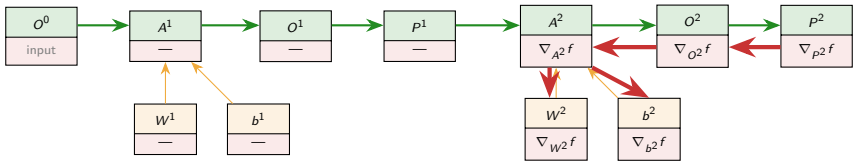
Pool backprop: $\nabla_{O^2} f$ routes $\nabla_{P^2} f$ to selected positions (max) or distributes (avg).

A3: The Backward Pass (DAG)



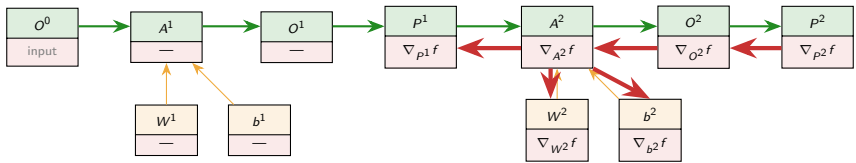
Activation: $\nabla_{A^2} f = \sigma'(A^2) \odot \nabla_{O^2} f$ (identical to MLP).

A3: The Backward Pass (DAG)



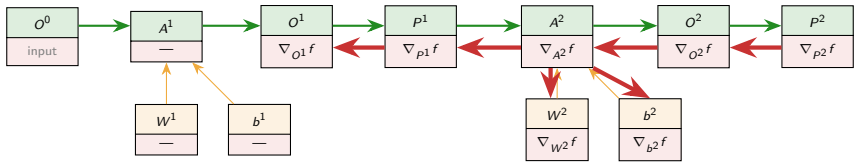
Parameters: $\nabla_{W^2} f = (\nabla_{A^2} f)\,(\widetilde{M}^1)^\top$, $\nabla_{b^2} f = (\nabla_{A^2} f) \cdot \mathbf{1}$.

A3: The Backward Pass (DAG)



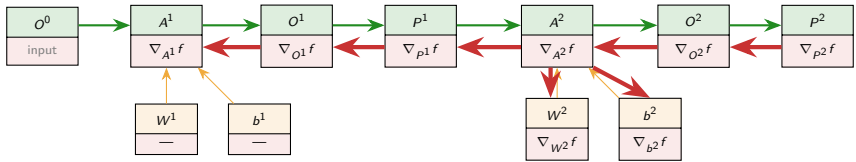
Propagation: $\nabla_{\widetilde{M}^1} f = (W^2)^\top \nabla_{A^2} f \;\Rightarrow$ reshape to $\nabla_{P^1} f$.

A3: The Backward Pass (DAG)



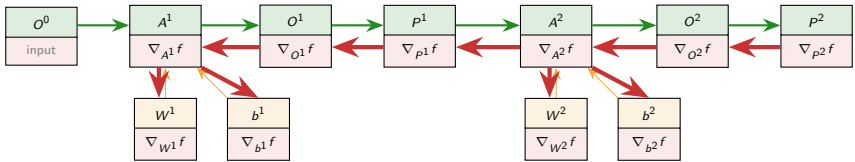
Pool backprop again: $\nabla_{O^1} f$ from $\nabla_{P^1} f$.

A3: The Backward Pass (DAG)



Activation: $\nabla_{A^1} f = \sigma'(A^1) \odot \nabla_{O^1} f$.

A3: The Backward Pass (DAG)



Parameters: $\nabla_{W^1} f = (\nabla_{A^1} f)(\widetilde{M}^0)^\top$, $\nabla_{b^1} f = (\nabla_{A^1} f) \cdot \mathbf{1}$. **Done.**

A4: Backprop Through One Conv Layer

Focus on a single conv layer ℓ in im2col form. The input is the *pooled* output of the previous layer:

$$A^{\ell} = W^{\ell}\,\widetilde{M}^{\ell-1} + b^{\ell}\mathbf{1}^{\top}, \qquad O^{\ell} = \sigma(A^{\ell})$$

with $W^{\ell} \in \mathbb{R}^{c_{\ell} \times c_{\ell-1}k_1k_2}$, $\widetilde{M}^{\ell-1} = \mathsf{im2col}(P^{\ell-1}) \in \mathbb{R}^{c_{\ell-1}k_1k_2 \times d_1'd_2'}$, $A^{\ell} \in \mathbb{R}^{c_{\ell} \times d_1'd_2'}$.

A4: Backprop Through One Conv Layer

Focus on a single conv layer ℓ in im2col form. The input is the *pooled* output of the previous layer:

$$A^\ell = W^\ell \, \widetilde{M}^{\ell-1} + b^\ell \mathbf{1}^\top, \qquad O^\ell = \sigma(A^\ell)$$

with $W^\ell \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2}$, $\widetilde{M}^{\ell-1} = \mathrm{im2col}(P^{\ell-1}) \in \mathbb{R}^{c_{\ell-1} k_1 k_2 \times d_1' d_2'}$, $A^\ell \in \mathbb{R}^{c_\ell \times d_1' d_2'}$.

Given: $\nabla_{A^\ell} f \in \mathbb{R}^{c_\ell \times d_1' d_2'}$ (from activation backprop: $\nabla_{A^\ell} f = \sigma'(A^\ell) \odot \nabla_{O^\ell} f$).

A4: Backprop Through One Conv Layer

Focus on a single conv layer ℓ in im2col form. The input is the *pooled* output of the previous layer:

$$A^\ell = W^\ell \, \widetilde{M}^{\ell-1} + b^\ell \mathbf{1}^\top, \qquad O^\ell = \sigma(A^\ell)$$

with $W^\ell \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2}$, $\widetilde{M}^{\ell-1} = \text{im2col}(P^{\ell-1}) \in \mathbb{R}^{c_{\ell-1} k_1 k_2 \times d_1' d_2'}$, $A^\ell \in \mathbb{R}^{c_\ell \times d_1' d_2'}$.

Given: $\nabla_{A^\ell} f \in \mathbb{R}^{c_\ell \times d_1' d_2'}$ (from activation backprop: $\nabla_{A^\ell} f = \sigma'(A^\ell) \odot \nabla_{O^\ell} f$).

This is *exactly* the MLP setup $a = Wo + b$. The MLP backprop formulas transfer directly:

$$\begin{aligned}
\nabla_{W^\ell} f &= (\nabla_{A^\ell} f)\,(\widetilde{M}^{\ell-1})^\top && \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2} \\
\nabla_{b^\ell} f &= (\nabla_{A^\ell} f)\,\mathbf{1} && \in \mathbb{R}^{c_\ell} \\
\nabla_{\widetilde{M}^{\ell-1}} f &= (W^\ell)^\top \nabla_{A^\ell} f && \in \mathbb{R}^{c_{\ell-1} k_1 k_2 \times d_1' d_2'}
\end{aligned}$$

A4: Backprop Through One Conv Layer

Focus on a single conv layer ℓ in im2col form. The input is the *pooled* output of the previous layer:

$$A^\ell = W^\ell \, \widetilde{M}^{\ell-1} + b^\ell \mathbf{1}^\top, \qquad O^\ell = \sigma(A^\ell)$$

with $W^\ell \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2}$, $\widetilde{M}^{\ell-1} = \text{im2col}(P^{\ell-1}) \in \mathbb{R}^{c_{\ell-1} k_1 k_2 \times d_1' d_2'}$, $A^\ell \in \mathbb{R}^{c_\ell \times d_1' d_2'}$.

Given: $\nabla_{A^\ell} f \in \mathbb{R}^{c_\ell \times d_1' d_2'}$ (from activation backprop: $\nabla_{A^\ell} f = \sigma'(A^\ell) \odot \nabla_{O^\ell} f$).

This is *exactly* the MLP setup $a = Wo + b$. The MLP backprop formulas transfer directly:

$$\begin{aligned}
\nabla_{W^\ell} f &= (\nabla_{A^\ell} f)\,(\widetilde{M}^{\ell-1})^\top && \in \mathbb{R}^{c_\ell \times c_{\ell-1} k_1 k_2} \\
\nabla_{b^\ell} f &= (\nabla_{A^\ell} f)\,\mathbf{1} && \in \mathbb{R}^{c_\ell} \\
\nabla_{\widetilde{M}^{\ell-1}} f &= (W^\ell)^\top \nabla_{A^\ell} f && \in \mathbb{R}^{c_{\ell-1} k_1 k_2 \times d_1' d_2'}
\end{aligned}$$

The last quantity $\nabla_{\widetilde{M}^{\ell-1}} f$ is reshaped (via col2im, the inverse of im2col) to recover $\nabla_{P^{\ell-1}} f$, which is then passed through pool backprop.

That's all the new math. The only new ingredients beyond MLP backprop are pool backprop (next slides) and the use of $\widetilde{M}^{\ell-1} = \text{im2col}(P^{\ell-1})$ instead of $o^{\ell-1}$.

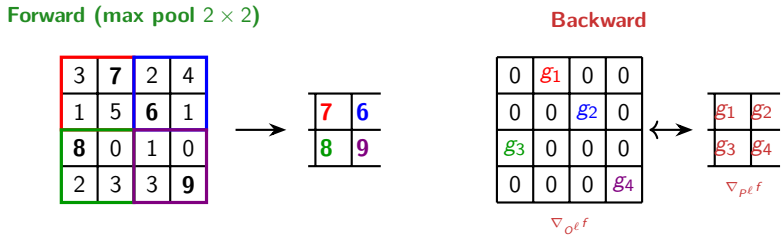
A5: Backprop Through Max Pooling

Max pooling forwards each pool window's maximum to the output.

A5: Backprop Through Max Pooling

Max pooling forwards each pool window's maximum to the output.

Backward: the gradient routes entirely to the **argmax position**; all other positions receive zero.



Formally: let $\mu(p, q)$ be the argmax index within the pool window at position (p, q). Then

$$(\nabla_{O^\ell} f)_{a,b} = \begin{cases} (\nabla_{P^\ell} f)_{p,q} & \text{if } (a, b) = \mu(p, q) \\ 0 & \text{otherwise} \end{cases}$$

The argmax positions are recorded during the forward pass ("switch variables").

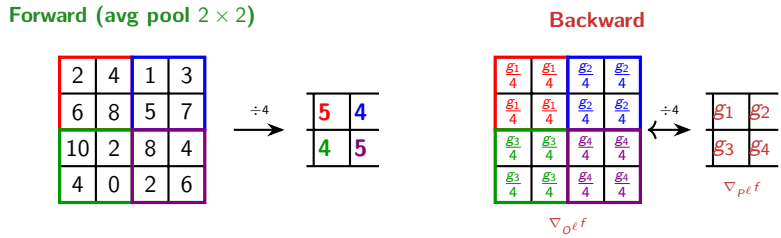
A6: Backprop Through Average Pooling

Average pooling replaces each $s \times s$ window with its mean.

A6: Backprop Through Average Pooling

Average pooling replaces each $s \times s$ window with its mean.

Backward: each upstream gradient is distributed *uniformly* to all s^2 positions in the window.



Formally: for an $s \times s$ average pool with stride s (non-overlapping windows):

$$(\nabla_{O^\ell} f)_{a,b} = \frac{1}{s^2} (\nabla_{P^\ell} f)_{\lfloor a/s \rfloor, \lfloor b/s \rfloor}$$

Average pooling is a linear operation, so its Jacobian is constant (no values to store from the forward pass, unlike max pool).

A7: Summary — All Gradients in im2col Form

Conv layer ℓ forward: $A^\ell = W^\ell \widetilde{M}^{\ell-1} + b^\ell \mathbf{1}^\top$, $O^\ell = \sigma(A^\ell)$, $P^\ell = \text{pool}(O^\ell)$.

Given: $\nabla_{P^\ell} f$ from pool backprop or from layers above.

Quantity	Formula
$\nabla_{O^\ell} f$	pool backprop: max $\to$ route to argmax; avg $\to$ distribute $\div s^2$
$\nabla_{A^\ell} f$	$\sigma'(A^\ell) \odot \nabla_{O^\ell} f$ (elementwise, same as MLP)
$\nabla_{W^\ell} f$	$(\nabla_{A^\ell} f)\,(\widetilde{M}^{\ell-1})^\top$ (GEMM)
$\nabla_{b^\ell} f$	$(\nabla_{A^\ell} f)\,\mathbf{1}$ (sum over spatial dims)
$\nabla_{\widetilde{M}^{\ell-1}} f$	$(W^\ell)^\top \nabla_{A^\ell} f$ (GEMM) $\xrightarrow{\text{col2im}}$ $\nabla_{P^{\ell-1}} f$

Bottom line: CNN backprop = MLP backprop + pool backprop. The im2col representation makes this explicit: the same GEMM calls that implement the forward

Appendix B — Convolution: 0-Indexed Implementation Form

The main slides use a centered kernel index (center at $(0, 0)$) for notational clarity. Frameworks such as PyTorch and TensorFlow use 0-indexed kernels where the *top-left* corner is $(0, 0,)$:

$$(I * K)_{i,j} \;=\; \sum_{c=0}^{k_1-1} \sum_{r=0}^{k_2-1} K_{c,r}\, I_{i+c,\, j+r}$$

Valid output positions: $i \in \{0, \ldots, d_1 - k_1\}$, $j \in \{0, \ldots, d_2 - k_2\}$.

Relationship to centered form: substituting $p = c - m$, $q = r - n$ (with $m = \lfloor k_1/2 \rfloor$, $n = \lfloor k_2/2 \rfloor$) and reindexing $i \leftarrow i + m$, $j \leftarrow j + n$ recovers exactly the centered formula on the main slides. The two forms are equivalent; only the indexing convention differs.

The output-size formula $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1)$ holds for both forms.

Appendix B — Deriving the Sobel Filters

Goal: design a 3×3 kernel that approximates the horizontal derivative $\partial I/\partial x$ (which detects *vertical* edges).

Step 1 — finite difference. The simplest horizontal derivative kernel is

$$D_x = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

applied to a single row: $(D_x * I)_{i,j} \approx I_{i,j+1} - I_{i,j-1}$.

Step 2 — reduce noise by averaging vertically. Images are noisy; a single-row derivative is sensitive to noise in adjacent rows. Convolve D_x with a vertical smoothing kernel $S = (1\ 2\ 1)^\top/4$:

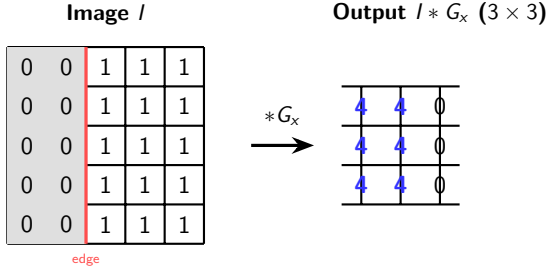
$$G_x = S \otimes D_x = \frac{1}{4}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

The $1/4$ is absorbed into the learned scale, so the Sobel kernel is usually written without it.

G_y is obtained symmetrically: differentiate vertically ($D_y = (1\ 0\ -1)^\top$), smooth horizontally.

Appendix B — Sobel G_x: Toy Example (Vertical Edge)

Setup: 5×5 binary image with a sharp vertical edge between columns 1 and 2 (0-indexed).



Interpretation: the two output columns ($j = 0, 1$) whose receptive fields *straddle* the vertical edge both give response
$4 = (-1)(0) + (0)(0) + (1)(1) \cdot 1 + (-2)(0) + \cdots + (2)(1) \cdot 2 + (-1)(0) + \cdots + (1)(1) \cdot 1.$
The rightmost column ($j = 2$), whose patch lies entirely in the bright region, gives 0. G_x fires at vertical boundaries.