
Implementing Neural Network Training in PyTorch (continued)

ECE 57000 — October 19, 2026

David I. Inouye

| Training separates **gradient computation, updates, and evaluation**

- **Forward and backward:** record the computation, save needed values, then accumulate parameter gradients.
- **Update:** clear old gradients, compute the new batch loss and gradients, then call `optimizer.step()`.
- **Evaluate and log:** use held-out data with `no_grad()`; store scalar or detached values without their training graphs.

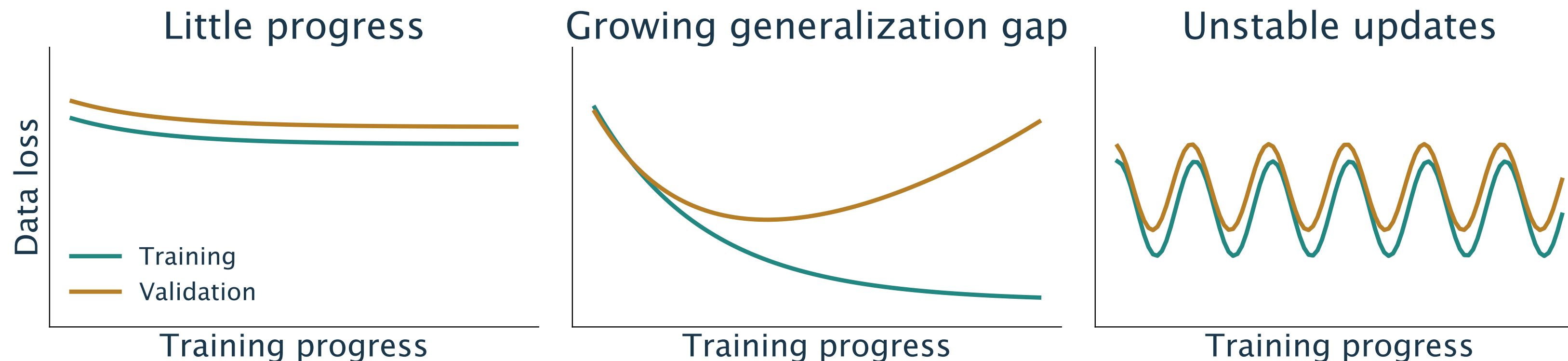
Today: finish the training recipe, then change the model to use image structure.

| Initialization is **part of the training recipe**

- **PyTorch provides useful defaults:** standard layers initialize their parameters when constructed; `nn.Linear` scales its initial values with the input width.
- **Match the method you reproduce:** follow a paper's specified initialization, including its scale, distribution, and bias settings. Defaults may differ.
- **Initialization can materially change training and the final model.** Record the strategy and random seed; use multiple seeds when comparing results.

`torch.nn.init` provides strategies such as **Xavier** and **Kaiming** initialization. Choose for the architecture and activation; do not replace a paper's recipe blindly.

| Training curves help **diagnose the complete loop**



Both losses remain high

Check code, fitting, and representation

Training falls, validation rises

Revisit capacity, regularization, or stopping

Loss oscillates sharply

Inspect initialization, scaling, and step size

Running training loss is cheap and useful on large datasets, but mixes changing weights. For a clearer divergence diagnosis, evaluate a fixed model on a representative training subset and validation data using the same loss.

Illustrative curves, not experiment results. Keep test data out of these choices.

| PyTorch connects a model, its gradients, and repeated updates

Stage	What we control
Define	Dataset, batches, model, initialization, scalar loss
Differentiate	Forward builds the graph; backward adds parameter gradients
Update	Clear old gradients, compute new ones, then call <code>optimizer.step()</code>
Evaluate	Use held-out data and log values without retaining graphs

The training loop stays useful when the architecture changes. Next, replace unrestricted pixel connections with local, shared filters.

Convolutions and Image Structure

| Image structure changes **how we connect and share weights**

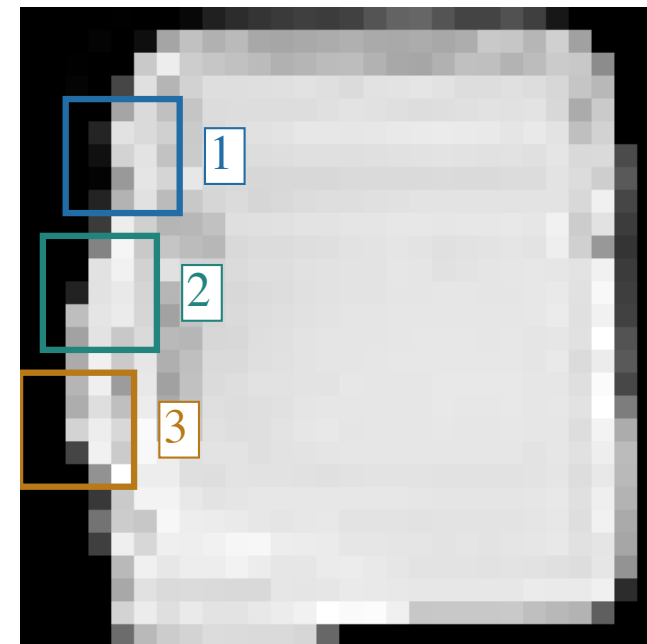
Arc 5: representations → gradients → PyTorch → **image structure** → evidence.

1. **Patterns:** why use local, shared rules?
2. **Filters:** how does convolution compute features?
3. **Networks:** how do shapes and layers fit together?

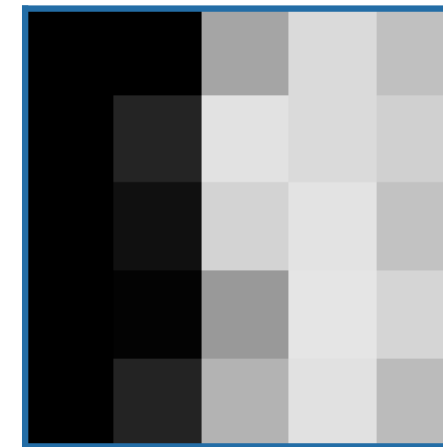
The training loop is ready. Now change the model's assumptions.

Nearby pixels reveal **local image patterns**

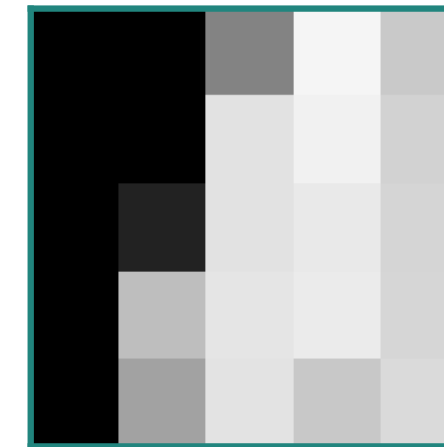
Where do these patches belong?



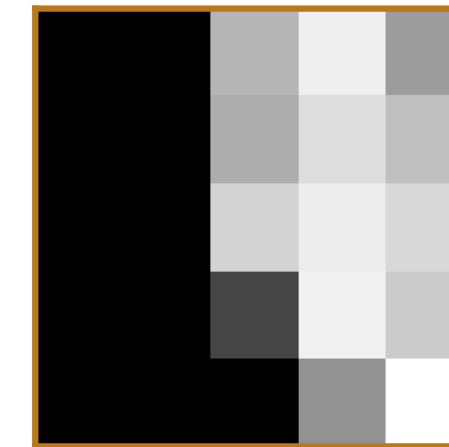
Patch 1



Patch 2



Patch 3



Three 5×5 crops; same brightness scale

Where do these patches belong?

A. Left edge B. Interior C. Right edge

A. Left edge. Each patch goes from dark to bright. One local detector can find this pattern at many positions.

| Image features have meaningful spatial neighbors

General feature vector: in [age, income, height], adjacent entries need not be more related than distant entries. Their order identifies features, not spatial neighbors.

Image: neighboring pixels measure nearby locations. Their arrangement forms edges, textures, and shapes.

Fully connected layer	Convolutional layer
Each hidden unit can use every pixel	Each output uses a local patch
Different units have independent weights	Reuse the same filter across locations
Can learn spatial relationships from data	Builds local reuse into the architecture

Convolution uses this structure: local neighborhoods matter, and a useful pattern may recur at different locations.

Filters turn local patterns into **computed features**

Image structure · 2/3 Filters · Patterns → **Filters** → Networks

| A convolution reuses **one local weighted sum**

Same three weights at every position

1	2	0	3	1	2	4
---	---	---	---	---	---	---

1	0	-1
---	---	----

Input: 7 values

Kernel: $[1, 0, -1]$

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

1	-1	-1	?	?
---	----	----	---	---

Output: $7 - 3 + 1 = 5$ positions

At each position: multiply the window entries by the same kernel entries, then add. A learned bias may be added to every result.

| A 2D kernel slides **across rows and columns**

4 × 4 input

1	2	0	1
0	1	3	2
2	0	1	1
1	2	0	3

2 × 2 kernel

1	0
0	-1

3 × 3 output

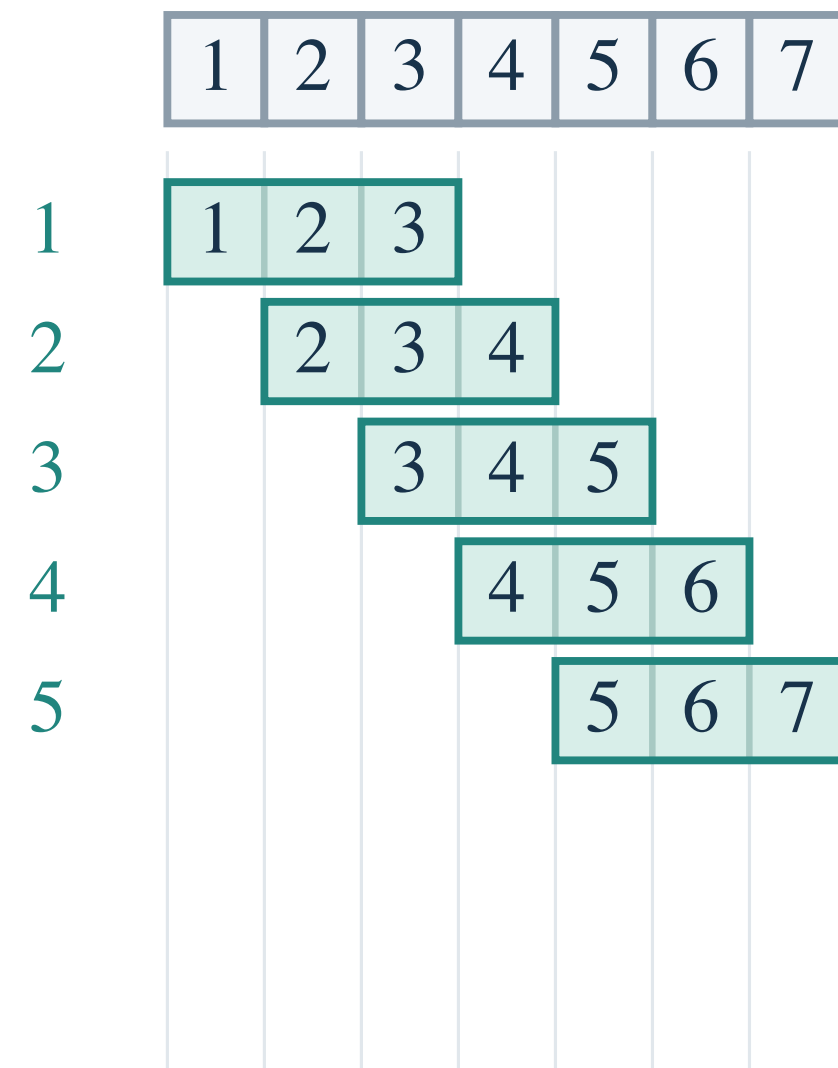
0	-1	-2
0	?	?
?	?	?

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

A 2 × 2 window fits at **three row starts and three column starts** in a 4 × 4 image: the output is 3 × 3.

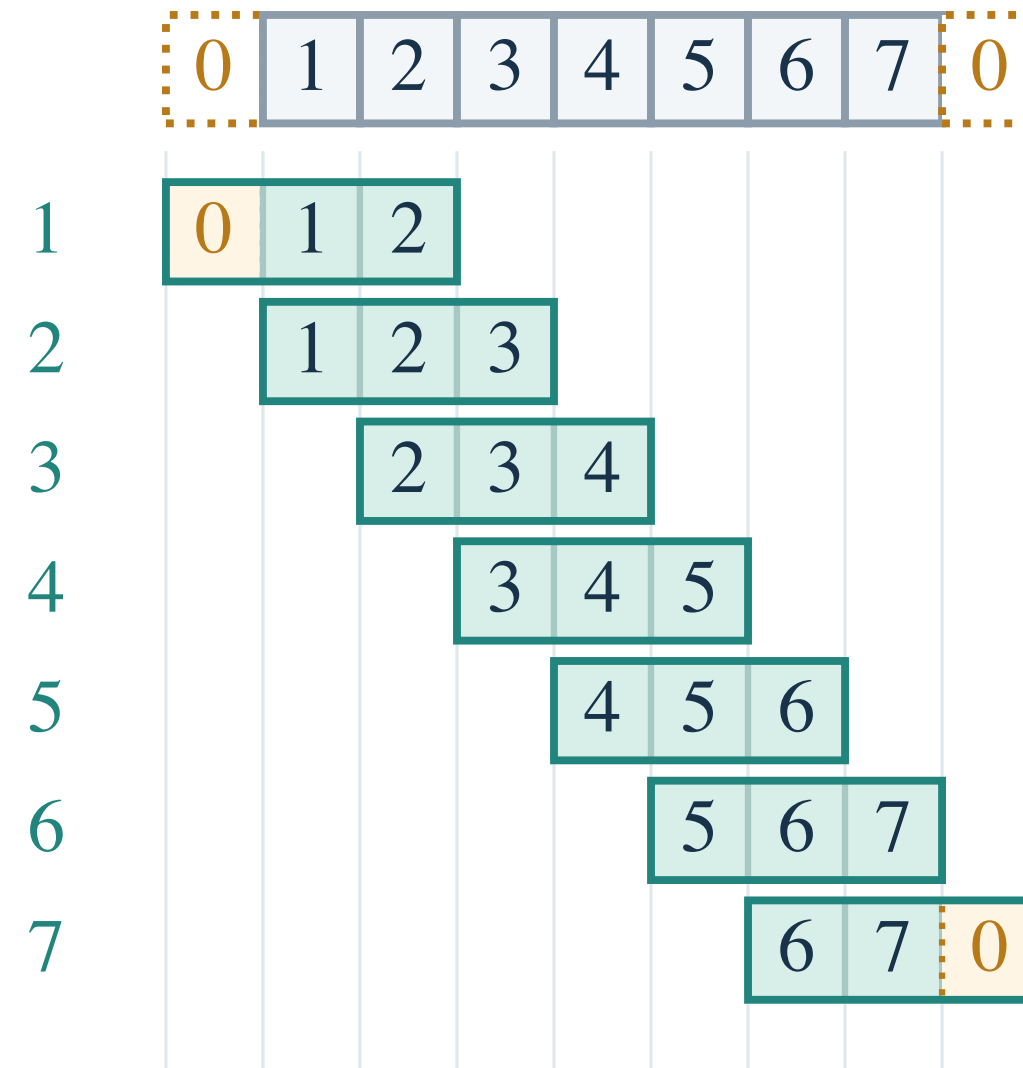
| Padding adds a border; stride controls how quickly the window moves

Padding 0 · stride 1



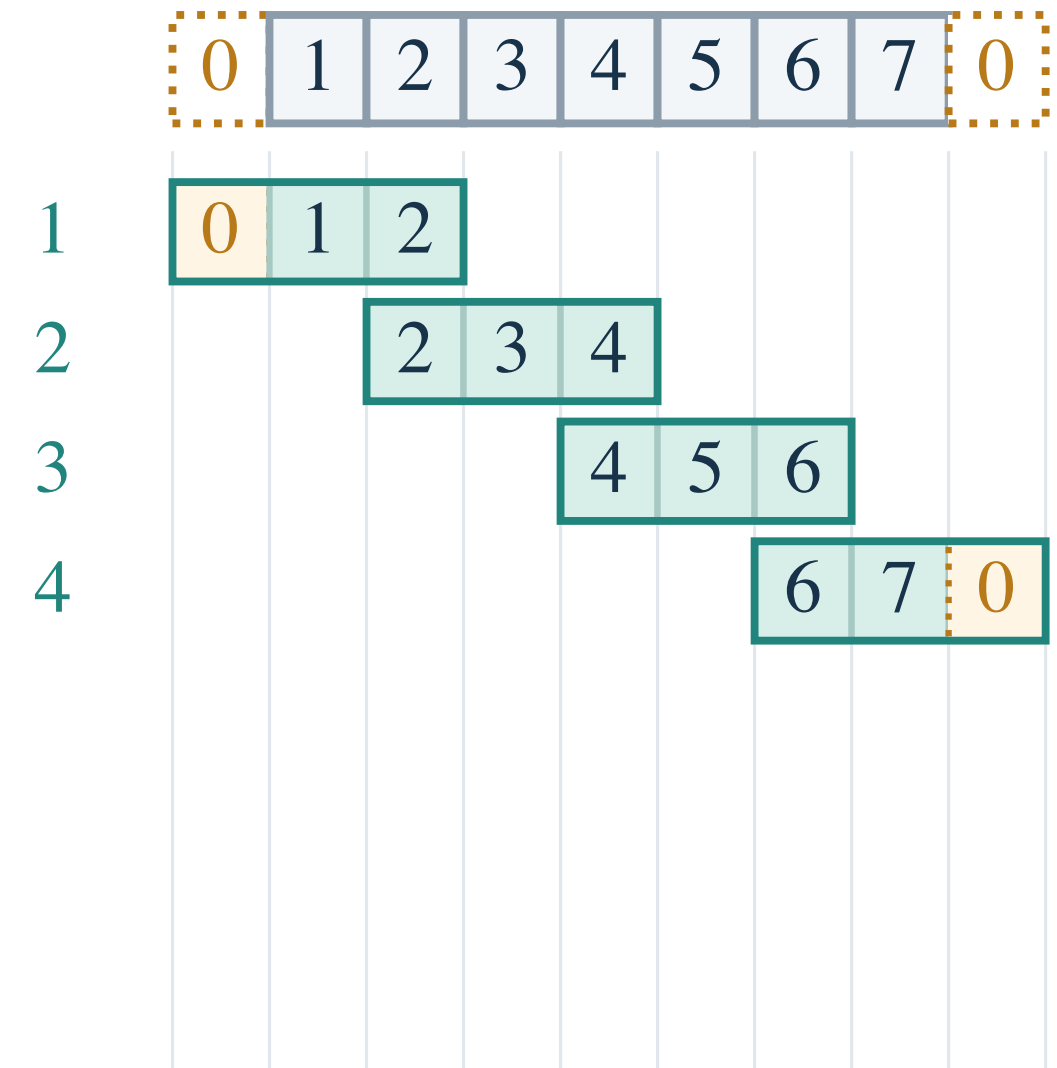
5 windows → 5 outputs

Padding 1 · stride 1



7 windows → 7 outputs

Padding 1 · stride 2



4 windows → 4 outputs

Each numbered window uses the same 3 weights. Dotted zeros are implicit padding.

Padding p : p implicit zeros on each side. **Stride s :** move s entries per step.

| Spatial output size counts **how many complete windows fit**

Apply this separately to **height and width**. For either spatial dimension, use input size n , kernel size k , padding p , and stride s :

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1.$$

- $n + 2p - k$: distance from the first valid start to the last possible start.
- Divide by s and round down: number of full moves.
- Add one: include the first window.

Example: $n = 7$, $k = 3$, $p = 1$, $s = 2$ gives $\lfloor (7 + 2 - 3)/2 \rfloor + 1 = 4$.

Next: channels. Their output count follows a different rule.

| One filter spans all input channels and creates **one output channel**

For RGB input, a 5×5 spatial patch contains values from **3 color channels**:

$$\underbrace{3 \times 5 \times 5}_{\text{input channels} \times \text{kernel height} \times \text{kernel width}} \longrightarrow \text{weighted sum} + b.$$

At each position, the filter combines **all input channels** into one value. Sliding that same filter across the image creates **one output channel (one feature map)**.

One bias per filter, shared across all spatial positions.

Input channels	One filter contains	Output from one filter
Grayscale: 1	25 weights + 1 bias = 26 parameters	1 output channel
RGB: 3	75 weights + 1 bias = 76 parameters	1 output channel

| Multiple filters create **multiple output channels**

One filter $\rightarrow$ **one output channel**; c_{out} **filters** $\rightarrow c_{\text{out}}$ **output channels**.

$$[B, c_{\text{in}}, h, w] \longrightarrow [B, c_{\text{out}}, h_{\text{out}}, w_{\text{out}}].$$

Part of the output	What determines it?
Batch size B	Unchanged: apply the same filters to each example
Output channels c_{out}	Number of learned filters we choose
Height and width	Kernel size, padding, and stride

`nn.Conv2d(3, 16, kernel_size=5, padding=2)` maps **[64, 3, 28, 28]** $\rightarrow$ **[64, 16, 28, 28]**.

The weight tensor has shape **[16, 3, 5, 5]**: sixteen filters, each spanning three input channels.

Parameter reuse is a core neural-network design principle

Same input/output: $c \times h \times w$; counts below use $1 \times 28 \times 28$, including biases.

MAC = multiply-accumulate; excludes bias additions. Kernels are $k \times k$, stride 1.

Layer	Parameters	MACs / image	Parameter complexity	MAC complexity
Fully connected	615,440	614,656	$O(c^2 h^2 w^2)$	$O(c^2 h^2 w^2)$
Conv 3×3 , padding 1	10	7,056	$O(k^2 c^2)$	$O(k^2 c^2 h w)$
Conv 5×5 , padding 2	26	19,600	$O(k^2 c^2)$	$O(k^2 c^2 h w)$

Parameters = weights + biases: convolution $k^2 c^2 + c$; fully connected $(c h w)^2 + c h w$.

- **Efficiency:** local connections reduce computation; shared weights reduce parameter memory.
- **Generalization:** reuse can help when the same patterns recur across positions.
- **Looking ahead:** transformers also reuse parameters across positions.

Networks connect filters through compatible shapes

Image structure · 3/3 Networks · Patterns → Filters → **Networks**

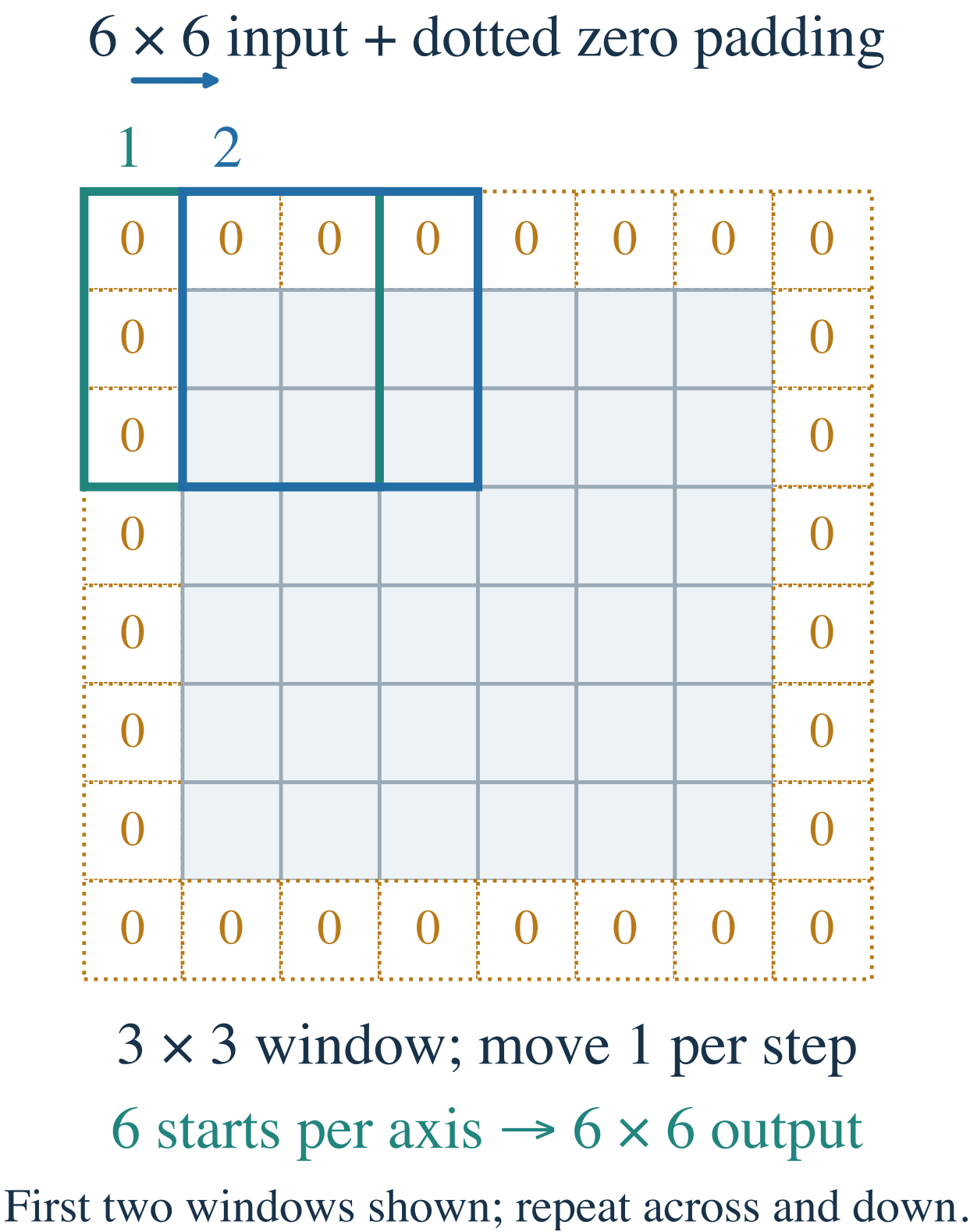
| Odd kernels with matching padding **preserve spatial size**

With stride 1, choose $p = (k - 1)/2$ for odd k .

Kernel	Padding	Size 28 → output size
1×1	0	$28 - 1 + 1 = 28$
3×3	1	$28 + 2 - 3 + 1 = 28$
5×5	2	$28 + 4 - 5 + 1 = 28$

Apply to height and width: $28 \times 28 \rightarrow 28 \times 28$.

Same spatial size does not mean same channel count.
We choose the number of filters.



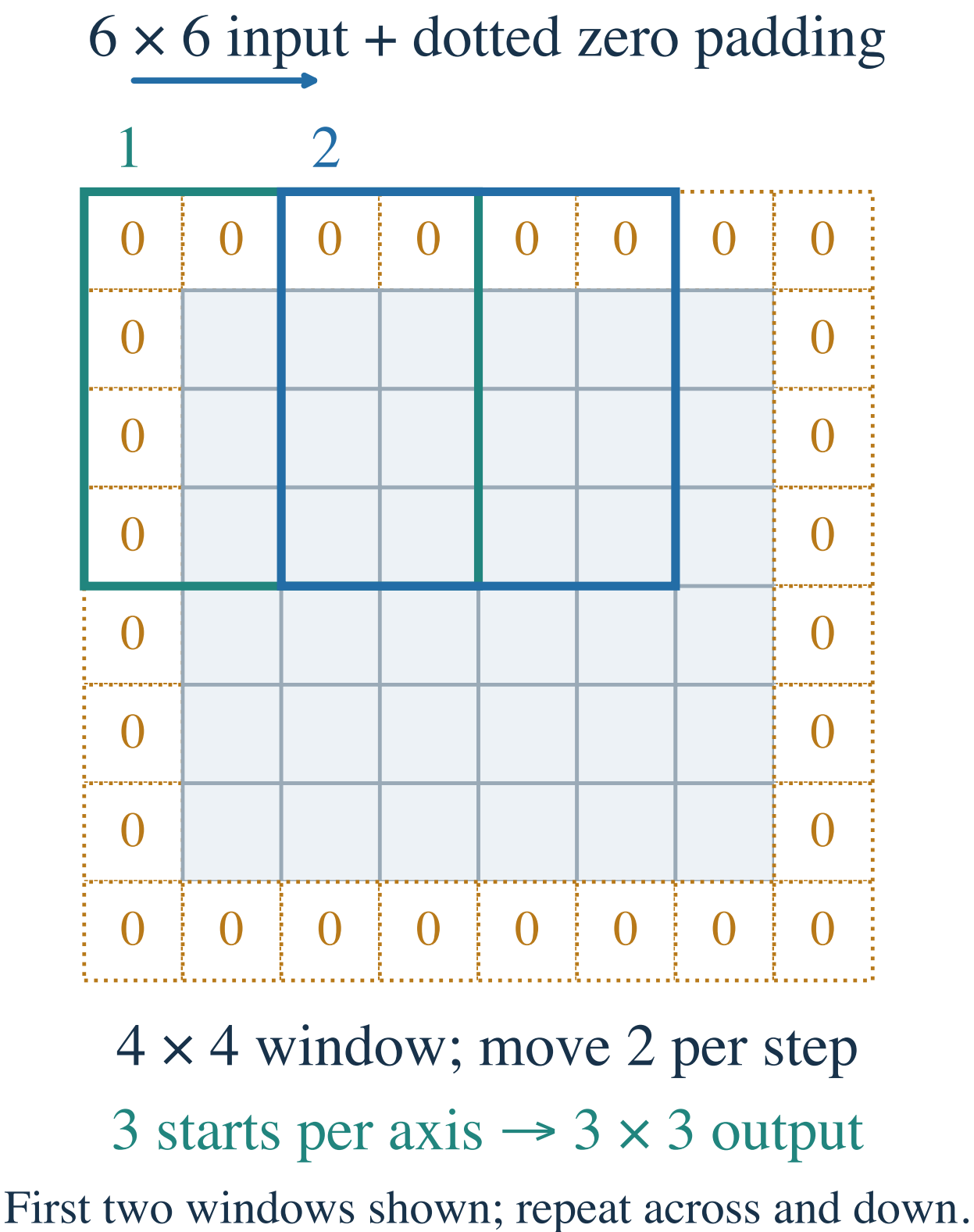
| Stride two can halve each spatial dimension

With **stride 2**, count starts along each axis:

Kernel	Padding	Size 28 → output size
2×2	0	$\lfloor (28 - 2)/2 \rfloor + 1 = 14$
4×4	1	$\lfloor (28 + 2 - 4)/2 \rfloor + 1 = 14$
3×3	1	$\lfloor (28 + 2 - 3)/2 \rfloor + 1 = 14$

$28 \times 28 \rightarrow 14 \times 14$: one quarter as many spatial locations.

Check odd sizes: from 27, the first two cases produce 13; the third produces 14.



| A 1×1 convolution mixes channels at each location

At each pixel location, apply the same linear layer to its vector of channel values.

$$[B, 16, 28, 28] \xrightarrow{1 \times 1, s=1, p=0} [B, 8, 28, 28].$$

- No neighboring pixels are combined.
- Sixteen input channels become eight learned output features at each location.
- Parameter count with bias: $8(16 + 1) = 136$.

A one-pixel kernel can still learn a useful transformation.

| Stacking local layers **combines increasingly broad patterns**

Two 3×3 convolutions with stride 1 let an interior output depend on a 5×5 **input region**.

$$1 \times 1 \xrightarrow{\text{first } 3 \times 3} 3 \times 3 \xrightarrow{\text{second } 3 \times 3} 5 \times 5.$$

This dependency region is the **receptive field**. Each new local window combines neighboring features from the previous layer.

Insert ReLU between layers to learn nonlinear combinations of those features.

A small CNN changes the model but reuses the training loop

```
1 cnn = nn.Sequential(  
2     nn.Conv2d(1, 8, 3, padding=1),  
3     nn.ReLU(),  
4     nn.Conv2d(8, 16, 3, stride=2, padding=1),  
5     nn.ReLU(),  
6     nn.Flatten(),  
7     nn.Linear(16 * 14 * 14, 10))  
8 a = torch.zeros(64, 1, 28, 28)  
9 for layer in cnn:  
10     a = layer(a)  
11     print(type(layer).__name__, tuple(a.shape))
```

Output:

```
1 Conv2d (64, 8, 28, 28)  
2 ReLU (64, 8, 28, 28)  
3 Conv2d (64, 16, 14, 14)  
4 ReLU (64, 16, 14, 14)  
5 Flatten (64, 3136)  
6 Linear (64, 10)
```