

CNN from scratch

Aayush Bajaj

July 10, 2026

Contents

1	convolution vs cross-correlation	1
2	parameter sharing and equivariance	2
3	the geometry: padding, stride, dilation, pooling	2
4	receptive fields	3
5	canonical architectures	4
6	code: conv2d from scratch, verified	4
7	see also	6

a convolutional neural network is a **feedforward network** with its linear layers put on a diet: instead of every unit seeing every input, each unit sees a small local window, and every window is processed by the *same* small set of weights. this page builds the operation from its definition, gets the geometry formulas straight, walks the canonical architectures, and ends with a convolution written in loops and checked against scipy (Goodfellow, Ian, 2016).

the licence for this diet is an assumption about images: nearby pixels are related, faraway pixels mostly are not, and a feature worth detecting is worth detecting everywhere

1 convolution vs cross-correlation

precision first, because the field is sloppy here. for a 2-d input I and kernel K , the *discrete convolution* is

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n), \quad (1)$$

while the *cross-correlation* is

$$(I \star K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n). \quad (2)$$

the difference is a 180-degree flip of the kernel: $I * K = I \star \tilde{K}$ where $\tilde{K}(m, n) = K(-m, -n)$. consequences worth knowing:

- true convolution is commutative ($I * K = K * I$) and is what signal processing and mathematics mean by the word; the flip is what makes the fourier convolution theorem hold.

- virtually every deep-learning “convolutional” layer computes *cross-correlation* (Goodfellow, Ian, 2016). nobody cares in practice: the kernel is learned, and a learned flipped kernel is just another kernel. the distinction only bites when you import fixed kernels (sobel, gaussian) or compare against `scipy` — as the code below demonstrates by matching both conventions.
- with multiple input channels the sum also runs over channels: a $k \times k$ kernel on a C -channel input is really a $C \times k \times k$ tensor producing one output channel; a layer stacks F of them.

2 parameter sharing and equivariance

a fully-connected layer from a $224 \times 224 \times 3$ image to 4096 units needs about 616 million weights. a conv layer with 64 kernels of size $3 \times 3 \times 3$ needs 1,728 weights (1,792 with biases) — five orders of magnitude fewer. two structural priors buy this:

- **sparse connectivity**: each output unit depends only on a $k \times k$ window, so weight count scales with kernel size, not image size.
- **parameter sharing**: the same kernel slides everywhere. the layer’s answer to “is there an edge here?” is computed by the same detector at every location — one feature, one set of weights, reused $H \times W$ times. the *perceptron*’s “error times input” learning signal is correspondingly *summed* over all positions where the kernel was applied.

sharing has a theorem-shaped consequence: **translation equivariance**. shifting the input shifts the output by the same amount,

$$(T_{\Delta} I) \star K = T_{\Delta} (I \star K), \quad (3)$$

where T_{Δ} translates by Δ (exactly true for stride 1 and infinite/periodic support; boundaries and striding break it at the edges). note this is equivariance, not invariance — the response *moves with* the cat. invariance to small shifts is added separately, by pooling.

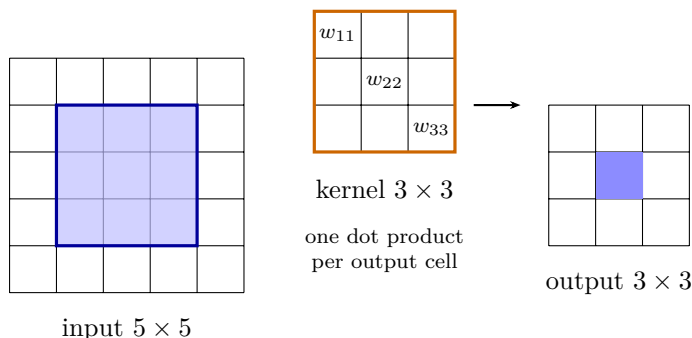


Figure 1: a 3×3 kernel slides over a 5×5 input (valid padding, stride 1) producing a 3×3 output map. the shaded window is the receptive field of the shaded output cell.

3 the geometry: padding, stride, dilation, pooling

for a square input of side n , kernel side k , symmetric zero-padding p , and stride s , the output side is

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1. \quad (4)$$

- **padding** controls boundary loss. $p = 0$ (“valid”) shrinks the map by $k - 1$; $p = (k - 1)/2$ with odd k (“same”, stride 1) preserves size — which is why kernels are odd-sided.
- **stride** subsamples: $s = 2$ roughly halves each side, quartering the pixel count and doubling the spatial reach of everything above it.
- **dilation** inflates the kernel’s footprint without adding weights: spacing taps d apart gives an effective kernel side $k' = k + (k - 1)(d - 1)$, which replaces k in the formula above. cheap receptive-field growth, popular in segmentation.
- **pooling** replaces a neighbourhood with a summary statistic — max or average over (typically) a 2×2 window at stride 2. max-pooling buys approximate invariance to translations smaller than the window, discards exact position, and has no parameters. its output size obeys the same formula. modern architectures often drop explicit pooling in favour of stride-2 convolutions (a *learned* downsampler).

4 receptive fields

the receptive field of a unit is the patch of input pixels that can influence it. stack layers and it compounds: with kernel sides k_ℓ and strides s_ℓ ,

$$r_L = 1 + \sum_{\ell=1}^L (k_\ell - 1) \prod_{i=1}^{\ell-1} s_i, \quad (5)$$

so with all strides 1 the field grows *linearly* — three stacked 3×3 layers see 7×7 — while every stride-2 layer *doubles* the growth rate of everything above it. this is the arithmetic behind “deep nets see globally through local ops”, and behind vgg’s discovery that stacked small kernels beat single large ones: three 3×3 layers match a 7×7 ’s field with $27C^2$ weights instead of $49C^2$, plus two extra nonlinearities (Simonyan, Karen and Zisserman, Andrew, 2015).

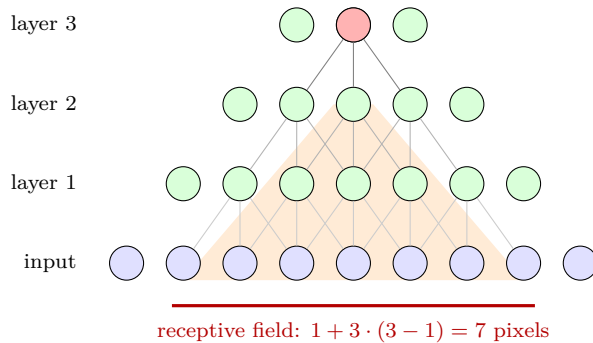


Figure 2: the receptive-field pyramid (1-d slice, kernel 3, stride 1): a single unit three layers up is influenced by seven input pixels. each layer adds $k - 1 = 2$ to the field.

5 canonical architectures

lenet-5 (1998). the blueprint: alternate convolution and subsampling, then flatten into fully-connected layers — conv, pool, conv, pool, fc, fc, ten digits out. trained end-to-end by backpropagation to read cheques, at a time when the rest of pattern recognition was hand-crafted features plus a classifier. everything since is elaboration (LeCun, Yann and Bottou, Léon and Bengio, Yoshua and Haffner, Patrick, 1998).

alexnet (2012). lenet scaled onto two gpus and 1.2 million imagenet images: eight learned layers, relu activations (training deep nets suddenly several times faster than with saturating units), dropout against overfitting, aggressive data augmentation. its 2012 imagenet win — a top-5 error around 15% against roughly 26% for the best non-neural runner-up — is the conventional starting gun of the deep-learning era (Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E., 2012).

vgg (2014). an ablation in discipline: only 3×3 kernels, only 2×2 pooling, doubling channel width at each downsampling, to 16–19 layers. proved that *depth with uniform small kernels* is the winning recipe (see the receptive-field arithmetic above) and became the field’s default feature extractor for years — at the cost of a bloated 138 million parameters, most of them in the final fc layers (Simonyan, Karen and Zisserman, Andrew, 2015).

resnet (2015). naively stacking past ~20 layers made networks *worse on the training set* — a degradation problem, not overfitting: deeper nets failed to even emulate their shallower selves plus identity layers. the fix reframes the task: let a block of layers learn the *residual* $F(x) = H(x) - x$ and output

$$y = F(x) + x. \tag{6}$$

if the identity is what is needed, the block only has to drive F to zero — trivial — instead of contorting nonlinear layers into an identity map. and the shortcut is a gradient highway: $\partial y / \partial x = \partial F / \partial x + I$, so backpropagation always has an unattenuated additive path to earlier layers, the depth-wise twin of the **lstm**’s constant error carousel. with identity shortcuts, 152-layer networks trained cleanly and won imagenet 2015 (He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian, 2015).

the identity term is doing the same job the forget-gate-near-one does in an lstm: protect the gradient from the multiplication mill

batch normalisation (2015), the enabling ingredient threaded through resnet and successors: normalise each channel’s pre-activations to zero mean and unit variance over the mini-batch, then restore expressiveness with a learned scale γ and shift β per channel. it stabilises the distribution each layer sees, permits much larger learning rates, and adds a mild regularising noise (batch statistics fluctuate); at inference the batch statistics are replaced by running averages collected during training.

6 code: conv2d from scratch, verified

naive loops — one dot product per output pixel — checked against `scipy.signal` under *both* conventions, then the output-size formula, then a sobel kernel doing its one famous trick:

```
1 import numpy as np
2 from scipy.signal import correlate2d, convolve2d
3
4 def conv2d(x, k, stride=1, pad=0, flip=False):
```

```

5  """naive 2d convolution (flip=True) / cross-correlation (flip=False)."""
6  if flip:
7      k = k[::-1, ::-1]          # true convolution flips the
      ↪ kernel
8  if pad:
9      x = np.pad(x, pad)
10     n, m = x.shape
11     kh, kw = k.shape
12     oh = (n - kh) // stride + 1
13     ow = (m - kw) // stride + 1
14     out = np.zeros((oh, ow))
15     for i in range(oh):
16         for j in range(ow):
17             patch = x[i*stride : i*stride + kh, j*stride : j*stride + kw]
18             out[i, j] = np.sum(patch * k)      # one dot product per output
             ↪ pixel
19     return out
20
21 rng = np.random.default_rng(3)
22 x = rng.normal(size=(7, 7))
23 k = rng.normal(size=(3, 3))
24
25 # --- verify against scipy, both conventions ---
26 mine_corr = conv2d(x, k)          # cross-correlation,
      ↪ valid
27 ref_corr = correlate2d(x, k, mode="valid")
28 mine_conv = conv2d(x, k, flip=True) # true convolution,
      ↪ valid
29 ref_conv = convolve2d(x, k, mode="valid")
30 print("cross-correlation matches scipy:", np.allclose(mine_corr, ref_corr))
31 print("true convolution matches scipy: ", np.allclose(mine_conv, ref_conv))
32 print("corr == conv without the flip: ", np.allclose(mine_corr, mine_conv))
33
34 # --- the output-size formula: floor((n + 2p - k)/s) + 1 ---
35 for (n, p, kk, s) in [(7, 0, 3, 1), (7, 1, 3, 1), (7, 1, 3, 2), (28, 2, 5, 1)]:
36     out = conv2d(rng.normal(size=(n, n)), np.ones((kk, kk)), stride=s, pad=p)
37     formula = (n + 2*p - kk) // s + 1
38     print(f"n={n:>2} p={p} k={kk} s={s}: output {out.shape[0]:>2} formula
      ↪ {formula:>2} agree {out.shape[0] == formula}")
39
40 # --- an edge detector, to see the operation do something ---
41 img = np.zeros((8, 8)); img[:, 4:] = 1.0      # vertical edge at
      ↪ column 4
42 sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], float)
43 resp = conv2d(img, sobel_x)
44 print("\nsobel response (nonzero only along the edge):")
45 print(resp.astype(int))

```

```

cross-correlation matches scipy: True
true convolution matches scipy: True
corr == conv without the flip: False
n= 7 p=0 k=3 s=1: output 5 formula 5 agree True
n= 7 p=1 k=3 s=1: output 7 formula 7 agree True
n= 7 p=1 k=3 s=2: output 4 formula 4 agree True
n=28 p=2 k=5 s=1: output 28 formula 28 agree True

```

```

sobel response (nonzero only along the edge):
[[ 0  0 -4 -4  0  0]
 [ 0  0 -4 -4  0  0]
 [ 0  0 -4 -4  0  0]
 [ 0  0 -4 -4  0  0]
 [ 0  0 -4 -4  0  0]

```

[0 0 -4 -4 0 0]]

three sanity checks in one run: the loop implementation agrees with `scipy` under both flip conventions (and the two conventions genuinely differ on a random kernel); the output-size formula predicts every shape including the strided and padded cases; and the sobel kernel responds only in the two columns whose 3×3 window straddles the edge — locality made visible. (the third check is also the flip distinction in disguise: sobel responds with sign -4 here because cross-correlation does *not* flip it.)

7 see also

- **perceptron** — the unit whose weights the kernel shares
- **multilayer perceptron** — what the flattened head of a cnn is
- **feedforward deep neural networks** — training machinery, backprop and regularisation
- **long short-term memory** — the same additive-shortcut idea, applied through time
- **transformers** — attention as dynamic, content-dependent weight sharing

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian (2015). *Deep Residual Learning for Image Recognition*.

Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*.

LeCun, Yann and Bottou, Léon and Bengio, Yoshua and Haffner, Patrick (1998). *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE.

Simonyan, Karen and Zisserman, Andrew (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*.