Convolutional Networks: From Sliding Windows to ResNets

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

Prerequisites: ***NN_BASICS*** *(layers as $XW + \mathbf{b}$, ReLU, softmax + cross-entropy, the training loop, vanishing gradients). Optional companion:* ***CNN_ADVANCED****.*

1 Recap and the Big Picture: an MLP Looking at an Image

HW5's network for Fashion-MNIST flattened every 28×28 image into 784 numbers and fed them to a fully connected layer of 256 units. That layer has $784 \cdot 256 + 256 = 200{,}960$ weights, one for every (pixel, unit) pair; with the output layer, the MLP has **203,530** parameters. Two things are wrong with this for images:

- **It ignores where pixels are.** Shuffle the 784 pixels the same way in every image and the MLP learns exactly as well: it has no idea that pixel 300 is next to pixel 301.
- **It must learn the same thing many times.** A sleeve at the left of the image and a sleeve at the right hit completely different weights. Whatever the network learned about one does not transfer to the other.

The instructor's board (Lecture CNN, p2) says what we want instead: *to recognize a pattern, compute on a small frame of the image; use the same computation for every frame* — "frame × kernel window", with $\text{Frame}_1 \times K \approx \text{Frame}_2 \times K$ when the two frames show the same pattern. These are the **two issues** a layer for images must handle:

1. **Locality.** A pixel's meaning depends on its neighbours, not on pixels far away, so each output should look only at a small window.
2. **The same detector at every position.** A pattern means the same wherever it appears, so the *same* weights should be used at every position (**weight sharing**).

Careful with what this buys: if the sleeve moves, the detector's *response map moves with it*; the map is not unchanged. Whether the final *label* stays the same is a separate question (pooling helps a little, §4; augmentation teaches the rest, §8).

A layer that does both is a **convolution**. Stack a few of them, shrink the image between them (**pooling**), and finish with a small MLP: that is a **convolutional neural network (CNN)**. Figure 1 is the first famous one, LeNet (LeCun et al., 1998), with the shapes you will see in HW6. It has **61,706** parameters, under a third of the MLP's, and (once modernized, §5) it is at least as accurate.

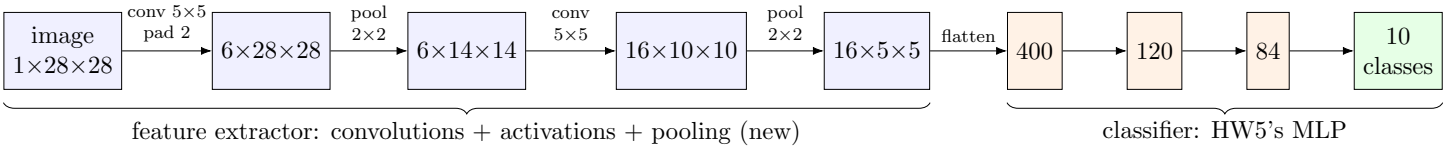


Figure 1: LeNet on a Fashion-MNIST image. Each box is the tensor between layers (channels × height × width).

Almost everything in a CNN is something you have already built (Table 1).

Part	**Where you have seen it**	**What changes**
Weighted sum + bias, then ReLU	every layer of HW5's MLP	the sum runs over a small window only
Softmax + cross-entropy, Adam, minibatches	HW5	nothing
Backprop through stacked layers	NN_BASICS	PyTorch does it for convolutions too
Weighted average over neighbours	kernel smoothers (Weeks 2–3)	the weights are *learned*
Standardizing features	HW1 (z-scores)	done inside the network, per channel, per batch
Convolution: one kernel at every position (§2–3)	**NEW**	weight sharing + locality
Pooling (§4)	**NEW** (an easy layer)	no parameters
Batch normalization (§6)	**NEW**	a layer with a training and an evaluation mode
Residual (skip) connections (§7)	**NEW**	$x + R(x)$ instead of $R(x)$

Table 1: The parts of a CNN. Five are reused; four are new.

Remark. A roadmap for Weeks 11–14 (the instructor's timeline, Lecture AdvNN p1): CNNs for images (≈1995, this note) → recurrent networks for sequences (≈2009, Week 12) → attention (2014) → the Transformer (2017) → large GPT models (2022), Weeks 13–14. Two ideas from this note come back in every Transformer: the residual connection (§7) and a normalization layer (§6).

2 Cross-Correlation: One Kernel Slid over the Image

Take a single-channel image X ($n_h \times n_w$) and a small **kernel** K ($k_h \times k_w$). Place the kernel on the top-left window of the image, multiply entry by entry, add up: that is the first output. Slide one pixel right and repeat; at the end of a row, move one row down. In symbols, with i, j the output position (in this note i, j index *pixels*, not datapoints):

$$Y_{i,j} \;=\; b + \sum_{a=0}^{k_h-1} \sum_{b'=0}^{k_w-1} K_{a,b'}\, X_{i+a,\, j+b'}. \tag{1}$$

Read it as: *output (i, j) is how strongly the window starting at (i, j) matches the kernel's pattern.* The same $k_h k_w$ weights (and one bias b) are used at every position. The output is smaller than the input: $(n_h - k_h + 1) \times (n_w - k_w + 1)$.

Worked example: the lecture's 3×3 image. $X = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$, $K = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$, $b = 0$. Top-left window: $0\cdot0 + 1\cdot1 + 3\cdot2 + 4\cdot3 = 19$. Moving right: $1\cdot0 + 2\cdot1 + 4\cdot2 + 5\cdot3 = 25$. The full output is $Y = \begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$ (Figure 2). This is HW6's first checkpoint.

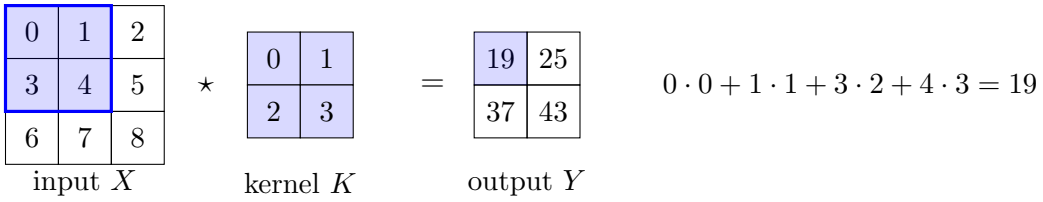


Figure 2: Cross-correlation: the shaded window of X times K, summed, gives the shaded output.

"Convolution" is really cross-correlation. In mathematics a convolution flips the kernel first (it uses $X_{i-a,\,j-b}$); Eq. 1 does not, which makes it a *cross-correlation*. That is why the board crosses out "2-D convolution". For a CNN the difference does not matter: the kernel is learned, so a network trained with one convention simply learns the flipped kernel of the other. Deep learning calls Eq. 1 "convolution" anyway, and so does PyTorch: its `F.conv2d` and `nn.Conv2d` do *not* flip.

What a kernel detects. Take a white image with a dark vertical bar (rows like $1, 1, 0, 0, 0, 0, 1, 1$) and the kernel $K = [1, -1]$. Each output is "this pixel minus its right neighbour": $0, 1, 0, 0, 0, -1, 0$ — a $+1$ where white turns dark and a -1 where dark turns white, zero elsewhere (Figure 3). This kernel is a **vertical-edge detector**. Move the image one pixel right and the edge map moves one pixel right with it. Hand-designed kernels like this (edges, blur, sharpen) were the old way; a CNN learns its kernels from data, and the first layer of a trained network ends up with edge and colour detectors much like these.

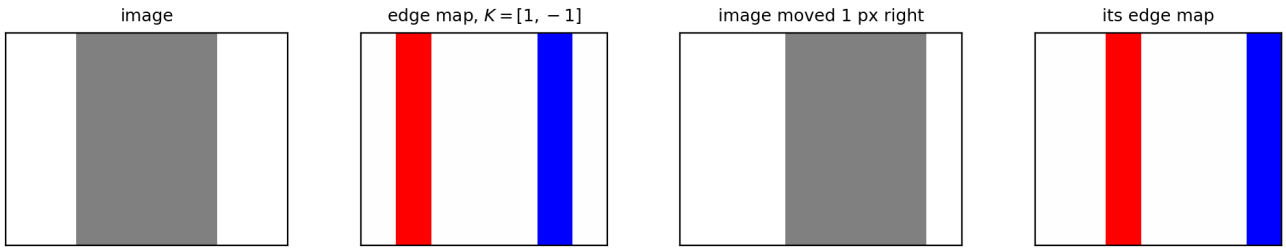


Figure 3: A vertical-edge kernel. White = 1, black = 0 in the images; red = $+1$, blue = -1 in the edge maps. Shifting the image shifts the edge map by the same amount.

Padding. Each layer shrinks the image by $k - 1$ pixels, and border pixels are covered by fewer windows than central ones. The instructor's two reasons to **pad** (add zero rows and columns around the image): to compute outputs at the border ($i = 1$ or $i = D$), and to keep the image size standard from layer to layer. With a 5×5 kernel and padding 2 on every side, a 28×28 image stays 28×28 (LeNet's first layer).

Stride. The **stride** is the step of the window (default 1). Stride 2 halves the output's height and width: a cheap way to shrink the image fast. With total padding p_h (both sides together) and stride s_h, the output height is

$$\left\lfloor \frac{n_h - k_h + p_h + s_h}{s_h} \right\rfloor, \tag{2}$$

and the same for the width. Check: $n = 28$, $k = 5$, $p = 4$ (2 per side), $s = 1$: 28.

Worked example: padding and stride on the 3×3 image. Padding 1 on every side ($p_h = p_w = 2$) gives a 4×4 output, $\begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$: the original 2×2 block sits in the middle. Adding stride $(3, 2)$ keeps every third row and every second column: $\begin{bmatrix} 0 & 8 \\ 6 & 8 \end{bmatrix}$; Eq. 2 gives $\lfloor (3 - 2 + 2 + 3)/3 \rfloor = 2$ and $\lfloor (3 - 2 + 2 + 2)/2 \rfloor = 2$.

Remark. PyTorch's `padding=1` means one row/column *on each side* ($p_h = 2$ in Eq. 2).

3 Channels, and Convolution as One Matrix Product

Several input channels. A colour image has three channels (R, G, B), so it is a $3 \times n_h \times n_w$ tensor. The kernel gets one $k_h \times k_w$ slice *per input channel*, and the results are **added**: one output map that looks at all colours at once. (The board: "ch1: red pixel values, K for red; ch2: blue ...; different kernels".)

Several output channels. One kernel detects one pattern. To detect c_o patterns (edges at several angles, colour blobs, ...), use c_o kernels and **stack** their output maps: the output has c_o channels, and these are the input channels of the next layer. The general formula, written on the board, with d the output channel and c the input channel:

$$Y_{i,j,d} \;=\; b_d + \sum_a \sum_{b'} \sum_c K_{a,b',c,d}\; X_{i+a,\, j+b',\, c}. \tag{3}$$

The layer has $c_o \cdot c_i \cdot k_h k_w + c_o$ parameters, *independent of the image size*. LeNet's second convolution ($6 \to 16$ channels, 5×5) has $16 \cdot 6 \cdot 25 + 16 = 2{,}416$; a fully connected layer between the same two tensors ($6 \cdot 14 \cdot 14 \to 16 \cdot 10 \cdot 10$) would have about 1.9 million.

Worked example: two input channels. Channel 0 is the 3×3 image above with kernel slice $\left[\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right]$; channel 1 is $\left[\begin{smallmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{smallmatrix}\right]$ with slice $\left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$. Adding the two single-channel outputs gives $\left[\begin{smallmatrix} 56 & 72 \\ 104 & 120 \end{smallmatrix}\right]$.

Remark. PyTorch stores a convolution's weight as (c_o, c_i, k_h, k_w): output channel first. A 1×1 **convolution** ($k_h = k_w = 1$) mixes channels at each pixel and nothing else: it is a fully connected layer from c_i to c_o numbers, applied separately at every pixel. ResNets use it to change the number of channels (§7).

Convolution as one matrix product ("unfold", also called im2col). Loops over output positions are slow. Libraries instead copy every input window into one column of a **patch matrix** P; then *all outputs at all positions* are one matrix product, the operation GPUs do best:

$$\underbrace{Y}_{c_o \times (h'w')} \;=\; \underbrace{W}_{c_o \times (c_i k_h k_w)} \;\; \underbrace{P}_{(c_i k_h k_w) \times (h'w')}, \tag{4}$$

where row d of W is output channel d's kernel flattened, and column p of P is the input window at output position p, flattened in the same order. This is the HW5 layer XW again, only with "examples" replaced by "windows".

Worked example: the patch matrix of the 3×3 image. With a 2×2 kernel there are four windows; written as rows ($P^\top$):

$$P^\top = \begin{bmatrix} 0 & 1 & 3 & 4 \\ 1 & 2 & 4 & 5 \\ 3 & 4 & 6 & 7 \\ 4 & 5 & 7 & 8 \end{bmatrix}, \qquad P^\top \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 19 \\ 25 \\ 37 \\ 43 \end{bmatrix},$$

the same numbers as Figure 2, now from one product.

How to build P without looping over positions. Look at it one kernel weight at a time. Weight $K_{a,b'}$ multiplies, at output position (i, j), the pixel $X_{i+a,\, j+b'}$. Over *all* output positions these pixels form a shifted $h' \times w'$ slice of the image: rows a to $a + h' - 1$, columns b' to $b' + w' - 1$. For the example and offset $(a, b') = (0, 1)$ that slice is $\left[\begin{smallmatrix} 1 & 2 \\ 4 & 5 \end{smallmatrix}\right]$ — exactly the second entries of the four windows. So P is just the $k_h k_w$ shifted slices (one per kernel offset, per channel), each flattened into one row. The cost is memory: every pixel is copied up to $k_h k_w$ times, which is why your from-scratch LeNet in HW6 runs several times slower than PyTorch's optimized convolution, while computing the same numbers.

Key point:

Shapes in code (a minibatch of n images). PyTorch keeps the channel axis *before* the pixels: images are (n, c_i, h, w), while Eq. (3) lists the channel last only to read like the board. With $h' = h - k_h + 1$, $w' = w - k_w + 1$:

$$X : (n, c_i, h, w) \xrightarrow{\text{unfold}} P : (n, c_i k_h k_w, h'w') \xrightarrow{W:(c_o,\, c_i k_h k_w) \text{ times } P} (n, c_o, h'w') \xrightarrow{\text{reshape}} Y : (n, c_o, h', w').$$

The batch axis just rides along: the same W multiplies every image's patch matrix.

4 Pooling: Shrink, and Forgive Small Moves

A **pooling** layer slides a window over each channel and outputs one number per window: the **maximum** (max pooling) or the **average** (average pooling). It has no parameters ("an EASY layer", the board says) and works on each channel separately. The board's example: the window $\left[\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right]$ pools to 3 (average) or 6 (max). On the 3×3 image with a 2×2 window and stride 1: max $\left[\begin{smallmatrix} 4 & 5 \\ 7 & 8 \end{smallmatrix}\right]$, average $\left[\begin{smallmatrix} 2 & 3 \\ 5 & 6 \end{smallmatrix}\right]$.
In a CNN pooling usually uses non-overlapping 2×2 windows (stride 2), so it halves height and width. It does two jobs:

- **Simplify.** Fewer numbers for the next layer, and each later layer sees a larger part of the image.
- **Forgive small moves.** Max pooling asks "did this detector fire *somewhere* in the window?", not "exactly where?". Max keeps the strongest response ("better for ML", the board says); average blurs it.

Worked example: pooling *reduces* shift sensitivity; it does not remove it. A 1-D edge response with a single 1, max-pooled with window 2 and stride 2:

edge at position	response	max-pooled	
2	$0,0,1,0,0,0,0,0$	$0,1,0,0$	
3 (moved 1)	$0,0,0,1,0,0,0,0$	$0,1,0,0$	unchanged: still in the same window
4 (moved 2)	$0,0,0,0,1,0,0,0$	$0,0,1,0$	changed: crossed a window boundary

A small move *within* a window disappears; a move across a boundary does not.

Key point: Convolution *moves with* the input: shift the image and the feature maps shift the same way (**equivariance**). Pooling *reduces* the effect of small shifts, but only sometimes removes it. Neither does anything about rotation or a change of size: a vertical-edge kernel simply does not fire on a horizontal edge. Those must be learned from examples (§8).

5 LeNet, Then and Now

LeNet (Figure 1) is two convolution + pooling blocks followed by a three-layer MLP:

layer	output shape	what it does	parameters
conv 5×5, $1 \to 6$, pad 2	$6 \times 28 \times 28$	6 pattern detectors	$6 \cdot 25 + 6 = 156$
pool 2×2	$6 \times 14 \times 14$	halve	0
conv 5×5, $6 \to 16$	$16 \times 10 \times 10$	16 detectors of detector patterns	$16 \cdot 6 \cdot 25 + 16 = 2{,}416$
pool 2×2	$16 \times 5 \times 5$	halve	0
flatten, dense $400 \to 120$	120		48,120
dense $120 \to 84$	84		10,164
dense $84 \to 10$	10	class scores (softmax in the loss)	850
total			**61,706**

Note where the parameters are: 78% sit in the first dense layer, almost none in the convolutions. The convolutions are cheap in parameters and expensive in computation (each weight is used at every position).

Algorithm: the forward pass of a CNN

1. Input: a minibatch X of shape (n, c, n_h, n_w).
2. For each convolution block: convolution (Eq. 3) $\to$ [batch norm] $\to$ activation $\to$ [pooling]. The shape becomes (n, c_o, h', w').
3. Flatten to $(n, c_o h' w')$; then the dense layers and the activations, as in HW5.
4. Logits $(n, 10)$ $\to$ softmax + cross-entropy. Backprop and Adam exactly as in HW5.

1998 vs today. The original LeNet used **sigmoid** activations and **average** pooling. Modern CNNs use **ReLU** (its slope is 1 for positive inputs, so the gradient does not shrink by a factor $\leq \frac{1}{4}$ at every layer, as with the sigmoid; NN_BASICS) and **max** pooling. What to expect when you compare them with HW5's MLP (same optimizer, same number of epochs):

- **Fewer parameters is not automatically better.** A sigmoid network learns slowly, so the 1998 LeNet can end up *behind* a plain MLP after a short training budget, even though its design suits images better. Look at whether its curve is still rising when training stops.
- **The modern version** (ReLU + max pooling) should reach the MLP's level with a fraction of its parameters: the architecture, not the parameter count, does the work.
- **Shifted images are where the difference shows.** A kernel detects a pattern wherever it is, so a CNN degrades gracefully when the test images move by a pixel or two; an MLP's weights are tied to pixel positions, so it degrades fast. Neither is shift-proof: the dense layers at the end of LeNet still care about positions, so a large shift hurts both.

In your run, check each of these against the numbers and the shift curve your notebook prints, and use your own numbers in the written answers.

6 Batch Normalization

The problem. In HW1 you standardized features before gradient descent because badly scaled inputs force a small learning rate. Inside a network the same thing happens at every layer, and worse: the inputs to layer 5 are the outputs of layers 1–4, whose scale changes every time those layers are updated. A step that is safe for one layer can blow up or squash the inputs of the layers after it.

The fix: standardize inside the network. A **batch-normalization** (BN) layer standardizes each channel using the statistics of the current minibatch, then lets the network choose the scale it wants. For one channel, collect all its values in the batch: for a convolution's output (n, c, h, w), that is $m = n \cdot h \cdot w$ values (every example *and* every position); for a dense layer's output (n, c), $m = n$ values. Then

$$\hat{\mu} = \frac{1}{m}\sum x, \qquad \hat{\sigma}^2 = \frac{1}{m}\sum (x - \hat{\mu})^2, \tag{5}$$

$$\hat{x} = \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}}, \qquad y = \gamma\,\hat{x} + \beta. \tag{6}$$

Eq. 5 measures the channel's centre and spread in this batch; Eq. 6 makes it mean 0, variance 1 ($\epsilon \approx 10^{-5}$ avoids dividing by zero), then rescales it with two *learned* numbers per channel, γ and β. Without γ, β the layer could never output anything but standardized values; with them it can even undo the standardization if that is best. Figure 4 shows which values are pooled together.

Shapes in code. For an input (n, c, h, w), average over axes $0, 2, 3$: $\hat{\mu}$, $\hat{\sigma}^2$, γ, β and the running averages are all vectors of length c. To combine them with the (n, c, h, w) tensor, view them as $(1, c, 1, 1)$ so they *broadcast*: the same channel value is used for every example and every pixel. For a dense layer's (n, c) output, average over axis 0 and use $(1, c)$.

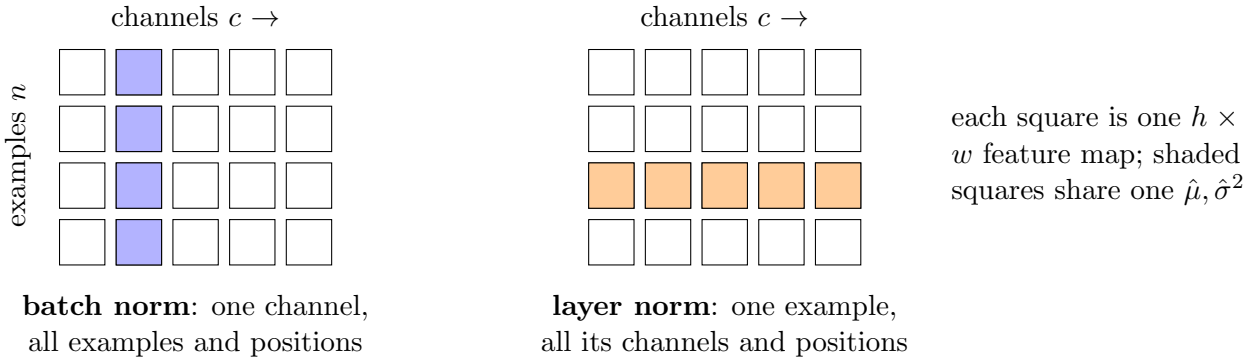


Figure 4: Which values are averaged together. Batch norm: down a column (per channel). Layer norm (HW7's Transformers): along a row (per example).

Worked example: batch norm vs layer norm on a toy batch (the lecture's matrix). Three examples (rows), five features (columns):

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 10 & 20 & 30 & 40 & 50 \\ 7 & 14 & 21 & 28 & 35 \end{bmatrix}.$$

Column 0 is $(1, 10, 7)$: mean 6, biased variance 14, so BN gives $(-1.336,\ 1.069,\ 0.267)$; every column is a multiple of column 0, so *every* column becomes $(-1.336,\ 1.069,\ 0.267)$. Layer norm standardizes each *row*

instead: every row becomes $(-1.414, -0.707, 0, 0.707, 1.414)$. BN removes differences between examples' scales feature by feature; LN removes each example's own scale.

Training mode and evaluation mode. At test time we may classify a single image — there is no batch to take statistics from, and a prediction should not depend on which other images happen to be in the batch. So BN keeps **running averages** of $\hat{\mu}$ and $\hat{\sigma}^2$ during training and uses those at test time:

- **training mode**: Eqs. 5–6 with the batch's statistics, and update running $\leftarrow (1-0.1)\,\text{running} + 0.1 \cdot$ batch value;
- **evaluation mode** (`model.eval()`): Eq. 6 with the running averages; nothing is updated.

Remark. PyTorch conventions to match exactly. (1) `momentum=0.1` is the weight of the *new* batch (the opposite of the β in Adam's averages). (2) `running_var` stores the *unbiased* variance (divide by $m-1$), although the normalization itself uses the biased one. In the toy batch, starting from running_var $= 1$: $0.9 \cdot 1 + 0.1 \cdot 21 = 3.0$ (unbiased variance 21); with the biased 14 it would be 2.3. The running mean after one step is $0.1\times$ the column means, $(0.6, 1.2, \ldots, 3.0)$.

Where it goes: after a convolution or dense layer, before the activation. Every BN layer is its own module, with its own γ, β and running statistics: two BN layers can't share one module, because they normalize different channels with different statistics. (A convolution followed by BN needs no bias: β already shifts.)

What it buys. BN adds only $2c$ parameters per layer (γ, β for each channel; 452 in all for BN-LeNet). Two effects to look for: training gets **faster** (compare the first epoch, not only the last), and it becomes **more forgiving of the learning rate**: raise the learning rate a lot, and the plain network's curve turns bumpy and ends lower, while the BN version keeps working. The usual explanation is the problem we started from: each layer's inputs are re-standardized at every step, so one large update cannot rescale everything downstream. (Why BN helps *exactly* is still debated; CNN_ADVANCED.)

7 Going Deeper: Residual Blocks

Deeper networks can build features in more stages (edges $\rightarrow$ textures $\rightarrow$ parts $\rightarrow$ objects), but plain deep networks are hard to train. NN_BASICS showed why: the gradient reaching an early layer is a *product* of one factor per layer above it, and products of many factors drift to 0 or blow up ($0.5^{10} \approx 10^{-3}$, $0.5^{20} \approx 10^{-6}$).

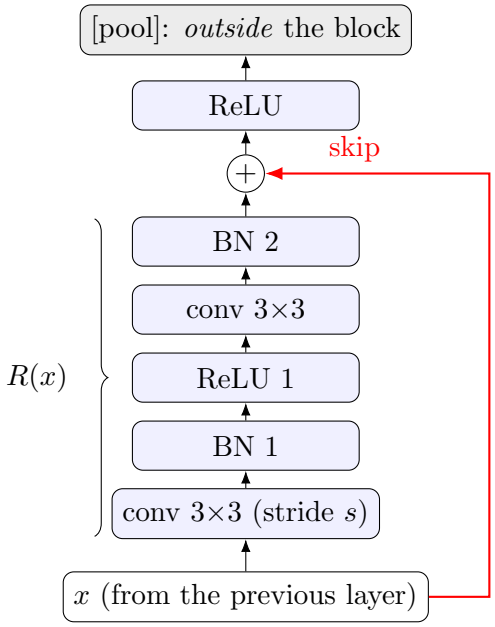
The idea. Don't ask a block to compute its output $H(x)$ from scratch; ask it for the *change* to its input:

$$H(x) \;=\; x + R(x), \tag{7}$$

where R (the **residual**) is a couple of layers and x is added back through a **skip connection**. The instructor's handwritten page lists what this buys:

- the gradient can "**go around**" the block, through the skip, to the layers before it: $\partial H/\partial x = I + \partial R/\partial x$, an identity *plus* something, instead of only a product of small factors;

- if passing x on unchanged is useful, $R(x)$ only has to be close to 0 — easy — instead of having to learn the identity through two convolutions;
- hence deep stacks become **easier to train** (the vanishing-gradient problem eases).



The board's bullets:

- backprop can "go around" via the skip
- easier to train
- $H(x) = x + R(x)$, R = residual
- if $H(x) \approx x$ is useful, $R(x)$ can be ≈ 0
- eases the vanishing gradient
- pooling (if any) goes *after* the $\oplus$, outside the block

If the block changes the shape (more channels, stride $s = 2$), the skip becomes a 1×1 convolution with stride s.

Figure 5: A residual block, after the instructor's handwritten page (Week 11).

When shapes change. The addition $x + R(x)$ needs both terms to have the same shape. If the block doubles the channels or halves the image (stride 2), the skip passes x through a 1×1 convolution with the same stride. Shape walk-through for $16 \to 32$ channels, stride 2, on a 32×32 input: conv 1 gives $32 \times 16 \times 16$, conv 2 keeps $32 \times 16 \times 16$, the 1×1 convolution maps x from $16 \times 32 \times 32$ to $32 \times 16 \times 16$ (with $16 \cdot 32 = 512$ weights), and the sum is $32 \times 16 \times 16$.

Why pooling stays outside. Inside the block it would shrink $R(x)$ but not x, and the sum would no longer line up (the board's last bullet).

Remark. **Two honest caveats.** (1) The skip gives the signal a direct path; it does *not* make a block start as the identity — with ordinary random initialization $R(x)$ is not small at first. It only makes "close to identity" easy to reach. (2) The convolutions in a block have no bias, because BN follows them.

The small ResNet of HW6. A 3×3 stem convolution with 32 channels, then five residual blocks ($32 \to 32$, $32 \to 64$ stride 2, $64 \to 64$, $64 \to 128$ stride 2, $128 \to 128$), global average pooling and one dense layer: 11 convolutions deep, 677,674 parameters. **Global average pooling** (`AdaptiveAvgPool2d(1)`) turns the last block's maps into one vector: each channel's 8×8 map is averaged to a single number, $(n, 128, 8, 8) \to (n, 128, 1, 1) \to$ flatten $\to (n, 128)$, and that vector feeds the dense layer. (It also means the classifier no longer cares *where* in the image a feature fired.) On CIFAR-10 (32×32 colour photos) expect it to beat a batch-normalized LeNet adapted to colour (83,578 parameters) by a wide margin. That much is observed. The comparison changes depth, width *and* shortcuts together, so it does not show which of them caused the gain; the explanation for why a network this deep trains so easily is the one above, and optional Problem 7 separates depth from shortcuts.

Remark. **Forward link.** Every block of a Transformer (Weeks 13–14) is a residual block, $x +$ sublayer(x), followed or preceded by a normalization layer: the two new ideas of §6–7, reused.

8 Training CNNs on Little Data: Augmentation and Transfer

A network as large as the small ResNet can fit CIFAR's 50,000 training images almost perfectly while scoring clearly lower on the test set: a large train/test gap means it is partly memorizing. Two remedies.

Data augmentation. A photo of a cat is still a cat when mirrored left–right or moved by a few pixels. So every time a training image is used, flip it at random (probability $\frac{1}{2}$) and cut a random 32×32 window from the image padded by 4 pixels (Figure 6). The network rarely sees exactly the same image twice, so memorizing becomes much harder, and it learns that these changes do not change the label. **Only training images are augmented**; test images are used as they are.



Figure 6: One CIFAR-10 training image and eight random flip + crop draws. The padded border shows as black here; the notebook pads the *standardized* images, so there the border is the average colour.

What to look for in your run: the training task got harder, so training accuracy should *drop*; the question is what happens to the test accuracy and to the gap. Compare both curves with and without augmentation, not only the final numbers.

Remark. **Equivariance vs invariance.** A function is *equivariant* to shifts if shifting the input shifts the output the same way (convolution, away from the borders); it is *invariant* if the output does not change at all (what a classifier wants). Convolution gives equivariance; pooling and global averaging push toward invariance for small shifts; *augmentation teaches the rest* — but only the changes it actually applies. To know what yours can have taught, list exactly which transformations it applies (and how large); anything not on that list, the network has never been shown.

Transfer learning. Labelled data is often scarce. Suppose we have only 5,000 CIFAR images. Instead of learning features from these few images, borrow them: ResNet-18 was trained on ImageNet (1.3 million photos, 1,000 classes), and its layers already detect edges, textures and object parts. Three options (Figure 7):

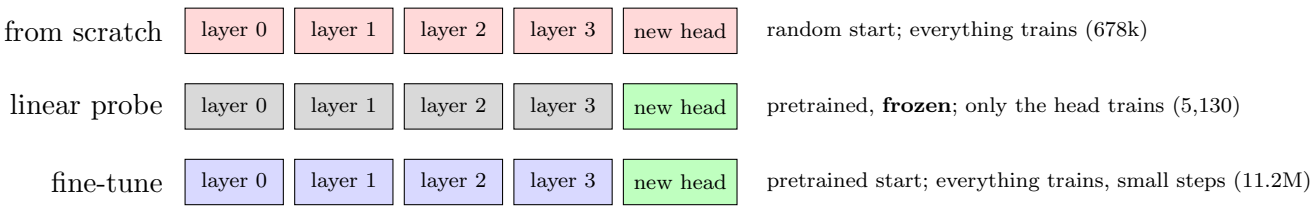


Figure 7: Three ways to use 5,000 labelled images. Grey = frozen pretrained weights; blue = pretrained and trainable; red = random and trainable; green = a new 10-class output layer.

- **From scratch:** HW6's small ResNet trained on the 5,000 images only.
- **Linear probe:** *freeze* the pretrained network, replace its last layer by a new $512 \to 10$ layer, and train only that (5,130 weights).
- **Fine-tuning:** start from the pretrained weights and let *all* of them adapt with a small learning rate.

Frozen is not the same as evaluation mode. Freezing (`requires_grad=False`) stops the *weights* from changing. Evaluation mode (`.eval()`) stops the batch-norm layers from using batch statistics and updating their running averages. A linear probe needs both: otherwise the "frozen" network's BN layers keep drifting. With both, the frozen part computes exactly the same features every time, so the notebook computes them once and trains the probe on the stored features (much faster).

Input preparation. ResNet-18 was pretrained on 224×224 ImageNet photos normalized with ImageNet's channel means and standard deviations. The notebook uses the same normalization, but resizes the 32×32 images only to 128×128, a compromise between accuracy and run time.

The general lesson: a few thousand labels are too few to *create* good features, but they can *adapt* features learned from millions of images; how much the adaptation should change those features depends on how different your images are from the pretraining images. **Pretrain on a huge dataset, then fine-tune on yours** is also exactly how large language models are built (Week 14).

9 HW6 Map: Problems 1, 2 and 6

HW6 part	**sections**	**the effect to see**
P1(A) corr2d, channels, pooling (unfold given)	§2–4	your outputs equal the hand-computed numbers (19/25/37/43, 56/72/104/120, the patch matrix) and `F.conv2d`
P1(B) LeNet yours vs `nn`; vs the MLP	§1, §5	the same function; under a third of the parameters; the 1998 version trails the MLP
P1(C) ReLU + max pooling; shifts	§4–5	level with the MLP; far more robust to 1–2 pixel shifts
P1(D) batch norm	§6	faster first epochs; fine at a 30× larger learning rate
P2(A) residual block, small ResNet	§7	far better than BN-LeNet on CIFAR-10; large train/test gap
P2(B) augmentation	§8	the gap shrinks; test accuracy up a little
P2(C) scratch / probe / fine-tune	§8	fine-tuning wins by a wide margin with 5,000 labels
P6 (theory) robustness	§4 key point, §8 remark	
P7 (optional) plain vs residual at depth	CNN_ADVANCED	

Common bugs.

- Flipping the kernel (`F.conv2d` does not flip; your `corr2d` must not either).
- Patch matrix in the wrong order: the rows of P must follow the order of the flattened kernel (c, a, b'), the same order as `W.reshape(c_o, -1)`.

- Pooling stride: PyTorch's pooling defaults to stride = window size; a from-scratch pool with stride 1 is a different layer.
- Batch norm statistics over the batch only, forgetting the positions (for a convolution, average over $n \cdot h \cdot w$); the biased vs unbiased variance; forgetting `model.eval()` at test time (the test accuracy then depends on the batch).
- One BN module reused twice; a skip connection whose shape doesn't match (missing 1×1 convolution, or a different stride on the skip).
- Augmenting the test set; flipping images vertically (an upside-down car is not a training example you want).

Key point: Debug small, run full. The HW6 notebook has a `MODE` switch. In `"dev"` mode the data are small subsets and the networks are narrower, so every cell runs in about a minute; in `"full"` mode you get the real experiments. The checkpoint cells are *just as trustworthy* in dev mode: they test that your code computes the right function on fixed inputs, which does not depend on how much data the model saw. Dev *accuracies*, on the other hand, mean nothing — 3 epochs on 2,000 images measure the tiny budget, not the architecture. So: write and debug in dev until every check passes, then run full for every number you report. (Optional Problem 11 asks you to explain dev mode's choices.)

10 Where This Leads

- **CNN_ADVANCED** (optional): how backprop runs through a convolution and a max-pool; why two 3×3 layers beat one 5×5; the family tree from AlexNet to ConvNeXt; batch, layer, group and instance norm side by side; plain vs residual networks at 20+ layers; transfer learning in more detail.
- **RNN note** (Week 12): weight sharing over *time* instead of space; the same product-of-factors gradient problem, now across time steps.
- **Weeks 13–14**: every Transformer block is a residual block with a *layer* norm (the other axis of Figure 4); a Vision Transformer cuts an image into patches — the columns of the unfold matrix of §3 — and treats them like words.