CNN Deep Dive & Classic Architectures

Lecture 7 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. State the **output-size formula** and compute it.
2. Explain **receptive field** and its growth with depth.
3. Apply the **VGG insight** · 3 stacked 3×3 ≈ one 7×7.
4. Describe **1×1 conv** as channel mixing / bottleneck.
5. Name the **three inductive biases** of conv and when they fail.
6. Place classic architectures · LeNet → AlexNet → VGG → GoogLeNet.

Recap · where we are

- **Training stack**: PyTorch recipe, debug ladder, error analysis (L3).
- **Optimizer**: AdamW + warmup + cosine (L4–L5).
- **Regularization**: weight decay, aug, Mixup, dropout, BN/LN/RMSNorm (L6).

> REFERENCE
>
> Today maps to **UDL Ch 10** · *Convolutional networks* (early sections).
>
> ES 654 covered LeNet and CNN basics. We skim those and spend time on **receptive fields, the classic architecture progression, and inductive biases**.

Four questions

1. What does convolution *actually compute*? (brisk — prereq knows)
2. How does receptive field grow with depth?
3. Why did architectures converge on stacked 3×3 convs?
4. What inductive biases does convolution bake in — and why do they help?

PART 1

Convolution mechanics

A brisk recap with new diagrams

Convolution · the feature-detector view

KEY IDEA

A convolution kernel is a **learned feature detector**.

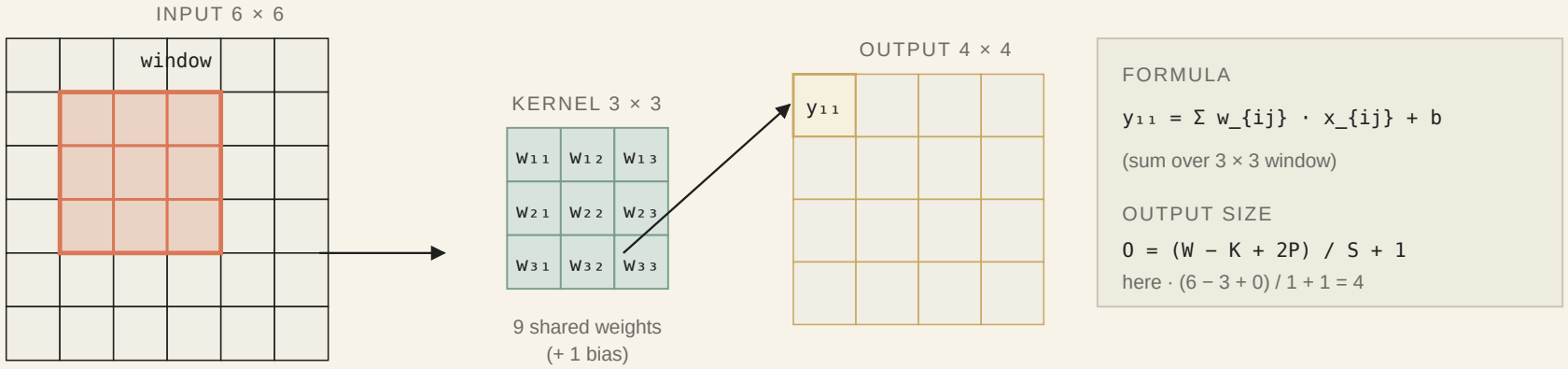
- Kernel A · responds to vertical edges
- Kernel B · responds to horizontal edges
- Kernel C · responds to red-green color transitions

Slide each detector across the entire image · the output map "lights up" wherever its specific feature appears.

In a CNN we don't hand-design these kernels · the network *learns* the most useful detectors during training. Early layers learn edges and textures; deeper layers compose those into parts and objects.

Convolution — sliding window, shared weights

Convolution — slide a kernel, take dot products, build a feature map



*Convolution slides the SAME 9-weight kernel across the whole image — **parameter sharing** + **sparse connectivity** + **translation equivariance**.*

IN PRACTICE

▶

Interactive: drag the kernel across an image, see the feature map fill in live — convolution-visualizer.

The lawnmower analogy

INTUITION

Your input is a rectangular **lawn**. The kernel is your **lawnmower**.

- **Input width** W — width of the lawn.
- **Kernel size** K — width of the mower blades.
- **Padding** P — paved sidewalk added so the mower can start with its centre on the lawn's edge.
- **Stride** S — step size after each pass. $S = 1$ careful, $S = 2$ fast (half coverage).
- **Output size** O — how many passes until the lawn is mown.

Building the output-size formula step-by-step

1D toy: $W = 10$ pixels, $K = 3$.

1. **No padding, stride 1.** Kernel centre fits at positions 1…8 → $O = W - K + 1 = 8$.
2. **Add padding** $P = 1$ (one zero on each side). Effective lawn width: $W + 2P = 12$.

$$O = (W + 2P) - K + 1 = 10$$

3. **Add stride** $S = 2$. Available distance for the kernel centre to roam: $W + 2P - K$. Number of S-sized steps: $\lfloor (W + 2P - K)/S \rfloor$. Plus the starting position:

$$\boxed{O = \left\lfloor \frac{W + 2P - K}{S} \right\rfloor + 1}$$

This is exactly the formula students see in textbooks — but now we know **where each term comes from.**

Output-size · worked numeric

VGG's first layer: $W = 224,\ K = 3,\ P = 1,\ S = 1.$

$$O = \left\lfloor \frac{224 - 3 + 2}{1} \right\rfloor + 1 = 223 + 1 = 224$$

Output is **same size** as input — this is **"same" padding** ($P = (K-1)/2$ and $S = 1$). Used in nearly every modern CNN block.

ImageNet stem: input `(3, 224, 224)`, apply `Conv2d(3, 64, kernel_size=7, stride=2, padding=3)`:

$$O = \lfloor (224 - 7 + 6)/2 \rfloor + 1 = 112$$

Output: `(64, 112, 112)`.

Why convolution is cheap. MLP doing the same thing: $224^2 \cdot 3 \cdot (64 \cdot 112^2) \approx 10^{11}$ ops. Conv: $7^2 \cdot 3 \cdot 64 \cdot 112^2 \approx 1.2 \cdot 10^8$ ops — **~1000× cheaper.**

INTUITION

Parameters are *shared* across positions, and each output depends on a small region. That's what makes the conv *orders of magnitude* cheaper than an equivalent MLP — not a marginal win.

The four hyperparameters, in one picture

For every conv layer, think in this order:

1. **kernel size** — the *field of view* (3 for modern nets, 7 only at the stem).
2. **stride** — how fast the filter steps (1 to preserve, 2 to halve).
3. **padding** — how much zero-boundary to add (so output size matches or shrinks predictably).
4. **dilation** — gap between kernel taps (used in segmentation to widen RF without more params).

KEY IDEA

99% of production CNNs only use (1, 2). Segmentation (L9) uses (4). Advanced audio / video sometimes uses (4). For images, reach for `kernel_size=3, padding=1, stride=1 or 2` — nearly every other choice is a specific research idea.

Max-pool · worked numeric example

DERIVATION

Input · 4 × 4 matrix

$$\begin{bmatrix} 1 & 2 & 8 & 3 \\ 4 & 6 & 5 & 1 \\ 9 & 7 & 2 & 3 \\ 5 & 3 & 1 & 0 \end{bmatrix}$$

2×2 max-pool, stride 2 · slide a 2×2 window non-overlapping.

WINDOW	VALUES	MAX
top-left	1, 2, 4, 6	**6**
top-right	8, 3, 5, 1	**8**
bottom-left	9, 7, 5, 3	**9**
bottom-right	2, 3, 1, 0	**3**

Output · $\begin{bmatrix} 6 & 8 \\ 9 & 3 \end{bmatrix}$

Halved spatial size · kept the strongest activation per region · gained translation invariance.

The "where's the cat?" detector

INTUITION

Two kinds of cat detector:

- **Equivariant robot** · beeps and points a laser at the cat. Move the cat → laser moves with it. Output *changes in step* with the input. **This is convolution.**
- **Invariant robot** · just yells "CAT!" if a cat is anywhere. Move the cat → still yells "CAT!". Output **doesn't change**. **This is the goal of a classifier — pooling helps us get there.**

Equivariance vs invariance · concretely

Convolution is translation equivariant. "Equivariant" = changes the same way.

- Shift input by 10 px right → feature map shifts by 10 px right. The same edge detector lights up at every position.

  Pooling adds local translation invariance. "Invariant" = doesn't change.

$$\begin{bmatrix} 1 & 2 \\ 4 & 6 \end{bmatrix} \xrightarrow{\max} 6 \qquad \begin{bmatrix} 6 & 2 \\ 4 & 1 \end{bmatrix} \xrightarrow{\max} 6$$

Move the strongest feature within the window → **same output**. Stack many such layers and the local invariance compounds into **global** invariance: the final layer cares *that* a cat is present, not *where*.

$$\text{MaxPool}(x) = \max_{i,j \in \text{window}} x_{ij}$$

In 2026, stride-2 convs often replace max-pooling entirely (less info loss).

Why pooling works · the invariance argument

Imagine a cat in the corner of an image vs centered. Should the network's answer depend on *where* the cat is?

- For classification · **no** (it's still a cat).
- For segmentation · **yes** (we need the pixels back).

> DERIVATION
>
> Pooling creates a small *"I don't care exactly where"* window. Stacks of pool+conv compound this: by layer 5, the network cares about the cat's presence, not its exact pixel position.

This is the **invariance gradient** a classification CNN builds — conv1 nearly equivariant (tells you where), final global pool fully invariant (tells you what).

PART 2

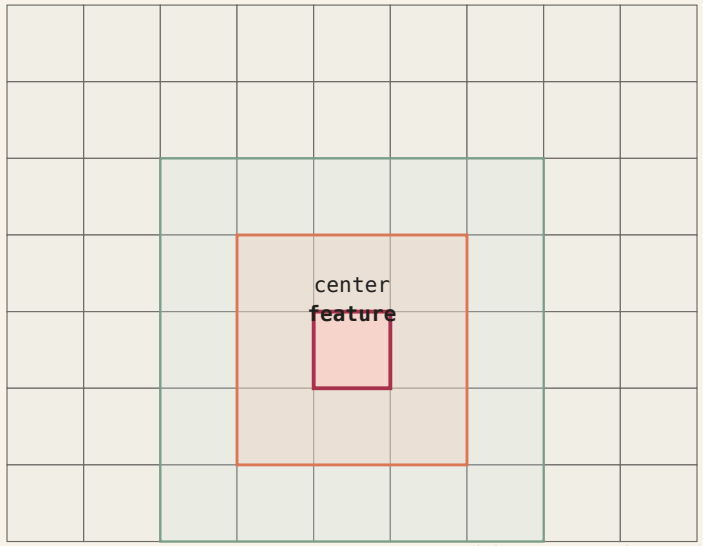
Receptive field

How deep features "see" far-away pixels

RF grows with depth

Receptive field — how much of the input a deeper feature "sees"

INPUT · STACKING 3×3 CONVS GROWS RF



RECEPTIVE FIELD GROWTH

after depth 3 (via 3× 3×3 conv)	**RF = 1**
after depth 2	**RF = 3**
after depth 1	**RF = 5**
input (no conv yet)	RF = 7

FORMULA

$$RF_\ell = RF_{\ell-1} + (K_\ell - 1) \cdot \Pi\, S_i$$

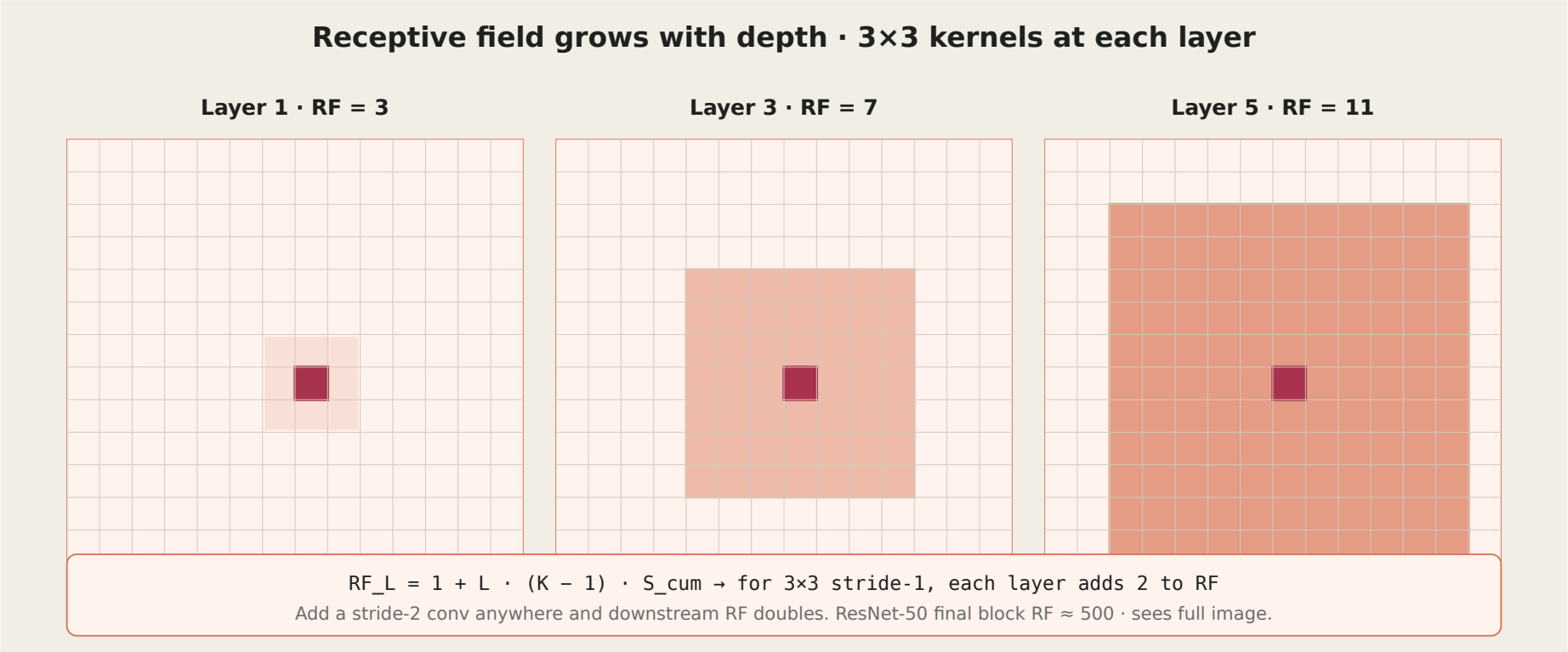
so three stacked 3×3 convs ≈ one 7×7 conv but with fewer parameters and more non-linearities (VGG idea).

Depth buys you reach — a single deep feature sees a large input region indirectly.

IN PRACTICE

▶ Interactive: drag depth/kernel/stride and watch RF grow live — receptive-field-grower.

Receptive field grows with depth · picture



One expert vs. three locals

INTUITION

You need to cross a complex city.

- **7×7 approach** · ask one expert for the entire 7-turn route. Complex instruction; many parameters; one shot to process.
- **Stacked 3×3 approach** · ask a local for the next 2 turns, walk there, ask another for the next 2, walk there, ask a third. Each instruction is **simpler** (fewer params per layer); you **re-evaluate** at each stop (a non-linearity!); you cover the same area.

The 3-layer stack has the same coverage as the 7×7 expert — but more efficient and more "thinking" between hops.

Let's prove it · RF and parameter math

Three stacked 3×3 convs vs. one 7×7 conv (stride=1, $C \to C$ channels).

Receptive field (RF) — trace how far back one output pixel "sees":

- After conv 1 (3×3): RF = **3**.
- After conv 2: RF grows by $K - 1 = 2$ → RF = **5**.
- After conv 3: RF = **7**. ✓ Same as a single 7×7.

Parameters (formula: $K \cdot K \cdot C_{\text{in}} \cdot C_{\text{out}}$):

- One 7×7 conv: $7 \cdot 7 \cdot C \cdot C = 49\,C^2$.
- Three 3×3 convs: $3 \cdot (3 \cdot 3 \cdot C \cdot C) = 27\,C^2$. **45% cheaper.**
- Plus we get **3 ReLUs** instead of 1 → richer function class.

	1× (7×7)	3× (3×3) STACKED
RF	7	7
Params	$49\,C^2$	$27\,C^2$

VGG (2014) built its whole architecture around this trade.

Worked numeric · counting parameters

Mid-network layer with $C = 256$ channels.

- **One 7×7 layer.** $49 \cdot 256^2 = 49 \cdot 65{,}536 = 3{,}211{,}264$ params.
- **Three stacked 3×3 layers.** $27 \cdot 256^2 = 27 \cdot 65{,}536 = 1{,}769{,}472$ params.

  Saving: **~1.5 million parameters per block**, plus 2 extra non-linearities. Repeated across many blocks → enormous total saving. This is the core design principle of VGG.

Receptive field · the chain-reaction view

INTUITION

Analogy · dominoes. Push the last domino — how many dominoes were involved? Each conv layer is like one push. A 3×3 kernel "pushes" a 3×3 region; the next layer's 3×3 pushes a region already pushed by the first. The effect propagates.

For a stack with stride 1, the RF formula is:

$$\mathrm{RF}_{\text{new}} = \mathrm{RF}_{\text{prev}} + (K - 1)$$

5-layer CNN, all 3×3, $K - 1 = 2$:

LAYER	CALCULATION	RF
Input	—	1
Conv1	$1 + 2$	**3**
Conv2	$3 + 2$	**5**

Stride. A layer with stride S multiplies the $(K - 1)$ term by all *previous* strides. Stride is a multiplier — that's why ResNet-50 reaches RF ≈ 500.

What can a 11×11 patch "see"?

The RF size tells us the **scale of features** a layer can detect.

RF	WHAT FITS IN THIS WINDOW
3×3	a single corner; a fragment of an edge
5×5	a longer edge or a junction
11×11	on a 28×28 MNIST digit, the **loop of a "6"** or the cross of a "4"
49×49	on a 224×224 ImageNet photo, an entire **eye**, **nose**, or small object
~500	the **whole image** (ResNet-50 final block)

Each layer learns to use this growing context — edges → textures → parts → objects. Deeper nets = larger RF = richer hierarchies.

Effective receptive field

INTUITION

Theoretical RF grows linearly with depth.
Effective RF (Luo et al. 2016) is more **Gaussian-shaped** — pixels near the centre contribute much more than edge pixels.

Two consequences:

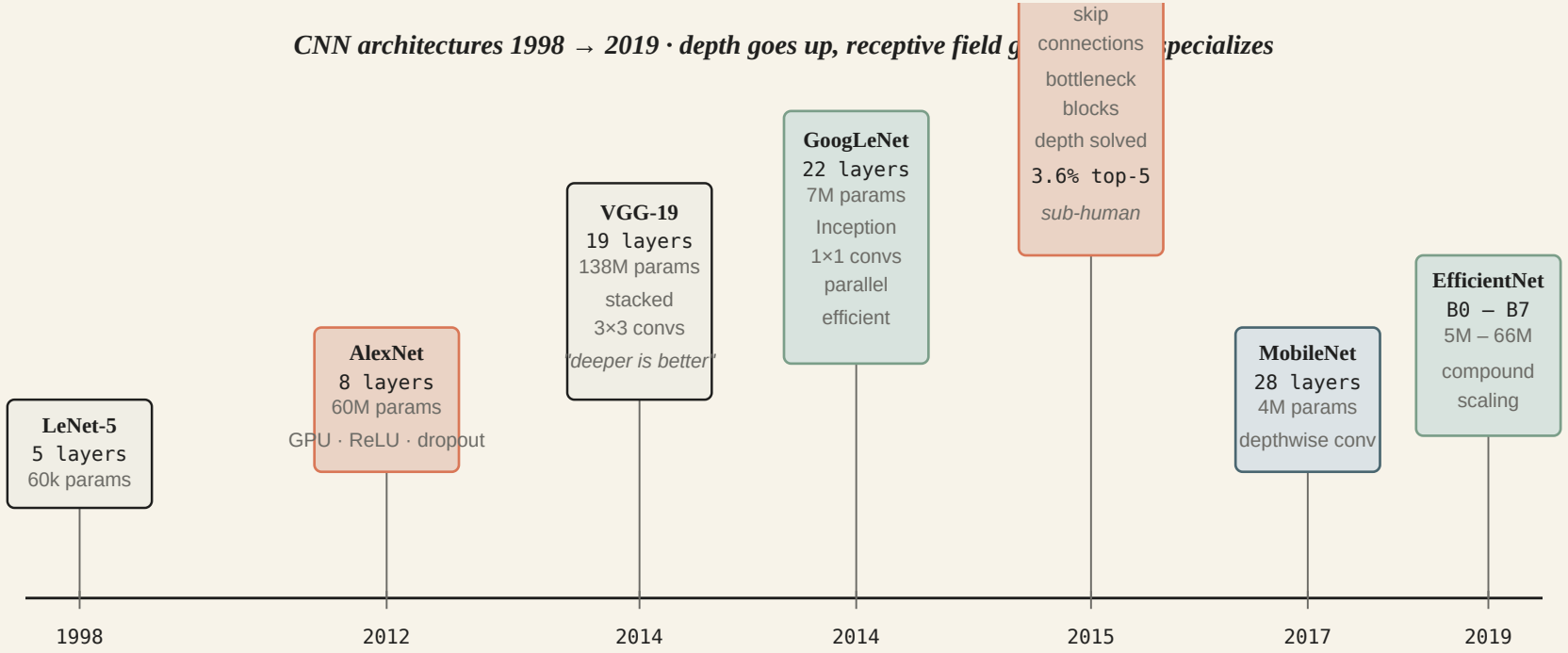
1. Modern architectures often need **dilated convs** or **attention** to get truly global RF.
2. CNNs and Vision Transformers differ here — ViT has uniform global RF from layer 1 (coming in L18).

PART 3

Classic architecture evolution

LeNet → AlexNet → VGG → Inception → ResNet → MobileNet → EfficientNet

The progression



*Depth increased **20×**; params roughly constant after 2014; efficiency became the new axis after ResNet solved depth.*

What each era got right

YEAR	MODEL	KEY CONTRIBUTION
1998	**LeNet-5**	first successful CNN (digits, MNIST)
2012	**AlexNet**	GPU training, ReLU, dropout, big data (ImageNet)
2014	**VGG**	*"depth with small kernels"* — 3×3 only, stacked
2014	**GoogLeNet**	1×1 bottlenecks, parallel branches (Inception)
2015	**ResNet**	skip connections → 152 layers trainable
2017	**MobileNet**	depthwise separable convs (edge devices)
2019	**EfficientNet**	compound scaling (depth × width × resolution)

1×1 conv · the recipe-mixer

KEY IDEA

A 1×1 convolution sounds useless · it only sees one pixel! But the power is in the **depth dimension**.

At each pixel · you have 256 channel values (your "ingredients"). The 1×1 conv learns the best **recipes** to mix them down into 64 new "flavors" (output channels).

It's an extremely cheap way to · reduce channels (bottleneck) · expand them after a 3×3 · or remix a feature map without spatial mixing.

Used everywhere in modern CNNs and Transformers (the "output projection" of attention is a 1×1 conv applied to the channel axis).

1×1 conv · the math, with a worked example

A 1×1 conv = a small linear (FC) layer applied **at every pixel independently**.

At pixel (i, j):

- Input vector: $x_{i,j} = [x_1, x_2, \ldots, x_{C_{\text{in}}}]$.
- Weight matrix W of shape $(C_{\text{out}}, C_{\text{in}})$.
- Output: $y_{i,j} = W\, x_{i,j}$ — matrix–vector product, repeated $H \cdot W$ times.

  Worked numeric · 3 channels (RGB) → 2 channels. Pixel value $x = [255, 100, 50]$.

- Recipe 1 (grayscale): $w_1 = [0.30, 0.59, 0.11]$
  $y_1 = 0.30 \cdot 255 + 0.59 \cdot 100 + 0.11 \cdot 50 = 76.5 + 59 + 5.5 = \mathbf{141}$
- Recipe 2 (red – green): $w_2 = [0.5, -0.5, 0.0]$
  $y_2 = 0.5 \cdot 255 - 0.5 \cdot 100 + 0 = \mathbf{77.5}$

  Output at this pixel: $[141, 77.5]$. The network **learns** the best recipes during training.

Uses everywhere in modern networks · reduce channels before 3×3 (GoogLeNet bottleneck) · expand after (ResNet $1\times1 \to 3\times3 \to 1\times1$) · "attention output projection" in Transformers.

1×1 conv · worked example

Input tensor `(256, 14, 14)` — 256 channels at 14×14 spatial resolution.
Apply `Conv2d(256, 64, kernel_size=1)` → `(64, 14, 14)`.

DERIVATION

- Parameters: $256 \cdot 64 = 16{,}384$ (plus bias).
- FLOPs: $256 \cdot 64 \cdot 14 \cdot 14 \approx 3.2\text{M}$ per forward pass.

What did we just do? Took **256 channels at each spatial position, mixed them linearly to 64 channels.** Spatial structure preserved. Channel structure compressed.

A 3×3 conv immediately after now runs 4× cheaper because the depth is 4× smaller. That's the **bottleneck trick**: sandwich 3×3 convs between 1×1 compressions.

AlexNet → VGG · the "just add depth" years

Between 2012 and 2014, the field converged on a recipe:

1. Keep convolution kernels **small** (3×3) and **many**.
2. **Stack deeper** until you run out of memory or accuracy plateaus.
3. Use **ReLU + dropout + batch-norm** to make deeper nets trainable.

> INTUITION
>
> By 2015, people tried to go deeper than 25 layers and networks **stopped learning**. Adding layers made *training* loss worse — not a generalization issue. This pointed at the *optimization* problem that ResNet (next lecture) solved with skip connections.

PART 4

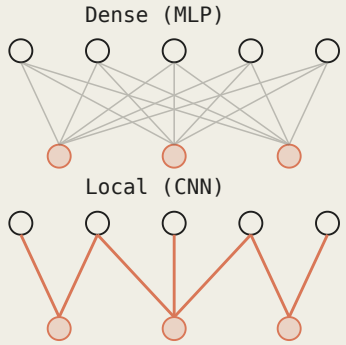
Inductive biases

Why convolution beats MLP on images

Three biases baked into convolution

CNN inductive biases — the three assumptions baked into convolution

1 · SPARSE CONNECTIVITY



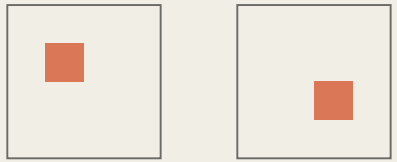
2 · WEIGHT SHARING

`same 9 weights`

reused across the image

`a feature detector works everywhere`

3 · TRANSLATION EQUIVARIANCE



`output feature map`
`shifts by the same amount`

(not the same as invariance — that's pooling)

These three assumptions reduce an MLP's 150k weights per neuron to a **CNN's 9 weights**. Fewer params · better inductive bias · works from tiny datasets.

In words · what each bias does

1. Locality

Each output sees only a small window. Forces the network to first extract local features (edges, textures) before combining them.

Why correct for images · nearby pixels are semantically related (same edge, same object).

2. Translation equivariance

Shift input → shift output. The same feature detector runs at every position.

Why correct for images · a cat is a cat whether it's in the corner or centre.

3. Hierarchy of scales

Stacking convs builds larger receptive fields. Deep networks *compose* features at each scale.

Why correct for images · visual world is hierarchical (edge → texture → part → object).

Assumptions are a shortcut

INTUITION

Analogy · searching for keys.

- **MLP** · no idea where they could be. Check **every square inch** of the house — ceiling, inside the toaster, under the rug. Forever.
- **CNN** · apply assumptions:
  i. **Locality** · "they're near other things." → check surfaces.
  ii. **Translation invariance** · "the concept *keys-on-a-table* is the same in every room." → use the same search pattern.

The right assumptions shrink the search space *enormously*.

Parameter math · MLP vs CNN

Scenario. One hidden layer. Input: 224×224 RGB image → $224 \cdot 224 \cdot 3 = 150{,}528$ features.

MLP (fully connected). Hidden layer = 4096 neurons. Every input connects to every neuron.

$$\text{params} = 150{,}528 \cdot 4{,}096 \approx \mathbf{616}\,\text{million}$$

CNN. 3×3 kernel, $C_{\text{in}} = 3$, $C_{\text{out}} = 64$.

$$\text{params} = (K \cdot K \cdot C_{\text{in}}) \cdot C_{\text{out}} = (3 \cdot 3 \cdot 3) \cdot 64 = 27 \cdot 64 = \mathbf{1{,}728}$$

Difference: 616,000,000 vs. 1,728 — a factor of ~350,000.

This is what locality (small kernel) and weight sharing (sliding the same kernel) buy. The CNN is *forced* to learn reusable patterns; the MLP must learn every connection from scratch.

> INTUITION
>
> Vision Transformers (L18) give up most of this inductive bias — they need pretraining on far more data to compensate.

Inductive bias · the data-efficiency plot

Small data ($\leq 10^4$ images)

- **CNN wins** — the prior does the heavy lifting.
- MLP barely learns anything; ViT is worse than CNN.

Huge data ($\geq 10^8$ images)

- **ViT matches or beats CNN** — enough data to overcome the missing prior.
- The "ImageNet-21k threshold" from Dosovitskiy 2020.

KEY IDEA

The bias is a free data multiplier. A CNN at 50k images behaves like a ViT at 500k. If your dataset is small, use a CNN (or start from a pretrained CNN).

A CNN block in PyTorch

```python
class ConvBlock(nn.Module):
    def __init__(self, c_in, c_out):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1),
            nn.BatchNorm2d(c_out),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.net(x)
```

Every classic CNN is a stack of these (plus pooling or stride-2 for downsampling). VGG is literally this block repeated.

Feature visualization · what each layer learns

LAYER	TYPICAL FEATURES (FOR A TRAINED CNN ON NATURAL IMAGES)
Conv1	oriented edges, colour blobs (~like Gabor filters)
Conv2	junctions, simple textures
Conv3	repeated patterns (fur, grid, stripes)
Conv4	object parts (eyes, wheels)
Conv5	whole objects / object arrangements

REFERENCE

Zeiler & Fergus 2014 · *Visualizing and Understanding Convolutional Networks* — canonical reference for layer visualization.

What's next

INTUITION

L7 covered the "classic CNN" era — the 1998–2014 progression that ended with VGG.

L8 (next lecture) picks up where we paused: Inception modules, ResNet in CNNs, MobileNet's depthwise separable convs, EfficientNet scaling, and transfer learning — the one practical skill you'll use most often.

Common questions · FAQ

Q. What's a typical kernel size in 2026?
A. 3 everywhere, except stem layer uses 7×7 (more RF for first layer). Very occasionally 5×5 for specific blocks.

Q. When do I need big-kernel convs (ConvNeXt 7×7)?
A. When replicating Transformer-style long-range mixing in CNNs. ConvNeXt showed 7×7 depthwise conv can match attention on some tasks.

Q. How do I choose number of channels?
A. Double channels every time you halve spatial (32 → 64 → 128 → 256 → 512). Keeps params-per-layer roughly constant. VGG, ResNet, EfficientNet all follow this.

Q. Padding · 'same' vs 'valid'?
A. `padding='same'` (PyTorch) keeps output size = input size. Default for most blocks. `'valid'` (no padding) shrinks · used when downsampling is the point.

Summary · Lecture 7 — summary

- **Convolution** = sliding window with shared weights · translation-equivariant · 3 biases (sparse, shared, local).
- **Output size** · $O = (W - K + 2P)/S + 1$; pad to preserve, stride to downsample.
- **Receptive field** grows with depth — 3 stacked 3×3 convs ≈ one 7×7 with fewer params.
- **Stacked 3×3 (VGG)** beat single 7×7 — fewer params, more non-linearities.
- **1×1 convs** mix channels — they're everywhere.
- **LeNet → AlexNet → VGG** is the classic era; ResNet (next) finally solved depth.

Read before Lecture 8

Prince — Ch 10 (advanced), Ch 11 (residual in CNNs).

Next lecture

Modern CNNs + Transfer Learning — GoogLeNet bottlenecks, ResNet-in-CNN, MobileNet, EfficientNet, fine-tuning pretrained backbones.

NOTEBOOK

Notebook 7 · `07-cnn-from-scratch.ipynb` — build a VGG-style mini-CNN for CIFAR-10; print tensor shapes at each layer; compute receptive field per layer.