Convolutional Neural Network Architectures

Applied Deep Learning — Chapter 6

Yin Yang

Foundation Books Project

Roadmap

Why Architectures Matter

Vocabulary for Comparing CNNs

LeNet, AlexNet, VGG

Inception, ResNet, DenseNet

Mobile and Efficient Designs

Comparison and Practice

Summary

Why Architectures Matter

Reading a CNN Architecture

An architecture is the directed arrangement of layers that maps an image tensor to logits.

Layer one operation: convolution, pooling, normalization, activation.

Block reusable local pattern of layers.

Stage group of blocks at one spatial resolution.

Stem first layers; turn pixels into early feature maps.

Head final layers; turn features into logits.

Read paths, not names. Ask how each design changes shape, compute, memory, and validation evidence.

The Running Example: CIFAR-10

- Input: $32 \times 32 \times 3$ RGB image, ten classes.
- Output: ten logits, one per class.
- A Keras mini-batch has shape `(128, 32, 32, 3)`: examples, height, width, channels.

Many CNN experiments fail before the optimizer has a fair chance:

- the stem downsamples a 32×32 image too aggressively,
- the head keeps the wrong class count,
- channel order silently changes.

First reading task: trace shapes before judging names.

Vocabulary for Comparing CNNs

Output Size, Parameters, MACs

Output size of a convolution with stride s and padding p:

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

Weights of a $k_h \times k_w$ conv from C_{in} to C_{out}:

$$k_h k_w C_{\text{in}} C_{\text{out}}.$$

Multiply-adds for one output:

$$H_{\text{out}} W_{\text{out}} C_{\text{out}} (k_h k_w C_{\text{in}}).$$

Channels multiply cost. Doubling C_{in} and C_{out} roughly quadruples weights at the same kernel size.

Effective Stride and Receptive Field

Recurrence with kernel k_ℓ and stride s_ℓ:

$$j_\ell = j_{\ell-1} s_\ell, \qquad r_\ell = r_{\ell-1} + (k_\ell - 1) j_{\ell-1}, \qquad j_0 = r_0 = 1.$$

- Two stride-2 layers $\Rightarrow$ effective stride $j = 4$: neighboring later cells correspond to positions four input pixels apart.
- Each kernel grows the receptive field by the spacing of the previous feature map.

Activation memory is different from parameter memory: a batch of 128 maps of shape $32 \times 32 \times 64$ uses ≈ 32 MiB at `float32`.

A Tiny CIFAR-10 Classifier

Stage	Operation	Shape	Weights
Input	CIFAR-10 image	$32\times32\times3$	0
Stem conv	3×3, s=1, p=1, 16 ch	$32\times32\times16$	432
Pool	2×2 max pool, s=2	$16\times16\times16$	0
Conv 2	3×3, s=1, p=1, 32 ch	$16\times16\times32$	4,608
Global avg pool	average each map	32	0
Head	dense to 10 logits	10	320
Total			5,360

Global average pooling keeps the head small: flattening a $16\times16\times32$ tensor into a 10-logit dense head would need 81,920 weights instead of 320.

LeNet, AlexNet, VGG

LeNet: First Practical CNN Skeleton

- Convolution–pooling–classifier pattern for digit and document recognition.
- First-conv parameter count depends on input channels:
  - grayscale: $5 \cdot 5 \cdot 1 \cdot 6 = 150$ weights.
  - CIFAR-10 RGB: $5 \cdot 5 \cdot 3 \cdot 6 = 450$ weights.
- No-pad 5×5 conv: $32 \to 28$, then 2×2 pool: $28 \to 14$.

Tiny baselines diagnose. Use small models to separate pipeline bugs from architecture limits — not to win CIFAR-10.

AlexNet: Scale, GPUs, and Aggressive Stride

- ImageNet breakthrough: GPU training, ReLU, dropout, large dense head.
- First conv: input 227, $k = 11$, $s = 4$, no pad:

$$\lfloor (227 - 11)/4 \rfloor + 1 = 55.$$

- Irregular by modern standards: kernels of mixed sizes, overlapping max pool, two-GPU split, local response normalization.

Stride trades detail. An 11×11 stride-4 first layer is wrong for 32×32 CIFAR-10; AlexNet's choices belong to its hardware era.

VGG: Depth Through Repeated 3×3

Two stride-1 3×3 convs cover a 5×5 receptive field with fewer weights than one 5×5:

$$2 \cdot 3 \cdot 3 \cdot C^2 = 18C^2 \quad \text{vs.} \quad 25C^2.$$

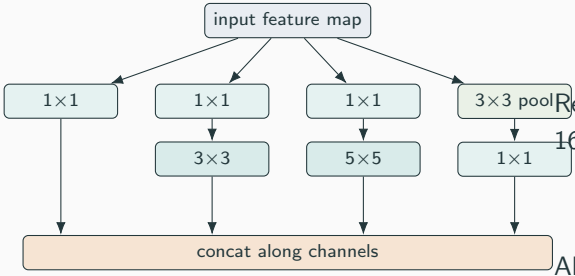
And there is a nonlinearity between them.

- Repeat 3×3 convs, downsample with max pool, increase channels.
- VGG19 = VGG16 + 3 more conv layers; same idea, more cost.
- Original ImageNet head used very large fully connected layers.

Readable but heavy: regular 3×3 stacks ease shape tracing, but high channel widths make the model parameter- and memory-heavy.

Inception, ResNet, DenseNet

Inception: Branches with Bottlenecks



Bottleneck saving — a 5×5, $192\to32$ conv:

$$5 \cdot 5 \cdot 192 \cdot 32 = 153{,}600.$$

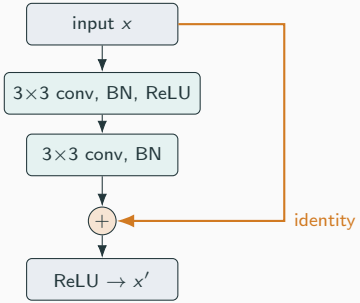
Replace with 1×1, $192\to16$ then 5×5, $16\to32$:

$$192 \cdot 16 + 5 \cdot 5 \cdot 16 \cdot 32 = 15{,}872.$$

About one tenth.

Branch shapes must agree along H, W before channel concatenation.

ResNet: Residual Corrections



A residual block computes $x' = x + F(x;\omega)$:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial x'}\left(I + \frac{\partial F}{\partial x}\right).$$

- Block learns a *correction* to the shortcut.
- Shortcut gives gradients a direct path back.
- If shapes differ, project with 1×1 stride-2.
- Basic block (R18/34): two 3×3.
- Bottleneck (R50+): $1\times1 \to 3\times3 \to 1\times1$.

Addition needs same shape. Identity shortcut only when H, W, and channels match.

ResNet50 Stages on CIFAR (CIFAR-style stem)

Part	Bottleneck branch	Output	Shortcut
Stem	3×3, s=1, 64 ch	$32\times32\times64$	—
conv2	$3\times$ $64\to64\to256$	$32\times32\times256$	first projects
conv3	$4\times$ $128\to128\to512$	$16\times16\times512$	first s=2 proj.
conv4	$6\times$ $256\to256\to1024$	$8\times8\times1024$	first s=2 proj.
conv5	$3\times$ $512\to512\to2048$	$4\times4\times2048$	first s=2 proj.
Head	global avg pool + 10-class dense	10	—

Depth count: $1+16\cdot3+1=50$. ImageNet stem (7×7, stride-2 + max pool) is wrong for 32×32 inputs. Use a CIFAR-style stem.

DenseNet: Reuse by Concatenation

Layer t in a dense block:

$$x_t = T_t([x_0, x_1, \ldots, x_{t-1}]).$$

Brackets mean channel concatenation.

- Growth rate k is the number of new channels each layer adds.
- Block starts at 64 ch with $k = 32$: after 4 layers, $64 + 4 \cdot 32 = 192$ ch.
- Transition layers: 1×1 conv (compress) + pool (downsample).

Growth rate adds width. Activation memory grows during training, even when parameters look modest.

Mobile and Efficient Designs

MobileNetV2: Depthwise Separable + Inverted Bottleneck

Standard 3×3 conv $32\to64$:

$$3 \cdot 3 \cdot 32 \cdot 64 = 18{,}432 \text{ weights.}$$

Depthwise 3×3 over 32 ch + pointwise 1×1, $32\to64$:

$$9 \cdot 32 + 32 \cdot 64 = 2{,}336 \text{ weights} \quad (\approx 7.9\times \text{ fewer}).$$

Inverted bottleneck: expand channels 1×1, depthwise 3×3, linear 1×1 projection back.

Shortcut only when stride is 1 and shapes match. Linear bottleneck: *no* ReLU after the narrow projection (avoids destroying information in low-dim space).

EfficientNet: Compound Scaling

Once an efficient block (MBConv with squeeze-and-excitation) exists, scale depth, width, resolution *together*:

$$m_d = c_d^q, \quad m_w = c_w^q, \quad m_r = c_r^q, \qquad c_d c_w^2 c_r^2 \approx 2.$$

- B0 baseline found by neural architecture search.
- B1 – B7 are scaled variants of B0.
- Squared exponents reflect FLOP cost: width and resolution scale compute quadratically; depth scales linearly.

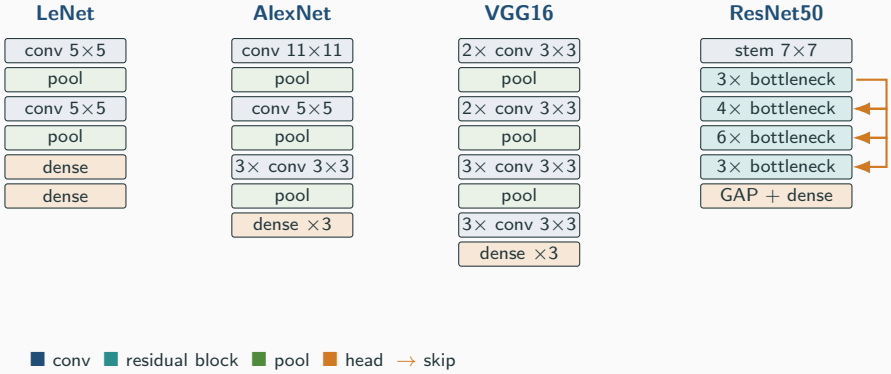
Don't upscale CIFAR-10 to 224 pixels just to use B0. Interpolation adds compute without adding visual evidence.

Comparison and Practice

Architecture Comparison Table

Family	Strength	Caution
LeNet	fast, understandable	low capacity
AlexNet	historical breakthrough	rarely a modern default
VGG16/19	easy to read & debug	parameter-heavy
GoogLeNet	parameter-efficient	complicated bookkeeping
ResNet	strong default	needs training recipe
DenseNet	feature reuse	activation-memory heavy
MobileNetV2	deployment-friendly	hardware-dependent gains
EfficientNet	balanced scaling	many moving parts

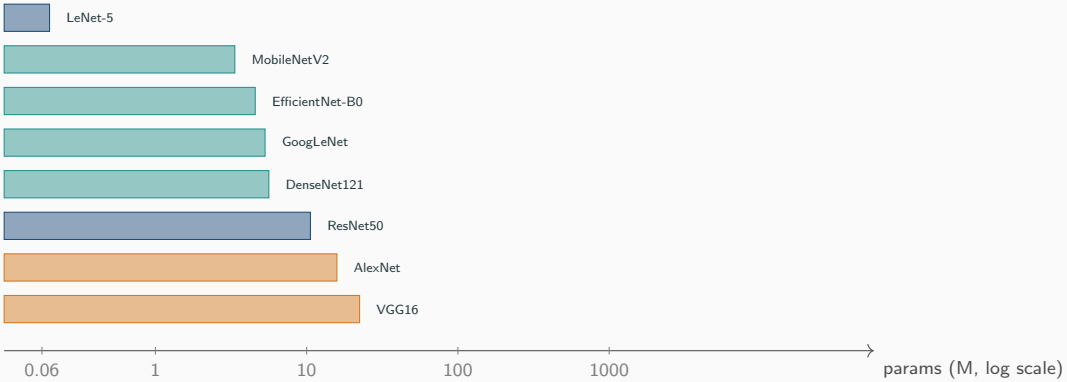
Stacked Schematic: Four Architectures Side by Side



Representative Scale (ImageNet inputs)

Variant	Params	MACs	Reading
LeNet-5	0.06M	0.0004B	debugging baseline
AlexNet	61M	0.72B	ImageNet/GPU era
VGG16	138M	15.5B	regular but heavy
VGG19	144M	19.6B	3 extra convs over VGG16
GoogLeNet v1	6.8M	1.5B	branches + bottlenecks
ResNet50	25.6M	4.1B	strong default
DenseNet121	8.0M	2.9B	efficient reuse
MobileNetV2	3.4M	0.30B	deployment baseline
EfficientNet-B0	5.3M	0.39B	search + compound scaling

Parameters at a Glance



Bigger is not always better. GoogLeNet beats AlexNet at a fraction of the parameters; ResNet50 holds its own against VGG16 with fewer weights.

Worked Example: ResNet on CIFAR-10

- Variant: ResNet50, bottleneck blocks.
- Stem: 3×3 stride-1, no initial max pool.
- Head: global average pool + 10-class dense.
- Parameters: 23,520,842 (CIFAR head).
- Bounded run: 512 train / 256 val / 256 test images, write training-curve, confusion-matrix, and metadata artifacts.

Architecture record names variant, block type, stem, downsampling pattern, classifier head, and parameter count — so the result can be reconstructed.

Troubleshooting Before Blaming the Family

Symptom	Architecture record to inspect
Accuracy near chance	10 logits? loss expects logits? class order?
Validation lags badly	ImageNet stem on 32×32? try CIFAR-style stem
Bigger variant slower, no gain	compare ResNet18/34 under same recipe
Train improves, val does not	augmentation, normalization, schedule, weight decay

Change one thing at a time; pipeline first, then architecture.

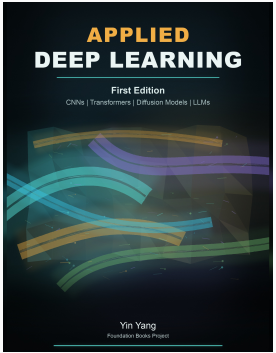
Summary

Summary

- LeNet $\rightarrow$ AlexNet: convolution–pooling–classifier scaled to ImageNet with GPUs and aggressive stride.
- VGG: depth through repeated 3×3; readable but heavy.
- Inception: parallel branches and 1×1 bottlenecks.
- ResNet: residual corrections with shortcut paths — trainable depth.
- DenseNet: feature reuse via concatenation; activation-memory cost.
- MobileNetV2: depthwise separable + inverted residual + linear bottleneck.
- EfficientNet: compound scaling of an efficient block family.

Names without tensor shapes are not arguments. Report variant, stem, blocks, stages, head, and parameter count alongside accuracy and runtime.

Next: when to fine-tune a pretrained body instead of training from scratch.



Applied Deep Learning

Chapter overview slides for the book by Yin Yang.

Book page	`foundation-books.github.io/applied-deep-learning`
Code	`github.com/foundation-books/applied-deep-learning-code`
Print	`amazon.com/dp/B0GZF7QRSH`
Kindle	`amazon.com/dp/B0GZF9221X`
License	CC BY-NC-ND 4.0

These free overview slides may be shared non-commercially with attribution and without modifications. Full explanations, exercises, and companion-code context are in the book and linked repository.