Modern CNNs & Transfer Learning

Lecture 8 · 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. Explain the **Inception** multi-branch idea and 1×1 bottlenecks.
2. Describe **ResNet basic / bottleneck** blocks with skip connections.
3. Apply **depthwise separable convolutions** (MobileNet) for efficiency.
4. Use **EfficientNet's compound scaling** rule.
5. Execute the **three transfer-learning recipes** correctly.
6. Pick a backbone from `timm` for a given dataset size.

Recap · where we are

- **Classic CNN era** (L7): LeNet, AlexNet, VGG — stacked 3×3 convs work.
- **Receptive field** grows with depth.
- **Inductive biases** — sparse connectivity, weight sharing, equivariance.

> REFERENCE
>
> Today maps to **UDL Ch 10** (advanced sections) and **Ch 11** (residual / skip connections in CNNs).

Two halves today:

1. The architectures that came after VGG — Inception, ResNet, MobileNet, EfficientNet.
2. Transfer learning — the single most practical CNN skill.

PART 1

Inception · parallel kernels

Let SGD pick the right receptive field

Inception · the buffet idea

KEY IDEA

VGG taught us · stacking 3×3 convs is powerful. But every layer has the **same fixed view** of the world.

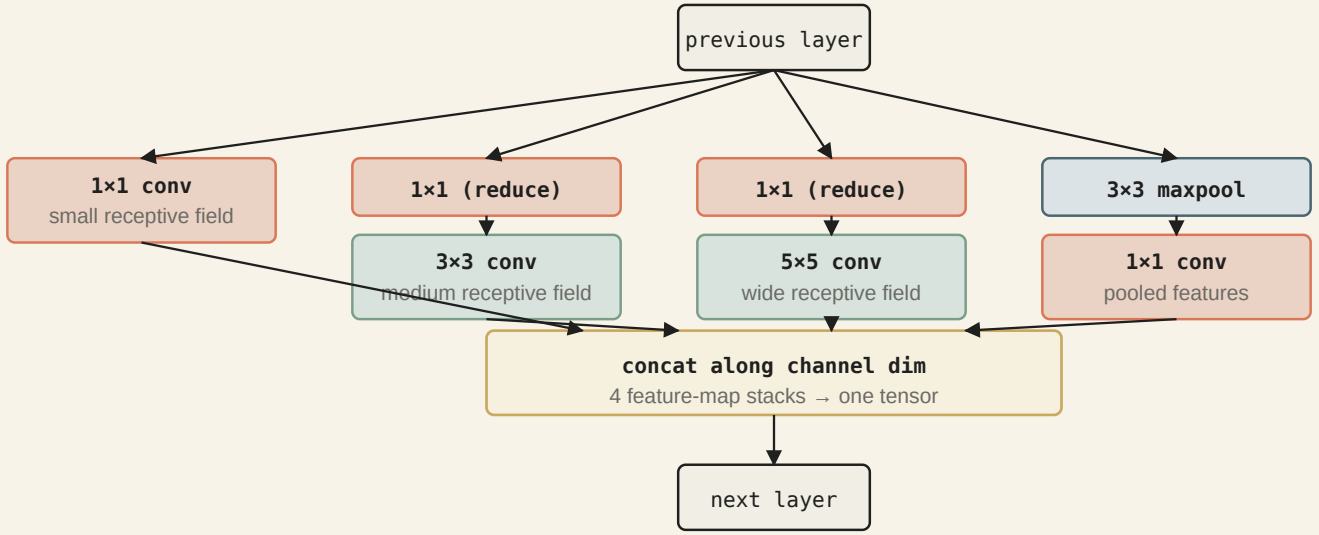
What if some features are tiny (need 1×1) and others are large (need 5×5)?

Inception's answer · **don't pick** · offer a buffet of kernel sizes (1×1, 3×3, 5×5, pool) at every layer · let SGD pick the right ratio.

The buffet is concatenated along the channel axis · the next layer sees all kernel sizes' outputs simultaneously.

The Inception module (Szegedy 2014)

Inception module — try multiple kernel sizes in parallel, let SGD pick



***1×1 convolutions** reduce channel count before the expensive 3×3 and 5×5 — the parameter-efficiency trick that made 22-layer GoogLeNet feasible.*

Why 1×1 convolutions matter · derivation

Conv-layer cost formula: $(K_h \cdot K_w \cdot C_{\text{in}}) \cdot C_{\text{out}}$.

Direct 3×3, 256 → 256.
$(3 \cdot 3 \cdot 256) \cdot 256 = 2304 \cdot 256 = \mathbf{589{,}824}$ params.

1×1 bottleneck. Three steps:

1. **Squeeze** (1×1, $256 \to 64$): $(1 \cdot 1 \cdot 256) \cdot 64 = 16{,}384$.
2. **Spatial mix** (3×3, $64 \to 64$): $(3 \cdot 3 \cdot 64) \cdot 64 = 36{,}864$.
3. **Expand** (1×1, $64 \to 256$): $(1 \cdot 1 \cdot 64) \cdot 256 = 16{,}384$.

Total: $16{,}384 + 36{,}864 + 16{,}384 = \mathbf{69{,}632}$ params.

Ratio: $589{,}824 / 69{,}632 \approx 8.47$ → **~8.5× cheaper.** Same input/output shape, three quarters less computation.

Worked numeric · 1×1 bottleneck on small numbers

Input `(14, 14, 16)` → Output `(14, 14, 32)`, kernel 3×3.

Without bottleneck. $(3 \cdot 3 \cdot 16) \cdot 32 = 144 \cdot 32 = \mathbf{4{,}608}$ params.

With bottleneck (squeeze to 4 channels).

1. Squeeze (1×1, $16 \to 4$): $(1 \cdot 1 \cdot 16) \cdot 4 = 64$.
2. 3×3 ($4 \to 4$): $(3 \cdot 3 \cdot 4) \cdot 4 = 144$.
3. Expand (1×1, $4 \to 32$): $(1 \cdot 1 \cdot 4) \cdot 32 = 128$.

   Total: $64 + 144 + 128 = \mathbf{336}$ params. **~13.7× cheaper** for the same external shape.

1×1 convs appear everywhere modern · GoogLeNet reduction · ResNet bottleneck · Transformer FFN projections.

PART 2

ResNet in CNNs

Skip connections · bottleneck blocks

Vanishing gradients · the telephone game

INTUITION

Analogy. A game of telephone over 100 people. By the end, the message is gibberish. Gradients are the **correction message** sent backwards: *"Hey person 1, you said 'purple monkey' — should've been 'purple donkey'."*

By the time that correction reaches person 1, it's diluted to a meaningless whisper. Person 1 can't learn. **Vanishing gradient.**

A skip connection is a **gradient superhighway** — a direct, uninterrupted path from the end back to the beginning.

Skip connections · derive the gradient

Plain layer. $h_{\text{out}} = F(h_{\text{in}})$. Chain rule:

$$\frac{\partial \mathcal{L}}{\partial h_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial h_{\text{out}}} \cdot F'(h_{\text{in}})$$

With 100 layers: 100 of these F' multiplied → if each is < 1, the product **collapses to 0**. Vanishing.

Residual layer. $h_{\text{out}} = h_{\text{in}} + F(h_{\text{in}})$.

$$\frac{\partial h_{\text{out}}}{\partial h_{\text{in}}} = \underbrace{\frac{\partial h_{\text{in}}}{\partial h_{\text{in}}}}_{=1} + F'(h_{\text{in}}) = 1 + F'(h_{\text{in}})$$

So:

$$\frac{\partial \mathcal{L}}{\partial h_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial h_{\text{out}}} \cdot \left(1 + F'(h_{\text{in}})\right)$$

The **+1** is the highway. Even when F' is near zero, the gradient still flows through.

Worked numeric · gradient flow

Upstream gradient $\partial \mathcal{L}/\partial h_{\text{out}} = 0.5$. Tiny weights → $F'(h) = 0.01$.

	PLAIN	RESIDUAL
One layer	$0.5 \cdot 0.01 = 0.005$	$0.5 \cdot (1 + 0.01) = 0.505$
10 layers	$0.5 \cdot (0.01)^{10} = 5 \times 10^{-21}$	$0.5 \cdot (1.01)^{10} \approx 0.55$

Plain → completely vanished after 10 layers. Residual → still strong.

This is why ResNet trains 152 layers easily, while plain 34 layers couldn't even fit the training data. Identity path stops vanishing **at construction time**, not through training luck. Same vector form:

$$\frac{\partial \mathcal{L}}{\partial h_l} = \frac{\partial \mathcal{L}}{\partial h_{l+1}} \cdot \left(I + \frac{\partial F}{\partial h_l} \right)$$

Why skip connections also help forward pass

When a new block isn't useful yet, residual = 0 is the easiest thing to learn — the identity mapping just passes h_l through.

INTUITION

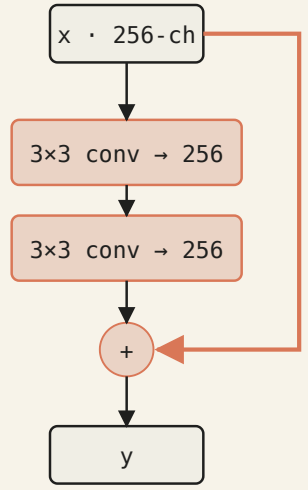
Adding blocks can only improve or at worst no-op, never hurt. Before ResNet, adding layers to a working network often made it worse (degradation problem). After ResNet, deeper ≥ shallower — you just keep adding.

Same idea shows up as LSTM's cell state (L10) and Transformer's residual stream (L13). Skip connections are the single most load-bearing design in modern deep learning.

The ResNet-CNN block

ResNet bottleneck — squeeze, convolve, expand · stack this 50× for ResNet-50

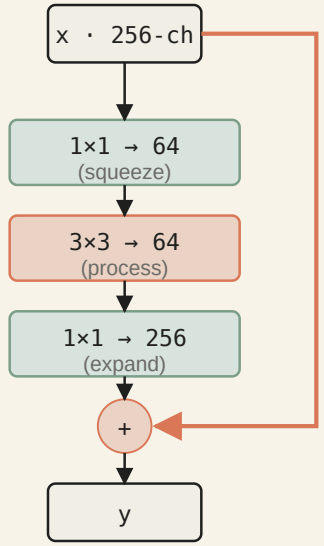
BASIC BLOCK · 2 × (3×3)



2 × (3·3·256·256)
≈ 1.2 M params

→
bottleneck form

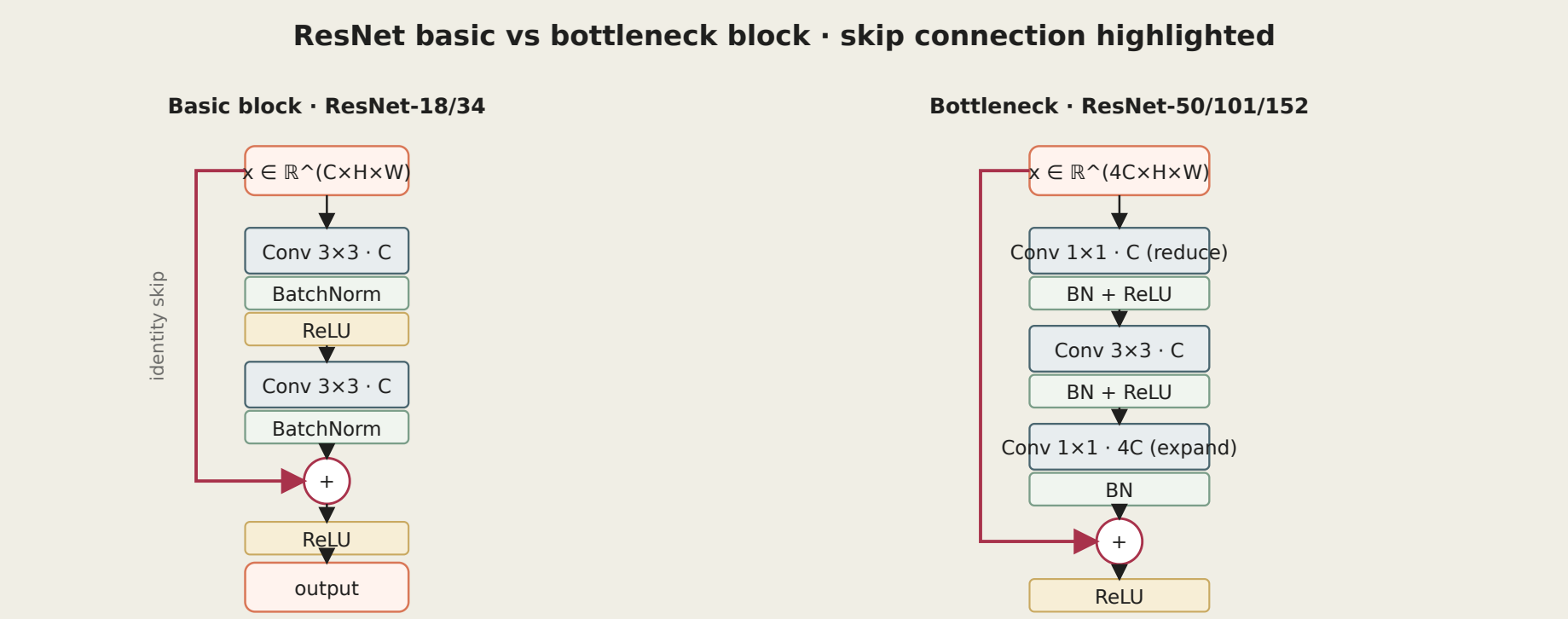
BOTTLENECK · 1×1 → 3×3 → 1×1



256·64 + 3·3·64·64 + 64·256
≈ 70 K params · 17× cheaper

*Same residual topology, **17× fewer parameters per block** — lets you stack 50 blocks in ResNet-50.*

Basic vs bottleneck · annotated



When the shortcut shape doesn't match

The simple skip $y = x + F(x)$ only works if x and $F(x)$ have the **same shape**.

What if the main path:

1. **Downsamples** with stride 2? $F(x)$ is half the spatial size of x. Can't add 14×14 to 28×28.
2. **Changes channel depth?** $F(x)$ has 512 channels, x has 256. Can't add.

> INTUITION
>
> **Analogy · adapter plug.** Your wall socket (x) has one shape; the new appliance plug ($F(x)$) is different. You need an **adapter** (W_s). The projection shortcut is a learnable adapter for the skip connection.

Projection shortcuts · the math

When dimensions don't match:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{W}_s\,\mathbf{x}$$

$\mathbf{W}_s$ is a **1×1 conv with the same stride** as the main branch:

- Same stride (e.g. 2) → matches spatial size.
- Same `C_out` → matches channels.

Now we can add. Everything else stays residual.

```
class Bottleneck(nn.Module):
    expansion = 4
    def __init__(self, c_in, c, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(c_in, c, 1)
        self.conv2 = nn.Conv2d(c, c, 3, stride, padding=1)
        self.conv3 = nn.Conv2d(c, c * self.expansion, 1)
        self.bn1, self.bn2, self.bn3 = [nn.BatchNorm2d(x) for x in (c, c, c*self.expansion)]
        self.shortcut = nn.Conv2d(c_in, c*self.expansion, 1, stride) if stride != 1 or c_in != c*self.expansion else nn.Identity()
```

The ResNet family by depth

MODEL	DEPTH	PARAMS	IMAGENET TOP-1
ResNet-18	18	12 M	69.8%
ResNet-34	34	22 M	73.3%
ResNet-50	50	26 M	76.1%
ResNet-101	101	45 M	77.4%
ResNet-152	152	60 M	78.3%

IN PRACTICE

ResNet-50 is the workhorse. Unless you have a specific reason, start there for any CNN task you face in 2026.

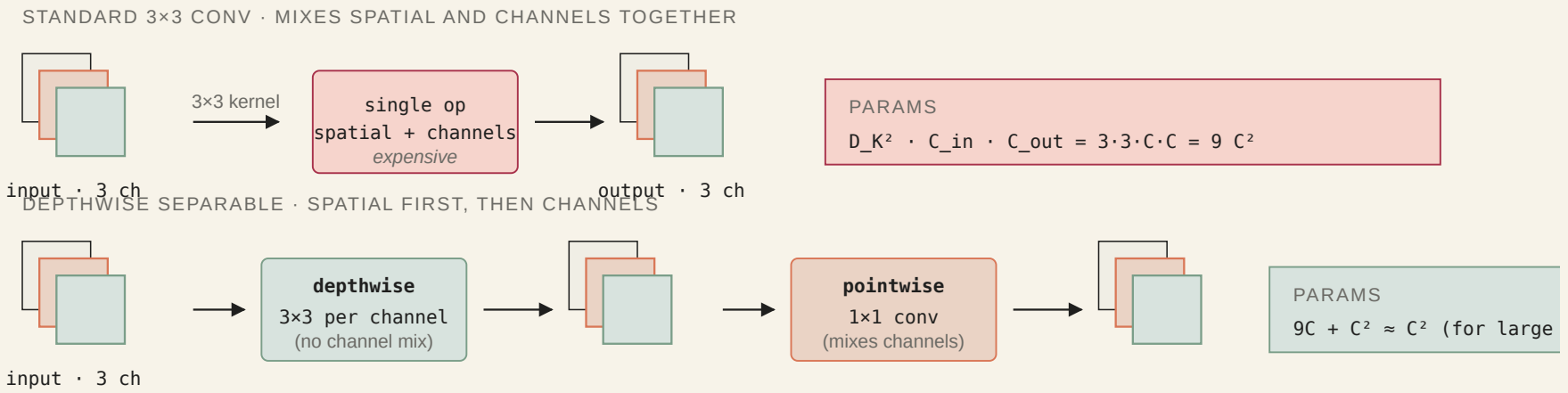
PART 3

MobileNet · efficient CNNs

Depthwise separable convolutions

Depthwise separable · split the work

Depthwise separable — split spatial mixing from channel mixing



COST COMPARISON (C = 128)

Standard 3×3	$9 \cdot 128^2 \approx 147$ K params	→ **9× cheaper**
Depthwise separable	$9 \cdot 128 + 128^2 \approx 17$ K params	almost no accuracy loss

Depthwise separable · the smoothie analogy

A standard convolution does two jobs at once · spatial mixing **and** channel mixing.

INTUITION

Analogy · making a smoothie.

- **Standard conv** · one giant blender. Throw all fruits (channels) in at once → mixes them spatially and combines flavours simultaneously.
- **Depthwise separable** · two-step process.
  i. **Depthwise (spatial):** small blenders, one per fruit. Only blends each fruit *spatially* — never mixes flavours.
  ii. **Pointwise (1×1):** the chef takes one spoonful from each puree and combines flavours into the final smoothie.

Splitting these two jobs makes the operation **dramatically cheaper**.

Depthwise separable · param math

Standard 3×3 conv with $C_{\text{in}} = C_{\text{out}} = C$:

$$\text{cost} = (D_K \cdot D_K \cdot C) \cdot C = D_K^2 \, C^2$$

Depthwise-separable splits into:

1. **Depthwise** · one $D_K \times D_K \times 1$ filter per input channel:
   $\text{cost}_1 = (D_K \cdot D_K \cdot 1) \cdot C = D_K^2 \, C$
2. **Pointwise** · 1×1 conv mixing the C channels into C new ones:
   $\text{cost}_2 = (1 \cdot 1 \cdot C) \cdot C = C^2$
   Total: $D_K^2 \, C + C^2$.

Numeric ($D_K = 3,\ C = 128$):

- Standard: $9 \cdot 16{,}384 = 147{,}456$.
- Depthwise: $9 \cdot 128 = 1{,}152$. Pointwise: $128^2 = 16{,}384$. Total: $\mathbf{17{,}536}$.

~8.4× cheaper. Accuracy drop ~1%. Speedup 8–10×. In every mobile model since 2017.

Worked numeric · depthwise separable

Input `(14, 14, 16)` → Output `(14, 14, 32)`, kernel 3×3.

Standard. $(3 \cdot 3 \cdot 16) \cdot 32 = \mathbf{4{,}608}$ params.

Depthwise separable.

1. **Depthwise.** One 3×3 filter per channel: $(3 \cdot 3 \cdot 1) \cdot 16 = 144$. Output `(14, 14, 16)`.
2. **Pointwise.** 1×1 mixing 16 → 32: $(1 \cdot 1 \cdot 16) \cdot 32 = 512$.

- Total: $144 + 512 = \mathbf{656}$ params. **~7× fewer**, same I/O shape.

Why edge models care about FLOPs

A phone has ~10 W total power; a datacenter GPU draws 300 W just to idle. Efficient nets let you run:

MODEL	FLOPS	IPHONE INFERENCE
ResNet-50	4.1 G	~90 ms
MobileNetV2	0.3 G	~15 ms
MobileNetV3-S	0.06 G	~4 ms

KEY IDEA

Real-time on-device tasks (camera AR, live caption, wake-word) need ≤30 ms budget. MobileNet-style splits are the reason such apps exist.

MobileNet variants · a decade of improvements

MODEL	YEAR	KEY IDEA
MobileNet v1	2017	Depthwise separable + width multiplier
MobileNet v2	2018	Inverted residuals + linear bottlenecks
MobileNet v3	2019	Neural-architecture-search + SiLU
EfficientNet-B0	2019	Compound scaling foundation

PART 4

EfficientNet · compound scaling

Scale depth, width, and resolution together

Compound scaling · tuning a car engine

KEY IDEA

You can scale up an engine by · making it **bigger** (depth) · using **wider** pistons (width) · running on **higher-octane** fuel (resolution).

Any one alone helps a bit. Doing all three together · in balance · gives the best engine.

EfficientNet's insight · the same is true of neural networks. Depth, width, and input resolution should grow *together* for a given compute budget · not one at a time.

How do we make a network "bigger"?

You have a baseline net (a small car engine). Three knobs to make it more powerful:

1. **Depth** · add more layers (more cylinders).
2. **Width** · more channels per layer (wider cylinders).
3. **Resolution** · feed it bigger images (higher-octane fuel).

The old way · pick one knob, turn it all the way up (VGG → depth · WideResNet → width · ProGAN → resolution).

EfficientNet's idea · turn **all three** up *in balance*.

Compound scaling · the rule

Define a single scaling knob ϕ. Choose constants α, β, γ once via grid search.

$$\text{depth } d = \alpha^{\phi}, \quad \text{width } w = \beta^{\phi}, \quad \text{resolution } r = \gamma^{\phi}$$

subject to $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$ (so doubling ϕ doubles compute).

For EfficientNet-B0..B7, the paper found roughly $\alpha = 1.2,\ \beta = 1.1,\ \gamma = 1.15$.

Worked numeric · scaling B0 → B2 ($\phi = 2$).

- Depth: $1.2^2 = 1.44$ → ~44% deeper.
- Width: $1.1^2 = 1.21$ → ~21% more channels.
- Resolution: $1.15^2 \approx 1.32 \rightarrow 224 \cdot 1.32 \approx 296$ (rounded to 300 for B3).

Single ϕ gives a principled way to scale the whole architecture instead of guessing.

EfficientNet scale-up at a glance

MODEL	DEPTH	WIDTH	RES	PARAMS	IMAGENET TOP-1
B0	1.0	1.0	224	5.3 M	77.3%
B1	1.1	1.0	240	7.8 M	79.2%
B3	1.4	1.2	300	12 M	81.6%
B5	1.6	1.6	456	30 M	83.6%
B7	2.0	2.0	600	66 M	84.3%

IN PRACTICE

EfficientNet set the accuracy/param Pareto frontier for 2019–2021 before Vision Transformers took over.

PART 5

Transfer learning

The skill you'll use 90% of the time

The premise

What transfer learning is, in one line

Start from a model that already knows something relevant, adapt it to your new task with a fraction of the data and compute.

KEY IDEA

The pretrained network is a **learned prior**. You're not training from random init — you're fine-tuning a near-working initialization. In most vision tasks this is worth ≈ 1–2 orders of magnitude of training data.

The premise

ImageNet pretraining gives you a **generic vision stack**:

- Early layers detect edges, textures — *universal*, works for any image domain.
- Mid layers detect parts — *mostly transferable* across domains.
- Late layers detect ImageNet-specific object categories — *domain-specific*, usually replaced.

KEY IDEA

Transfer learning rule — more data in your new domain → unfreeze more layers.

Transfer learning · the core problem

Scenario. A botanist gives you 5,000 photos of flowers and wants a 102-class classifier.

Option 1 · train from scratch. Design a ResNet-50, train on 5,000 images.
Problem · 5,000 isn't enough to learn what an *edge*, *texture*, or *petal* even is from random noise. The model overfits badly.

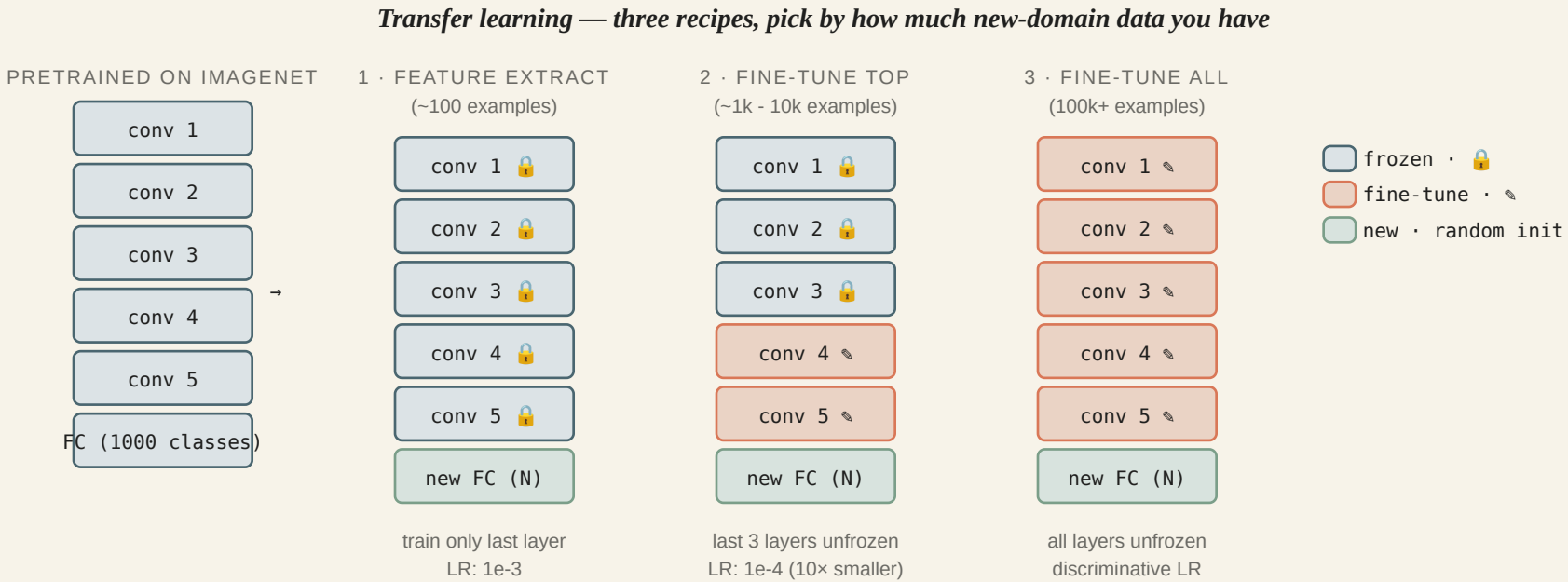
Option 2 · transfer learning. Take a ResNet-50 already trained on ImageNet (1.2M images). It already knows edges, textures, fur, eyes. **Adapt** this powerful feature extractor to flowers.

Almost always Option 2 wins when labels are scarce. Now · how to adapt?

Jargon unpacked

- **Backbone** · the conv body of the pretrained net. The "feature extractor."
- **Head** · the final classifier layers. We discard the original ImageNet head (1000 classes) and add our own (e.g. 102 flower classes).
- **Freezing** · setting `requires_grad=False`. Optimizer skips that layer; it's a fixed feature extractor.

Three recipes



*Decision rule · **more data → unfreeze more layers**; always use a smaller LR for the pretrained layers.*

Transfer learning · by data size

DERIVATION

YOUR DATA	RECOMMENDED RECIPE	WHAT TO FREEZE
< 100 labels	Linear probe	everything except head
100 - 1k	Fine-tune top layers	freeze conv1-3
1k - 10k	Fine-tune whole backbone	nothing (with discriminative LR)
10k - 100k	Fine-tune + LR scheduling	nothing, bigger LR
> 100k	Probably train from scratch	—

INTUITION

Smaller data → more frozen. Larger data → more trainable. If you have 1M labels in your domain, you likely don't need transfer at all (but it rarely hurts as a warm start).

The bitter truth · negative transfer

WATCH OUT

Sometimes transfer learning **hurts**. Medical MRI → ImageNet-pretrained ResNet. Natural photos teach features that don't transfer to grayscale medical modalities.

Symptoms:

- Val loss starts higher with pretraining than from scratch
- Fine-tuned performance plateaus
- Early layers still detect ImageNet-like edges; later layers never adapt

  Fix · try from-scratch, or use **domain-specific pretraining** (RadImageNet for medical, SatCLIP for satellite). Generic features aren't universal.

Discriminative (layer-wise) learning rates

When you do unfreeze early layers, they should learn more *slowly* than late layers — early layers are already good.

```
# PyTorch — different LR per param group
params = [
    {"params": model.conv1.parameters(), "lr": 1e-5},  # early   · slow
    {"params": model.conv2.parameters(), "lr": 1e-5},
    {"params": model.conv3.parameters(), "lr": 3e-5},
    {"params": model.conv4.parameters(), "lr": 1e-4},
    {"params": model.conv5.parameters(), "lr": 3e-4},  # late    · fast
    {"params": model.fc.parameters(),    "lr": 1e-3},  # new     · fastest
]
opt = torch.optim.AdamW(params, weight_decay=0.01)
```

IN PRACTICE

fastai popularized "1cycle + discriminative LRs" for transfer learning — often the right defaults for small-data fine-tuning.

When transfer learning fails

WATCH OUT

Large domain gap. ImageNet (natural photos) → medical X-rays, satellite imagery, microscopy. Early-layer features may still transfer; late-layer features definitely won't.

Very different image sizes. ImageNet is 224×224; medical imaging may be 1024+. You may need to resize or re-pretrain.

Very small target dataset (≪ 100 examples). Even linear probing won't save you — consider self-supervised pretraining on your domain first (L17).

In those cases — pre-train on a closer domain, or use self-supervised methods (coming in L17).

Pick-the-right-backbone table

SCENARIO	BACKBONE	WHY
ImageNet-like natural photos	ResNet-50 or ConvNeXt	strong + well-supported
Very small labeled data	CLIP ViT features	cross-domain generalization
Edge / real-time	MobileNet-V3	latency budget
High-res medical / satellite	ConvNeXt-large or DINOv2	captures fine detail
Arbitrary RGB, unknown domain	DINOv2 frozen features	best general-purpose SSL

IN PRACTICE

In 2026, **DINOv2** (self-supervised on 142M images) is often the starting point for vision-backbone features — even better than ImageNet-supervised ResNets for downstream tasks.

Loading a pretrained backbone · PyTorch

```python
import torch, torch.nn as nn
import torchvision.models as M

# 1. Load pretrained weights
model = M.resnet50(weights=M.ResNet50_Weights.IMAGENET1K_V2)

# 2. Freeze everything
for p in model.parameters():
    p.requires_grad = False

# 3. Replace the 1000-way classifier with N-way
n_classes = 10
model.fc = nn.Linear(model.fc.in_features, n_classes)   # new, trainable by default

# 4. Train only the new fc
opt = torch.optim.AdamW(model.fc.parameters(), lr=1e-3)

# 5. Later, unfreeze progressively
for p in model.layer4.parameters(): p.requires_grad = True
```

The `timm` ecosystem

For 2026, stop hand-rolling architectures:

```python
import timm

# 500+ pretrained vision models in one line
model = timm.create_model('resnet50',        pretrained=True, num_classes=10)
model = timm.create_model('efficientnet_b3', pretrained=True, num_classes=10)
model = timm.create_model('convnext_base',   pretrained=True, num_classes=10)
model = timm.create_model('vit_base_patch16_224', pretrained=True, num_classes=10)
```

IN PRACTICE

`timm` by Ross Wightman · the de facto vision model zoo. Every competitive Kaggle vision solution uses it.

Summary · Lecture 8 — summary

- **Inception module** — parallel branches of 1×1, 3×3, 5×5, pool; let SGD pick the receptive field.
- **1×1 convolutions** — channel mixing + cheap bottlenecks; in every modern architecture.
- **ResNet bottleneck** — 1×1 → 3×3 → 1×1 + skip; 17× fewer params per block than basic residual.
- **Depthwise separable** — split spatial from channel mixing; MobileNet's foundation.
- **Compound scaling** (EfficientNet) — scale depth × width × resolution together.
- **Transfer learning** recipes — feature-extract · fine-tune top · fine-tune all. Match to data size.
- **Practically** · start from `timm.create_model('resnet50', pretrained=True)` and go from there.

Read before Lecture 9

Bishop Ch 10 + CS231n OD notes (UDL doesn't cover detection/segmentation).

Next lecture

Detection & Segmentation — R-CNN → Faster R-CNN, YOLO, IoU, NMS, U-Net, Mask R-CNN, zero-shot SAM.

NOTEBOOK

Notebook 8 · `08-transfer-learning.ipynb` — fine-tune ResNet-50 on Flowers-102 with discriminative LRs, measure effect of freezing depth.