WEEK 2: Deep Convolutional Models - Case Studies

Why Study Case Studies in ConvNets?

- **Building Blocks Recap**: Last week covered basics — convolutional layers, pooling layers, & fully connected layers.
- **How Architectures Evolve**: Recent computer vision research focuses on combining these blocks effectively to form state-of-the-art ConvNets.
- **Learning by Examples**:
  - Just like learning coding by reading others' code, you gain intuition for neural nets by studying architectures already proven effective.
  - Neural network structures that perform well on one task (like recognizing cats/dogs/people) often excel in other CV tasks too — transferable designs!
- **Applicability**:
  - Can take architectures from other tasks (e.g., self-driving cars) and adapt them for your own CV problem.
  - Reviewing real architectures makes concepts concrete and actionable.
- **Research Literacy**:
  - After reviewing case studies, you'll be able to read, understand, and appreciate cutting-edge research papers in computer vision.
  - Satisfying to grasp ideas behind famous works (helps in both classes and real-world applications).

What's Next in the Course?

- **Classic Networks**: You'll study key architectures:
  - ***LeNet-5*** (from 1980s)
  - ***AlexNet*** (game changer in deep learning)
  - ***VGGNet*** (known for simplicity)
- **Deep Networks**:

- ***ResNet* ("Residual Network"):** Enabled training networks with 152+ layers via clever design tricks.
- ***Inception Network*:** Introduces novel approaches for effective deep learning.

Big Picture

- **Intuition from Examples**: Studying these will develop your intuition to build effective neural nets.
- **Cross-Disciplinary Impact**: Innovations from CV architectures (e.g., ResNet, Inception) spread to other AI fields—valuable knowledge, even beyond vision.
- **Takeaway**: Case studies help you become a better designer and consumer of neural network architectures.

Focus on *how* architectures are constructed, *why* certain design choices work, and what patterns emerge across successful models. This knowledge will help you adapt, invent, and succeed in advanced AI/ML projects!

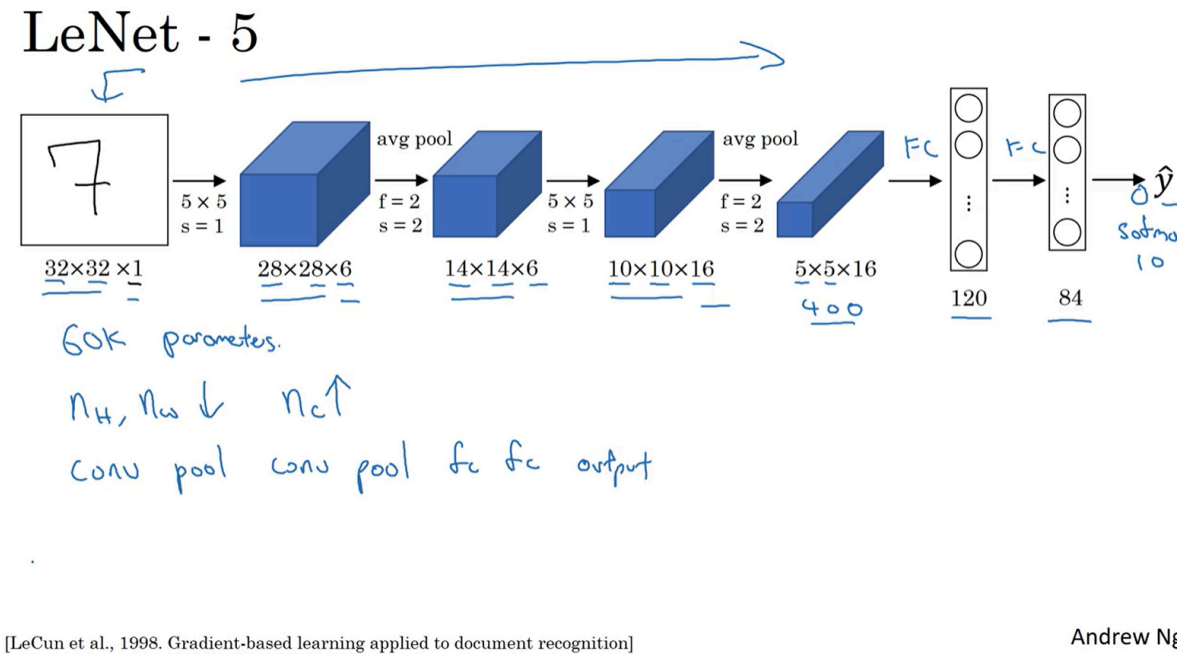
Case Studies

Classic Neural Network Architectures: Overview

- Discussed three landmark CNN architectures: LeNet-5, AlexNet, and VGGNet (VGG-16).
- Key features: progression in structure, parameter scale, and performance.

---

1. LeNet-5



[LeCun et al., 1998. Gradient-based learning applied to document recognition]

Andrew Ng

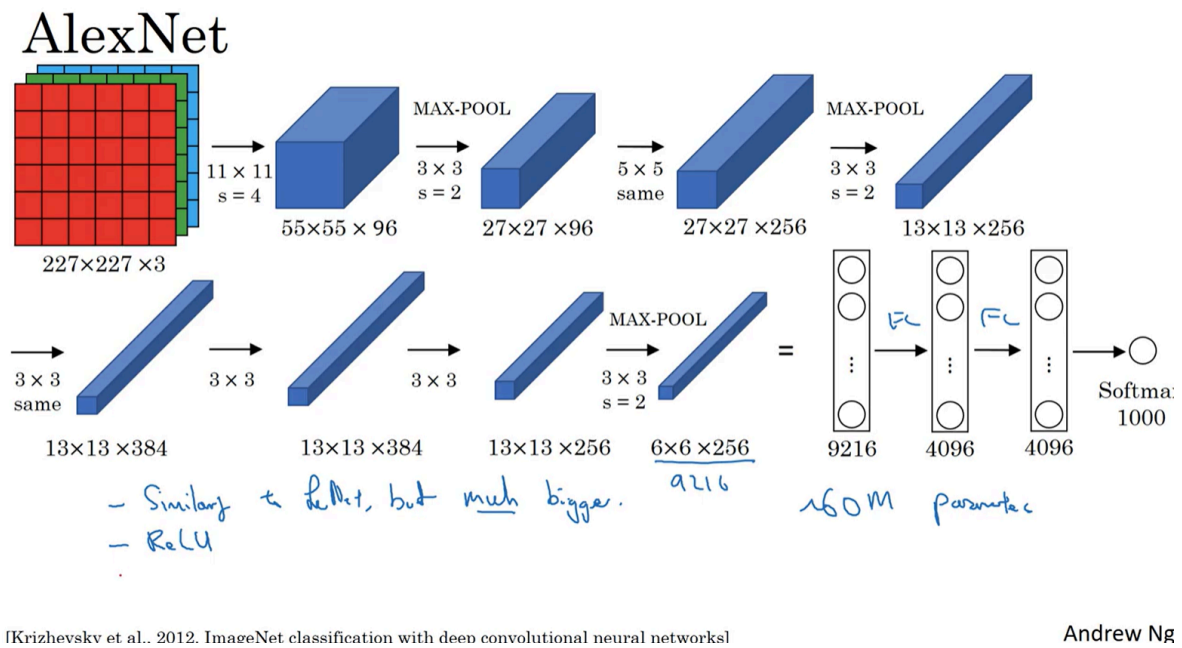
- **Invented for handwritten digit recognition (grayscale images)**
  - Input: 32x32x1 images (MNIST dataset).
- **Architecture:**
  - First Conv Layer: 6 filters of size 5x5, stride 1, no padding → output: 28x28x6
  - Pooling Layer: Average pooling (filter 2x2, stride 2) → output: 14x14x6
  - Second Conv Layer: 16 filters of 5x5 → output: 10x10x16 (no padding, “valid” convolutions)
  - Pooling Layer: Another average pooling → output: 5x5x16
  - Flatten: 400 nodes, connected fully to next layer
  - Fully Connected Layer: 120 neurons
  - Another Fully Connected Layer: 84 neurons
  - Final Output: softmax over 10 classes (digits 0–9); originally used a different classifier
- **Parameters:** ~60,000—tiny by today’s standards.
- **Design Patterns:**
  - As depth increases, height and width decrease (spatial dims lower, channel dims rise).

- Common sequence: Conv → Pool → Conv → Pool → FC → Output

- **Historical Notes:**
  - Used sigmoid/tanh activations; no ReLU.
  - Complex filter-channel wiring for computational savings.
  - Non-linearity (sigmoid) applied after pooling (rare in modern nets).
  - Paper hard to read; best to focus on sections 2–3.

---

2. AlexNet



- **Revolutionized computer vision with deep learning (ImageNet challenge).**
- Input: 227x227x3 (official paper mentions 224, but numbers match 227).

ImageNet dataset in bullet points:

- ImageNet is a large-scale image dataset containing over 14 million high-resolution images annotated with labels.
- Images are organized according to the WordNet hierarchy into about 22,000 categories (synsets), mostly nouns.
- Each category typically has hundreds to thousands of images illustrating that concept.
- Over 1 million images have bounding box annotations marking object locations.
- The dataset supports key computer vision tasks like image classification, object detection, and localization.
- ImageNet was the foundation for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark for visual recognition algorithms.
- Popular deep learning models (like AlexNet, VGG, ResNet) were trained and evaluated on ImageNet.
- Images vary widely in resolution; standard input size for training models is typically resized to 224 x 224 pixels.
- ImageNet is publicly available for academic research and educational use but does not own image copyrights.

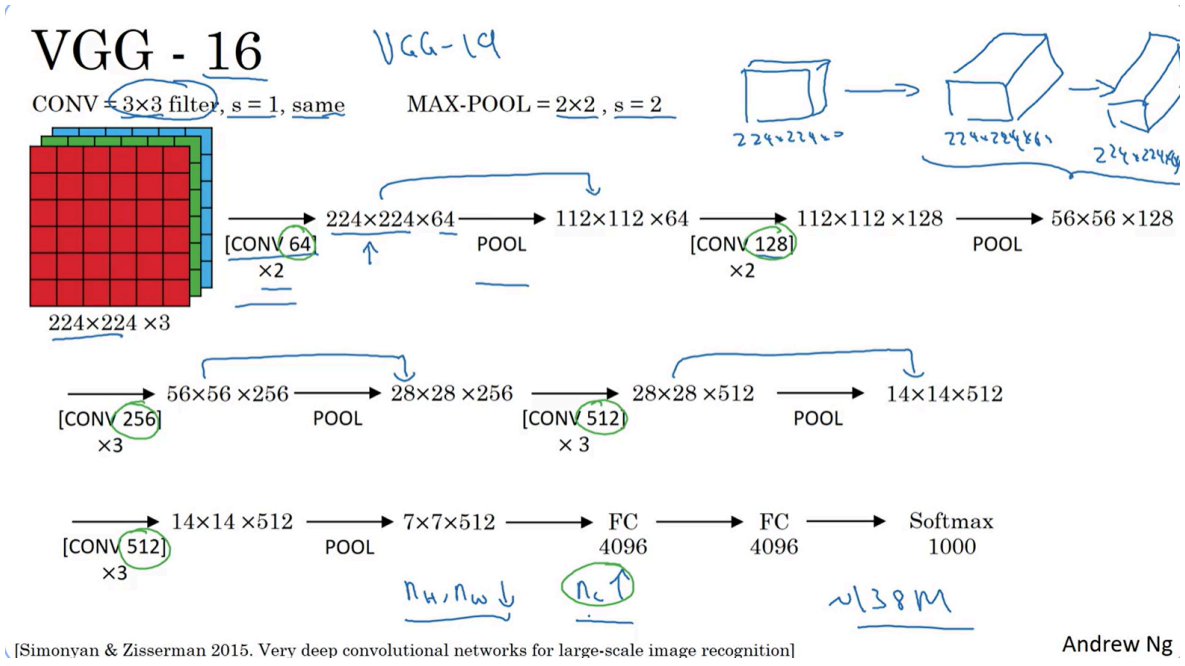
This dataset has been instrumental in advancing computer vision and deep learning research.

- **Architecture:**
  - First Conv Layer: 96 filters, size 11x11, stride 4 → output quickly shrinks.
  - Max Pooling: 3x3 filter, stride 2.
  - Second Conv Layer: 256 filters, size 5x5 (“same” padding).
  - Max Pooling again; output: reduced height/width after each pooling.
  - Later Conv Layers: Multiple 3x3 convolutions, increasing filters (up to 384, 256).
  - Final Conv Layer: max pooling to 6x6x256.
  - Flatten: 9,216 nodes → Fully Connected layers.
  - Output: 1000-class softmax (for ImageNet).
- **Parameters:** ~60 million (1000x more than LeNet!).

- **Key Innovations:**
  - ReLU activations (improved performance).
  - Trained across two GPUs (for speed).
  - Local Response Normalization (LRN) layer—rarely used in modern nets.
- **Impact:** Proved deep learning works for computer vision. ImageNet dataset was instrumental.

---

3. VGGNet / VGG-16



- **Designed for simplicity and uniformity.**
  - Input: 224x224x3
- **Architecture:**
  - Only 3x3 Conv filters, stride 1, always “same” padding.
  - Max Pooling: always 2x2 with stride 2.
  - Pattern: Stacks of Conv layers with increasing filter counts.
    - 2 x Conv (64 filters) → pool → 2 x Conv (128) → pool → 3 x Conv (256) → pool → 3 x Conv (512) → pool → 3 x Conv (512) → pool.

💡

- **“2 x Conv (64 filters)” means:**
  There are two convolutional layers stacked one after another.
  - Each layer uses **64 filters** (also called kernels).
  - Each filter is usually of size 3×3 (for VGG architecture) with stride 1.

Breaking it down step-by-step:

- “2 x Conv (64 filters)” →
  - First Conv layer: 64 filters
  - Second Conv layer: 64 filters
- Then a **pooling layer** (usually max pooling) is applied.
- Next, “2 x Conv (128 filters)” →
  - Third Conv layer: 128 filters
  - Fourth Conv layer: 128 filters
- Pooling.
- “3 x Conv (256 filters)” →
  - Fifth Conv layer: 256 filters
  - Sixth Conv layer: 256 filters
  - Seventh Conv layer: 256 filters
- Pooling.
- … and so on for 3 x Conv (512) ...

  - Final Conv output: 7x7x512 → Flatten → FC layers (sizes up to 4,096) → 1000-way softmax.

- **Parameters:** ~138 million (huge even by modern standards).
- **Design Principles:**
  - Doubling filters at each stage (64 → 128 → 256 → 512).
  - Height/width halved at each pooling layer.
  - Network very uniform and simple—popular with researchers.

- **Variants:** VGG-19 (larger, but VGG-16 widely used).
- **Downside:** Large number of parameters (memory intensive).

General Patterns in Modern CNNs:

- As you go deeper, spatial dimensions reduce and number of channels increases.
- Common layer sequence: [Conv → Pool]*N → [FC]*M → Output
- Use of softmax for classification in output layer.
- ReLU activations now common (AlexNet onward).
- Modern nets are much larger, up to hundreds of millions of parameters.

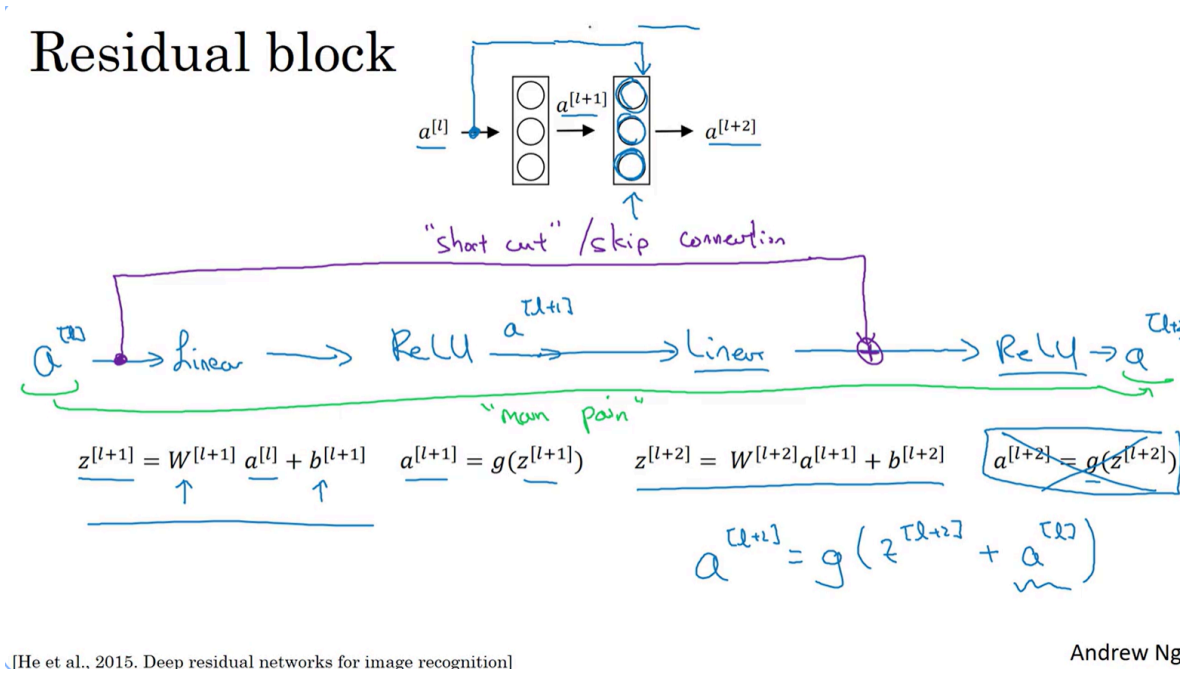
Suggestions for Further Reading:

- For *easier* reading: start with AlexNet and VGG papers.
- LeNet-5 paper is more complex (historical quirks).

Key Takeaways:

- LeNet: Introduced core CNN structure; small scale.
- AlexNet: Deep learning breakthrough; big scale, ImageNet.
- VGG: Simple, stackable design; huge scale, uniform architecture.
- All classic architectures show how deeper networks, more data, and systematic design improve visual recognition dramatically.

ResNets (Residual Network)



ResNets: Key Concepts & Intuitions

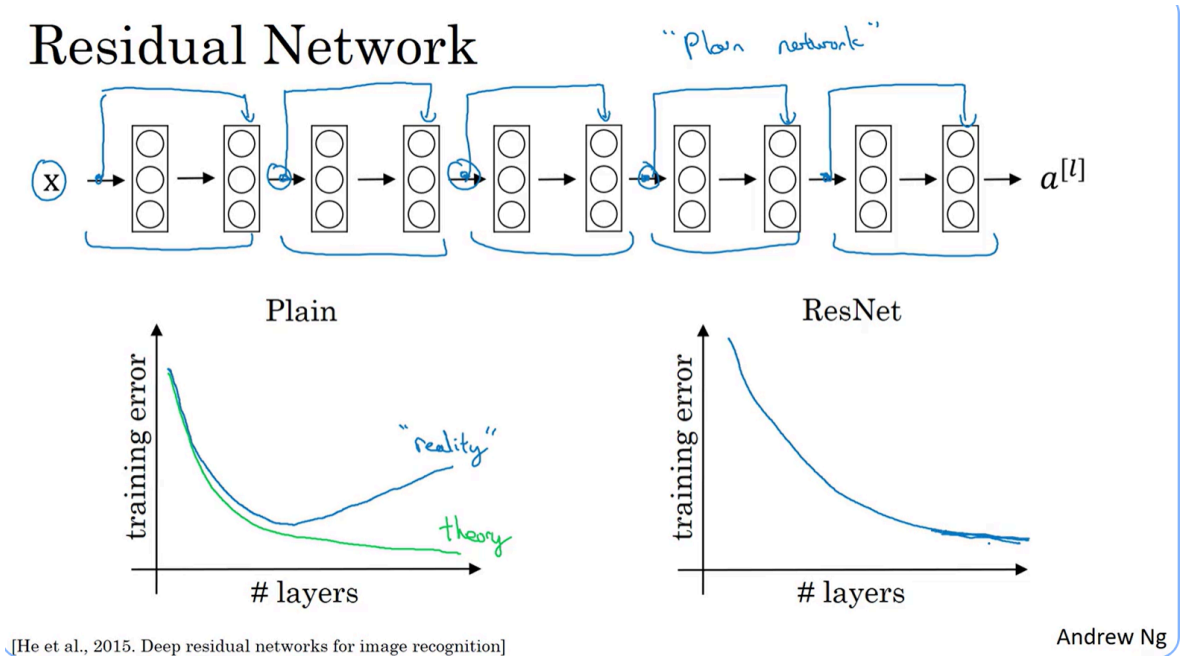
- **Problem in Deep Networks:**
  - Very deep neural networks are tough to train due to *vanishing* and *exploding gradients*.
- **ResNet Solution:**
  - Uses **skip connections** (aka shortcuts) that allow activations from an earlier layer to be fed directly into a much deeper layer.
  - Enables training of networks with *over 100 layers*.

Residual Block: The Building Block

- Traditional block:
  - Input activation $a^{[l]}$ goes through:
    - **Linear transformation**: $z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$
    - **ReLU activation**: $a^{[l+1]} = g(z^{[l+1]})$
    - **Next layer** (same steps) → final output $a^{[l+2]}$
- **Information Pathways:**
  - Normally, all info must flow through every transformation step ("main path").
- **Residual/Shortcut Connection:**

- Directly copies $a^{[l]}$ to be added to the output **before the ReLU nonlinearity**.
- *New output*: $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$
- Called *skip connection* (information "skips" layers and goes deeper directly).

ResNet Architecture Construction



- ResNet is built by **stacking** multiple residual blocks with skip connections.
- "Plain network": no skip connections.
- "Residual network": every two layers have added skip connections, forming residual blocks.

Empirical Results & Theoretical Insights

- *Theory*: Deeper networks should perform better (lower training error).
- *Practice* (Plain network): After some depth, **training error increases** again—optimization is harder.
- With ResNets:
  - Training error **continues to decrease** as network grows deeper (even >100 layers).
  - ResNets help overcome vanishing/exploding gradients.
  - Deeper networks become trainable without performance loss.
- *Note*: Some experiments reached over 1000 layers (rarely practical).

Main Intuition & Takeaway

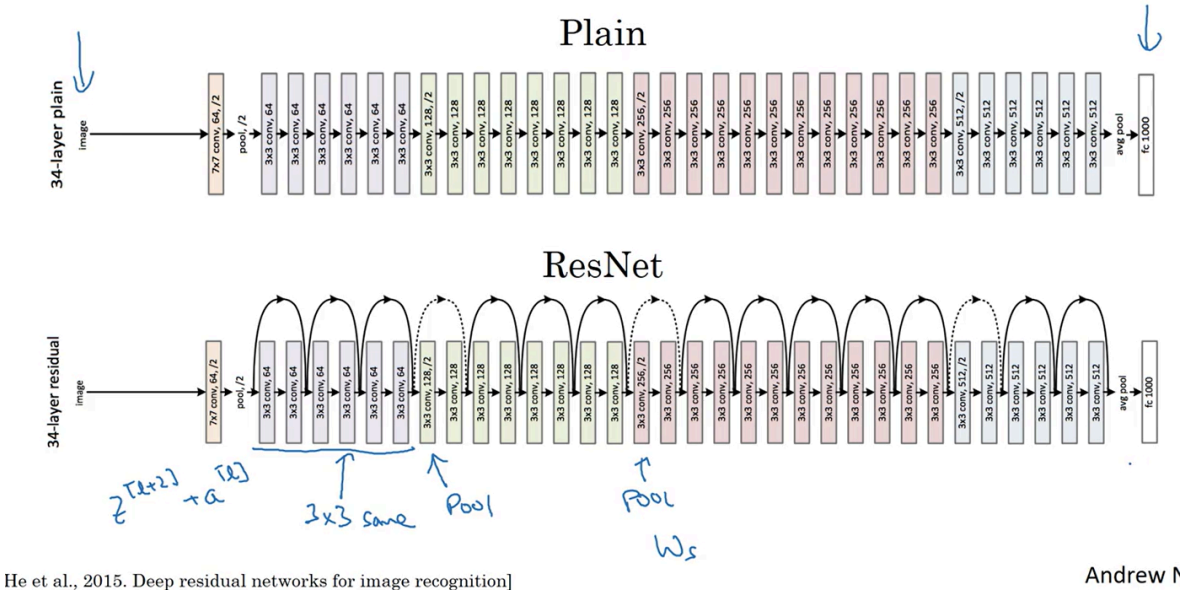
- **Skip connections** allow direct information/gradient flow.
- Major help for training *very deep* neural networks.
- Enables state-of-the-art models and architectures in deep learning.
- You will implement residual blocks in programming exercises for hands-on intuition.

Inventors

- ResNet developed by **Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun**.

> 💡 ***The name "skip connection" comes from the way these connections skip over one or more layers in a neural network. Instead of passing data only from one layer to the next in sequence, a skip connection takes the output from an earlier layer and feeds it directly to a deeper layer, bypassing (or "skipping") the intermediate layers in between***

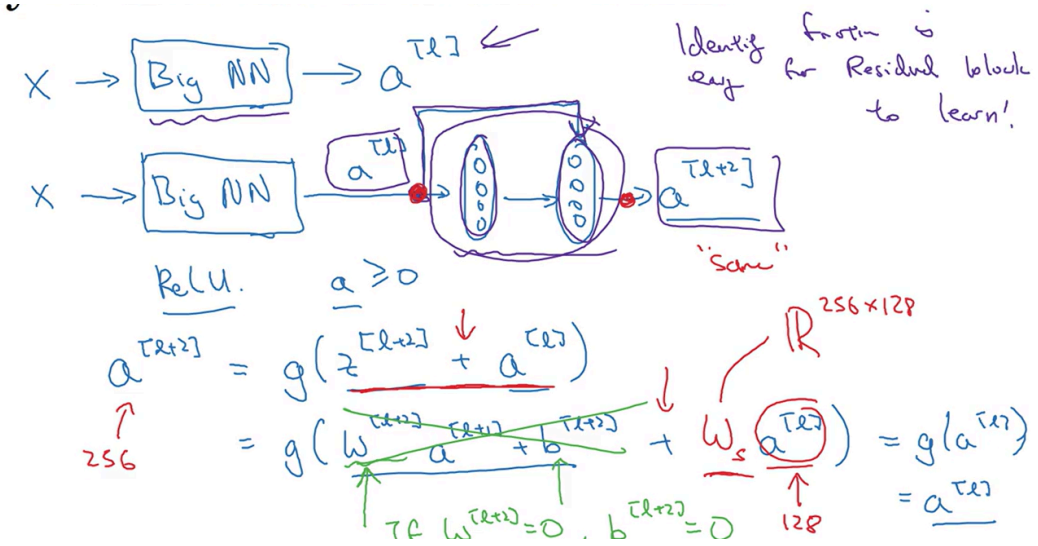
Why ResNets Work?



He et al., 2015. Deep residual networks for image recognition]

Andrew N

- **ResNet Architecture:**



  - Uses *residual blocks* which add a shortcut/skip connection.
  - Allows deeper networks to be trained without hurting training performance.

- **Training Deep Networks:**
  - In plain (regular) deep networks, increasing depth often degrades training accuracy.
  - ResNets overcome this by making it easy to learn the *identity function* via skip connections.
  - Key: If deeper layers don't help, the network can simply copy inputs like a shallower model.
- **How the Residual Block Works:**
  - Suppose you have a neural network outputting $a^{[l]}$.
  - You add two extra layers (making it deeper) and wrap them as a residual block.
  - Output after block:$a^{[l+2]} = g(z^{[l+2]}) + a^{[l]}$, where g is the activation function (e.g., ReLU).
    - The skip connection provides $a^{[l]}$.
    - If weights and bias in new layers become zero, block simply passes input ($a^{[l]}$) forward—realizing the identity function.
- **Role of ReLU Activation:**
  - ReLU gives outputs ≥ 0.
  - If new layers in residual block have zero weights & bias, output is: $a^{[l+2]} = ReLU(a^{[l]}) + a^{[l]} = a^{[l]}$.

    - Easy for network to simply "copy" if deeper layers aren’t helpful.
- **L2 Regularization:**
    - Tends to shrink weights, making it even easier for the residual block to learn the identity function if needed.
- **Impact on Performance:**
    - Adding residual blocks means extra layers don’t degrade performance—they may *improve* it.
    - Unlike plain nets, ResNets are guaranteed not to hurt performance by extra depth, and often help.
- **Dimension Matching:**
    - For addition, dimensions of $z^{[l+2]}$ and $a^{[l]}$ must match.
    - Uses "same convolution" to keep output dimensions equal.
    - If mismatch (e.g., 128-d input, 256-d output), introduce a matrix W_s to transform dimensions for addition.
        - W_s could be learnable or a fixed transformation (e.g., zero padding).
- **ResNets in Convolutional Architectures:**
    - Residual connections commonly sit atop stacks of 3x3 same convolution layers.
    - Dimensions stay equal, enabling straight addition for the skip connection.
    - In case of pooling or change in feature map size, use matrix W_s for adjustment.
- **Practical Example (from paper):**
    - Plain network: Image → Conv layers → Softmax output.
    - ResNet: Same, but includes skip connections over conv blocks.
    - Stacks: Conv-Conv-Conv-pool, then more convs; at end, fully connected + softmax.

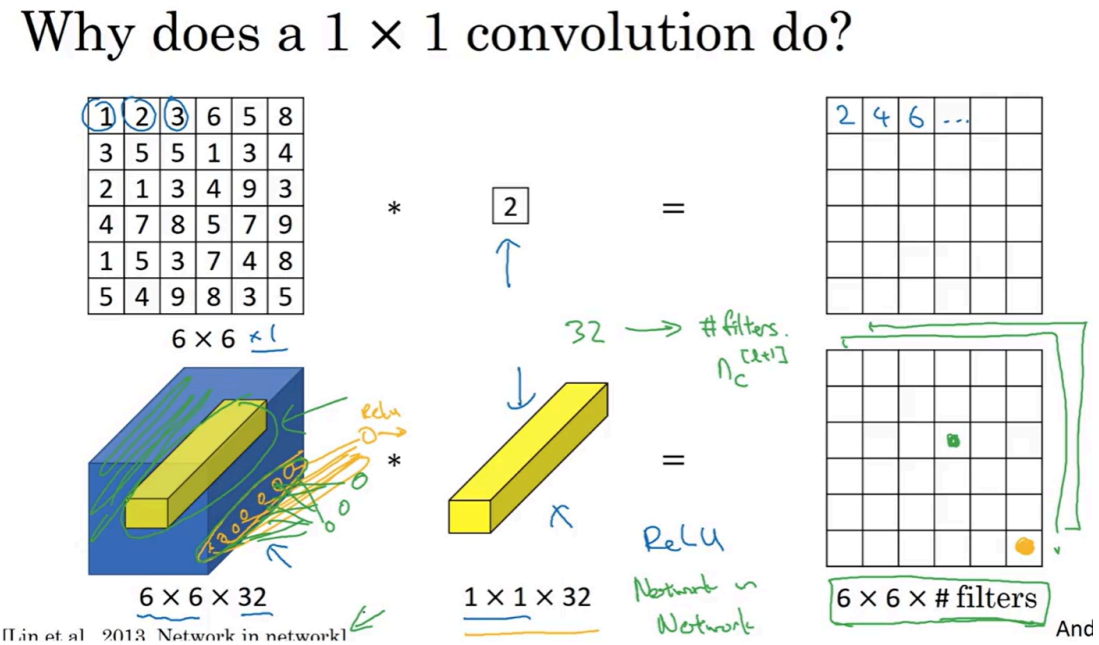
Summary:

- ResNets succeed because their architecture always lets deeper layers revert to the identity function if deeper modeling isn’t useful.
- Key: *Depth doesn’t harm; skip connections enable always matching or improving the shallow baseline*.
- Facilitates very deep, trainable networks with high accuracy and stable training.

- Cost Effectiveness:
  - **ResNet models are actually more cost-effective for deep learning tasks**, since you get better results for similar computational cost.
  - Training a plain deep network may waste cost with poor accuracy, while ResNet achieves higher accuracy, faster convergence.
  - Lightweight variants like **ResNet-18 are used for cost-conscious scenarios**; larger ones (ResNet-50, ResNet-101) for advanced cases.

Networks in Networks and 1×1 Convolutions

1x1 Convolutions: Core Concept



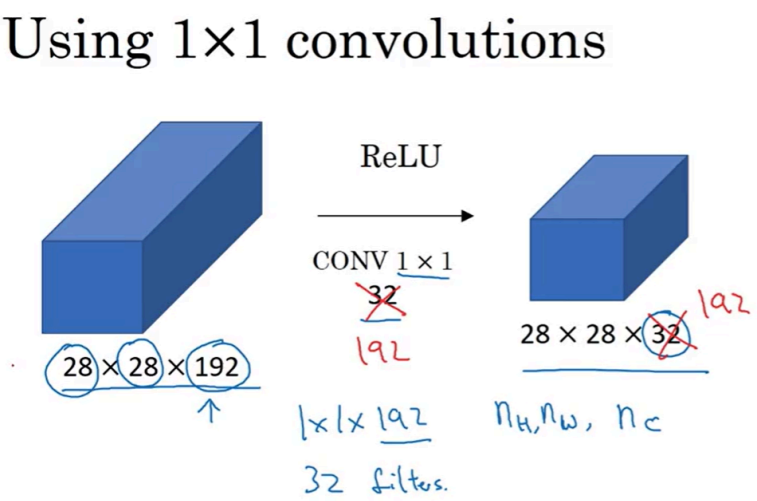
- A **1x1 convolution** uses a filter of size 1x1 (per channel) to perform operations at each spatial position, across all channels of the input.
- For an input image (e.g., 6x6x1), convolving with a 1x1 filter essentially multiplies each pixel by a constant—*not very useful for single-channel*.

- However, with multi-channel input (e.g., 6x6x32), a 1x1 filter operates across the **channel dimension**.

How 1x1 Convolution Works

- At each spatial location (e.g., "36 positions" in a 6x6 image), the 1x1 convolution takes all the channel values, applies learned weights, and outputs a single value.
- Mathematically, this is similar to a **fully connected (FC) layer** for each spatial position:
  - *Input*: all channel values at a given position (e.g., 32 numbers)
  - *Weights*: filter weights per channel
  - *Non-linearity*: typically a ReLU applied after
- Can use multiple filters in parallel to output multiple channels, not just one.

Networks in Networks Interpretation



- 1x1 convolution = applying a small neural network independently at each spatial position.
- Called **Network in Network** because each filter acts as a mini-FC network at each position in height/width.
- Key reference: M. Lin, Q. Chen, S. Yan paper ("Network in Network").

Use Cases: Channel Reduction

- Useful for reducing the number of channels (feature maps) in a volume:
  - Example: 28x28x192 to 28x28x32 using 32 filters of shape 1x1x192.

    - Each filter looks across all 192 channels and produces one output channel.
- *Pooling layers* reduce spatial dimensions (height/width), while **1x1 convolutions** reduce channel dimensions without affecting spatial dimensions.

Functions and Flexibility

- **Channel adjustment**: Reduce, keep, or even increase the number of channels.
- **Non-linear transformation**: Adds extra non-linearity, making the network more expressive.
- **Computational efficiency**: Reducing channels saves memory and computation, especially in deep networks.

Applications and Further Insights

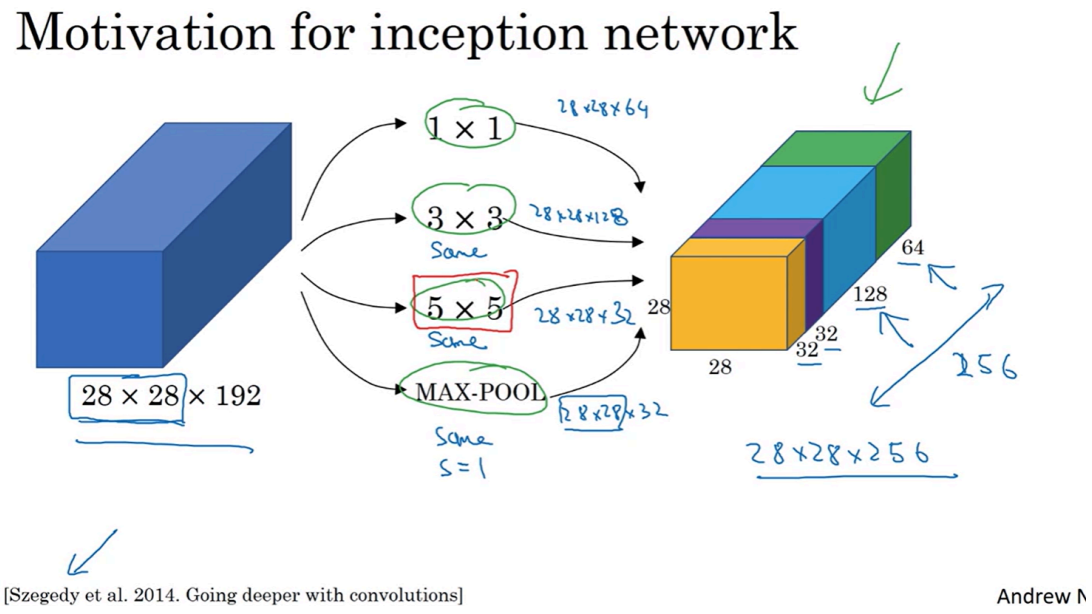
- Widely used in architectures like the **Inception Network** (introduced in follow-up lectures).
- Helps build *complex functions* by adding extra layers without increasing spatial size.
- Can be used wherever a more complex, non-linear mapping across channels is needed, e.g., after pooling, before merging multi-path outputs.

Summary

- **1x1 convolution = spatially local fully connected layer applied per pixel**
- Lets you *control and transform channel depth* at any level of the network.
- Foundational for modern architectures aiming for efficiency and flexibility in deep learning.

Inception Network Motivation

Core Intuition



- **Why inception?** In classic CNNs, choosing a filter size (like 1x1, 3x3, 5x5, or pooling) is hard—each works best in different scenarios.
- **Solution:** Inception module runs **all types of convolutions (and pooling) in parallel**—the network learns which mix of features to extract at each layer.
- **Result:** Higher representational power, spatial flexibility, and feature diversity.

Standard Notations

- H: Height of input image/feature map
- W: Width of input image/feature map
- D_{in}: Number of channels in input (Depth)
- $k \times k$: Filter spatial dimensions
- Number of filters: Determines output channel size (D_{out})
- **Input volume notation:** $H \times W \times D_{in}$
- **Output volume notation:** $H' \times W' \times D_{out}$ (often H′=H,W′=W with “same” padding)

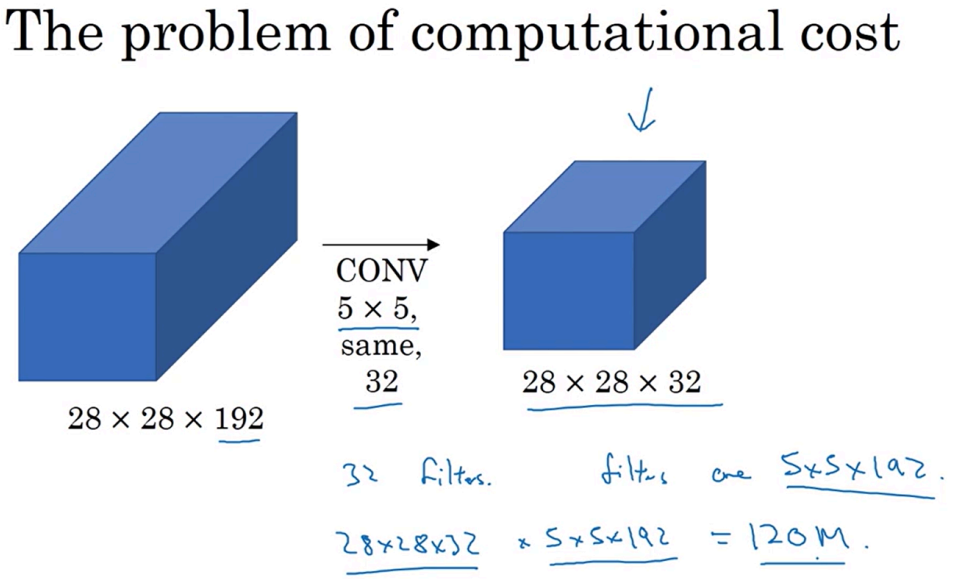
Typical Inception Module Structure

- **All branches receive the same input (e.g., 28×28×192):**
  - **1x1 convolution:** →28×28×64

- **3x3 convolution:** →28×28×128
- **5x5 convolution:** →28×28×32
- **Max pooling + 1x1 conv:** →28×28×32

- **Branch outputs are concatenated**
  - Total output: 28×28×(64+128+32+32)=28×28×256

Multiplications Required (Computational Cost)



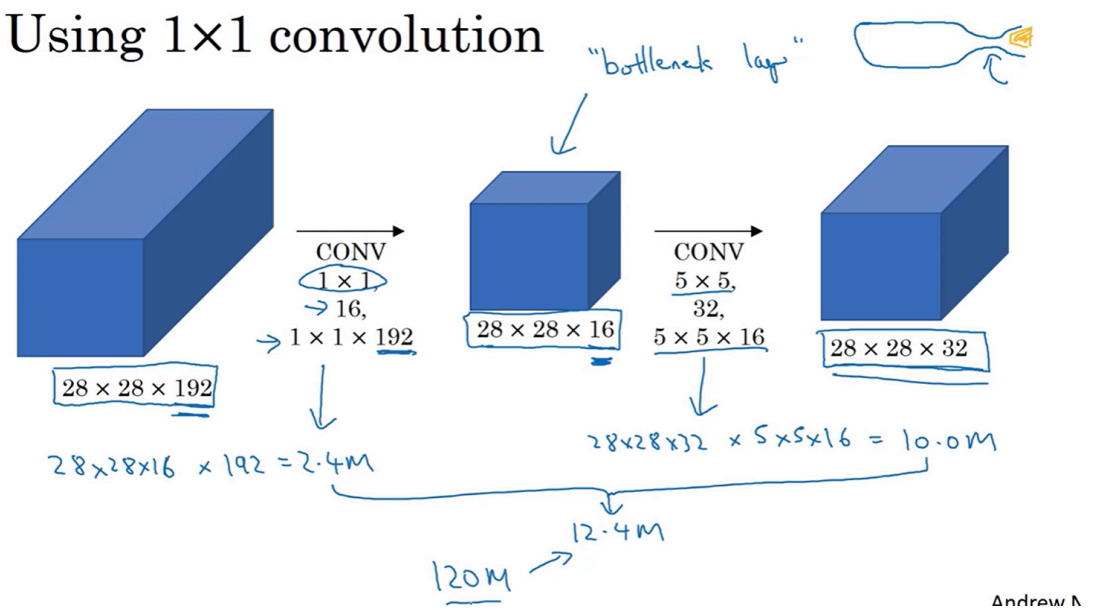
- Formula for any $k \times k$ conv branch:

$$H \times W \times D_{out} \times k \times k \times D_{in}$$

- **Example:** 5x5 conv, 32 filters: 28×28×32×5×5×192=120M multiplications

---

Bottleneck Layer (1x1 Convolution)



- **Motivation:** Direct large filters (e.g., 5x5) are expensive if input is deep (D_{in} is large)
- **Solution:** Apply 1x1 convolution first (shrinks depth), then larger $k \times k$ convolution
- **How it works:**
  - Apply 1x1 conv: 28×28×16 filters (cost: 2.4M mults)
  - Then apply 5x5 conv on reduced channels: (cost: 10.0M mults)
  - **Total:** 12.4M, much less than 120M without bottleneck

💡 **Bottleneck Process in Inception Networks:**

- A bottleneck layer uses 1×1 convolutions to reduce the number of feature channels before applying larger kernels like 3×3 or 5×5.
- The main goal is to decrease computational cost by lowering the number of parameters in subsequent layers.
- When implemented correctly, the bottleneck process maintains the network's performance, provided the representation is not shrunk excessively.
- Proper channel reduction preserves important information while making the model much more efficient.
- Notable effect: computational savings (up to 10x less computation) without accuracy loss.
- **Conclusion:** Bottleneck layers are a smart technique for building deep neural networks efficiently, as they balance speed and accuracy.

Advantages of Inception Design

- **Feature selection:** Allows the network to select features at multiple scales per location
- **Efficiency:** Computational optimization via bottleneck layers means deep/wide networks are possible
- **Modularity:** Easy stacking; can tweak filter sizes/outputs for performance/power needs

---

Research Scope and Impact

- **Origin:** Introduced in GoogleNet/Inception v1 paper (Szeg edy et al., 2014)
- **Influence:** Inspired further architectures: Inception v2/v3/v4, ResNeXt, Xception, EfficientNet, MixNet
- **Applications:** Vision tasks—classification, detection, segmentation (winning ImageNet 2014; robust in mobile/efficient settings)
- **Further reading:** Modular architectures, adaptive computation, parameter-efficient designs

---

Summary Table: Inception Module Branches Example

Branch	Filter Size	Output Channels	Output Volume	Bottleneck Used?
1x1 Conv	1x1	64	28×28×64	No
3x3 Conv	3x3	128	28×28×128	Yes
5x5 Conv	5x5	32	28×28×32	Yes
MaxPool + 1x1	Pool/1x1	32	28×28×32	Yes

Final concatenated output: 28×28×256

Exam/Research Note

- **Always mention:** Bottleneck layers (1x1 conv) are critical for practical deep networks.
- **Understand formula:** Multiplication counts drive efficient architecture design.
- **Highlight impact:** Inception led to huge advances in scalable, efficient deep learning.

Summary formulas to remember

- **Output depth after concatenation:**

$$D_{final} = D_{1x1} + D_{3x3} + D_{5x5} + D_{pool}$$

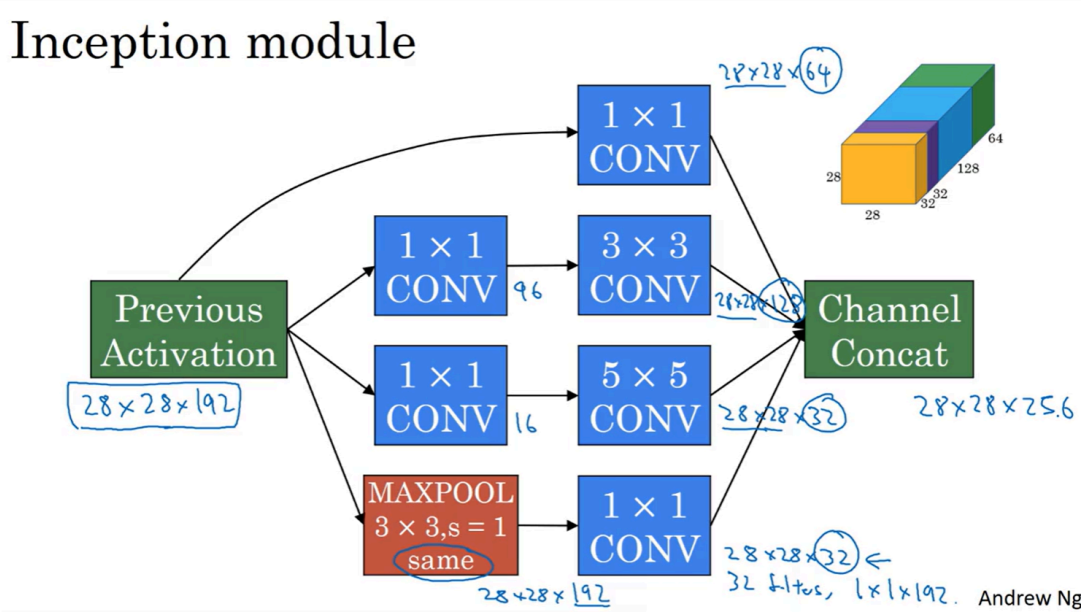
- **Cost per convolution:**

$$\text{Mults} = H \times W \times D_{out} \times k \times k \times D_{in}$$

Clarity Tip: Always track how shapes change after each block and the motivation for parallel multi-scale convolution.

Detailed Inception Network

Inception Network (GoogleNet)



1. Inception Module:

- **Input:** Typically receives an activation tensor from a previous layer, e.g., 28×28×192.
- **Parallel Paths:**
  - **1x1 Convolution:**
    - Reduces depth (number of channels) and computational cost.
    - Example: 16 filters, output shape 28×28×16
  - **1x1 followed by 5x5 Convolution:**
    - 1x1 conv reduces depth, then 5x5 conv extracts features.
    - Example: 1x1 with 16 filters, then 5x5 with 32 filters, output 28×28×32.
  - **1x1 followed by 3x3 Convolution:**
    - 1x1 conv reduces depth, then 3x3 conv extracts features.
    - Example: 1x1 with 16 filters, then 3x3 with 128 filters, output 28×28×128.
  - **1x1 Convolution (standalone):**
    - Directly applies 1x1 conv, e.g., 64 filters, output 28×28×64.
  - **Pooling Path:**
    - 3x3 max pooling with 'same' padding (output 28×28).
    - Followed by 1x1 conv to reduce depth, e.g., 32 filters, output 28×28×32.

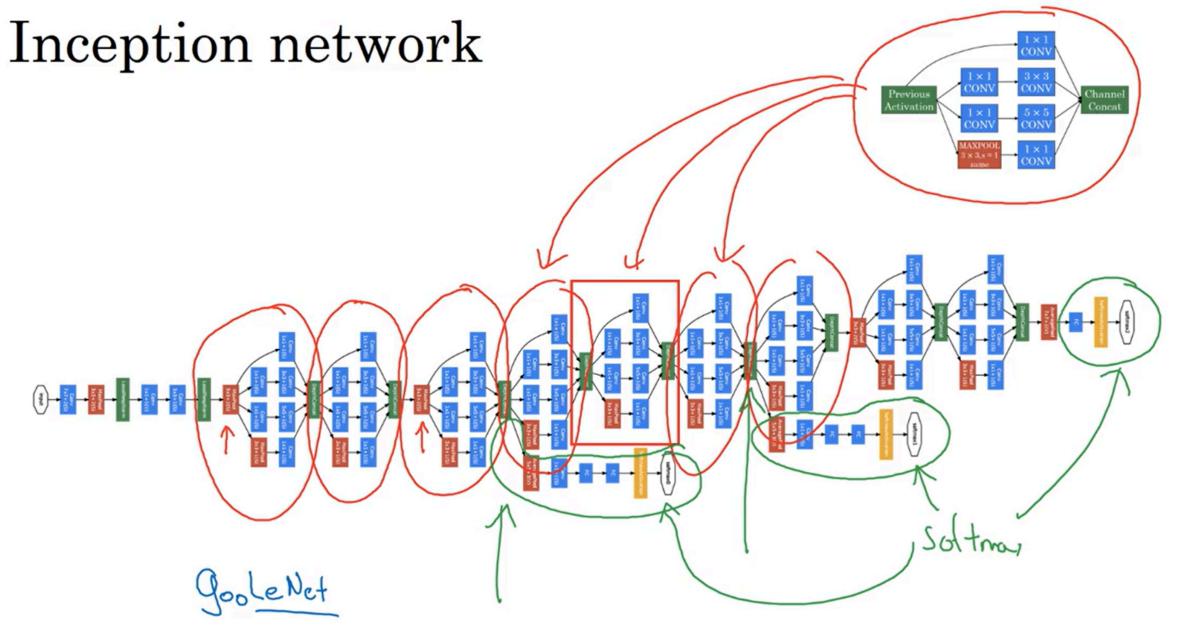
Why 1x1 Convolutions?

- **Dimensionality Reduction:**
  - Reduces the number of input channels before expensive convolutions (like 3x3, 5x5).
  - **Computation for** $5 \times 5 \times D_{in} \times D_{out} \times H \times W$
    - If you first reduce D_{in} with a 1x1 conv, the cost drops significantly.

2. Channel Concatenation

- **Purpose:** Combine outputs from all parallel paths along the channel axis.
- **Example Calculation:**
  - 1x1 conv: 64 channels
  - 3x3 conv: 128 channels
  - 5x5 conv: 32 channels
  - Pooling + 1x1 conv: 32 channels
  - **Total output:** 28×28×(64+128+32+32)=28×28×256
- **Operation:** Concatenate all outputs along the depth (channel) dimension.

3. Stacking Inception Modules



[Szegedy et al., 2014, Going Deeper with Convolutions]

Andrew Ng

- **Inception Network:** Built by stacking multiple inception modules.
- **Network Structure:**
  - Repeated inception blocks, sometimes separated by max pooling layers to reduce spatial dimensions.
  - Each block is essentially the same inception module, possibly with minor variations.

4. Auxiliary Classifiers (Side Branches)

- **Purpose:**
  - Added to intermediate layers to help with gradient flow and regularization.
  - Each side branch consists of a few layers ending in a softmax output.
  - These branches encourage intermediate layers to learn useful features and help prevent overfitting.
- **Final Output:**
  - The main branch ends with a fully connected layer and softmax for the final prediction.
  - Side branches also output predictions, but are mainly used during training.

5. Naming and Versions

- **Original Name:** GoogleNet (named in homage to LeNet).

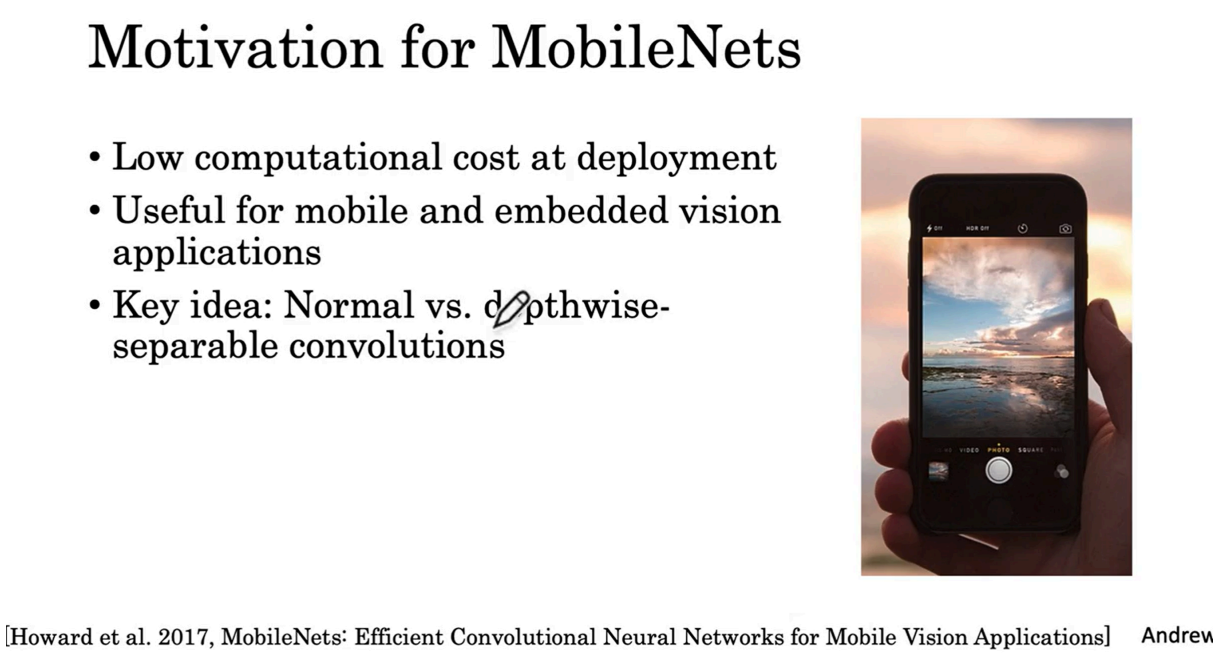
- **Inspiration:** The name "Inception" comes from the meme "we need to go deeper," which is even cited in the original paper.
- **Later Versions:**
  - Inception v2, v3, v4, and Inception-ResNet (combining inception modules with residual connections).
  - All versions build on the core idea of parallel convolutions and stacking inception modules.

6. Key Takeaways

- **Inception modules** allow the network to capture features at multiple scales in parallel.
- **1x1 convolutions** are crucial for reducing computation and enabling deeper networks.
- **Auxiliary classifiers** help with training deep networks by improving gradient flow and regularization.
- **Inception networks** are built by stacking these modules, sometimes with pooling layers in between to reduce spatial size.
- **Later versions** (v2, v3, v4, Inception-ResNet) refine and extend these ideas, but the core concept remains the same.

MobileNet

1. What is MobileNet?



- MobileNet is a special type of convolutional neural network (CNN) designed for devices with less computing power, like mobile phones.
- It is used for computer vision tasks (like image classification, object detection) where speed and efficiency are important.
- Main advantage: **Much faster and lighter** than traditional CNNs (like ResNet, Inception).

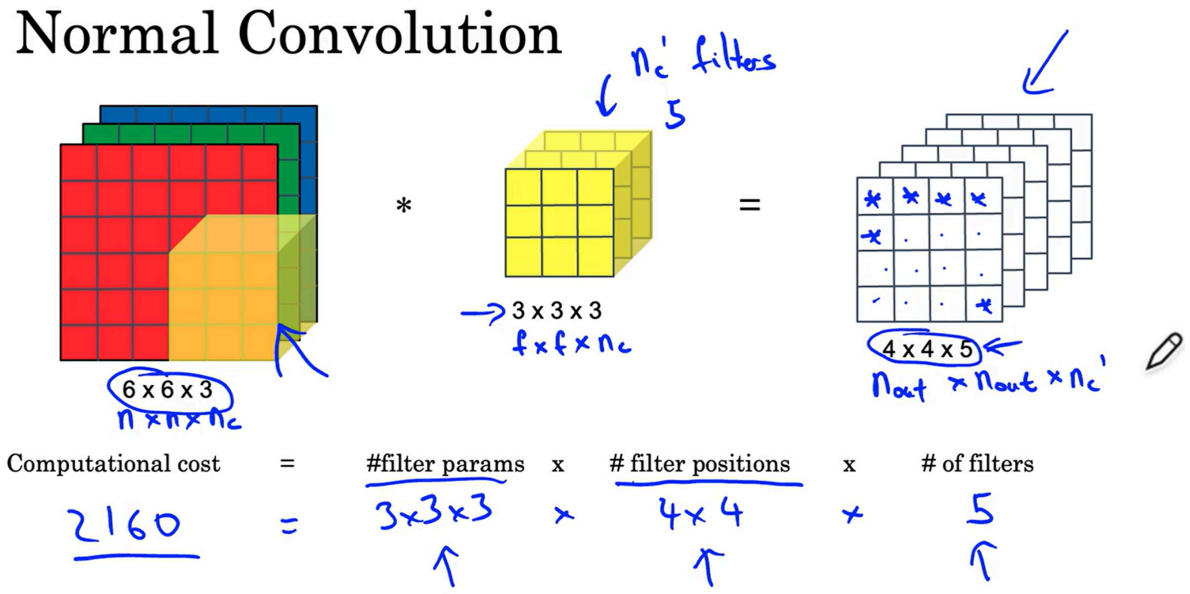
2. Why do we need MobileNet?

- Standard CNNs are heavy and need powerful CPUs/GPUs.
- If you want to run your model on a mobile or embedded device, you need a lighter model.
- MobileNet solves this by reducing the number of computations required.

3. Key Concept: Depthwise Separable Convolution

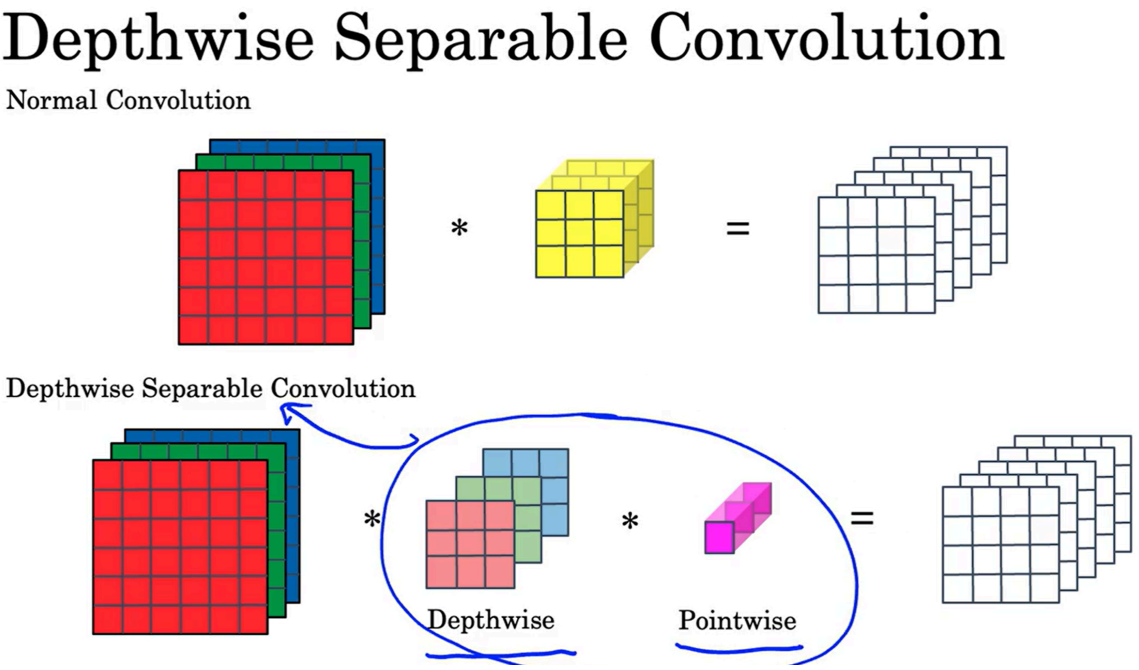
- MobileNet uses a new type of convolution called **depthwise separable convolution** instead of normal convolution.
- This operation is split into two steps:
    1. **Depthwise Convolution**
    2. **Pointwise Convolution**

A. Normal Convolution (for comparison)

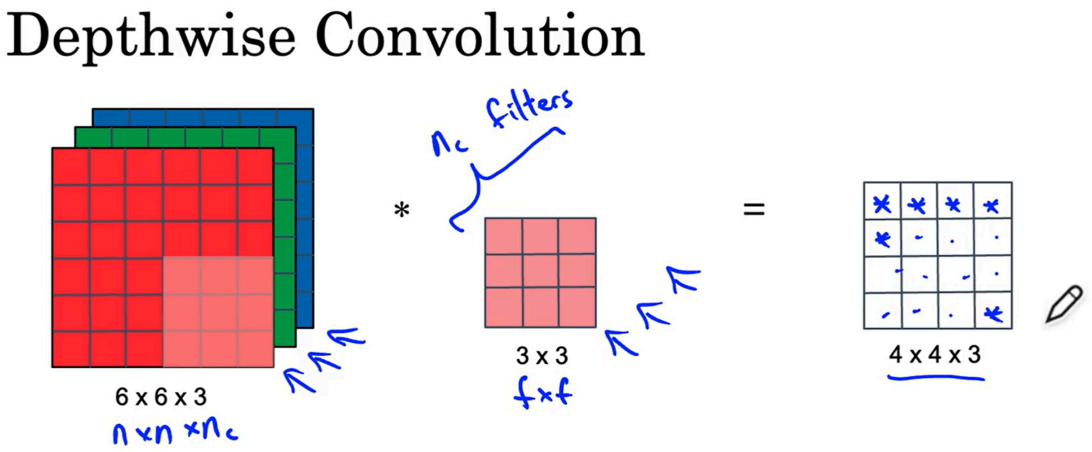


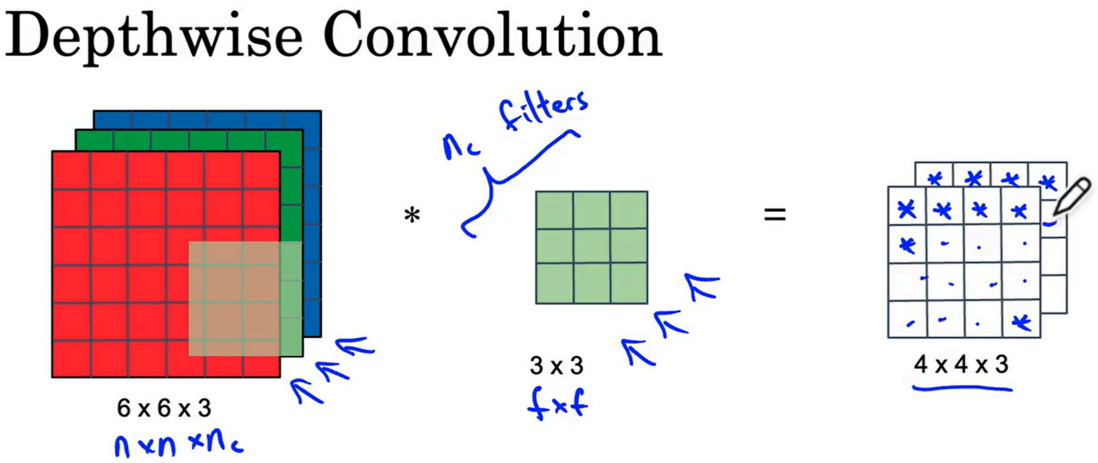
- Input: e.g., 6x6 image with 3 channels (RGB).
- Filter: 3x3x3 (height x width x channels).
- For each position, you do 27 multiplications (3x3x3).
- If you use 5 filters, output is 4x4x5.
- **Total multiplications:** 2160 (for this example).

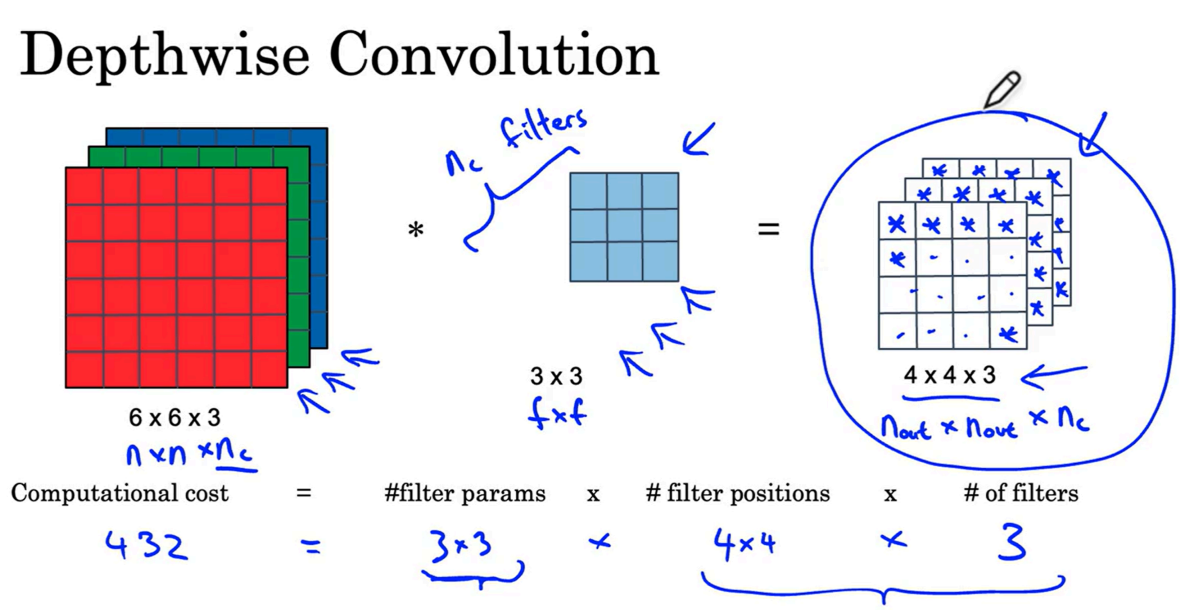
B. Depthwise Separable Convolution (MobileNet style)



Step 1: Depthwise Convolution

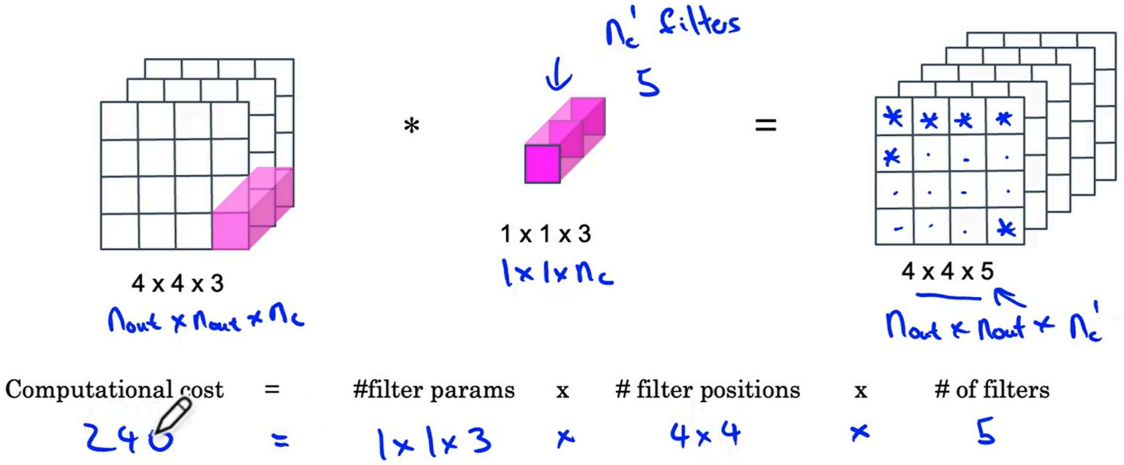






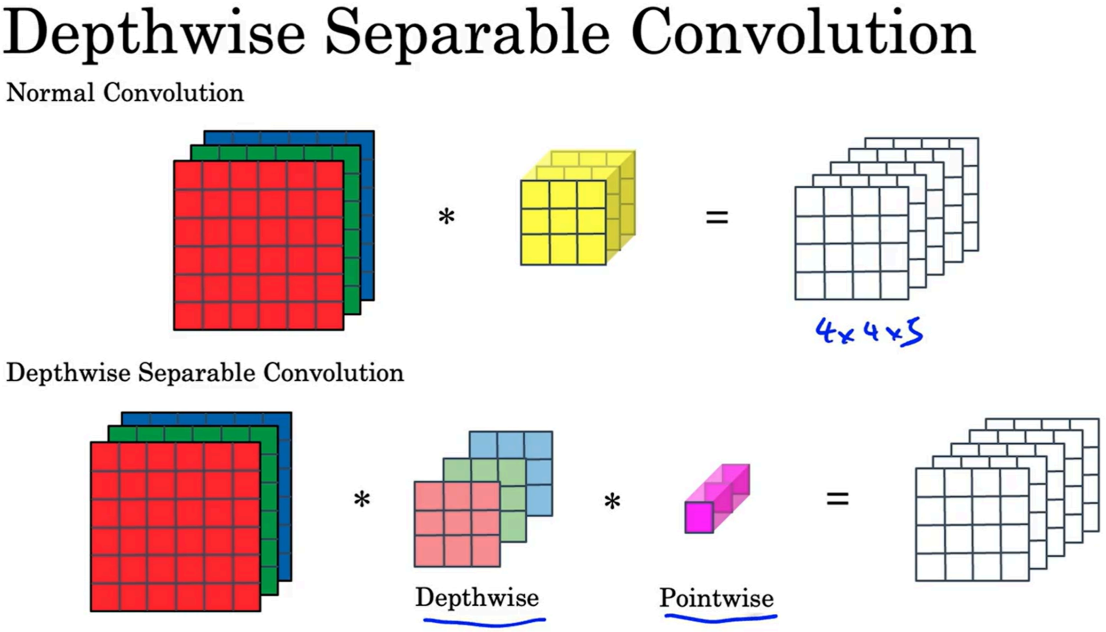
- Apply one filter per input channel (e.g., one for red, one for green, one for blue).
- Each filter is 3x3 (not 3x3x3).
- Output: 4x4x3 (for 3 channels).
- **Total multiplications:** 432

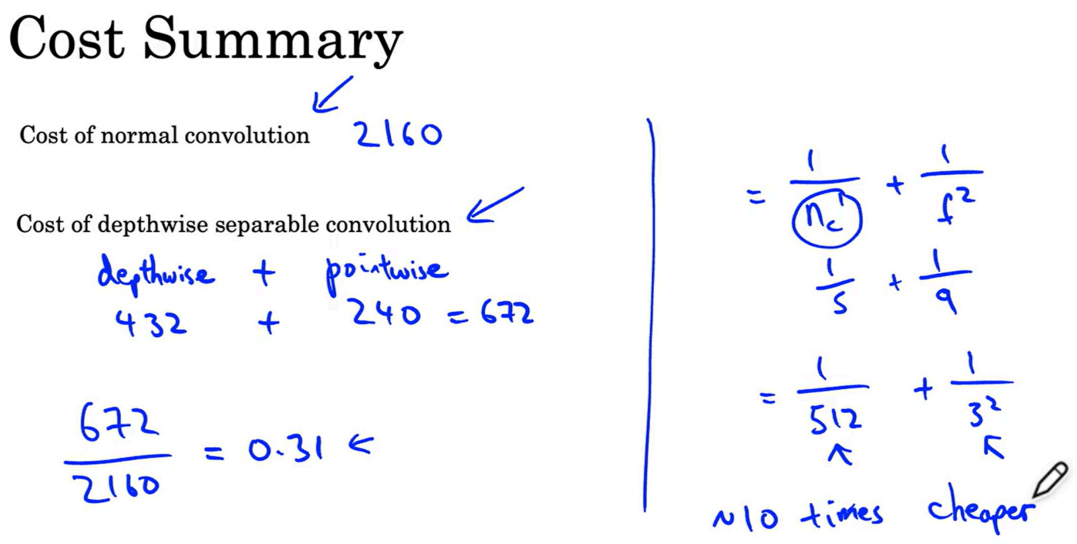
Step 2: Pointwise Convolution



- Use 1x1x3 filters to mix information across channels.
- For 5 filters, output is 4x4x5.
- **Total multiplications:** 240

Total for MobileNet style:





[Howard et al. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications] Andrew N

- 432 (depthwise) + 240 (pointwise) = **672** multiplications
- Compared to normal convolution: 672 vs 2160 (about **31%** of the computation)

4. General Formula for Computation Savings

- Ratio of computation (MobileNet vs normal): $\text{Ratio} = \frac{1}{n_c'} + \frac{1}{f^2}$
  - Where n_c' = number of output channels, f = filter size
- For large n_c', MobileNet can be **9–10 times cheaper** than normal convolution.

5. Visual Representation

- Diagrams often show 3 channels for simplicity, but MobileNet works for any number of channels.
- Depthwise convolution: separate filter for each channel.
- Pointwise convolution: 1x1x(number of channels) filter.

6. Summary Table: Computation Comparison

Operation	Multiplications (Example)
Normal Convolution	2160
Depthwise Separable Conv	672

- **MobileNet saves a lot of computation, making it ideal for mobile and edge devices.**

7. Key Takeaways

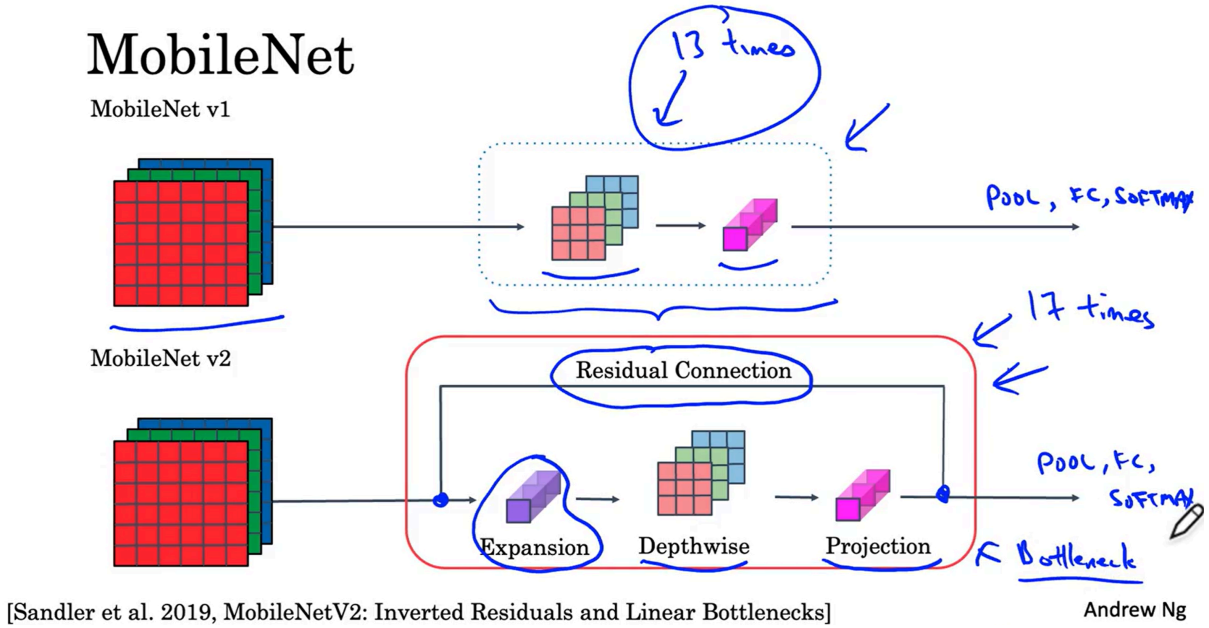
- **Depthwise separable convolution** is the main building block of MobileNet.
- It splits the normal convolution into two simpler steps, reducing computation and model size.
- MobileNet is perfect for real-time applications on devices with limited resources.

8. Extra Tips for Indian Students

- If you are preparing for interviews or GATE, remember the formula for computation savings.
- Practice drawing the two-step process (depthwise + pointwise) for better understanding.
- If you get a question on "how to make CNNs efficient for mobile devices," always mention MobileNet and depthwise separable convolution.

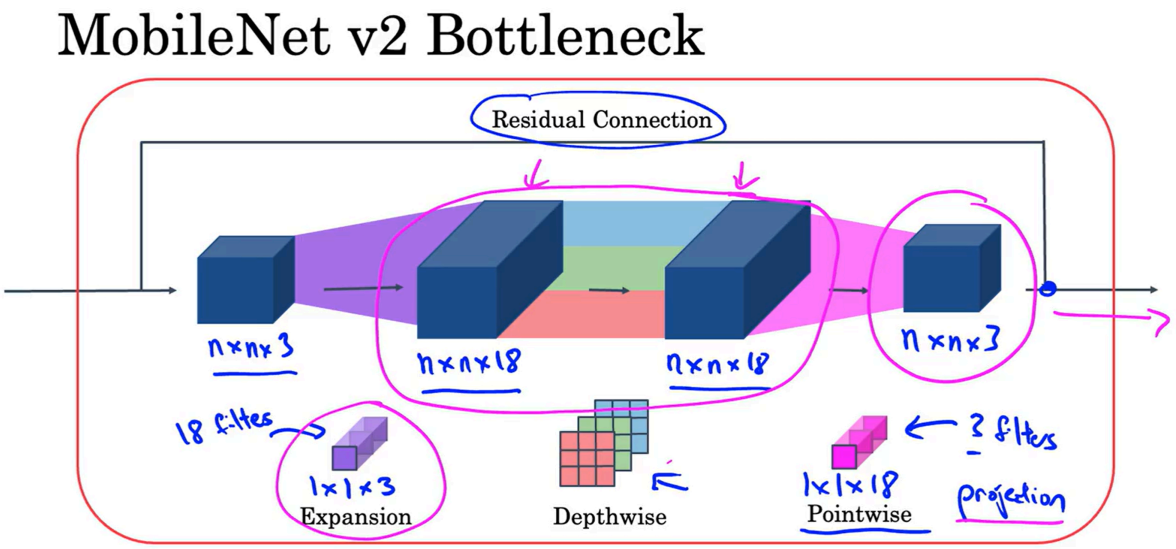
MobileNet Architecture

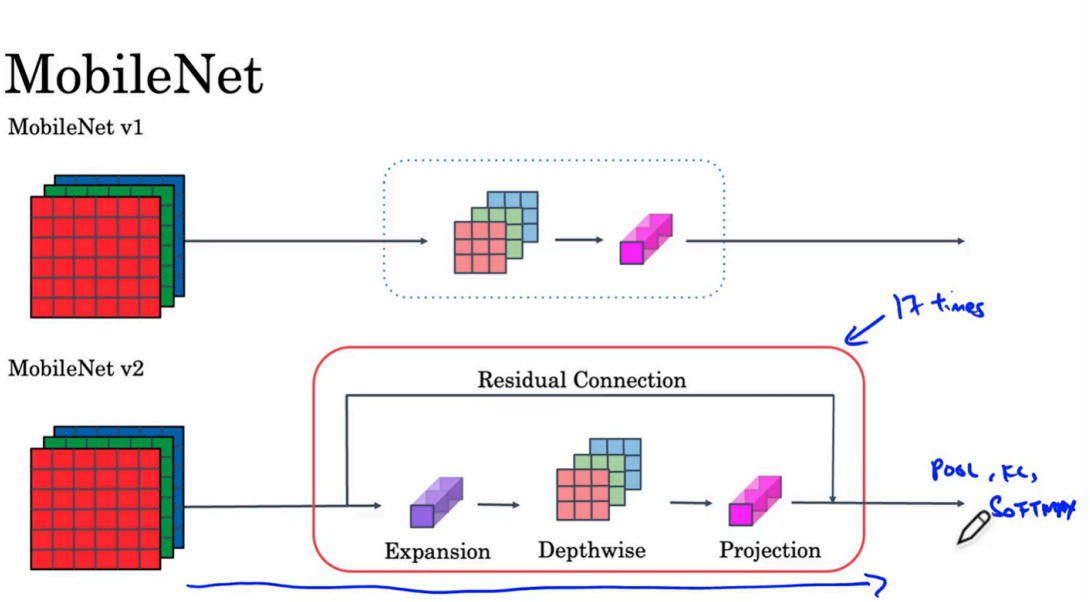
1. MobileNet v1: Main Points



- **Purpose:** MobileNet is designed for devices with less computing power (like mobiles, edge devices).
- **Key Idea:** Replace standard convolution with **depthwise separable convolution** everywhere in the network.
    - This operation splits normal convolution into two steps:
        1. **Depthwise convolution:** Applies a single filter per input channel.
        2. **Pointwise convolution:** Uses 1x1 convolutions to mix information across channels.
- **Architecture:**
    - The main block (depthwise + pointwise) is repeated **13 times**.
    - After these, the network has:
        - A **Pooling layer** (to reduce spatial size)
        - A **Fully Connected layer**
        - A **Softmax layer** (for classification output)
- **Advantage:**
    - Much less computationally expensive than older CNNs using normal convolution.
    - Good accuracy with much lower resource usage.

2. MobileNet v2: Improvements Over v1





- **Two Main Changes:**
  1. **Residual (Skip) Connections:**
     - Like in ResNet, input is added directly to the output of the block.
     - Helps gradients flow better during training, making deep networks easier to train.
  2. **Expansion Layer:**
     - Before depthwise convolution, a 1x1 convolution expands the number of channels (usually by a factor of 6).
     - This allows the network to learn richer features.
     - After expansion and depthwise convolution, a 1x1 pointwise convolution (called **projection**) reduces the channels back down (bottleneck).
- **Block Structure (Bottleneck Block):**
  - **Input:** n x n x c (e.g., n x n x 3 for RGB)
  - **Expansion:** 1x1 conv, increases channels to n x n x (6c)
  - **Depthwise Convolution:** 3x3 conv, keeps channels at n x n x (6c)
  - **Projection:** 1x1 conv, reduces channels back to n x n x c
  - **Residual Connection:** Input is added to output if dimensions match

- **Architecture:**
  - This bottleneck block is repeated **17 times** in MobileNet v2.
  - Final layers: pooling, fully connected, softmax (for classification)
- **Why Bottleneck Block?**
  - **Expansion:** Lets the network learn more complex functions (richer features)
  - **Projection:** Reduces memory and computation for the next block
  - **Residual connection:** Makes training deeper networks easier
- **Result:**
  - MobileNet v2 is more accurate and still efficient for mobile/edge devices.

3. Summary Table: v1 vs v2

Feature	MobileNet v1	MobileNet v2
Main Block	Depthwise + Pointwise Conv	Expansion + Depthwise + Projection + Residual
Block Repeats	13	17
Residual Connections	No	Yes
Expansion Layer	No	Yes (factor ~6)
Efficiency	High	High
Accuracy	Good	Better

4. Key Takeaways for Indian Students

- **Depthwise separable convolution** is the main trick for efficiency in both v1 and v2.
- **MobileNet v2** adds expansion and residual connections for better accuracy and easier training.
- **Bottleneck block**: expand (more features), depthwise (efficient spatial filtering), project (reduce size), skip connection (easy training).
- **Use case:** If you want to deploy deep learning models on mobile or edge devices, MobileNet is a top choice.

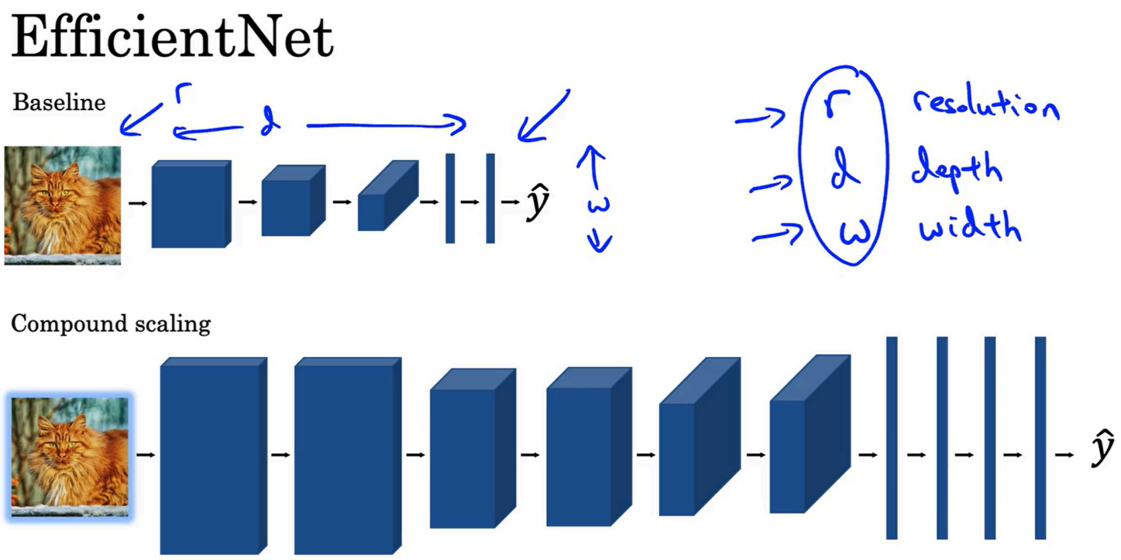
5. Extra Tips

- For interviews or GATE, remember the block structure and why each step is used.

- If you get a question on efficient CNNs, always mention depthwise separable convolution and bottleneck blocks.
- For more details (like exact layer sizes), refer to the original MobileNet v2 paper by Mark Sandler et al.

EfficientNet

1. Why EfficientNet?



- Earlier models like MobileNet V1 and V2 help you build neural networks that are computationally efficient.
- But different devices (mobiles, edge devices) have different levels of computational power.
- Sometimes you want a bigger network for more accuracy, sometimes a smaller one for faster speed.
- **EfficientNet** gives a systematic way to scale your neural network up or down for any device.

2. Three Ways to Scale a Neural Network

- You can change three main things:

    - **Resolution (r):** Size of the input image (higher resolution can improve accuracy but needs more computation).
    - **Depth (d):** Number of layers in the network (deeper networks can learn more complex features).
    - **Width (w):** Number of channels in each layer (wider layers can capture more information).
- The challenge: How much should you increase or decrease each of these for your device?

3. Compound Scaling – EfficientNet's Main Idea

- Instead of changing only one factor (like just depth or just resolution), EfficientNet uses **compound scaling**.
- This means you scale up or down all three (r, d, w) together, in a balanced way.
- The authors (Mingxing Tan and Quoc Le) found that scaling all three together gives better accuracy for the same computational cost.
- Example: Instead of just doubling the depth, you might increase resolution by 10%, depth by 50%, and width by 20%—all at once.

4. How to Choose the Best Scaling?

- EfficientNet provides a formula and method to find the best trade-off between resolution, depth, and width for your computational budget.
- You don't have to guess—EfficientNet helps you select the right combination for your device.
- There are open-source implementations available, so you can easily adapt EfficientNet for your own project.

5. Summary Table: Scaling Factors

Factor	What it Means	Effect on Model
Resolution	Input image size (r)	Higher = more details, more computation
Depth	Number of layers (d)	Higher = deeper features, more computation
Width	Channels per layer (w)	Higher = more info per layer, more computation

6. Key Takeaways for Indian Students

- EfficientNet is useful when you want to deploy neural networks on devices with different resources.
- Compound scaling (r, d, w together) gives better results than scaling only one factor.
- For interviews or GATE, remember: EfficientNet = balanced scaling of resolution, depth, and width.
- If you want to build models for mobiles or embedded devices, EfficientNet is a top choice.

7. Extra Tips

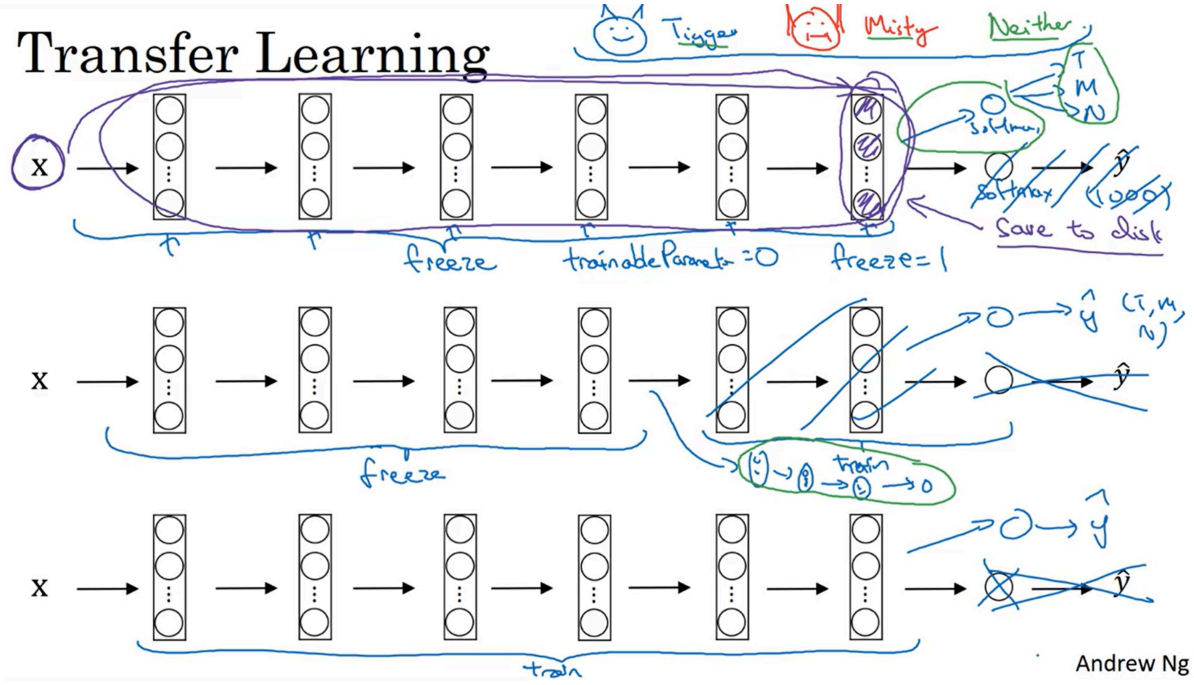
- If you are adapting a neural network for a specific device, check EfficientNet's open-source code for best practices.
- EfficientNet helps you get the best accuracy possible within your computational limits.

Using Open-Source Implementation

- Many deep learning models are difficult to reproduce exactly from just reading research papers, even for experienced researchers.
- Most deep learning researchers now open-source their code (often on GitHub), making it easier for others to use and build upon their work.
- If you want to use a model from a research paper, first check online for an open-source implementation instead of re-coding from scratch.
- Using the original author's code helps you get started much faster and ensures you are using a correct, well-tested version.
- Downloading code from GitHub is straightforward: search for the model (e.g., ResNet), find a suitable repository, and use `git clone <URL>` to download it.
- Always check the license (e.g., MIT License) to understand how you can use or modify the code.
- Open-source repositories often include configuration files (like prototxt for Caffe) that specify the model architecture in detail.
- If you prefer a different framework (like TensorFlow or PyTorch), you can often find multiple implementations of the same model.

- A common workflow: pick a model architecture, find an open-source implementation, download it, and start building your project from there.
- Pretrained models are often available, which saves time and resources since you can use transfer learning instead of training from scratch.
- If you develop your own implementation, consider contributing it back to the open-source community to help others.
- Starting with open-source code is usually the fastest and most reliable way to begin a new deep learning project.

Transfer Learning



- In computer vision, **transfer learning** helps achieve faster progress compared to training from random initialization.
- Instead of training from scratch, you can **download pre-trained weights** trained by others (often on large public datasets like ImageNet, MS COCO, Pascal VOC) and use their network architecture as a starting point.
- Pre-training typically involves weeks of high-performance search with GPUs; by downloading these weights, you benefit from knowledge acquired over large and diverse

datasets.

- **Typical transfer learning workflow:**
  - Download a network pre-trained on a public dataset (e.g., ImageNet, which outputs one of 1,000 classes via softmax).
  - Remove the existing softmax layer and replace it with a new one suitable for your own problem (e.g., your own three-class softmax for cat detector: Tigger, Misty, Neither).
  - **Freeze the parameters** in all previous layers except your new output layer, and train only the final layer's weights on your data.
  - Most deep learning frameworks let you specify which layers to freeze (e.g., using flags like `trainable=False` or `freeze=True`).
- If your **training set is small**, this approach leverages feature extraction learned from large datasets and works very well.
- **Optimization trick:** Since earlier layers are frozen, you can **pre-compute activations/features for all training images**, save them, and directly train a shallow classifier on top—this saves computation time.
- If you have **more labeled data**, you can freeze fewer layers and train more layers on your custom dataset.
  - You may use pre-trained weights for initialization of certain layers, then train a few or all layers (except the output).
  - You can also discard and substitute last few layers with your own hidden units and outputs.
- **Rule of thumb:**
  - More data = train more layers, less freezing.
  - Extreme case (a lot of data): Use pre-trained weights as initialization only and train the full network using your custom data.
- **Conclusion:** In computer vision, unless you have a huge dataset yourself, **transfer learning is almost always advisable** rather than training everything from scratch.

Data Augmentation

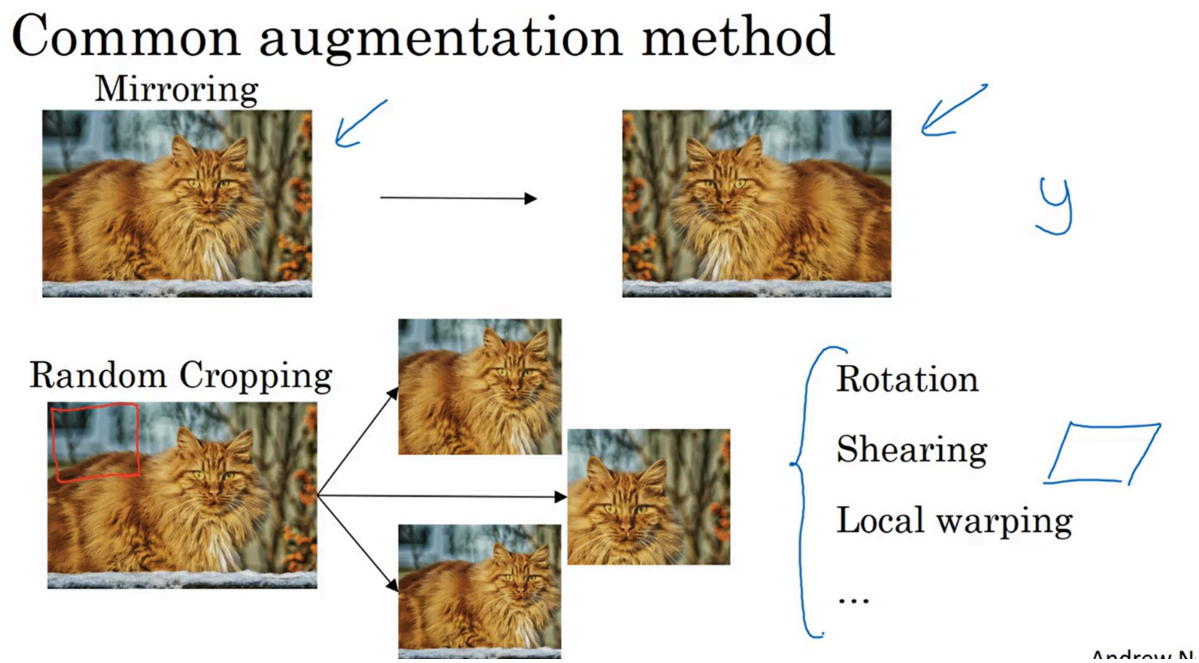
What is Data Augmentation?

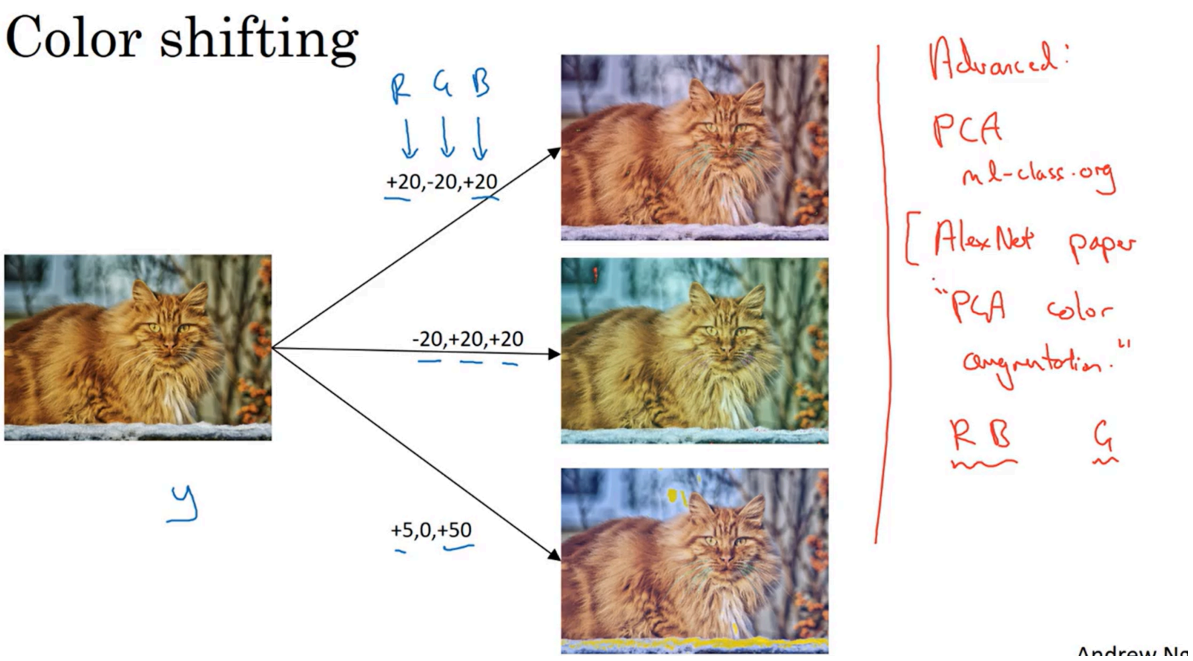
- Data augmentation is a set of techniques used to **artificially increase the amount and diversity of data** available for training computer vision models.
- **Key reason:** In computer vision, "more data = better models," but collecting more real data is often hard or expensive.

Why is it Needed?

- Computer vision deals with images, which have lots of pixels and complexity.
- Having **more training data** helps model learn better patterns and perform better, just like a student who practices more problems learns better.
- In many machine learning areas, you might have enough data. For **computer vision, you almost always feel you don't have enough.** So, augmentation helps fill the gap.

Common Data Augmentation Techniques





- These are methods to create "new" images from existing ones by modifying them.

1. Mirroring/Flipping

- **Description:** Flip the image horizontally (like looking in a mirror).
- **Example:** If you have a photo of a cat facing left, flipping it makes it face right.
- **Real-life use:** Both images still represent a cat! For tasks where the orientation doesn't matter (cat vs. not cat), this is very useful.

2. Random Cropping

- **Description:** Randomly select a smaller region of the image (a "crop") to use as a new sample.
- **Example:** From a full photo of a cat, take just the upper right or lower left part showing just the cat's face or paws.
- **Real-life use:** Helps the model focus on different parts, improving robustness when images are off-center or zoomed.
- **Note:** Crops should be large enough to still contain important features.

3. Rotation, Shearing, and Local Warping

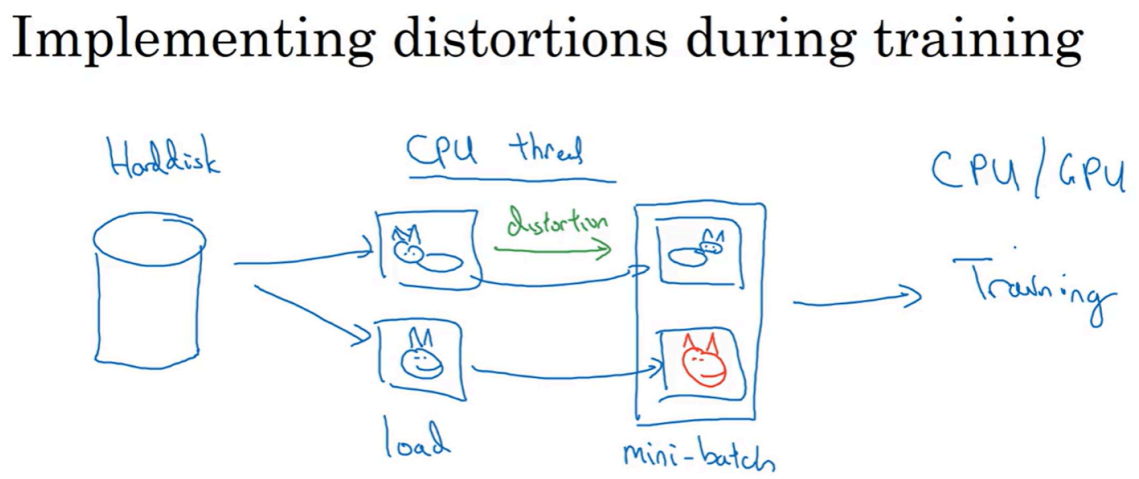
- **Description:** Slightly rotate, "shear" (slant), or locally distort images.

- **Example:** Rotating a traffic sign image a few degrees mimics the real-world situation where the camera might not be perfectly aligned.
- **Real-life use:** Makes models more robust to different angles or camera positions.
- **Note:** Used less frequently due to complexity, but can be very effective in some tasks.

4. Color Shifting/Distortion

- **Description:** Randomly change the color channels (R, G, B) a little.
- **Example:** Add more red and blue, remove some green—picture becomes more purple.
- **Real-life use:** Images in the real world might have different lighting (sunny, cloudy, yellowish indoor light). The model should recognize cats regardless of lighting/colors.
- **How:** Adjust colors by a random amount for each channel. In practice, these changes are usually small.
- **Advanced:** Some methods, like PCA color augmentation, use algorithms to adjust channels more intelligently (explained in the AlexNet paper).

How Data Augmentation is Used in Practice



- **Implementation:** When training, a thread/process loads images from disk, applies augmentation (e.g., crops, flips, color changes), then passes them to the model for training.
- **Parallelism:** Augmentation and data loading can happen in parallel to training (can be on CPU while training is on GPU).

- **Open-source:** Many frameworks and open-source libraries (like TensorFlow, PyTorch) have ready-made functions for augmentation.

Hyperparameters in Data Augmentation

- You can adjust “how much” you augment (e.g., max angle to rotate, range for color change). These choices are **hyperparameters**.
- A good starting point is to use the defaults from a well-tested open-source implementation.
- Customizing them can help if your data or problem is special.

Summary and Tips

- Data augmentation **makes your model more robust and helps it learn more general patterns**.
- It’s widely used in both training from scratch and fine-tuning pre-trained models (transfer learning).
- **Bottom line:** If your vision model isn’t performing well and you don’t have enough data, always try data augmentation!

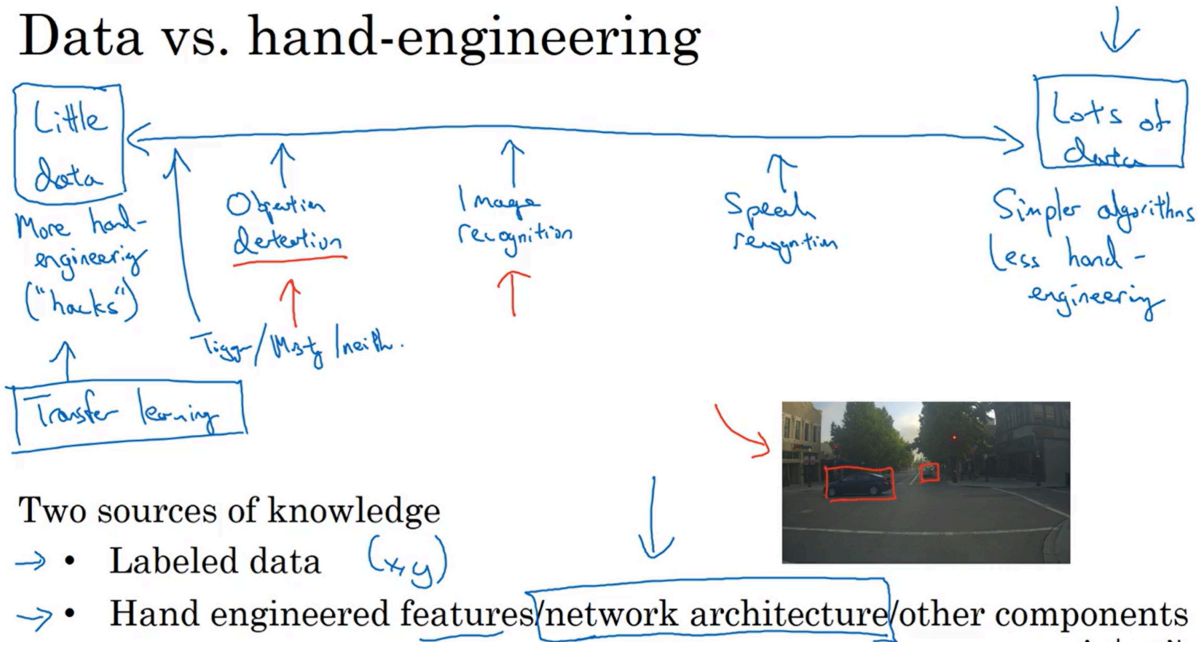
Real-world Example

- Imagine you're building a smartphone app to recognize faces. You have 1,000 photos, but they're all taken at eye level, indoors. Users will take photos in all sorts of conditions—sideways, outdoors, under yellow light.
- By augmenting your photos (flipping, cropping, color changes), you can make your model more likely to succeed in all these new, unseen situations.

If you want even more technical detail, explore PCA color augmentation in the original AlexNet paper or open-source code provided by popular ML libraries!

State of Computer Vision

1. Deep Learning's Impact on Computer Vision



- **Deep learning** has revolutionized computer vision, but the field is still unique: even with large datasets (millions of images), the complexity means we often *wish for more data*.
- **Object detection** (finding and labeling objects in images) has even less data than image recognition, because labeling bounding boxes is expensive and time-consuming.

2. Data Regimes: Lots vs. Little Data

- **Machine learning problems** fall on a spectrum from *little data* to *lots of data*.
- With **lots of data**:
  - Simpler algorithms and less hand-engineering are needed.
  - Large neural networks can learn directly from data.
- With **less data**:
  - More hand-engineering (feature design, architecture tweaks) is required.
  - Sometimes, "hacks" are necessary, but these are often the best way to get good performance in low-data regimes.

3. Sources of Knowledge in ML Systems

- Two main sources:
  1. **Labeled data** (supervised learning: (x,y) pairs)
  2. **Hand-engineering** (feature design, architecture, system components)

- When data is scarce, hand-engineering becomes more important.

4. Why Computer Vision Relies on Hand-Engineering

- Computer vision tasks are complex and often lack enough data.
- Historically, this led to:
  - Heavy reliance on hand-engineered features and architectures.
  - Development of complex network architectures.
- Hand-engineering is a *skillful* and *valuable* contribution, especially when data is limited.

5. Recent Trends

- **Data availability** for computer vision has increased, reducing (but not eliminating) the need for hand-engineering.
- Still, network architectures and hyperparameter choices in CV are often more complex than in other fields.
- **Object detection** tasks, with even less data, require even more specialized architectures.

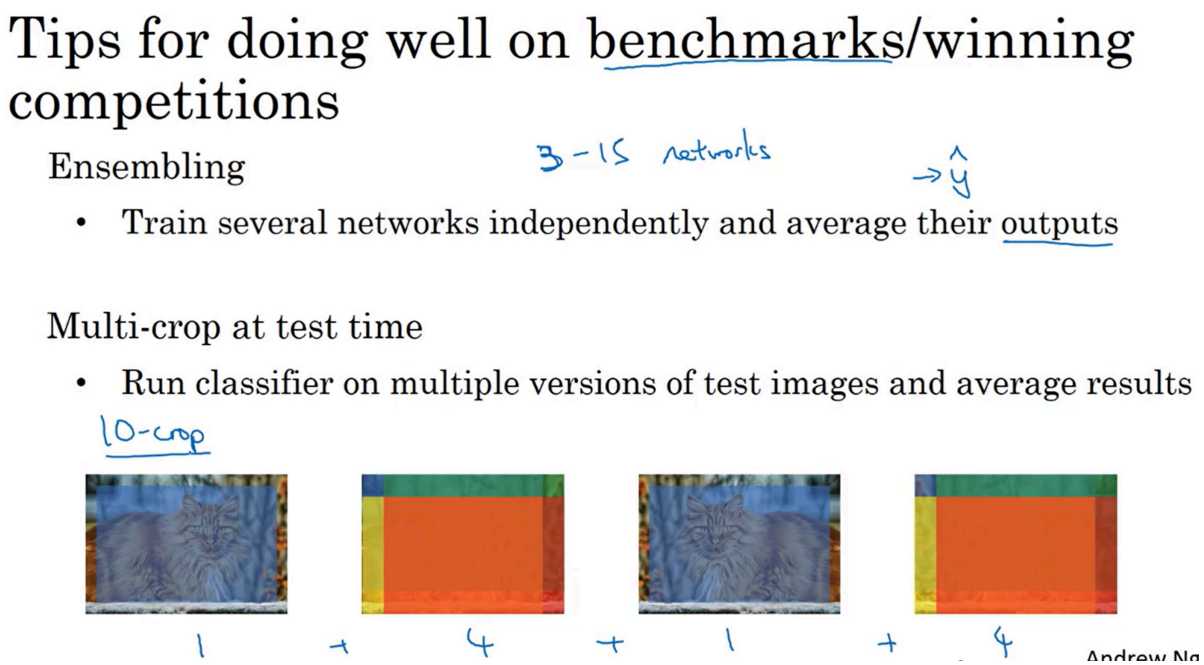
6. Transfer Learning

- **Transfer learning** is crucial when data is limited.
- Pretrained models (on large datasets) can be fine-tuned for new tasks with less data.

7. Benchmarking and Competitions

- The CV community is highly focused on standardized benchmarks and competitions.
- Doing well on benchmarks helps get papers published and advances the field.
- However, some techniques used to win benchmarks are not practical for production systems.

8. Tips for Doing Well on Benchmarks (But Not Always for Production)



- **Ensembling:**
  - Train multiple networks independently and average their outputs $(\hat{y})$.
  - Can improve accuracy by 1–2%.
  - Not practical for production due to high computational cost (slower inference, more memory).
- **Multi-crop at test time:**
  - Apply data augmentation to test images (e.g., 10-crop: center + 4 corners + their mirrors).
  - Run all crops through the network and average results.
  - Improves benchmark scores, but slows down inference.

9. Practical System Building

- **Production systems** usually avoid ensembling and multi-crop due to efficiency concerns.
- **Transfer learning** and using open-source implementations are practical:
  - Start with someone else's architecture and pretrained weights.
  - Fine-tune on your dataset for faster, more reliable results.
  - Open-source code often includes well-tuned hyperparameters.

- If you want to invent new algorithms, you may need to train from scratch.

10. Summary

- Computer vision is challenging due to data limitations and task complexity.
- Hand-engineering is essential when data is scarce, but less so as data grows.
- Transfer learning and open-source models are practical tools.
- Some benchmark-winning techniques are not suitable for production.

Quick Check:

- Can you explain why hand-engineering is more common in computer vision than in NLP?
- What are the trade-offs of using ensembling or multi-crop at test time?
- How would you approach a new CV task with only a small labeled dataset?