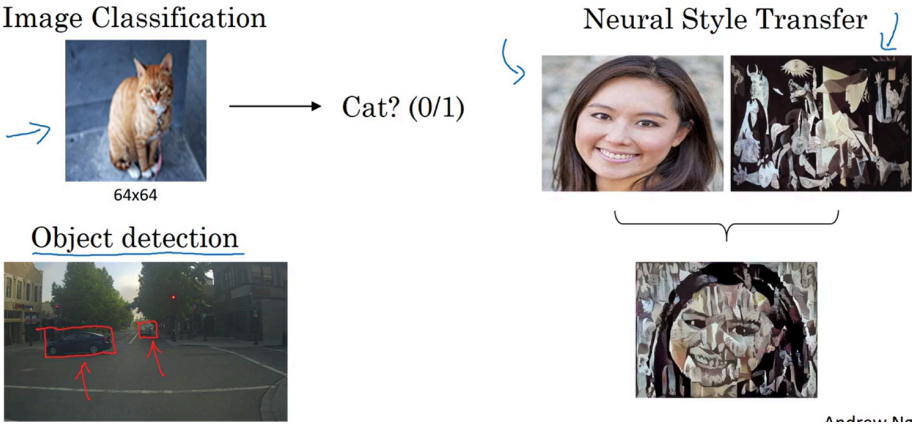
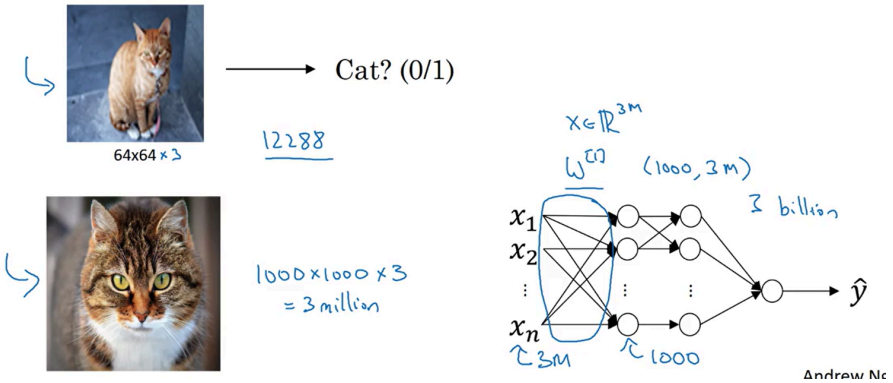
WEEK 1: Foundations of Convolutional Neural Networks

Computer Vision

Computer Vision Problems



Challenges of Computer vision



Core Concept:

Computer vision aims to enable computers to interpret and understand visual information from the world, like images and videos, using deep learning techniques. It's rapidly advancing thanks to neural networks, powering applications from self-driving cars to face recognition.

Breakdown:

How It Works:

- Computer vision employs **deep learning**, especially Convolutional Neural Networks (CNNs), to process images.
- Fundamental steps:
  - **Image Classification:** Determine what is present in an image (e.g., cat or dog).
  - **Object Detection:** Locate and categorize objects within an image, drawing bounding boxes around them.
  - **Creative Tasks:** Like neural style transfer—transforming an image's style to mimic artwork.

- Processing images with standard neural networks involves transforming each pixel (e.g., a 64x64 RGB image has 12,288 features), which may lead to millions of parameters as image size grows—posing challenges of computational efficiency and overfitting.
- Convolution operations (explained further in the next lecture) are used to make it feasible to process large images by greatly reducing the number of trainable parameters.

Key Formulas:

- For a color image:

  Input features = Image width × Image height × 3 (RGB channels)

  Example: $64 \times 64 \times 3 = 12,288; \quad 1000 \times 1000 \times 3 = 3,000,000$

- Fully Connected Layer:

  Number of weights = (Number of hidden units) × (Number of input features)

  Example: For 1,000 hidden units and 3 million input features:

  $1,000 \times 3,000,000 = 3,000,000,000$

Explanation:

- Each variable represents the size/dimensionality of the data and parameters—illustrating how quickly complexity grows with image size, justifying why convolutions are required.

Why It's Used:

- **Problem solved:** Enables computers to "see" and interpret images at scale, handling vast amounts of raw data efficiently and accurately.
- **Advantages:**
  - Detects complex patterns automatically (feature extraction).
  - Reduces the need for handcrafted features.
  - Enables new applications (e.g., style transfer, autonomous driving).
  - Handles high-dimensional data more tractably using convolutions.

Practical Insights:

Real-World Applications:

- **Self-driving cars:** Identifying vehicles, pedestrians, and obstacles to navigate safely.
- **Face recognition:** Unlocking phones or securing doors using your face as the key.
- **Content curation:** Curating images for apps, sorting by attractiveness, style, or relevance.

Common Pitfalls:

- **Overfitting:** With too many parameters and not enough data, models memorize instead of generalize.

  Prevention: Use techniques like data augmentation, dropout, or regularization.

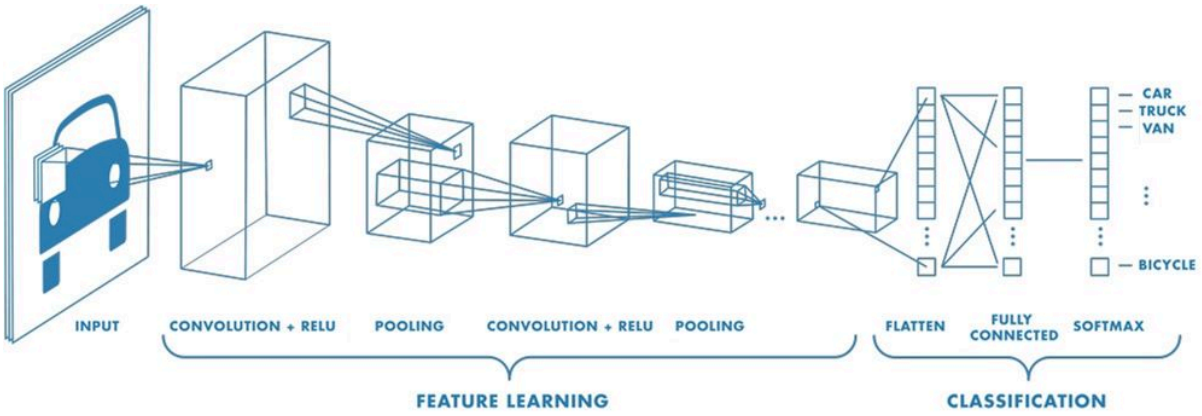
- **Computational bottleneck:** Standard networks on large images result in enormous computation.

  Solution: Use convolutional layers to reduce parameter count and computational requirements.

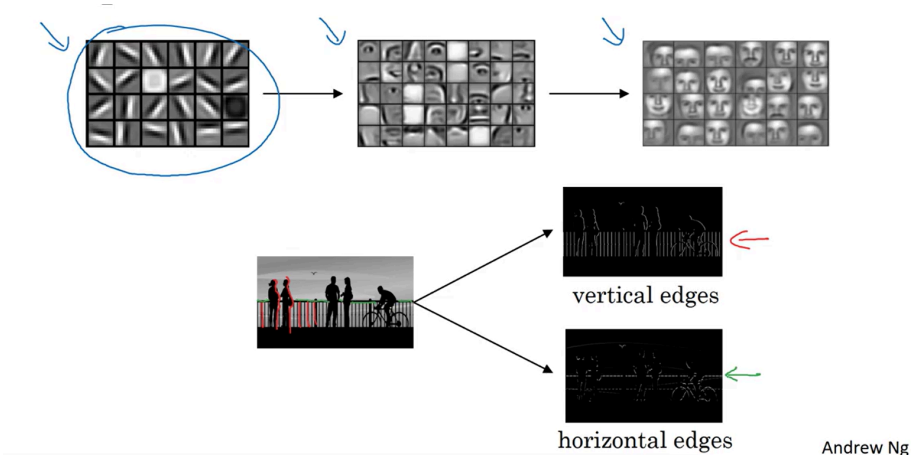
Creative Extension:

Project Idea:

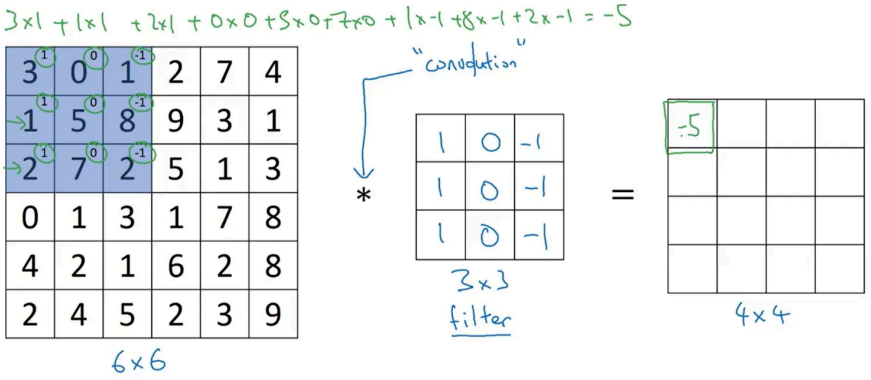
Build a simple image classifier that distinguishes between different types of objects (e.g., cats vs. dogs) using a convolutional neural network. As an extension, try implementing a bounding box detector to locate each object in the image.

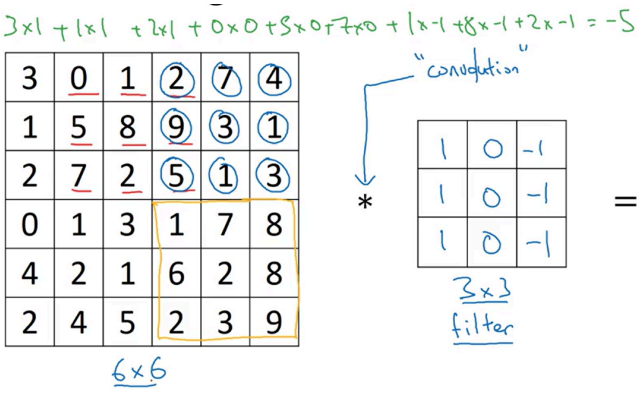


Edge Detection Example



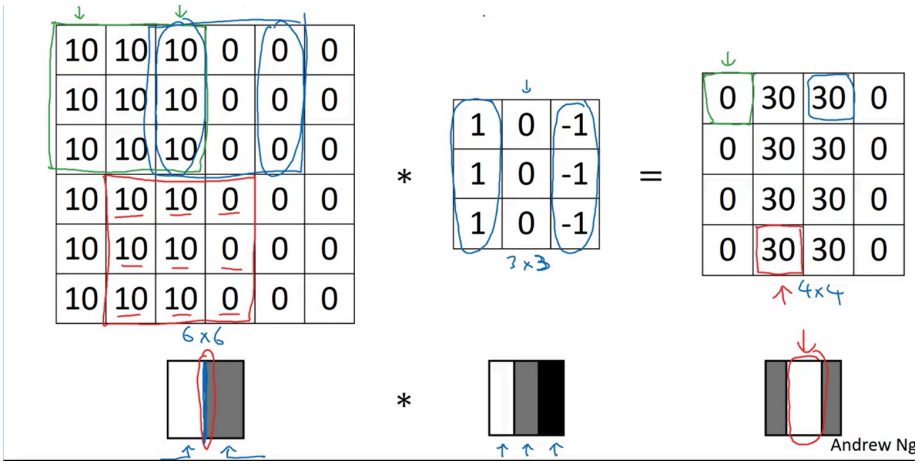
Vertical Edge detection





here in labs we are going to use for filter

1. Python: `conv_forward` function
2. Tensorflow: `tf.nn.conv2d`
3. Keras: Conv2D



1. Methodology & Context

- **Convolution Operation**: A key block in Convolutional Neural Networks (CNNs). Explored here via *edge detection*.
- **Application**: Detecting simple patterns (edges) using small filters; first step in visual understanding.
- **Example**: Using a small (6×6) grayscale image and applying a 3×3 filter (vertical edge detector).

---

2. Step-by-Step Process

- **Image Representation**: Grayscale (6×6×1), not RGB (6×6×3).
- **Filter Used** (Kernel/Mask):

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

> 💡 You always write the filter values row by row in code and definitions; the pattern (whether for vertical or horizontal edge detection) depends on the placement of the values inside that row-wise structure.

- **Convolution**:
  - Move the filter over the image, calculate elementwise product & sum for each region.
  - Fill new matrix (output size 4×4) with these values.
- **Output**: Result is another matrix/image highlighting edges (especially vertical).

---

3. Illustrative Example

- Filter slides over the original image, region by region.
- For each position:
  - Multiply corresponding elements → Sum → Place in output.
  - Example: First top-left region is summed for value -5
- After full traversal:
  - Matrix represents **vertical edge intensities**.
- **Note**: No. of output rows/cols = (input size – filter size) + 1 (if no padding).

---

4. Practical Points & Tips

- **Terminology**:
  - *Filter* ≈ *Kernel* (used interchangeably in literature).
- **Programming**:
  - Python: may mean elementwise/other multiplications; frameworks use special functions (`tf.nn.conv2d` in TensorFlow, `conv2d` in Keras).
- The left of the filter reacts to bright pixels, the right to dark → transition yields large response.
- **Why it detects vertical edges**:

---

5. Visualization Insight

- If image has a sharp change from white (10) to black (0) at center (as in split example), the filter outputs highest value along the center.
- Larger images/real data: Detector is robust for edge identification.

---

6. Core Takeaways for Exams/Interviews

- **Early CNN layers**: Automatically learn such filters (don't handcode in real systems – here to build understanding).
- **Filter output = feature map**: Used by later layers for higher-level pattern recognition.
- **Common Errors**: Misunderstanding output dimensions, filter orientation, or assuming filters are always hardcoded.

---

7. Real-World Relevance

- **Fundamental for**: Object detection, image segmentation, and feature engineering.
- **Building Blocks**: Stacking such layers → recognises increasingly complex patterns (edges → textures → object parts → full objects).

---

8. Advanced Insights

- **Sensitivity**: Filter design (values, orientation) affects detection type (vertical/horizontal/diagonal).
- **Padding/Stride**: Controls output size and information retention.

- **Mathematical Notation**: is traditional for convolution; in code, always check actual function.

9. Pro Tips & Best Practices

- When designing CNNs, start with intuition from these edge filters for initial understanding.
- Visualize feature maps at different layers when debugging network behavior.
- Use different filter sizes/types for richer feature extraction if dealing with noisy or complex input.

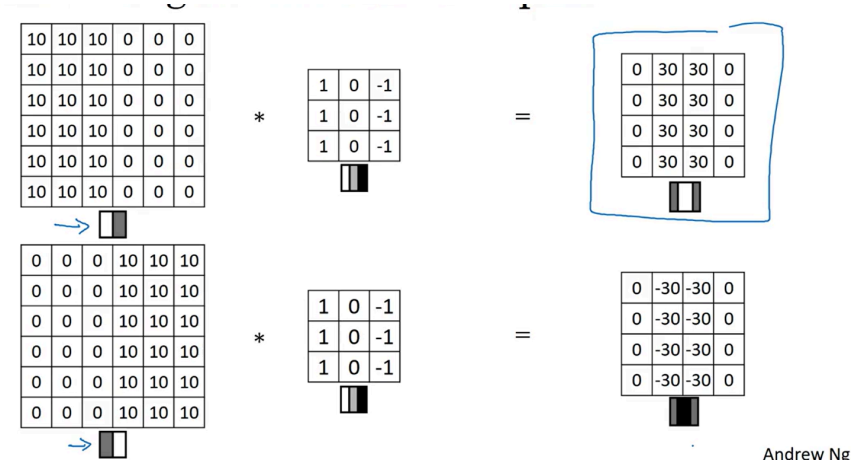
10. Curiosity Sparks

- Try experimenting in software by convolving example images with various custom kernels (not just for edges!).
- Observe changes when flipping kernel or using horizontal/diagonal versions.

Summary Table

Concept	Key Point
What is convolution?	Sliding filter, elementwise multiply & sum
Output size formula	(Input – Filter) + 1 (each dimension)
Vertical edge filter	$\begin{bmatrix}1&1&1;0&0&0;-1&-1&-1\end{bmatrix}$
Practical use	Early CNN feature extraction
Real-world outcome	Highlighting sharp vertical transitions in images
Key realization	CNN learns and combines such filters for full object ID

More Edge Detection



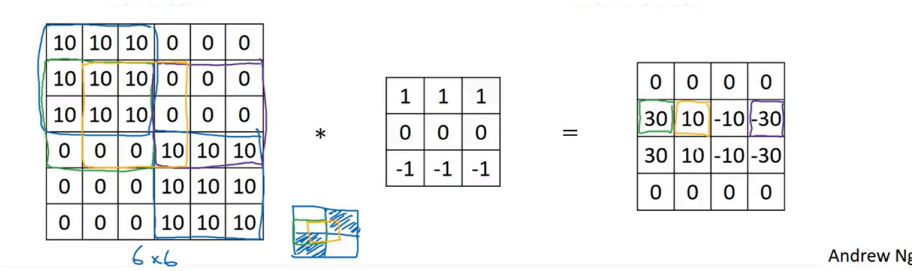
1. Review & Context

- Convolution enables the implementation of edge detectors in images.
- Prior example: Vertical edge detected when light (high values) on one side and dark (low values) on the other.

---

2. Positive vs Negative Edges

- **Edge Polarity:**
  - Positive Edge: Light-to-dark transition (e.g., left side bright, right side dark → strong positive value in filter output).
  - Negative Edge: Dark-to-light transition (e.g., if sides flip, same filter outputs strong *negative* values).
  - The filter distinguishes the direction of transitions—the sign in the output matrix reflects this polarity.
- **Mathematical Example:**



  - Image (6x6): all 10s on left, 0s on right → filter yields +30 at the edge.
  - Flipped (10s on right): same filter yields -30 at the edge.
- **Tip:** If orientation is not important, take the absolute value of the output to ignore direction.

---

3. Multiple Types of Edge Detectors

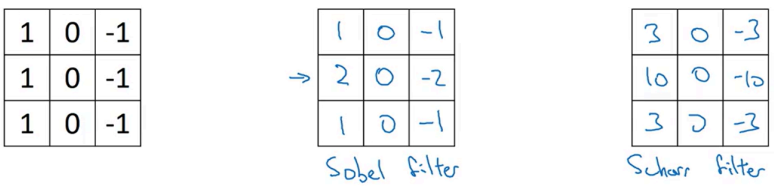
- **Horizontal Edge Detector:**
  - Typical filter for horizontal edges:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

  - Responds strongly where top is bright and bottom is dark.
  - Negative value for bottom bright, top dark.
- **Example:** Checkerboard-style patterns demonstrate both horizontal and vertical edge responses, including intermediate values for mixed regions.

---

4. Classic (Handcrafted) Filters



- **Sobel Filter (vertical):**

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

  - Emphasizes the central row, increasing robustness to noise.

- **Scharr Filter (vertical):**

$$\begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

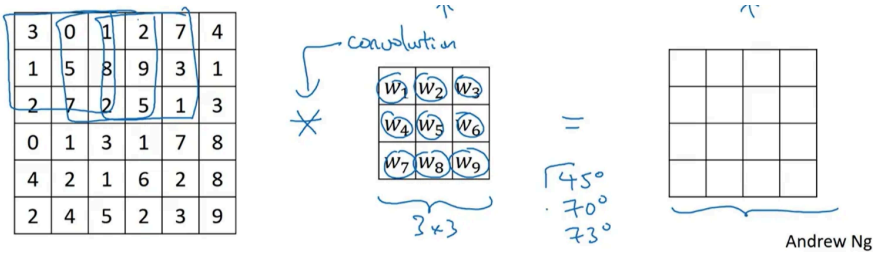
  - Places much stronger weight on the center, produces sharper edge detection.

- **Note:**
  - Flip filters 90° for horizontal edge detection.
  - Historically, much debate in vision research about optimal filter coefficients.

---

5. Data-driven Filter Learning

- Rather than hand-designing filters, **modern deep learning learns filter values as parameters** via backpropagation.



  - Each filter is a set of weights (3×3=9 parameters).
  - The network learns optimal filters for its task: might find classic solutions (Sobel/Scharr), or invent new, more data-suited filters.
  - Networks can also discover edge detectors at arbitrary angles (not just vertical/horizontal).

- **Key Insight:**

  Parameterizing all filter coefficients and learning them from data is far more powerful than manual design and is the foundation of convolutional layers in neural networks.

---

6. Practical Points & Applications

- Edge detectors (vertical, horizontal, custom) = entry point of visual understanding in computer vision pipelines.
- Absolute value trick: to ignore orientation of edge but care about strength.
- Small images show strong edge responses; larger images lead to cleaner edge maps.

---

7. Visualization & Intuition

- The *sign* of the filter output tells you edge direction.
- Classic named filters (Sobel, Scharr) are reliable, but learned filters may adapt better to real images/noise.

---

8. Exam & Interview Tips

- Explain filter construction for different edge orientations.
- Show understanding of how learned filters work (parameterization vs hand-crafted).
- Know use of absolute value for edge strength irrespective of orientation.

---

9. Research/Project Ideas

- Develop hybrid frameworks mixing hand-designed and learned filters for special applications (e.g., medical imaging, noisy environments).
- Explore learned filters for multi-angle or non-linear edge detection.
- Analyze how filter learning changes with dataset characteristics (contrast, noise).

---

10. Summary Table

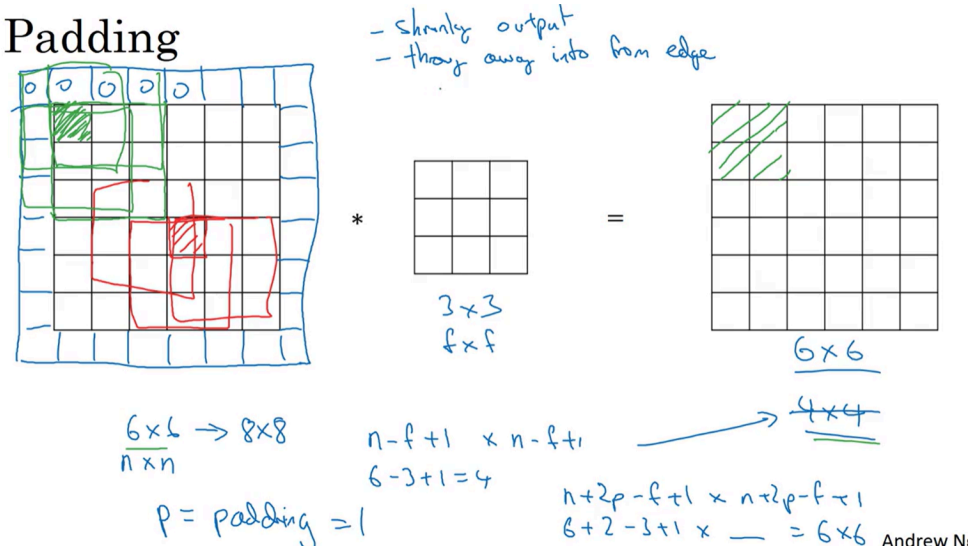
Topic	Key Point
Polarity (direction)	Sign of output distinguishes light→dark vs dark→light edges
Horizontal vs Vertical	Change values in rows (horizontal) or columns (vertical)
Classic filters	Sobel, Scharr—emphasize central pixels for robustness
Learned filters	Parameters updated by backprop—might outperform classical
Real-world utility	Common start point in object detection/image analysis

---

Bottom line:

Deep networks harness classic edge detection principles—with critical extensions—by learning filter weights directly from massive data, empowering robust, adaptive visual feature extraction.

Padding



1. Definition & Motivation

- **Padding:** The process of adding extra border pixels (usually zeros) around the input image before convolution.
- **Key reasons:**
  - Prevent rapid shrinking of the output after repeated convolutions.
  - Preserve information near the image borders, which would otherwise be under-represented in outputs.

---

2. Output Size Without Padding

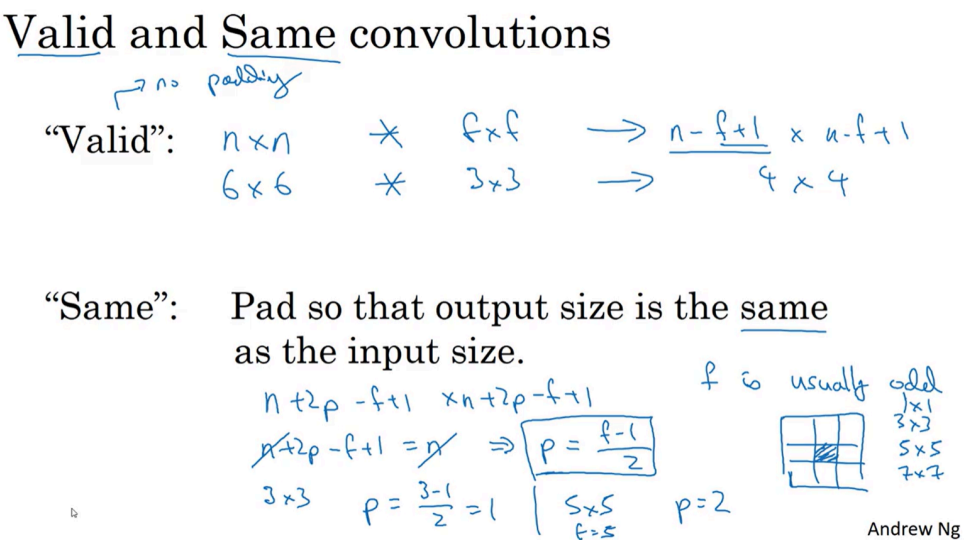
- If you convolve an n×n image with an f×f filter (kernel) *without padding*:
  - **Output size:** (n−f+1)×(n−f+1)
  - **Example:** 6×6 image and 3×3 filter → output is 4×4.

- **Downsides:**
  - Each convolution shrinks dimensions.
  - Border pixels contribute to fewer outputs than central pixels → border info is gradually lost.

---

3. How Padding Works

- **Pad by p pixels:** Add p layers of zeros to every border.
- **Padded input size:** (n+2p)×(n+2p)
- **Output formula (with padding):** (n+2p−f+1)×(n+2p−f+1)
  - Pad a 6×6 image with 1-pixel border (p=1): becomes 8×8.
  - Convolve with a 3×3 filter: output is 6×6 (original size preserved).
- You may also pad more (e.g., p=2), for additional border(s).



4. Types of Padding in Practice

- **Valid Convolution ("no padding"):**
  - p=0
  - Output size: (n−f+1)×(n−f+1)
- **Same Convolution ("zero padding"):**
  - Add enough padding so the output size equals input size.
  - Solve: $n + 2p - f + 1 = n \Rightarrow p = \frac{f-1}{2}$
  - Works cleanly when filter size f is *odd* (e.g., 3, 5, 7).

---

5. Mathematical Examples

- **For 3×3 filter:**
  - $p = \frac{3-1}{2} = 1$
  - Pad 1 pixel all around → output matches input size.
- **For 5×5 filter:**
  - $p = \frac{5-1}{2} = 2$
  - Pad 2 pixels all around.

---

6. Practical Design & Intuition

- **Avoid output shrinking:** Especially important in *deep networks* with many layers—otherwise, repeated convolutions would shrink data to unusable sizes.
- **Preserve edge/corner information:** Ensures all pixels in the input have a fair chance to influence the output, not just those in the center.
- **Zero padding ("same")** is standard for most CNN architectures.
- **Odd-sized filters (3x3, 5x5):**
  - Preferable for symmetric padding and well-defined centers.
  - Even-sized filters risk asymmetric, more complex padding needs.

---

7. Tips for Implementation & Exams

- Know the output dimension formula (with & without padding).
- Understand when to use 'valid' vs 'same'.
- Explain why odd filter sizes are preferred (symmetry, central pixel, easier coding).
- Default to padding with zeros unless otherwise specified.

---

8. Advanced Application & Insights

- Flexible amount of padding possible (not limited to 1 pixel).
- In custom architectures, sometimes reflection or replication (not zero) padding is used (not discussed here).
- Deeper models (e.g., ResNet, VGG) almost always rely on "same" padding for consistent feature map sizing.

---

9. Common Pitfalls

- Forgetting to pad leads to mismatch in expected output sizes in deep nets.
- Using even-sized filters complicates padding logic (avoid if possible).
- Not accounting for border effects can lead to loss of critical image information.

---

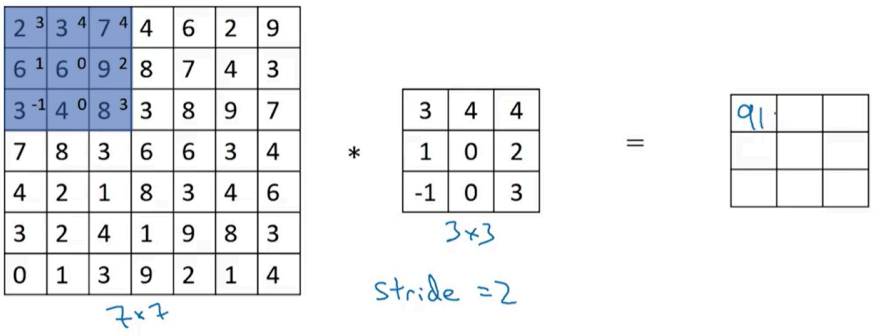
10. Summary Table

Concept	Formula / Meaning	Use Case
No padding (valid)	Output: (n−f+1)×(n−f+1)	Minimal use, shrinking size
Zero padding (same)	Pad $p = \frac{f-1}{2}$; Output size = input size	Standard for deep CNNs
Odd filter size	Prefer 3x3, 5x5	Center pixel, symmetry

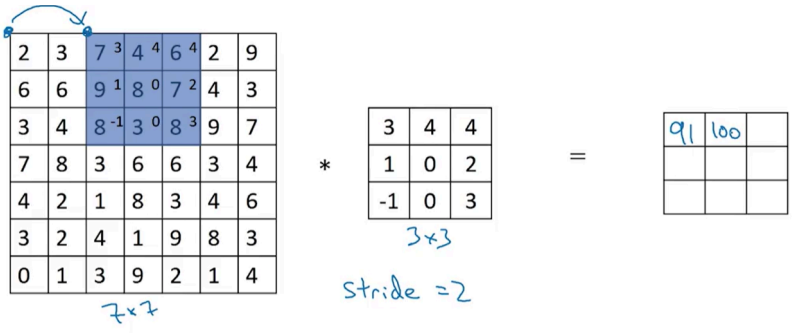
Bottom line:

Padding is essential for preserving feature map size and information content in deep convolutional networks, and is almost always implemented with zero-padding ("same") and odd filter sizes for mathematical convenience and performance.

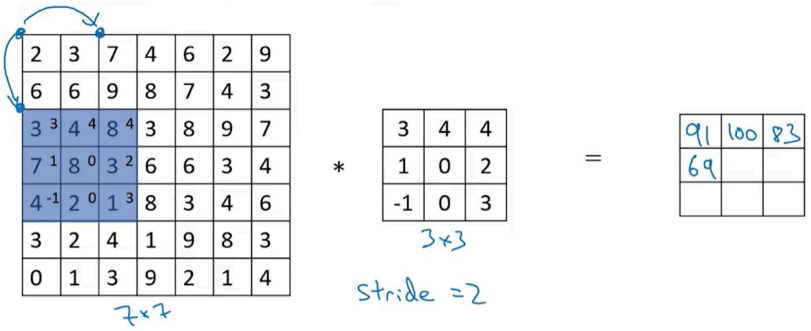
Strided Convolutions



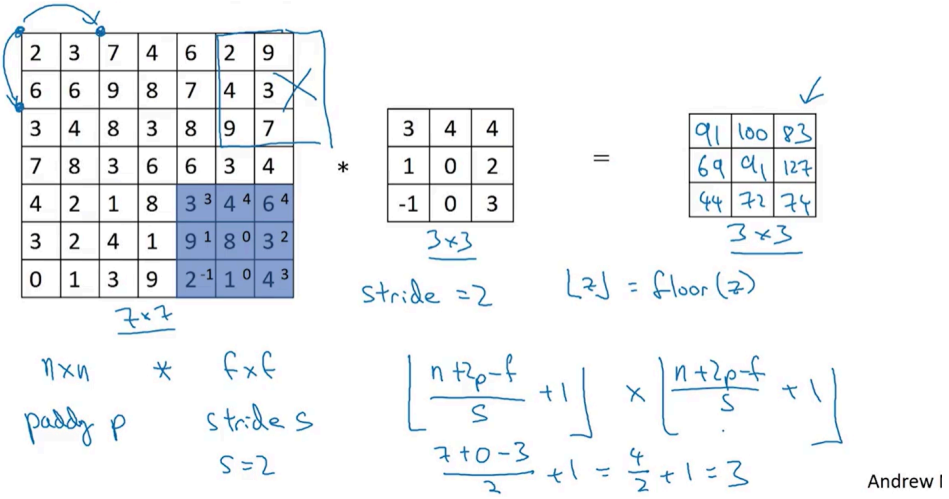
we start as usual



we move 2 steps instead of 1 step till end of all columns



for down also we move two steps instead of 1 step



Andrew I

1. What is a Strided Convolution?

- In regular convolution, the filter moves 1 pixel at a time (stride = 1).
- In strided convolution, the filter skips pixels, moving by more than one pixel at a time (stride = s > 1).
- Purpose: Reduces dimensionality, controls the output size, speeds up computation.

---

2. Illustration and Example

- Given: 7×7 input image, 3×3 filter, stride s=2.
- Instead of moving the filter one step, move (jump) it by 2 steps at a time, both horizontally and vertically.
- The output is determined only at those positions where the filter fully overlaps the image.

---

3. Formula for Output Dimension

$$n \times n \text{ image} \qquad f \times f \text{ filter}$$

$$\text{padding } p \qquad \text{stride } s$$

Output Size:

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \quad \times \quad \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

For an input image of size n×n, filter size f×f, padding p, and stride s:

$$\text{Output size} = \left\lfloor \frac{n+2p-f}{s} \right\rfloor + 1$$

- $\lfloor \cdot \rfloor$: Floor function, round down to the nearest integer.
- If the final filter window does not fully fit inside the (padded) input, it is ignored.
- Example calculation:
  - n = 7, f = 3, p = 0, s = 2
  - Output: $\left\lfloor \frac{7+0-3}{2} \right\rfloor + 1 = \lfloor 2 \rfloor + 1 = 3$
  - Output is 3×3 matrix.

---

4. Implementation Details

- Only perform computation if filter (blue box) is fully contained within the input.
- For padding, add zeros around the border if required before convolution.

---

5. Convention: Floor in Output Size

- In case $\frac{n+2p-f}{s}$ is not an integer, always **round down** (floor).
- Ensures only valid, fully-overlapped filter computations are included in the output.

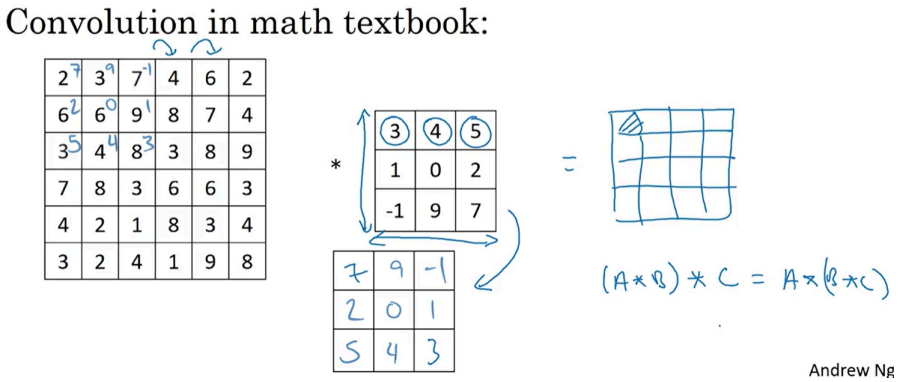
---

6. Stride & Dimensionality Control

- Higher strides decrease output dimension more aggressively (results in smaller output).
- Used to down-sample feature maps (similar to pooling, but with trainable weights).

---

7. Cross-correlation vs. Convolution (Important Note)



technical note on cross-correlation vs convolution

- Math textbooks: Convolution requires flipping filter both vertically and horizontally before sliding (true convolution).
- Deep learning: Flipping is usually skipped. The operation used is technically **cross-correlation**.
- In DL literature, this operation is almost always called "convolution."

---

8. Why Skipping Flip is Acceptable

- For neural networks, flipping the kernel does not change the outcome, since weights are learned.
- For signal processing, flipping enables properties like associativity, but this is less relevant in DL applications.

---

9. Practical Takeaways

- Most DL frameworks implement cross-correlation but call it convolution.
- Always check the stride and padding settings in the framework you use.
- Know the formula—very useful for architectures and debugging shapes.

---

10. Next Step

- Strided convolutions so far explained for 2D (matrix) data.
- In next lectures, extends to 3D (volumetric) convolutions.

Formula Recap (For Revision)

$$\text{Output size} = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1$$

Key Applications

- Downsampling, feature map reduction, neural network architecture control.
- Commonly used in CNNs after initial convolutions to rapidly reduce feature map size, prior to flattening or dense layers.

Pro-Tips for GATE/Data Science Interview

- Be able to derive and explain the output size formula given n, f, p, s.
- Know the terminological differences between convolution and cross-correlation.
- Recognize that stride is a form of sub-sampling and dimensionality reduction.

Summary Table

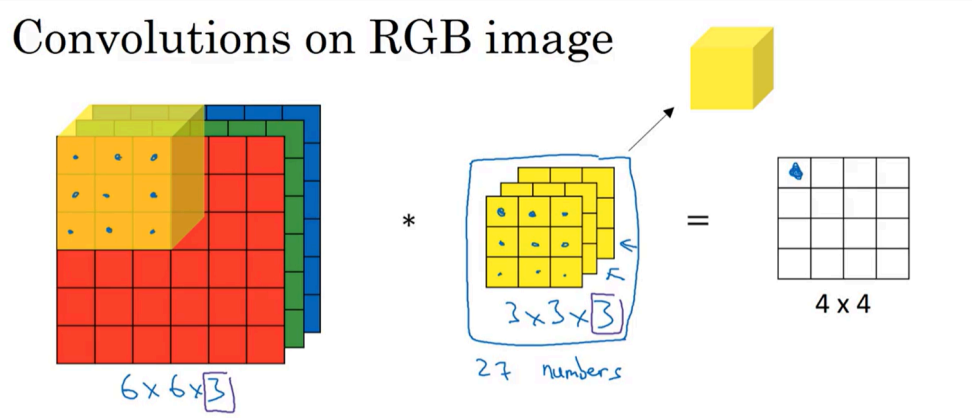
Parameter	Role	Example Value (from video)
n	Input size	7

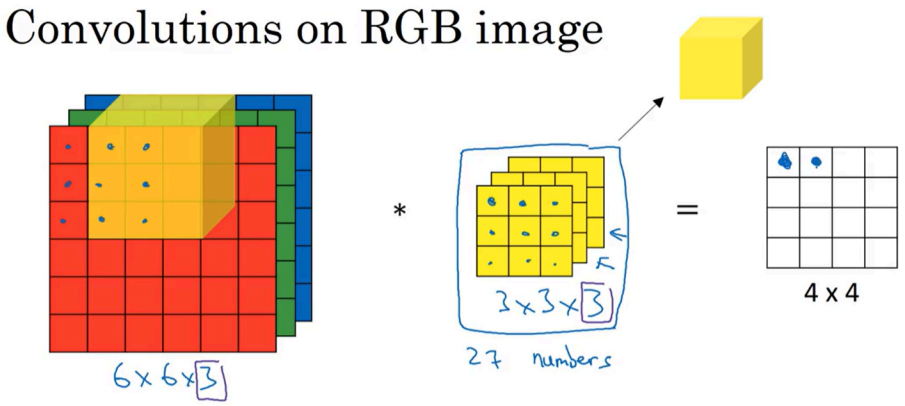
Parameter	Role	Example Value (from video)
f	Filter size	3
p	Padding	0
s	Stride	2
Output	Output size	3×3

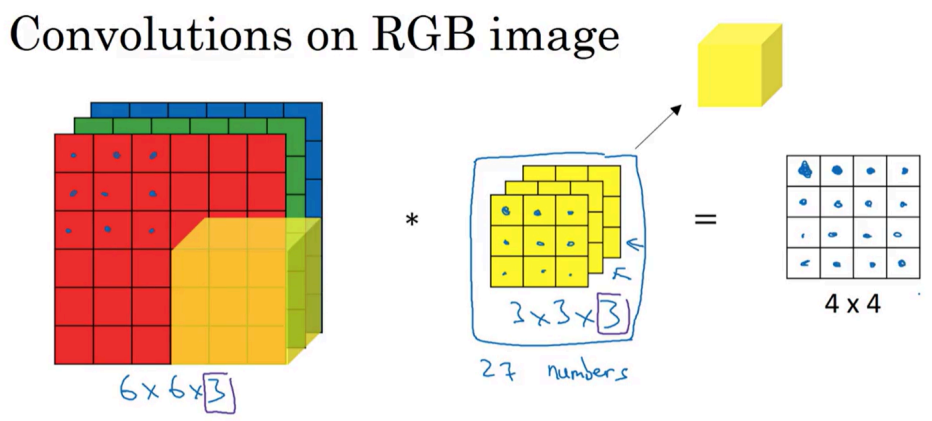
Convolutions Over Volume

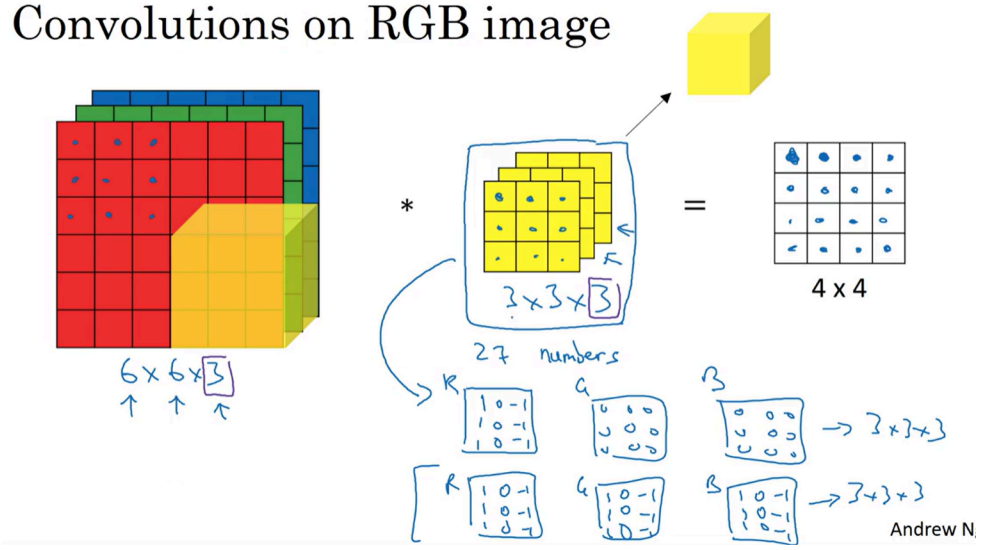
1. Introduction to 3D Convolution

- Standard convolution on images involves sliding a 2D filter over a 2D (grayscale) image.
- For color images (RGB), each image is a volume: $Height \times Width \times Channels$ $(e.g., 6 \times 6 \times 3)$
  - *Channels* typically represent colors: Red, Green, Blue (R, G, B).
- **Key Point:** The filter for such images must also be 3D to span all channels, e.g., 3×3×3.









2. Dimensionality Deep Dive

- **Input Volume:** $H \times W \times C$, e.g., $6 \times 6 \times 3$.
- **Filter (Kernel):** $f \times f \times C$, e.g., $3 \times 3 \times 3$.
  - Important: The *depth* of the filter (**number of channels**) must match the input's channels.
- **Resulting Output:**
  - For stride = 1 & no padding:
    - Spatial size: (n−f+1)
    - Output shape: (n−f+1)×(n−f+1)×1
- **Example:** Input = 6×6×3, filter = 3×3×3 → Output = 4×4×1.

---

3. Mechanism: Convolution Over Volume

- Place the filter over the image at a specific (top-left) position.
- Multiply each filter entry with the corresponding pixel/channel value.
- Sum all the products —> place in output volume.
- Slide filter by stride and repeat over entire input.

- Repeat for all output positions.

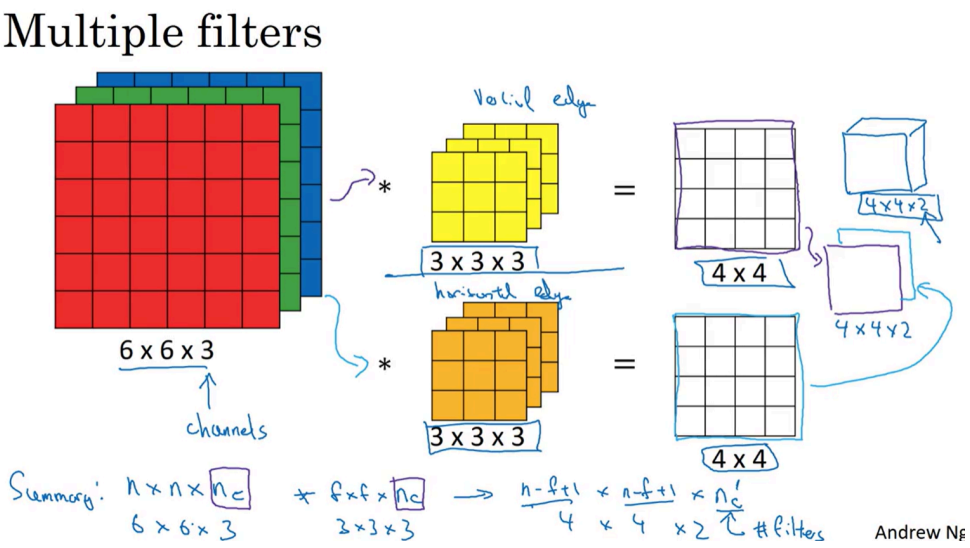
---

4. Feature Detection Examples

- **Channel-sensitive filters:**
  - You can construct filters to look for edges/features only in one color channel (e.g., edge detector only on the Red channel, with zeros in Green and Blue).
  - **General-purpose filter:** Set the same pattern across all channels if you wish to detect features across all colors.
- **Different parameter settings** in the filter result in different patterns/features being detected.

---

5. Multiple Filters: Multi-channel Output



- In practice, you often want to detect *multiple* features (vertical/horizontal/angled edges, etc).
- Have a collection of filters, e.g., two filters.
  - Each filter produces its own 4×4 output.
  - Stack these outputs in depth dimension to get 4×4×2.
- **Generalization:**
  - Input: $n \times n \times n_C$
  - Filter: $f \times f \times n_C$ (n_C = input channels)
  - Number of filters: n'_C
  - Output: $(n - f + 1) \times (n - f + 1) \times n'_C$

---

6. Mathematical Summary

If input size is $n \times n \times n_C$:

- Each filter: $f \times f \times n_C$
- For n'_C filters, output: $(n - f + 1) \times (n - f + 1) \times n'_C$ (assuming stride=1, no padding).

Notation:

- *Channels* (aka *depth* in some literature) = size of the third dimension, but “channels” is preferable to avoid confusion with network “depth”.

---

7. Practical Insights

- **Why use 3D convolutions?** Enables the network to *directly* operate on color images and detect diverse features in all color channels.
- **Power:** Enables detection of multiple features (edges, textures, color patterns) simultaneously.
- The last dimension (number of channels/features in output) = number of filters used.
- Ready to define a convolutional layer using these principles.

Possible Applications & Notes for Research

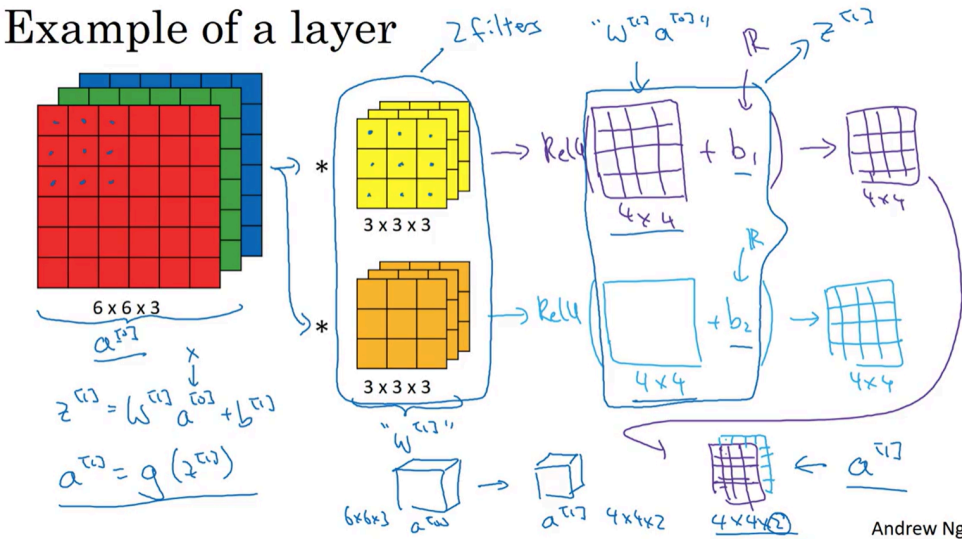
- **Color feature extraction:** Use in medical imaging, satellite, and any multi-spectral image analysis.
- **Custom filter design:** Targeting specific features in specific channels (e.g., anomaly detector only in infrared channel).
- **Stacking multiple filters:** For rich, hierarchical feature maps, crucial for deep learning applications like image classification, object detection, and segmentation.
- **Creativity boost:**
  - Experiment with varying the number of filters and filter shapes to boost model’s expressive power or to make “curiosity-driven” feature detectors (e.g., “what-if” channel-specific kernels).

Memory/Exam Short Formulae

- Output spatial dimension (stride S, padding P): $(n + 2P - f)/S + 1$
- Output channels: = Number of filters used

One Layer of a Convolutional Network

1. Overview: What is One Layer in a ConvNet?



- It transforms a 3D input volume (e.g., image) into an output volume using multiple **filters (kernels)**, each producing its own feature map (output plane).
- Each filter slides (convolves) over the input, computes dot-products, and gets a 2D activation map.
- **Bias** is added (one bias per filter), and a **non-linearity** (e.g., ReLU) is applied.

---

2. Detailed Workflow

2.1. Convolution Operation

- Given an **input** of shape $n_H \times n_W \times n_C$ (height, width, channels), e.g., $6 \times 6 \times 3$.
- Convolve with n_F filters, e.g., two filters ($3 \times 3 \times 3$ each).
- Each filter produces a 2D matrix (here, 4×4 after padding and stride).

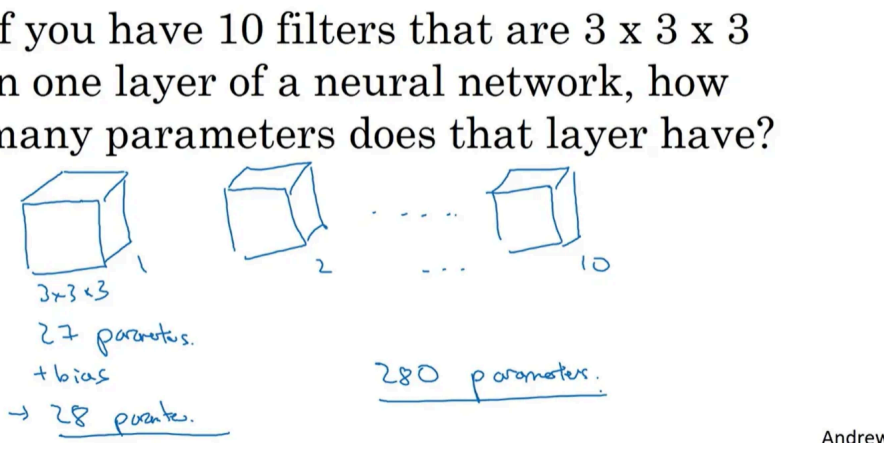
2.2. Adding Bias and Non-linearity

- For each filter:
  - Add a single bias (scalar) to the entire feature map (broadcasted).
  - Apply activation function (typically ReLU): $A = \mathrm{ReLU}(Z + b)$.

2.3. Stacking Outputs

- Final output: Stack all filter activations along the "depth" axis.
  - If two filters: output is $4 \times 4 \times 2$.
  - For 10 filters: output will be $4 \times 4 \times 10$.

2.4. Parameter Counting



- **Each filter**: $f \times f \times n_C$ weights + 1 bias.
- For 10 filters of $3 \times 3 \times 3$: Each has 27 weights + 1 bias = 28 params/filter.
- Total: 28×10=280 parameters.
  - Note: This does **not** depend on the input image size! Even with large inputs, number of params is fixed.
- This property helps CNNs generalize better and makes them less prone to overfitting.

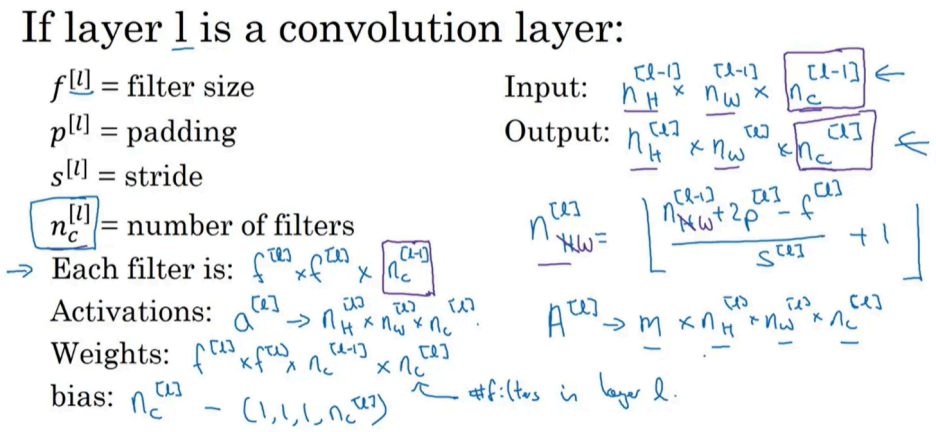
---

3. Analogy to Standard NN Layer

- Regular NN layer: $Z = W \cdot x + b, A = g(Z)$.
- Here:
  - **Input volume** analogous to x
  - **Filters** analogous to weights
  - **Bias** and **activation** same as in standard NN.
- The convolution operation is a spatially-shared linear transformation.

---

4. Notation and Formulae



4.1. Standard Notation

- $f^{[l]}$: filter size at layer l (e.g., 3×3)
- $p^{[l]}$: padding size at layer l
- $s^{[l]}$: stride at layer l

4.2. Volume Shapes

- **Input to layer** l: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- **Output of layer** l: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
  - $n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1$
  - $n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1$
  - $n_C^{[l]} =$ number of filters

4.3. Filter Tensor

- **Shape of one filter**: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- **Stacked filters**: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$
- **Biases**: vector of shape $(1, 1, 1, n_C^{[l]})$

4.4. Output

- For batch size m: $[m, n_H^{[l]}, n_W^{[l]}, n_C^{[l]}]$

---

5. Practical Conventions

- Two common tensor orderings:
  - **(height, width, channels)** (as in this lecture, e.g., TensorFlow)
  - **(channels, height, width)** (e.g., PyTorch, some other codebases)
- Be *consistent* with conventions; both work if you're systematic.

---

6. Key Takeaways

- **Convolutional Layer** = Filters + bias + activation, outputing a volume whose depth = number of filters.
- **Parameter-sharing** (weight reuse) makes ConvNets efficient and generalizable.
- **Number of parameters** depends only on filter size and number of filters, not input size.
- **Notation**: Understand filter, padding, stride, and volume shapes.
- **Output** after one layer is a stack of activation maps, ready for next Conv layer or flattening for Dense layers.

---

7. Formula Recap

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1, \quad n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1,$$
$$n_C^{[l]} = \text{number of filters at layer } l$$

How to Remember:

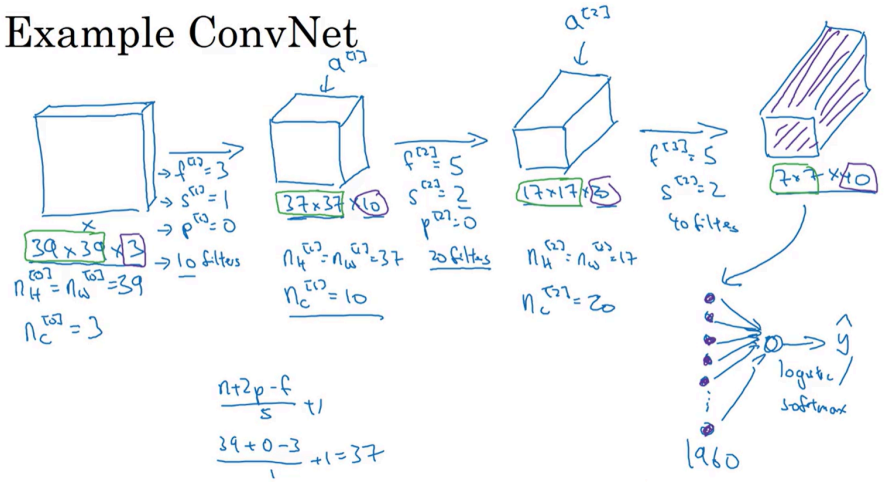
- Focus on: filter, padding, stride, input/output shape, filter stacking.
- Practice with small numbers and drawings to gain intuition.
- Don’t stress about notation details—get the structural logic and operations clear.

Applications and Intuition

- Feature detectors (edges, textures, shapes, objects) are learned as filters.
- Stacking layers lets the network build ever more complex representations.
- Efficient on large images due to fixed parameter count per layer.

Simple Convolutional Network Example

1. Problem Setup



- **Task**: Image classification (e.g., recognizing if an image is a cat or not).
- **Input Example**: 39 x 39 x 3 RGB image (Height x Width x Channels).
- *Note*: Chosen for easier calculation, but concepts are generalizable.

---

2. Network Architecture Overview

Convolutional neural networks (ConvNets/ CNNs) consist of layers that transform input volumes to output volumes.

Layer① – First Convolutional Layer

- **Filter size (f)**: 3×3
- **Stride (s)**: 1
- **Padding (p)**: 0 (valid convolution, no padding)
- **Number of filters**: 10
- **Output volume calculation**:
  - $n_H = n_W = \frac{n+2p-f}{s} + 1$
    $= \frac{39+0-3}{1} + 1 = 37$
  - **Output shape**: 37×37×10
- **Channels**: Number of filters = number of output channels.

Layer② – Second Convolutional Layer

- **Filter size (f)**: 5×5
- **Stride (s)**: 2
- **Padding (p)**: 0
- **Number of filters**: 20
- **Input volume**: 37×37×10
  - $n_H = n_W = \frac{37+0-5}{2} + 1 = 17$
  - **Output shape**: 17×17×20

Layer③ – Third Convolutional Layer

- **Filter size (f)**: 5×5

- **Stride (s)**: 2
- **Padding (p)**: 0
- **Number of filters**: 40
- **Input volume**: 17×17×20
  - Resulting size: 7×7×40

---

3. Flattening

- **Final output volume**: 7×7×40=1,960 units.
- **Flatten** this tensor into a one-dimensional vector (length 1,960).
- Pass to a final **softmax** or **logistic regression** unit for classification ("cat" vs "no cat" or multiple objects).

---

4. Key Formula: Output Dimensions

For any convolutional layer:

$$\text{Output size} = \frac{\text{Input size} + 2p - f}{s} + 1$$

where:

- f: Filter size
- s: Stride
- p: Padding

---

5. Insights and Trend Observations

- As you go **deeper** into the network:
  - The **height and width** of the activations **decrease** (e.g., 39 → 37 → 17 → 7).
  - The **number of channels** (**depth**) **increases** (e.g., 3 → 10 → 20 → 40).
- **Designing CNNs**: Most work is in choosing hyperparameters (filter size, stride, padding, number of filters per layer).

---

6. Types of Layers in ConvNets

1. **Convolutional layers (Conv)**
2. **Pooling layers (Pool)**
   - Reduces spatial dimensions for parameter + computation efficiency.
3. **Fully Connected layers (FC)**
   - Classical fully connected neurons, typically at the output.

---

7. Application and Creativity Tips

- **Experiment**: Try changing filter sizes, strides, and layers to see effects on output volume and parameter count.
- **Practical use**: This structure is the backbone for systems in image classification, face recognition, and more.
- **Research curiosity**: Analyze the tradeoff between network depth, width, and accuracy vs compute cost.
- **Boosting creativity**: Visualize how feature detection evolves layer by layer—from edges/texture (early) to complex patterns (deep).

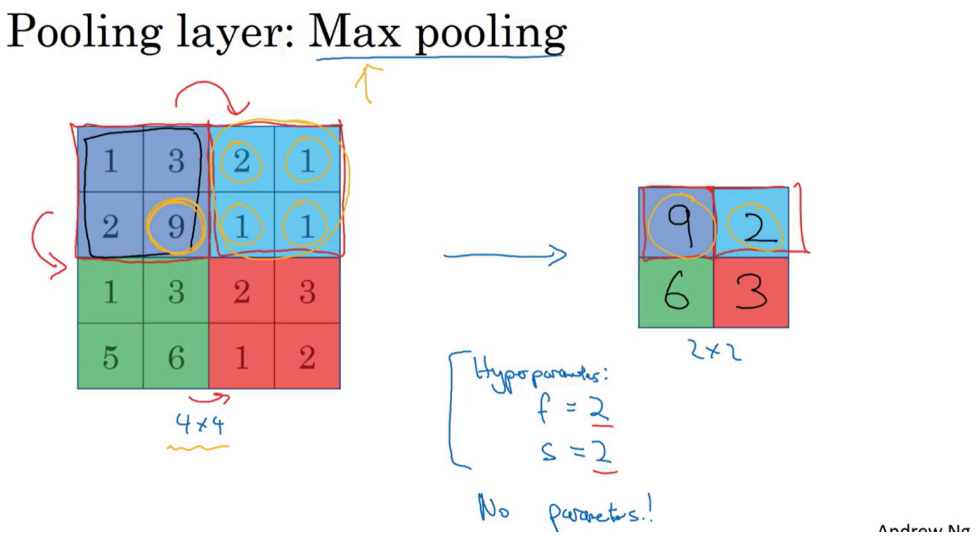
---

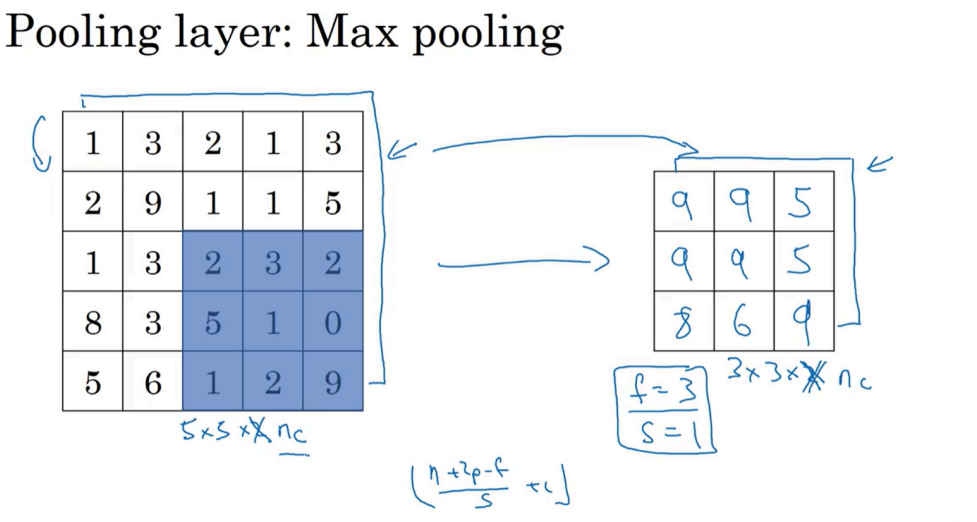
8. Quick Revision (Cheat Sheet)

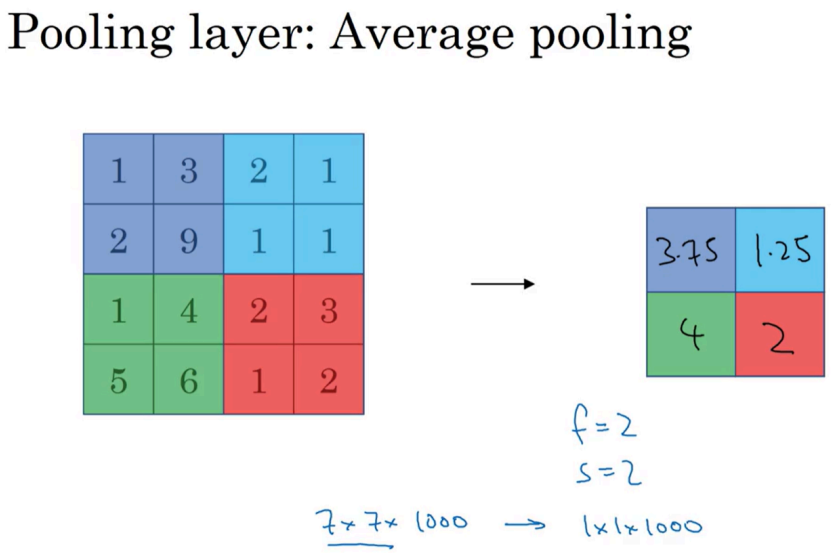
Layer	Input Size	Filter	Stride	Padding	# of Filters	Output Size
1	39 x 39 x 3	3x3	1	0	10	37 x 37 x 10
2	37 x 37 x 10	5x5	2	0	20	17 x 17 x 20
3	17 x 17 x 20	5x5	2	0	40	7 x 7 x 40
FC	7 x 7 x 40 (1960)	-	-	-	-	Output vector

Pooling Layers

Pooling Layers in Convolutional Neural Networks (CNNs)







Purpose of Pooling Layers

- Used alongside convolutional layers in ConvNets.
- Main roles:
  - **Reduces the spatial size** (height & width) of the representation, cutting down computation and memory usage.
  - **Provides translation invariance** (makes features more robust to small shifts and distortions).
  - Helps prevent overfitting.

Types of Pooling

- **Max Pooling:** Selects the maximum value in each region (most commonly used).
- **Average Pooling:** Computes the average value in each region (rarely used, with few exceptions).

How Max Pooling Works – Step-by-Step Example

Suppose you have a 4×4 input:

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

- **Hyperparameters**:
  - **Filter size** (f): What area you look at (e.g., 2×2 or 3×3).
  - **Stride** (s): How much you slide the window each step (e.g., 2 or 1).
- **For f=2,s=2:**
  - Divide the input into non-overlapping 2×2 regions.
  - In each region, pick the **maximum value**.
  - Output size: 2×2.

E.g. $\begin{bmatrix} \boxed{1\,3\,5\,6} & \boxed{2\,1\,1\,2} \\ \boxed{0\,1\,2\,6} & \boxed{9\,3\,3\,8} \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 2 \\ 6 & 9 \end{bmatrix}$

]

- **For f=3,s=1 on a 5×5 input:**
  - The output size is calculated as:

    $$\text{Output size} = \left\lfloor \frac{n+2p-f}{s} \right\rfloor + 1$$

    (where n: input size, p: padding; often p=0 in pooling)
  - Move the 3×3 window by 1 step at a time, apply max operation in each region.

Pooling over Multiple Channels

- Pooling is applied **independently for each channel** in the output volume.
- If input is $N_H \times N_W \times N_C$ output will be $N'_H \times N'_W \times N_C$.
- **Channels are preserved** (no mixing) — e.g., for 3 color channels, pool each channel separately.

Average Pooling Example

- Instead of the max, compute the mean of values in each filter region.
- Used rarely, mostly in some deep network architectures to collapse a $7 \times 7 \times 1000$ volume to $1 \times 1 \times 1000$ before softmax.

Common Hyperparameter Choices

- f=2,s=2: Most common, shrinks representation by factor of 2.
- Sometimes f=3,s=2.
- **Padding (p)**: Rarely used (almost always p=0 for pooling).

Key Properties

- **No Parameters to Learn:**

  Pooling layers have no weights; nothing is learned during backpropagation. Only the hyperparameters (f,s,p) are set.
- **Fixed Function:**

  Once setup, pooling is a deterministic operation—not optimized during training.

Summary Table

Parameter	Purpose	Defaults / Common Choice
Filter Size (f)	Size of pooling window	2 or 3
Stride (s)	Step size to move filter	2 or 1
Padding (p)	Border around input	0 (rarely used)
Type	Kind of pooling	Max (usually), Avg (rarely)

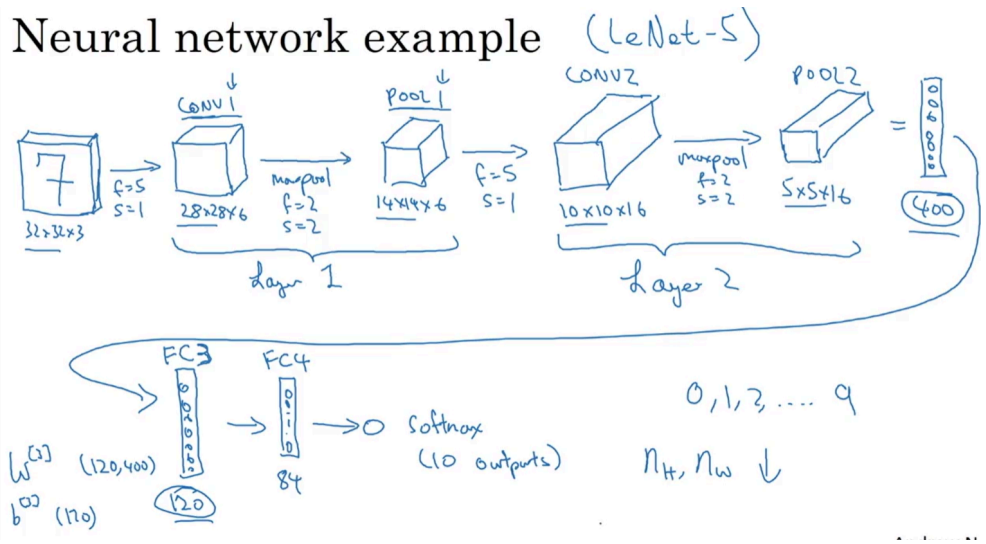
Bonus Notes:

- Pooling increases feature robustness and reduces overfitting by decreasing spatial dimensions.
- Most modern ConvNets use max pooling, sometimes omitting pooling altogether in favor of convolution with stride.
- Output size formula (assuming zero padding
- $\text{Output Height} = \left\lfloor \frac{N_H - f}{s} + 1 \right\rfloor$
- $\text{Output Width} = \left\lfloor \frac{N_W - f}{s} + 1 \right\rfloor$

Practical Tips for Competitive Exams and Research

- **Illustrate with examples** — always draw the pooling grid over sample inputs.
- **Explain intuition:** Max pooling maintains the most important feature in the region, helping in abstracting features.
- **Equations for Output Size** — Be ready to derive or use them in interview questions.
- **Know when to use average pooling:** Rare, mostly for collapsing to very low dimensions before the classifier head in deep networks.

CNN Example



Context of the Example

- **Input:** 32×32×3 RGB image (e.g., digit '7')
- **Goal:** Recognize which digit (0–9) is depicted
- **Architecture Inspiration:** LeNet-5 (by Yann LeCun); not an exact replica but parameter choices are inspired

---

Step-by-Step Layerwise Description

1. Convolutional Layer 1 (Conv1)

- **Input Size:** 32×32×3
- **Filter Size:** 5×5
- **Stride:** 1
- **Number of Filters:** 6
- **Padding:** None
- **Output Size:** $\text{Output height/width} = \frac{32-5}{1} + 1 = 28$

  So, 28×28×6
- **Nonlinearity:** Usually ReLU
- **Conv1 Output:** 28×28×6

2. Pooling Layer 1 (Pool1)

- **Pooling Type:** Max pool
- **Filter/Pool Size (f):** 2
- **Stride (s):** 2
- **Output Size:** $\frac{28-2}{2} + 1 = 14$

  So, 14×14×6
- **Common Convention:** Often groups Conv1 + Pool1 as “Layer 1” (because only Conv layers contain parameters)

---

3. Convolutional Layer 2 (Conv2)

- **Input Size:** 14×14×6
- **Filter Size:** 5×5
- **Stride:** 1
- **Number of Filters:** 16
- **Padding:** None
- **Output Size:** 10×10×16

4. Pooling Layer 2 (Pool2)

- **Pooling Type:** Max pool
- **Filter/Pool Size (f):** 2
- **Stride (s):** 2
- **Output Size:** 5×5×16
- **Conv2 + Pool2 grouped as “Layer 2”**

---

5. Flatten

- **Flatten Shape:** 5×5×16=400
- **Now treated as a vector (size 400)**

6. Fully Connected Layer 1 (FC3)

- **Input Size:** 400
- **Output Size:** 120
- **Parameter Weights:** 120×400
- **Bias:** 120-dimension
- **Standard dense neural network layer**

9. Fully Connected Layer 2 (FC4)

- **Input Size:** 120
- **Output Size:** 84

10. Final Softmax Layer

- **Input:** 84
- **Output:** 10 (classification outputs, one per digit 0−9)

General Patterns and Guidelines

- **Max pooling layers:** Do **not** introduce parameters, only shrink spatial size.
- **Conv layers:** Relatively few parameters compared to fully connected layers.
- **FC layers:** Major source of network parameters.
- **Shape progression:** Height and width decrease as we go deeper; channel count (i.e. feature dimension) typically increases.
  - E.g., $3 \to 6 \to 16$ for channels
  - E.g., $32 \to 28 \to 14 \to 10 \to 5$ for spatial dimensions

Common CNN Pattern

$$[\text{Convolutional Layer(s)}] \longrightarrow [\text{Pooling Layer}] \longrightarrow [\text{Convolutional Layer(s)}] \longrightarrow [\text{Pooling Layer}] \longrightarrow [\text{FC Layers}] \longrightarrow [\text{Softmax}]$$

- Use existing, successful architectures as templates for your own tasks.
- As you go deeper, spatial dimension drops, depth increases.

Calculation Tips

- **Activation sizes/Number of parameters:**
  - Activation shape at each layer =$h \times w \times c$ (height, width, channels)
  - For input: 32×32×3=3,072
  - Pooling/Conv layer: update spatial sizes as above
- **Pooling NEVER adds learnable parameters**
- **Drop in activation size should not be too fast—gradual shrinkage is optimal for performance**
- **Most parameters end up in the fully connected layers at the end**

Intuition and Best Practices

- **Do not invent new hyperparameters**—start from the literature (e.g., LeNet, VGG, ResNet).
- Assembling networks effectively takes intuition built from studying existing models.
- Channel count going up and width/height going down is a general CNN rule.
- Pooling is critical for compact representations and translation invariance.

Visual Block Summary

Layer	Operation	Output Shape	Parameters?
Input	—	32×32×3	No
Conv1	5x5x6, s=1	28×28×6	Yes
Pool1	2x2, s=2	14×14×6	No
Conv2	5x5x10, s=1	10×10×16	Yes
Pool2	2x2, s=2	5×5×16	No
Flatten	—	400	—
FC1	Linear	120	Yes
FC2	Linear	84	Yes
Softmax Output	Classification	10	Yes

Key Equations

- **Convolution Output:**

  $W_{out} = \frac{W_{in} - F + 2P}{S} + 1$

  Where W_{in} is input width, F is filter size, P is padding, S is stride

- **Parameter Calculation in Conv Layer:**

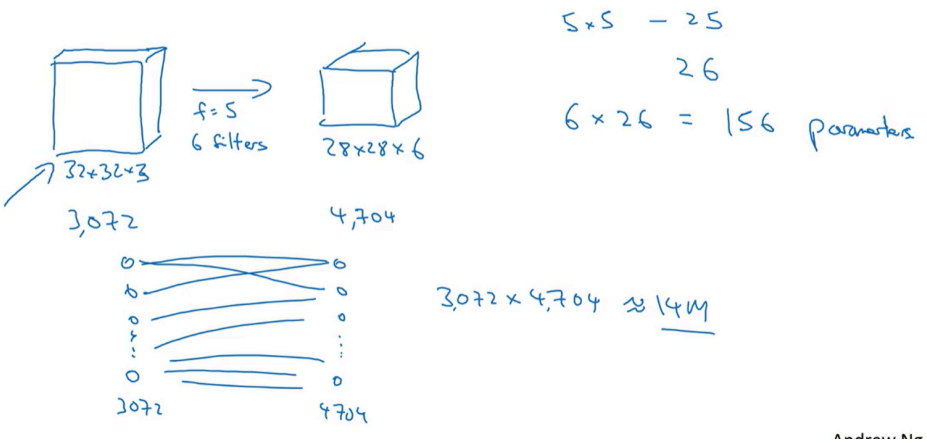
  $$\text{Parameters} = (F \times F \times C_{in}) \times \text{number of filters} + \text{number of filters (for bias terms)}$$

Final Takeaways

- Start from proven architectures and modify for your task.
- Pooling layers reduce feature size without parameters.
- Most network “memory” is in the dense layers.
- Study existing model patterns for MOST robust performance.

Why Convolutions?

1. Motivation for Convolutional Layers over Fully Connected Layers



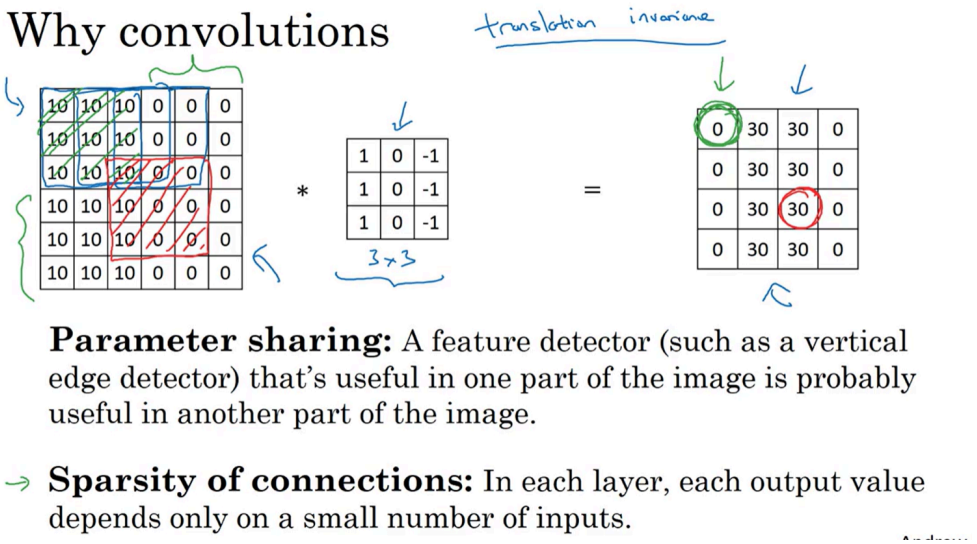
- Fully connected (dense) layers quickly become impractical for images, as every input pixel connects to every output neuron.
  - **Example**: A 32×32×3 image (RGB) has 3,072 input units. A convolutional layer output of 28×28×6 has 4,704 units.
  - Connecting all pairs: 3,072×4,704≈14 million parameters, even for small images!
  - Scaling this approach up for larger images (e.g., 1000×1000) becomes **computationally intractable**.

Convolutional Layers are Efficient

- **Parameter Calculation for Conv Layer**:
  - Each filter (e.g., 5×5x3 +1) has 75 weights + 1 bias = 7**6 parameters/filter**.
  - For 6 filters: 76×6=456 parameters!
- **Key Point**: *Massive reduction* in parameters compared to fully connected layers.

---

2. Why Can Convolutional Layers Use So Few Parameters?



a. Parameter Sharing

- The same feature detector (filter) is used at every position of the input.
  - *Example*: A vertical edge detector trained for one image region can be reused elsewhere.
  - **Mathematically**: The same weight values are applied in a sliding window convolution operation across the image.
- *Intuition*: Features like "edges", "corners", or even "eyes" (for faces) are likely useful everywhere in the image, not just one spot.

b. Sparse Connectivity

- Each output unit interacts with only a **local patch** of input units (the *receptive field*).
- *Example*: In a 3×3 convolution over a 6×6 input, an output is affected by only **9** input units, not all 36.
- *Sparsity* drastically reduces the number of weights and computations.

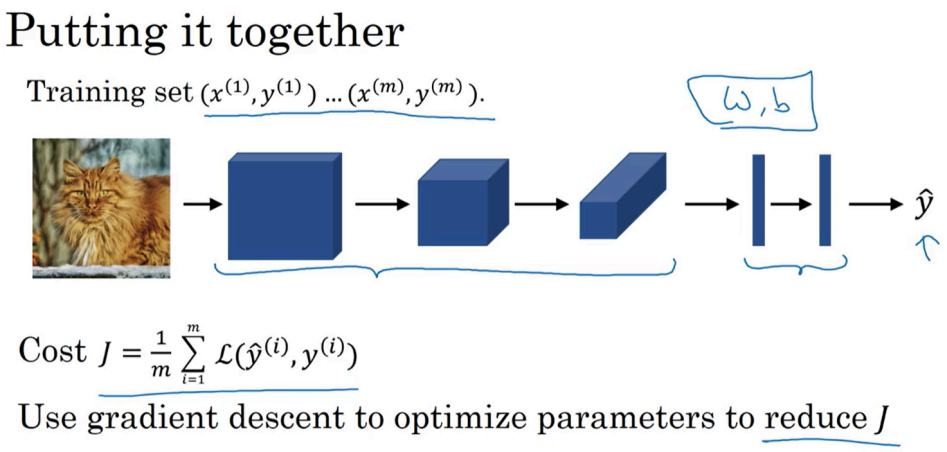
---

3. Benefits of Convolutional Structure

- **Translation Invariance:**
  - If an object shifts position in the image, the feature will still be detected with the same filter.
  - The network learns that "cat on the left" and "cat on the right" are both cats; position matters less.
- **Fewer Parameters = Less Data Needed:**
  - Simpler models reduce risk of overfitting and enable CNNs to train effectively even with less data.
- **Encodes Useful Priors:**
  - Local connectivity and weight sharing encode knowledge about *spatial locality* and *repeated patterns* in images.

---

4. Training a Convolutional Neural Network



- Typical structure:
  - Input → *Convolutional+Pooling Layers* → *Fully Connected Layers* → Output (e.g., softmax)
- Parameters (W, b) are initialized randomly.
- Define a loss/cost function J(W,b) based on labeled data (X,y).
- **Optimization Algorithms:**
  - Minimize J with **gradient descent** or variants (e.g., Momentum, RMSProp, Adam).
- The design enables highly effective image classifiers (e.g., cat detector).

---

5. Summary/Takeaways

- Convolutions allow:
  - *Massive parameter efficiency* via sharing and sparse connections.
  - *Translation invariant* feature detection.
  - *Effective and scalable* image recognition for real-world data sizes.
- The next part of the course covers practical hyperparameter choices and advanced architectures.

Math & Examples

- For a single convolutional layer:
  - For f filters of size $k \times k$:

    Parameters per filter: $k^2 + 1$

    Total parameters: $f \times (k^2 + 1)$
- For a fully connected layer:
  - $n_{in} \times n_{out}$ parameters (n_{in}: no. of input units, n_{out} : no. of output units)

Core Equations

Convolution Operation:

$$\text{output}[i,j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \text{filter}[m,n] \times \text{input}[i+m, j+n] + b$$

Parameter Efficiency:

$$\text{Total parameters (Conv layer)} = (\text{filter size}) \times (\text{number of filters}) + (\text{number of biases})$$

$$\text{Total parameters (FC layer)} = (\text{input units}) \times (\text{output units})$$

Key Intuitions

- Focus on **parameter sharing** and **sparse connections** for efficiency—relate everything back to "why not just make everything fully connected?"
- *Translation invariance* is a core benefit—understand in real image terms (“cat moves but is still a cat”).
- CNNs enable practical deep learning for vision by drastically reducing trainable parameters while retaining key learning ability.
- *Always relate math (number of parameters) to the image and convolution sizes.*

End of Week Summary:

- *You now know* fundamental limitations of fully connected layers for images, why convolutions solve them, and how CNN architectures are built for scalable, robust image understanding!