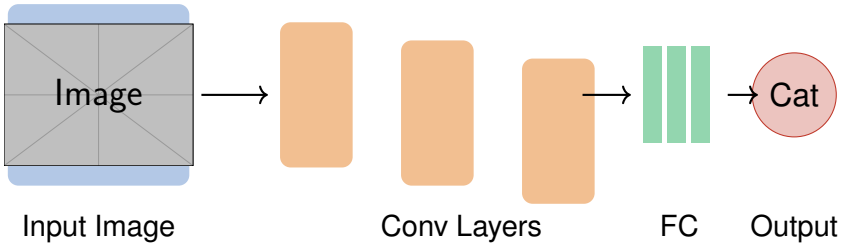
Deep Learning for Perception

Lecture 05: Convolutional Neural Networks



Topics Covered in This Lecture:

- Neural Networks for Spatial Data
- History of CNNs
- Convolutional Layers
- Kernels/Filters
- Padding Strategies
- Stride
- Pooling Layers
- CNN Architecture Overview

Instructor: Aqsa Younas

Department of Computer Science
FAST-NU, CFD Campus

Contents

Advance Organizer — What You'll Learn

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

1. **Explain** why CNNs are designed for spatial data like images
2. **Trace** the historical development of CNNs
3. **Calculate** convolution operations step-by-step
4. **Apply** padding and stride to determine output dimensions
5. **Compute** pooling layer outputs
6. **Design** basic CNN architectures
7. **Solve** numerical problems on CNN layer calculations

Prior Knowledge Required:

- Fully connected neural networks
- Activation functions
- Forward and backward propagation

1 Neural Networks for Spatial Data

Why It Matters

Images have **spatial structure** — neighboring pixels are related. Fully connected networks ignore this structure and treat each pixel independently, which is inefficient and loses important information.

1.1 The Problem with Fully Connected Networks for Images

Definition

Spatial Data refers to data where the **position** and **arrangement** of elements carries meaning. Examples include:

- Images (2D spatial structure)
- Videos (3D: 2D spatial + time)
- Audio signals (1D temporal structure)
- 3D point clouds

Why FCNs Fail for Images

Consider a small 28×28 grayscale image (like MNIST):

- **Input size:** $28 \times 28 = 784$ pixels
- **First hidden layer (256 neurons):** $784 \times 256 = 200,704$ parameters!
- For a $224 \times 224 \times 3$ color image: $224 \times 224 \times 3 = 150,528$ inputs
- **First layer alone:** $150,528 \times 256 = 38.5$ million parameters!

Problems:

1. Too many parameters $\rightarrow$ overfitting
2. Ignores spatial relationships (nearby pixels are related)
3. Not translation invariant (cat in corner $\neq$ cat in center)

1.2 Key Properties of Spatial Data

Memory Hook — Remember This!

Three Key Properties CNNs Exploit:

1. **Local Connectivity:** Pixels are most related to their neighbors
2. **Translation Invariance:** A cat is a cat regardless of position
3. **Hierarchical Features:** Edges $\rightarrow$ Shapes $\rightarrow$ Parts $\rightarrow$ Objects

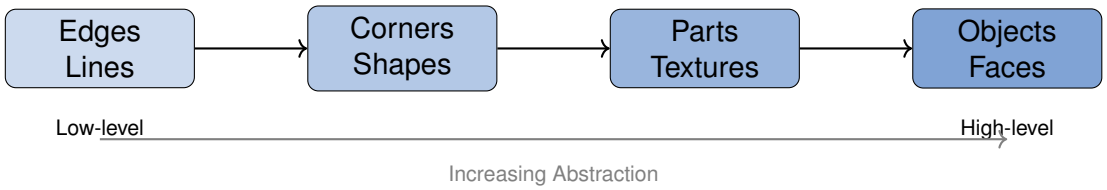


Figure 1: CNNs learn hierarchical features from simple to complex

2 History of Convolutional Neural Networks

Why It Matters

Understanding CNN history helps appreciate the key innovations and why CNNs are designed the way they are. Many ideas came from neuroscience!

2.1 Timeline of Key Developments

1959: Hubel & Wiesel — Visual Cortex

Nobel Prize-winning research showed that cat visual cortex has neurons that respond to specific **local** patterns (edges, orientations). This inspired the idea of local receptive fields in CNNs.

1980: Neocognitron (Fukushima)

First neural network with convolution-like operations. Introduced the concept of **simple cells** (feature detection) and **complex cells** (pooling for invariance).

1989: LeNet (Yann LeCun)

First successful CNN trained with backpropagation. Used for handwritten digit recognition (ZIP codes). Introduced the modern CNN architecture: Conv $\rightarrow$ Pool $\rightarrow$ Conv $\rightarrow$ Pool $\rightarrow$ FC.

2012: AlexNet (Krizhevsky et al.)

Won ImageNet competition by a huge margin. Key innovations:

- ReLU activation (faster training)
- Dropout regularization
- GPU training
- Data augmentation

This marked the beginning of the **deep learning revolution**.

2014–Present: Modern Architectures

- **VGGNet (2014):** Deeper networks with 3×3 filters
- **GoogLeNet/Inception (2014):** Multiple filter sizes in parallel
- **ResNet (2015):** Skip connections, 152+ layers
- **EfficientNet (2019):** Optimal scaling of depth/width/resolution

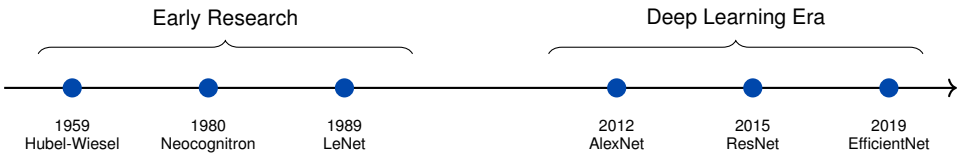


Figure 2: Timeline of CNN development

3 Convolutional Layers

Why It Matters

The convolutional layer is the core building block of CNNs. It applies learnable filters to detect local patterns like edges, textures, and shapes.

3.1 What is Convolution?

Definition

Convolution is a mathematical operation that slides a small matrix called a **kernel** (or **filter**) over the input and computes element-wise multiplications followed by a sum at each position.
The output is called a **feature map** or **activation map**.

Key Formula

2D Convolution Operation:

$$(\mathbf{I} * \mathbf{K})[i,j] = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \mathbf{I}[i+m, j+n] \cdot \mathbf{K}[m,n]$$

Where:

- $\mathbf{I}$ = Input image/feature map
- $\mathbf{K}$ = Kernel (filter) of size $k_h \times k_w$
- $[i,j]$ = Position in output feature map

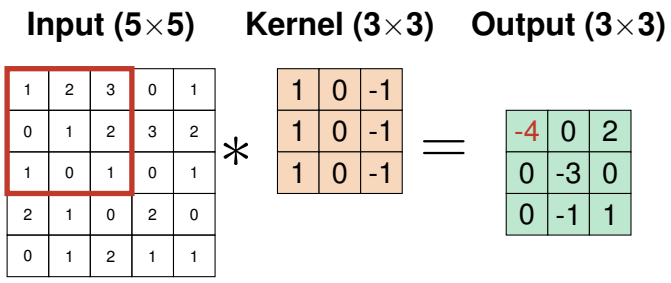


Figure 3: Convolution: Kernel slides over input, producing feature map

3.2 Common Kernels and Their Effects

Kernel Name	Kernel	Effect
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	No change
Vertical Edge	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	Detects vertical edges
Horizontal Edge	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$	Detects horizontal edges
Blur (Box)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	Smooths/blurs image
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	Enhances edges

Table 1: Common convolution kernels and their effects

Solved Example 1: Basic Convolution

Given Data:
Input Image (5×5):

$$\mathbf{I} = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 0 & 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 2 & 0 \\ 0 & 1 & 2 & 1 & 1 \end{bmatrix}$$

Kernel (3×3) — Vertical Edge Detector:

$$\mathbf{K} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Task: Calculate the output feature map (no padding, stride=1).

Solution:
Step 1: Determine output size

$$\text{Output size} = \frac{5-3}{1} + 1 = 3 \times 3$$

Step 2: Calculate output[0,0] (top-left 3×3 region)

Extract region: $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 0 & 1 \end{bmatrix}$

Element-wise multiply with kernel: $\begin{bmatrix} 1 \cdot 1 & 2 \cdot 0 & 3 \cdot (-1) \\ 0 \cdot 1 & 1 \cdot 0 & 2 \cdot (-1) \\ 1 \cdot 1 & 0 \cdot 0 & 1 \cdot (-1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & -3 \\ 0 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$

Sum: $1 + 0 - 3 + 0 + 0 - 2 + 1 + 0 - 1 = \mathbf{-4}$

Step 3: Calculate output[0,1] (shift right by 1)

Region: $\begin{bmatrix} 2 & 3 & 0 \\ 1 & 2 & 3 \\ 0 & 1 & 0 \end{bmatrix}$

Sum: $2(1) + 3(0) + 0(-1) + 1(1) + 2(0) + 3(-1) + 0(1) + 1(0) + 0(-1) = 2 + 0 + 0 + 1 + 0 - 3 + 0 + 0 + 0 = \mathbf{0}$

Step 4: Continue for all 9 positions...

(Calculations verified with Python)

Output Feature Map (3×3):

$$\begin{bmatrix} -4 & 0 & 2 \\ 0 & -3 & 0 \\ 0 & -1 & 1 \end{bmatrix}$$

Interpretation: Negative values indicate transitions from light-to-dark (left-to-right), positive values indicate dark-to-light transitions. This kernel detects vertical edges!

4 Padding and Stride

Why It Matters

Padding and stride are hyperparameters that control the **spatial dimensions** of the output feature map. They also affect how much context is captured and computational cost.

4.1 Padding

Definition

Padding adds extra pixels (usually zeros) around the border of the input before convolution.
Types:

- **Valid (No Padding):** $p = 0$, output shrinks
- **Same (Zero Padding):** Output size = Input size
- **Full Padding:** Kernel just touches input at edges

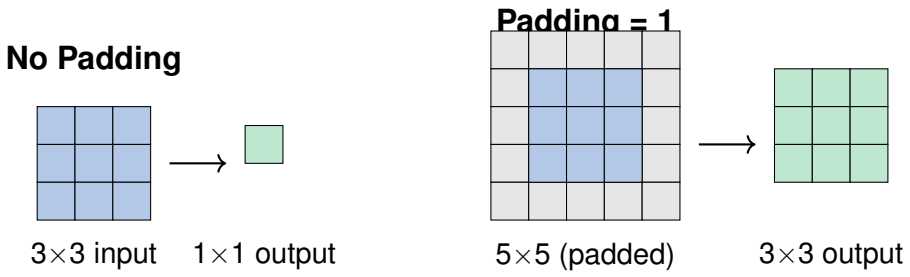


Figure 4: Effect of padding: maintains spatial dimensions

4.2 Stride

Definition

Stride (s) is the step size the kernel moves at each step.

- Stride = 1: Move 1 pixel at a time (default)
- Stride = 2: Skip every other position (downsamples by 2×)

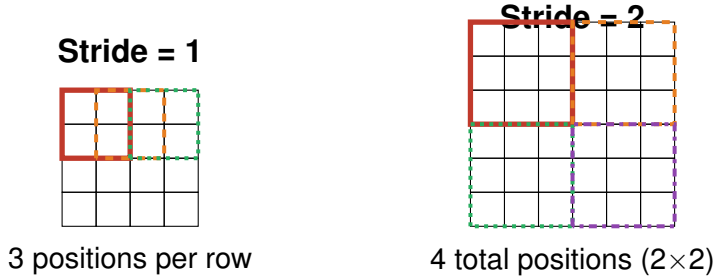


Figure 5: Effect of stride on kernel positions

4.3 Output Size Formula

Key Formula

Output Dimension Formula:

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

Where:

- O = Output size (height or width)
- W = Input size
- K = Kernel size
- P = Padding
- S = Stride
- $\lfloor \cdot \rfloor$ = Floor function

Solved Example 2: Output Size Calculations

Given Data:
Calculate the output dimensions for the following configurations:

Case	Input	Kernel	Padding	Stride
A	32×32	3×3	0	1
B	32×32	3×3	1	1
C	32×32	5×5	0	1
D	224×224	7×7	3	2

Solution:
Using formula: $O = \frac{W-K+2P}{S} + 1$

Case A: $O = \frac{32-3+2(0)}{1} + 1 = \frac{29}{1} + 1 = \mathbf{30}$

Case B: $O = \frac{32-3+2(1)}{1} + 1 = \frac{32-3+2}{1} + 1 = \frac{31}{1} + 1 = \mathbf{32}$ (same padding!)

Case C: $O = \frac{32-5+2(0)}{1} + 1 = \frac{27}{1} + 1 = \mathbf{28}$

Case D: $O = \frac{224-7+2(3)}{2} + 1 = \frac{224-7+6}{2} + 1 = \frac{223}{2} + 1 = 111.5 + 1 = \mathbf{112}$

Answers:

Case	Output Size
A	30×30
B	32×32 (same as input)
C	28×28
D	112×112

Key insight: For "same" padding with stride 1, use $P = \frac{K-1}{2}$ (only works for odd K).

Solved Example 3: Convolution with Padding

Given Data:

- Same 5×5 input image from Example 1
- Same 3×3 kernel
- Padding = 1 (zero padding)
- Stride = 1

Task: What is the output size? Calculate output[0,0].

Solution:

Step 1: Add zero padding (border of zeros)

Padded input becomes 7×7:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 & 3 & 2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Step 2: Calculate output size

$$O = \frac{5 + 2(1) - 3}{1} + 1 = \frac{7 - 3}{1} + 1 = 5 \times 5$$

Step 3: Calculate output[0,0]

Top-left 3×3 of padded image: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$

Convolution: $0(1) + 0(0) + 0(-1) + 0(1) + 1(0) + 2(-1) + 0(1) + 0(0) + 1(-1) = -2 - 1 = -3$

Answer:

- Output size: **5×5** (same as input!)
- output[0,0] = -3

5 Pooling Layers

Why It Matters

Pooling layers reduce the spatial dimensions of feature maps, making the network more computationally efficient and providing some translation invariance.

5.1 Types of Pooling

Definition

Pooling applies a summary operation over local regions to downsample the feature map.
Common Types:

- **Max Pooling:** Takes the maximum value in each region
- **Average Pooling:** Takes the average of values in each region
- **Global Average Pooling:** Averages entire feature map to single value

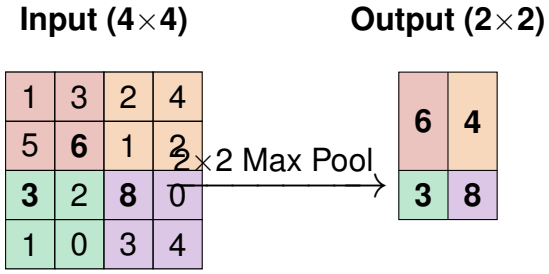


Figure 6: Max Pooling (2×2, stride=2): Takes maximum from each colored region

Key Formula

Pooling Operations:
Max Pooling:

$$\text{output}[i,j] = \max_{m,n \in \text{window}} \text{input}[i \cdot s + m, j \cdot s + n]$$

Average Pooling:

$$\text{output}[i,j] = \frac{1}{k^2} \sum_{m,n \in \text{window}} \text{input}[i \cdot s + m, j \cdot s + n]$$

Output Size: Same formula as convolution: $O = \frac{W-K}{S} + 1$

Solved Example 4: Max Pooling

Given Data:
Input Feature Map (4×4):

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

Pooling: 2×2 Max Pooling, Stride = 2

Solution:
Step 1: Calculate output size

$$O = \frac{4-2}{2} + 1 = 2 \times 2$$

Step 2: Apply max pooling to each 2×2 region

Top-left (rows 0-1, cols 0-1): $\begin{bmatrix} 1 & 3 \\ 5 & 6 \end{bmatrix} \rightarrow \max(1,3,5,6) = \mathbf{6}$

Top-right (rows 0-1, cols 2-3): $\begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix} \rightarrow \max(2,4,1,2) = \mathbf{4}$

Bottom-left (rows 2-3, cols 0-1): $\begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} \rightarrow \max(3,2,1,0) = \mathbf{3}$

Bottom-right (rows 2-3, cols 2-3): $\begin{bmatrix} 8 & 0 \\ 3 & 4 \end{bmatrix} \rightarrow \max(8,0,3,4) = \mathbf{8}$

Output after Max Pooling:

$$\boxed{\begin{bmatrix} 6 & 4 \\ 3 & 8 \end{bmatrix}}$$

Note: The spatial dimensions reduced from 4×4 to 2×2 (factor of 2 reduction).

Solved Example 5: Average Pooling

Given Data: Same 4×4 input as Example 4, but using Average Pooling.

Solution:
Top-left: $\frac{1+3+5+6}{4} = \frac{15}{4} = \mathbf{3.75}$
Top-right: $\frac{2+4+1+2}{4} = \frac{9}{4} = \mathbf{2.25}$
Bottom-left: $\frac{3+2+1+0}{4} = \frac{6}{4} = \mathbf{1.5}$
Bottom-right: $\frac{8+0+3+4}{4} = \frac{15}{4} = \mathbf{3.75}$

Output after Average Pooling:

$$\begin{bmatrix} 3.75 & 2.25 \\ 1.5 & 3.75 \end{bmatrix}$$

5.2 Max Pooling vs Average Pooling

Aspect	Max Pooling	Average Pooling
Operation	Takes maximum value	Takes average of values
Effect	Preserves strongest activation	Smooths activations
Translation invariance	Strong	Moderate
Common use	Hidden layers of CNNs	Often at final layer (GAP)
Information preserved	Dominant features	All features equally

Table 2: Comparison of Max Pooling and Average Pooling

6 Putting It All Together: CNN Architecture

Why It Matters

A complete CNN combines convolutional layers, pooling layers, and fully connected layers to transform an input image into a class prediction.

6.1 Typical CNN Architecture

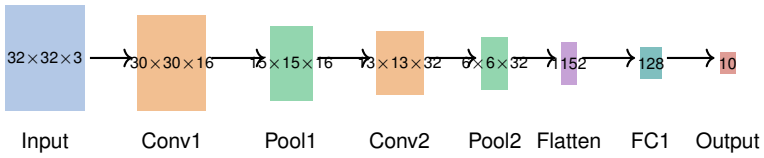


Figure 7: Typical CNN architecture for image classification (e.g., CIFAR-10)

Solved Example 6: Complete CNN Dimension Calculation

Given CNN Architecture:

1. Input: 32×32×3 (RGB image)
2. Conv1: 16 filters, 5×5 kernel, padding=0, stride=1
3. Pool1: 2×2 max pooling, stride=2
4. Conv2: 32 filters, 3×3 kernel, padding=0, stride=1
5. Pool2: 2×2 max pooling, stride=2
6. Flatten
7. FC1: 128 neurons
8. Output: 10 classes

Task: Calculate dimensions at each layer and total parameters.

Solution:
After Conv1:

- Size: $\frac{32-5+0}{1} + 1 = 28$
- Output: $28 \times 28 \times 16$
- Parameters: $(5 \times 5 \times 3 + 1) \times 16 = 76 \times 16 = \mathbf{1,216}$

After Pool1:

- Size: $\frac{28-2}{2} + 1 = 14$
- Output: $14 \times 14 \times 16$
- Parameters: 0 (pooling has no learnable parameters)

After Conv2:

- Size: $\frac{14-3+0}{1} + 1 = 12$
- Output: $12 \times 12 \times 32$
- Parameters: $(3 \times 3 \times 16 + 1) \times 32 = 145 \times 32 = \mathbf{4,640}$

After Pool2:

- Size: $\frac{12-2}{2} + 1 = 6$
- Output: $6 \times 6 \times 32 = 1,152$ (flattened)

FC1: $1,152 \times 128 + 128 = \mathbf{147,584}$ parameters
Output: $128 \times 10 + 10 = \mathbf{1,290}$ parameters

Summary:

Layer	Output Shape	Parameters
Input	32×32×3	0
Conv1	28×28×16	1,216
Pool1	14×14×16	0
Conv2	12×12×32	4,640
Pool2	6×6×32	0
Flatten	1,152	0
FC1	128	147,584
Output	10	1,290
Total Parameters:		**154,730**

Key insight: Most parameters are in the fully connected layers! This is why Global Average Pooling is used in modern architectures.

7 Summary

Key Takeaways

1. Why CNNs for Spatial Data

- Local connectivity: only connect to nearby pixels
- Parameter sharing: same kernel applied everywhere
- Translation invariance: detect features anywhere

2. Convolutional Layers

- Slide kernel over input, compute element-wise multiplication + sum
- Learn kernels that detect edges, textures, patterns

3. Output Size Formula

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

4. Pooling Layers

- Max Pooling: Take maximum (most common)
- Average Pooling: Take average
- Reduces spatial dimensions by factor of stride

5. Parameter Count

- Conv layer: $(K \times K \times C_{in} + 1) \times C_{out}$
- Pooling: 0 parameters
- FC layer: $(N_{in} + 1) \times N_{out}$

Self-Test — Check Your Understanding

Quick Quiz:

1. What is the output size of a 64×64 input with 3×3 kernel, padding=1, stride=2?
2. How many parameters in a conv layer: 3 input channels, 32 output channels, 5×5 kernel?
3. What is the output of 2×2 max pooling on $\begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix}$?

Answers:

1. $O = \frac{64-3+2}{2} + 1 = \frac{63}{2} + 1 = 32$ (32×32)
2. $(5 \times 5 \times 3 + 1) \times 32 = 76 \times 32 = 2,432$
3. max(1, 4, 2, 3) = 4

8 Glossary

Term	Definition
Convolution	Sliding window operation that detects local patterns
Kernel/Filter	Small learnable weight matrix applied in convolution
Feature Map	Output of a convolutional layer
Stride	Step size for sliding the kernel
Padding	Adding pixels (usually zeros) around input border
Max Pooling	Takes maximum value in each local region
Average Pooling	Takes average value in each local region
Receptive Field	Region of input that affects a single output neuron
Translation Invariance	Detecting features regardless of position

9 Coming Up Next

Preview: Advanced CNN Topics

Week 08: Object Detection

- YOLO (You Only Look Once)
- Semantic Segmentation

Week 09: Deep CNN Architectures

- AlexNet, VGGNet
- InceptionNet (GoogLeNet)
- EfficientNet
- Fine-tuning pre-trained models
- ImageNet dataset