Welcome to Computer Vision

Computer Vision

Dr. Muhammad Tahir

DEPARTMENT OF COMPUTER SCIENCE,

FAST-NUCES, Peshawar

Course Details

LECTURES: Monday & Wednesday

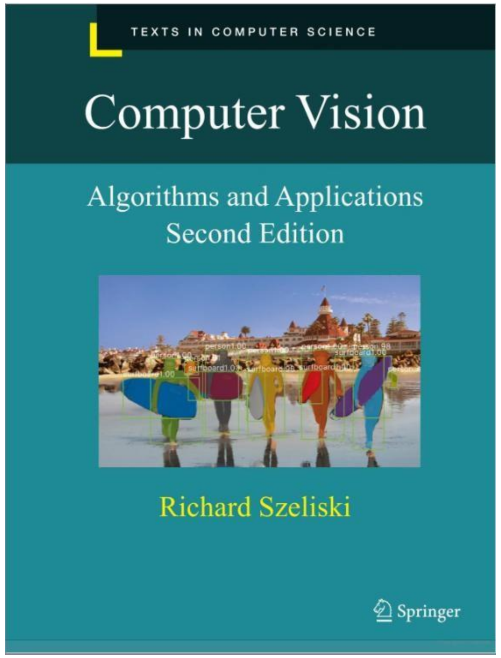
TIMINGS:

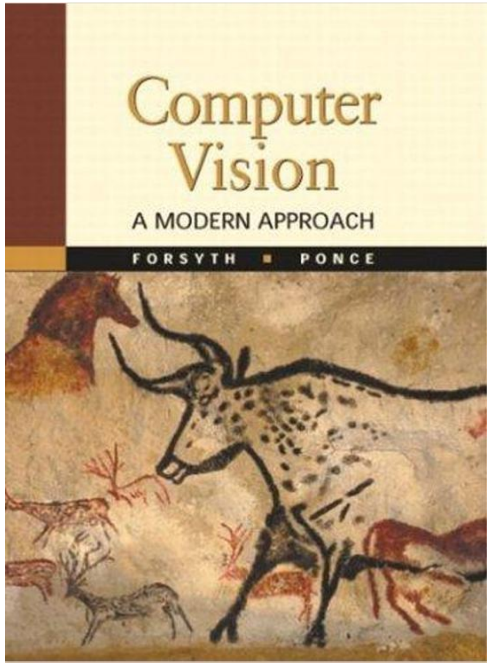
9:30 am – 11:00 am

MY OFFICE:

OFFICE HOURS:

EMAIL: m.tahir@nu.edu.pk





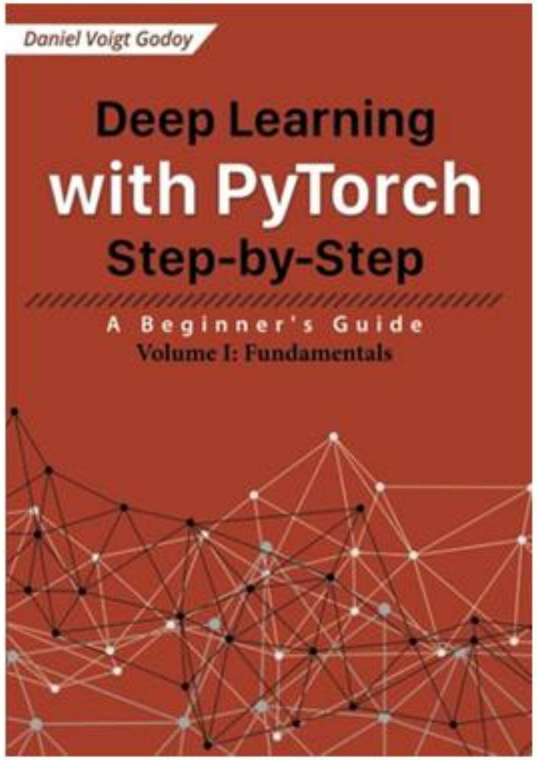
References

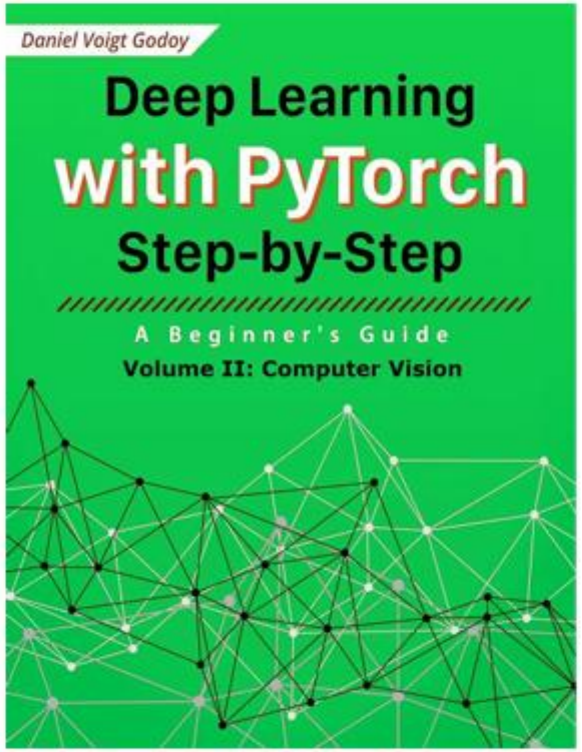
The material in these slides are based on:

1. Rick Szeliski's book: Computer Vision: Algorithms and Applications
2. Forsythe and Ponce: Computer Vision: A Modern Approach

Recommended Books

Deep Learning with PyTorch Step-by-Step by Daniel Voigt Godoy





Course Learning Outcomes

No	CLO	Domain	Taxonomy Level	PLO
1	Understand the view geometry concepts, multi-scale representation, edge detection and detection of other primitives, stereo, motion and object recognition.	Cognitive		
2	Assess which methods to use for solving a given problem, and analyse the accuracy of the methods Skills	Cognitive		
3	Apply appropriate image processing methods for image filtering, image restoration, image reconstruction, segmentation, classification and representation	Cognitive		

Outline

Deep Learning Fundamentals

[Deep Learning Tutorial - GeeksforGeeks]

AlexNet

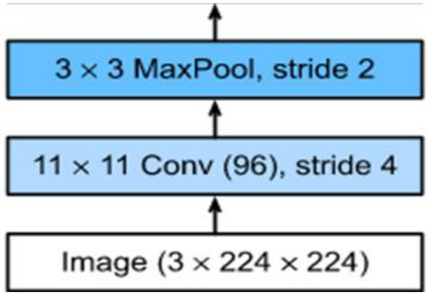
AlexNet

- Classify a 224×224 RGB image into 1000 ImageNet classes.
- Five convolutional stages → three fully-connected (FC) layers → softmax.
- ReLU (instead of tanh), **overlapping** max-pooling, **data augmentation** + heavy dropout, and training split across **two GPUs** with “grouped” convolutions.
- ~61M parameters (most are in the two 4096-unit FC layers).

AlexNet Architecture (224 × 224 Input Version)

Stage 1

1. **Conv1:** 96 filters, **11×11**, stride **4**, padding **2**
   Output: 55×55×96
   Activation: **ReLU**
   Followed by **LRN** (local response normalization)
2. **MaxPool1:** 3×3, stride 2 (overlapping pooling)
   Output: 27×27×96



Note: Standard AlexNet used 227×227 input size

AlexNet Architecture (224 × 224 Input Version)

Stage 2

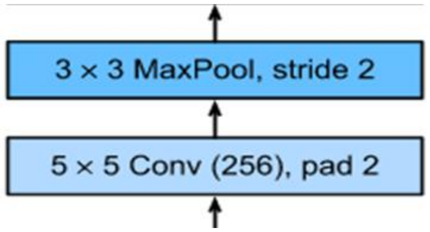
3. **Conv2:** 256 filters, **5×5**, stride 1, **pad 2**

Output: 27×27×256

Activation: ReLU

LRN

4. **MaxPool2:** 3×3, stride 2 → 13×13×256



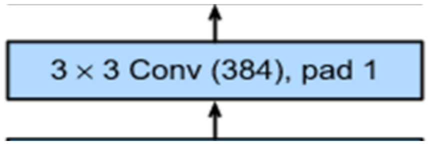
AlexNet Architecture (224 × 224 Input Version)

Stage 3

5. **Conv3:** 384 filters, **3×3**, stride 1, pad 1

Output: 13×13×384

Activation: ReLU



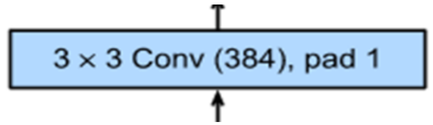
AlexNet Architecture (224 × 224 Input Version)

Stage 4

6. **Conv4:** 384 filters, **3×3**, stride 1, pad 1

Output: 13×13×384

Activation: ReLU



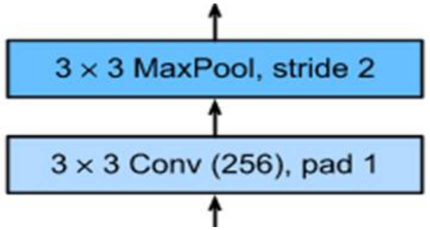
AlexNet Architecture (224 × 224 Input Version)

Stage 5

7. Conv5: 256 filters, **3×3**, stride 1, pad 1

Output: 13×13×256

Activation: ReLU



8. MaxPool3: 3×3, stride 2 → 6×6×256

AlexNet Architecture (224 × 224 Input Version)

Flatten Layer

Input: $6 \times 6 \times 256 = 9216$

$\rightarrow$ flattened into a 1D vector of length **9216**

AlexNet Architecture (224 × 224 Input Version)

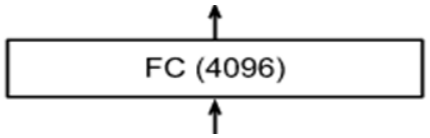
9. **Fully Connected Layer 1**

   Input units: 9216

   Output units: 4096

   Activation: ReLU

   Dropout: 0.5



AlexNet Architecture (224 × 224 Input Version)

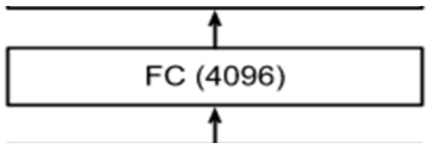
10. Fully Connected Layer 2

Input units: 4096

Output units: 4096

Activation: ReLU

Dropout: 0.5



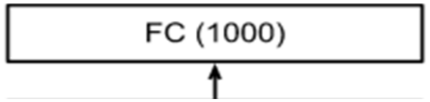
AlexNet Architecture (224 × 224 Input Version)

11. Fully Connected Layer 3 (Output Layer)

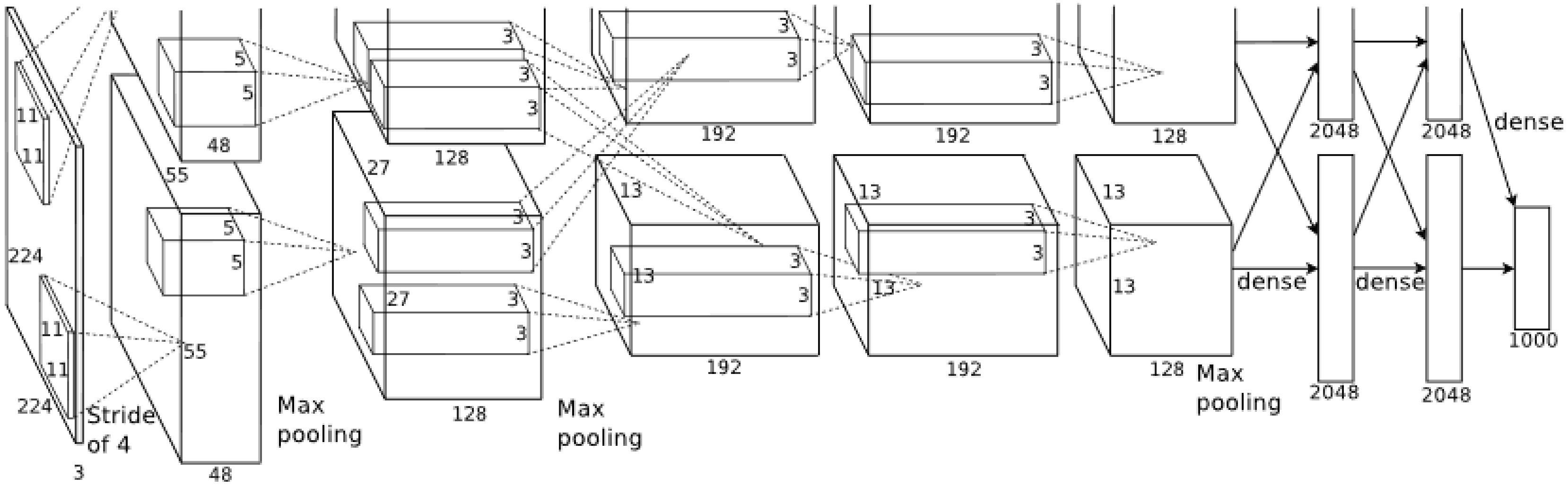
Input units: 4096

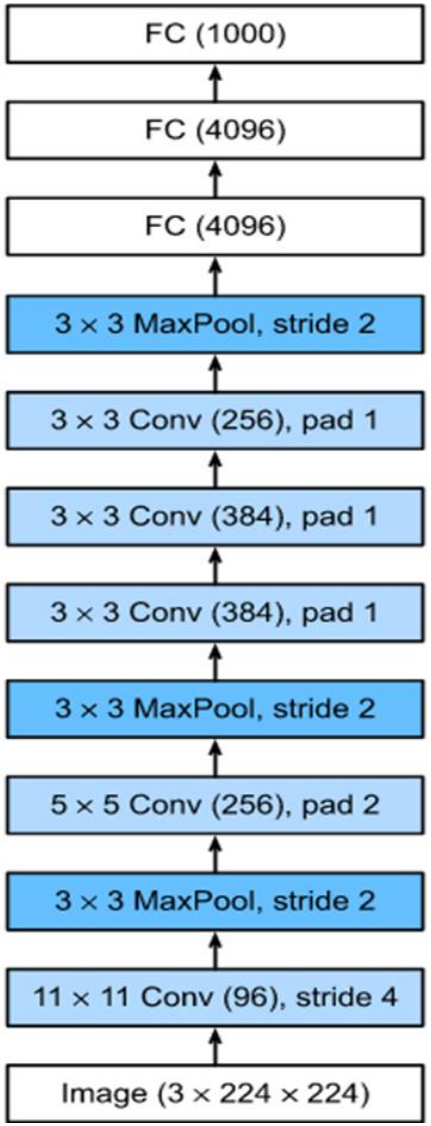
Output units: 1000

Activation: Softmax (for 1000 ImageNet classes)



AlexNet





Alex Krizhevsky et al.: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Normalization & activations

- **ReLU** dramatically sped up convergence compared to tanh/sigmoid used pre-2012.
- **LRN** (across channels) encouraged competition between features; it's **obsolete** now—**BatchNorm** is better.

Training setup (2012, from the paper)

- **SGD with momentum** 0.9, weight decay 5e-4
- Initial LR ~0.01, dropped ×10 on plateau.
- **Batch size 128.**
- **Data augmentation:** random crops and horizontal flips; "**PCA color jitter**" (additive RGB lighting noise).
- **Dropout 0.5** on FC6 and FC7 to curb overfitting.

Layer	Type	Filter / Kernel	Stride	Padding	Output Volume	Activation
1	Conv	11×11, 96 filters	4	2	55×55×96	ReLU
2	Max Pool	3×3	2	0	27×27×96	-
3	Conv	5×5, 256 filters	1	2	27×27×256	ReLU
4	Max Pool	3×3	2	0	13×13×256	-
5	Conv	3×3, 384 filters	1	1	13×13×384	ReLU
6	Conv	3×3, 384 filters	1	1	13×13×384	ReLU
7	Conv	3×3, 256 filters	1	1	13×13×256	ReLU
8	Max Pool	3×3	2	0	6×6×256	-
9	FC	-	-	-	4096	ReLU + Dropout
10	FC	-	-	-	4096	ReLU + Dropout
11	FC	-	-	-	1000	Softmax

AlexNet Architecture (224 × 224 Input Version)

- **Conv1** learns large edge and color blobs (coarse features).
- **Conv2–Conv5** learn increasingly complex features — corners, textures, object parts.
- **FC6 & FC7** act as high-level feature combiners.
- **FC8 (Softmax)** outputs the final classification probabilities.

VGG

VGG-16 Architecture (224×224 Input Version)

Input

Image size: $224 \times 224 \times 3$

- VGG was designed to show that using **very small filters (3×3)** repeatedly can be as effective as larger filters (like 7×7 or 11×11), but with fewer parameters and more non-linearities.

VGG-16 Architecture (224 × 224 Input Version)

Conv Block 1

- **Conv1_1:** 64 filters, 3×3, stride 1, pad 1
  - **Input:** 224×224×3 → **Output:** 224×224×64
  - **Activation:** ReLU
- **Conv1_2:** 64 filters, 3×3, stride 1, pad 1
  - **Output:** 224×224×64
  - **Activation:** ReLU
- **MaxPool1:** 2×2, stride 2
  - **Output:** 112×112×64

VGG-16 Architecture (224 × 224 Input Version)

Conv Block 2

- **Conv2_1:** 128 filters, 3×3, stride 1, pad 1
  - **Input:** 112×112×64 → **Output:** 112×112×128
  - **Activation:** ReLU
- **Conv2_2:** 128 filters, 3×3, stride 1, pad 1
  - **Output:** 112×112×128
  - **Activation:** ReLU
- **MaxPool2:** 2×2, stride 2
  - **Output:** 56×56×128

VGG-16 Architecture (224 × 224 Input Version)

Conv Block 3

- **Conv3_1:** 256 filters, 3×3, stride 1, pad 1
  - **Input:** 56×56×128 → **Output:** 56×56×256
  - **Activation:** ReLU
- **Conv3_2:** 256 filters, 3×3, stride 1, pad 1
  - **Output:** 56×56×256
  - **Activation:** ReLU
- **Conv3_3:** 256 filters, 3×3, stride 1, pad 1
  - **Output:** 56×56×256
  - **Activation:** ReLU
- **MaxPool3:** 2×2, stride 2
  - **Output:** 28×28×256

VGG-16 Architecture (224 × 224 Input Version)

Conv Block 4

Conv4_1: 512 filters, 3×3, stride 1, pad 1

Input: 28×28×256 → **Output:** 28×28×512

Activation: ReLU

Conv4_2: 512 filters, 3×3, stride 1, pad 1

Output: 28×28×512

Activation: ReLU

Conv4_3: 512 filters, 3×3, stride 1, pad 1

Output: 28×28×512

Activation: ReLU

MaxPool4: 2×2, stride 2

Output: 14×14×512

VGG-16 Architecture (224 × 224 Input Version)

Conv Block 5

Conv5_1: 512 filters, 3×3, stride 1, pad 1

Input: 14×14×512 → **Output:** 14×14×512

Activation: ReLU

Conv5_2: 512 filters, 3×3, stride 1, pad 1

Output: 14×14×512

Activation: ReLU

Conv5_3: 512 filters, 3×3, stride 1, pad 1

Output: 14×14×512

Activation: ReLU

MaxPool5: 2×2, stride 2

Output: 7×7×512

VGG-16 Architecture (224 × 224 Input Version)

Fully Connected Layers

- **Flatten:** $7 \times 7 \times 512 = 25088$features
  - **FC6:** 25088 → 4096
    - **Activation:** ReLU, Dropout (0.5)
  - **FC7:** 4096 → 4096
    - **Activation:** ReLU, Dropout (0.5)
  - **FC8:** 4096 → 1000
    - **Activation:** Softmax (ImageNet classes)

VGG-16 Architecture (224 × 224 Input Version)

Training Summary

Total parameters: ≈ 138 million

Optimizer: SGD + Momentum (0.9)

Batch size: 256

Weight decay: 5×10^{-4}

Training time: ≈ 2–3 weeks on 4 GPUs

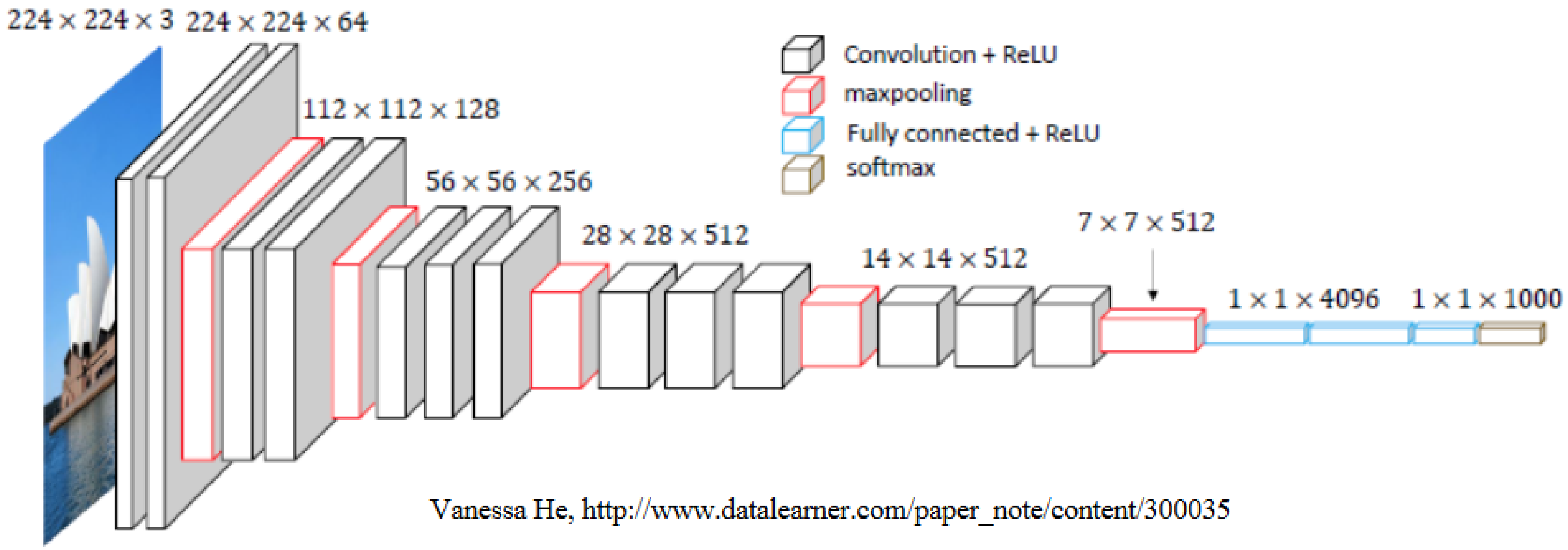
Data augmentation: random cropping, horizontal flips, RGB jitter

VGG-16 Architecture (224 × 224 Input Version)

Design Philosophy

- Only **3×3 filters** (instead of AlexNet's mixed sizes).
- **Deeper network** (16 layers with weights).
- Each max-pool halves spatial size.
- **ReLU after every convolution.**
- No LRN — replaced with deeper stacking.
- Large parameter count dominated by FC layers.

VGG-16 Architecture (224 × 224 Input Version)



- 138 million parameters
- Training: 2-3 week (4 GPUs)

Block	Layers	Output Volume
Input	RGB image	224×224×3
Conv Block 1	2 × (3×3 conv, 64 filters) + MaxPool	112×112×64
Conv Block 2	2 × (3×3 conv, 128 filters) + MaxPool	56×56×128
Conv Block 3	3 × (3×3 conv, 256 filters) + MaxPool	28×28×256
Conv Block 4	3 × (3×3 conv, 512 filters) + MaxPool	14×14×512
Conv Block 5	3 × (3×3 conv, 512 filters) + MaxPool	7×7×512
Flatten	—	25088 (7×7×512)
FC6	Fully Connected (4096) + ReLU + Dropout	4096
FC7	Fully Connected (4096) + ReLU + Dropout	4096
FC8	Fully Connected (1000) + Softmax	1000

ResNet

ResNet (Residual Network – ResNet-152)

ResNet (Residual Network) was introduced by **Kaiming He et al., 2015** in the paper *"Deep Residual Learning for Image Recognition"* (arXiv:1512.03385).

- It **revolutionized deep learning** by enabling networks with **over 100 layers** to train effectively — solving the **vanishing gradient problem** using *skip (shortcut) connections*.

ResNet (Residual Network – ResNet-152)

Property	Description
Input image	224×224×3
Layers (ResNet-152)	152 (with 50, 101, and 152 variants)
Parameters	~60 million
Training time	2–3 weeks (on 8 GPUs, 2015 standard)
Building blocks	7×7 conv, 3×3 conv, BatchNorm, Max/Average Pooling, Skip connections
Optimizer	SGD + Momentum

ResNet (Residual Network – ResNet-152)

ResNet comes in two types:

- **Basic Block** – used in shallower models (ResNet-18, ResNet-34)
  - Conv3x3 → BN → ReLU → Conv3x3 → BN → Add skip connection → ReLU
- **Bottleneck Block** – used in deeper models (ResNet-50, ResNet-101, ResNet-152)
  - Conv1x1 → BN → ReLU → Conv3x3 → BN → ReLU → Conv1x1 → BN → Add skip connection → ReLU

The **bottleneck** reduces and then restores the number of channels to make training faster.

ResNet (Residual Network – ResNet-152)

Stage 1: Initial Layers

- **Conv1:** 64 filters, 7×7, stride 2, pad 3
  - **Output:** 112×112×64
  - **Activation:** ReLU
  - **Batch Normalization** applied
- **MaxPool:** 3×3, stride 2
  - **Output:** 56×56×64

ResNet (Residual Network – ResNet-152)

Stage 2: Conv2_x (3 Bottleneck Blocks)

Each block:

1×1, 64 filters → 3×3, 64 filters → 1×1, 256 filters

Shortcut: identity or 1×1 projection

Output: 56×56×256

ResNet (Residual Network – ResNet-152)

Stage 3: Conv3_x (8 Bottleneck Blocks)

Each block:

1×1, 128 filters → 3×3, 128 filters → 1×1, 512 filters

First block stride = 2 (reduces spatial size)

Output: 28×28×512

ResNet (Residual Network – ResNet-152)

Stage 4: Conv4_x (36 Bottleneck Blocks)

Each block:

1×1, 256 filters → 3×3, 256 filters → 1×1, 1024 filters

First block stride = 2

Output: 14×14×1024

ResNet (Residual Network – ResNet-152)

Stage 5: Conv5_x (3 Bottleneck Blocks)

Each block:

1×1, 512 filters → 3×3, 512 filters → 1×1, 2048 filters

First block stride = 2

Output: 7×7×2048

ResNet (Residual Network – ResNet-152)

Global Average Pooling

Averages each feature map (7×7) into a single value.

Output: 1×1×2048

ResNet (Residual Network – ResNet-152)

Fully Connected Layer

- **Input:** 2048
- **Output:** 1000
- **Activation:** Softmax (ImageNet classes)

ResNet (Residual Network – ResNet-152)

Residual Block Example (Inside the Network)
Let’s illustrate one **bottleneck residual block:**

Layer	**Type**	**Filters**	**Kernel**	**Stride**	**Output Size**	**Skip Connection**
1	Conv + BN + ReLU	64	1×1	1	56×56×64	Identity
2	Conv + BN + ReLU	64	3×3	1	56×56×64	—
3	Conv + BN	256	1×1	1	56×56×256	—
Add	—	—	—	—	56×56×256	($y = F(x) + x$)
Output	ReLU	—	—	—	56×56×256	—

ResNet (Residual Network – ResNet-152)

Stage	Output Size	Block Type	#Blocks	Parameters
Conv1	112×112×64	7×7, 64	1	—
Conv2_x	56×56×256	Bottleneck	3	—
Conv3_x	28×28×512	Bottleneck	8	—
Conv4_x	14×14×1024	Bottleneck	36	—
Conv5_x	7×7×2048	Bottleneck	3	—
AvgPool + FC	1×1×1000	—	—	—

ResNet (Residual Network – ResNet-152)

Feature	Description
Skip connections	Enable training of >100 layers easily
Batch Normalization	Stabilizes and speeds up convergence
Global Average Pooling	Reduces overfitting (no large FC layers like VGG)
Bottleneck structure	Efficient depth expansion
ReLU everywhere	Non-linearity after each conv
Identity mapping	Keeps features stable through depth

Why ResNet Succeeded

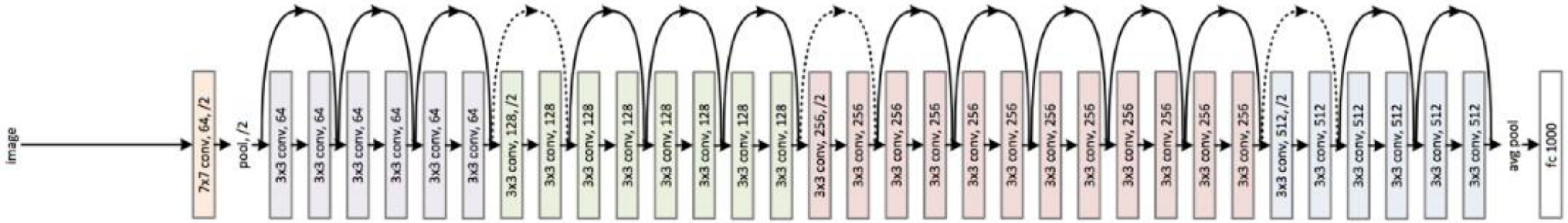
Deep networks can be trained **without degradation**.

Each block learns **only the residual difference**.

Skip connections act like **shortcuts for gradients**.

Model achieved **1st place in ILSVRC 2015** with **3.6% Top-5 error**, a huge leap in performance.

ResNet



Kaiming He, https://arxiv.org/pdf/1512.03385.pdf

152 layers!!!

7x7 convolutional layers, 3x3 convolutional layers, batch normalization, max and average pooling.

Parameters: 60 million

Training time: 2-3 weeks (8 GPUs)

1x1 Convolution

Input feature map: $3 \times 3 \times 3$(Height × Width × Channels)

$$R = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, G = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Apply a **1×1 convolution with 2 filters** (so output depth = 2).

Each 1×1 filter has **3 weights** (one per input channel).

Filter A (average-ish): $[0.5\ 0.25\ 0.25]$, $bias\ = 0$

Filter B (color contrast):$[1\ -1\ 0.5]$, $bias\ = 0$

1x1 Convolution

How a 1×1 works at one pixel

Take the **center pixel** (row 2, col 2):

$$(R, G, B) = (5, 2, 0)$$

Filter A output: $0.5 \times 5 + 0.25 \times 2 + 0.25 \times 0 = 2.5 + 0.5 + 0 = 3.0$

Filter B output: $1 \times 5 - 1 \times 2 + 0.5 \times 0 = 3.0$

So, at that pixel the new 2-channel vector is **[3.0, 3.0]**.

Because the kernel is 1×1 (stride 1, no padding), we do this **independently at every spatial location**—it mixes channels but does **not** look at neighbors, and the spatial size stays 3×3.

Compute the full output maps

Since 1×1 is just a per-pixel linear combination across channels, we can write:

$$Y_A = 0.5R + 0.25G + 0.25B, \qquad Y_B = 1 \cdot R - 1 \cdot G + 0.5B$$

Y_A **(from Filter A)**

$$0.5R \quad = \begin{bmatrix} 0.5 & 1 & 1.5 \\ 2 & 2.5 & 3 \\ 3.5 & 4 & 4.5 \end{bmatrix}, \qquad 0.25G = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}, \qquad 0.25B == \begin{bmatrix} 0 & 0.25 & 0 \\ 0.25 & 0 & 0.25 \\ 0 & 0.25 & 0 \end{bmatrix}$$

Add them elementwise:

$$\boxed{Y_A = \begin{bmatrix} 1.00 & 1.75 & 2.00 \\ 2.75 & 3.00 & 3.75 \\ 4.00 & 4.75 & 5.00 \end{bmatrix}}$$

Y_B **(from Filter B)**

$$R - G = \begin{bmatrix} -1 & 0 & 1 \\ 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}, \qquad 0.5B = \begin{bmatrix} 0 & 0.5 & 0 \\ 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0 \end{bmatrix}$$

Add them:

$$\boxed{Y_B = \begin{bmatrix} -1.0 & 0.5 & 1.0 \\ 2.5 & 3.0 & 4.5 \\ 5.0 & 6.5 & 7.0 \end{bmatrix}}$$

Final result (shape change)

Input: $3 \times 3 \times 3$

Output: $3 \times 3 \times 2$ (the two maps Y_A and Y_B stacked)

So, a **1×1 conv with 2 filters** *shrinks channels* from **3** → **2** while keeping the spatial size **3×3 unchanged**.

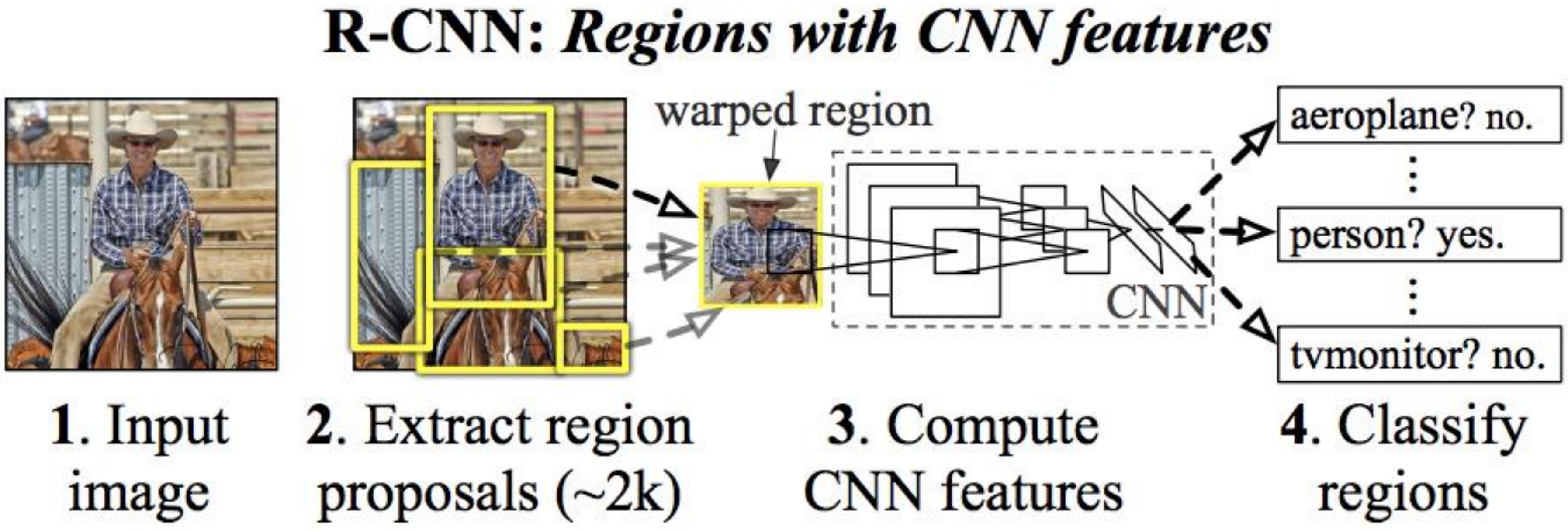
RCNN: Region-based Convolutional Neural Network

Motivation for RCNN

- Before R-CNN (2013–2014), object detection relied on:
  - **Traditional methods:** sliding windows, HOG, SIFT, and handcrafted features.
    - These methods were slow and had poor accuracy on complex datasets (e.g., PASCAL VOC).
- R-CNN, proposed by **Ross Girshick et al. (CVPR 2014)**, was the first to **combine region proposals with deep CNN feature extraction**, bringing deep learning to detection.

RCNN

- The model first extracts *region proposals* (candidate bounding boxes), then applies a **CNN** on each region to classify it and refine its bounding box.



RCNN

1. **Combining Region Proposals + CNN Features**
   - Earlier CNNs (like AlexNet) worked only on whole images.
   - R-CNN applied CNNs to *regions*, bringing **deep features** into **object localization**.
2. **Decoupling Detection Stages**
   - R-CNN separated detection into 3 independent parts:
     1. Proposal generation
     2. CNN-based feature extraction
     3. Classification + regression
   - This modularity made it flexible but also computationally heavy.

RCNN

3. Fine-Tuning

- After pretraining the CNN on ImageNet classification, R-CNN fine-tuned it for detection using proposed regions labeled as positive/negative examples.
- This was one of the first demonstrations of **transfer learning** in object detection.

4. Multi-Stage Training

- R-CNN required **three separate training stages**:
    1. Train CNN on region proposals.
    2. Train SVMs for classification.
    3. Train bounding box regressors.
- These were done independently — time-consuming and complex.

R-CNN Architecture

Step 1 — Input Image

Start with the full image (e.g., 600×1000 pixels).

R-CNN Architecture

Step 2 — Region Proposal Generation

- Use a **Selective Search** algorithm (not deep learning!) to generate around **2000 candidate regions (RoIs)** likely to contain objects.
  - It merges pixels based on color, texture, and edges.
  - Outputs bounding boxes of varying sizes/aspect ratios.
- *These are "object proposals."*

R-CNN Architecture

Step 3 — Region Warping

- Each region proposal is:
  - Cropped from the original image.
  - **Warped (resized)** to a fixed size (e.g., 227×227 for AlexNet).
- This ensures each region can be fed into the CNN.

R-CNN Architecture

Step 4 — Feature Extraction (CNN)

- Each warped region is passed through a **pre-trained CNN** (like AlexNet, VGG16, etc.) to extract a **feature vector**.
  - The CNN is used only **as a feature extractor**.
  - Usually, the output is taken from one of the **fully connected layers (e.g., FC7)** → a **4096-dimensional vector**.
- So, for 2000 regions → 2000 feature vectors.

R-CNN Architecture

Step 5 — Classification (SVM)

- The CNN features for each region are then classified using **class-specific SVMs**.
  - One SVM per object class (e.g., dog, cat, car, etc.).
  - SVM decides whether the region contains that object or background.
- This step replaces the softmax classifier of the CNN.

R-CNN Architecture

Step 6 — Bounding Box Regression

- For each detected region, a **bounding box regressor** fine-tunes the coordinates of the box to better match the object boundaries.
- This regression is trained separately using the CNN features.

R-CNN Architecture

Step 7 — Non-Maximum Suppression (NMS)

- Multiple boxes may predict the same object → NMS keeps the highest-confidence one and suppresses overlapping duplicates.

Architecture Summary

Stage	Component	Description
1	Selective Search	Generate ~2000 object proposals
2	Warp each region	Resize each to fixed size
3	CNN (AlexNet/VGG)	Extract deep features (e.g., 4096-d)
4	SVM Classifiers	Classify region features into object classes
5	Bounding Box Regressor	Adjust region coordinates
6	NMS	Remove redundant boxes

Thank you