

OBJECT DETECTION — DETAILED NOTES (FROM THE VERY BEGINNING)

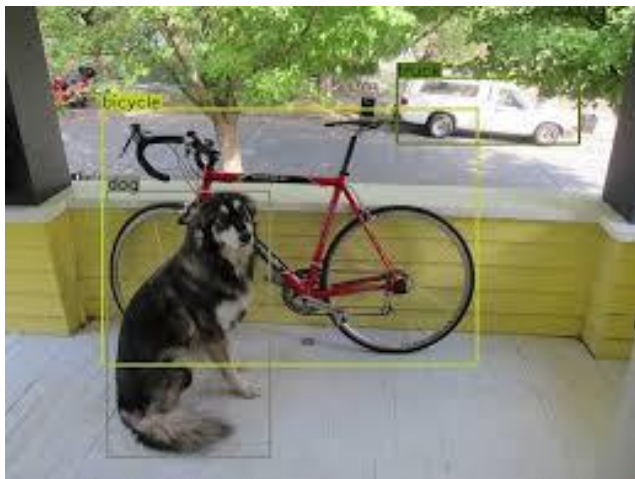
(Explained in simple, clear, step-by-step style)

1. What is Object Detection?

Step 1: Understand the Goal

Object detection is a Computer Vision task where a model:

- **Detects** objects in an image
- **Classifies** each detected object
- **Locates** each object using **bounding boxes**



So the output is:

- *What* object is present
- *Where* it is in the image

Example output:

- Car → bounding box
- Person → bounding box
- Dog → bounding box

Step 2: How It Differs From Classification

Task	Output	Limitation
------	--------	------------

Task	Output	Limitation
Image Classification	One label for whole image	Can't say <i>where</i> the object is
Object Detection	Multiple objects + location	Harder to compute

Step 3: How It Differs From Semantic Segmentation

Task	Output
Object Detection	Bounding box around object
Semantic Segmentation	Pixel-wise mask (more precise)

2. When Do We Use Object Detection?

Object detection is used wherever we want *location* + *category* information.
Examples:

- Self-driving cars
 - Face detection
 - Pedestrian detection in surveillance
 - Object counting in retail
 - Animal detection in drone videos
 - Medical imaging (tumor localization)
-

3. The Problem with Direct Classification

Step 1: Why classification fails

In classification, the model sees the whole image as one class.
If the image contains:

- A cat
- A dog
- A car

Classification can only output ONE label — this is useless.

Step 2: We need “multi-object” recognition

We need:

- multiple objects
- at different scales

- at different positions
- even overlapping

This is where object detection starts.

4. Early Approaches (Before Deep Learning)

Before CNNs, object detection was based on:

- **Hand-crafted features**
- **Sliding windows**
- **Traditional ML classifiers**

We cover these step-by-step.

4.1 Sliding Window Approach (First Major Idea)

Step 1: How it Works

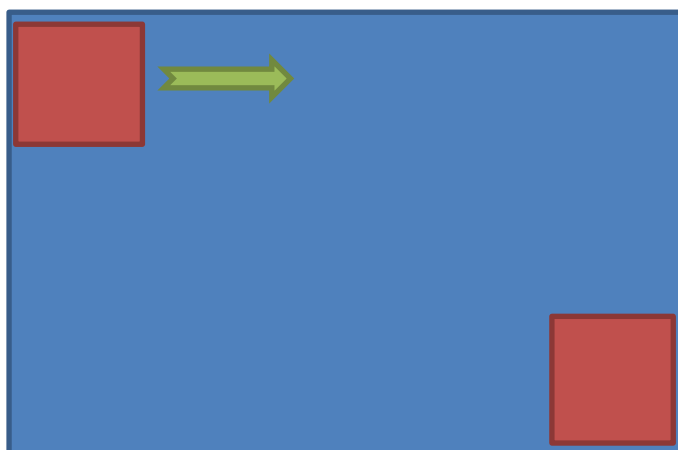
Imagine placing a small window on the image and sliding it over pixel-by-pixel.

For each window:

1. Crop the part of the image
2. Extract features
3. Use a classifier (SVM, AdaBoost, etc.) to say “object or not?”

Step 2: Problems

- **Too slow:** thousands of windows per image
- **Scale variation:** objects are small, medium, large → need different window sizes
- **Class imbalance:** 99% of windows contain nothing
- **Poor accuracy at times**



6. The Deep Learning Era Begins (~2012)

In 2012, **AlexNet** won ImageNet.

CNNs started outperforming all hand-engineered features.

Researchers asked:

❓ *Can we use CNNs for object detection?*

❓ *Can CNNs learn features automatically instead of hand-designing them?*

This created the shift from:

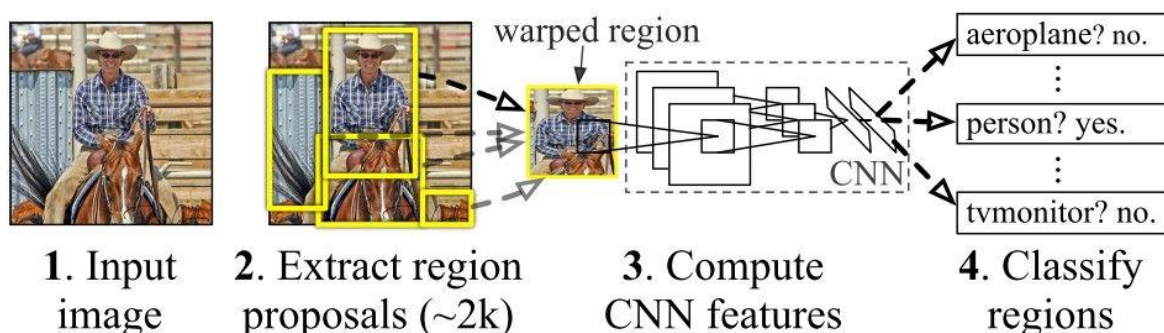
- handcrafted HOG features
to
- CNN feature maps

This leads into:

- ✓ Region-based CNN (R-CNN)
- ✓ Fast R-CNN
- ✓ Faster R-CNN
- ✓ YOLO
- ✓ SSD

But first, we needed one big idea:

7. Region Proposals (Big Breakthrough Before R-CNN)



Step-by-step reasoning

Problem with sliding windows

We were checking millions of windows → extremely slow.

Solution: Region Proposal

Instead of scanning every possible window:

☐ Use an algorithm that proposes *only 1000–2000 possible object regions*.

These methods included:

- Selective Search
- EdgeBoxes
- Objectness measures

This dramatically reduced computation.

Once we had proposal regions...

☐ The next idea:

Use CNN to classify each region → leads to R-CNN.

8. Transition to the Deep Learning Detectors

The flow of evolution becomes:

1. **Sliding window + Haar / HOG**
2. **Region proposals (Selective Search)**
3. **CNN + Regions → R-CNN**
4. **Faster R-CNN** removes selective search
5. **YOLO / SSD** avoid region proposals and directly predict boxes
6. **Transformers (DETR)** avoid anchors and use attention

But for now, we have built the foundation for understanding R-CNN.

❓ R-CNN (Region-Based Convolutional Neural Network)

Introduced in 2014 by Ross Girshick

1. Why Was R-CNN Needed? (The Motivation)

Before R-CNN, object detection used:

- **Sliding windows** → extremely slow
- **HOG, Haar, SIFT features** → manually designed, not learnable
- **Traditional ML classifiers** → not powerful for many object categories

Deep learning revolution started in 2012 (AlexNet), showing **CNNs learn features automatically**.

So researchers asked:

❓ **Can we use CNNs for object detection instead of handcrafted features?**

The problem:

- CNNs need fixed-size input
- Object detection requires variable-size object regions
- Sliding windows with CNN → millions of crops → too slow

Solution: R-CNN = Combine region proposals + CNN.

This is why R-CNN is called a **two-stage detector**:

1. **Stage 1** → Find possible object regions
 2. **Stage 2** → Run a CNN on each region
-

2. The Core Idea of R-CNN

R-CNN takes the image

- generates a small set of possible object locations (1000–2000)
 - runs a CNN on each location
 - classifies the region
 - refines the bounding box.
-

3. The R-CNN Pipeline — Step by Step

Let's break it into the **most important steps**.

Step 1: Input Image

Take the entire image → send it to a region proposal algorithm.

Step 2: Region Proposals (Selective Search)

Before CNN, we need regions likely to contain objects.

R-CNN uses **Selective Search** (a classical algorithm).

What Selective Search does:

1. Starts with small segments (superpixels)
2. Groups similar regions together
3. Produces ~2,000 region proposals

Each proposal is a bounding box *that might* contain an object.

Why this step matters:

- Removes sliding windows
 - Reduces computation from *millions* to ~2,000 windows
-

Step 3: Warp Each Region to a Fixed Size

CNNs like AlexNet require **fixed-size input** (e.g., 227×227).

But region proposals come in many shapes:

- Some tall
- Some wide
- Some square

R-CNN **warps** each proposed region to a fixed input size.

So each region becomes a normalized crop that fits the CNN.

Step 4: Extract Features Using a CNN

For each warped region, we run a CNN forward pass.

CNN outputs a **feature vector** (e.g., 4096 values from FC layers).

This vector represents the object inside that region.

Important:

- Same CNN for all regions
 - CNN is pretrained on ImageNet and fine-tuned for detection tasks
-

Step 5: Classification Using SVMs

CNN features are fed into **separate linear SVM classifiers**.

There is:

- 1 SVM per object class (e.g., car, dog, person)
- 1 background SVM (to reject false regions)

Why SVM?

- At the time, researchers trusted SVMs more for classification.
-

Step 6: Bounding Box Regression

Region proposals are not perfect — sometimes slightly off.

R-CNN trains a **regression model** to adjust:

- box center
- height
- width

It predicts a correction:

$\text{true_box} = \text{predicted_box} + \text{delta}$

This significantly improves localization accuracy.

4. Summary of R-CNN Pipeline (Short Version)

1. **Selective Search** → 2000 boxes
 2. **Warp each box** → fixed size
 3. **Run CNN** → extract features
 4. **Classify with SVM**
 5. **Refine box with regression**
-

5. What Did R-CNN Achieve?

Improvements:

- First model to use **deep learning features** for detection
- Huge accuracy leap vs. HOG/SVM
- Achieved **mAP jump from 33% → 58%** on Pascal VOC
- Introduced the idea of **two-stage detectors**

This was the birth of deep learning–based object detection.

6. Problems with R-CNN

Although it was revolutionary, R-CNN was **very slow**.

Problem 1: 2000 forward passes of CNN per image

If each region is processed separately:

- 2000 crops
- 2000 CNN forward passes

A single image took **40–60 seconds** → unusable in real-time.

Problem 2: Disk storage explosion

For each image, CNN features of all regions were stored on disk. This required **hundreds of GBs**.

Problem 3: Training was a mess (multi-stage)

Three separate training steps:

1. Train the CNN
2. Extract features and train SVMs
3. Train bounding box regressor

Not end-to-end.

Problem 4: Region Proposals were not learned

Selective Search is slow and handcrafted — not learnable.

7. Why R-CNN Was Still Important

Even with its weaknesses, R-CNN introduced:

- Region-based deep learning
- The idea of combining CNN + proposals
- A pipeline that became the foundation for later models

It paved the way for:

- ☐ **Fast R-CNN** (solves speed problem)
 - ☐ **Faster R-CNN** (learns region proposals inside the network)
 - ☐ **YOLO & SSD** (real-time one-stage detectors)
-
- .
-

❓ 2. FAST R-CNN (2015)

(An upgrade to solve R-CNN's biggest problems)

1.1 Why Fast R-CNN Was Needed

R-CNN problems:

- CNN runs **2000 times** per image → extremely slow
- Training is **multi-stage** (CNN → SVM → regressor)
- Features stored on disk → hundreds of GB
- Region crops break spatial continuity

Girshick solved these by changing the pipeline.

1.2 The Key Idea of Fast R-CNN

Process the whole image with CNN only once.

Then:

- share feature maps
- extract regions from feature map instead of cropping original image

This removes the 2000 CNN forwards.

1.3 Fast R-CNN Pipeline Step-by-Step

Step 1: Input Image → Single CNN Forward Pass

Pass the entire image through a CNN (e.g., VGG-16).

This produces a feature map (e.g., $7 \times 7 \times 512$).

One forward pass only.

Step 2: Region Proposals from Selective Search

Still using Selective Search (not deep-learned yet).

It generates ~2000 region proposals.

But now, **we do not crop them**.

Step 3: RoI Pooling Layer (Game-Changer)

Each region proposal is mapped onto the CNN feature map.

RoI Pooling:

- crops the corresponding area from the feature map
- converts it into fixed-size features (e.g., 7×7)
- regardless of region size

RoI Pooling allows:

- variable-size regions \rightarrow fixed-size vectors
 - without warping the original image
-

Step 4: Fully Connected Layers

Every RoI produces a feature vector.

Step 5: Two Outputs from Final Layer

Each region has:

1. **Softmax classifier**
 - probability over object classes + background
 2. **Bounding Box Regressor**
 - one regressor per class
 - predicts improved box coordinates
-

1.4 Advantages of Fast R-CNN

- **One CNN run per image** (not 2000)
- **Much faster** (10× → 20× faster)
- **End-to-end training with a single loss function**
- **Higher accuracy**

But still one big bottleneck:

❓ **Selective Search is slow (2 seconds per image)**

This prevented real-time performance.

Thus the next breakthrough...

❓ 3. FASTER R-CNN (2016)

(Solves the last bottleneck: slow region proposals)

2.1 Why Faster R-CNN Was Needed

Fast R-CNN solved CNN speed, but:

- Selective Search was slow
- Not end-to-end
- Not learnable

Solution:

Let the neural network itself learn to propose object regions.

This led to the **Region Proposal Network (RPN)**.

2.2 Key Idea: Region Proposal Network (RPN)

RPN is a small CNN that:

- slides over the feature map
- predicts possible bounding boxes
- predicts objectness (object vs background)

Using **anchors**, it can generate proposals at many scales/aspects.

This removes Selective Search completely.

2.3 Faster R-CNN Full Pipeline (Step-by-Step)

Step 1: Input Image → CNN Backbone

Same as Fast R-CNN.

Backbone examples:

- VGG16
 - ResNet
 - ResNeXt
-

Step 2: RPN (Region Proposal Network)

RPN takes the shared feature map and for each location:

- predicts **k anchors** (commonly $k = 9$)
- outputs:
 - objectness score
 - bounding box deltas

RPN → typically ~300 top proposals (instead of 2000).

No more Selective Search.

Step 3: RoI Pooling / RoI Align

Proposals from RPN are mapped to the feature map.

Then RoI Pooling (or Align) extracts fixed-size feature vectors.

Step 4: Classification + Regression Head

Similar to Fast R-CNN but using learned proposals.

Output:

- class scores
 - bounding box adjustments
-

2.4 Why Faster R-CNN is Huge

- **Fully end-to-end training**
- **15× faster**
- **Better proposals than Selective Search**
- **Much more accurate**
- Became the backbone of nearly all two-stage detectors

It achieved:

- **70%+ mAP** on VOC
 - **~40 mAP** on COCO (depending on backbone)
-

2.5 Main Weakness

Although fast, still **not real-time** compared to YOLO.

🔗 3. MASK R-CNN (2017)

(Extends Faster R-CNN with segmentation masks)

3.1 Why Mask R-CNN Was Needed

Faster R-CNN gives:

- class
- bounding box

But many tasks require **pixel-level segmentation**, such as:

- medical imaging

- autonomous driving
- human pose estimation
- instance segmentation

Mask R-CNN adds a **parallel branch** that:

- predicts a binary pixel mask
 - for each object
 - in each RoI
-

3.2 Mask R-CNN Architecture Step-by-Step

Step 1: Use Faster R-CNN backbone

Same backbone (e.g., ResNet + FPN)

Step 2: Replace RoI Pooling → RoI Align

RoI Pooling caused misalignment due to rounding.

RoI Align:

- does bilinear interpolation
 - preserves exact spatial alignment
 - vital for pixel-level tasks
-

Step 3: The Mask Branch

For each region:

- 3–4 convolutional layers
- output: **28×28 mask** per class

It is independent from classification and box regression.

Classification head does not influence mask head.

3.3 Why Mask R-CNN Was Revolutionary

- **State-of-the-art accuracy in segmentation**
- **Very flexible**
 - masks
 - keypoints
 - human pose
 - dense annotations
- **Better backbone: FPN** (Feature Pyramid Network)

Mask R-CNN = fastest path to high-quality instance segmentation.

🔍 4. R-FCN (Regional Fully Convolutional Network)

(Alternative to Fast/Faster R-CNN)

Optional but important historically.

4.1 Why R-FCN Was Needed

Faster R-CNN still uses:

- fully connected layers on RoIs
- heavy computation per region

R-FCN tried to reduce per-RoI computation.



yolo-algorithm-1.webp

4.2 Key Idea

Almost all computation becomes shared at the feature map level, not per region.

It uses **position-sensitive score maps**.

No FC layers per RoI → more efficient.

4.3 Result

- Faster than Faster R-CNN
- Slightly lower accuracy
- Used mainly before Mask R-CNN took over

📊 Final Summary Table of All R-CNN Family

Model	Region Proposals	CNN Passes	Speed	Accuracy	Key Innovation
R-CNN (2014)	Selective Search	2000 CNN runs	Very slow	Big improvement vs classical	Deep features + proposals
Fast R-CNN (2015)	Selective Search	1 CNN run	Faster	Better	RoI Pooling + shared CNN
Faster R-CNN (2016)	RPN (learned)	1 CNN run	Much faster	Excellent	RPN replaces Selective Search
Mask R-CNN (2017)	RPN	1 CNN run	Slightly slower	Best	Adds segmentation mask + RoI Align
R-FCN (2016)	RPN	1 CNN run	Fast	Slightly lower	Fully convolutional per-region scoring

📊 YOLO FAMILY — FULL EXPLANATION (v1 to v8)

YOLO = **You Only Look Once**

Created by Joseph Redmon in 2015–16.

The core YOLO philosophy:

📊 **Detection in one pass, on the whole image.**

Not like R-CNN which uses region proposals.

YOLO = one-stage detector

- No region proposal
- No sliding windows
- Predicts bounding boxes + classes **directly from pixels**

❓ YOLOv1 (2016) — The Beginning

1. Why YOLOv1 Was Created

R-CNN family had problems:

- multiple stages
- slow region proposals
- not real-time

Joseph Redmon wanted **real-time detection**.

YOLOv1 speed:

❓ **45 FPS**

Fastest detector at the time.

2. How YOLOv1 Works

Step-by-step:

Step 1: Divide the image into $S \times S$ grid (default $S = 7$)

Example: 7×7 grid = 49 cells.

Step 2: Each grid cell predicts:

- 2 bounding boxes
- 1 class prediction

Step 3: Entire prediction done in ONE forward pass

3. Weaknesses of YOLOv1

- Localization error (poor bounding box accuracy)
- Struggles with small objects
- Grid cell constraint: only 1 object per cell

- Lower mAP compared to Fast/Faster R-CNN

Still revolutionary because of **speed**.

YOLO Pipeline (Explained in Ultra-Simple Words)

We will go from **input image** → **convolutions** → **feature map** → **grid** → **boxes** → **final detections**.

❓ STEP 1 — Input Image Goes Into the CNN

You start with an image, for example:

416 × 416 × 3

YOLO does **NOT** cut this image into S×S grid here.

Instead:

✓ **The whole image goes straight into the backbone CNN**

(backbone = Darknet, CSPDarknet, etc.)

❓ STEP 2 — Convolutions Extract Features

The CNN applies many convolution layers:

- detect edges
- detect textures
- detect shapes
- detect object patterns

After going through these layers:

Input 416×416

↓ □

Becomes something like 13×13 (a small feature map)

❓ IMPORTANT POINT

This **13×13** feature map *is* the **S×S grid** used for object detection.

You do NOT manually divide the original image into 13×13.

Convolution automatically shrinks the image and creates the grid.

❓ STEP 3 — Feature Map = S×S Detection Grid

Example:

- Layer 1: 52×52 grid (small objects)
- Layer 2: 26×26 grid (medium objects)
- Layer 3: 13×13 grid (large objects)

Each grid cell will try to detect objects.

❓ Why does each grid cell correspond to part of the original image?

Because of **receptive field**:

- Each cell in the 13×13 sees a large region of the input
- Each cell in 52×52 sees a smaller region

This is how YOLO detects objects at multiple scales.

❓ STEP 4 — Anchor Boxes

Each grid cell predicts multiple bounding boxes using **anchor boxes**.

Example:

If YOLO uses **3 anchors**, then each cell predicts **3 potential boxes**.

For each anchor YOLO predicts:

- box center (bx, by)
- box width, height (bw, bh)
- objectness
- class scores

So one cell outputs a vector like:

[bx, by, bw, bh, objectness, classes...] * anchors

🔍 STEP 5 — YOLO Predicts Box Coordinates

YOLO does not output coordinates directly.

Coordinates are predicted relative to the grid cell:

bx, by = center

calculated as:

$bx = \text{sigmoid}(tx) + cx$
 $by = \text{sigmoid}(ty) + cy$

bw, bh = size

calculated as:

$bw = pw * \exp(tw)$
 $bh = ph * \exp(th)$

where (pw, ph) = anchor size

This makes predictions stable and precise.

🔍 STEP 6 — Objectness Score

YOLO predicts **objectness**:

- High → cell thinks there is an object
- Low → background

Objectness helps YOLO ignore useless boxes.

❓ STEP 7 — Class Probabilities

For every predicted box YOLO also predicts class probabilities.

Example:

- person: 0.95
- dog: 0.02
- car: 0.01

Final class confidence = objectness × class score

❓ Now YOLO Has Many Raw Predictions

But many predictions are:

- low confidence
- overlapping
- duplicated

So now we clean them.

❓ STEP 8 — Confidence Thresholding

Remove all predictions with:

- low objectness
- low class confidence

This removes the majority of bad boxes.

❓ STEP 9 — NMS (Non-Maximum Suppression)

NMS removes duplicate detections.

Example: YOLO may predict 5 boxes around the same dog.

NMS keeps the **best one** and removes the rest using IoU.

🔍 STEP 10 — Final Output

After NMS you get clean detections:

Person – 0.89 – [x1,y1,x2,y2]

Dog – 0.78 – [x1,y1,x2,y2]

Car – 0.65 – [x1,y1,x2,y2]

This is the final result.

🔍 FINAL SUMMARY (VERY SIMPLE VERSION)

YOLO pipeline:

1. **Take whole image**, NOT divided into grid
 2. **Run convolutions** on the whole image
 3. Output feature map (e.g., 13×13) = **the grid**
 4. Each grid cell predicts **boxes + objectness + class**
 5. YOLO adjusts box coordinates using **anchors**
 6. YOLO filters low-confidence boxes
 7. **NMS removes duplicates**
 8. Outputs final clean detections
-

🔍 YOLOv2 / YOLO9000 (2017)

1. What Problems YOLOv2 Solved

YOLOv1 weaknesses:

- Poor localization
- Struggled with different object shapes

- Low resolution feature maps

YOLOv2 introduced several breakthroughs.

2. Key Improvements (Differences)

✓ 1. Anchor Boxes (big jump)

Inspired by Faster R-CNN.

Bounding box priors make predictions more accurate.

✓ 2. Higher resolution input

YOLOv1 used 224×224.

YOLOv2 used **416×416**, improving small object detection.

✓ 3. Batch normalization everywhere

Improved stability and convergence.

✓ 4. Darknet-19 Backbone

More powerful but still fast.

Result:

📊 **2× more accurate than YOLOv1**

📊 **Still real-time (40–90 FPS)**

📊 FINAL SUMMARY — How YOLO Versions Differ

Version	Main Idea	Improvements
YOLOv1	One-stage detection	Real-time, simple grid-based prediction
YOLOv2	YOLO9000 + anchors	Better localization, Darknet-19
YOLOv3	Multiscale detection	FPN + Darknet-53 + 3 scale outputs
YOLOv4	Community SOTA	CSPDarknet, Mosaic, CloU, PANet

Version	Main Idea	Improvements
YOLOv5	PyTorch YOLO	User-friendly, modular, fast training
YOLOv6	Industrial YOLO	EfficientRep, quantization, deployment
YOLOv7	New architecture	E-ELAN, reparam conv, best accuracy/speed
YOLOv8	Anchor-free YOLO	Decoupled head, modern pipeline

Evaluation Metrics in Object Detection — Detailed Notes

Object detection evaluation is more complex than classification because the model must output:

1. **What** the object is (class label)
2. **Where** the object is (bounding box)

So evaluation must measure **classification correctness** AND **localization accuracy**.

1. Intersection over Union (IoU)

Purpose: Measures how accurate a predicted bounding box is.

Step-by-step:

1. Take the predicted bounding box and ground truth box.
2. Compute their overlap area (intersection).
3. Compute the total area covered by both boxes (union).
4. $\text{IoU} = (\text{Area of Intersection}) / (\text{Area of Union})$

Example:

- Intersection area = 40
- Union area = 100
- $\text{IoU} = 40/100 = \mathbf{0.40}$

Usage:

- $\text{IoU} \geq \text{threshold} = \mathbf{\text{True Positive (TP)}}$
- $\text{IoU} < \text{threshold} = \mathbf{\text{False Positive (FP)}}$

Common thresholds:

- PASCAL VOC: 0.5 IoU
 - COCO: multiple thresholds (0.5, 0.55, 0.6, ..., 0.95)
-

2. Confusion Terminology in Detection

Unlike classification, object detection has extra complications.

True Positive (TP)

Predicted a box correctly (IoU \geq threshold + correct class).

False Positive (FP)

Happens when:

- Wrong class predicted.
- IoU $<$ threshold.
- Duplicate boxes (extra detections for same object).

False Negative (FN)

Model missed a ground truth object (no predicted box matches).

True Negative (TN)

Not used in detection (because background is too large).

3. Precision & Recall

Detection uses **Precision** and **Recall** mainly.

Precision

“How many predicted objects were correct?”

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

High precision = few wrong detections.

Recall

“How many ground truth objects were found?”

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

High recall = model misses fewer objects.

4. Precision–Recall Curve (PR Curve)

We vary the confidence threshold of predictions:

- High threshold → fewer detections → high precision, low recall
- Low threshold → more detections → low precision, high recall

We compute Precision vs Recall for all thresholds → draw curve.

A better detector has:

- **Curve closer to the top-right corner.**
-

5. Average Precision (AP)

AP summarizes the entire PR curve into one metric.

Step-by-step AP calculation (simplified):

1. Compute precision at different recall levels.
 2. Integrate the curve (calculate the area under PR curve).
 3. The resulting value is **AP**.
-

AP at specific IoU levels

- **AP@50** → IoU threshold = 0.5
- **AP@75** → IoU threshold = 0.75

Higher threshold = more strict → harder to score high.

6. mAP — Mean Average Precision

If your dataset has multiple classes (dog, cat, car, etc.):

1. Compute AP for each class.
2. Take the average → **mAP**

Datasets define mAP differently:

PASCAL VOC (mAP@0.5)

- IoU threshold = 0.5 only.
- Good for older models (R-CNN, Fast R-CNN).

COCO mAP (Stricter and Modern Standard)

COCO uses:

- IoU thresholds from **0.5 to 0.95** with step 0.05 → 10 values (0.5, 0.55, 0.6, ..., 0.95)

The final metric:

mAP = average of AP across all IoU thresholds + all classes

This is much harder.

7. COCO Additional Metrics

Apart from mAP, COCO also reports:

AP_small, AP_medium, AP_large

COCO divides objects by area:

- **Small** (< 32×32)
- **Medium** (32–96)

- **Large** (> 96)

Some detectors fail at small objects — this shows their weakness.

AR — Average Recall

“How many objects can the model find at a given number of detections per image?”

COCO reports:

- AR@1
 - AR@10
 - AR@100
 - AR_small
 - AR_medium
 - AR_large
-

8. F1 Score (Sometimes Used)

Combines precision and recall into one number:

$$\mathbf{F1} = 2 \times (\mathbf{Precision} \times \mathbf{Recall}) / (\mathbf{Precision} + \mathbf{Recall})$$

Balanced score:

- High only if both precision and recall are high.

Not commonly used in research, but useful in custom projects.

9. FPS / Latency (Speed Metrics)

Object detection also cares about speed for real-time use.

Common metrics:

- **FPS (Frames per Second)** → how many frames a model processes per second
- **Latency (ms per image)**

Real-time requirement:

- ≥ 30 FPS

YOLO models usually focus on speed.

10. Summary Table

Metric	Measures	Important Because
IoU	Box overlap quality	Determines TP/FP
Precision	Correctness of predictions	Lower false positives
Recall	How many objects are detected	Lower false negatives
PR Curve	Precision vs Recall trade-off	Visual performance understanding
AP	Area under PR curve	Single accuracy number
AP@50 / AP@75	AP at specific IoU threshold	Measures strict vs easy matching
mAP (VOC)	AP averaged over classes at IoU=0.5	Simpler benchmark
mAP (COCO)	AP over IoU=0.5–0.95	Modern strict benchmark
AR	Recall across thresholds	How many objects found overall
FPS / Latency	Model speed	Real-time detection

✓ Want the next topic?

I can continue with:

- 🔗 **Non-Maximum Suppression (NMS)**
- 🔗 **Loss functions in object detection (YOLO, Faster R-CNN)**
- 🔗 **Anchor-based vs Anchor-free detectors**
- 🔗 **Two-stage vs one-stage detectors summary**

Just tell me!

You said:

Non-Maximum Suppression (NMS)

ChatGPT said:

Below is a **very detailed, step-by-step explanation** of **Non-Maximum Suppression (NMS)** — what it is, why it exists, and exactly how it works.

Non-Maximum Suppression (NMS)

NMS is a **post-processing step used in object detection** to remove duplicate bounding boxes for the same object.

A detector (YOLO, R-CNN, SSD, etc.) typically predicts **multiple boxes around the same object**, and NMS selects the **best one** while discarding the rest.

1. Why do we need NMS? (The core problem)

Object detectors output **many overlapping boxes** because:

1. **Grid cells** or **anchors** predict boxes independently.
2. Many boxes point to the **same object**.
3. Each box has a **confidence score**.

Example:

A dog is present in the image.

YOLO or Faster R-CNN may predict:

- Box A (confidence: 0.92)
- Box B (confidence: 0.85)
- Box C (confidence: 0.78)
- Box D (confidence: 0.60)

All overlapping the same dog.

Without NMS → you would see 4 boxes on top of each other.

2. What NMS Does (Simple Explanation)

NMS keeps **only the highest-confidence box** and removes all other boxes that:

- overlap too much with it
 - $IoU > threshold$
-

3. Step-by-step Process of NMS

Let's break down the algorithm:

Step 1: Get all predicted boxes for one class

(example: All "dog" predictions)

Each prediction contains:

- Bounding box (x1, y1, x2, y2)
 - Confidence score
-

Step 2: Sort the boxes by confidence (High → Low)

Example sorted:

1. Box A → 0.92
 2. Box B → 0.85
 3. Box C → 0.78
 4. Box D → 0.60
-

Step 3: Pick the box with the highest score

- Choose **Box A (0.92)** as a final detection.
-

Step 4: Compute IoU of Box A with all remaining boxes

Calculate IoU with:

- Box B
 - Box C
 - Box D
-

Step 5: Remove boxes with IoU > threshold

Common thresholds:

- 0.5 (VOC)
- 0.5–0.6 (YOLO)
- 0.7 (Faster R-CNN)

Example:

Box IoU with Box A Remove?

B	0.72	Yes
C	0.55	Yes
D	0.20	No

Remove B and C (overlapping too much).
Keep D for later.

Step 6: Repeat the process

Remaining boxes:

- Box D (0.60)

Pick it → compute IoUs with others → keep/remove.

Continue until no boxes remain.

4. Final Output After NMS

Example output might be:

- Box A (0.92)
- Box D (0.60)

This eliminates duplicates.

5. Types of NMS

There are multiple variants. Each improves speed or quality.

1. Standard (Hard) NMS

Boxes with IoU > threshold are **fully removed**.

Problem:

- Over-suppression (removes useful boxes)
-

2. Soft-NMS (Improved Version)

Instead of removing boxes:

- Reduce (decay) their confidence score based on IoU.

Formula idea:

$\text{new_score} = \text{old_score} \times (1 - \text{IoU})$

Benefits:

- Helps detect overlapping objects (e.g., crowd scenes).
 - Used in Faster R-CNN and many modern systems.
-

3. DIoU-NMS (Distance IoU NMS)

Uses **distance between box centers** + IoU.

Better when:

- Two objects are very close together.
-

4. CIoU-NMS

Includes:

- IoU

- center distance
- aspect ratio consistency

Used in newer YOLO versions (YOLOv5+).

6. Problems that NMS Helps Solve

1. Eliminates duplicate detections

Without it, detectors generate too many boxes.

2. Improves precision

Removes false positives caused by overlapping predictions.

3. Required for real-world usage

Every mainstream object detector uses NMS.

7. Visualization Example

Imagine three overlapping boxes around a cat:

```
+-----+  <— High score (keep)
+-----+
+-----+
```

NMS keeps the best one → deletes the rest.

8. Parameters to Tune in NMS

1. IoU Threshold

Low threshold (0.3):

- Removes many boxes
 - Might miss objects
- Used for strict suppression.

High threshold (0.7):

- Keeps more boxes
- Good for crowded scenes

2. Confidence threshold

Remove low-confidence predictions **before NMS** to speed up.

9. NMS in YOLO vs R-CNN

YOLO:

- Uses NMS *per class*
- Faster
- Often uses CloU-NMS

Faster R-CNN:

- Uses NMS in both:
 1. RPN proposals
 2. Final detections
 - Uses higher thresholds (0.7)
-

10. Summary (Quick Revision)

- NMS removes duplicate boxes.
- Picks the highest-confidence box, removes overlapping ones.
- Uses IoU threshold to decide suppression.
- Variants: Hard NMS, Soft-NMS, DIoU-NMS, CloU-NMS.
- Essential for all object detectors.