Understanding YOLO Object Detection:

A Visual Guide

Sergey Bertov
AI Team

Sofia, Bulgaria

Prerequisite reading before the main technical guide

2025

Abstract

This guide explains how YOLO (You Only Look Once) ~~object~~ detection works using visual diagrams and simple analogies. It is designed for new team members who may not have a deep learning background. After reading this guide, you will understand: (1) what happens to an image as it passes through YOLO, (2) why images get smaller but smarter, (3) what backbone, neck, and head mean, (4) how the model works in real-world deployments, and (5) the difference between training and inference.

This is NOT a technical manual. For technical details, hyperparameters, and production deployment, refer to the main comprehensive guide.

Contents

1 Introduction: What is YOLO?

1.1 YOLO is a CNN (Convolutional Neural Network)

Key Concept

Important: YOLO IS a Convolutional Neural Network (CNN). When we say ”YOLO layers” we mean CNN layers. The backbone, neck, and head are all made of standard CNN building blocks: convolutional layers, batch normalization, activation functions, etc.

1.2 The Big Picture

YOLO is a computer vision system that looks at an image and tells you:

- **What** objects are in the image (“boar”, “deer”, “person”)
- **Where** those objects are located (bounding box coordinates)
- **How confident** it is about each detection (0–100%)

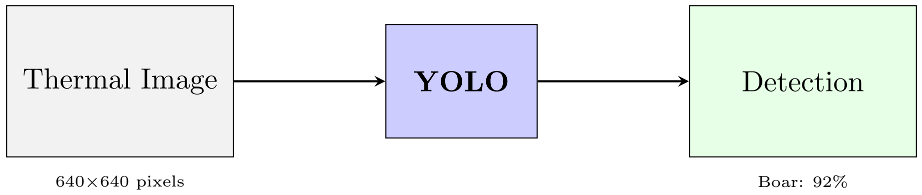


Figure 1: YOLO takes an image as input and outputs object detections

1.3 Why Is It Called YOLO?

YOLO = **You Only Look Once**

Unlike older detection systems that scan an image multiple times, YOLO looks at the entire image just once and makes all predictions simultaneously. This makes it very fast (30+ frames per second).

1.4 What You’ll Learn

This guide uses the factory analogy: imagine YOLO as a factory with three departments processing your image through multiple stations. By the end, you’ll understand what happens at each station and why.

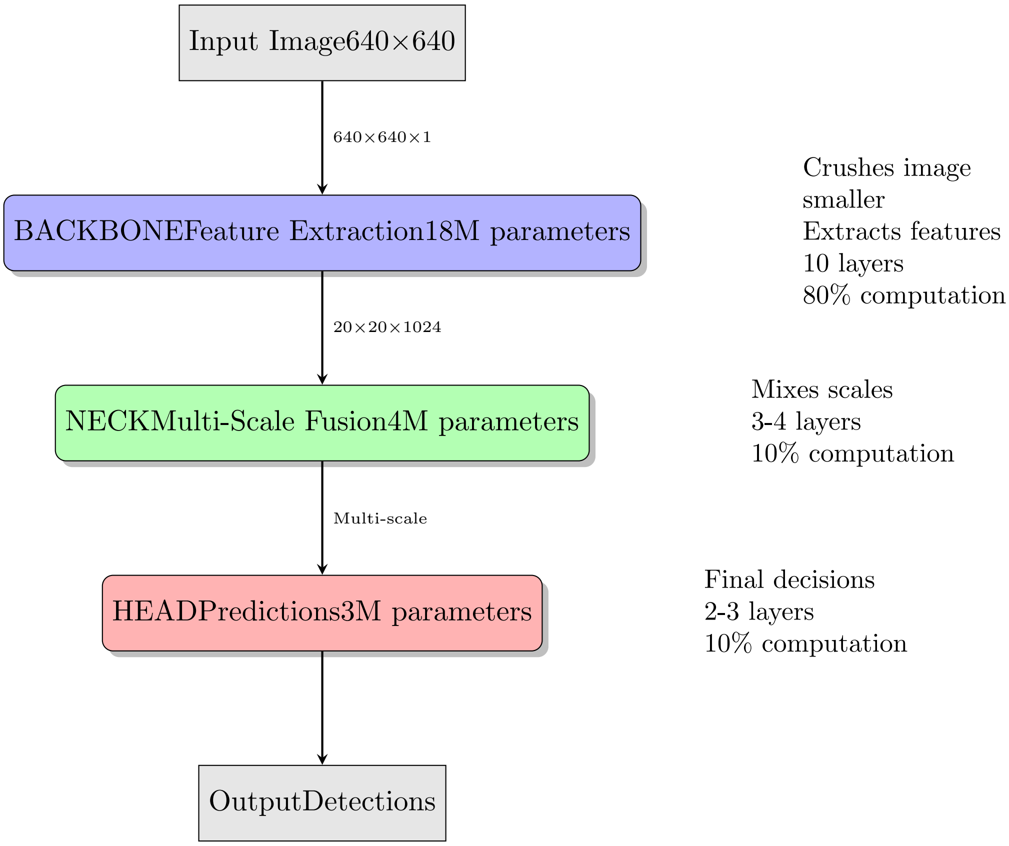


Figure 2: YOLO's three-department architecture

2 The Factory Analogy

2.1 YOLO as a Three-Department Factory

2.2 Why Three Departments?

Important Note

Reference Model: Throughout this guide, all numbers (parameters, memory, FPS) are based on **YOLOv11-Medium** unless otherwise specified. This is our "middle ground" model - not too small, not too large. See Section 3.9 for comparisons across all model scales.

Each department has a specialized job:

- **Backbone:** The hard work. Learns to see edges, textures, shapes, and patterns. Takes 70% of total parameters and 80% of computation time.
- **Neck:** The coordinator. Combines information from different scales so we can detect both tiny mice and huge vehicles.
- **Head:** The decision maker. Uses all the prepared features to say "That's a boar at position (320, 240) with 92% confidence."

3 Understanding CNN Layers

3.1 What is a CNN Layer?

Before we dive into YOLO's departments, you need to understand what a "layer" actually is in a CNN.

Key Concept

A "layer" in our backbone diagram is actually a **block of multiple operations** working together. It's not one single thing - it's a combination of convolution, normalization, and activation.

3.2 The Basic CNN Operations

3.2.1 1. Convolution (Conv)

This is the core operation. A small filter (e.g., 3×3 pixels) slides across the image, looking for patterns.

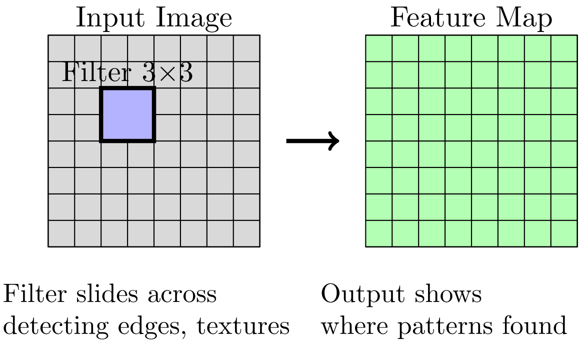


Figure 3: Convolution operation: filter slides across image

What it does:

- Detects patterns (edges, corners, textures)
- Each filter learns one type of pattern
- Multiple filters create multiple channels
- Example: 64 filters = 64 output channels

3.2.2 2. Batch Normalization (BN)

Normalizes the values to make training stable.

What it does:

- Prevents numbers from getting too large or too small
- Makes training faster and more stable
- Applied after every convolution

3.2.3 3. Activation Function (SiLU/ReLU)

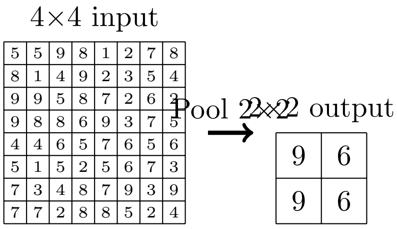
Adds non-linearity so the network can learn complex patterns.

What it does:

- Negative values → 0 (or small number)
- Positive values → keep as is
- Allows network to learn curves, not just straight lines

3.2.4 4. Pooling / Downsampling

Makes the image smaller by combining nearby pixels.



5	5	9	8	1	2	7	8
8	1	4	9	2	3	5	4
9	9	5	8	7	2	6	2
9	8	8	6	9	3	7	5
4	4	6	5	7	6	5	6
5	1	5	2	5	6	7	3
7	3	4	8	7	9	3	9
7	7	2	8	8	5	2	4

9	6
9	6

Take maximum or average
of each 2×2 block

Figure 4: Pooling reduces spatial size by 2×

What it does:

- Reduces image size (640 → 320 → 160...)
- Reduces computation
- Makes network see ”bigger picture”

3.3 One ”Layer” = Multiple Operations

What I’ve been calling ”Layer 2” in the backbone is actually this:

Important Note

When I say ”10 layers in the backbone,” I mean 10 blocks like this. Each block contains multiple CNN operations. In total, YOLO has approximately **200+ individual CNN operations** across all departments!

3.4 Modern Block: C2f / C3k2

Modern YOLO versions (v8-v12) use more advanced blocks:

Why this design?

- **Path 1 (direct):** Helps gradients flow during training
- **Path 2 (processing):** Learns new features
- **Concatenate:** Combines both paths
- Result: Faster training, better gradients

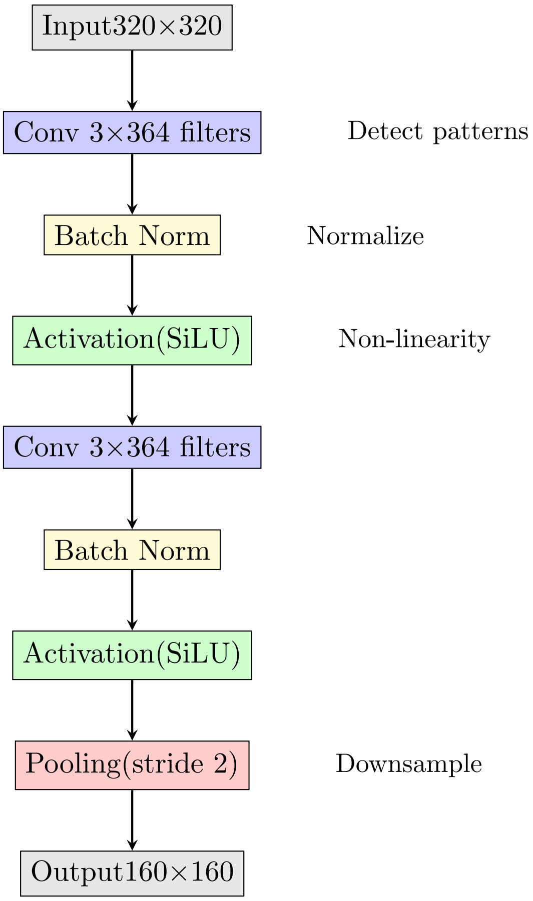


Figure 5: One ”layer” is actually 7-8 operations!

3.5 YOLO Backbone = Stack of CNN Blocks

Important Note

Terminology Clarification:

- **What I called ”Layer 2”:** One block containing 7-8 CNN operations
- **Actual CNN operations:** Conv, BatchNorm, Activation, Pooling
- **Total in YOLO:** 10 ”blocks” = 200+ individual CNN operations

In casual conversation, ML engineers often say ”layer” when they mean ”block.” Now you know both terms!

3.6 Parameters = Learned Weights in CNN Layers

Remember those 25 million parameters?

Each convolution filter has parameters:

- 3×3 filter = 9 parameters
- 64 filters = $9 \times 64 = 576$ parameters
- Layer with 512 input channels, 1024 output channels, 3×3 filter: $= 3 \times 3 \times 512 \times 1024 = 4.7$ million parameters!

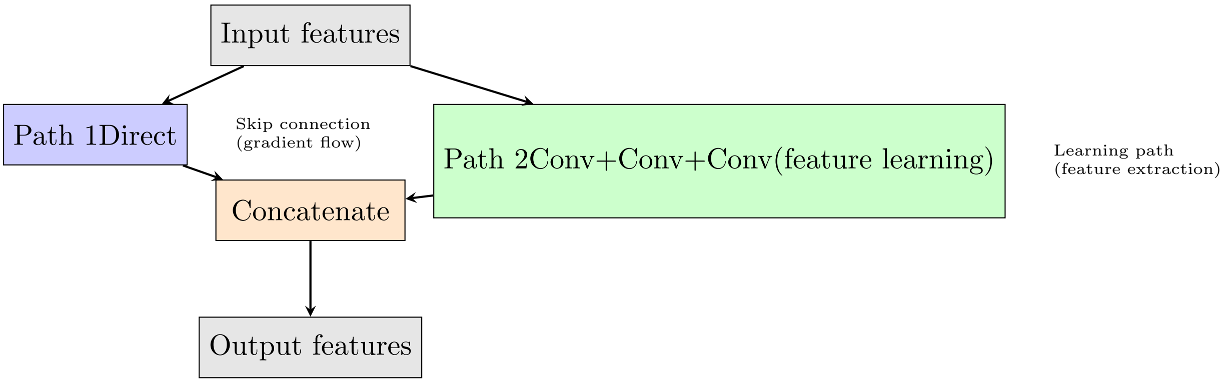


Figure 6: C2f block: split → process → concatenate

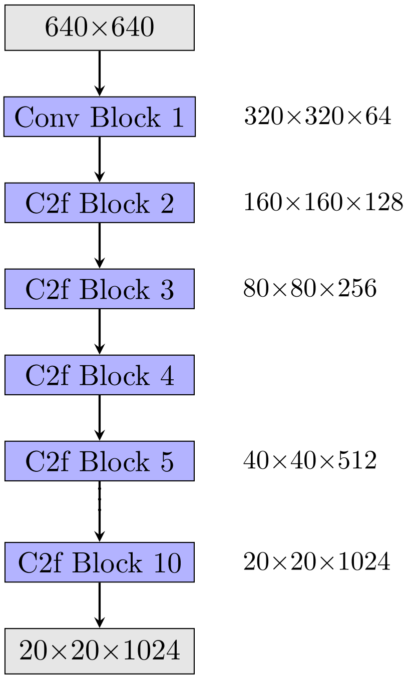


Figure 7: Backbone = 10 CNN blocks stacked together

3.7 Training = Adjusting CNN Parameters

During training:

1. Show image to network
2. Image passes through all Conv/BN/Activation layers
3. Network makes prediction
4. Compare prediction to ground truth
5. Calculate how wrong each parameter was
6. Adjust all 25 million parameters slightly
7. Repeat millions of times

After training, the parameters ”remember” what edges, textures, and objects look like.

Table 1: Where Parameters Live (YOLOv11-Medium)

Operation	**Params**	**What They Store**
Conv filters	22M	Pattern detectors (edges, textures, shapes)
Batch Norm	2M	Normalization statistics
Classification head	1M	Class-specific weights
Total	**25M**	Learned knowledge

3.8 Model Scale Comparison

The 25 million parameters we've been discussing is for **YOLOv11-Medium**. YOLO comes in 5 different sizes:

Table 2: YOLOv11 Model Scales - Parameter Breakdown

Component	**Nano**	**Small**	**Medium**	**Large**	**X-Large**
Backbone	2.0M	7.5M	18M	30M	48M
Neck	0.5M	2.0M	4M	8M	12M
Head	0.2M	1.5M	3M	5M	8M
Total	**2.7M**	**11M**	**25M**	**43M**	**68M**

Table 3: Model Scale Trade-offs

Scale	**Params**	**Memory**	**FPS**	**mAP**	**Use Case**
Nano	2.7M	600MB	120	0.86	Edge devices
Small	11M	1.2GB	80	0.89	Drones, embedded
Medium	25M	2.0GB	50	0.92	Workstations
Large	43M	3.2GB	30	0.94	Servers
X-Large	68M	4.8GB	18	0.95	Max accuracy

Key insight: The architecture (backbone → neck → head) is the SAME for all scales. Only the *width* and *depth* change:

- **Width:** Number of channels (Nano: 64, Medium: 256, X-Large: 1024)
- **Depth:** Number of C2f blocks repeated (Nano: 1×, Medium: 2×, X-Large: 3×)

Real-World Example

Example: Why We Use Medium as Reference

- **Training:** We use YOLOv11-Medium or Large (good accuracy, reasonable training time)
- **Deployment on Jetson Nano:** We export to Nano after training (28 FPS, low power)
- **Deployment on workstations:** We use Medium or Large (real-time processing)

Medium is the ”sweet spot” for explaining concepts because:

- Not too simple (like Nano)
- Not too complex (like X-Large)
- Most commonly used in production
- Balances accuracy and speed

3.9 How Model Scale Affects Real Deployment

Scenario: Thermal boar detection on drone

Table 4: Same Task, Different Models

Model	Training	Deploy	FPS	Accuracy
v11n	8 hours	Jetson Nano	28	88%
v11s	16 hours	Jetson AGX	22	91%
v11m	35 hours	RTX 3080	15	93%
v11l	60 hours	RTX 5090	10	94%

Decision factors:

- Battery-powered drone → Must use Nano (only model that fits power/memory budget)
- Fixed ground station → Can use Medium or Large (no power constraints)
- Safety-critical → Use Large (best accuracy)
- Cost-sensitive → Use Nano (cheapest hardware)

4 Department 1: The Backbone

4.1 What Does the Backbone Do?

The backbone’s job is to take your large image (640×640 pixels) and:

1. Make it progressively smaller (640 → 320 → 160 → 80 → 40 → 20)
2. Extract increasingly complex features at each step
3. Output rich, semantic information about what’s in the image

Key Concept

The backbone has **approximately 18 million parameters** (for YOLOv11-Medium, our reference model). These are the "memory" that stores learned knowledge about what edges, textures, and shapes look like.
Note: All numbers in this guide use YOLOv11-Medium unless specified. Other scales range from 2.7M (Nano) to 68M (X-Large) total parameters. See Section 3.9 for full comparison.

4.2 The Downsampling Journey

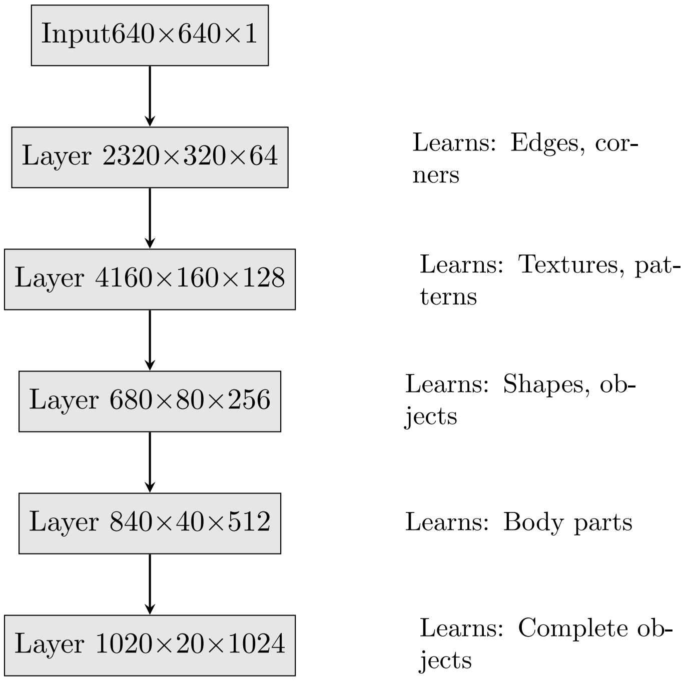


Figure 8: Image gets smaller but smarter through backbone layers

4.3 The Paradox: Smaller but Smarter

This is the most confusing concept for beginners:

Important Note

The image gets physically SMALLER but informationally RICHER

Table 5: Size vs Intelligence Trade-off

Stage	**Pixels**	**Info/Pixel**	**Understanding**
Input	640×640	1 number	"I'm gray"
Layer 2	320×320	64 numbers	"I see edges"
Layer 4	160×160	128 numbers	"I see texture"
Layer 6	80×80	256 numbers	"I see shapes"
Layer 8	40×40	512 numbers	"I see body parts"
Layer 10	20×20	1024 numbers	"I see animals"

Analogy: Think of compressing a book. The physical size shrinks, but if done smartly (extracting key concepts, summarizing), you can retain all important information in much smaller space.

4.4 Why This Design?

Question: Why not keep the image at 640×640 throughout?

Answer: Computational cost. Processing 640×640 pixels through 10 layers would require:

- 100× more memory
- 100× more computation time
- Would take 5 minutes per image instead of 30 milliseconds

By shrinking the image while increasing feature depth, we get the best of both worlds: fast processing with rich understanding.

5 Department 2: The Neck

5.1 The Multi-Scale Problem

The backbone gives us three feature maps at different sizes. Different objects need different scales:

- Small objects (mouse, bird): Need fine 80×80 features
- Medium objects (boar, person): Need medium 40×40 features
- Large objects (vehicle, house): Need coarse 20×20 features

Solution: The NECK mixes all three scales together!

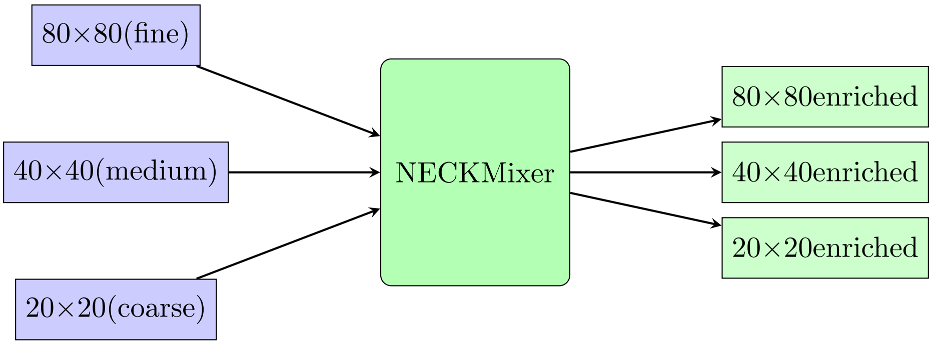


Figure 9: Neck mixes three scales from backbone

Key Concept

The neck has only **4 million parameters** for YOLOv11-Medium (16% of total). It's a quick processing step but crucial for multi-scale detection. Other scales: Nano has 0.5M, X-Large has 12M.

6 Department 3: The Head

6.1 The Final Decision Maker

The head receives enriched multi-scale features from the neck and produces final predictions. For YOLO, the head is **decoupled**, meaning it splits into two independent branches:

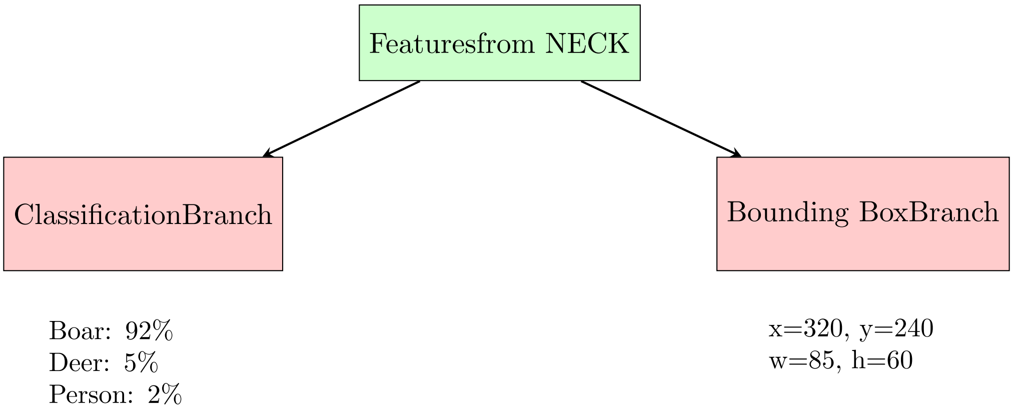


Figure 10: Head splits into classification and localization branches

6.2 Three Detection Heads for Three Scales

YOLO actually has THREE detection heads, one for each scale:

Table 6: Three Detection Heads

Head	Size	Stride	Best For
P3	80×80	8 pixels	Small objects (8–32px)
P4	40×40	16 pixels	Medium objects (32–96px)
P5	20×20	32 pixels	Large objects (>96px)

7 How YOLO ”Sees” and Predicts Objects

This is the most important section for understanding YOLO’s ”thinking process.”

7.1 The Grid System

YOLO doesn’t look at the image as a whole. Instead, it divides the image into a grid and each grid cell becomes responsible for detecting objects.

Key Concept

Key Concept: YOLO creates three grids at different scales:

- **80×80 grid (P3):** 6,400 cells for small objects
- **40×40 grid (P4):** 1,600 cells for medium objects
- **20×20 grid (P5):** 400 cells for large objects

Each grid cell asks: ”Is there an object center in my area? If yes, what is it and where exactly?”

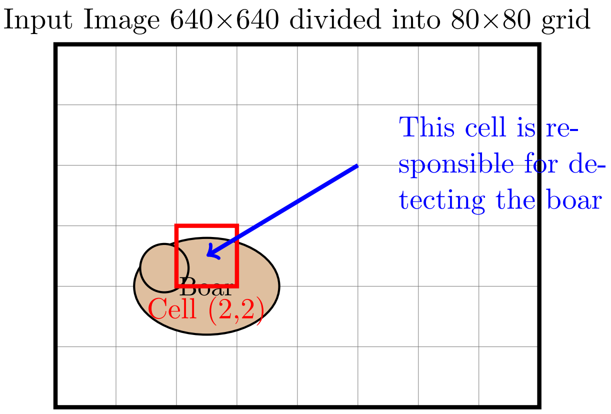

Input Image 640×640 divided into 80×80 grid
This cell is responsible for detecting the boar
Boar
Cell (2,2)

Figure 11: YOLO divides image into grid (e.g., 80×80 = 6400 cells)

7.2 What Each Grid Cell Predicts

Every single grid cell makes predictions. For a 40×40 grid, that's 1,600 predictions happening simultaneously!

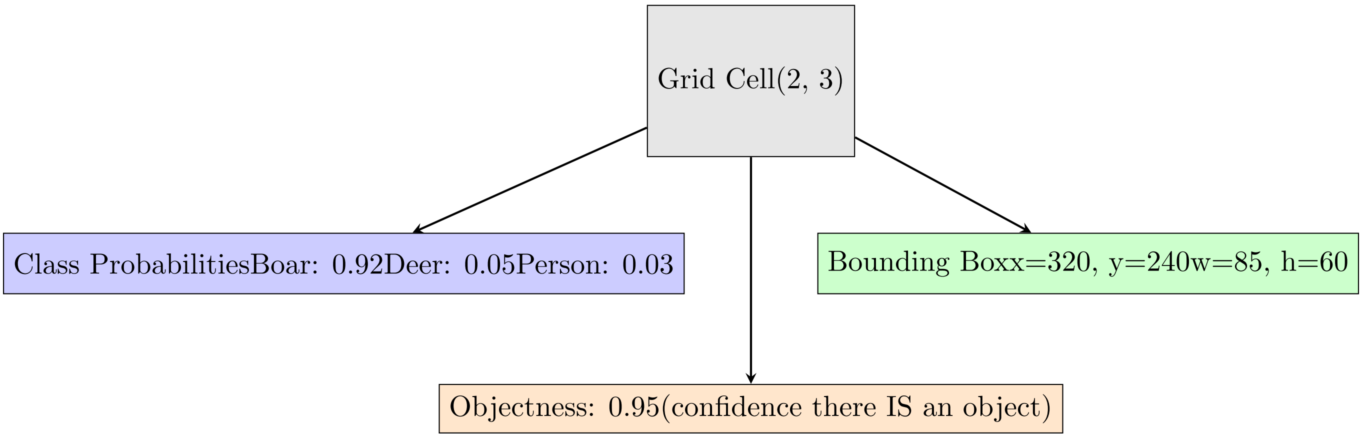

Grid Cell(2, 3)
Class ProbabilitiesBoar: 0.92Deer: 0.05Person: 0.03
Bounding Boxx=320, y=240w=85, h=60
Objectness: 0.95(confidence there IS an object)

Figure 12: Each grid cell predicts: class, bounding box, and confidence

For each grid cell, YOLO predicts:

1. **Objectness Score** (0–1): "How confident am I that there's an object here?"
   - 0.95 = "Yes, definitely an object here!"
   - 0.10 = "Probably just background"
2. **Class Probabilities** (for each class): "If there IS an object, what is it?"
   - Boar: 0.92 (92%)
   - Deer: 0.05 (5%)
   - Person: 0.03 (3%)
3. **Bounding Box** (x, y, w, h): "Where exactly is the object?"
   - x, y = center point coordinates
   - w, h = width and height
   - Relative to the grid cell position

7.3 From Grid Cell to Bounding Box

Let's trace how one grid cell makes a prediction:

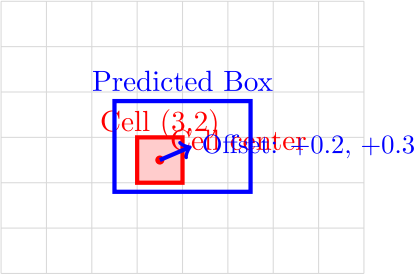


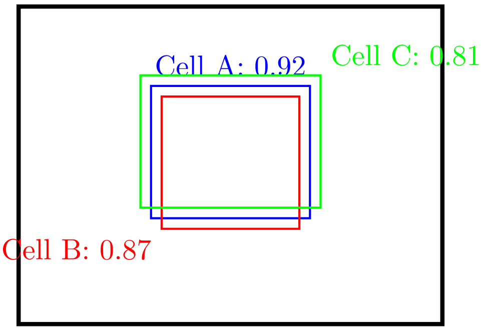
Figure 13: Grid cell predicts offset from its center + box size

Step-by-step prediction:

1. Cell (3, 2) detects object center is near
2. Predicts offset: "Object center is 0.2 cells right, 0.3 cells up from my center"
3. Predicts size: "Object is 3 cells wide, 2 cells tall"
4. Calculates actual box coordinates in image
5. Predicts class: "92% sure it's a boar"
6. Outputs: Boar, confidence 0.92, box [x, y, w, h]

7.4 Multiple Predictions and NMS

Problem: Multiple grid cells might detect the same object!



Three different cells all detected the same boar!
Need to remove duplicates...

Figure 14: Problem: Multiple overlapping detections

Solution: Non-Maximum Suppression (NMS)
NMS removes duplicate detections:
NMS Algorithm:

1. Sort all predictions by confidence (highest first)
2. Take highest confidence box (Box A: 0.92) → Keep it

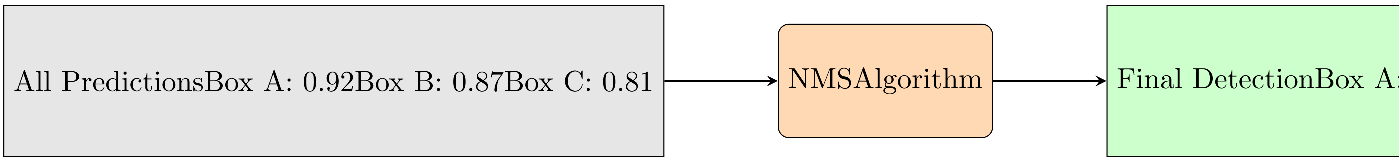


Figure 15: NMS keeps only the best prediction per object

3. Check all remaining boxes:
   - If Box B overlaps Box A by >50% → Remove B (duplicate)
   - If Box C overlaps Box A by >50% → Remove C (duplicate)
4. Repeat for next highest box
5. Result: One detection per object

7.5 Confidence Threshold

Not all predictions are kept. We filter by confidence:

Table 7: Confidence Filtering

Prediction	**Confidence**	**Decision**
Boar at (320, 240)	0.92	Keep (above 0.25)
Deer at (450, 180)	0.78	Keep (above 0.25)
Person at (200, 300)	0.15	Discard (below 0.25)
Tree at (500, 400)	0.08	Discard (below 0.25)

Typical threshold: 0.25 (25% confidence)

- Lower threshold (0.10): More detections, more false positives
- Higher threshold (0.50): Fewer detections, fewer false positives

7.6 Complete Prediction Pipeline

7.7 Why This Design Works

1. **Parallelism:** All 8,400 grid cells predict simultaneously → Very fast
2. **Multi-scale:** Three grid sizes detect small/medium/large objects
3. **Anchor-free:** No need to pre-define object shapes (modern YOLO)
4. **Single pass:** ”You Only Look Once” - entire image processed at once

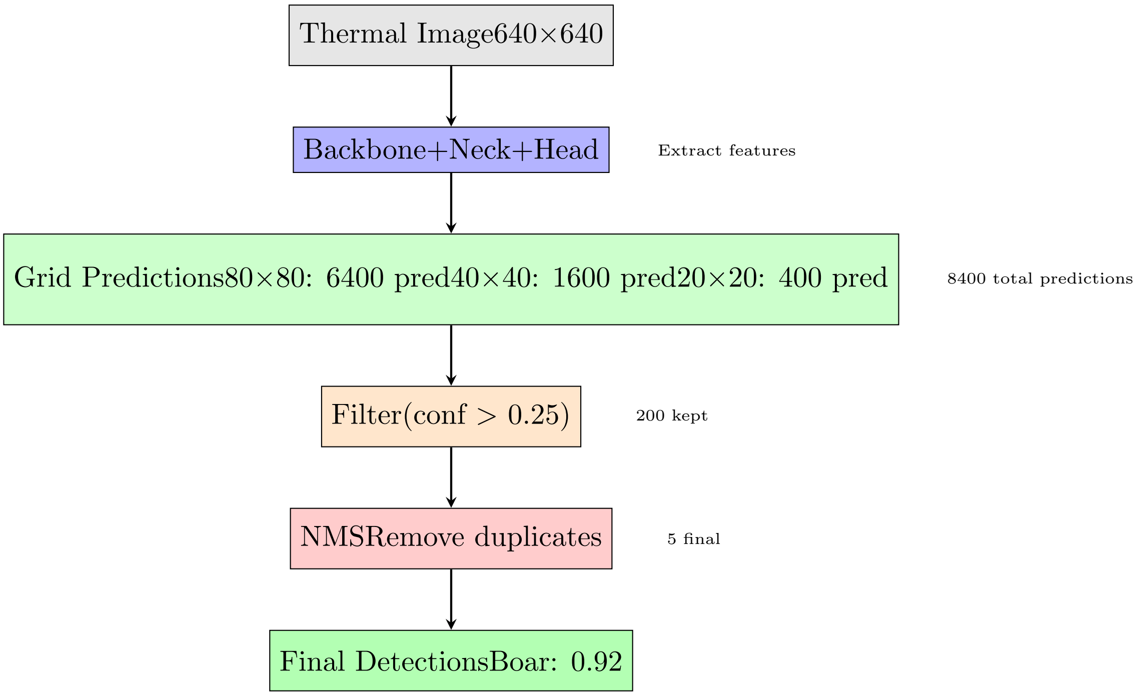


Figure 16: From image to final detections

Real-World Example

Real Example: Boar Detection
Input: 640×640 thermal image with one boar
Grid predictions (40×40 grid, 1600 cells):

- Cell (25, 18): ”Boar, 0.92 confidence, box [320, 240, 85, 60]”
- Cell (26, 18): ”Boar, 0.87 confidence, box [325, 242, 88, 62]” (duplicate)
- Cell (24, 19): ”Boar, 0.81 confidence, box [318, 245, 83, 58]” (duplicate)
- 1597 other cells: ”No object” or very low confidence

After confidence filter (>0.25): 3 predictions kept
After NMS: 1 prediction kept (the best one: 0.92)
Final output: Boar detected at (320, 240) with 92% confidence

7.8 Tracking vs Detection

Important Note

Important: YOLO only does **detection**, NOT tracking!

- **Detection:** ”There’s a boar at position (320, 240) in THIS frame”
- **Tracking:** ”This is the SAME boar from the previous frame, it moved from (310, 230) to (320, 240)”

YOLO processes each frame independently - it has NO memory between frames. For tracking, you need additional algorithms like:

- DeepSORT (most common)
- ByteTrack
- StrongSORT

These algorithms use YOLO detections + unique IDs to track objects across frames.

8 Hardware and Deployment (Simplified)

8.1 Real-World Deployment Scenarios

Now that you understand the architecture, let’s see how YOLO actually runs in production systems.

8.1.1 Scenario 1: Thermal Drone Wildlife Monitoring

Real-World Example

Hardware: Jetson Nano mounted on drone with thermal camera
Task: Detect boars in forest at night
Real-time operation:

1. Thermal camera captures frame: 640×512 @ 30 FPS
2. Frame resized to 640×640
3. YOLO processes in 35ms (28 FPS)
4. Detections sent to ground station via radio
5. Operator sees bounding boxes on live video feed

Constraints:

- Battery power: 20W total drone budget
- Must use YOLOv11-Nano (smallest model)
- TensorRT FP16 optimization required
- 600MB RAM footprint maximum

8.1.2 Scenario 2: Fixed Camera Surveillance

Real-World Example

Hardware: Workstation with RTX 5090 + multiple RGB cameras
Task: Monitor perimeter for intrusions (multi-class)
Real-time operation:

1. 4 cameras @ 1920×1080 @ 30 FPS each
2. YOLO processes all 4 streams in parallel
3. YOLOv11-Medium achieves 85 FPS per stream
4. GPU processes 340 FPS total (4 streams)
5. Detections logged to database with timestamps
6. Alerts triggered for specific classes (person, vehicle)

Advantages:

- No power constraints (wall power)
- Can use larger, more accurate models
- Multiple streams processed simultaneously
- Real-time video recording with annotations

8.2 Frame-by-Frame Processing

YOLO
33ms processing
(30 FPS)
Frame
Time

Figure 17: Continuous frame-by-frame processing

Key points:

- Each frame processed independently
- No memory between frames (stateless)
- Must complete processing before next frame arrives
- 30 FPS = 33ms per frame budget
- If processing takes 40ms, you get 25 FPS instead

8.3 Latency Breakdown

For a typical detection on Jetson Nano with YOLOv11-Nano:

Target: Keep total latency <33ms for real-time 30 FPS operation.

Table 8: Latency Components (YOLOv11-Nano on Jetson Nano)

Operation	Time (ms)	% Total
Camera capture	2	6%
Pre-processing (resize, normalize)	3	8%
YOLO inference (GPU)	25	69%
Post-processing (NMS, filtering)	4	11%
Drawing boxes	2	6%
Total	**36ms**	**100%**

Note: Using YOLOv11-Medium on same hardware would take 80ms (too slow for real-time). That's why edge devices must use Nano or Small models.

8.4 Multi-Camera Systems

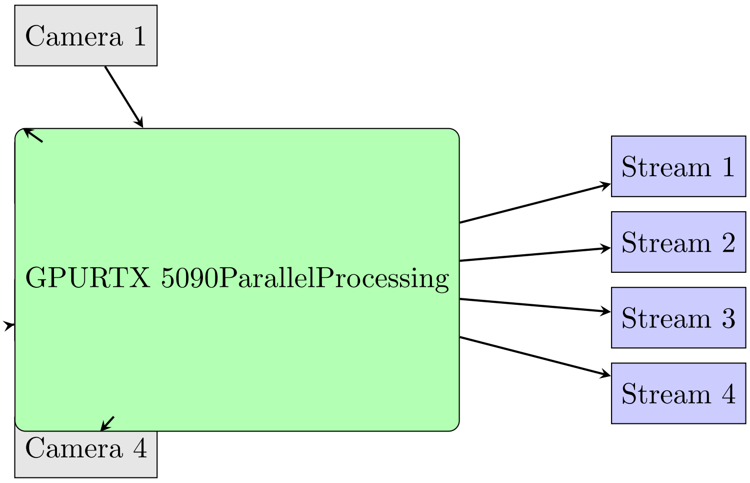


Figure 18: Multi-camera parallel processing

GPU batching: Instead of processing frames one-by-one, batch 4 frames from 4 cameras together. GPU processes all 4 simultaneously with minimal overhead.

8.5 Power and Performance Trade-offs

Table 9: Platform Comparison

Platform	Power	Model	FPS	Use Case
Jetson Nano	10W	v11n	28	Battery drones
Jetson AGX	30W	v11s	45	Robots, vehicles
RTX 3080	320W	v11m	85	Workstations
RTX 5090	450W	v11l	120	Servers
DGX Spark	10.5kW	v12x	200+	Data centers

Important Note

Critical Trade-off: Smaller models (Nano) run on low power but less accurate. Larger models (Large, X-Large) more accurate but require high power. Choose based on deployment constraints!

8.6 Integration with Existing Systems

8.6.1 REST API Deployment

Most common production pattern:

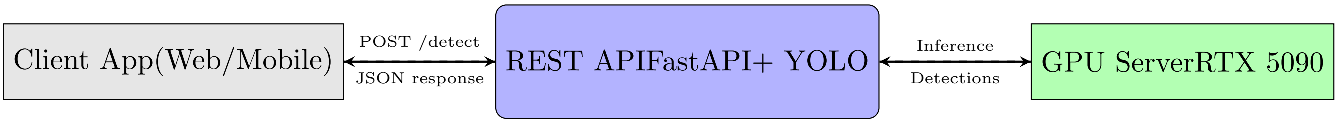


Figure 19: REST API deployment pattern

Client sends image via HTTP POST, receives JSON with detections:

```
{
  "detections": [
    {
      "class": "boar",
      "confidence": 0.92,
      "bbox": [320, 240, 85, 60]
    }
  ],
  "inference_time_ms": 25
}
```

8.6.2 Edge Deployment on Drones

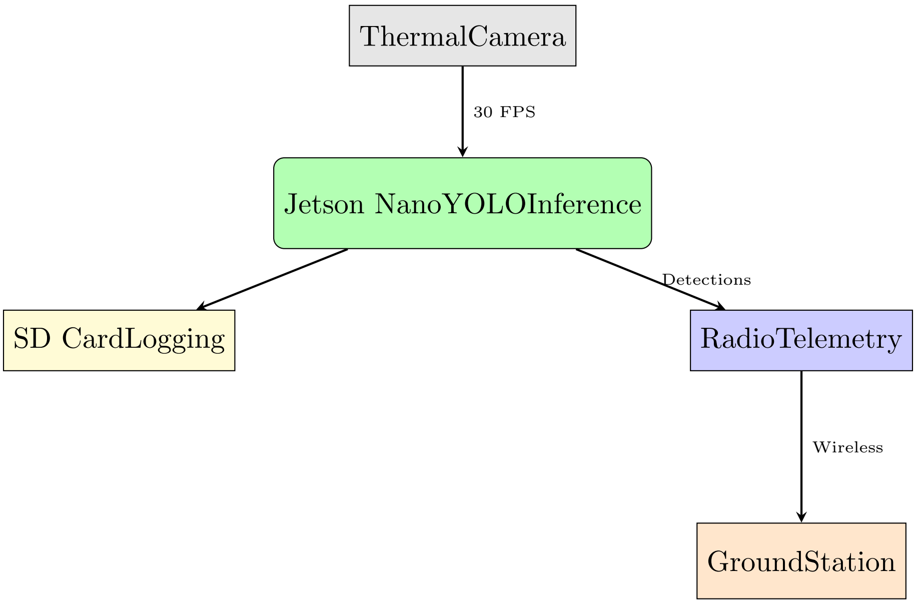


Figure 20: Edge deployment architecture for drones

Key features:

- All processing done on-board (no cloud dependency)
- Detections logged locally to SD card
- Real-time telemetry sent to ground station

- Works even if radio link lost (autonomous)
- Low latency (25-35ms total)

8.7 Real-World Performance Metrics

What matters in production:

1. **Throughput:** How many frames per second (FPS)
   - Target: $\geq$30 FPS for real-time
   - Measured: Average FPS over 1000 frames
2. **Latency:** Time from capture to detection
   - Target: $<$100ms for interactive systems
   - Measured: End-to-end timestamp difference
3. **Accuracy:** How often detections are correct
   - Target: $>$90% mAP50 for production
   - Measured: On held-out validation set
4. **Reliability:** Uptime and crash rate
   - Target: $>$99.9% uptime
   - Measured: Hours without crash
5. **Power consumption:** For battery-powered devices
   - Target: $<$15W for Jetson Nano drones
   - Measured: Wattmeter on power supply

8.8 Common Deployment Challenges

8.8.1 Challenge 1: Variable Lighting

Problem: Model trained on daylight fails at night
Solution:

- Use thermal camera (illumination-independent)
- OR train on mixed lighting conditions
- OR use separate day/night models

8.8.2 Challenge 2: Memory Overflow on Edge Devices

Problem: YOLOv11-Medium requires 1.2GB RAM, Jetson has 8GB total (shared with OS)
Solution:

- Use smaller model (Nano: 600MB)
- Enable TensorRT FP16 (halves memory)
- Reduce input resolution: 640×640 → 416×416

8.8.3 Challenge 3: Network Latency in Cloud Deployments

Problem: Sending video to cloud adds 100-300ms latency
Solution:

- Edge processing (on-device inference)
- Compress video before upload
- Use regional cloud servers (closer to devices)

9 Training vs Inference

9.1 Two Different Modes

Table 10: Training vs Inference Comparison

Aspect	Training	Inference
Purpose	Learn from data	Make predictions
Duration	Days/weeks	Milliseconds
Hardware	RTX 5090, DGX	Jetson, RTX, CPU
Memory	High (20GB+)	Low (600MB-2GB)
Power	High (450W+)	Low (10-30W)
Frequency	Once	Continuous
Gradient	Computed	Not computed
Dropout	Active	Disabled
Batch norm	Training mode	Eval mode

9.2 Training Mode: Learning Phase

What happens:

1. Show 10,000 images to YOLO
2. For each image:
   - YOLO makes prediction
   - Compare to ground truth label
   - Calculate error (loss)
   - Adjust 25 million parameters slightly
3. Repeat for 300-1000 epochs (months of GPU time)
4. Save best model weights to disk

Hardware: RTX 5090 or DGX Spark (expensive, high power)
Frequency: Once per model version (every few weeks/months)

9.3 Inference Mode: Production Use

What happens:

1. Load trained model weights (fixed, no changes)
2. Receive new image from camera

3. Forward pass through network (25ms)
4. Output detections
5. Repeat for every frame (30 times per second)

Hardware: Jetson Nano (cheap, low power, embedded)
Frequency: Continuous (millions of inferences per day)

Key Concept

Key Insight: You train once on expensive hardware, then deploy the trained model on cheap hardware for inference. The 25 million parameters are fixed during inference - no learning happens.

10 Common Beginner Questions

10.1 Q0: Is YOLO a CNN or something different?

A: YOLO IS a CNN! Everything in YOLO (backbone, neck, head) is built from standard CNN components:

- Convolutional layers (Conv)
- Batch normalization (BN)
- Activation functions (SiLU, ReLU)
- Pooling / downsampling

When we say "YOLO has 10 layers in the backbone," we mean 10 *blocks* of CNN operations. Each block contains multiple Conv+BN+Activation operations. In total, YOLO has 200+ individual CNN operations.

10.2 Q1: Do we retrain the model for every new image?

No! Training happens once (or periodically when you collect new data). During deployment, the model weights are frozen. Inference just uses the trained weights to make predictions - no learning happens.

10.3 Q2: Can the model learn during deployment?

No (usually). Standard YOLO deployments don't learn from new data. If you want the model to improve, you must:

1. Collect new labeled data
2. Retrain the model offline
3. Deploy the updated model

Some advanced systems use *online learning*, but that's rare and complex.

10.4 Q3: Why 30 FPS specifically?

A: Human perception threshold. Anything >24 FPS appears smooth to human eyes. 30 FPS is standard for video. Some applications need higher:

- Gaming, VR: 60-120 FPS
- High-speed motion: 240+ FPS
- Wildlife monitoring: 30 FPS is fine

10.5 Q4: What happens if YOLO is wrong?

In production:

- False positive: Detected object that doesn't exist → User ignores or system filters it
- False negative: Missed an object → Potentially dangerous in safety-critical applications
- Wrong class: Detected deer as boar → User corrects or system uses confidence threshold

To improve: Collect more examples of failure cases and retrain.

10.6 Q5: Can we run YOLO on CPU?

Yes, but slow. CPU inference:

- YOLOv11-Nano on modern CPU: 2-5 FPS (200ms per frame)
- Not real-time, but acceptable for non-time-critical tasks
- No GPU required (cheaper deployment)

Rule of thumb: Real-time needs GPU. Batch processing can use CPU.

11 Next Steps

11.1 Hands-On Learning

Now that you understand the concepts, try these:

1. **Watch a training run** on our lab workstation
   - See loss curves in real-time
   - Watch mAP improve over epochs
   - Observe GPU utilization
2. **Run inference** on a trained model
   - Load a .pt model file
   - Process test images
   - See detections appear
3. **Deploy to Jetson Nano**
   - Export model to TensorRT
   - Connect thermal camera

    - Run real-time detection

4. **Experiment with architectures**
    - Compare v11n vs v11s vs v11m
    - Measure FPS on different hardware
    - See accuracy vs speed trade-offs

11.2 Read the Main Technical Guide

You're now ready for the comprehensive technical guide which covers:

- Detailed hyperparameter tuning
- Sensor-specific training (thermal vs RGB)
- Production deployment patterns
- Troubleshooting and debugging
- Multi-class training strategies
- Hardware optimization techniques

11.3 Key Takeaways

1. YOLO has three parts: backbone (feature extraction), neck (multi-scale fusion), head (predictions)
2. Images get smaller but smarter through progressive downsampling + feature enrichment
3. Training happens once on powerful GPUs; inference runs continuously on edge devices
4. Real-world deployment requires careful trade-offs: accuracy vs speed vs power
5. Thermal requires more data and training time than RGB
6. Edge deployment (Jetson) enables autonomous operation without cloud dependency

Glossary

Activation Function: Non-linear function (ReLU, SiLU) applied after convolution. Allows network to learn complex patterns.

Backbone: The feature extraction component (layers 1–10) that learns to see edges, textures, shapes, and patterns. Contains 70–75% of network parameters. Made of CNN blocks (Conv+BN+Activation).

Batch Normalization (BN): Normalizes layer outputs to speed up training and improve stability.

Batch Size: Number of images processed simultaneously. Typical: 8–32 for training, 1 for edge inference.

C2f / C3k2: Modern CNN block designs used in YOLOv8-v12. Combines direct path (skip connection) with processing path (feature learning).

Channel: One "view" or "filter" of the image. RGB has 3 channels, deep layers have 256–1024 channels. Each channel produced by one convolution filter.

CNN (Convolutional Neural Network): Type of neural network using convolution operations. YOLO is a CNN.

Convolution (Conv): Core CNN operation. Small filter slides across image detecting patterns. Creates feature maps.

Downsampling: Making images smaller (640→320→160→80→40→20) to reduce computation. Done via pooling or strided convolution.

Edge Device: Low-power embedded computer (Jetson Nano) that runs inference locally without cloud.

Epoch: One complete pass through the entire training dataset.

FPS (Frames Per Second): How many images YOLO can process per second. Real-time = 30+ FPS.

Head: Final layers that make predictions (class labels + bounding boxes).

Inference: Running a trained model to make predictions on new images (production mode).

Latency: Time from image capture to detection result. Target: <100ms for interactive systems.

Loss: Measure of how wrong the predictions are. Training tries to minimize loss.

mAP (mean Average Precision): Accuracy metric. 0.90 = 90% accuracy.

Neck: Middle component that mixes features from different scales.

NMS (Non-Maximum Suppression): Removes duplicate detections of the same object.

Parameter: One learnable weight in the network. YOLOv11m has 25 million parameters.

TensorRT: NVIDIA's optimization library that makes inference 2-3× faster on GPUs.

Training: The learning phase where model adjusts its 25 million parameters using labeled data.

Contact

Questions about this guide? Contact the AI team:

- Team Lead: Sergey Bertov (sergiibertov@gmail.com)