Practical Object Detection with YOLO

A Step-by-Step Guide for Beginners

Book 1 | Practical Computer Vision with Deep Learning

Dr. Ali Khan | Dr. Somaiya Khan

March 2026

Copyright © 2026 Ali Khan and Somaiya Khan. All rights reserved.

Chapter 1

Introduction to Object Detection

1.1 What Is Object Detection?

Object detection is a fundamental task in computer vision that focuses on identifying and localizing objects of interest within an image. In practical terms, a detection system answers two questions simultaneously: *what* objects are present, and *where* they are located.

Unlike image classification which assigns a single label to an entire image, object detection produces a structured output consisting of multiple elements. For each detected object the model predicts a **bounding box** describing its spatial extent, a **class label** identifying the object category, and a **confidence score** reflecting the model's certainty. This makes object detection more demanding than image classification, but also more useful for many real-world applications.

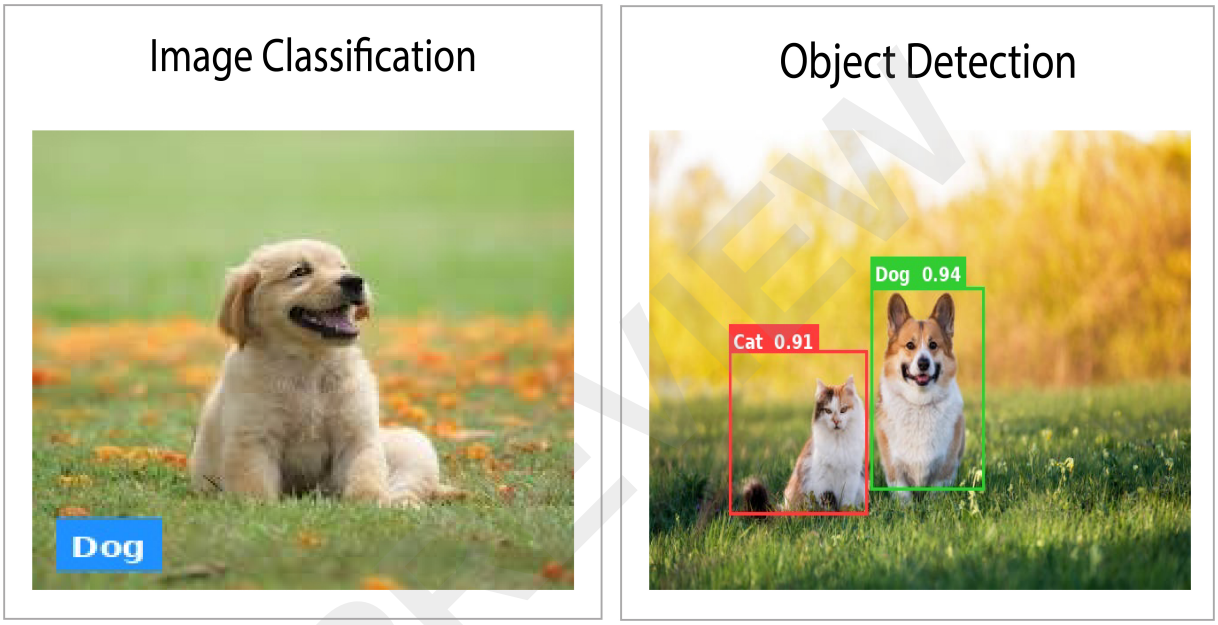

Figure 1.1: *Difference between image classification and object detection. Classification assigns one image-level label, while detection localizes each object with a bounding box, class label, and confidence score.*

1.2 Why Object Detection Matters in Practice

Object detection plays a central role in a wide range of modern intelligent systems. In many applications, understanding *where* something is located is just as important as knowing *what* it is.

Common application domains include:

- **Autonomous and assisted driving**—vehicles, pedestrians, cyclists, and traffic signs must be detected reliably at real-time speeds.
- **Aerial and UAV imagery analysis**—objects appear at varying scales and top-down viewpoints, often against complex backgrounds.
- **Medical imaging**—precise localization of abnormalities supports diagnosis and treatment planning.
- **Surveillance and security systems**—continuous monitoring requires fast and low-false-alarm detection.
- **Industrial inspection**—defects and foreign objects must be found automatically on production lines or airport runways.

Across all these domains, detectors are expected to operate accurately, efficiently, and often in real time, requirements that have strongly influenced the design of modern detection algorithms.

PREVIEW

Chapter 2

Understanding YOLO-Based Object Detection

2.1 From Region Proposals to Unified Detection

Earlier deep learning detectors treated object detection as a two-step process: first generate candidate regions that might contain objects, then evaluate each region independently. While effective, this strategy added complexity and computational overhead that limited real-time applicability.

YOLO-based detectors follow a fundamentally different philosophy. Rather than treating detection as a sequence of independent steps, YOLO formulates the entire task as a single learning problem. The network processes the input image once and directly predicts object locations and categories in a unified pass, allowing the model to reason about global image context during prediction.

2.2 Detection as a Regression Problem

At its core, YOLO treats object detection as a **regression** task. Given an input image, the network learns to predict numerical values that describe the presence and location of objects. These predictions include:

- Bounding box coordinates defining object location and extent.
- Class probabilities estimating the likelihood of each object category.
- A confidence score reflecting whether an object is present and how well the box is predicted.

By framing detection as regression rather than region classification, YOLO avoids redundant processing and enables faster inference. This is especially advantageous for real-time and large-scale applications where latency is a constraint.

2.3 Grid-Based Spatial Representation

To organize predictions spatially, YOLO-style detectors conceptually divide the input image into a grid. Each cell in this grid is responsible for detecting objects whose centers fall within that cell's region. This grid-based structure provides a principled way to distribute detection responsibility across the image and handle multiple objects simultaneously.

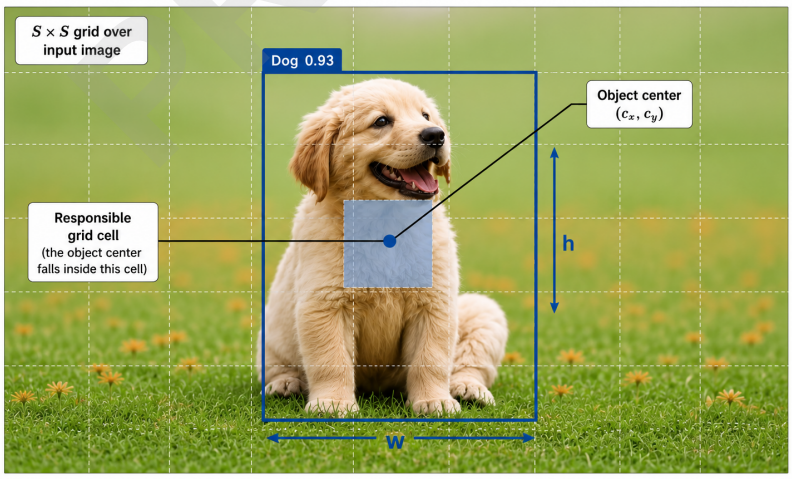

Figure 2.1: *YOLO imposes a conceptual grid over the input image. Each cell predicts bounding boxes only for objects whose center falls within it. The highlighted cell is responsible for the dog shown. The model predicts the normalized center coordinates (c_x, c_y), width w, height h, and a confidence score for each candidate box.*

Although the grid is a conceptual tool rather than a rigid constraint in modern architectures, it captures the spatial reasoning embedded in the detection process.

REVIEW

Chapter 3

Setting Up the Development Environment

3.1 Why Environment Setup Matters

A reliable development environment is a critical foundation for any deep learning project. In object detection workflows, subtle differences in software versions, hardware compatibility, or library configurations can significantly affect training behavior, performance, and reproducibility. Establishing a clean, well-structured setup early avoids a class of problems that are frustrating to diagnose mid-experiment.

3.2 Hardware Considerations

Object detection models are computationally intensive, particularly during training. CPU-only training is feasible for small experiments but quickly becomes impractical: training a YOLO model on a CPU can be tens to hundreds of times slower than on a GPU.

3.2.1 GPU Requirements

For local training, an **NVIDIA GPU** is strongly recommended. The dominant deep learning frameworks are optimized for NVIDIA's CUDA platform. A GPU with at least 6 GB of VRAM is sufficient for initial experiments with smaller model variants at standard resolutions. For larger models or high-resolution datasets, 8–16 GB of VRAM is preferable. Consumer-grade GPUs such as the RTX 3060 or RTX 4060 are adequate for the experiments in this book; research-grade cards such as the A100 or RTX 4090 reduce training time but are not required.

3.2.2 System RAM and Storage

A minimum of 16 GB of system RAM is recommended for loading datasets and running preprocessing pipelines; 32 GB is more comfortable for large datasets. For storage, an SSD is strongly preferred over a spinning hard drive to minimize data-loading bottlenecks during training.

3.3 Cloud-Based Alternatives

Readers without access to a suitable local GPU can use cloud platforms to run experiments without hardware investment:

- **Google Colab** provides free access to NVIDIA GPUs (typically a T4) through a browser-based Jupyter interface. No local installation is required. The free tier imposes session time limits; Colab Pro offers longer sessions.
- **Kaggle Kernels** offers free GPU access (P100 or T4) with up to 30 hours of weekly compute. Many standard detection datasets are already hosted on Kaggle and can be loaded without manual downloading.
- **Paperspace Gradient** and **Lambda Labs** are paid platforms offering persistent environments for longer training runs.

For the experiments in this book, Google Colab or Kaggle are sufficient and are recommended for readers who prefer not to configure a local environment.

3.4 Operating System and Platform Choices

Deep learning frameworks are supported on Linux, Windows, and macOS. Linux-based environments, particularly Ubuntu 20.04 and 22.04 LTS, are most common in research and production settings due to their stability and GPU driver support. Windows is a viable alternative and is supported by all major frameworks discussed here. macOS does not natively support NVIDIA CUDA; Apple Silicon Macs can leverage the MPS backend in PyTorch, though support is less mature. For GPU-intensive training on macOS, a cloud environment is recommended.

3.5 Python and Package Management

Python is the primary language for deep learning development. Python 3.10 is recommended, as it is stable, widely supported, and compatible with all libraries used in this book.

The recommended tool for managing Python environments is **Conda**, available through the lightweight Miniconda installer. Isolated environments prevent version conflicts between projects.

PREVIEW

Chapter 4

Object Detection Datasets

4.1 Why Datasets Matter

The quality, diversity, and annotation structure of a dataset directly determine a model's ability to learn meaningful representations and generalize to new scenarios. Unlike classification, object detection datasets must provide both semantic labels *and* precise spatial annotations for every object in every image. This added complexity makes dataset design and selection especially consequential.

4.2 Dataset Structure and Annotation Formats

An object detection dataset pairs images with annotation files. Each annotation describes objects present in an image, including their category labels and bounding box coordinates. Three annotation formats are commonly encountered:

4.2.1 YOLO Format

The native format for the Ultralytics library. Each image has a corresponding `.txt` file with one line per object:

Listing 4.1: *YOLO annotation format: one line per object, all values normalised.*

```
<class_id> <x_center> <y_center> <width> <height>
```

All spatial values are normalized to $[0, 1]$ relative to the image dimensions. A bounding box centered at pixel $(320, 240)$ in a 640×480 image with dimensions 100×80 pixels would be written as:

```
0  0.500  0.500  0.156  0.167
```

4.2.2 COCO JSON Format

All annotations are stored in a single JSON file. Each entry contains the image ID, category ID, and bounding box in `[x_min, y_min, width, height]` format using absolute pixel coordinates. COCO format is standard for benchmarking and is supported by most evaluation libraries.

4.2.3 Pascal VOC Format

Annotations are stored in individual XML files, one per image. Bounding boxes use absolute pixel coordinates in `[xmin, ymin, xmax, ymax]` form. While less common in modern workflows, VOC format is still encountered in older datasets.

When working with non-YOLO formats, conversion tools are available. The Ultralytics library includes built-in converters, and Roboflow (discussed in Section 4.7) handles format conversion automatically.

4.2.4 Dataset Splits

A well-structured dataset is divided into three non-overlapping subsets. The **training set** is used to update model parameters. The **validation set** monitors generalization during training and guides decisions such as early stopping. The **test set** is held out entirely and used only for final evaluation after training is complete.

Critical rule: the test set must never be used during model development. Evaluating on the test set at any earlier stage invalidates it as an unbiased performance estimate. This is one of the most common sources of overly optimistic reported results.

A common split ratio is 70 % training / 15 % validation / 15 % test, though 80/10/10 is often used for small datasets to maximize training data. The same image must not appear in more than one split.The most appropriate split depends on dataset size, class balance, and whether official benchmark splits already exist.

4.3 The COCO Dataset

The **Common Objects in Context (COCO)** dataset is one of the most influential benchmarks in object detection research. Released by Microsoft Research, it contains over 118,000 training images and 5,000 validation images across 80 object categories covering everyday scenes with multiple objects at different scales and levels of occlusion.

PREVIEW

Chapter 5

YOLO Training and Evaluation Workflow

5.1 Overview of the Pipeline

Training a YOLOv11 model follows a structured pipeline that transforms a labelled dataset into a trained detector capable of generalizing to unseen images. At a high level, five stages are involved:

1. **Dataset preparation and verification:** Images and annotations are organized, validated, and split into training, validation, and test sets.
2. **Model initialization:** An architecture is selected and weights are loaded, typically from a pretrained checkpoint.
3. **Iterative training:** The model processes batches of images, computes a loss, and updates its parameters through backpropagation.
4. **Periodic validation:** Performance is measured on the validation set at the end of each epoch to monitor generalization.
5. **Final evaluation:** The best checkpoint is evaluated on the held-out test set to produce an unbiased performance estimate.

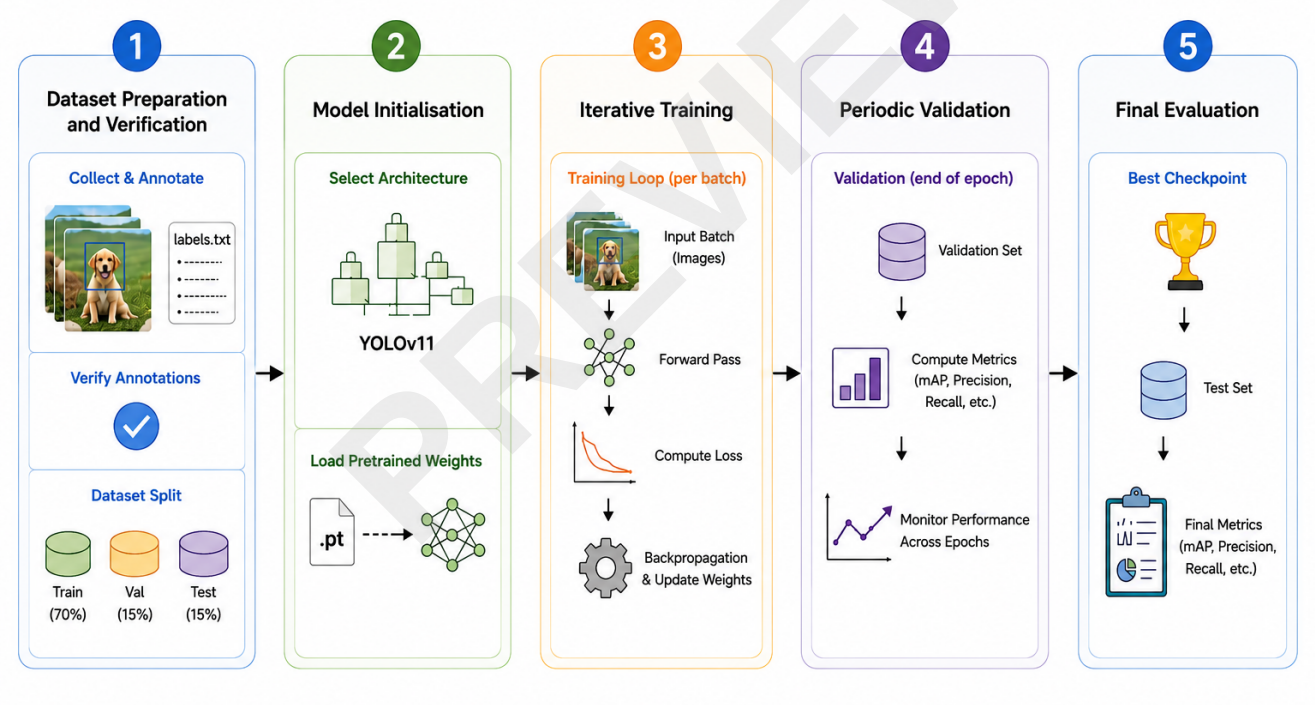

Figure 5.1: *The five-stage YOLOv11 training pipeline. The forward pass, loss computation, and backpropagation cycle repeats for every batch. A separate validation pass at the end of each epoch monitors generalization; the checkpoint with the best validation mAP50 is saved as* `best.pt`*. Final evaluation on the held-out test set is performed only after training is complete.*

5.2 Model Variants and Initialization

YOLOv11 is available in five variants that differ in size, speed, and accuracy. Table 5.1 summarizes their key characteristics.

PREVIEW

Chapter 6

Training YOLO on a Reference Dataset

6.1 Purpose of This Chapter

This chapter puts the concepts from Chapter 5 into practice by executing a complete, reproducible training experiment on COCO from start to finish. Every command shown here is real and executable. By the end, readers will have trained a YOLOv11 model, monitored its progress, evaluated its performance, and understood how to interpret the results; a workflow that transfers directly to custom datasets in Chapter 7.

6.2 Project Directory Structure

A well-organized project makes experiments easier to reproduce and extend. The following layout aligns with Ultralytics conventions:

Listing 6.1: *Recommended directory layout for a YOLO project.*

```
yolo-project/
  data/
    coco/             # downloaded automatically by Ultralytics
  configs/
    coco.yaml         # dataset configuration file
  runs/
    train/            # training outputs (created automatically)
    val/
  requirements.txt
  train.py            # optional: custom training script
```

The `runs/` directory is created automatically. Each experiment is saved in a numbered subdirectory so results from different runs do not overwrite one another.

6.3 Dataset Configuration

The Ultralytics library uses a YAML file to describe the dataset; paths, class count, and class names. For COCO, a built-in configuration handles downloading and setup automatically:

Listing 6.2: *Structure of the COCO dataset configuration YAML.*

```
# coco.yaml
path: ./data/coco          # root directory
train: images/train2017    # relative path to training images
val:   images/val2017      # relative path to validation images
test:  images/test2017
nc: 80                     # number of classes
names:
  0: person
  1: bicycle
  2: car
  # ... 80 classes total
```

The `nc` value must exactly match the number of entries in `names`. A mismatch is one of the most common configuration errors and will cause training to fail or produce incorrect class predictions silently.

The full COCO dataset is approximately 20 GB, so a stable internet connection and sufficient disk space are required for the first download.

6.4 Model Selection and Initialization

For a reference experiment, `yolo11s.pt` is a sensible starting point, small enough to train in a reasonable time on a mid-range GPU while large enough to achieve meaningful performance on COCO:

Listing 6.3: *Loading a pretrained YOLOv11s model and inspecting it.*

```
from ultralytics import YOLO
```

PREVIEW