Bullseye Detection using YOLOv8 and ONNX

Saksham Khandelwal

April 10, 2025

Contents

1 Introduction

Accurately detecting bullseye patterns in real-time is crucial for applications such as autonomous navigation, target tracking, and precision analysis. However, existing object detection models lack specialized training for bullseye recognition, leading to suboptimal performance in varying environmental conditions. This project aims to develop a robust bullseye detection system using computer vision and deep learning.

2 Dataset Preparation

2.1 Sources and Collection

Bullseye images were collected from:

- Photos of hand-drawn bullseyes taken under different lighting, backgrounds, and angles
- Google Images (Red and White bullseyes)
- Roboflow Universe

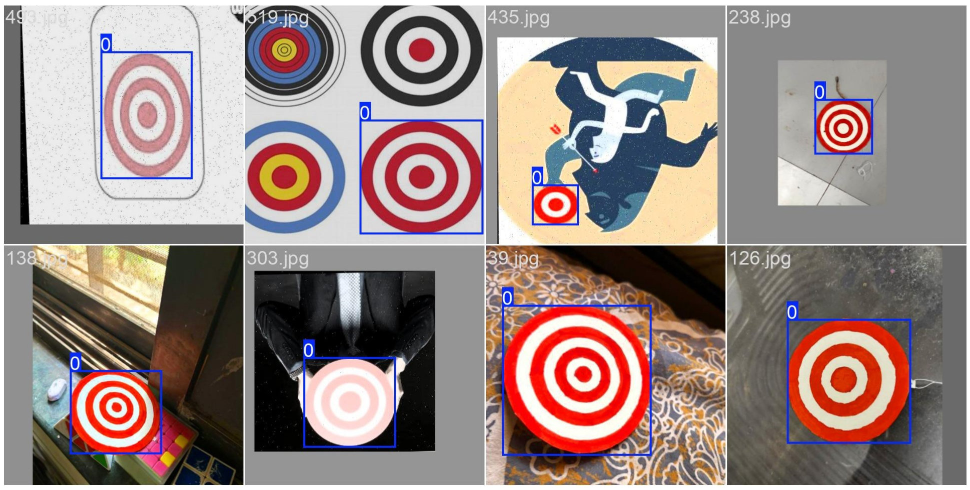


Figure 1: Sample bullseye training images with annotations

2.2 Preprocessing and Normalization

All images were converted to JPEG format with a resolution of 768x1024 (same aspect ratio as iPhone photos). The Roboflow dataset had a resolution of 640x640 and was not resized to the same aspect ratio as they were added later.
PIL.Image and pillow_heif were used to convert all images to.jpg format and, in addition, OpenCV was used to convert all to the same aspect ratio and renamed.

2.3 Annotations and Splitting

Annotations for the bullseye dataset were created using **CVAT (Computer Vision Annotation Tool)**. The annotations were saved in YOLO format to be compatible with the training framework.
All images and labels were split into ‘train‘ and ‘validation‘ folders with the following config:

```
path: C:\Users\...\Output
train: images/train
val: images/validation

names:
    0: bullseye
```

3 Model Training

YOLOv8m was used for training. It was chosen over nano or small because it offers a balance between speed and accuracy, making it suitable for real-time inference with reliable performance. The training followed the method in this tutorial.

CUDA Check and YOLOv8 Training Script

```
import torch
print("CUDA␣available:", torch.cuda.is_available())
print("Device:", torch.cuda.get_device_name(0) if torch.cuda.
   is_available() else "CPU␣only")

from ultralytics import YOLO

model = YOLO("yolov8m.yaml")

results = model.train(
    data="config.yaml",
    epochs=100,
    batch=32,
    workers=4,
    device=0
)
```

While only a few parameters such as model size, epochs, and batch size were explicitly modified, YOLOv8 includes many other hyperparameters that were kept at their default values due to their strong generalization across datasets. These include:

- Learning Rate
- Momentum
- IoU Thresholds

Parameter	**Value**
Model Architecture	YOLOv8m
Batch Size	32
Epochs	100
Workers	4
Optimizer	SGD (default YOLO)
Loss Functions	Box, Classification, DFL
Device	NVIDIA RTX 4070

Table 1: Training Configuration Summary

Data Augmentation Techniques

YOLOv8 applies a set of data augmentation techniques during training to improve generalization and robustness without requiring manual configuration. For this project, no custom augmentations were set, and all augmentations were used with their default values as configured by the YOLOv8 framework.
These default augmentations include:

- **Mosaic Augmentation (enabled)** – Combines 4 images into one during training to enhance context diversity and scale variation.
- **HSV Augmentation (default: `hsv_h=0.015, hsv_s=0.7, hsv_v=0.4`)** – Applies random changes to hue, saturation, and brightness to simulate lighting changes.
- **Horizontal Flip (default: `flipud=0.0, fliplr=0.5`)** – Randomly flips images horizontally with a 50% probability.
- **Scaling and Translation (enabled)** – Random zoom and shifts are applied as part of the mosaic and image transformations.
- **Rotation (enabled)** – Slight random rotation is applied as part of image warping in augmentations.
- **Shear (default: `degrees=0.0`)** – Disabled by default but can be enabled for angular distortions.
- **Perspective and Stretch (enabled)** – Applies slight perspective and aspect ratio shifts.
- **MixUp (default: disabled)** – Not used by default in YOLOv8.

These augmentations are handled internally during training and help prevent overfitting while improving the model's ability to detect bullseyes in varied scenarios such as lighting changes, partial occlusion, or unusual orientations.

Model Performance Assessment

The trained YOLO model was evaluated using standard object detection metrics such as precision, recall, mean Average Precision at IoU threshold 0.5 (mAP@0.5), and mean Average Precision averaged across IoU thresholds from 0.5 to 0.95 (mAP@0.5:0.95). At the final epoch, the model achieved

Precision : 0.995

Recall : 1

mAP@0.5 : 0.995

mAP@0.5:0.95 : 0.9785

These values indicate that the model is highly accurate in detecting and localizing objects. Additionally, the **training and validation losses**—including box loss, classification loss, and distribution focal loss (DFL)—remained low, suggesting that the model generalized well to unseen data. The loss values at the final epoch were:

- **Validation box loss:** 0.2176
- **Validation classification loss:** 0.1736
- **Validation DFL:** 0.8467

These results demonstrate that the model has successfully learned robust features for object detection, and the high mAP scores confirm its reliability for deployment or further inference tasks.

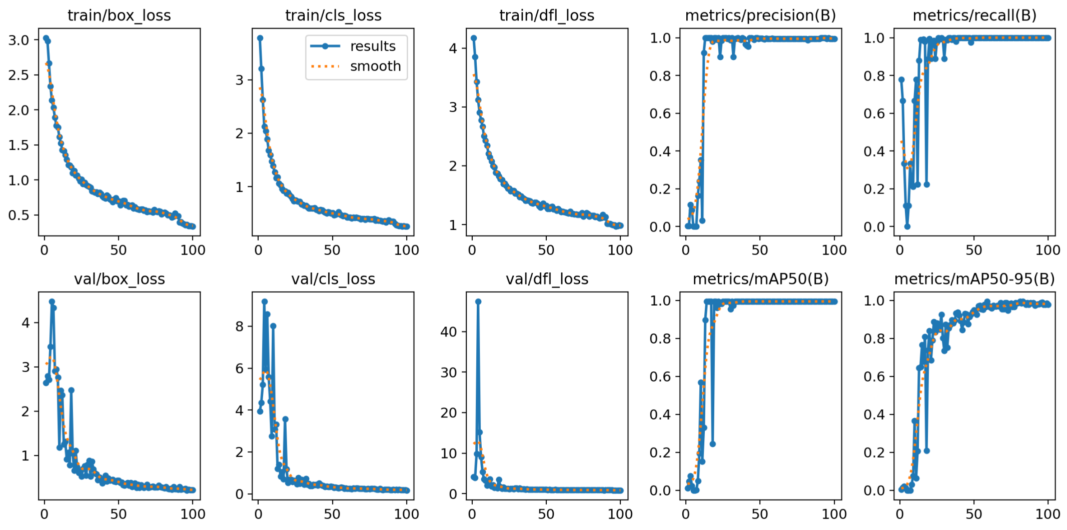


Figure 2: Training metrics plot generated by YOLOv8

4 ONNX Conversion

ONNX (Open Neural Network Exchange) format allows interoperability between different deep learning frameworks and is optimized for deployment on devices with limited resources. Converting to ONNX facilitates real-time inference and integration into production pipelines.
The trained `best.pt` model was converted to ONNX format for optimized inference.

```
yolo export model=runs/detect/train/weights/best.pt format=onnx
```

The exported ONNX model was 98MB, making it lightweight enough for edge deployment.

5 Conclusion

This project successfully built a bullseye detector using YOLOv8 with custom training data. The model achieved high accuracy, and inference on video demonstrated robust detection capabilities.

EXTRA: Model Evaluation and Testing

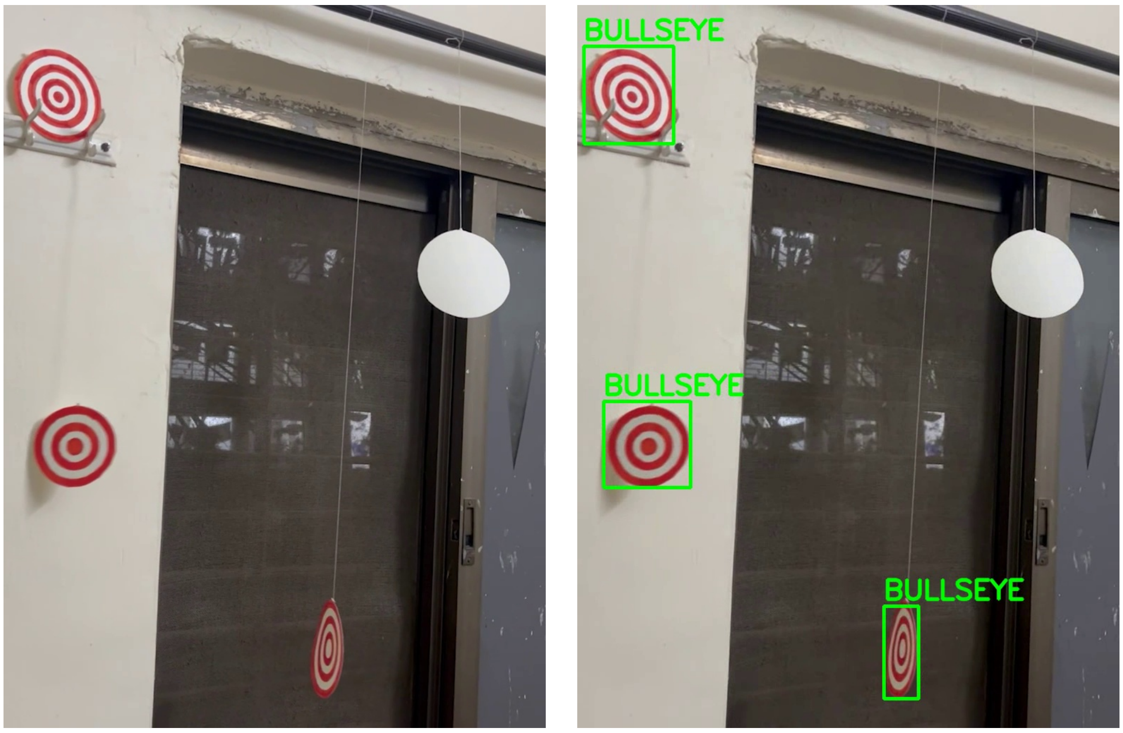
After training, the model was tested on a custom input video. The detection results are visualized frame-by-frame in a generated output video.

Video Inference Script

The trained model was loaded using YOLO and applied to video frames captured via cv2.VideoCapture. The processed frames were then saved as an MP4 file to visualize the model's detection performance.

Side-by-Side Comparison: Input vs Output

Below is a visual comparison between the original input video and the output after YOLOv8 detection.



Original Video Frame **Detection Output Video Frame**

Figure 3: Comparison of original and detected video output