Object Detection: How to Effectively Implement YOLOv8

This article is a practical guide that explains how to use the command-line interface and Python to detect objects in images, videos, and real-time network cameras.

Introduction:

Object detection is a subfield of computer vision that primarily involves identifying and locating objects in images or videos with a certain degree of confidence. The recognized objects typically come with a bounding box, providing information about the nature and position of the objects in the scene.

Since its debut in 2015, YOLO (You Only Look Once) has astounded the computer vision community with its remarkable accuracy and speed in achieving real-time object detection. Since then, YOLO has undergone multiple improvements, enhancing prediction accuracy and efficiency, ultimately introducing its latest member: YOLOv8 by Ultralytics.

YOLOv8 comes in five versions: nano (n), small (s), medium (m), large (l), and extra-large (x). Their improvements can be assessed through their average precision (mAP) and latency on the COCO val2017 dataset when compared to previous versions.

Compared to its predecessors, YOLOv8 is not only faster and more accurate, but it also requires fewer parameters to achieve this performance. Additionally, it is equipped with an intuitive command-line interface (CLI) and a Python package, offering a more seamless experience for users and developers.

In this article, I will demonstrate how to use the CLI and Python to apply YOLOv8 for object detection in static images, videos, and real-time network cameras.

Installation

To get started with YOLOv8, all you need to do is run the following command in your terminal:

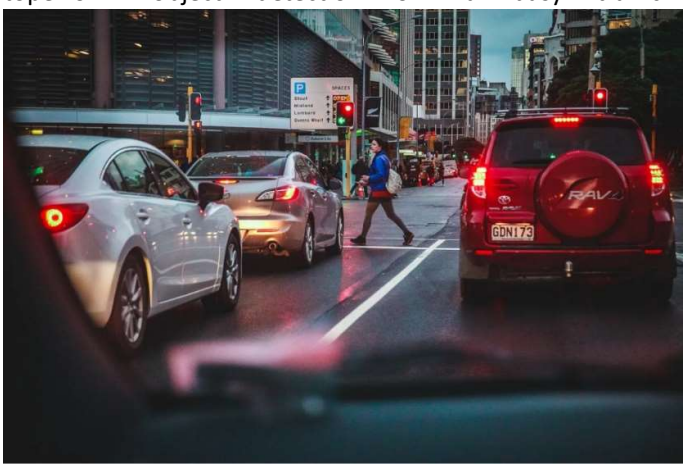
pip install ultralytics

This will install YOLOv8 through the ultralytics pip package.

Image Detection

Object detection in static images has proven to be valuable in various fields such as surveillance, medical imaging, or retail analytics. Regardless of the domain in which you intend to apply your detection system, YOLOv8 makes it easy for you. Here's an example of the original image we want

toperform object detection on: a busy traffic photograph in a crowded city.



To run YOLOv8, we will explore two implementation methods: CLI and Python. While we are using a jpg image in this specific case, YOLOv8 supports various image formats.

CLI

Assuming we want to run YOLOv8x (the extra-large version) on an image named "img.jpg," you can input the following command in the CLI:

Here, we specify the following parameters:

"detect" is used for object detection.

"predict" is used to perform prediction tasks.

"model" is used to select the model version.

"source" is used to provide the file path to our image.

"save" is used to save the processed image with its object bounding boxes and their predicted classes and class probabilities.

Python

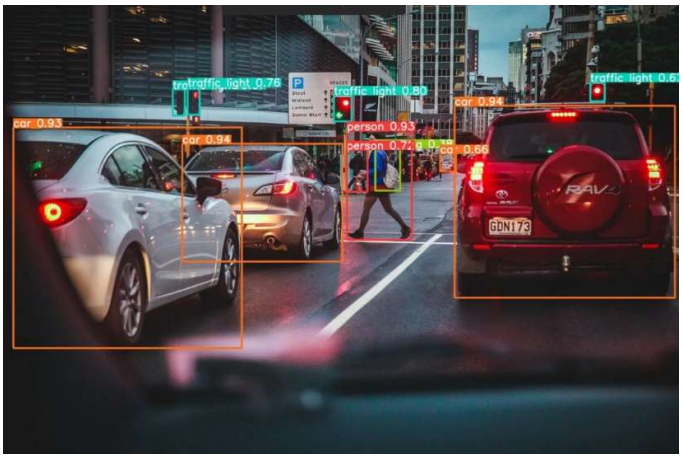
In Python, you can achieve the exact same task with an intuitive and low-code solution as follows:

```
from ultralytics import YOLO

model = YOLO('yolov8x.pt')

results = model('img.jpg', save=True)
```

Whether you are using the CLI or Python, in either case, the saved processed image would look as follows:



We can clearly see the bounding boxes around each detected object, along with their corresponding class labels and probabilities.

Video Detection

Performing object detection on a video file is very similar to working with image files, with the only difference being the source file format. Just like with images, YOLOv8 supports various video formats that can be processed as inputs for the model. In our case, we'll be using an mp4 file.

Let's once again explore CLI and Python implementations. To speed up the computation, we will now use the YOLOv8m model instead of an extra-large version.

CLI

yolo detect predict model=yolov8m.pt source="vid.mp4" save=True

python

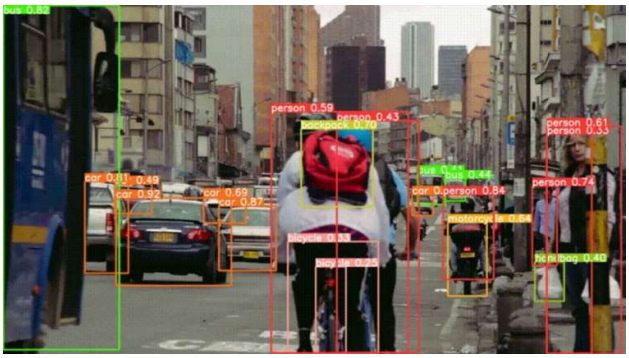
from ultralytics import YOLO

model = YOLO('yolov8m.pt')

results = model('vid.mp4', save=True)

First, before performing object detection, let's take a look at our original "vid.mp4" file:

The video displays a busy city scene, including cars, buses, trucks, and people riding bicycles, as well as some people on the right who appear to be waiting for a bus. After processing this file using the YOLOv8 medium version, we obtain the following results.



Similarly, we can see that YOLOv8m performs very well in accurately capturing objects within the scene. It can even detect smaller objects, like backpacks worn by people riding bicycles, as part of a larger whole.

Real-Time Detection

Finally, let's take a look at what's involved in detecting objects in a live network camera video. For this, I'll be using my personal network camera, and as before, there are both CLI and Python methods available. To minimize latency and reduce lag in the video, I'll be using the lightweight YOLOv8 nano version.

CLI

yolo detect predict model=yolov8n.pt source=0 show=True

Most of these parameters are the same as those used for images and video files that we saw earlier. The only difference is the "source" parameter, which allows us to specify which video source to use. In my case, it's the built-in network camera (0).

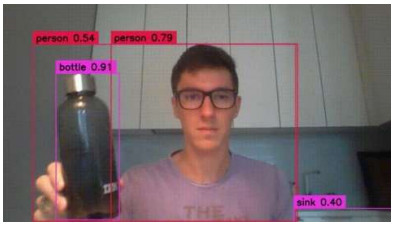
python

from ultralytics import YOLO

model = YOLO('yolov8n.pt')

model.predict(source="0", show=True)

Similarly, we can accomplish the same task with an extremely low-code Python solution. Here's an example image from YOLOv8n on a live network camera source.



Despite the lower video quality and challenging lighting conditions, YOLOv8 is still able to capture objects well and can even detect some objects in the background, such as the olive oil and vinegar bottles on the left and the sink on the right.

It's worth noting that while these intuitive CLI commands and low-code Python solutions are great ways to quickly get started with object detection tasks, they have some limitations when it comes to customizing configurations. For example, if you want to configure the aesthetics of the bounding boxes or perform simple tasks like counting and displaying the number of detected objects at any given time, you would need to write your own custom implementation using packages like cv2 or supervision.

In fact, the network camera recording shown above was captured using the following Python code to adjust the camera's resolution and customize the definition of bounding boxes and their annotations. (Note: This was primarily done to make the GIF above more expressive. The CLI and Python implementations shown earlier are sufficient to produce similar results.)import cv2

```
import supervision as sv

from ultralytics import YOLO

def main():

    # to save the video

    writer= cv2.VideoWriter('webcam_yolo.mp4',

                            cv2.VideoWriter_fourcc(*'DIVX'),

                            7,

                            (1280, 720))

    # define resolution

    cap = cv2.VideoCapture(0)

    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)

    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

    # specify the model

    model = YOLO("yolov8n.pt")

    # customize the bounding box

    box_annotator = sv.BoxAnnotator(

        thickness=2,

        text_thickness=2,

        text_scale=1

    )

    while True:

        ret, frame = cap.read()
```

```
        result = model(frame, agnostic_nms=True)[0]
        detections = sv.Detections.from_yolov8(result)
        labels = [
            f"{model.model.names[class_id]} {confidence:0.2f}"
            for _, confidence, class_id, _
            in detections
        ]
        frame = box_annotator.annotate(
            scene=frame,
            detections=detections,
            labels=labels
        )
        writer.write(frame)


        cv2.imshow("yolov8", frame)
        if (cv2.waitKey(30) == 27): # break with escape key
            break
    cap.release()
    writer.release()
    cv2.destroyAllWindows()
if __name__ == "__main__":
    main()
```

While the details of these codes go beyond the scope of this article, the provided reference is a valuable resource for those interested in enhancing their object detection capabilities using similar methods:

https://youtu.be/QV85eYOb7gk

Conclusion

YOLOv8 not only outperforms its predecessors in terms of accuracy and speed but also greatly improves the user experience through its easy-to-use CLI and low-code Python solutions. It offers five different model versions, giving users the flexibility to choose according to their specific needs and tolerance for latency and accuracy.

Whether your goal is to perform object detection on static images, videos, or live network cameras, YOLOv8 is capable of seamless execution. However, if your application requires custom configurations, you may need to use other computer vision packages such as cv2 and supervision.