YOLO: Real Time Object Detection

Dilip Epparapalli
Department of electrical and computer engineering
George Mason University
depparap@gmu.edu

***Abstract*—This study explores the YOLO (You Only Look Once) framework, a cutting-edge method in real-time object detection, by first examining its operational mechanics and then applying a pretrained model from the COCO database for object detection via webcam. The paper further extends to training the YOLO model on a custom dataset using the Basys 3 board, where the model's performance is scrutinized on both small and large datasets. Comapring the results from both large and small datasets and explaining the results.**

***Index Terms*—dataset, object detection, bounding boxes, COCO, train**

I. Introduction

The YOLO (You Only Look Once) [1] framework is a new and effective way to detect objects in real time using computer vision. What makes YOLO special is how it combines finding where objects are and identifying what they are into one quick step. Unlike older methods that looked at parts of an image separately, YOLO looks at the whole image at once, making it faster and more efficient.

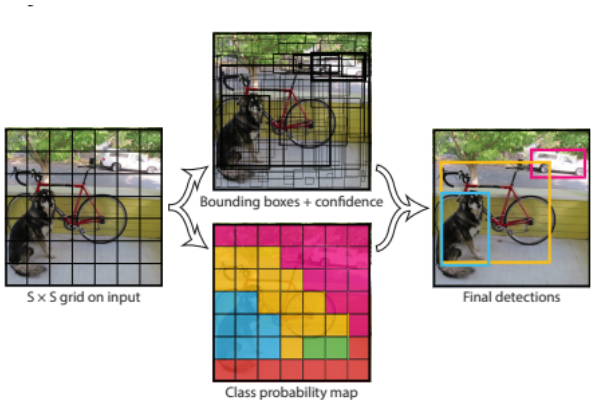

Fig. 1. How YOLO WORKS

Detecting objects in a picture using YOLO involves a systematic process that capitalizes on the model's unified approach and grid-based methodology. In the first step, the input image is divided into a grid, and each grid cell is tasked with predicting multiple bounding boxes along with associated class probabilities figure 1. These bounding boxes essentially act as candidates for potential objects present in the image. YOLO employs a sophisticated neural network that directly processes the entire image in one go, making predictions for each grid cell simultaneously. Each bounding box prediction includes information about the object's spatial location, size, and a confidence score indicating the model's certainty about the presence of an object within that bounding box. This simultaneous consideration of the entire image and the utilization of multiple bounding box predictions per grid cell enable YOLO to efficiently handle scenarios with multiple objects of different classes in a given picture.

Following the bounding box predictions, a post-processing step is crucial for refining the results. Non-Maximum Suppression (NMS) is employed to filter out redundant or low-confidence predictions. NMS identifies the most confident bounding box for each object and suppresses overlapping or less certain boxes. The result is a clean and accurate set of object predictions, each associated with a high-confidence bounding box. This elegant yet powerful process allows YOLO to achieve real-time object detection, making it well-suited for applications ranging from surveillance and autonomous vehicles to image analysis and more. The ability to seamlessly integrate detection and classification within a unified network sets YOLO apart as an efficient and versatile solution for object detection in images.

In this project, we use the COCO (Common Objects in Context) dataset, which is a large collection of images used for object detection. This dataset is important because it has many different kinds of images, making it great for testing how well our object detection works in various situations. I use a version of YOLO [3] [4]that's already been trained on this dataset to detect objects in real-time with a webcam. This part of the project shows how well YOLO works in everyday situations.

I also take a step further by training YOLO on two sets of custom images [2]. The first set, which we call the 'small' dataset, has regular pictures of the Basys 3 board. The second, the 'large' dataset, includes these pictures plus more that we changed slightly to make the dataset more challenging. This lets us see how well YOLO can learn and adapt when it's given more information.

The important part of this project is to compare how well YOLO works with these two different sets of images. I want to see if it does better with more images and more variety. This comparison will help us understand how flexible YOLO is and how it might be used in different real-world situations.

II. Pretrained Model

A. Working model of YOLO V3

In this project, I started by exploring how well an older version of YOLO, called YOLOv3, works in detecting objects.

YOLOv3 is one of the first models in the YOLO series. We used a ready-made YOLOv3 model [5], which already knew how to identify certain objects, and we tried it out on different pictures. Our goal was to see if this older YOLO model could still do a good job. The results were quite good and showed us that YOLOv3, even though it's old, works well for finding objects in images.

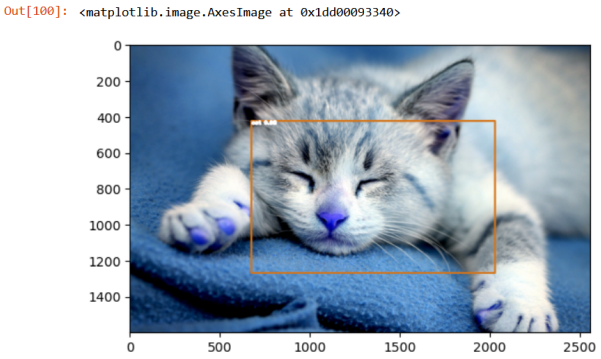

Fig. 2. Testing YOLO V3 on images

B. Working model of YOLO V8

After that, we moved to a newer and better version of YOLO, known as YOLOv8. This version is made by a group called Ultralytics [3] and is much more advanced than YOLOv3. For YOLOv8, we also used a model that was already trained to recognize objects using a big collection of images called the COCO database. We tested YOLOv8 by using a webcam [4]to see if it could detect objects in real-time. It turned out really well! The YOLOv8 model could quickly and accurately spot the objects that are known in the COCO database.

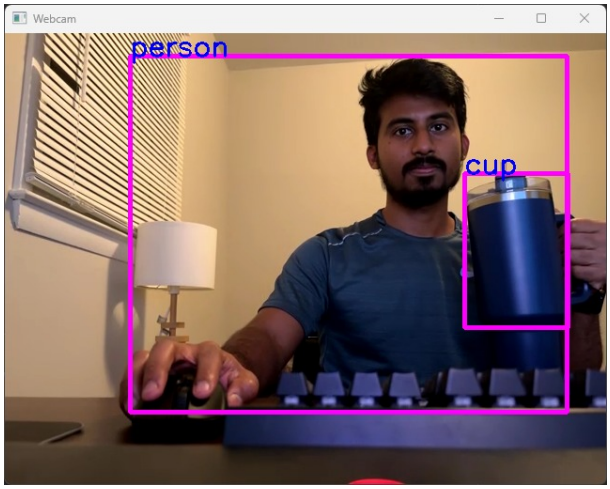

Fig. 3. Testing YOLO V8 on WebCam

By comparing these two versions of YOLO, we learned a lot about how the YOLO technology has improved over time. Our tests showed that the new version, YOLOv8, is better at recognizing objects, especially in real-world situations like using a webcam.

III. Training YOLO on Custom Data

A. Small Dataset

To make a computer model that can spot a Basys 3 board—a type of electronic board—in pictures, I needed to teach it what to look for. I started by gathering around 100 photos of this board. Some of these photos were taken by me, and others were found online to make sure we had lots of different types of pictures. This helps the model learn to recognize the board in many situations.

The below figures shows how I collected different set of images.

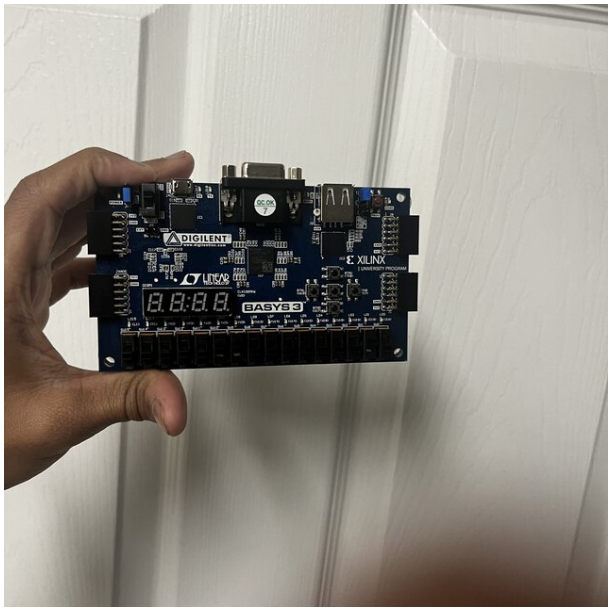

Fig. 4. Collecting sample 1

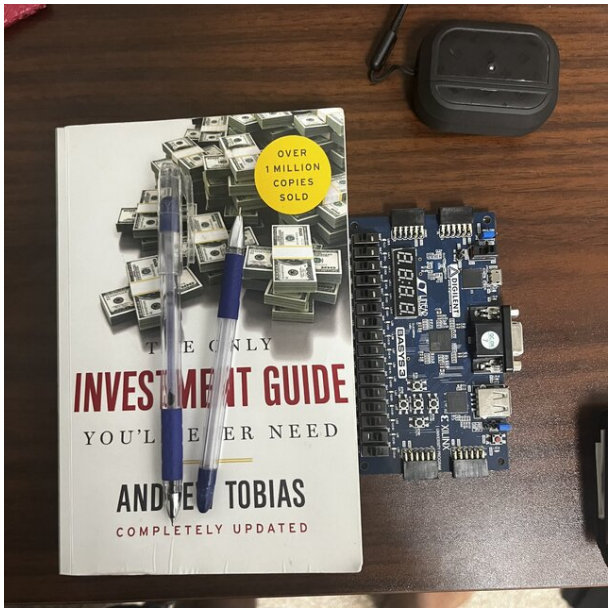

Fig. 5. Collecting sample 2

To teach the model, we first had to label our images, which means drawing boxes around the Basys 3 board in each photo and saying, "This is the board." We used a handy tool called

Roboflow for this. Roboflow is a tool that makes it easier to prepare images for a computer model to learn from. It lets you draw those boxes, split the images into groups (train, valid, and test), and get everything ready for training.

Creating a project in Roboflow is pretty straightforward, and we made ours public so anyone can take a look at it and use it if they want. Although labeling each photo by hand was a long task—taking 4-5 hours—Roboflow made it a lot simpler than it could have been.

Roboflow has this great feature where you can use API keys—a kind of special code—to download your prepared images directly into your code, which I found very easy to use.

Below are figures that shows the annotated images

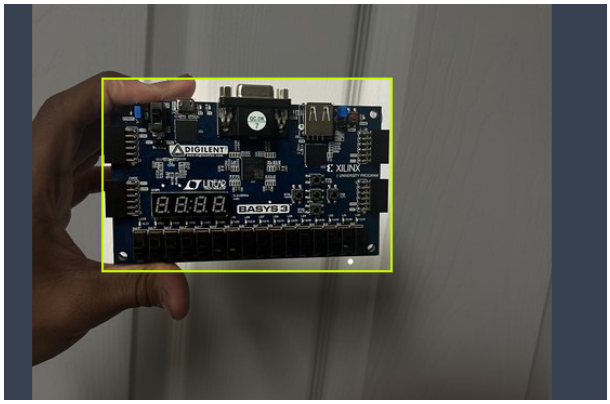

Fig. 6. annotated sample 1

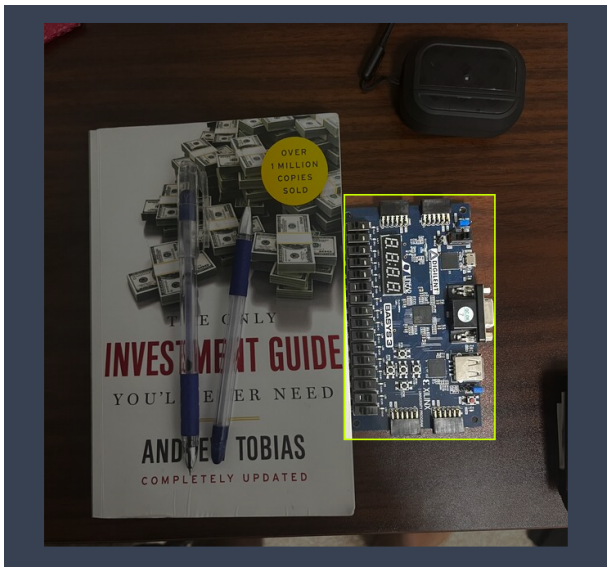

Fig. 7. annotated sample 2

We called our collection of around 300 images the ”small dataset” because it’s not a huge number of images. Once, I got these images from Roboflow, we changed some settings to make sure the computer model knew where to find the train, valid, and test images.

After preparing my YOLOv8 model, I trained it on GMU Hopper, a super powerful machine. At first, I tried using the CPU, but it was quite slow—it took a whopping 10 minutes just to finish three epochs. Then, I switched to using the graphics processor GPU on GMU Hopper, and the magic happened! The model breezed through 100 epochs in just 5 minutes.

Once the training wrapped up, I ended up with a special set of learned patterns.These patterns are called ”weights,” and they are what make the model so clever. I then used these weights to run our model. To see how well it could spot the Basys 3 board in new pictures from a webcam, I joined forces with the COCO dataset,so combining it with our trained model was like giving it extra knowledge.

Below figure shows the result of the custom trained YOLO model

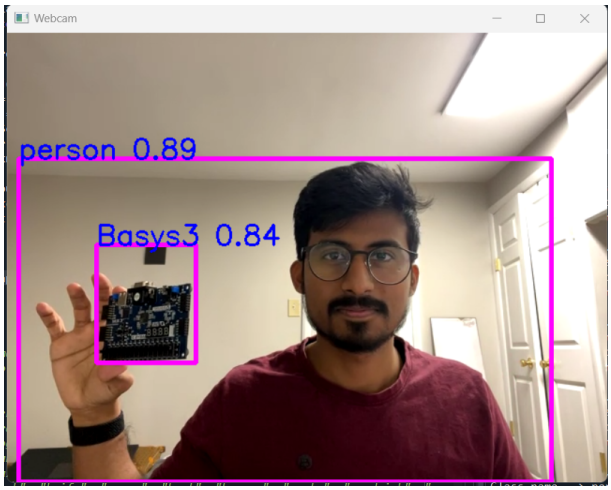

Fig. 8. Samll Dataset Result

B. large Dataset

Expanding the Dataset and Training with Augmentation

Once I finished training our computer model to recognize the Basys 3 board with a small set of pictures,I decided to make the model even better. So, we gathered twice as many pictures as before to create a larger dataset. This helps the model learn from more examples and get better at spotting the board in different situations.

To make sure our model doesn’t just memorize the pictures but learns to recognize the board no matter how it’s placed or turned, I used a technique called data augmentation. This is like showing the model a picture and then showing it the same picture but flipped upside down or sideways. I flipped the pictures horizontally and vertically and also rotated them by 90, 180, and 270 degrees. This way, the model can learn to recognize the board from all angles.

Roboflow, the tool we used before, helped us with this augmentation. It’s like a helper that takes our pictures and applies these flips and rotations automatically, which saves a lot of time

The below figure show how I augumented the images.

After we got all our pictures ready and split them into training, validation, and testing groups,I downloaded them just like we did with the small dataset. Then we were ready to train the model again, this time using GMU Hopper which

Fig. 9. Augumented Result

can process lots of pictures really fast. This meant that even though we had more pictures now, it didn't take too long to train the model.

Once the training was done, I picked the best set of learned patterns—these are called "weights"—and used them to run our model with a webcam. I found that the model, which was now trained with more pictures and had seen the board from all different angles, was better at recognizing the board than before when we only used the small set of pictures.

So, in simple words, by showing our model more and varied pictures of the board, it got smarter and could recognize the board better when we tried it out in real life with a webcam.

The below figure shows the result of model that has been trained with large dataset.

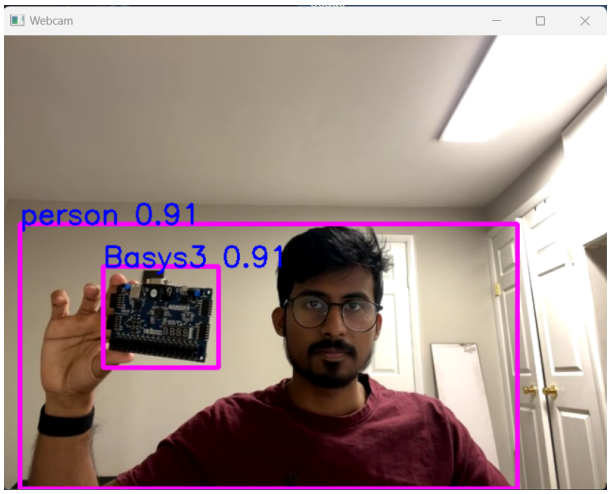

Fig. 10. Large Datset results

C. comparison of results from both datasets

- Precision Recall Curve: Both datasets yield very high precision, but the large dataset's slightly higher mAP suggests that the additional data variety helps the model's ability to discern and detect objects. The fact that the

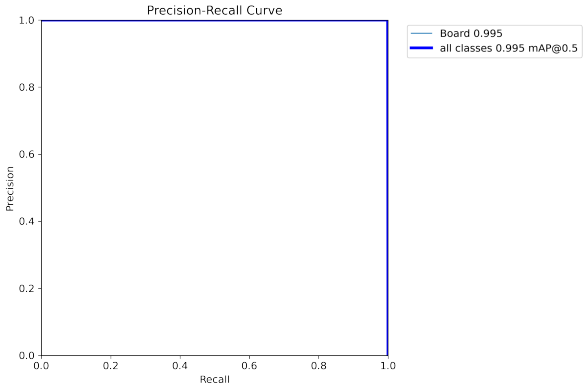

Fig. 11. PR curve for Large Dataset

large dataset's curve is almost indistinguishable from the small dataset's indicates that the model has effectively captured the features necessary for detection from the small dataset and that the benefits of additional data in the large dataset are marginal in this case. It's worth noting

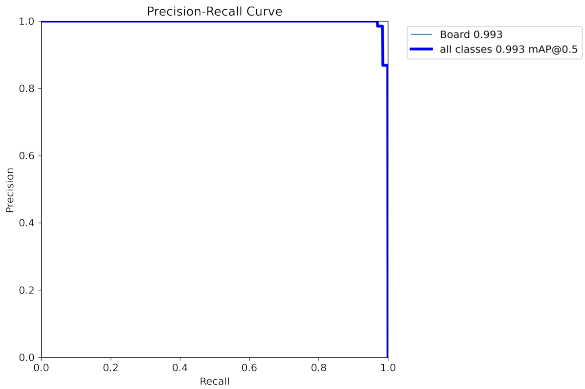

Fig. 12. PR curve for Small Datset

that while the small dataset seems almost as effective as the large one in this context, in practice, the robustness and generalizability of the model are often better served by larger and more varied datasets. This is especially true when the object of interest can appear in diverse scenarios not captured by the small dataset.

- F1 Curve: While both datasets yield high F1 scores, the large dataset achieves a perfect F1 score at a higher confidence level, which may suggest that the model is more discerning and precise when additional variety (through augmentation) is introduced. The model trained on the small dataset is still highly accurate but reaches its peak performance at a lower confidence threshold. This could mean that it doesn't require as much certainty to make a prediction, possibly due to less variability in the dataset. The sharp drop-off at high confidence levels for both datasets indicates that setting the confidence threshold too high could lead to missed detections (lower recall), as the model becomes too stringent in its predictions.

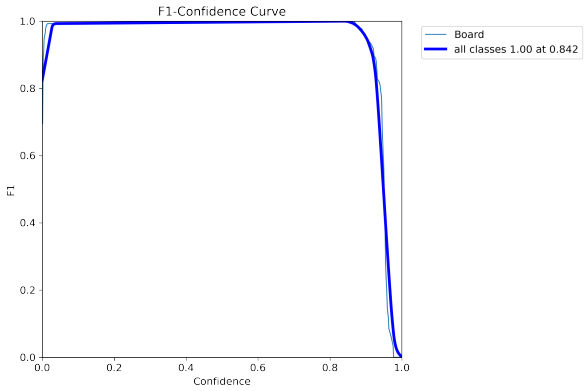

Fig. 13. Large Datset F1 results

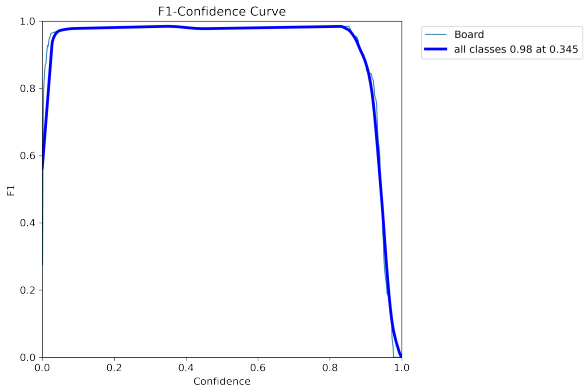

Fig. 14. Small Datset F1 results

- Results: Both sets of graphs for small and large datasets show variations in the raw results, which is normal when conducting multiple tests or measurements. The 'smooth' lines help us understand the overall trend by reducing the noise in the data.
  Small Dataset Graphs:
  The graphs related to the small dataset show a greater degree of fluctuation in the results, as seen by the vertical spread of the data points. This could indicate less consistency in the performance across different tests or under different conditions. The 'smooth' lines in the small dataset graphs show that despite the variability, there is a discernible trend that can be followed, which might represent a general performance pattern.

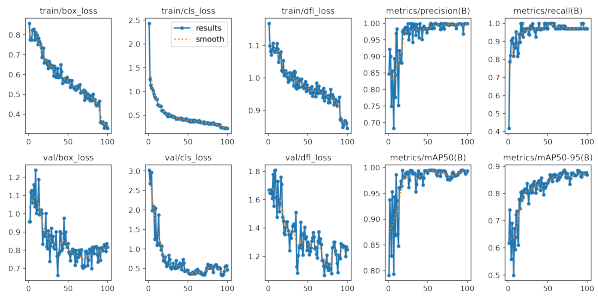

Fig. 15. Small Datset results

Large Dataset Graphs:
The large dataset graphs exhibit a tighter clustering of data points around the 'smooth' line, suggesting more consistent results across the various tests or measurements. This consistency could be due to a larger amount of data providing a more stable performance indicator. The 'smooth' lines in the large dataset graphs appear steadier and less affected by outliers, which can be interpreted as the model performing more reliably.

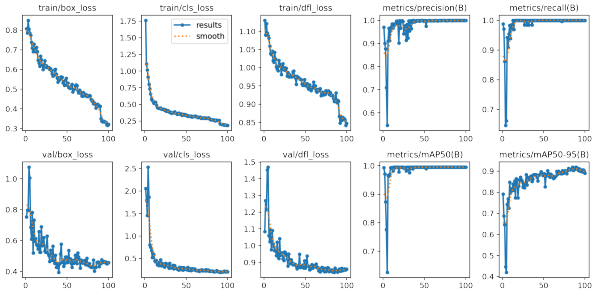

Fig. 16. Large Datset results

Overall Performance:
For both datasets, the performance seems to be converging to a stable state as indicated by the 'smooth' lines, which is a positive sign in model training and testing.
The large dataset seems to provide a more reliable set of results, which is often the case when the model is trained with more data. More data typically leads to better model generalization, which in turn can lead to more consistent performance.

TECHNICAL OR CODE PART

Using Roboflow made things super easy! Instead of tediously downloading the dataset manually and then uploading it to Hopper, I used a clever trick. With the help of Roboflow, I wrote a bit of code that directly fetched the dataset from its repository.

The figure 17 below illustrates this process. It's like a snapshot of how the dataset gets pulled from Roboflow using code. For security reasons, I've removed my API key, which is like a secret passcode that ensures only authorized users can access the dataset. So, think of it as a smoother and more secure way of getting the data we need for training our model.

```
!pip install ultralytics==8.0.196
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="[redacted]")
project = rf.workspace("georgemasonuniversity").project("basys-3-board")
dataset = project.version(2).download("yolov8")
```

Fig. 17. Downloading Dataset from Roboflow using code

```
from ultralytics import YOLO
import torch

# Load a model
model = YOLO("yolo-Weights/yolov8n.pt")  # build a new model from YAML
# print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(model.device)
model.to(device)
print(model.device)

# Train the model
results = model.train(data=r'Basys-3-Board-small-dataset/data.yaml', epochs=100, imgsz=640)
```

Fig. 18. Training the model

The figure 18 illustrates how the model is trained. The model is trained for 100 epochs.

The figure 19 illustrates the results of the above model training which was trained on GMU hopper account where I used GPU's to train my model.

```
100 epochs completed in 0.045 hours.
Optimizer stripped from runs/detect/train7/weights/last.pt, 6.3MB
Optimizer stripped from runs/detect/train7/weights/best.pt, 6.3MB

Validating runs/detect/train7/weights/best.pt...
Ultralytics YOLOv8.0.196 🚀 Python-3.9.9 torch-2.0.1+cu118 CUDA:0 (NVIDIA A100-SXM-80GB, 81251MiB)
Model summary (fused): 168 layers, 3005843 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 3/3 [00:00<00:00,
7.23it/s]
                   all         67         66      0.983      0.985      0.993      0.884
Speed: 0.7ms preprocess, 0.5ms inference, 0.0ms loss, 0.5ms postprocess per image
Results saved to runs/detect/train7
```

Fig. 19. Training results

The figure 20 illustrates how the boxes are printed around the objects.The program is dealing with "boxes" in images. These boxes are like rectangles drawn around things that the model has found in a picture. The model can find objects like cars, people, or anything else, and it tries to draw a box around them.

Now, for each box, the program does a few things. First, it looks at the four corners of the box (x1, y1, x2, y2), which are like the points that define the box's shape. Then, it checks how confident the model is that it found something important in that box. If the confidence is too low, it decides to ignore that box.

If the model is confident enough, it prints out the confidence level and the type of thing it thinks it found (like a car or a person). If the type of thing matches what the program is looking for (stored in a list called "detect"), it draws a colorful rectangle around that thing and writes down some details, like the type of thing and how sure the computer is.

```python
def print_box(boxes,classNames):
    for box in boxes:
        try:
            # bounding box
            x1, y1, x2, y2 = box.xyxy[0]
            x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)  # convert to int values

            # put box in cam


            # confidence
            confidence = math.ceil((box.conf[0] * 100)) / 100
            if confidence<0.5:
                continue
            print("Confidence --->", confidence)

            # class name
            cls = int(box.cls[0])
            print("Class name -->", classNames[cls])

            if classNames[cls] not in detect:
                continue

            cv2.rectangle(img, (x1, y1), (x2, y2), (255, 0, 255), 3)

            # object details
            org = [x1, y1]
            font = cv2.FONT_HERSHEY_SIMPLEX
            fontScale = 1
            color = (255, 0, 0)
            thickness = 2

            cv2.putText(img, classNames[cls]+" "+str(confidence), org, font, fontScale, color, thickness)
        except Exception as e:
            print(e)
```

Fig. 20. printing boxes code

The figure 21 illustrates how the objects are detected from the code.

Firstly, it sets up a connection to the webcam and adjusts its settings to make things run smoothly. Then, it brings in two models called "large.pt" and "yolov8n.pt" to help recognize things in the video. These two models were trained on different datasets. Next, it lists different types of things the models can recognize, like people, cars, and a specific board I trained the model for "Basys3." The cool part is that you can choose what to look for by adding or removing things from the list.

The code then goes into a loop, where it continuously grabs frames from the webcam. For each frame, it asks the models to figure out what's in it. If the models find something interesting, like a "Basys3" or a "person," it draws a box around it and shows the result on the screen.

Finally, it shows the live video from the webcam with these boxes around the recognized objects. If you press the 'q' key, the program stops.

```python
import math
# start webcam
cap = cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)

# model
model = YOLO(r"large.pt")
model2 = YOLO("yolo-Weights/yolov8n.pt")

# object classes
classNames = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck", "boat",
              "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", "bird", "cat",
              "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", "backpack", "umbrella",
              "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", "sports ball", "kite", "baseball bat",
              "baseball glove", "skateboard", "surfboard", "tennis racket", "bottle", "wine glass", "cup",
              "fork", "knife", "spoon", "bowl", "banana", "apple", "sandwich", "orange", "broccoli",
              "carrot", "hot dog", "pizza", "donut", "cake", "chair", "sofa", "pottedplant", "bed",
              "diningtable", "toilet", "tvmonitor", "laptop", "mouse", "remote", "keyboard", "cell phone",
              "microwave", "oven", "toaster", "sink", "refrigerator", "book", "clock", "vase", "scissors",
              "teddy bear", "hair drier", "toothbrush"
              ]

detect = ["person", "laptop","cup","keyboard","Basys3"]

# classNames.extend(names)

names2 = ["Basys3"]
# print(classNames)

while True:
    success, img = cap.read()
    results = model(img, stream=True)
    results2 = model2(img, stream=True)
    # coordinates
    for r in results2:
        boxes = r.boxes
        print_box(boxes,classNames)

    for r in results:
        boxes = r.boxes
        print_box(boxes,names2)


    cv2.imshow('Webcam', img)
    if cv2.waitKey(1) == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Fig. 21. Object detection main code

CONCLUSION

In conclusion, this project successfully delved into the world of YOLO (You Only Look Once), an advanced method for spotting things quickly in real-time. I began by understanding how it works, peeking into its inner workings. Then, I put it to the test by using a trained model on the COCO database to identify objects through images and webcam. Taking it a step further,I trained the YOLO model on a special dataset created using the Basys 3 board.I carefully looked at how well the model performed with both a small and a large set of pictures. By comparing the results from these different datasets,I gained insights into how our YOLO model handled different scenarios. In simple terms,I achieved all the things we set out to explore, from understanding YOLO to making it work with real-world datasets and even training it to be a savvy object recognizer on the Basys 3 board.

REFERENCES

[1] Redmon, Joseph and Divvala, Santosh and Girshick, Ross and Farhadi, Ali,You Only Look Once: Unified, Real-Time Object Detection.Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). June,2016.

[2] https://universe.roboflow.com/georgemasonuniversity/basys-3-board.

[3] https://github.com/ultralytics/ultralytics.

[4] https://dipankarmedh1.medium.com/real-time-object-detection-with-yolo-and-webcam-enhancing-your-computer-vision-skills-861b97c78993.

[5] https://pjreddie.com/darknet/yolo/.

[6] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, Bo Ma,A Review of Yolo Algorithm Developments,Procedia Computer Science, Volume 199,2022,Pages 1066-1073,ISSN 1877-0509,https://doi.org/10.1016/j.procs.2022.01.135.