Final Project

Project Topic: Identify and quantify multiple objects in an image

The current state of robotics education in the United States is analogous to where computer science education was 10-15 years ago. There is a large push to implement advanced robotics courses into the high school curriculum. To encourage adoption, there are state and federal funding programs created exclusively for robotics or CTE (Career Technical Education) implementations. This market is where my company, Sphero, operates. Sphero develops CTE solutions for k-12 schools, comprised of a product (physical robot, building system, or sensor), digital platform (control/programming environments), and curriculum (teacher and student lesson plans). The most recent platform launch was Sphero Blueprint, a modular mechanical building system – like Lego but tailored for the classroom (Figure 1).

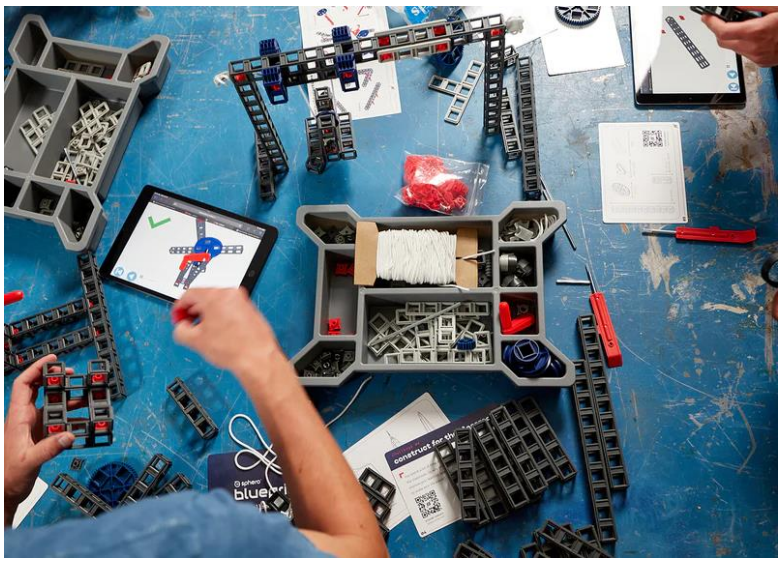

Figure 1 - Sphero Blueprint [1]

The Blueprint product line is early in its development pipeline, and it's inevitable that a robotics competition will be implemented to expand the curriculum and allow for more advanced (and more expensive) sensors to be used for additional capabilities. The Blueprint competition environment will share similarities to Vex and Lego, where students must build a robot to complete tasks on a game field (Figure 2).

Figure 2 - 2016 Vex Competition Game: Starstruck [2]

This may be stacking the tallest tower using cubes or throwing balls into a net. The game challenge, field, objects, rules, etc. are all specified by the company and are provided to schools each year. This creates an environment with very few unknowns. The game field and layout are known, the exact objects are known, and the initial positions of all game elements and robots are known. Before the user-controlled game begins, there is an autonomous period, where the robots have a few minutes to try and score as many points as possible without any inputs from the driver. Traditionally, this has been done by hard-coding motions or, more recently, using very basic sensors (such as color and distance) to score a couple of points. The simplest form of this is knowing there will be a ball 3ft in front of the starting position, so the robot is programmed to drive forward for 3 feet, grab the ball, and go score the ball. This is all done blindly, so the robot capabilities are severely limited, and they often don't score. A camera module that can identify objects, estimate their location, and create a real-time feedback loop for the robot's actions would create a competitive advantage so great that every team would need to have one to stay relevant. Additionally, this provides an avenue to integrate machine learning and computer vision into the classroom, something many educators are desperately trying to do. This is the long-term goal for my project. Many of the architectural decisions made can be traced back to this future application.

The immediate goal of this project is to develop a machine learning model that can identify and quantify the number of unique objects in an image, as well as providing the location for each of these objects. The scope of this project is limited to basic 2D shapes (square and circle) on a white background, but the images will become increasingly complex (number of shapes, size variation, overlap, etc.). This will allow the model to first be verified against a best-

case-scenario dataset, and then evaluated as the images become more representative of real-life scenarios.

Implementation

Execution Plan:

A breakdown of how this project will be implemented is shown in Figure 3, and additional details are provided in the subsequent sections. The image classifier used is a YOLO (You Only Look Once) model. Instead of starting from scratch, I fine-tuned a model that was pre-trained on Microsoft's COCO dataset. My fine-tuning dataset was created using a custom python script, which generates training/validation/testing images and labels, then stores them according to the Ultralytics YOLO dataset format. After the model has been verified to work, the images will be progressively augmented to increase prediction difficulty and more closely resemble real-world situations.

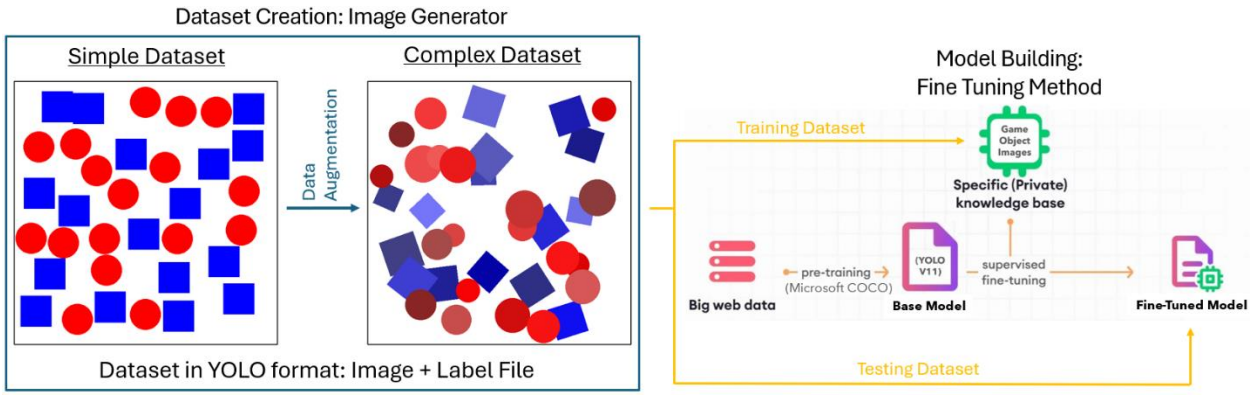

Figure 3 - Project Execution Process

YOLO Model:

YOLO (You Only Look Once) is an object detection model that identifies and classifies objects with a single pass of the image. In other words, the model only looks at the image once and from this 'single pass' can identify objects in the image. This is different from previous models (Segmentation + CNN, R-CNN, etc.) that required multiple passes to process an image..

YOLO can not only pinpoint where an object is in an image but also identify what it is. This works by passing the image through a neural network that allows the model to detect all objects simultaneously. Using grid-based prediction and bounding box prediction, the model is able to understand if objects fall within a certain cell or bounding box to identify its location.

Additionally, YOLO uses class probability to predict how to classify objects, which determines what the object is.

YOLO has a number of advantages in comparison to older methods. Since it only requires one pass, it is extremely fast and can be applied to real-time applications such as videos. Additionally, it has high accuracy (especially with natural images) and experiences fewer false positives since it analyzes an entire image at once (which provides greater contextual accuracy) [3]. It's also open source, which has led to rapid advancements since it's introduction in 2015 and allows for the model to be used commercially (as long as the source code is made public). Figure 4 highlights the improvements YOLO has made over the last 9 years, with YOLO11 being released by Ultralytics in October of 2024 [4].

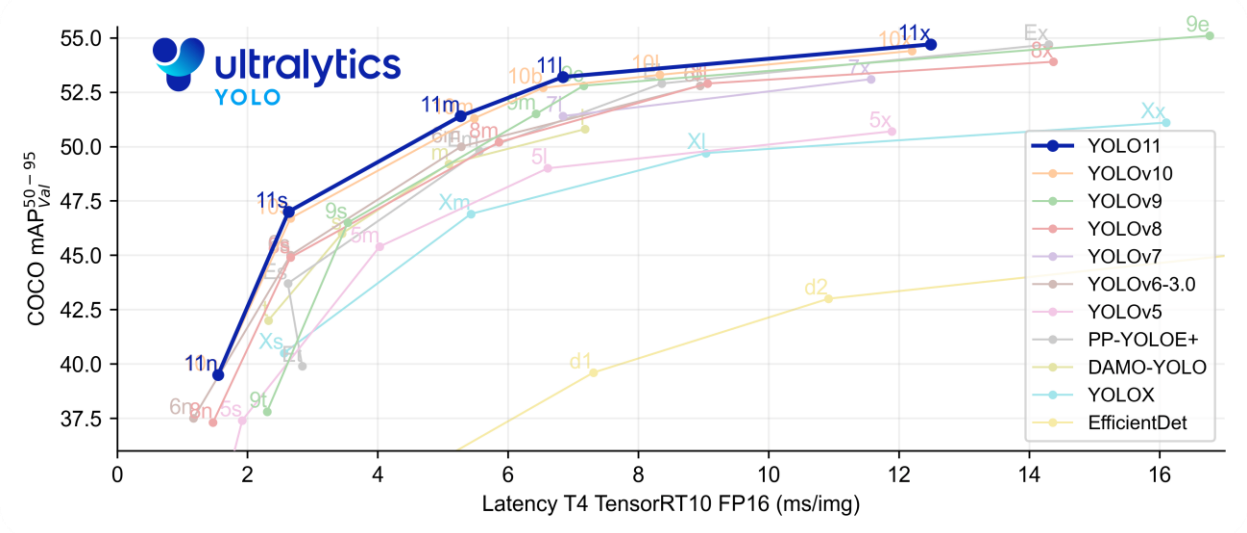

Figure 4 - YOLO's progress using the COCO dataset [5]

A large part of what makes YOLO different from other object detection models is it's use of bounding boxes. The model first splits the image up into an equal sized grid (Figure 5), then eliminates the grid spaces without any objects (Figure 6) and draws predictive bounding boxes for each object (Figure 7). When objects occupy more than a single grid space, the IOU (Intersection Over Unions) threshold is used to discard or include grid spaces (Figure 8).

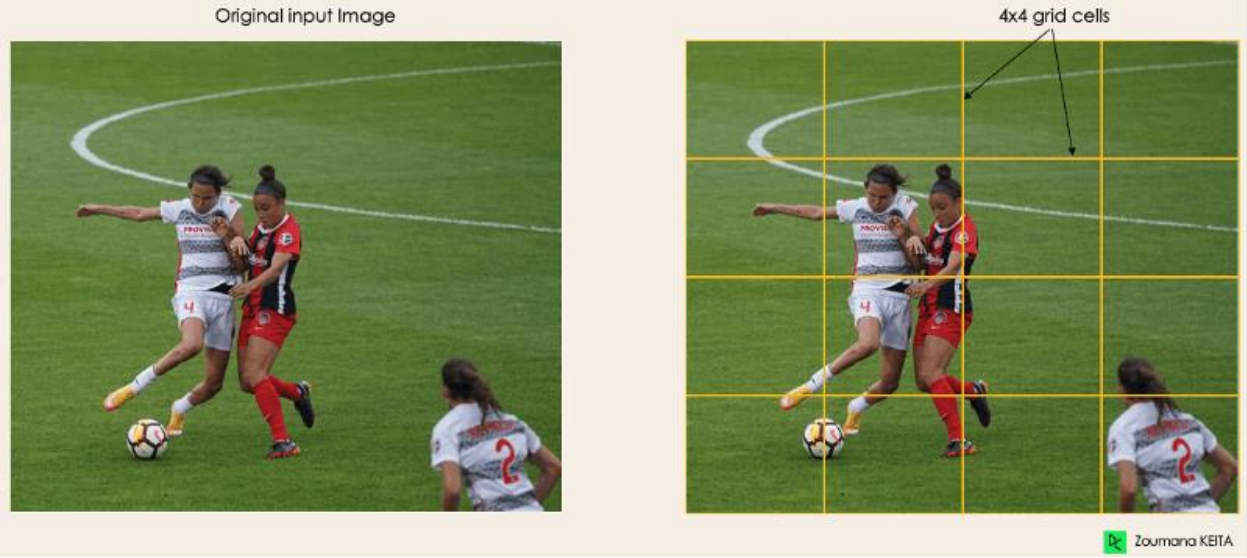

Figure 5 - YOLO Step 1: Residual Blocks (grid assignment) [4]

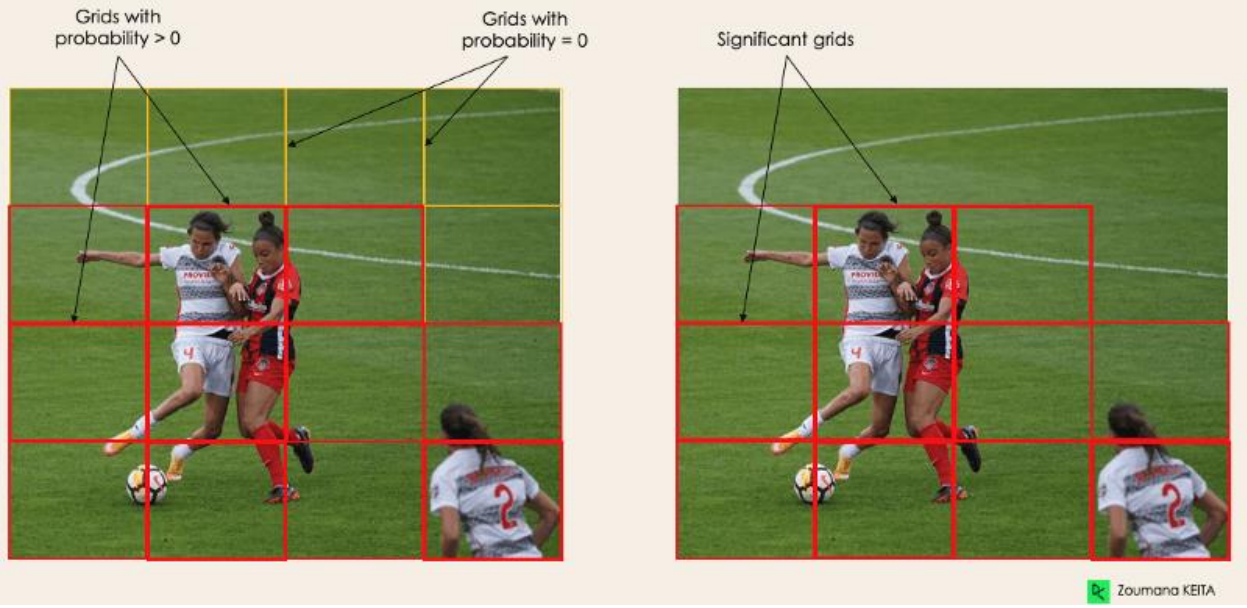

Figure 6 - YOLO Step 2: Insignificant Grid Elimination [4]

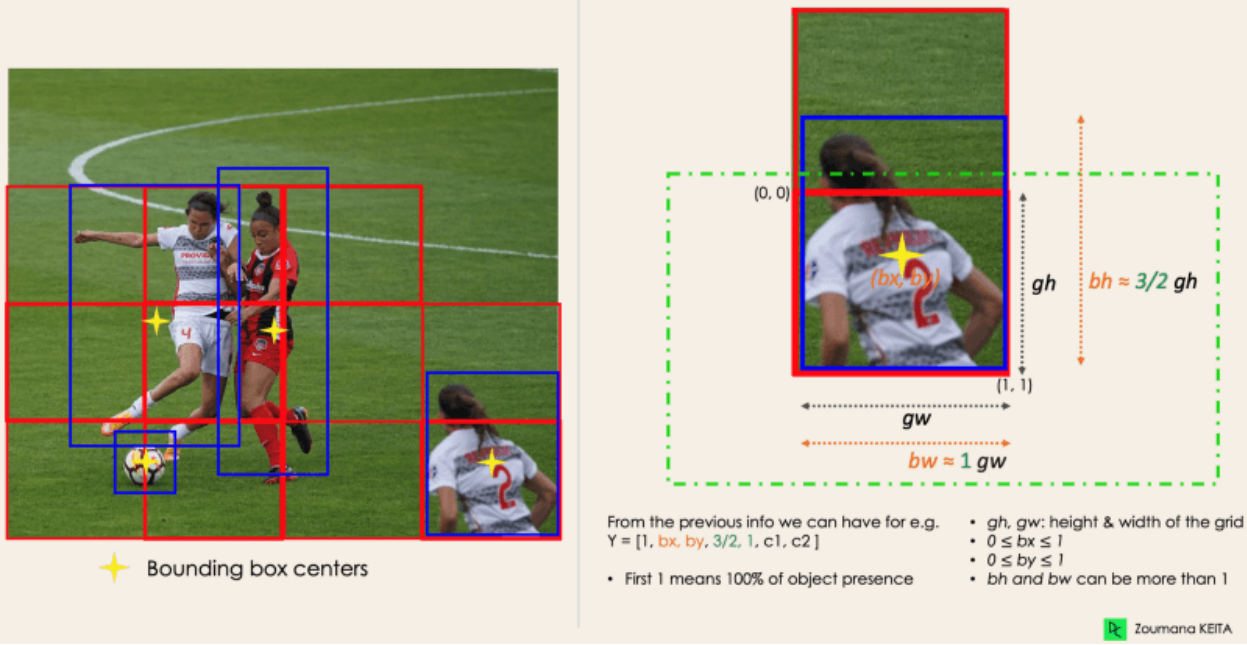

Figure 7 - YOLO Step 3: Bounding Box Regression [4]

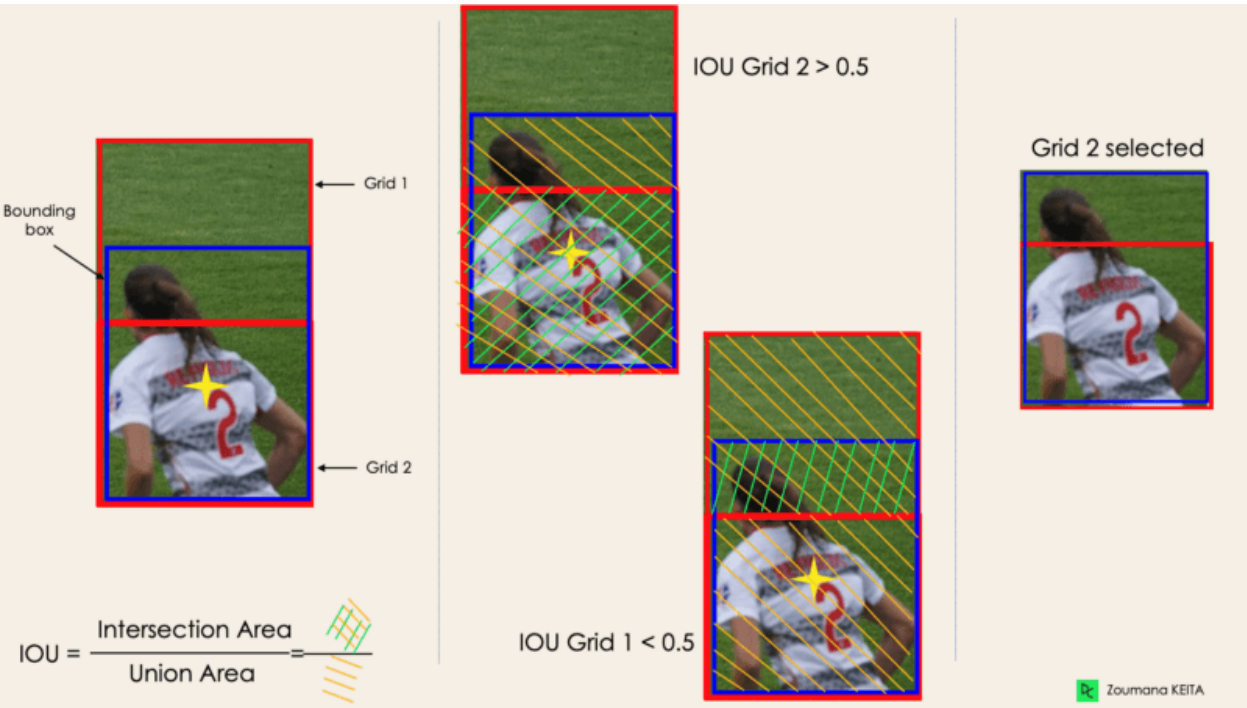

Figure 8 - YOLO Step 4: IOU Evaluation [4]

Another way to think about IOU is the percentage of the overflow grid that needs to be filled with the object for the classifier to utilize it during prediction. Figure 9 is a great example of this

concept. Unless the IOU threshold is set to 0.1%, the top grid space would be considered “does not contain a stoplight” and would be excluded from inference processing.

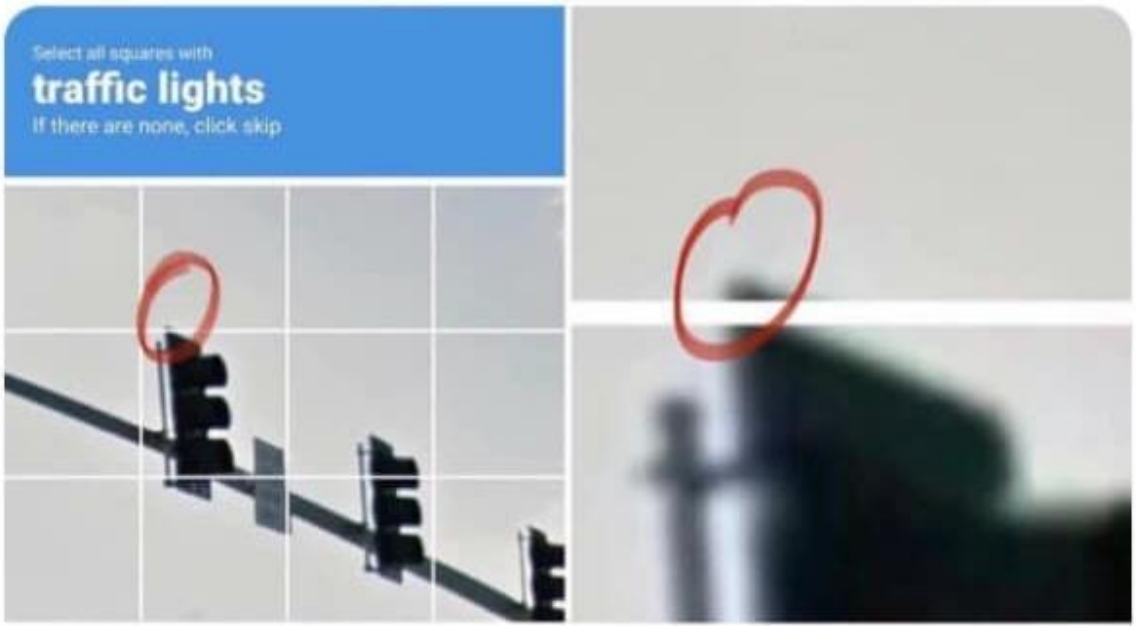

Figure 9 - IOU Example w/ Google Captcha [7]

Dataset Creation:

As with any machine learning model, the dataset is one of the most influential factors in a model’s performance. I chose to create a python program that generates images from scratch, which allowed me to control all aspects of the training data. This was necessary because manually annotating images would’ve taken too long, and there are no datasets that meet my exact requirements. I wanted to first validate my model using best-case-scenario images, where all objects are identical and clearly visible, then slowly ramp up the augmentation to reflect more realistic scenarios. As previously mentioned, the end goal is to apply this method to 3D objects using a virtual camera to take images in a game engine such as Unity. The data collection process would change for the 3D environment, but many of the fundamental principles stay the same.

The main augmentations I applied to the images are object: size, orientation, color shade, and overlap (Figure 3). The size augmentation simulates distance, where objects will appear larger or smaller depending on their distance from the camera. All shapes were randomly scaled +/- 25% of their nominal size of 75 square pixels. To simulate the physics of object orientation, the shapes were randomly rotated between 0° and 360°. To simulate the effect lighting conditions have on an object’s appearance, the color shades were manipulated in the HSV (Hue, Saturation,

Value) color space. This allowed each object to have a random saturation and value between 0.5 and 1.0, while having a fixed hue (or base color). This is very similar to how eyes and cameras perceive color, where an object's base hue stays constant, but the color looks different depending on the object's surface and environmental lighting conditions. The last, and most important, augmentation parameter was the overlap percentage. This simulates object permanence, where the camera's perspective only allows parts of an object to be visible. This parameter is set as a percentage. For example, a 60% overlap means an object may have a maximum of 60% of it's area obscured by other objects. The object locations and quantities are randomly chosen by the program, so if it places an object somewhere that breaks one of these parameters, it will delete the shape and try again. There is also an attempt limit to prevent infinite loops. If the program decides to put 30 shapes on an image, but after 1000 attempts it can't place the 28th shape without breaking one of the other parameters (such as obscuring another shape by 61%), it will re-start with different parameters. There are several checks to ensure the number of images generated match the requested quantity, and the quantities/locations of the shapes match the labels.

Since the model being used is a YOLO model, the images had to be saved as a YOLO-compatible dataset. Each image in the dataset needs an accompanying label in the form of a .txt file. This label file includes the locations and dimensions of each object (in the form of a bounding box), and the class identifier for what's enclosed in that bounding box (Figure 10).

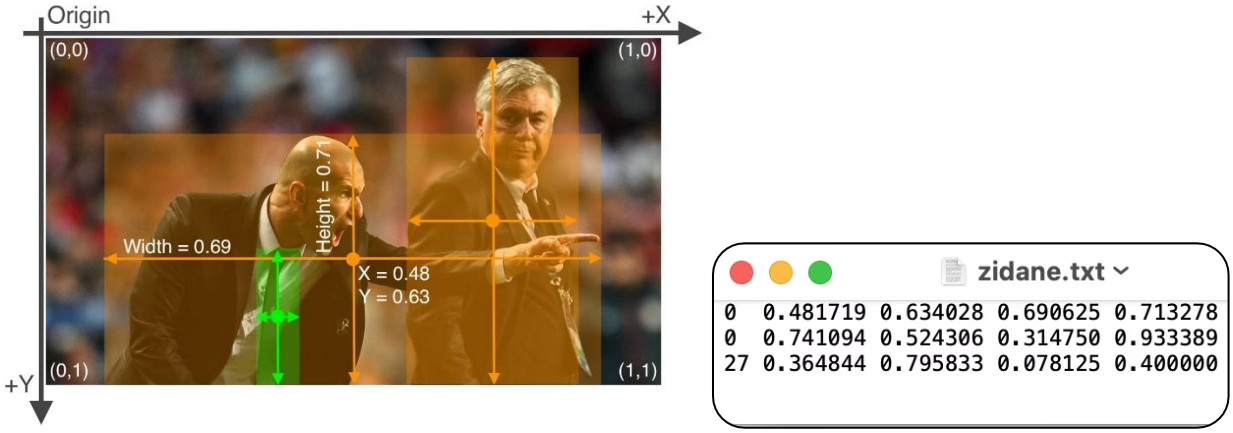

Figure 10 - YOLO Image Label (right) overlayed on corresponding image (left) [8]

All of the image file locations, label file locations, and object class definitions are described in a .yaml markup file (Figure 11).

```yaml
path: ../datasets/coco8 # dataset root directory
train: images/train # training images (relative to 'path')
val: images/val # validation images (relative to 'path')
test: # optional test images
names:
    0: person
    1: bicycle
    2: car
    # ...
```

Figure 11 - Accompanying .yaml file for YOLO dataset [8]

The image generation program automatically creates everything in the YOLO format (image files, image label files, .yaml file) including the train/validate/test data splits. For clarity, the folder structure is described below (Figure 12).

```
file_directory/iteration/
├── images/
|   ├── test/
|   ├── train/
|   └── val/
├── labels/
|   ├── test/
|   ├── train/
|   └── val/
├── Results/
|   ├── TrainingInfo/
|   └── PredictionInfo/
└── data.yaml
```

Figure 12 - Dataset File Structure

In the end, the program is able to generate ~930 images per minute, or a complete 1000 image dataset (with accompanying labels and other files) in 68 seconds.

Fine Tuning Method:

Instead of starting from scratch, I chose to use the fine-tuning method to create my object detection model. This is a form of transfer learning, which takes a pre-trained model that has been trained on a large dataset for general tasks, and then making minor adjustments to the model to better suit a specific application. In my case, I utilized the PFET (Parameter Efficient Fine-Tuning), which only exposes a few trainable parameters on downstream layers to be re-trained using my smaller and more specific custom dataset. This reduces the training computation, retains the weights and biases gained from a large dataset, and avoids over-fitting

the model since the fine-tuning dataset is significantly smaller than the original dataset the model was pre-trained on [6]. This allows me to build off the progress of others and utilize a much larger and more complex model that has been optimized for a similar dataset.

Model Architecture:

As previously mentioned, I'm using a pre-trained model as a starting point for my program. This model is "yolo11n" from Ultralytics, which is a detection model built off the YOLO v11 architecture and pre-trained on Microsoft's COCO (Common Objects in Context) dataset. This is their lightest model (Figure 13), with 319 layers and 2.6 million parameters (Figure 14). This should result in the fastest computation and prediction times, at the (potential) expense of prediction accuracy.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Figure 13 - Ultralytics YOLO v11 Detection Models

Since I'm fine-tuning the model using pre-trained weights, I left the layer structure as is, and only modified parameters exposed by Ultralytics's API. The model architecture is described in detail in the Ultralytics Yolo11n Docs, but an overview is shown below (Figure 14).

	from	n	params	module	arguments
0	-1	1	464	ultralytics.nn.modules.conv.Conv	[3, 16, 3, 2]
1	-1	1	4672	ultralytics.nn.modules.conv.Conv	[16, 32, 3, 2]
2	-1	1	6640	ultralytics.nn.modules.block.C3k2	[32, 64, 1, False, 0.25]
3	-1	1	36992	ultralytics.nn.modules.conv.Conv	[64, 64, 3, 2]
4	-1	1	26080	ultralytics.nn.modules.block.C3k2	[64, 128, 1, False, 0.25]
5	-1	1	147712	ultralytics.nn.modules.conv.Conv	[128, 128, 3, 2]
6	-1	1	87040	ultralytics.nn.modules.block.C3k2	[128, 128, 1, True]
7	-1	1	295424	ultralytics.nn.modules.conv.Conv	[128, 256, 3, 2]
8	-1	1	346112	ultralytics.nn.modules.block.C3k2	[256, 256, 1, True]
9	-1	1	164608	ultralytics.nn.modules.block.SPPF	[256, 256, 5]
10	-1	1	249728	ultralytics.nn.modules.block.C2PSA	[256, 256, 1]
11	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
12	[-1, 6]	1	0	ultralytics.nn.modules.conv.Concat	[1]
13	-1	1	111296	ultralytics.nn.modules.block.C3k2	[384, 128, 1, False]
14	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
15	[-1, 4]	1	0	ultralytics.nn.modules.conv.Concat	[1]
16	-1	1	32096	ultralytics.nn.modules.block.C3k2	[256, 64, 1, False]
17	-1	1	36992	ultralytics.nn.modules.conv.Conv	[64, 64, 3, 2]
18	[-1, 13]	1	0	ultralytics.nn.modules.conv.Concat	[1]
19	-1	1	86720	ultralytics.nn.modules.block.C3k2	[192, 128, 1, False]
20	-1	1	147712	ultralytics.nn.modules.conv.Conv	[128, 128, 3, 2]
21	[-1, 10]	1	0	ultralytics.nn.modules.conv.Concat	[1]
22	-1	1	378880	ultralytics.nn.modules.block.C3k2	[384, 256, 1, True]
23	[16, 19, 22]	1	431062	ultralytics.nn.modules.head.Detect	[2, [64, 128, 256]]

YOLO11n summary: 319 layers, 2,590,230 parameters, 2,590,214 gradients, 6.4 GFLOPs

Figure 14 - YOLO model architecture

Model Training:

The following parameters were adjusted from their default state for training: optimizer (AdamW), epochs (60), patience (5), batch size (32) and dropout rate (0.2). I also modified the validation confidence threshold to 85% (meaning any predictions with a confidence under 85% are discarded), and the IOU threshold to 70%. The full parameter list is included as a .yaml file at: <file_directory>/<iteration>/Results/Training/args.yaml. For the data segmentation, 10% of the dataset was used for testing, 70% was used for training, and 20% was used for training validation. These parameters remain constant for all prediction runs, and were chosen after testing revealed these parameters balance model performance and compute time. Increasing the fidelity of these parameters past what's shown here increased compute time and made little impact (<0.3%) on model performance parameters (best model: loss, recall, precision, mAP50, mAP50-95, Accuracy).

The images were all 640x640 pixels, which aligns with the model's pre-trained dataset (Microsoft's COCO). I wanted to try different aspect ratios and pixel sizes, but I ran up against my free Google Drive account's data storage limit. This is also why a dataset of 1000 images was used for the easiest 6 trials (basic, augmented, augmented w/ 10-40% overlap), but 2000 images were used for the subsequent 3 trials (augmented w/ 50-70% overlap). The 6th trial saw a dip in

performance (discussed in the next section), so the dataset was doubled. I would've liked to re-run all trials with the larger dataset, but I didn't have the Google Drive space for this. I also had my Google Colab account suspended for suspected bot-like activity, I presume because I was repetitively running large datasets and frequently connecting/suspending my runtime to conserve compute credits.

Results & Analysis

The purpose of this project was to see which environmental factors (dataset augmentations) affect the model's performance the most, and to make the best prediction model possible. This project acts as a feasibility study to evaluate whether this methodology would work for 3D objects from a camera, so that is where my testing focused. There were 3 categories of trials: basic (no augmentations or overlap, AKA best-case-scenario), augmented (all augmentations applied except for shape overlaps), and overlap (all augmentations applied + a percentage of the shape area was allowed to be obscured). There were 7 different overlap trials tested, ranging from 10% to 70% allowed overlap (in 10% increments). Examples of what these images looked like are shown in Figure 15.

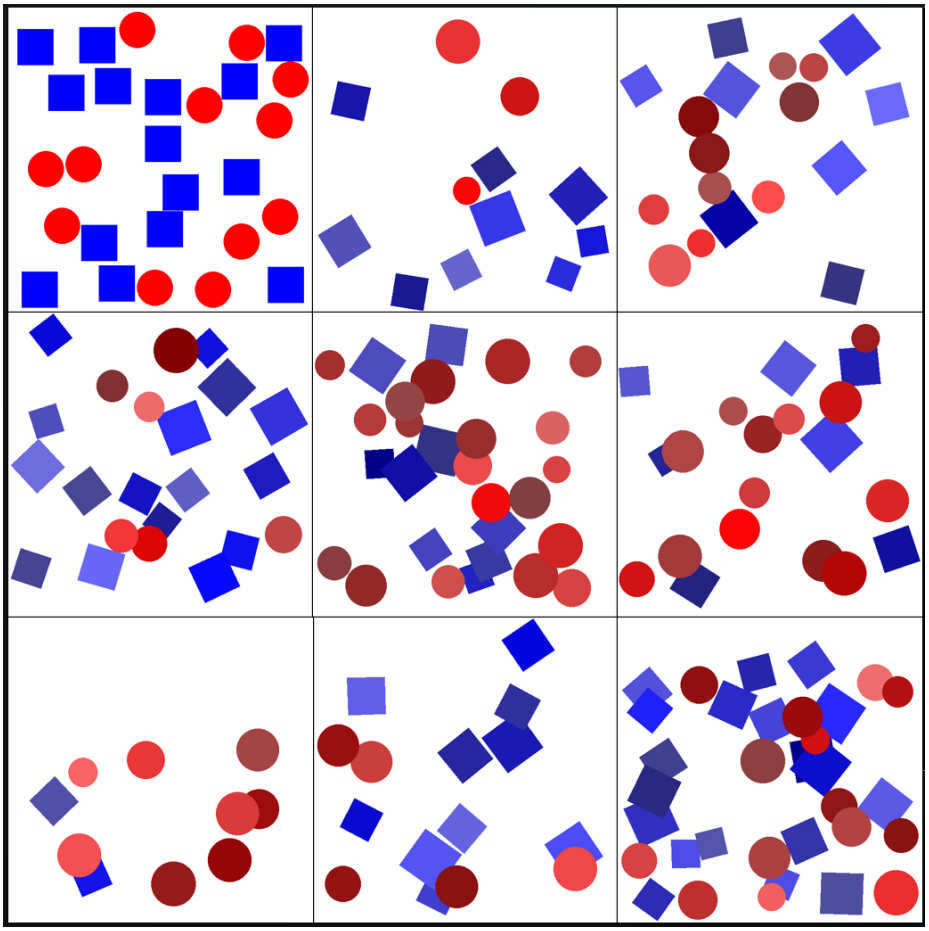

Figure 15 – Trial sample images, from left to right (top row): basic, augmented, 10% overlap. (Middle row): 20% overlap, 30% overlap, 40% overlap. (Bottom row): 50% overlap, 60% overlap, 70% overlap.

The prediction results using the test dataset corresponding to these 9 trials are shown below. It should be noted that the number of predictions each model had to make depended on the dataset, and each dataset was slightly different. Due to the different parameters (augmentations, overlap percentages), each dataset had to be randomly generated for each trial. The total number of shapes in each dataset varied (even though the number of images stayed constant). Additionally, the last 3 trials were done with a dataset twice the size of the others (for reasons explained previously). That being said, the number of total predictions for each dataset size is within 3% of one another (Figure 16).

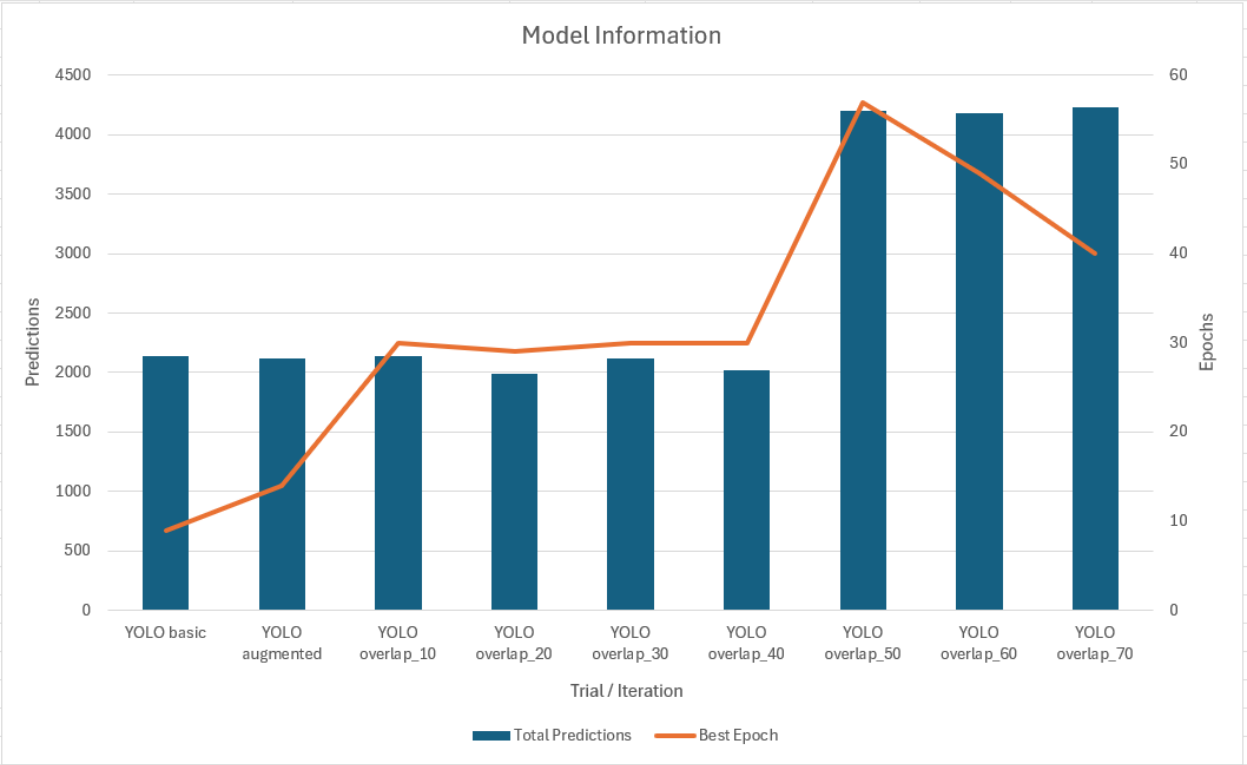

Figure 16 - Total Model Predictions (test data) & Epochs until best model

The prediction results show a correlation between trial type, dataset size, and number of epochs utilized. All models triggered the earlystop condition, which eliminates the number of epochs as a limiting factor. Once the models were trained, the prediction times between them were identical. The average prediction time for an image was 11ms regardless of the trial, augmentation, or number of objects in the image. This is very good news, highlighting one of the greatest benefits of YOLO. Since all predictions are made in a single pass, the prediction time per frame is the same, regardless of how many objects are in the image. Looking forward, a 30FPS camera records video at 0.33sec/frame. With the current conditions, the YOLO model finishes its prediction in one third of the time (0.11sec), meaning this model is more than capable of real-time prediction from a live video stream. This prediction was running on an Nvidia L4 GPU, so the prediction time on a lightweight NPU that could be attached to a camera is unknown.

Quick prediction doesn't come at the expense of accuracy or precision either. Figure 17 showcases the total accuracy and mAP50-95 for all trials using the test dataset (values out of 100%).

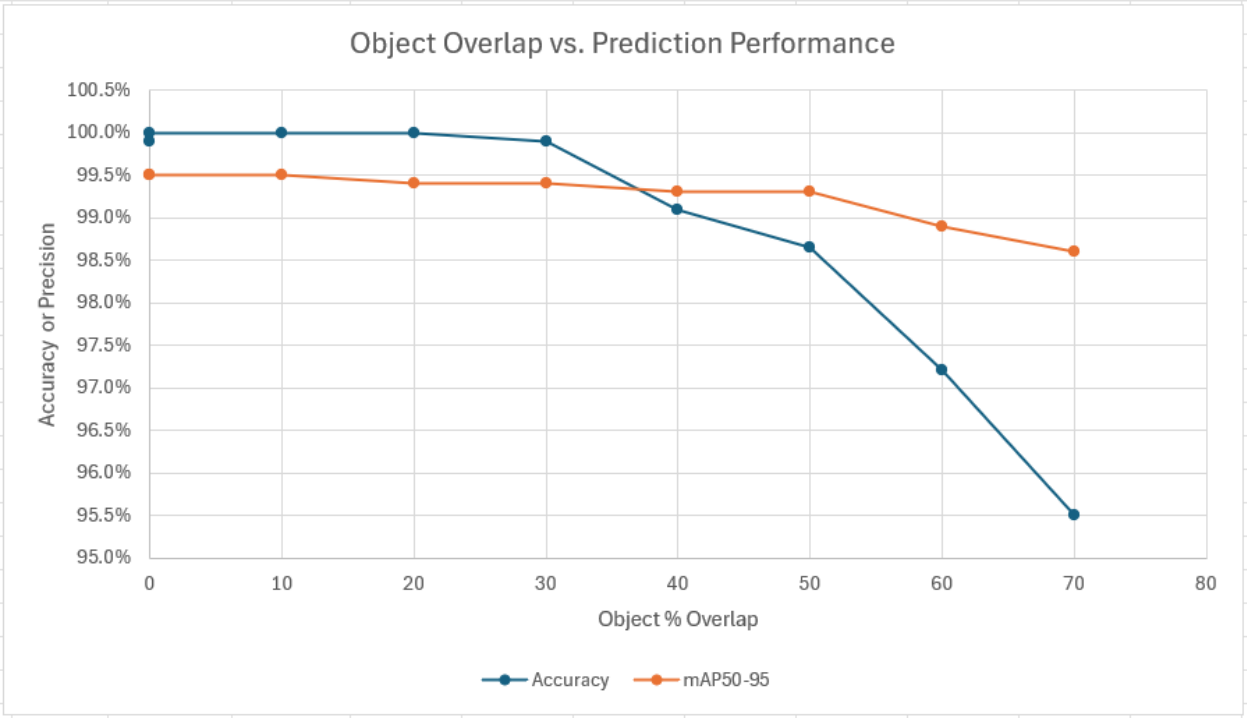

Figure 17 - Accuracy and Precision for each trial, organized by allowed object overlap

The correlation between overlap percentage and performance is very clear. After 50% overlap, the prediction performance starts to degrade, however, this is not the full story. As shown in Figure 18, nearly all prediction errors are from missed detections.

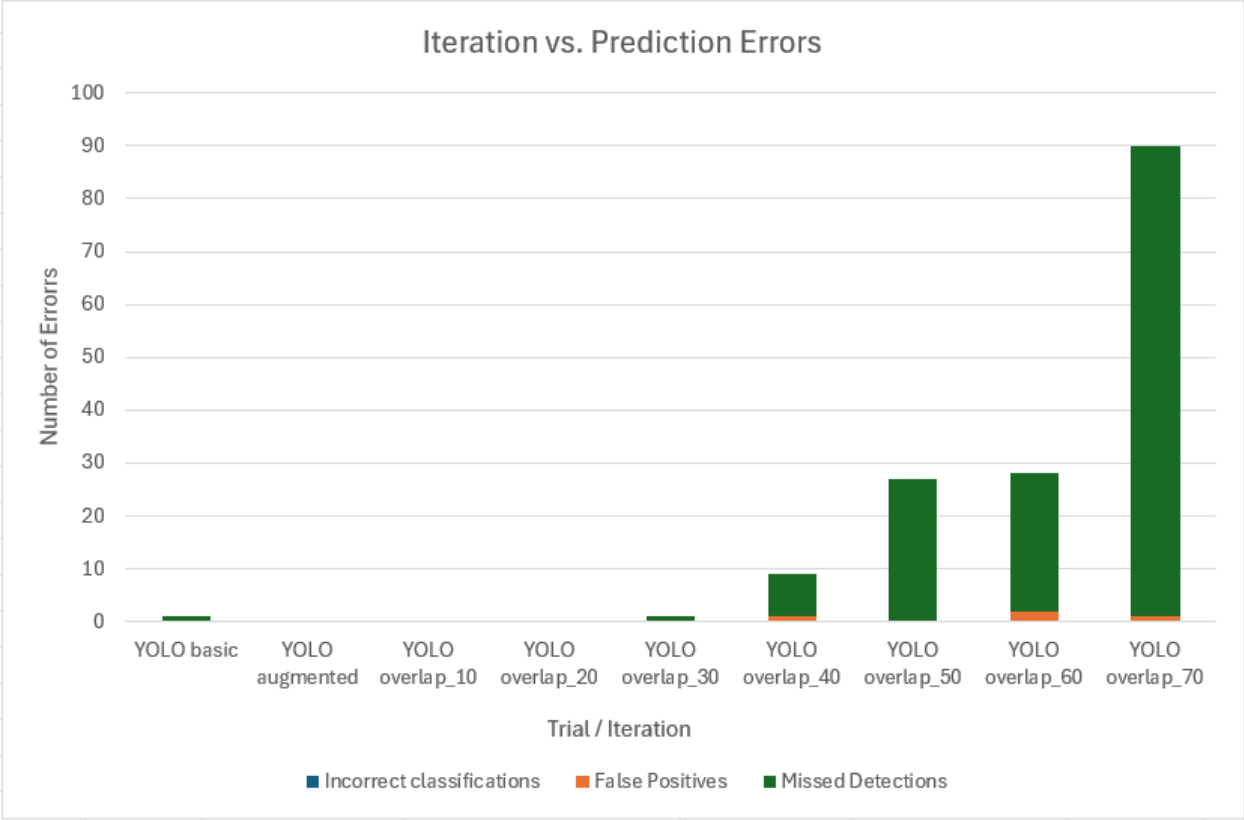

Figure 18 - Prediction Errors breakdown for each trial

After looking through the examples, the instances of missed detections are when small objects are covered, or two objects of the same color overlap one another (Figure 19).

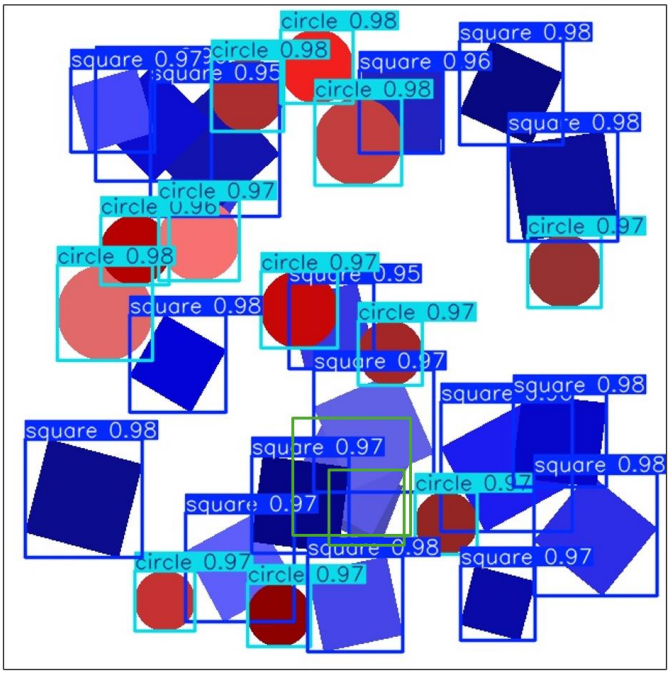

Figure 19 - Missed detection example (green squares) - Trial: 50% overlap

Even for a human, these are very difficult to identify. Given the 98.7% accuracy (for the 50% overlap trial), this is a very cherry-picked example. This exceptional accuracy is contextualized in the corresponding confusion matrix (Figure 20).

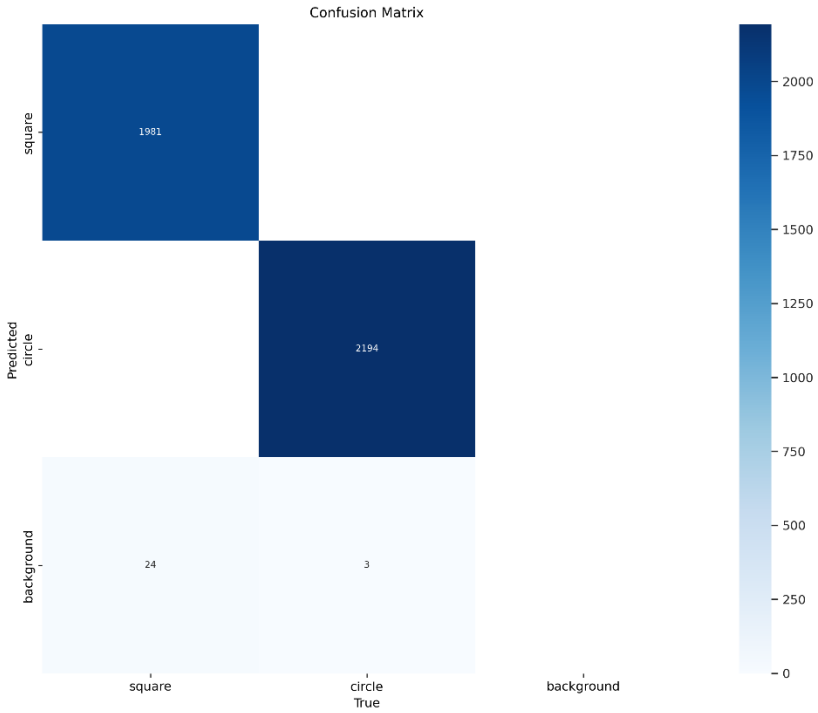

Figure 20 - Confusion Matrix for Trial: 50% overlap

A more typical prediction result is shown below (Figure 21). This image has a 100% detection accuracy, which is impressive given some of the squares are mostly covered. Even when much of the object is obscured (such as the dark square in the bottom right corner with a 94% confidence rating), the bounding boxes are still very accurate. This is another standout feature of the YOLO model.

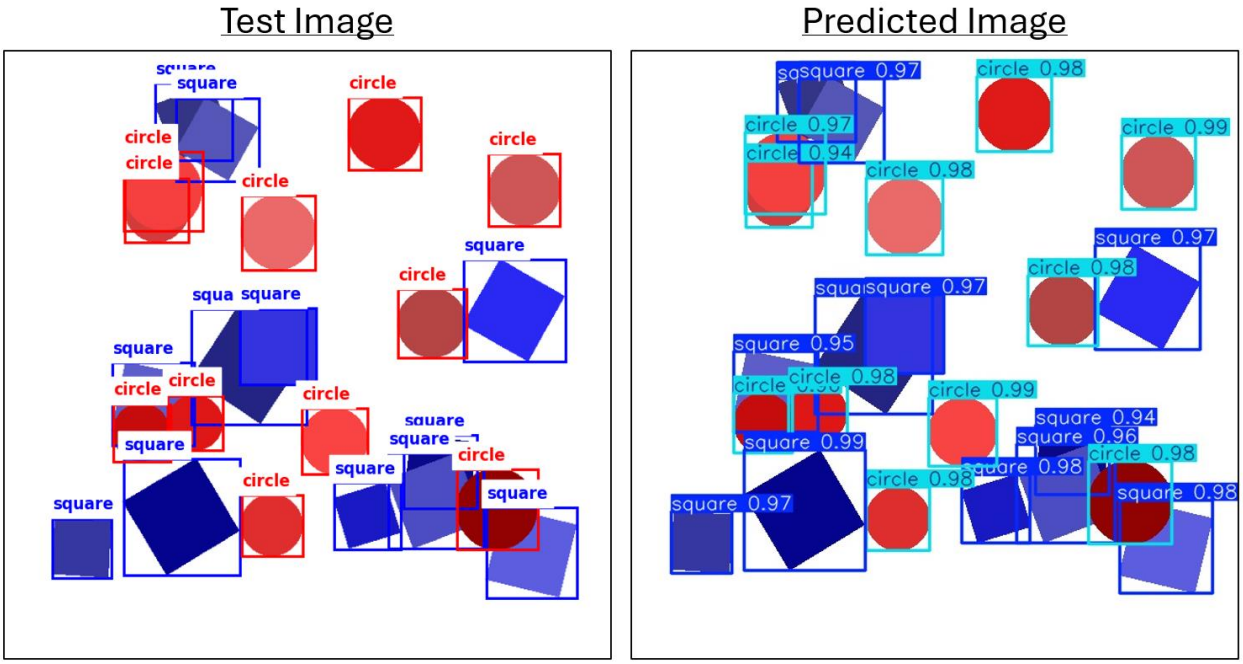

Figure 21 – Example image, Actual vs. Predicted results for Trial: 50% overlap

As shown in Figure 17, the accuracy of 60% and 70% overlap trials is still over 95.5%, which is impressive. The number of prediction errors and shape are positively correlated. Since these errors are missed detections, it's likely that a lower IOU value could improve this issue. For my application, this does not cause any problem. In a real environment, the camera will move around and expose more of the hidden objects with time. Additionally, the closest objects to the camera are the most important. This is why many of the analysis shown was done for the 50% trial, although the same data exists for the 60% and 70% overlap trials in the dataset directory.

Reflections

These are very promising results, and I'm very pleased with the successful completion of this project. Not only does the model quickly count how many of each object is in the image, but it also reports the location and bounding box for each object. If my Google Colab account wasn't

suspended, I'd experiment with changing the IOU value to see how this impacts performance. For future analysis, I'd like to see how the model performs if the images are converted to greyscale. Full color cameras require significantly more processing than greyscale cameras and are more expensive. This could improve performance for when it's run on a lightweight NPU instead of an Nvidia L4 GPU.

The majority of the code I have written is for creating the training dataset, and then parsing/extracting the prediction results. The Ultralytics API made the YOLO model implementation very straightforward, similar to Keras's API that I used for previous assignments.

Program Usage Instructions

The YOLO model was implemented in Python 3 using Google Colab as the programming environment. All necessary libraries are listed in the first notebook section. The library versions should be whichever version is the most recent as of 2024/12/03 (Colab automatically pulls the latest library by default).

1. Open the FinalProject_KR.ipynd file in Google Colab (or compatible python notebook editor).
2. In the 'Main Program' section:
    a. Set "createDataset" to "True" or "False" depending on whether or not a new dataset should be created.
        i. If "createDataset=True":
            1. Define the 'drive.mount' location so that Colab can connect to your google drive. (or local filepath location)
            2. Set the 'file_directory' to the directory path where the dataset should be created, and set the 'iteration' to whichever folder the dataset should be generated in. If this folder already exists, all contents will be deleted.
            3. Set the "num_images_to_generate" to the total number of images to generate for the dataset.
        ii. If "createDataset=False":

1. Ensure the ‘file_directory’ and ‘iteration’ are set up to point to an existing dataset with the correct YOLO structure.

b. Set “runModel” to “True” or “False” depending on whether or not to train and predict with the model using the specified dataset.
    i. If “runModel=True”:
        1. In the ‘Model > runModel’ section, ensure “!pip install ultralytics” is uncommented for the 1st run. Then, this line of code should be commented for future runs on the same session (trying to install a library that’s already installed will throw an error).

3. Run: all sections (ctrl+F9 in Colab) – best if connected to an Nvidia L4 GPU
4. The program will output performance graphs once the model is trained, and then it will evaluate the model using the test data. These graphs and prediction images are located in “/iteration/Results”.

Report References

- ChatGPT4.o and Google Gemini were used to edit this report with human-guided prompts and editing.

1. https://docs.ultralytics.com/datasets/detect/#ultralytics-yolo-format
2. *Blueprint Build.* (2024). Sphero.com. https://sphero.com/collections/blueprint/products/sphero-blueprint-build-kit
3. *2016-2017 Starstruck*. (2016). NYIT VEX ROBOTICS TEAM. https://nyitowieee.weebly.com/2016-2017-starstruck.html
4. *YOLO models for Object Detection Explained [Yolov8 Updated].* (n.d.). Encord.com. https://encord.com/blog/yolo-object-detection-guide/
5. Kelta, Z. (2022, September). *YOLO Object Detection Explained: A Beginner's Guide.* Www.datacamp.com. https://www.datacamp.com/blog/yolo-object-detection-explained
6. Ultralytics. (n.d.). *Models*. Docs.ultralytics.com. https://docs.ultralytics.com/models/
7. Bergmann, D. (2024, March 15). *What is Fine-Tuning? | IBM.* Www.ibm.com. https://www.ibm.com/topics/fine-tuning
8. (2024). Themeim.com. https://themeim.com/wp-content/uploads/2022/09/overthinking-captcha-meme-1.jpg
9. Ultralytics. (n.d.). *Object Detection Datasets Overview*. Docs.ultralytics.com. https://docs.ultralytics.com/datasets/detect/#ultralytics-yolo-format

Code References

- ChatGPT4.o and Google Gemini were used to debug error messages, explain API calls, and provide example solutions to coding problems with human-guided prompts and editing.

10. Jocher, G. (2020, May 18). *YOLOv8 Documentation*. Docs.ultralytics.com. https://docs.ultralytics.com/