ScottySeat

Aditi Raghavan, Mehar Goli, Chen Shen

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— Students waste time searching for study spaces, and the goal of ScottySeat is to streamline this process. Our solution is to capture images of study spaces and update a web application in real-time to reflect the seat availability in a given room. To accomplish this, we will be using a custom-trained YOLOv5 model to detect people and chairs.**

***Index Terms*—Computer vision, object detection, single board computer**

I. Introduction

Campus study spaces are a crucial commodity for many students at CMU. Whether it be for individual or group work - campus provides a reliable, safe location for students to work at. However, CMU has limited study space and free spots are hard to come by.

Looking at campus itself, popular areas tend to fill up quickly while smaller spots aren't widely known. With the size and complexity of CMU's campus, finding a single free space can take upwards of 30-40 minutes and even longer for group scenarios. Currently, classrooms/meeting rooms can be reserved but this doesn't account for open study areas that make up the majority of student study areas. Room reservations are usually restricted to organizations or students of the school where the room is located. Reservations also fill up quickly and don't allow for time flexibility to book closer to study time.

Our project seeks to ease the struggle in and reduce the time taken in finding campus study spots by providing students with easily accessible, real-time information on study spot availability. To do this, we will build a web platform showing digital maps of study areas. Each study area will have its own map indicating study spot locations and availability. Real-time camera footage of the areas will be monitored using computer vision to regularly update the site as well.

II. Use-Case Requirements

To meet our use case, we have created requirements for numerical accuracy, speed and spatial accuracy. Numerical accuracy in our use-case relates to the number of seats available in a study space. This is arguably the most important use-case requirement as low numerical accuracy can misguide users meaning that there would be no speed up in finding a study spot. Keeping this in mind, we aimed to have 90% accuracy in detecting available seats. Our second use-case requirement relates to the update speed from capturing an image to the results being displayed on our webpage. For our solution to be useful, the data provided to the user should reflect the study rooms current capacity. After polling some potential end users and taking their feedback, we have decided that changes in seat availability must be reflected within 45 seconds of them occurring. Our last requirement is regarding the user interface. For users to be able to use our UI with ease, the seat mapping needs to be easy to interpret. This means that the seat mapping should closely reflect the positions of the real positions. Initially, this spatial accuracy use-case requirement was quantified as being able to map chair positions to seating charts with a 20% error margin. This metric will be measured by comparing the seat mapping with a camera with an overhead view, is needed.

III. Architecture and/or Principle of Operation

Our system can be mainly divided into 3 subsystems, including the hardware module, the computer vision module, and the web application (UI) module, as shown in the diagram below (Figure 1).

The sensing module consists mainly of TedGem USB cameras. Such cameras will be deployed one for each study space being monitored. Its main purpose is to provide input for the CV algorithm for detection, and also provide training and testing data that would be used for verification.

Within a room, we have a wall-mounted camera, angle down at the room floor angle camera, we will sample a frame every few seconds, preprocess it, run it through a custom trained YOLOv5 object detection mode. We will take 15 samples per update, and take the highest confidence samples, post process them and update the web application. Further details have been provided in the graph, or explained later.

IV. Design Requirements

To achieve our first use-case requirement, we will need a single high quality camera per room and must have a highly accurate object detection algorithm.

For the camera footage, we require 1080p resolution to be able to accurately detect objects and their edges. The camera needs to have a large field of view to be and be placed correctly so that it captures the entirety of the room in a single shot. Our set-up must work in a variety of lighting conditions so the camera should have automated exposure adjustment.

To achieve our first use-case requirement, the object detection algorithm must be able to identify chairs and people

sitting on chairs with 90% accuracy. To aid with achieving this accuracy, we sample at least 15 times per 45 second update (overall 20 times per minute) and use the detected objects with the highest confidence. If the highest confidence detected in an image is less than 80% we will ignore the images taken in that minute. The algorithm will need to be able to also discern clearly between nearby objects with high accuracy and will also need to have very high accuracy on occluded objects.

To achieve our second-use case requirement, speed, we will need a relatively fast algorithm which does not compromise accuracy for speed. The website should automatically refresh every 45 seconds and we are giving 1 second for final changes (storing information to the database) to be reflected on the page. Given that we have a 44 seconds window between a change occurring and an update, and aim to have 20 samples/min, we will have ~14 samples between the change and the update. This means the algorithm needs to be able to run and complete within 3 seconds.

To achieve our spatial accuracy requirement, we will need to carry out perception correction as a final post processing step before displaying the maps to users. Specifically, the positions of the detected objects will be adjusted from the angled view of the camera to a top-down view.

V. Design Trade Studies

A. Hardware

The motivation for using hardware, instead of software for running our computer vision model was due to privacy. Our project captures images in a public space, and to help ensure privacy of the users, we thought it was best to use a single board computer. This way images will not have to be sent over the network, reducing the possibility of hacking, and the images can be easily deleted after use. The only data being sent over the network will be the locations of chairs and tables and so users can safely participate in our project while remaining anonymous.

The choice of using a single board computer (SBC) compared to an FPGA was that it would be much easier to scale an SBC model versus an FPGA model. Additionally, networking on an FPGA is non-trivial and translating the computer vision models would be out of the scope for the given project timeline.

The main difference between the NVIDIA Jetson Nano 2GB Developer Kit and the Raspberry Pi is their compute power. Since we intended on using machine learning to identify objects, a larger GPU is preferred as it is able to parallelize many of the matrix operations needed in machine learning. While analyzing YOLOv5, researchers found that the RPi, out of the box had an FPS of 1.6 FPS while the Jetson Nano had an FPS of 5 FPS [1]. Considering that we are using 2 cameras for our MVP (one camera per room), and have some preprocessing and postprocessing to do on our input image, using the RPi would possibly become a limiting factor in our speed requirement. A solution to this might be to use a smaller YOLO model but scaling down the algorithm results in a reduced accuracy according to benchmarks released by Ultralytics [2]. Another possibility was to use the RPi along with the Intel Neural Compute Stick 2. According to Feng et al., the RPi + NCS2 outperforms the Jetson Nano on both mean confidence and FPS, when using YOLOv3 [3]. However, the Intel NCS2 is out of stock, and for our project timeline, we needed our hardware as soon as possible.

B. Computer Vision Model

- The motivation behind computer vision as opposed to a physical method such as seat sensors was scalability. Scalability is simpler with computer vision, by adding a few new cameras to a space rather than individual sensors to every chair in the area.
- The main goal of our computer vision model is to detect chairs, tables and people in a given room from a given image. Chairs will be oriented in different positions and will likely be partially or mostly occluded by a table or a person. Taking up to 15 images per 45 second update period, images will be sampled every three seconds and the model will also run on hardware. As such some requirements we had for a given CV model were as follows:
  - Classify and locate multiple objects in an image
  - Scalable to more than one object class
  - Can run in under 2-3 seconds
  - Can handle object occlusion and orientation variability
  - Optimally, have the highest starting accuracy with lowest resource usage.
- Overall, we considered a variety of object detection architectures: namely neural network-based methods (Faster R-CNN, Yolo) and feature detection-based methods (BRIEF, SIFT, ORB). All the architectures noted can be implemented in Python through the use of common libraries and thus can run on the Jetson Nano. Specifically, the underlying libraries used are OpenCV (BRIEF/SIFT/ORB feature descriptors are built in), and Tensorflow/Pytorch for neural network models. They all also run relatively quickly - in close to real time. .

From there, the feature detection methods tend to have the lowest resource usage but also come with a number of risk areas due to how object detection is ultimately done. With feature detectors such as BRIEF/SIFT/ORB - object detection is generally performed through image matching [5]. The feature descriptors detect various ‘features’ in a source image (i.e. corner points, areas of high contrast and variability) and a thresholding model such as KNN is used to match those points to an object in a test image. Thus, it operates essentially as a one to one image matching architecture. This makes it difficult for these methods to account for object occlusion if not enough points are present. In our use case especially, chairs can’t be moved by a user to help object detection so the system needs to account for heavy object occlusion. The system also becomes difficult to scale to multiple object classes. Different sets of descriptors and models for each object class, that is for different types of tables, different chair designs will be needed. Lastly, image matching for object detection also makes this method unideal for person detection as well, a separate system will be needed to track people in the scene.

In contrast, the neural networks were pre-trained by the developers to work on multiple object classes with multiple objects per image. For example, YOLO in particular was pretrained on the COCO dataset to detect 80 different object

Commented [1]: Prof: Why 90%, (basically there's a why question for every sample/accuracy metric in this section).
Also, 'what is hance level, split into false/true positives'

Self Note: We need to explain the relation btwn 20 times vs 10-15 samples

Commented [2]: Add notes on security

Commented [3R2]: @akraghav@andrew.cmu.edu

classes. In testing we can see, out-of-the-box YOLO is able to detect tables, chairs, people and even some objects such as the TV and a backpack as well. We can also note that YOLO is able to detect the chairs despite heavy occlusion.

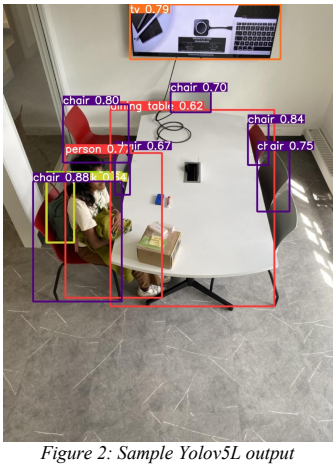


Figure 2: Sample Yolov5L output

It is possible to build a feature descriptor-based architecture to work for our use case, however with the issues that will need to be accounted for and the time that will be taken for training of either system type - a neural network based model is preferable for our case.

Of Faster-RCNN, YOLO - Faster R-CNN is noted to have the highest precision and slowest time. The creators of YOLO themselves found that a Faster R-CNN with VGG-16 architecture achieved a mAP of 73.2% with a speed of 7 frames per second as opposed to standard YOLO achieving 63.4 mAP and 45 FPS (both are when tested on COCO2007 - a CV dataset) [6]. Here mAP refers to Mean Average Precision, here it tells us the system with the highest true positive to false positive ratio. To see whether Faster R-CNN or YOLO will prove more accurate for our use case. We tested Facebook's Detectron 2 Faster R-CNN architecture and Ultralytic's YOLOv5L architecture locally using 6 iPhone test images of our study space. Each image showed a single human, one table and 7 chairs. Testing showed, YOLO was able to accurately detect more objects than Faster R-CNN even with lower resource use, though Faster R-CNN did generally seem to have higher confidence in its results through a qualitative analysis. Through this we determined we will finally use YOLOv5L for our project.

i. Table 1: Detection Testing off Faster RCNN vs Yolov5L

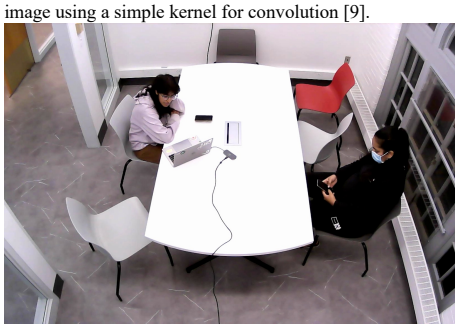
Architecture	Human Detected?	Table Detected?	Chairs Detected	Objects Detected
Faster RCNN + ResNet 50	6/6	6/6	18/42	30/54
Faster RCNN + ResNext 101				
Yolov5L	6/6	6/6	39/42	51/54

VI. System Implementation

A. Subsystem A - Computer Vision

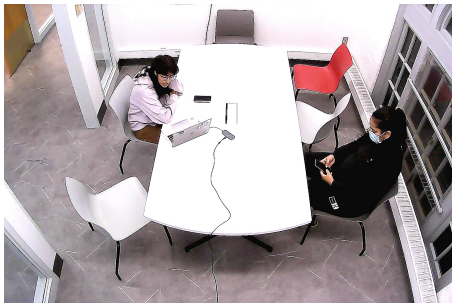
The computer vision pipeline comprises three sections: preprocessing, CV model and postprocessing (sample consolidation), denoted with bold text for the start of each section. Overall, the system will take in an image sampled from the camera feed and output list of object detected and confidence level for each object, and coordinates for a bounding box around each box.

Preprocessing: Initially, we were converting our image to grayscale and doing contrast limited adaptive histogram equalization (CLAHE). CLAHE was used as it equalizes the contrast within the image [6]. As you can see in Figure 4, the foreground and background are both dark with low contrast so applying equalization seemed like a good way to combat this [7] . That being said, we found that it was leading to lower recall on our custom CV model. After analyzing some of the missing objects, we suspected that the objects were being missed as they had low contrast compared to nearby objects, and since we were converting to grayscale discriminating features like chair color was being lost. To combat this, we decided to stay in RGB and increase brightness and contrast using convertScaleAbs [8] CV function. To increase distinguishability, we sharpened the image using a simple kernel for convolution [9].



■ *Figure 3: Initial webcam test image*

Commented [4]: Fro trade studies: Section C on webapp?



■ *Figure 4: Preprocessed webcam image*

CV Model: The sampled image is then run through a custom-trained YOLOv5l model to retrieve bounding boxes for objects detected, the classes of detections and the confidence level of the classifications. Figure 5 shows a visual representation of the data that will be outputted - except through the form of a list.

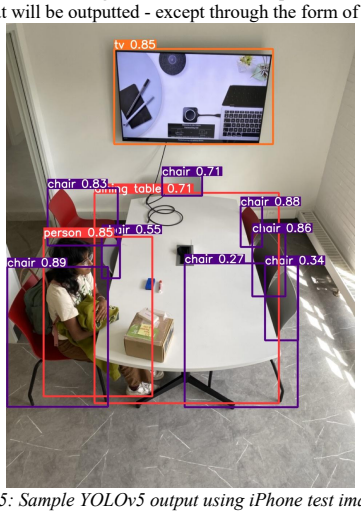


Figure 5: Sample YOLOv5 output using iPhone test image

Out-of-the-box, our base Yolov5l model was pre-trained by Ultralytics suign the 80-class COCO image dataset. We noted that we only required 4 out of the 80 classes for our use case: tables, chairs, people, backpacks and that the starting model had many false positives and low confidence detections in these classes. For these two reasons, we initially decided to replace the output mapping with just our four target classes and to use transfer learning to account for both these issues. For transfer learning, we wanted to train first on another existing dataset such as ImageNet to decrease occurrences of false positives and then a custom dataset to increase classification confidence.

To prepare the custom image dataset, we took pictures of the study room with people and chairs in various configurations and orientations, to capture as many real life scenarios as possible. To take these pictures, we used an angled camera to simulate the actual camera setup the model will have to operate with. All the collected data was labeled using Roboflow. Data augmentation was then used to artificially diversify and increase the dataset size by 7x.

For the existing dataset, we decided against ImageNet after finding it had many person-adjacent classes (ie hand, man, woman etc) rather than one 'person' class. Instead, we used Pascal VOC and the OpenImages dataset - both common for object detection with chairs. Both datasets were stripped of any images without the target 4 classes and were relabelled with the custom 4-class mapping as well.

While training, we found that training on existing datasets had limited impact on final training results. The types of background and settings shown in the existing datasets were too general to specify to our use case. As the model was already pretrained, what was needed was the specificity given by the custom dataset. We found that final training accuracy with the custom dataset remained the same with and without initial existing dataset training and that images trained with existing dataset had worse detection accuracy.

One other change we made was to switch from our custom 4-class mapping to the 80-class mapping. Initially, we found that the 4-class mapping was hindered by the backpack class for cases where a chair was completely covered by a backpack. Switching then to a 3-class mapping, while the model was able to reach high precision and recall metrics with the mapping, live testing showed many cases of high confidence, false positives that sampling could not account for. As the target classes were already part of the pre-trained 80-class mapping, we decided to switch back to this mapping to take advantage of YOLOv5L's existing accuracy. Once again training on the custom dataset with this mapping scheme, the occurrence of high confidence, false positives was greatly reduced while retaining high precision and recall metrics.

Full details on training accuracy metrics and outcomes are discussed in object accuracy testing.

Post-Processing: From here the post-processing of the CV pipeline is centered around consolidating the samples taken since the last update. The inference output of each image will be temporarily stored until 15 samples have been taken. Looking at our sample output we found that we were prone to undercounting chairs, specifically when people were sitting on them. The motivation behind this secondary step is that while we do want high confidence detections, objects are sometimes covered between frames - so we want to account for cases of non-detection due to excess occlusion as well. That being said, minute movements, within the span of 10 seconds, would often lead to a desirable result, a correctly sized bounding box above our confidence threshold. Keeping this in mind, we created a sampling algorithm that gave images with greater number of chairs higher priority. Our sampling algorithm also took into account the mean confidence of objects of interest.

Our algorithm removes all the objects in the image with a confidence of less than 0.5. It then filters out extraneous objects like backpacks and phones, and calculates the mean confidence of these images.

B. Subsystem B - Related Hardware Components

In our ideal scalable system, we have 1 Jetson Nano for every 2 rooms. The Jetson Nano takes in 1 video feed per room and runs an instance of the CV module per room.

A 1080p video camera with a large field of view is used to capture images used for inference. We chose a video camera with autofocus and auto brightness to make our preprocessing steps easier. The camera is placed on one wall of the room and angled down toward the table.

Inference is done using TensorRT to mitigate issues regarding memory usage and bandwidth which arises when using PyTorch for inference. We observed that loading large models on PyTorch on the Jetson maxed out the physical memory available. This also affected the time it took to load, which was approximately 20 seconds.

Additionally, one of these Jetsons will host our web server. In our MVP, we successfully had the Jetson running inference on 2 rooms while simultaneously running the web server. To test that our HTTP requests were working we hosted our web server on a different IP address and it worked as expected.

C. Subsystem C - User Interface

In order for the output of the CV algorithm to be displayed in a user-friendly way, a web application interface has been created. It consists of 2 parts: data post-processing and web application. Different from our last report, the data post-processing portion has been moved out of the Web server, and been put in the Jetson nano. This lightens the burden of the web server, and ensures the scalability of the project.

Data Post-Processing Post-processing is to ensure that the raw data outputted by the CV module is interpreted in a way so that the web application could use it directly and display it to the user. There are 2 main parts of the process: Perspective adjustment and occupancy calculation

1. Perspective Adjustment

   This process is due to the fact that we chose to mount the camera in an angle instead of mounting it overhead. The process is mainly divided into 2 parts: X coordinate adjustment, and Y coordinate adjustment.

   In a typical perspective adjustment function, such as *cv.getPerspectiveTransform* or *cv.findHomography*, the basic input we need is the 4 points of the quadrilateral shape we want to warp. In our case, the most ideal way is to gain the corners of the ground, such that we can warp the ground and create an overhead view of the room. However, due to how the camera is placed, it is difficult to see the lower corners of the room. Additionally, there are also quite a few difficulties.

   First, it could be the case that there is no upper corner, which means that the ground could be extending to areas outside of the camera. Secondly, the chairs could be blocking the corners. Thirdly, cropping to the ground is not good enough, because chairs have heights, thus if we only crop the quadrilateral shape according to the 4 corners defined above, then we might crop out some chairs.

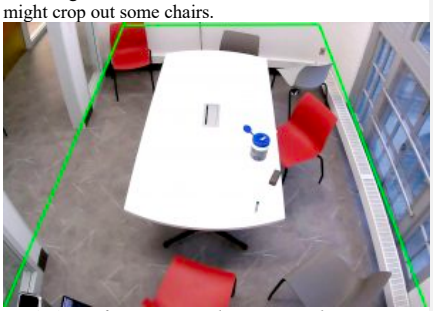


Figure 6: Area of perspective adjustment application

   Because of the difficulties above, we decided we could instead apply perspective transform to the image with the definition that the lower 2 corners are the 2 corners of the image, while the upper 2 corners are on the upper edge of the image, but with a shorter length. In other words, we are applying a perspective adjustment to a trapezoid. We have tried a few different methods to gain the ratio of the upper and lower vertice, one way is to apply canny edge detection on a the table to simulate the ratio or angle of the camera:

   We first apply image pre-processing on the image, such as increasing contrast, or applying a bilateral *filter* to the image to prepare for edge detection. After applying *Canny Edge Detection*, we select for the edge with the longest length, and ensure that it is a closed shape. We then apply *cv.approxPolyDP* so that the shape of the table is simplified to a quadrilateral shape. Then we select for the 4 corners of the shape, and thus we acquire the ratio of the upper and lower edge.

   However, besides the difficulty explained previously, this method requires a high contrast background with the table, a good lighting condition, and also, most importantly, that there are little objects around the table, which is rarely the case in our situation. An example of the longest outline we get by running the upper processes - best result after playing around with the values and parameters in image preprocessing:

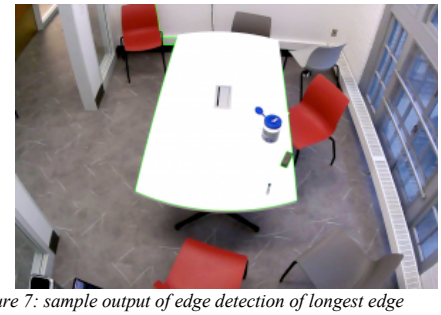


Figure 7: sample output of edge detection of longest edge

As shown above, the method above is unstable, and varies greatly from different ways of object placement and also lighting conditions. While there are other solutions such as image segmentation, for example, Detectron, it is not only an overkill to our application and greatly increases processing time , it also requires data to train on for it to function on ground.

Our Solution: For the X-coordinates, the idea is pretty straightforward; we simply stretch the upper vertices to the side to fill the image. This is inline with the majority of perspective adjustment functions.

For the Y-coordinates, however, we adopt the idea of "Psuedodepth",

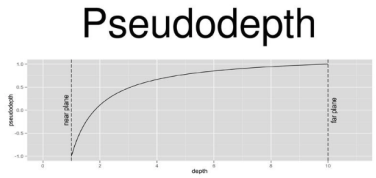


Figure 8: Pseudodepth versus depth graph

where we assume that we are dealing with a 2d plane, and, according to the fact that:

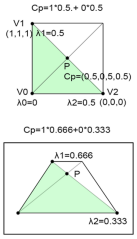


Figure 9: real center point overhead and at angle

In an angled view, P is still the real center of the "room", and the y coordinate of P is dependent on the ratio of the upper and lower vertices. Thus, applying those conditions, we can solve for an equation with y respective to the ratio of the upper and lower vertice. This ratio is the only variable, and greatly affects how accurate the algorithm is.

There is one problem with this algorithm, which is that any chair outside of the "trapezoid" would be excluded, thus, to fix this, we added one more step to locate anything outside of the bounds to be on the edge of the map.

Another problem is that after adjustment, and even before adjustment, the center of some of the chairs are located inside of the table. Thus, there is one more step to move each chair outside of the table. The way we are moving it, in other words, whether we are moving it upwards or downwards, leftwards or rightwards, depends on the closest edge of the table it is to.

There is also the problem that, since we are only adjustment for the center point, this is fine in the case of a chair, however, in the case of a table, since the width and height of the table also needs to be adjusted, we need to apply the perspective adjustment function to top right, bottom right, and bottom left points respectively.

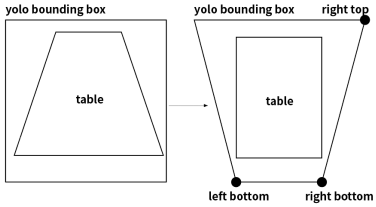


Figure 10: Table before and after perspective adjustment

As shown above, the yolo output bounding box is not a perfect outline of the table before adjustment: for the upper two corners, it is significantly off in terms of x-coordinates. Thus, as we can see after adjustment, the width of the table is actually the difference between left and right bottom x coordinate value, but height is the difference between the right bottom and top y coordinate value. Errors occurred before making this change with the map showing a really "fat" and shifting to the right table, due to using only the top right and bottom left xy coordinates.

2. Occupancy Calculation

The problem with defining "occupied", is that we can not differentiate between a standing and a sitting person. Thus, for a person to be defined as "sitting on a chair", we have to have a way to define that the person is "close enough". Since the conditions of the

camera differ room by room, we can not depend on the pixel values of the image. This is a problem that we have to deal with throughout this project.

In calculating occupancy, we first find the closest seat to each person, with the coordinates after perspective adjustment. This ensures that a person can only sit on one chair. After that, we check whether or not the bounding box of the chair and person overlap, and this is our definition of "close enough", or that the person is sitting on the chair.

Thus, error will occur if a person is standing next to a chair, as the system will reckon it as sitting on the chair.

Backend: The framework we chose is Django. There is not much difference between different frameworks, such as *FLASK,* an alternative to Django, thus we chose Django based on familiarity. Additionally, since we are transferring Data via JSON format, data will be stored in the form of a JSON file.

The server is now much simpler will be only be responsible for 2 things , after the systematic adjustment mentioned above: handle information sent by jetsons, and handle information requests sent by users.

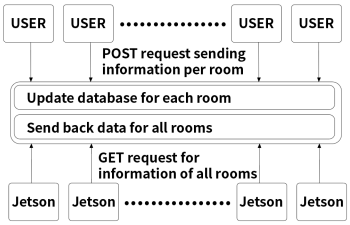


Figure 11: Web Application Server Diagram

The room name will be provided when the information is sent to the server. This allows for the server to update information only relevant to that room. All room information will be sent back to the user, so that the user does not have to send a request to the server to acquire information on that specific room whenever the user switches rooms, also, this allows for the user to search/order the rooms according to a specific value without sending a request to the server for the server to do it. This is a trade off between scalability/speed and memory.

Frontend: The frontend, namely the website including a html and css file, and a javascript file that , on loading, will send a request to the server side requesting the information of a certain room with a "GET" request. This is different from before as now the frontend will be requesting information of all rooms, instead of just that one room the website is looking at. All the information on the page will be updated with "AJAX"(Asynchronous JavaScript And XML).

In the javascript file, there are mainly 3 functions: First, one that sends a request to the server, one that updates the map, and one that reorders the roomlist according to the ordering.

The send request function will be called when the page is initialized, and then called every 45 seconds. When the server responds with all the data, the javascript will save the JSON data, and call the other 2 functions, namely update map and update roomlist. The update map function draws the map according to the coordinates of the seats and chairs, while the update roomlist function refreshes the room lists.

The update map function will also be called when another room in the room list is clicked, showing the map of that specific room, with information acquired from the locally saved JSON response.

The update room list function will also be called when the order function is called, as we reorder the rooms depending on what the ordering parameter is(number of seats/occupied seats/available seats etc.)

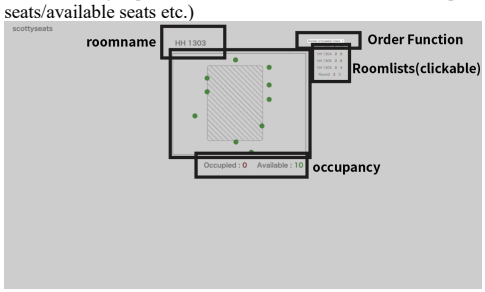


Figure 12: Web Application Layout

VII. Test, Verification and Validation

A. Tests for Use- Case: Object Accuracy

Table 6 at the end of the report shows comprehensive metrics across all training for the Yolov5L model. The overall goal metric was to achieve an overall accuracy of 90%.

Through testing, we found that transfer learning becomes optimized when freezing 22 layers or the backbone of the YOLO architecture - with both producing the precision accuracy (mAP 0.5) after 50 epochs of training (0.69 and 0.70). Removing the backpack class once again produced a jump in accuracy of almost 17% when freezing 22 layers and 27% when freezing the backbone.

However this also resulted in high confidence false positives during live testing. Training was redone by including on PASCAL VOC dataset training to combat this, however this resulted in a drop in mAP accuracy to 0.77. The Pascal VOC dataset was too general of dataset to effectively train out high confidence, false positives while also retaining overall accuracy.

Commented [5]: FINAL REPORT: The Test, Verification and Validation section must be updated to include your testing
results as well as QUANTITATIVE verification of your design requirements, comparing
test results with your design analysis when appropriate, and QUANTITATIVE validation
of your use-case requirements. Include discussion and assessment of the results that
explains the relation between the results and the design requirements and use-case
requirements, especially in cases where aspects of the requirements were not fully met.

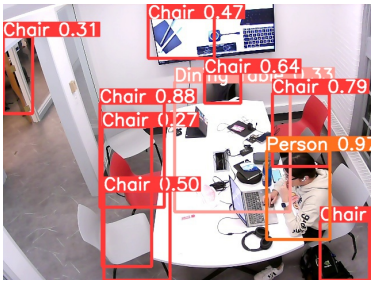


Figure 13: Sample Output after PASCAL VOC Training

One final change made was to switch from the custom class mapping to the original pretrained 80-class mapping. Doing this switch preserved the existing sensitivity of the model against false positives while also increasing detection confidence of the target classes: chairs, tables, people. Training just on the custom dataset with this mapping achieved the highest overall accuracy at 0.909 while also preventing high, confidence false positives in live testing.

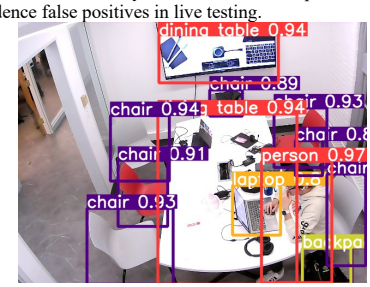


Figure 14: Sample Output after Custom Training with COCO Class Mapping

B. Tests for Use- Case: Occupancy Accuracy

Table 2: Occupancy Testing

Room Number	Predicted Occupancy	True Occupancy	Predicted Seatcount	True Seatcount
A	1/8	1/8	8	8
B	0/8	0/8	8	8
C	2/8	1/8	8	8
D	4/8	3/8	8	8
E	2/8	2/8	8	8
F	6/8	4/8	8	8
G	4/8	4/8	8	8
H	0/8	0/8	8	8
I	2/7	0/7	7	7
J	0/8	0/8	8	8

As shown in the graph above, among the 10 images we have run, out of 79 chairs, the occupancy of 7 chairs was incorrectly noted. The accuracy this turns out to be around 91.1%. Having a deeper look into the reasons that cause the error, we found out that the only case where there is an error in occupancy is when a person is standing close enough to the chair(meaning that the building boxes of the chair and person are overlapping).

Due to our limited testing conditions, however, we expect to have a different result for a different room, and the results may vary depending on the test images given(how many "standing near a chair" cases occur).

Overall, our occupancy accuracy seems to meet the user-case requirements.

C. Tests for Use-Case: Speed

Table 3: Processing Time vs Architecture

	Pre processing (s)	Inference (s)	Post Processing (s)	Total Time (s)
Quad-Core Intel Core i5 with PyTorch inference	0.02	1.59	0.01	1.62
Jetson Nano 2GB with PyTorch inference	0.05	0.80	0.05	0.94
Jetson Nano 2GB with TensorRT inference	0.05	0.33	0.05	0.44

As seen by Table 2, though the preprocessing and postprocessing are faster on Mac, with Quad-Core Intel Core i5, the speed up time on inference on the Jetson Nano means that the total time to process an image is much faster on a Jetson Nano. Switching from Pytorch inference to TensorRT resulted in a ~2.5x speedup in inference time. The speedup was not needed to meet our update speed requirement but because running Pytorch out of the box on the Jetson maxes out the physical memory available on the Jetson. TensorRT optimizes the GPU's memory and bandwidth by fusing nodes within the kernel [10], which enabled us to run 2 instances of inference at the same time.

Our CV pipeline takes 10 samples before calculating the sample with the highest confidence. Once this is determined it is sent

as a POST request to the web server. Taking 10 samples, once every second, so the total time between taking the first sample and sending a POST request is ~10.5 seconds.
Since it is a POST request, changes are immediately reflected in the webserver. Users receive updates to the front-end via a GET request which has been set to 10 seconds. This means in the worst case scenario that a change will be reflected in approximately 21 seconds after it has been observed which meets our use case requirement for update speed.

D. Tests for Use- Case: Spatial Accuracy

To ensure sufficient Spatial Accuracy, we initially aimed for an average positional error of less than 20% amongst all objects in a room. To measure this, we aimed for an MAPE of the relative distance between the chairs and table of under 20%.

To achieve this, we took 13 images of approximately 100 chair positions with a secondary overhead camera, as shown in figure 10, to act as a 'ground-truth' of the true positions of objects in the room. The chair/table positions were then calculated using hand-drawn bounding boxes on each image.

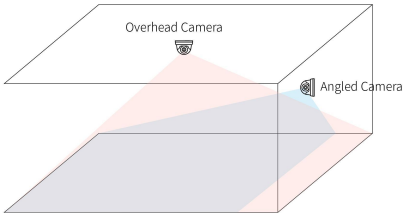


Figure 15: Spatial Accuracy Testing Setup

We then measured the difference in chair/table positions between this ground truth top-down view and our system output using MAPE, and compared how this value changed both before and after perception correction.

$$\text{MAPE} = \frac{100\%}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

X^*_n : Ground-truth (True) distance of chair to centerline of the table in top down view
X_n : System's predicted distance of chair to centerline of the table
F_t : X_n / X^*_n of a single chair
A_t : Average X_n / X^*_n, simulating the true ratio (1)

(1) shows how MAPE was calculated, we found the distances between the centerpoints of each chair and the table in an image, and then took a ratio (F_t) between our system's predicted distances (X_n) and the ground truth distance (X^*_n) for each chair. Using the average distance ratio (A_t) of all chairs in the room as the 'true ratio', we calculated MAPE between the average distance ratio and the distance ratio of all chairs in the room. MAPE values were calculated for both the X-dimension (horizontal axis) and Y-dimension (vertical axis) as perception correction was done independently for each dimension.

Table 4. Average MAPE With and Without Perception Correction

Dimension	*Without Perception Correction*	*With Perception Correction*	*Change in MAPE*
X (Width)	96.02%	104.88%	**+8.87%**
Y (Length)	82.12%	78.29%	**-3.83%**

In total, with perception correction, we found that we weren't able to reach an MAPE of under 20%, but we were able to see considerable improvement in some MAPE in the Y-dimension. We saw an average drop of 3.83% MAPE in the Y-dimension and 8.87% increase in the X-dimension. A full table of all measured MAPE values can be found at the end of the report (Table) Our perception correction algorithm did allow for some improvement in the X-dimension, but requires further tweaking overall to fully improve MAPE as intended.

Two potential sources of error in our MAPE calculation come from the setup of the overhead camera and manually drawing the bounding box positions for the ground-truth images.

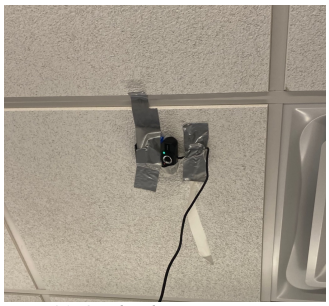


Figure 16: Overhead camera mounting setup

Mounting the overhead camera proved considerably difficult as a wire on the back of the camera prevented us from mounting it facing fully downward. The ceiling panels also pushed up easily, making it difficult to stick a camera to the ceiling as well.

Commented [6]: There are 2 kinds of positions in which the camera would be placed: an angled camera and an overhead camera (Figure 2).

Figure 1: System Block Diagram

The angled camera would be responsible for gaining input for detection, the training data, which is used to train for further training the weights of the CV model, and also the testing data, which would be used to verify the accuracy of the model output. Both the training and testing dataset would be processed later with labeling tools such as the Computer Vision Automated Tool (CVAT). The overhead head camera would only be used for collecting testing dataset for verification of the accuracy of the position of the seat after the perspective adjustment on the server side of the webapp.

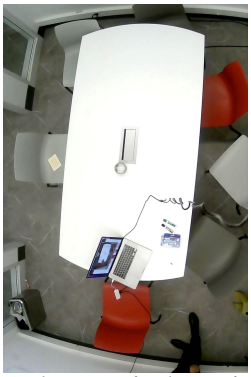


Figure 17: Top-down view of study room from overhead camera

As can be seen from the sample photo, chairs were sometimes cut-off from view due to a low ceiling and camera angle. This made it difficult to draw bounding boxes that accurately reflected chair positions.

VIII. Ethical Issues

The largest ethical issue that comes up is regarding privacy and confidentiality. Data sovereignty is preserved by processing images from the location they are captured, on the Jetson Nano[1]. After data is processed it is deleted, meaning historical data cannot. This allows us to keep sensitive data safe by never transferring over a network, the only data being sent is the location of the seats and tables. Since we are calculating the occupancy before sending it to the web server, people are completely anonymized and cannot be tracked. Additionally, we have our website accessible only within the CMU network to ensure that our data is accessible to possible users of our application. A possible edge case is that users abuse the system if they understand how occupancy is calculated. Users who want more space could fool our algorithm to ensure others do not take their space. Anyone looking for seats would be affected by the issue. While these were not implemented for the scope of the project, encrypting JSON requests sent from the Jetson to the server and adding in firewall and cryptography protocols to our system can help to mitigate potential attacks to our system.

IX. Project Management

A. Schedule

The major modifications to our project schedule were to do with model training, Jetson setup and restructuring. We had a lot of iterations of our model before finding one that met our requirements. A few weeks were lost trying to meet the software requirements for the rest of our pipeline, in the end we downgraded our libraries to be compatible with the Jetson. After meeting with the professors we found that our solution approach was not as planned out as it should be. After a few discussions, we changed some of the interactions between our components to be more modular and scalable. These implementation changes delayed us by a week or so

A detailed copy of our final schedule is presented in Figure 8.

B. Team Member Responsibilities

Team member responsibilities, primarily, are delegated based on each member's expertise - Aditi: Hardware, Mehar: Computer Vision, Chen:Web Interface. Larger sections like CV and integration/interfacing tasks, are also split across members to make sure work is evenly split. Throughout our timeline as well, members will be working together for tasks that may need more hands.

Aditi:
- Jetson Setup (YOLOv5, web server)
- Converting PyTorch models to TensorRT
- Sampling Algorithm
- Planning and Team Status Report

Mehar:
- Data Preparation: Data Augmentation, Cleaning and Relabelling
- Data Collection
- Yolov5 Model Training
- CV Pipeline Integration

Chen:
- Web Application Development
- Data Post-Processing
- Translating CV output for Seat Map
- Interfacing from Jetson to Web App

Mehar and Aditi:
- Image Preprocessing
- Custom Data Labeling
- Object Classification Research

Tasks were also split with final testing amongst team members as well. Chen worked on occupancy accuracy testing, Aditi on speed testing, and Mehar on object accuracy testing. The spatial accuracy testing metric methodology was developed by Chen, Aditi and Mehar, while final calculations and testing were carried out by Mehar.

C. Bill of Materials and Budget

- The bill of materials and budget can be found in Figure 7.

[1] https://blogs.nvidia.com/blog/2022/01/05/difference-between-cloud-and-edge-computing/

D. *Amazon Web Services Usage*

○ Custom training of the Yolov5 model was made possible through AWS services. Specifically, $100 worth of credits and another $40 were used in training the Yolov5 model, and data preparation and storage. Credits were used for 180+ hours of p2 and p3 EC2 instance usage, and 7.7 GB of data storage on S3. We would like to thank AWS for providing us the resources to make training possible.

E. *Risk Management*

One of the risks we initially had was occlusion of the chair a person may be sitting on. In practice, we found that we were able to place the camera high enough and our dataset was robust enough to account for most cases of this. Instead, a problem that similar color clothing to the chair can cause the system to miss the chair entirely, thinking the chair is part of the person. We were able to mitigate this through our preprocessing by greatly increasing contrast and overall exposure of the image.

Another risk we had was oversensitivity to changes - specifically, if someone were to leave the room to go to the bathroom for a few minutes. We decided to keep this case as further features once we reached MVP for our project. Due to delays in integration, we did not have enough time to fully implement this feature. However, our plan to mitigate this issue was to use an extra class to detect the presence of backpacks, laptops and other items to also tell us whether a chair is potentially occupied. If we detected a person or multiple school items by the chair, then the chair would be counted as occupied.

X. Related Work

We are mainly aware of 4 projects that are similar to ours: Fall 2020 Team B4 Smart Library, Fall 2021 Team A3 FreeSeats and Spring 2022 Team E4. Ours differs from Smart Library as we have dynamic seat mapping which accounts for more cases and situations, while FreeSeats are using sensors mounted on a chair to detect and monitor each chair and reflect it on an app.

XI. Summary

With the implementation of our design, students no longer have to travel around campus to find a study space as they could do so just by a single click. Our solution is to place cameras in study rooms and use computer vision to identify available seats. We will display the available seats and where they are located on a web application. We hope to save users time, and more efficiently utilize all of the study spaces CMU has to offer.

In terms of future work, we have found that our CV module is not as accurate in other spaces. Our custom dataset was limited and only had pictures of rooms within Hammerschlag. It would be beneficial to train our machine learning model with different chairs, with different perspectives and with more variations in lighting. Additionally, it would be good to have our website to only be accessible to people signed into their CMU accounts to keep this information more confidential. Though we were able to scale our MVP to 2 cameras, to better utilize the Jetson, an expensive resource, it might be beneficial to use the third USB A to host a third room. This may require porting our inference to DeepStream.

We have definitely learned the importance of communication. Making sure there are clearly defined and reachable deadlines are critical to having a successful project. Reaching out for help early is also an important lesson learned. We found that we were able to find solutions to our issues much faster.

Commented [7]: FINAL REPORT: explain how we handled project risks

Glossary of Acronyms

AJAX - Asynchronous JavaScript And XML
CV– Computer Vision
CLAHE - Contrast Limited Adaptive Histogram Equalization
R-CNN - Region-Based Convolutional Neural Network
ML - Machine Learning
RPi – Raspberry Pi
JSON - JavaScript Object Notation
XML - Extensible Markup Language
YOLO -You Only Look Once
CVAT - Computer Vision Annotation Tool
POST - request method supported by HTTP [2]

References

[1] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "Yolov4: Optimal speed and accuracy of object detection." *arXiv preprint arXiv:2004.10934* , 2020.

[2] Ultralytics, "Ultralytics/yolov5: Yolov5 in PyTorch > ONNX > CoreML > TFLite," *GitHub*. [Online]. Available: https://github.com/ultralytics/yolov5. [Accessed: 14-Oct-2022].

[3] H. Feng, G. Mu, S. Zhong, P. Zhang and T. Yuan, "Benchmark Analysis of YOLO Performance on Edge Intelligence Devices," 2021 Cross Strait Radio Science and Wireless Technology Conference (CSRSWTC), 2021, pp. 319-321, doi: 10.1109/CSRSWTC52801.2021.9631594.

[4] Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[5] S. A. K. Tareen and Z. Saleem, "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK," 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), 2018, pp. 1-10, doi: 10.1109/ICOMET.2018.8346440.

[6] "Clahe histogram equalization - opencv," *GeeksforGeeks*, 09-Nov-2021. [Online].Available:https://www.geeksforgeeks.org/clahe-histogram-eqalization-opencv/. [Accessed: 16-Dec-2022].

[7] "Histogram equalization," *Wikipedia*, 29-Jun-2022. [Online]. Available: https://en.wikipedia.org/wiki/Histogram_equalization. [Accessed: 16-Dec-2022].

[8] "Changing the contrast and brightness of an image!," *OpenCV*. [Online]. Available:

[2] https://en.wikipedia.org/wiki/POST_(HTTP)

https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html. [Accessed: 16-Dec-2022].

[9] "Python opencv - filter2d() function," *GeeksforGeeks*, 05-Nov-2021. [Online]. Available: https://www.geeksforgeeks.org/python-opencv-filter2d-function/. [Accessed: 16-Dec-2022].

[10] "Faster yolov5 inference with TENSORRT, run YOLOV5 at 27 FPS on Jetson Nano!," *Latest Open Tech From Seeed*, 29-Aug-2022. [Online]. Available: https://www.seeedstudio.com/blog/2022/08/23/faster-inference-with-tensorrt-on-nvidia-jetson-run-yolov5-at-27-fps-on-jetson-nano/. [Accessed: 16-Dec-2022].

[11] T. Yeung, "What's The difference: Edge computing vs cloud computing ," *NVIDIA Blog*, 17-Sep-2022. [Online]. Available: https://blogs.nvidia.com/blog/2022/01/05/difference-between-cloud-and-edge-computing/. [Accessed: 16-Dec-2022].

[12] "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91 [Accessed: 13-Oct-2022].

Item	Manufacturer	Model Number	Quantity	Cost	Notes	Link to Buy	Description
Jetson Nano 2GB	NVIDIA	P3541	1	0	From course list	From course list	GPU for hardware acceleration and hosting web server
USB Cameras	TedGem	CE0140_01	2	0	From course list	From course list	Capture video from study room
USB Extension Cable	AINOPE	N/A	1	13.99	amzn1.sym.67f8cf21-ade4-4299-b433-69e404e		Extend cameras further
Ethernet Cable	Amazon Basics	HL-001764	1	9.76	Back-up for WIFI adapter	cf21-ade4-4299-b433-6	Connecting Jetson to LAN (Backup option)
WiFi Adapter	Edimax	EW-7811Un V2	1	9.99	zn1.sym.67f8cf21-ade4-4299-b433-69e404eeec		Connecting Jetson to WIFI
Micro-USB to USB C	Cable Matters	201003-BLK-1m	1	7.99	m.67f8cf21-ade4-4299-b433-69e404eeecf1&pf		Connecting Jetson to Computer
Reclosable Fastners	3M	N/A	1	7.88	To mount and unmount cameras	433-69e404eeecf1&pf_r	Mounting camras
AWSCredits	Amazon Web Services	N/A	1	100	$100 credits		GPU access for model training and indvidual testing
			Total	149.61			
			Budget Left	450.39			

Figure 18: Table of Materials and Costs

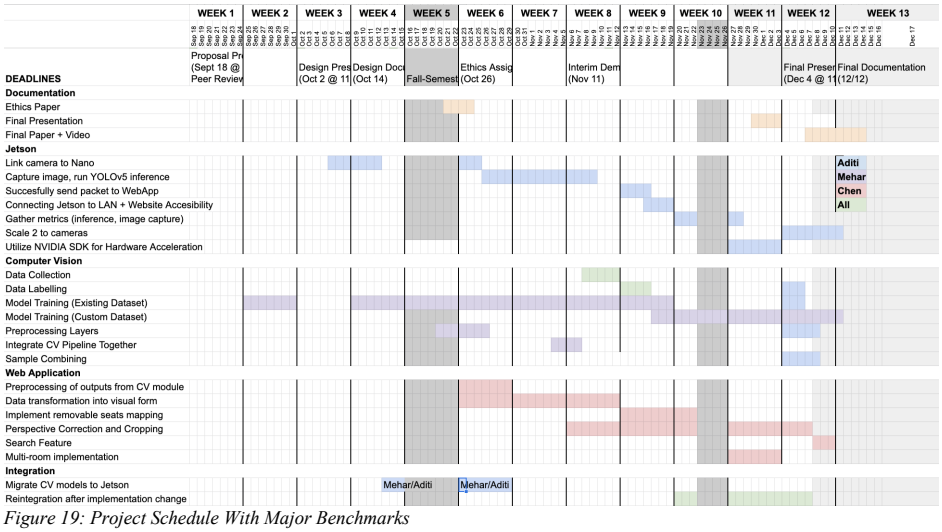


Figure 19: Project Schedule With Major Benchmarks

Table 5. Spatial Accuracy MAPE with vs without Perception Correction

	X-dimension (Width)			Y-dimension (Height)		
Sample #	***Without Perception Correction***	***With Perception Correction***	***Change in MAPE***	***Without Perception Correction***	***With Perception Correction***	***Change in MAPE***
1	156.60%	160.08%	3.48%	72.58%	79.96%	7.39%
2	37.04%	72.47%	35.43%	59.69%	44.89%	-14.80%
3	40.39%	66.73%	26.34%	94.68%	59.72%	-34.96%
4	48.60%	119.84%	71.24%	96.06%	68.06%	-27.99%
5	136.62%	144.17%	7.55%	103.97%	79.54%	-24.43%
6	159.33%	161.81%	2.48%	92.10%	95.85%	3.75%
8	133.72%	118.19%	-15.53%	124.19%	114.88%	-9.32%
9	133.81%	132.99%	-0.81%	68.17%	83.14%	14.98%
10	147.42%	136.43%	-10.99%	93.02%	67.05%	-25.96%
12	47.14%	50.07%	2.93%	63.76%	69.36%	5.60%
13	51.54%	59.63%	8.08%	74.97%	111.04%	36.07%
14	34.40%	32.28%	-2.12%	65.45%	39.57%	-25.88%
15	121.64%	108.81%	-12.82%	58.92%	104.66%	45.74%
Average	**96.02%**	**104.88%**	**8.87%**	**82.12%**	**78.29%**	**-3.83%**

Table 6. Model Training Results

Model Description	**Epochs**	**mAP 0.5**	**mAP 0.5:0.95**	**Precision**	**Recall**
Testing for Optimal Transfer Learning Configuration					
Custom Data, Freeze 24 layers	50	0.54103	0.28801	0.69936	0.50833
Custom Data, Freeze 23 layers	50	0.61379	0.36274	0.84041	0.52138
Custom Data, Freeze 22 layers	50	0.69160	0.41573	0.85648	0.64115
Custom Data, Freeze Backbone	50	0.70822	0.44626	0.89579	0.67487
Custom Dataset Training Without Backpack Class					
Custom Data with No Backpack Class, Freeze 22 Layers	50	0.87396	0.58691	0.85267	0.84129
Custom Data with No Backpack Class, Freeze Backbone	50	0.97384	0.66036	0.90455	0.93743
Custom Dataset Training With Pascal VOC using Custom Labeling and No Backpack Class					
Pascal VOC w/Custom Labeling, Freezing 10 Layers + Custom Dataset,	50+10	0.53313	0.34305	0.94968	0.40855

Freezing 22 Layers					
Pascal VOC w/Custom Labeling, Freezing 10 Layers + Custom Dataset, Freezing 22 Layers	50+30	0.76944	0.47584	0.85187	0.72637
Custom Dataset Training With Pascal VOC using COCO Labeling					
Custom Data w/ COCO Labeling, Freezing Backbone	50	0.90979	0.72143	0.97086	0.87832
Custom Data w/ COCO Labeling, Freezing Backbone + Pascal VOC w/COCO Labeling, Freezing 22 Layers	50+10	0.82936	0.58633	0.77764	0.78755
Pascal VOC w/COCO Labeling, Freezing Backbone	50	0.86833	0.66212	0.81220	0.80924
Pascal VOC w/COCO Labeling, Freezing Backbone + Custom Data w/ COCO Labeling, Freezing 22 Layers	50+10	0.65957	0.40164	0.65979	0.63132
Pascal VOC w/COCO Labeling, Freezing Backbone + Custom Data w/ COCO Labeling, Freezing 22 Layers	50+50	0.90029	0.72916	0.97410	0.87656