MMDBot: A proposal and partial implementation of an object-detection-based robotic navigation and object manipulation system.

Justin Dannemiller and Keith Martin Machina

***Abstract*—The field of robotics has grown tremendously and more impressive applications are being built every single day. Some of these applications in robotics include: delivery robots, object picking robots in warehouses, rescue robots and many more. Many contemporary robotic systems, however, are inherently limited in their generality and have excessive reliance on human operators. To meet the widespread adoption of robotic systems that has long been envisioned, it is imperative that robots be designed to handle the complexity and uncertainty of the real world through greater integration of artificial intelligence (AI) components. This paper proposes a system that could be used to afford autonomous navigation and object manipulation in robots and demonstrates and discusses the object detection component of that system.**

***Index Terms*—Artificial Intelligence, Object Detection, Object Manipulation, Robotics, Supervised Learning, Transfer Learning.**

I. Introduction

Many have envisioned of a future world in which our lives are heavily assisted by robotic systems. Example applications include robotic care-giving for the elderly, relegation of laborious household chores to household robots, and autonomous crop cultivation. Despite improvements in robotics, artificial intelligence (AI), and hardware, this widespread adoption of robotic systems in our society has yet to be realized. Robots have seen only limited deployment, often being used only settings such as in hospitals performing telesurgey or in industry to afford high precision manufacturing.

The success of robots in these environments, however, can mostly be attributed to the constrained nature and simplicity of the environments in which they are performing. Such robotic systems are often either directly controlled by a human or are otherwise only executing very defined tasks in settings where the conditions are always are always known and constant. Success in such applications can be achieved relatively easily, as the controlled and certain nature of these environments can effectively be handled with static coding of the robot's movements. The real world, however is very uncertain and unknown, and in future robotic systems are to be successful and independent of human input in that world, it is imperative that they be instilled with more generalized and response decision making and action planning faculties. This can be achieved through greater integration of artificial intelligence (AI) components.

In light of this consideration, this paper discusses a deep learning leveraging system that was proposed in an attempt to afford autonomous navigation and object manipulation in the Telebot-3-R. The Telebot-3-R is a humanoid, telepresence robot designed for search and rescue operations, and is shown in Fig. 1. Despite being an effective tool, it is significantly limited in AI capabilities, with most of its operations depending completely on a team of human operators. Unfortunately, this dependence inherently limits its reliability in real-world environments where network connections may often be weak or unstable. The design of the system proposed, if implemented in its entirety, could potentially be used to autonomously manipulate objects within a robot's environment using a robot arm. Due to the complexity of and the inherent danger of working with the Telebot, the design of this system was tested with a smaller robot, as discussed in . This paper, moreover, focuses its discussion the implementation of the object detection component of the system.

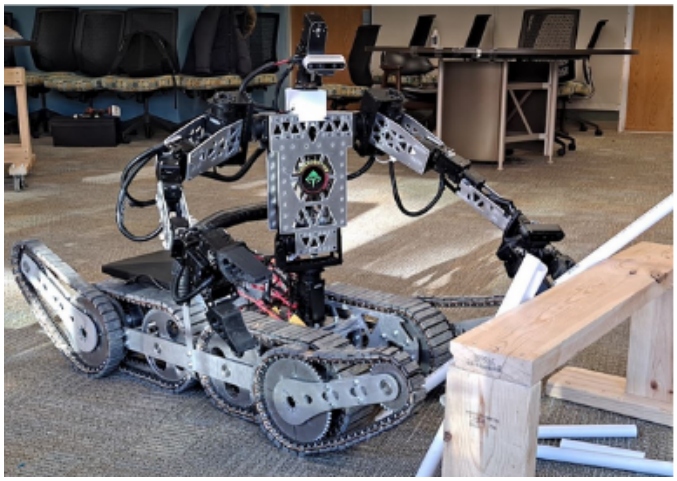

Fig. 1: The Telebot-3-R operating in a test environment

The following section, section II, provides an overview of other works that have similarly investigated the application of deep learning in informing the decision making and motion planning operations of robots. Next, section III describes the components of the proposed system. Section IV continues with an outline of how the object detection component was implemented, and the results of and hindrances experienced in implementing this system are explained in section VI. Finally, section VII concludes with a discussion of the limitations of implemented object detection system and the future steps for development of the system.

II. Survey

Robotics is a being a multidisciplinary field, encompasses multiple disciplines of computer science. Some of these disciplines include: machine learning, algorithms, computer vision, security just to mention a few. Motion planning happens to be one of the key topics that arise at the mention of robotics. Motion planning involves determining an efficient path to a Target object or Target state. In detail, this involves determining how many degrees, the relevant degrees of freedom should move. Some of the strategies to motion planning that have been suggested, proven and tested include: use of computer vision, neural networks, reinforcement learning, inverse kinematics and more. Numerous algorithms have been suggested in the past its still a challenging to achieve efficient and effective motion planning.

In [1] , researchers suggest that efficient motion planning can be achieved by amalgamating a number of techniques, namely: object detection, use neural networks, reinforcement learning.In their proposed solution, firstly, object detection had to be performed to determine the exact location of various objects in the space. To perform this You Only Look Once (YOLO) model was used. Through YOLO the researchers were able to determine a target, and obstacle and a bounding box around the the identified objects. Right after the YOLO model, they then went ahead to extract coordinates of each object within the space in form of planar coordinates. A neural network would then be applied to convert these planar coordinates into a spatial coordinate system which is a simplified version. The context being a robot arm trying to grab an object, the next step was to determine the arm movements that would be appropriate to get to the object. This involved determining whether the arm moves right, left, up or down. The series of movements was determined using a reinforcement learning technique known as Qlearning, where robot was rewarded or penalized based on the action in took in regard to the position of the target object. Any move in the direction of the target earned the model 50 points while any move that was not in line with the target was awarded a -100 points. This proved to be quite efficient as in most cases as the robot only moved more towards the target. Last step in this proposed solution was the use of a neural network to convert the series of arm movements outputted by the reinforcement learning step into a series of join states. This basically involved calculating the angles between each arm movement. The researchers claimed yield quite desirable results.

At the heart of robotics, is automation. In [2], researchers proposed use of inverse kinematics as means to efficient motion planning with the intention being to autonomously grab a target object. To achieve this, they set up their experiment using a depth camera, KINETC II to be precise, in order to obtain RGBD images. The D in the RGBD images represents the distance to the image which was later useful when performing arm manipulation. A YOLO model was then applied to extract CNN features on the image through semantic segmentation. This extracts only the pixels of the object of interest. Next, the researchers would apply inverse kinematics now that the position of the arm is known. Inverse Kinematics would use the equation[1] to determine how many degrees each degree of freedom would move. Despite inverse kinematics being efficient it was quite slow since in involved a lot of mathematical calculations.

$$\Delta\theta = J^T \left(JJ^T + \lambda^2 I\right)^{-1} \vec{e} \tag{1}$$

In [3] researchers proposed automation of pick-and-place of objects such as glass bottles. The robot had to pick items from a starting position to a predefined end position. To effectively do this they had to determine the starting and an end position. First was to determine where the object is located to the arm, a process performed using stereo vision. RGB markers were used to mark the centre to continuously track the object. Two cameras were used to find the 2D location of the object in the image and two equations shown below later applied.

$$S_L * \begin{pmatrix} u_L \\ v_L \\ 1 \end{pmatrix} = h_L \begin{pmatrix} X_{c,L} \\ Y_{c,L} \\ Z_{c,L} \\ 1 \end{pmatrix} \tag{2}$$

$$S_R * \begin{pmatrix} u_R \\ v_R \\ 1 \end{pmatrix} = h_R \begin{pmatrix} X_{c,R} \\ Y_{c,R} \\ Z_{c,R} \\ 1 \end{pmatrix} \tag{3}$$

The points are then triangulated using another equation to transfer the location of the object into a single coordinate. Once the position to a single coordinate, to move the object of interest, the position has to be inline with the arm. This position is then transformed into arm coordinates. Path trajectory is then predicted using regression involved Non-Linear Principal Component Analysis (NLPCA) and an artificial neural network (ANN).

Recently, motion planning techniques are beginning to leverage machine learning as a means to efficient motion planning. In [4], researchers propose MPNet, a motion planning model consisting of two sub-models (Enet and Pnet). Enet encodes point cloud of environmental objects into a latent space. Then, Pnet takes this encoded spacial data of obstructions and predicts a collision-free path around them traversing from a starting position to a goal position. Another study examines the use of computer vision for predicting a path for a 6-Degrees-of-Freedom robotic arm [5]. This study uses two cameras to triangulate the position of an object in 3D space, then uses an artificial neural network to calculate trajectories for the arm at each time step until a full path is planned. By comparing the results of the model to experimentally-derived data, they found that the system had a 91.8% accuracy in prediction.

III. System Overview

To achieve autonomous navigation and manipulation, a three component was proposed as illustrated in Fig. 2. The first part of this system is the raw data manager which manages all the data collected by the robot's sensors and its cameras through its vision agent and spatial agent. Moreover, the vision agent is comprised of an Intel Real Sense's RGB camera, an Intel Real Sense's depth camera, and a LIDAR sensor. The RGB

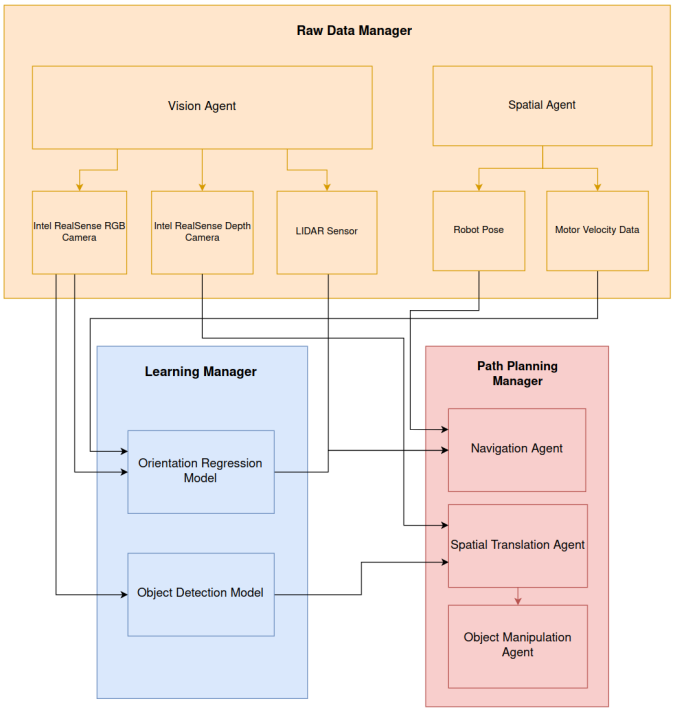

Fig. 2: System architecture diagram

camera and the Intel Real Sense depth camera are used to obtain RGB images and depth images of objects in front of the robot. The LIDAR scanner, moreover, is used to perform scanning of the robot's environment. The second component of the raw data manager is the spatial agent which manages all the spatial robot of the data. This includes the robotic pose data measured from the robot's encoders and its built-in IMU and the velocity of its base motors.

The second component of the system is the learning manager which is comprised of all the deep learning models used to perform object detection and manipulation. The learning manager is comprised of two parts: an object detection model and a orientation regression model. The object detection model takes images the RGB images and detects any objects within them. For any detected object that is to be manipulated, the center of the object would then be approximated by taking the centroid of the detection bounding box. Moreover, in order to standardize and better simplify the grasping of objects, the robot should directly face any object to be manipulated. This task would be realized with the orientation regression model. The orientation regression model would predict, given current motor velocities and images from the RGB camera, how the robot's base must turn to directly face the object. This data, moreover, would be obtained from tests in which a human operator teleoperates the base motors with a controller.

In addition to recognizing objects, the robot must also have the ability of mapping its environment and navigating to any previously detected objects. This components is achieved by the navigation agent of the path planning manager. More specifically, the navigation agent uses the LIDAR scans and robotic pose to generate two-dimensional maps of the robot's environment. Moreover, when objects are detected, the position of the objects on the map is calculated by the navigation agent. This would be performed by taking the robot's pose data and offsetting it by the object's distance from the camera, as measured by the depth image. This 2D location of the object would then be stored in XML files. This would allow the user to later instruct the robot to go to a previously detected object's position by looking up that position in the XML files. After determining the object's location, the navigation agent would publish that location to the "simple_move_base/goal" topic of Robot Operating System (ROS), effectively setting a goal position for the robot. ROS's DWA local planner would then be used to calculate and execute a path to move to that goal position given the map generated by the navigation agent.

After traversing to the object to be manipulated, the robot would then autonomously determine how to grasp the object. This would be performed by the spatial translation agent and the object manipulation agent of the path planning manager. In the first step, the spatial translation agent calculates a 3D coordinate of the object relative to the robot's camera. To do this, it first takes 2D centroid coordinate to obtain a two-dimensional estimate of the object. Next, the depth value would be extracted from a depth image at that centroid location to determine the distance of the object from the camera. These two points would then be combined to create a three dimensional estimate of the object. This spatial coordinate would then be made relative to the robot arm by transforming the coordinate using the distance between the robot arm's base and the camera. Finally, after obtaining this value, the ROS motion planning framework, "MoveIt!" would finally be used to calculate the series of positions for each motor in the robot arm required to move the end effector to the object.

It is important to note that due to time constraints and the many hardware-related outlined in section VI, only the object detection and navigation components of the system.

IV. Methodology

A. Robot Design

Due to the complexity of the Telebot-3-R and the inherent danger imposed by working with its powerful motors, the implementation of the proposed system was instead carried out with a simpler, smaller form-factor robot. More specifically, the robot used for this investigation, shown in Fig. 3, was a custom-designed robot modeled after This robot, was modeled after the ROBOTIS' Turtle Bot3 WafflePi. Compared to the original, this modified version utilizes an Intel Real Sense RGBD camera to simultaneously capture both RGB and depth images. Additionally, this system's version used a Dell Optiflex 3070 PC as opposed to the provided Raspberry Pi4 to provide the computational power needed by the models. Finally, to afford manipulation capabilities, the robot was equipped with a robotic arm known as ROBOTIS' OpenManipulatorX. Moreover, due to the custom nature of this robot, many of the robot's structural components, such as the vertical spacers in the base and the shafts in the arms were 3D printed.

The software to control the robot was built around packages of the Robot Operating System (ROS). ROS is a communication framework specifically designed for standardized communication within and between robotic systems [6].

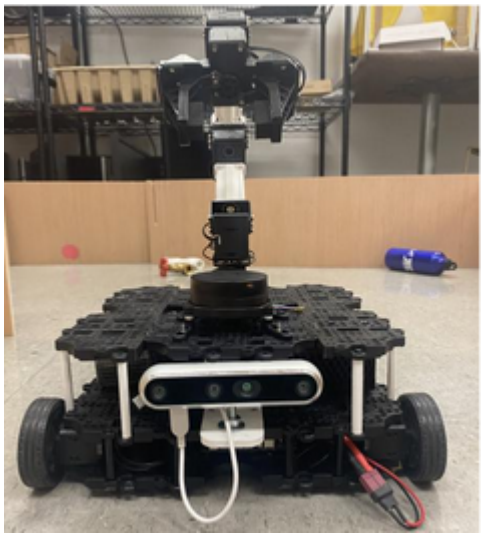

Fig. 3: The customized version of the Turtle Bot3 WafflePi with robotic arm

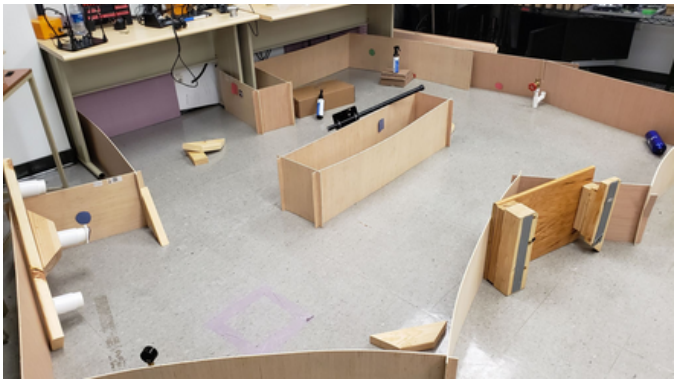

Fig. 4: Illustration of the test bed used for system evaluation

(a) The "chemical" object

(b) The PVC pipe top object"

Fig. 5: Sample objects from the test bed

B. Environment Design

To test this robot, a test bed was constructed using plywood, wooden blocks, and six classes of objects for the robot to detect. The test bed is shown in Fig. 4. The six object classes were chosen such to diversify the manipulation techniques that would be required for proper object interaction.

For instance, picking up the chemical bottle, shown in Fig. 5a from extracting the PVC pipe top, shown in Fig. 5b. The six classes of objects in the test bed, moreover, were a bottle, an obstacle, a valve, a gauge, a chemical bottle, and a manipulable PVC pipe top object.

C. Data Preparation

To achieve motion planning though computer vision the first step taken was to collect the relevant image data set that robot would encounter. This was an imperative step as it would enable the robot perceive the objects. A custom python script was used to collect the images by driving the robot around the test bed where the objects of interest had been placed strategically. There were exactly 6 classes of objects namely: chemical, obstacle, pipe top, bottle, gauge, and valve. As the robot was being driven around, images there objects on the test-bed were constantly being taken. At the end of this process, a total of 1,112 images had been collected. This data would then be loaded into an annotation software by the name roboflow. On roboflow, a bounding box around each object class would be drawn manually and a label is attached to it. Once the data set was annotated and labelled, it was segmented using a 70:20:10 split for the training, validation, and testing sets respectively.

D. Model Training and Deployment

1) YOLOv5: Once the data was ready, YOLO model was used to train the computer vision model. YOLO, which stands for, You Only Look Once, is a state of the art object detection model which implements convolution neural network to detect real-time objects, where detection is performed in one step. YOLO version 5 was the go to model for the following reasons: YOLO is fast and highly accurate. The model was trained on 160 epochs while achieving an accuracy of 99.2 percent.

This model would then be deployed on the robot for testing purposes.

2) Transfer Learning: FasterRCNN : Transfer learning was one of the proposed strategies to implement object detection capabilities in the robot. Transfer learning is a technique in machine learning that is used to transfer learnt knowledge from one domain to another or from one task to another. A good example to demonstrate transfer learning would be, a doctor treating humans can easily treat animals when need arises. The assumption here is that, the domain being the same, that is medicine, the only difference is in the tasks that are carried out. In machine learning, transfer learning is conducted by transferring of weights from a fully trained machine learning model and used in another as a base model to aid and improve training process. Basically, the idea is to provide a base, so that the model doesn't start learning from scratch and that way it can leverage on already learned knowledge.

To implement transfer learning in this project, the idea was train a custom image classification model and apply the learnt weights on a faster recurrent convolutional neural network. The image classification was a custom pytorch model that trained on the data yielded from roboflow. To train the PyTorch model, the first step was to perform some pre-processing on the data set. The following were some of the pre-processing

steps that were taken when training the image classification model. First was to resize the all images into a size of 224 by 224. Secondly, was to perform normalization. The idea here was to ensure that the pixels of the image are within the same scale. The uniformity helps improve prediction. Next would be to package the train, valid and test using the data loader to make it suitable to fit a machine learning model.

Next was to design the PyTorch model. The idea was to have convolutional layer followed by a normalization layer, and a max pooling layer that would then form one block. In total, the neural network, had four blocks. The last layer was slightly different since it had a fully connected layer at the end instead of a max pooling layer. It is important to note that the normalization layer, the batch normalization layer, was added in a each block in order to standardize the output of the convolutional layer. Max pooling was also a key component to reduce complexity through down scaling as the output is passed to the next convolutional block. Rectified Linear unit activation function was used in most of the layer in need of an activation function. ReLu activation function was the go to activation function as it solves the vanishing gradient problem caused by activation functions such as sigmoid activation function. ReLu activation function solves this problem by strictly maintaining positive values. Only positive values are selected.A cross entropy loss function was used and a stochastic gradient descent optimizer.The model was trained on 100 epochs obtaining at least an accuracy of 96 percent.

The weights of the model would then be saved and be used to train a faster RCNN object detection model. This step was quite straight forward and did not need any kind of pre-processing or tweaking. A backbone model, vgg-torch was defined to aid the training process, and the weights from the image classifier passed into the FasterRCNN model as the initial weights. The resulting weights yielded from this training process would then be save and applied during testing of the resulting model.

3) Customized TensorFlow Object Detection Model: To gain greater insight into object detection models work, a customized model was developed using TensorFlow. The model, illustrated in Fig. 6 takes, as input, a 512 by 512 RGB image and predicts 11 values for each feature in the output feature map. The 11 values include 1 value for the confidence level of the detection, four values for the two points defining the bounding box, (x_{min}, y_{min}) and (x_{max}, y_{max}), and the final 6 values provide a one-hot encoding of the class. Moreover, this 11 attribute vector is provided for each feature in the output feature map such that detected objects can be mapped to and detected from a larger number of locations from within the image. Intuitively, these attribute values are all zero for any location within the feature map within which no object was detected.

After the input layer for the input RGB image, a rescaling layer is used in the beginning of the model to rescale the RGB image's channel values from their [0, 255] input range to [0, 1] to achieve normalization of the channels and prevent biasing towards any particular color intensity. The remaining body of the model consists primarily of two-dimensional convolutional layers, two dimensional maximum pooling layers,

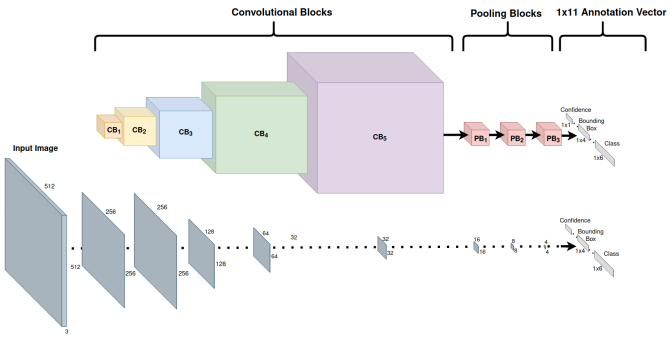

Fig. 6: Architecture Diagram for the Custom Object Detection Model

batch normalization layers, and occasional dropout layers. The convolutional layers were used to extract features from the feature maps of previous layers' output. The batch normalization layers were used address the adverse effects of the internal covariate shift during training. Moreover, the pooling layers were used to pool features together and gradually reduce the dimensions of the feature maps. Finally, dropout layers were used as a regularization technique to prevent the overfitting of the model.

The number of layers the configurations of each layer were determined experimentally by creating different model structures, training them, and them evaluating them on the testing set. In the beginning of these evaluations, an output feature map of size 32x32 was used. The evaluations revealed, however, that such a large feature map caused the model to underfit. As such, the number of pooling layers used in the model was gradually increased between the models until reaching a 4x4 map that provided considerable accuracy in predicting the 11 attribute vectors.

The number of convolutional layers and the number of convolutional features in each layer were also empirically determined. The first model tested had just three convolutional layers, two batch normalization layers, and four max pooling layers. However, the evaluation results revealed that this model was underfitting and couldn't adequately capture the complexity of the detection problem. As such, the number of convolutional layers and number of convolutional features in those layers was increased was gradually increased between models. This was performed to provide the model with enough parameters to allow it to be properly tuned to the complexity of the underlying detection problem. However, the number of features or parameters must also be balanced since too many parameters can cause the model to overfit. As such, the the number of convolutional features and layers were no longer increased once the model started overfitting.

The model that proved to the most effective from these evaluations is illustrated in Fig. 6. It consists of five convolutional blocks followed by three pooling blocks. The convolutional blocks each consist of a two-dimensional convolutional layer, a two-dimensional maximum pooling layer, a batch normalisation layer, and an occasional dropout layer. The convolutional layers all use a 3x3 kernel and an ReLU activation function, and progressively increase in their number of convolutional

features. For example, the first convolutional layer has 32 convolutional features, the second has 128 features, and third and fourth have 256 and 512, respectively. Moreover, while the pooling blocks also each contain a convolutional layer, their primary purpose is to gradually decrease the feature map from the input 512 by 512 size to the 4 by 4 dimensions of the output feature map.

Since the object detection model was custom, a customized loss function also had to be defined to guide the training of the model. This loss function is the summation of three individual loss functions, one for each type of data predicted within the 11 attribute vector. The loss functions for the detection confidence and the one hot encoding of the object class are both evaluated using binary cross entropy. This is because both the classification and class encoding are defined as being either 1 or 0 in the encoding of the dataset. The confidence of the prediction should be 1 if there is a detected object in the given region of the feature map and 0 otherwise. Likewise, the one-hot encoding for a class is 1 if the object belongs to the class and 0 otherwise. Moreover, the mean-squared error was used as the loss function for the bounding box detections.

Given the 4 by 4 feature map of the output layer and the 11 attributes to be predicted per feature. The complete shape of the output of the output layer was 4x4x11. Since, however, the the calculation of loss requires the predicted value of this output to be of the same format and structure as the true output value, the input dataset had to be reformatted. More specifically, each detected object within a given image had to be encoded into a 4x4x11 tensor. To do this, the bounding box information was first mapped from their pixel values to one of the 16 feature values. This was achieved by performing integer division to essentially segment the input image into a grid a 4x4 grid of blocks. The block that the top left corner of the bounding box (x_{min}, y_{min}) fell into was feature that the defined object was encoded within. In addition to this mapping, the annotation data for each detection were also normalized to prevent the biasing of the network toward any particular configuration. This included normalizing the four values specifying the bounding boxes, x_{min}, y_{min}, box width, and box height.

Finally, after all of these steps were carried out, the model was trained. The model was trained for 2000 epochs total using an early stopping callback with 400 epoch patience and a stopping threshold of 0.005 reduction in validation loss. The model ended up stopping early at the 543 epoch after no improvement in validation loss was experienced after 400 straight epochs.

E. Navigation

The navigation component of the system was realized using the turtle bot3_navigation package in ROS. This package provides a number of key functionalities needed to perform autonomous navigation with ROS. One of these essential functionalities is path planning. Given a map of the robot's environment, a goal position, and a continuous reading of the robot's current position, the navigation package can be used to calculate, continuously update, and execute a path from a robot's starting position to the goal position. As such, this package was used in an attempt to get the robot to move to the positions of previously detected objects. As mentioned in section III, the mapped positions of previously detected objects were published to the "simple_move_base
goal" topic to instruct the robot of the positions objects positions to be traversed to.

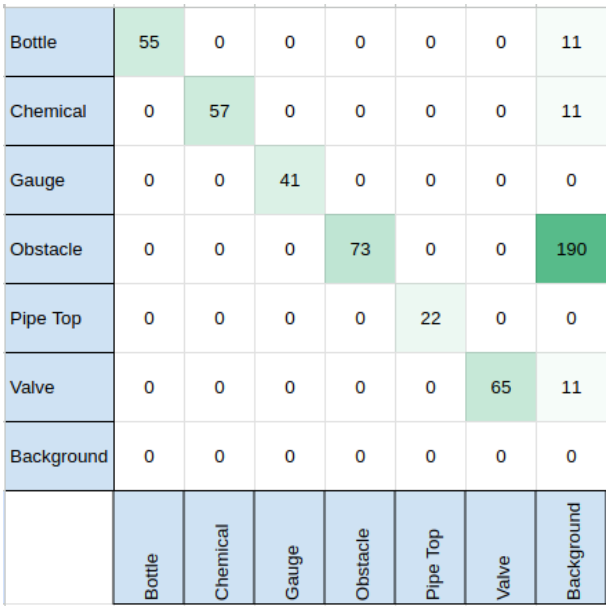

Fig. 7: Confusion Matrix.

V. Results

A. YOLOv5

1) sectionConfusion matrix, Mean Absolute Percentage, and Recall: This section seeks to outline the performance of the YOLOv5 model evaluated against the testing set. For a classification machine leaning model, parameters such as mean absolute error, mean absolute percentage error, recall, are often used in the evaluation of performance. Due to object detection model's localization component, parameters, such as class loss, box loss, objectness loss, mean average precision and recall were instead used for model evaluation. The model managed to produce remarkable mean average precision and recall scores which were 99.2 percent and 99.6 percent, respectively.

Despite the desirable results , the model struggled to correctly distinguish between background and obstacle classes. The main reason for this could possibly be because both classes appeared to be similar in terms of color and in most cases, they were so close to each other during collecting of the data-set. Also, the fact that they were both wooden material could also be a contributing factor of this confusion.

2) Model Training Results: Fig. 8 clearly illustrates the training the results of the model. During training of an object detection model multiple parameters could be tracked during training but for the trained model, the following parameters were used to evaluate the model: box loss, objectness loss, class loss, precision, and recall.

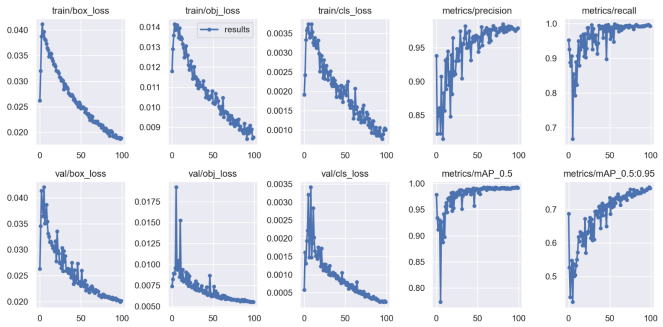

Fig. 8: Model training results.

The box loss parameter is a measure of how accurately the model is able to locate the centre of an object and how well it is able to surround the detected object with a bounding box. This could be summarized as the probability of an object lying within the proposed bounding box. Looking at the graph for this parameter, it is evident that the loss decreases gradually making the model able to accurately predict the position of an object within the space. This occurred for both training and validation box loss but it is evident that the model struggled a little bit on the validation since there were too many fluctuations, occasionally, the box loss went too high but towards the last moments of training, at around epoch 80, it began to be stable. On average, the box was pretty descent since it was able to reduced significantly towards the end.

The next parameter is objectness loss. This parameter penalizes the model based on the presence or absence of the object within a specified bounding box. The object loss is computed using binary cross entropy. Looking at the results, the training objectness loss converged at a low value of around 0.008. Interestingly, despite having large fluctuations initially, the objectness loss of the validation loss converged at a lower value, namely 0.50.

The classification loss, classification loss, seeks to penalize the model based on the wrong predictions assigned to an object. Looking at the graphs, the training classification loss graph had quite some fluctuations as well as the validation graph especially at the start. Right after the convergence of the model, the graph was quite stable.

The precision parameter measures the ability of the model to correctly predict correct labels in a given prediction. Its essentially a measure of true positives given all true positive values divided by a sum of true positive values and false negative values. Looking at the precision graphs, it is evident that the model struggled a little bit with getting correct labels. The fluctuations on the graph were to many giving an impression of a not very accurate precision. On the other hand, recall, also known as true positive rate. Just as in precision, the same applies to the recall graphs, the fluctuations in the graph were too many thus not so accurate but gets stable towards the last epochs on the model training.

3) Customized TensorFlow Object Detection Model: During training, the custom TensorFlow Object Detection Model reached a minimum training loss of 0.0184 and a minimum validation loss of 43. The large discrepancy in the scale of these values suggests that the model still over-fitted. Despite this, the model was evaluated against the testing set to better evaluate its accuracy. The confusion matrix for the predictions made against the testing set are shown in Fig. 9. The results indicate that the model is considerably accurate with a 86.5% accuracy rate in its classifications.

		Predicted Classs					
		Bottle	Chemical	Gauge	Obstacle	Pipe Top	Valve
True Class	Bottle	19	1	0	0	0	1
	Chemical	4	18	0	0	0	0
	Gauge	1	0	24	0	0	0
	Obstacle	1	2	4	29	0	2
	Pipe Top	0	0	0	1	14	0
	Valve	0	0	1	1	0	24

Fig. 9: Confusion Matrix of Custom TensorFlow Model

Despite this, the models' annotations of the testing images revealed its localization of the objects was inaccurate. Specifically, the bounding boxes of the detected objects were frequently misaligned or off the object entirely.

VI. Discussion

A. Hindrances

Besides YOLO, there were attempts to train multiple object detection model as discussed above in the methodology. However, some of them were not successful for various reasons. A good example was FasterRCNN. FasterRCNN was the first method proposed for performing the object detection component discussed in this paper. Transfer learning using FasterRCNN initially sounded like a good idea but it was not ideal given the goal we wanted to achieve. Ideally, during the planning step, the goal was to train a custom classification model and use the resulting weights of the model as backbone of the FasterRCNN. This was not possible as Faster RCNN only takes in resnet101, vgg-torch, vgg-16, vgg-19 as back bone models. As such, it was clear that the requirements of the object detection system could not be easily met with a FasterRCNN model, and the solution was not pursued further as attention was shifted to more promising results.

There were also unforeseen effects of the robot's design that hindered the complete development of this system. For instance, since the robot was custom made, many of the components had to be soldered or 3D printed. Moreover, there were some unforeseen effects of the robot's design that troubled the system. As mentioned previously for example, the robot replaced the Raspberry Pi 4 found in ROBOTIS' TurtleBot3 WafflePi with Dell Optiflex 3070 PC to provide greater computational capabilities. Using this larger form factor PC also required a larger battery to power it. Unfortunately, the larger battery was also taller than the provided vertical spacers. As such, larger vertical spacers had to be printed to fit the larger computer and its battery. Additionally, the Optiflex and its battery also through the weight distribution of the robot off. Specifically, the computer and the battery were positioned near the back of the robot. The back of the robot is supported by two rolling caster balls while front is driven by its two motors. Due to this, the loading of the weight to the back of

the robot caused many issues in the creation of the robot's map.

This is because Simultaneous Localization and Mapping (SLAM) uses both mapping and localization to map the robot's environment and determine where the robot lies within that environment. The localization component moreover, relies on the encoding of the robot's motors. Since the robot's weight was backloaded, the front wheels would often lose contact with the ground, causing the odometry of the robot to get thrown off, thereby ruining the map and the consequent ability to perform autonomous navigation.

Additionally, custom design of the robot made the navigation component of the system perform very poorly. More particularly, when navigation goals were set by publishing object positions to the "simple_move_base
goal" topic, a path connecting the robot's initial position to that goal position would be properly generated, but the execution of that path was extremely poor. As such, the system's navigation requirement of getting the robot to autonomously traverse to the object to go to it to manipulate it could not be met.

B. Analysis of Results

While the custom TensorFlow object detection model showed considerable accuracy in its classification of detected objects, its estimation of bounding box locations for those objects was very inaccurate. This can likely be attributed to the small size of the feature map. Since each bounding box in the dataset is being mapped to a feature in the feature map, decreasing the size of the feature map results in greater information loss when the bounding boxes are reconstructed from the feature maps. Despite this, the methods used to derive this model revealed that too large of a feature map resulted in under-fitting. As such, future investigations could be conducted to find the balance between these two factors. The size of the feature map could be increased just enough to improve the localization of bounding boxes while preventing the model under-fitting issue.

Additionally, mean-squared error (MSE) was used as the loss function for the prediction of detection boxes. However, limitations arise when MSE gradients are calculated with MSE as the loss function and are well-documented. As such, future investigations could also delved into replacing MSE with other loss functions such as Huber loss to address the bounding box issue. The formatting of the object detections into an 11 attribute vector itself may, itself, be a limitation on the level of accuracy that can be achieved. As such, future work could also investigate the use of different methods for prediction of object annotation information. Overall, the accuracy of bounding box localization is very important for this proposed system since the bounding box is used to estimate the object's centroid. As such, if this component of the custom object detection model can not be imrpove On the other hand, the results demonstrated that the trained YOLOv5 model had considerably high accuracy in both the classification of detected objects and the localization of their bounding boxes. Moreover, the YOLO model could easily be expanded to an increased number of objects and offers real-time prediction

Fig. 10: Confidence level during testing.

capabilities. Due to these factors, the further development of this system should integrate YOLOv5 for its object detection requirements.

VII. Conclusion

Motion planning is quite an imperative part of robotics as the ask is always about how to move efficiently. Some of the approaches to efficient motion planning mentioned earlier include: neural networks, reinforcement learning, computer vision, inverse kinematics just to mention a few. This paper went over the implementation of motion planning with a huge chuck going into computer vision strategies as well as reporting the results of various tried out strategies. Among the many strategies that were tried and proved to yield desirable results were, state of the art object detection, the YOLO model and a TensorFlow object detection model built from scratch. Some of the challenges presented by this models were as follows. The YOLO model offered quality object detection with a high accuracy of 99.2 percent but had a lower level of confidence values averaging at around 88 percent. On the other hand, the TensorFlow object detection model built from scratch was unable to accurately fit the bounding boxes round the object of interest, rather, for the most part, it would fit a bounding box round a certain section of the image so thing which would disadvantage an application if deployed in a real world scenario. Other strategies that were mentioned earlier but were not seen through to the end, included FasterRCNN due to the inability to specify a custom backbone mode. Also, issues with the ROS navigation API were presented as being unreliable and in efficient as path planning to goal state using this strategy took forever.

In future, integration of all features mentioned in III would be priority to achieve 100 percent success on this project.

Acknowledgment

Thanks to Dr. Arvind Bansal for the support and guidance in Advance Artificial Intelligence and to Dr. Kim from ATR Lab for providing us with the necessary material to make the project a success.

References

[1] A. Abdi, M. Ranjbar, and J. H. Park, "Computer vision-based path planning for robot arms in three-dimensional workspaces using q-learning and neural networks," *Sensors*, vol. 22, 02 2022.

[2] l. Tian, N. Thalmann, D. Thalmann, Z. Fang, and J. Zheng, *Object Grasping of Humanoid Robot Based on YOLO*, 06 2019, pp. 476–482.
[3] A. Shahzad, X. Gao, A. Yasin, K. Javed, and S. Anwar, "A vision-based path planning and object tracking framework for 6-dof robotic manipulator," *IEEE Access*, 03 2021.
[4] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," 2019.
[5] A. Shahzad, X. Gao, A. Yasin, K. Javed, and S. M. Anwar, "A vision-based path planning and object tracking framework for 6-dof robotic manipulator," *IEEE Access*, vol. 8, pp. 203 158–203 167, 2020.
[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.