“Robust Computer Vision for Obstacle Detection and Situation Awareness for Kuka Robot”

Masters in Information Technology
Autonomous Intelligent System WS 2019/2020
Harshal Vaze - 1269879 vaze@stud.fra-uas.de
Abhishek Sharma - 1274291 asharma@stud.fra-uas.de

Abstract- **This paper presents computer vision’s task of object detection algorithm to identify the objects captured by the robot’s camera. A detailed solution and analysis are also part of this paper.**

Obstacle or object detection is one of the challenging tasks in computer vision. Many research is going on in the field of object detection for better accuracy and precision. In recent years, Convolutional neural network (CNN) has become more popular for object detection and image processing because of its great ability of feature extraction and transfer learning. OpenCV also has its own set of computer vision libraries for image processing and object detection. In this paper, we have presented a project in which OpenCV libraries and CNN both work together for object detection. Object detection is divided into two parts, the first part is Image segmentation and second part is Image classification. Image segmentation is done by OpenCV functions and Image classification is done by CNN. The result shows that this approach proved to be successful in detecting three classes of the chosen object along with its object type as static or dynamic.

Keywords- CNN, ROS, OpenCV, Object detection, Situation awareness, Deep learning, Image processing, Object recognition, Robot, Automation, Autonomous intelligent system.

I. INTRODUCTION

This complete project is a part of Automation Lab and Autonomous intelligent system modules of Frankfurt University of Applied Sciences. This project is proposed to solve the computer vision problem of obstacle detection for the Kuka robot. Also it efficiently responsible for deciding the type of object as static and dynamic.

Today’s world is becoming more and more technology-based and machines are replacing Human beings in most of the jobs. Humans have a tremendous capability to recognize and detect any kind of object in seconds. This recognition skills of humans are developed over a period of time after watching every object many times. Similarly, Machines can also be trained. If a computer can able to learn the feature of an object such as edges, curves, etc. by looking at it over and over again, then this concept is nothing but passing the feature of images in a series of neural networks called a convolutional neural network. To get an accurate result from the machines, we need to build and train them accordingly to be very useful. In this project, we have tried to build a program which can guide the Robot to detect the object accurately and act accordingly.

Object detection is done in two main parts. The first part is to prepare the dataset and build Convolutional Neural Network (CNN) model. In machine learning, preparing the dataset is a crucial part. For better training, a large number of data is required. In this project, we have purposefully chosen a three common objects i.e. Apple, Scientific Calculator and a Computer Mouse since these are objects more frequently observed on the desk. We used a Kinect Xbox One camera for capturing the images. A huge number of pictures of these objects were taken from different angles and distances from the Kinect camera. After capturing the raw images they were modified and 1800 images of each class were created for training of CNN model.

Once the model was trained, second part is of testing the model we used ROS bag files. It was performed in two steps, the first step is the image segmentation part. Image segmentation is a fundamental step to extract data from the image. In our project, we have focused on an image segmentation technique called contour. Contours are the continuous curve along the boundary of the object in an image. We made a square contour around the object using a series of OpenCV functions such as grayscale conversion, thresholding, noise removal, watershed algorithm.

Details are discussed under program implementation.

The second step is Image classification using a CNN. The Main aim of CNN is to accept the input image and define its class. Here in our project, the class of object is Apple, Scientific Calculator and Computer Mouse. CNN model has a three-layer architecture that contains an input layer, hidden layers, and output layer. The input layer takes the image, output layer gives the trained output and the hidden layer is a interconnect series of convolution and max-pooling layers which is responsible for the actual training of data. In our project, the output of image segmentation is given as input to CNN for identifying the class of objects.

This project also has a situation awareness part of the robot. In this section, we are calculating the distance of an object from the camera. Also, we are defining object type as static or dynamic for each of three objects. All the outputs are saved in an Excel sheet with detected object list with its other characteristics.

II. PARAMETERS OF PROJECT

A. OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. These OpenCV libraries are used to solve various computer vision problems and implemented in various computer vision applications such as Image Segmentation, Image Filtering, Image Transformation, Object Tracking, Feature detection, etc.

OpenCV has a modular structure, which means that the package includes several shared or static libraries. The following modules are available:

- Core functionality (core) - a compact module defining basic data structures, including the dense multi-dimensional array Mat and basic functions used by all other modules.
- Image Processing (imgproc) - an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), colour space conversion, histograms, and so on.
- Video Analysis (video) - a video analysis module that includes motion estimation, background subtraction, and object tracking algorithms.
- Camera Calibration and 3D Reconstruction (calib3d) - basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction.
- 2D Features Framework (features2d) - salient feature detectors, descriptors, and descriptor matchers.
- Object Detection (objdetect) - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).
- Some other modules, such as High-level GUI, Video I/O, Python bindings, and others.

In our project, we have used OpenCV for two main operations; Image segmentation and Distance calculation. The modules used for this were Image Processing (imgproc) and Camera Calibration and 3D Reconstruction (calib3d). Image processing helps the computer to understand the content of the image. The image could be anything such as photographs, video frames (collection of images) [1].

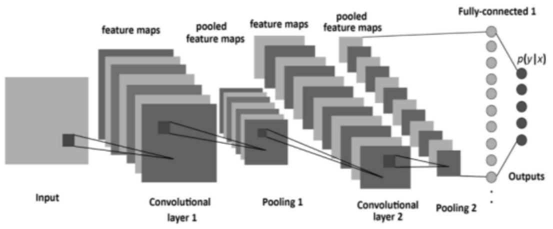
B. Convolutional Neural Network (CNN)

A neural network is a system of interconnected artificial “neurons” that exchange messages between each other. The connections have numeric weights that are tuned during the training process, so that a properly trained network will respond correctly when presented with an image or pattern to recognize. The network consists of multiple layers of feature-detecting “neurons”. Each layer has many neurons that respond to different combinations of inputs from the previous layers. As shown in [Figure 1.0], the layers are built up so that the first layer detects a set of primitive patterns in the input, the second layer detects patterns of patterns, the third layer detects patterns of those patterns, and so on. CNN is one of the variant of deep neural network used to solve computer vision and image processing problems such as image recognition, image classification, object detection, face detection, etc. [2]. Training is performed using a “labelled” dataset of inputs in a wide assortment of representative input patterns that are tagged with their intended output response. Training uses general-purpose methods to iteratively determine the weights for intermediate and final feature neurons.

A convolutional neural network (CNN or ConvNet) is one of the most popular algorithms for deep learning, a variant of machine learning in which a model learns to perform classification tasks directly from images, video data, texts or acoustic data. CNNs are especially useful for finding patterns in pictures and thus recognizing objects, faces and scenes. CNNs learn directly from image data. They use patterns to classify images and make manual extraction of features unnecessary. CNNs is kind of model whose capacity can be controlled by varying their depth and breadth and also it make more correct assumptions with good accuracy about the nature of image (pixel, colour, static or dynamic). Thus, compared to standard feed forward neural network with similar-sized layer, CNNs have much fewer correction and they are easy to train [3].

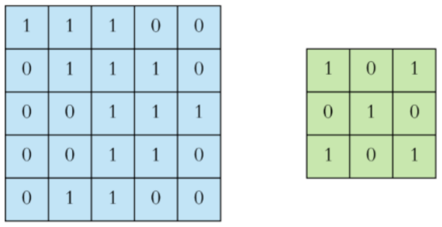
ARCHITECTURE OF CNN

CNN have multiple layers input layer, some hidden convolution layers, pooling layers and output layers.



1. **Input Layer**: The input layer contains the image data (pixel value). Image data is stored in the form of a matrix and transforms into the next convolution layer for feature extraction.
2. **Convolution Layer**: Convolution is a mathematical operation applied to input data (image data) using a convolution filter to produce a feature map. Convolution filter is also called Kernel.

Taking an arbitrary example to explain convolution. Consider a 3x3 kernel matrix, therefore, convolution is also called 3x3 convolution due to the shape of the kernel filter as shown below [4].



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

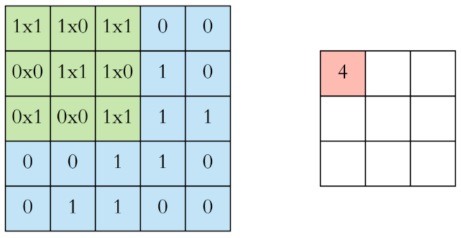
Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

On the left side is the input image and on the right is the convolution filter (Kernel)

The convolution operation is performed by sliding the filter over the input image. Then, element-wise multiplication is done at every location and after that sum of the result goes into feature map as shown in below figure.



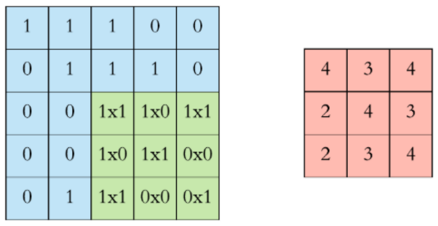
1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4		

Feature Map

Same process is continued to whole input image and aggregated convolution result in feature map is shown below.



1	1	1	0	0
0	1	1	1	0
0	0	1x1	1x0	1x1
0	0	1x0	1x1	0x0
0	1	1x1	0x0	0x1

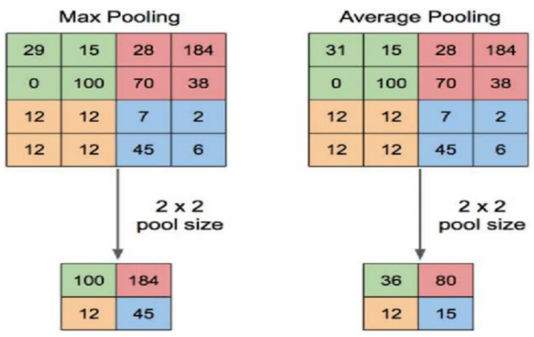
4	3	4
2	4	3
2	3	4

1. ***Pooling Layer:***

Pooling is done after convolution operation to reduce the dimension of the feature map. Pooling layer down sample each feature map independently, reducing the height and width, but keeping the depth intact.

There are two types of pooling Max pooling and Average pooling.

Max Pooling works by returning the maximum value from the kernel-covered portion of the image. Similarly Average Pooling works by returning the average value from the kernel-covered portion of the image. Pooling does not have any parameter but it need a window size and stride. Below figure shows max and average pooling with window size 2x2 and stride 2.



Max Pooling

29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

100	184
12	45

Average Pooling

31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

36	80
12	15

2. *Non-Linearity (ReLU):*

ReLU stands for Rectified Linear Unit for a non-linear operation. The output of ReLU is given by,

y = max (0, x). It is used as activation function to ensure size of input and output are same. ReLU activation function is much faster than other activation function such as tanh and sigmoid. Faster learning has a great influence on the performance of large models trained on large datasets [5].

$$RELU(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$

3. *Fully Connected Layer:*

Once the features extracted by the convolution layers and down sampled by the pooling layers are created, they are mapped by a subset of fully connected layers to the final outputs of the network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes [6]. Finally, activation function softmax is used to classify the output.

Hyperparameter of CNN:

Hyperparameter is set before the actual beginning of learning process. Hyperparameters can directly affect the training of algorithms for machine learning. It is therefore important to understand how to optimize them in order to accomplish maximum performance [7].

Hyperparameter can be divided into two types:

a) Hyperparameter that determines the network structure such as:

- Kernel Size - the size of the filter.
- Kernel Type - values of the actual filter (e.g., edge detection, sharpen).
- Stride - the rate at which the kernel pass over the input image.
- Padding - add layers of 0s to make sure the kernel pass over the edge of the image.
- Hidden layer - layers between input and output layers.
- Activation functions - allow the model to learn nonlinear prediction boundaries.

b) Hyperparameter that determines the network trained such as:

- Learning rate - regulates on the update of the weight at the end of each batch.
- Momentum - regulates the value to let the previous update influence the current weight update.
- A number of epochs - the iterations of the entire training dataset to the network during training.
- Batch size - the number of patterns shown to the network before the weights are updated.

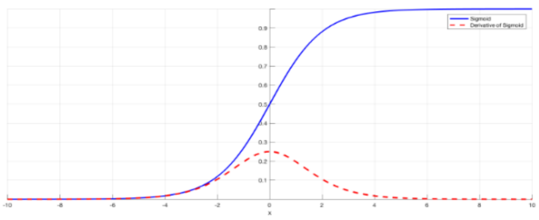
Advantage of activation function ReLU over other functions:

Activation function are Non-linear function. Non-linear functions have degree more than one and have curvature shape in their graph. The role of the activation function is to generate non-linear mappings from inputs to outputs. Activation function allow network to learn something complex and complicated from data and also represent non-linear complex arbitrary functional mapping between inputs and outputs. Activation function must be differentiable in order to perform back propagation algorithm to calculate gradient of error (Loss) with respect to weight and accordingly reduce error [8].

Popular activation function:

1. *Sigmoid:*
   Sigmoid [σ(x)] = $\frac{1}{1+e^{-x}}$

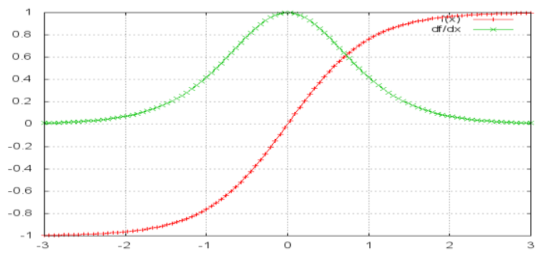
Sigmoid function maps value from 0 to 1 and it derivates maps value from 0 to 0.25 as shown in below graph.



Problem: It has vanishing gradient. Also it has slow convergence rate therefore, difficult to update gradient.

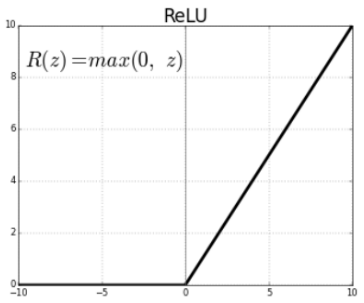
2. *Hyperbolic tangent function:*
   Tanh (x) = $\frac{e^{x}-e^{x}}{e^{x}+e^{x}}$

tanh function maps value from -1 to +1 and its derivative maps value from 0 to 1 as shown in below graph.



Problem: It also has vanishing gradient problem.

3. *ReLU (Rectified Linear Unit):*
   F(x) = max (o,x)
   If, x < 0 then, f(x) = 0 or If x > 0 then, f(x) = x.
   It maps value from 0 to infinite.



ReLU function and its graph is both Monotonic.

Benefit: It avoids and rectifies the vanishing gradient problem.

Limitation: It can only be used with in hidden layer of Neural Network.

C. *Creation of Dataset*

In this project we are detecting three objects namely, Apple, Scientific Calculator and Computer Mouse. The reason behind choosing these objects was that these objects are more frequently observed on the desk and since this project is part of navigation system of Kuka Robot, these objects will serve the complexity of the system. For creation of Dataset we used Kinect Xbox One Camera and pictures of these objects were captured from different angles and different distances from the camera. The raw images had a big size and train CNN model we required images of objects with clearly focused and of less size. The procedure we followed to scale the images down automatically was it performs segmentation using the Watershed transform in OpenCV over each original image and gets rectangles around every resulting contour. Then, it turns those rectangles into squares by taking the largest rectangle side, it enlarges them by 75 pixels in all 4 directions and keeps only the squares with sides bigger than 300 pixels. After that, only the square with its midpoint (the intersection of its two diagonals) closest to the centre of the image is kept. The image is cropped to that square, scaled down to 220x220 pixels and saved in a different folder that will contain the new images. Afterwards, bad images in the folder (too blurry or not capturing the whole object) are deleted one by one manually.

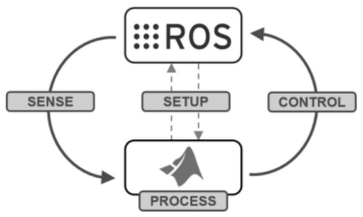
For data augmentation each image from that set were taken through following operations:

i. Flip it vertically
ii. Flip it horizontally
iii. Rotate it by 30 degrees
iv. Add Gaussian noise to it
v. Blur it

Images after each operation were saved, thus multiplying the number of images in the dataset by 6. This gave 1800 images for Scientific Calculators and same for the other two classes. Since for training the same number of images of each class should be used, we deleted images from the other classes until we got 1800 of each.

D. *ROS (Robotic Operating System)*

ROS, an open-source robot operating system. ROS is not an actual operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogeneous compute cluster.



Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages is to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

Some of the important files/directories inside Packages are:

i. *Nodes*: A node is a process that performs computation.
ii. *CMakeLists.txt*: It is the input to the CMake build system for building software packages.

iii. *Package.xml* : It defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
iv. *.yaml* files: To run a rosnode you may require a lot of parameters e.g, Kp, Ki, Kd parameters in *PID control*. We can configure these using YAML files.
v. *launch files:* To run multiple nodes at once in ROS we use launch files.

Any code that will be written should be in the form of packages. And the packages should be inside workspace. We used a catkin workspace.

A *catkin workspace* is a folder where one can modify, build, and install catkin packages. It can contain up to four different spaces which each serve a different role in the software development process.

i. The *source space* contains the source code of catkin packages. This is where you can extract/checkout/clone source code for the packages you want to build. Each folder within the *source space* contains one or more catkin packages.
ii. The *build space* is where CMake is invoked to build the catkin packages in the *source space*. CMake and catkin keep their cache information and other intermediate files here.
iii. The *development space* (or *devel space*) is where built targets are placed prior to being installed. The way targets are organized in the *devel space* is the same as their layout when they are installed. This provides a useful testing and development environment which does not require invoking the installation step.
iv. Once targets are built, they can be installed into the *install space* by invoking the install target, usually with make install.

When we run ROS nodes, they perform computations and obtain results. But they may require results from other nodes in order to perform some other functions. Hence we need a mechanism that can help us transfer data from one node to other. One of the first thing we need to do is to setup a ROS Master. [9]

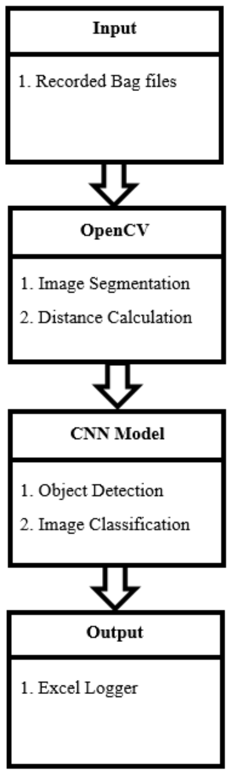
The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

The transfer of data takes place via topics. If one wants to send the data, they can publish it to topics and whomever needs it, can subscribe to it using publishers. This is helpful in writing code and even more helpful when we are using ROS bags. ROS bags are helpful when we want to record some data, so we can play it later and replicate a behaviour if we want to. [10]

While running the sensors, we may need to visualize the data. We use RViz for this, RViz is a 3D Visualization tool for ROS. It is one of the most popular tools for visualization. It takes in a topic as input and visualizes that based on the message type being published. It lets us see the environment from the perspective of the robot.

III. ARCHITECTURE

Project is divided into two main components, one is OpenCV and another is CNN model. The data flow while testing the project is as shown below.



A. *Input*

The input is given to the OpenCV as bag file recorded with the Kinect Xbox camera. Kinect is a single set-top unit that combines an RGB visible spectrum camera, an infrared (IR) spectrum 3D camera and a multiarray microphone. Bag is a file format in ROS for storing ROS message data. They are called bags because of their .bag extension. Bags have an important role in ROS and a variety of tools have been written to allow you to store, process, analyse, and visualize them.

Bags are typically created by a tool like rosbag, which contributes to one or more ROS topics, and store the sequential message data in a file as it is recorded. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics. Bags are the primary mechanism in ROS for data logging, which means that they have a variety of offline uses. Researchers have used the bag file tool chain to record datasets, then visualize, label them, and store them for future use.

B. OpenCV

The OpenCV carries out the image segmentation on bag file and it also calculates the distance of Object from the camera which is also stored in the bag file.

Image segmentation is a process by which we partition images into different regions. It helps us to identify the location of a single object in the given image. In case we have multiple objects present, we then rely on the concept of object detection. We can predict the location along with the class for each object using object detection. By dividing the image into segments, we can make use of the important segments for processing the image. An image is a collection or set of different pixels. We group together the pixels that have similar attributes using image segmentation. Object detection builds a bounding box corresponding to each class in the image. But it tells us nothing about the shape of the object. We only get the set of bounding box coordinates. Image segmentation creates a pixel-wise mask for each object in the image. This technique gives us a far more granular understanding of the objects in the image. The contours are the continuous lines or curves that bound or cover the full boundary of an object in an image. And, here we will use image segmentation technique called contours to extract the parts of an image. Also contours are very much important in Object detection and Shape analysis.

One simple way to segment different objects could be to use their pixel values. An important point to note is that pixel values will be different for the objects and the image's background if there's a sharp contrast between them. In this case, we can set a threshold value. The pixel values falling below or above that threshold can be classified accordingly as an object or the background. This technique is known as Threshold Segmentation.

- If we want to divide the image into two regions (object and background), we define a single threshold value. This is known as the global threshold.
- If we have multiple objects along with the background, we must define multiple thresholds. These thresholds are collectively known as the local threshold.

Bag file contains three-channel image (RGB), first we need to convert it into grayscale so that we only have a single channel. Next, we have to apply a certain threshold to this image, this threshold should separate the image into two parts - the foreground and the background. The height and width of the image is noted and we take the mean of the pixel values and use that as a threshold. If the pixel value is more than our threshold, we can say that it belongs to an object. If the pixel value is less than the threshold, it will be treated as the background. We can define multiple thresholds as well to detect multiple objects. We can set different threshold values and check how the segments are made. Some of the advantages of this method are:

- Calculations are simpler
- Fast operation speed
- When the object and background have high contrast, this method performs really well

But there are some limitations to this approach. When we don't have significant grayscale difference, or there is an overlap of the grayscale pixel values, it becomes very difficult to get accurate segments.

After thresholding we perform Morphological opening for noise removal. Morphological opening removes small objects from the foreground like bright pixels of an image, placing them in the background. This technique can also be used to find specific shapes in an image. Next we apply distance transform in order to obtain the derived representation of a binary image i.e. foreground, where the value of each pixel is replaced by its distance to the nearest background pixel.

Afterwards we perform watershed transform for segmentation in order to isolate objects in the image from the background. Then we create the bounding rectangles around the isolated objects to define them and get the Bit depth of the image, Bit depth is the amount of colour information contained in each pixel in an image. An image with a bit depth of 1 means each pixel in the image can either be black or white, as the pixel can only contain 2 values. After that we find the closest distance from each contour and form a bounding square around it.

Another task for OpenCV is to calculate the distance of the object from the camera. To accomplish this task we utilized the parameters of camera matrix which are acquired from the depth of the camera information.

Computer vision and image processing algorithms can be used to automatically determine the perceived width and height of the object in pixels and also get the information about the focal length. Then, in subsequent images we simply need to find our marker/object and utilize the computed focal length to determine the distance to the selected object from the camera in centimetres.

The final output message we get from the OpenCV is segmented image which is ready to be recognized by CNN model.

C. *CNN Model*

The two main objectives of the CNN Model are to detect the object and classify the image.

The input for the CNN Model we get from the OpenCV is segmented image with bounding boxes around the objects in the image. The CNN Model we created consist of three convolution 2D layers, each layer followed by a max pooling layer. These six layers are followed by the Flattening layer and then two Dense layers. The CNN takes tensors of shape image_height, image_width, color_channels. We did this by passing the argument input_shape to our first layer with image_height = 220, image_width = 220 and color_channels = 3 representing colour image. Max Pooling is an important layer in CNN for scaling down the results from the previous layer, it basically takes the resulting image from the layer (matrix) and turns it into a vector by concatenating all the rows together into the single node. After that Flattening layer is used, Flattening layer just takes the image from the previous layer (matrix) and turns it into a vector by concatenating all the rows together into a single one. And finally the Dense layers are used, Dense layers are layers as in the simplest case of a neural network, an MLP (multi-layer perceptron). Since the output is a 3x1 vector, the last layer should be of size 3. Three iterations of Convolution-Max pooling followed by Flattening layer and then Dense layers of 512 neurons and another one of 3 neurons (3 classes) for the output.

This model was trained to identify three different types of objects: Apple, Scientific Calculator and the Computer Mouse. It is able to detect the objects in an image, a video and in real-time through a Kinect camera. The accuracy of the Model is around 98% on the test data.

Object detection is a computer vision technique that works to identify and locate objects within an image or video. Precisely, object detection draws bounding boxes around these detected objects, which allow us to locate where required objects are in or how they move through a given scene. After drawing bounding boxes around each objects detected it labels the boxes with object names for which the model is trained. The model predicts where each object is and what label should be applied. In that way, object detection provides more information about an image.

Machine learning based object detection models normally have two parts; an encoder and a decoder. An encoder takes an image as input and runs it through a series of blocks and layers that learn to extract statistical features used to locate and label objects. Outputs from the encoder are then passed to a decoder, which predicts bounding boxes and labels for each object detected in an image.

The simplest decoder is a pure regressor. The regressor is connected to the output of the encoder and predicts the location and size of each bounding box directly. The output of the model is the X, Y coordinate pair for the object and its extent in the image. Though simple, this type of model is limited. The user needs to specify the number of boxes ahead of time. If the image has two objects, but the model was only trained to detect a single object, one object will go unlabelled.

An extension of the regressor approach is a region proposal network. In the decoder, the model proposes regions of an image where it believes an object might reside. The pixels belonging to these regions are then fed into a classification subnetwork to determine a label or reject the proposal. It then runs the pixels containing those regions through a classification network. The benefit of this method is that it is more accurate and flexible model that can propose arbitrary numbers of regions that may contain a bounding box. The added accuracy, though, comes at the cost of computational efficiency.

Object detectors output the location and label for each object, but to know how correctly the model is working for an object's location, the most commonly used metric is intersection-over-union. Given two bounding boxes, we compute the area of the intersection and divide by the area of the union. This value ranges from 0 which defines no interaction to 1 which terms perfectly overlapping.

Another task for CNN is of image classification. The main purpose of image classification is acceptance of the input image and the following definition of its class. Just like people learn this skill from their birth and are able to easily determine which object is present in the given image. But the computer realizes the pictures quite differently; instead of the image, the computer understands an array of pixels. For example, if image size is 220x220, in this case, the size of the array will be 220x220x3. Where 220 is width, next 220 is height and 3 is RGB channel values. The computer is assigned a value from 0 to 255 to each of these numbers which describes the intensity of the pixel at each point.

To resolve this problem the computer searches for the characteristics of the base level. In human understanding such characteristics are for example the size or colour of the object whereas for the computer, these characteristics are boundaries or curvatures. And then through the groups of convolutional layers the computer constructs more abstract concepts. The image is passed through a series of convolutional, pooling layers, flattening and dense layers and then generates the output.

The Convolution layer is always the first who accepts the image which is matrix with pixel values entered into it. The reading of the input matrix begins at the top left of image, next the software selects a smaller matrix there, which is called a filter or neuron. Then the filter produces convolution, i.e. moves along the input image. The filter's task is to multiply its values by the original pixel values. All these multiplications are summed up and one number is obtained in the end. Since the filter has read the image only in the upper left corner, it moves further and further right by one unit performing a similar operation. After passing the filter across all positions, a matrix is obtained which is smaller than an input matrix.

This operation, from a human perspective, is analogous to identifying boundaries and simple colours on the image. But in order to recognize the properties of a higher level such as the shape and size, the whole network is needed. The network will consist of several convolutional networks mixed with nonlinear and pooling layers. When the image passes through one convolution layer, the output of the first layer becomes the input for the second layer. And this happens with every further convolutional layer, as discussed earlier.

The nonlinear element is added in each convolution operation. It has an activation function, which brings nonlinear property. Without this property a network would not be sufficiently intense and will not be able to model the response variable as a class label.

The pooling layer follows after that, which works with width and height of the image and performs a down sampling operation on them. As a result the image volume is reduced which means that if some features like boundaries have already been identified in the previous convolution operation, than a detailed image is no longer needed for further processing and it is compressed to less detailed pictures. [11]

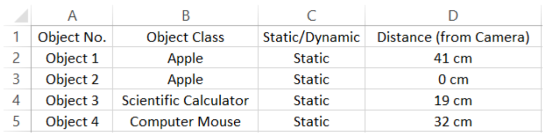
After completion of series of convolutional and pooling layers, it is necessary to attach a flattening layer. Flattening just takes the image from the previous layer in form of matrix and turns it into a vector by concatenating all the rows together into a single one. For image classification model, which requires these processed data needs to be good input to the model. It needs to be in the form of a 1-dimensional linear vector. Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which are called dense layers. In other words, we put all the pixel data in one line and make connections with the final layer. The Dense layers are layers as in the simplest case of a neural network, an MLP (multi-layer perceptron). Since the output is a 3x1 vector, the last layer should be of size 3. A dense layer is just a regular layer of neurons in a neural network. Each neuron receives input from all the neurons in the previous layer, thus densely connected. Dense layers of 512 neurons and another one of 3 neurons for three classes are used for the output.

D. *Output*

The final output we obtain requires the list of objects detected by our model with some of its characteristics. We presented this list as an Excel sheet which forms the table like contain with all its parameters. The parameters we assigned are as follows:

- Object Number
- Object Class
- Whether the object is Static or Dynamic
- Distance of the object from the Kinect camera

A different ROS node was created for creating the output in an excel format and it was named as excel_logger. While initialising this node, the name of the Excel file has to be passed with it. The parameters defined in this node are Object no. (number), Object Class, Static / Dynamic and Distance (from Camera). Every object detected in an image goes through all four of these parameters and values associated with that object are noted down in the Excel sheet. In case of video input, functionality of all parameters remains the same except for the Static or Dynamic parameter. If in the video, the object detected is non-stationary i.e. if it is moving then the object is classified as Dynamic object. This concept works on position of detected object in the image. If the detected object changes position from more than 5 cm in the next detection, then the object is classified as dynamic object. Following is an example of one Excel sheet output after testing the bag file containing objects listed below.



	A	B	C	D
1	Object No.	Object Class	Static/Dynamic	Distance (from Camera)
2	Object 1	Apple	Static	41 cm
3	Object 2	Apple	Static	0 cm
4	Object 3	Scientific Calculator	Static	19 cm
5	Object 4	Computer Mouse	Static	32 cm

IV. MATHEMATICS

Transformations within RGB (Red, Green, Blue colour spectrum) space like adding or removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB colour, as well as conversion to/from grayscale.

$$\text{RGB[A] to Gray:} \quad Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

and

$$\text{Gray to RGB[A]:} \quad R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow \max(ChannelRange)$$

Thresholding takes place on the grayscale image who has pixel values from 0 to 255. Here we are using Binary Inverted Thresholding, we are passing it pixel range values from 0 to 255 and it chooses one pixel value from depending on the overall pixel values of whole image.

THRESH_BINARY_INV
Python: cv.THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$$

where, src = Source 8-bit single-channel image.
dst = Destination image of the same size and the same type as src.
maxValue = Non-zero value assigned to the pixels for which the condition is satisfied
thresh = Threshold value

After processing this image gets some internal noise added. To remove this noise we use Morphological Opening process. Morphological Opening is just another name of erosion followed by dilation. The basic idea of erosion is it erodes away the boundaries of foreground object. The kernel slides through the image as in 2D convolution. A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero). Hence, all the pixels near boundary will be discarded depending upon the size of kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises, detach two connected objects.

In dilation, a pixel element is '1', if at least one pixel under the kernel is '1'. So it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won't come back, but our object area increases. It is also useful in joining broken parts of an object.

For bounding boxes; we require Aspect ratio and Extend parameters of the image.

Aspect Ratio is the ratio of width to height of bounding rectangle of the object.

$$Aspect\ Ratio = \frac{Width}{Height}$$

Extent is the ratio of contour area to bounding rectangle area.

$$Extent = \frac{Object\ Area}{Bounding\ Rectangle\ Area}$$

The functions in this section use a so-called pinhole camera model. In this model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where, (X,Y,Z) are the coordinates of a 3D point in the world coordinate space
(u,v) are the coordinates of the projection point in pixels
(c_x, c_y) is a principal point that is usually at the image centre
(f_x, f_y) are the focal lengths expressed in pixel units

Camera matrix is fiven by the following matrix whixh contains the details of the focal lengths and principal point.

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

To scale an image from the camera by a factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed. The joint rotation-translation matrix **[R|t]** is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of a still camera. That is, **[R|t]** translates coordinates of a point **(X,Y,Z)** to a coordinate system, fixed with respect to the camera. [1]

The input for the CNN Model we get from the OpenCV is segmented image with bounding boxes around the objects in the image.

For simplicity we assume a grayscale output image from image segmentation to be defined by a function,

$$I: \{1, \dots, n_1\} \times \{1, \dots, n_2\} \rightarrow \mathrm{W} \subseteq \mathbb{R}, (i, j) \mapsto I_{i,j}$$

such that the image ***I*** can be represented by an array of size n1xn2. Given the filter $\boldsymbol{K} \in \mathbb{R}^{2h1+1 \times 2h2+1}$, the discrete convolution of the image ***I*** with filter ***K*** is given by,

$$(I * K)_{r,s} := \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{u,v} \, I_{r+u,s+v}$$

where, the filter ***K*** is given by,

$$K = \begin{bmatrix} K_{-h_1,-h_2} & \cdots & K_{-h1,-h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,h_2} & \cdots & K_{h_1,h_2} \end{bmatrix}$$

Note that the behaviour of this operation towards the borders of the image needs to be defined properly. A commonly used filter for smoothing is the discrete Gaussian filter which is defined by,

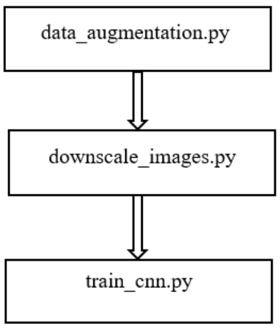
$$\left(K_{G(\sigma)}\right)_{r,s} = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(\frac{r^2 + s^2}{2\sigma^2}\right)$$

where, ***s*** is the standard deviation of the Gaussian distribution.

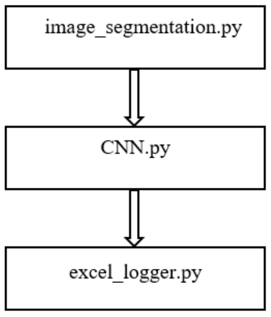
V. ALGORITHM

Explanation of algorithm used in this project is divided into two parts, first part contains creation of CNN model and second part contains testing the model.

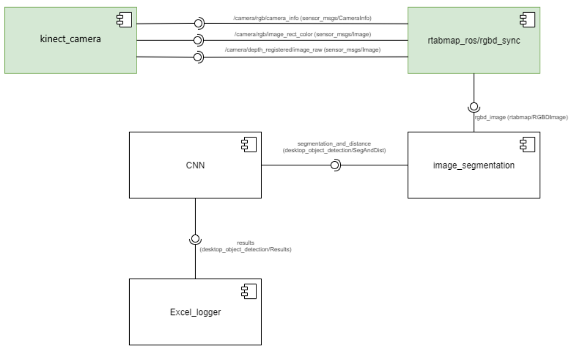
Before building the CNN model we first created the dataset for training of model. For creating dataset on which CNN model will be trained, we performed some editing on images of our objects. To train the model precisely, we tried to create such dataset of the objects which will train the model more accurately. We performed data augmentation and downscaled quality on the raw images. By performing these steps before training the model we believe that our CNN model was trained more precisely and faster. After the dataset was ready, we build the CNN model and trained the model on the created dataset.



Data flow diagram while creating dataset and training model



Data flow diagram while testing model


kinect_camera
/camera/rgb/camera_info (sensor_msgs/CameraInfo)
/camera/rgb/image_rect_color (sensor_msgs/Image)
/camera/depth_registered/image_raw (sensor_msgs/Image)
rtabmap_ros/rgbd_sync
rgbd_image (rtabmap/RGBDImage)
CNN
segmentation_and_distance (desktop_object_detection/SegAndDist)
image_segmentation
results (desktop_object_detection/Results)
Excel_logger

UML diagram of Architecture

A. *data_augmentation.py:*

In Machine learning, a large amount of data is required for model training because larger the data better the model will be trained. Instead of creating a new dataset (images in our case), we redesigned and recreated the data from the available data by performing some operations on the original data set.

Operations which we performed are:

- Vertically flipped image
- Horizontally flipped image
- Image rotation by 30 degree
- Add noise to the image
- Blur image

These sets of operations are available in the OpenCV library. We have taken an original image and then perform OpenCV method to recreate the larger number of dataset as shown below in code snippet.

```
# Iterate over all images in the given directory
for filename in os.listdir(directory + "/" + child_dir):
    # If file is a jpg or png image
    if filename.endswith(".jpg") or filename.endswith(".png"):
        # Get image from filename
        original = cv2.imread(directory + "/" + child_dir + "/" + filename, cv2.IMREAD_COLOR)
        orig_scikit = util.img_as_float(original)

        # Get vertically flipped image
        vflipped = cv2.flip(original, flipCode=0)

        # Get horizontally flipped image
        hflipped = cv2.flip(original, flipCode=0)

        # Get rotated images by 30 degrees
        rot30 = transform.rotate(orig_scikit, angle=30)

        # Add noise
        noisy = util.random_noise(orig_scikit)

        # Blur image
        blurry = cv2.GaussianBlur(orig_scikit, (5,5), 0)
```

B. *downscale_images.py:*

In this program, we have performed series of operations to downscale the quality of an image into 220x220 pixels. Again all the operations are performed using OpenCV functions and NumPy python library.

Grayscale conversion: It is necessary as it provides a single colour channel from three (RGB) colour channels and reduces the complexity of the image (pixels) for further processing.

```
# Get gray image
gray = cv2.cvtColor(original, cv2.COLOR_BGR2GRAY)
```

Thresholding: Thresholding is carried out to separate the image into two parts - the foreground and the background.

```
# Apply thresholding
ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

Here, we are getting two output first is ret, thresh. 'ret' gives the threshold value and 'thresh' gives the threshold image. Two thresholding types are used cv2.THRESH_BINARY_INV and cv2.THRESH_OTSU.

THRESH_BINARY_INV Python: cv.THRESH_BINARY_INV	$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$
THRESH_OTSU Python: cv.THRESH_OTSU	flag, use Otsu algorithm to choose the optimal threshold value

Morphological Opening: It is necessary to reduce the noise from the images. Here, kernel takes the matrix size 3 and image is convolved with kernel matrix.

```
# Morphological opening for noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)
```

Opening operation is erosion followed by dilation operation. It is used to remove the white noise.

In the previous step, noise is removed but at the same time our image is get shrink due to erosion. Now, dilation will increase the area of image.

```
# Find sure background
sure_bg = cv2.dilate(opening,kernel,iterations=3)
```

Distance transform to find foreground:

```
# Apply distance transform to find foreground
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
dist_transform = cv2.convertScaleAbs(dist_transform)
ret, sure_fg = cv2.threshold(dist_transform,0.4*dist_transform.max(),255,0)
```

Marker labelling:

```
# Marker labeling
ret, markers = cv2.connectedComponents(sure_fg)
markers = markers+1
markers[unknown==255] = 0
```

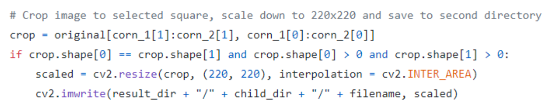
Watershed algorithm for segmentation:

```
# Watershed transform for segmentation
markers = cv2.watershed(original,markers)
```

Creating bounding rectangle around the objects in the image:

```
# Find bounding rectangles
markers1 = markers.astype(np.uint8)
ret, m2 = cv2.threshold(markers1, 0, 255, cv2.THRESH_BINARY|cv2.THRESH_OTSU)
_, contours, hierarchy = cv2.findContours(m2, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
contours_poly = [None]*len(contours)
boundRect = [None]*len(contours)
for i, c in enumerate(contours):
    contours_poly[i] = cv2.approxPolyDP(c, 3, True)
    boundRect[i] = cv2.boundingRect(contours_poly[i])
```

Scale down Image to 220x220 pixel:



```python
# Crop image to selected square, scale down to 220x220 and save to second directory
crop = original[corn_1[1]:corn_2[1], corn_1[0]:corn_2[0]]
if crop.shape[0] == crop.shape[1] and crop.shape[0] > 0 and crop.shape[1] > 0:
    scaled = cv2.resize(crop, (220, 220), interpolation = cv2.INTER_AREA)
    cv2.imwrite(result_dir + "/" + child_dir + "/" + filename, scaled)
```

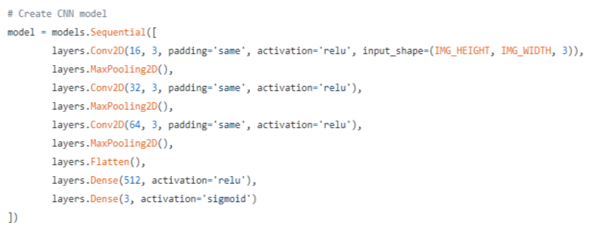
After performing these two steps, we generated our dataset. Images after each operation were saved, thus multiplying the number of images in the dataset by 6. This gave 1800 images for Scientific Calculators and same for the other two classes. Since for training the same number of images of each class should be used, we deleted images from the other classes until we got 1800 of each.

C. *train_cnn.py:*

In this file, we created the CNN model and trained our dataset. We have created CNN model of three 2D convolution layers with 16, 32, 64 filters, each of these layer is followed by the max pooling layer. After these six layers, one flattening and two dense layers are built to create the output.

Max Pooling is an important layer in CNN for scaling down the results from the previous layer, it basically takes the resulting image from the layer (matrix) and turns it into a vector by concatenating all the rows together into the single node. Flattening just takes the image from the previous layer (matrix) and turns it into a vector by concatenating all the rows together into a single one. The Dense layers are layers as in the simplest case of a neural network, an MLP (multi-layer perceptron). Since the output is a 3x1 vector, the last layer should be of size 3.

Three iterations of convolution-max pooling followed by Flattening and Dense layers of 512 neurons and another one of 3 neurons (3 classes) for the output.



```python
# Create CNN model
model = models.Sequential([
        layers.Conv2D(16, 3, padding='same', activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
        layers.MaxPooling2D(),
        layers.Conv2D(32, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.Dense(3, activation='sigmoid')
])
```

We trained our CNN model for around six hours. After the model is trained, it is saved with the weights and is ready to detect the objects with which the model was trained. [12]

After the model is ready to detect the objects, we created three ROS nodes; namely, image_segmentation, CNN and excel_logger for testing the model and creating a output of object list.

D. *image_segmentation.py:*

While testing we give input as the ROS bag files which contains the video of the objects with the distance information from the camera. These bag files undergo some similar prerequisites which were carried out for the creation of dataset while training the CNN model.

Instead of process entire image, we will divide the image into segments, and then that segments are used for further processing of image. Output of image segmentation will create a bounding box around the object present in the image. Image segmentation only create a pixel-wise mask for each object present in the image without telling the shape of the object.

Main python library we used here is Scikit-image, it is an open-source image processing library for the Python programming language which includes algorithms for segmentation, geometric transformations, colour space manipulation, analysis, filtering, morphology, feature detection and many more.

In the below code, we convert the RGB colour image into gray scale image, which will reduce the complexity of image by converting the image into single channel from three colour channels. After colour conversion next step is thresholding, which takes three arguments i.e. gray scale image, pixel range from 0 to 255 and thresholding type. Two types of thresholding i.e. THRESH_BINARY_INV and THRESH_OTSU are combined together which choose optimal value of threshold within the range of 0 to 255. After thresholding, the image we receive contains white noise. So to reduce the effect of noise, we have to do the noise removal by Morphological Opening, which is nothing but erosion followed dilation. Afterwards, we have to calculate the foreground and background of an image by distance transform and dilate functions of OpenCV respectively. Now we have found the unknown region of image by subtracting foreground with background which give a border region of an image. Next step is marker labelling. Once we know the foreground and background region we will label it with positive integer starting with 1 and the area which we don’t know will be label as 0. For this cv2.connectedComponents() is used. It marks the unknown region with 0 and known region with 1. Since the marker is ready now we can apply watershed algorithm for segmentation.

```
# Convert image to grayscale
gray = cv2.cvtColor(cv_rgb_image,cv2.COLOR_BGR2GRAY)

# Apply thresholding
ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

# Morphological opening for noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)

# Find sure background
sure_bg = cv2.dilate(opening,kernel,iterations=3)

# Apply distance transform to find foreground
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
dist_transform = cv2.convertScaleAbs(dist_transform)
ret, sure_fg = cv2.threshold(dist_transform,0.4*dist_transform.max(),255,0)


# Find unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)

# Marker labeling
ret, markers = cv2.connectedComponents(sure_fg)
markers = markers+1
markers[unknown==255] = 0

# Watershed transform for segmentation
markers = cv2.watershed(cv_rgb_image,markers)
```

Now, we will convert our image segmentation into bounding box as shown in below code. For bounding box, we need to find contours. Contours are nothing but a curve joining along the boundary having same color or intensity.

In OpenCV, cv2.findContours is the function to find contour which take three arguments i.e. source image (m2), contour retrieval mode (cv2.RETR_TREE), contour approximation method (CHAIN_APPROX_SIMPLE). Its output will be image, contour and hierarchy.

For every found contour, we now apply approximation to polygons with accuracy +-3 and stating that the curve must be closed. After that we find a bounding rectangle for every polygon and save it to boundRect [1].

```
# Find bounding rectangles
markers1 = markers.astype(np.uint8)
ret, m2 = cv2.threshold(markers1, 0, 255, cv2.THRESH_BINARY|cv2.THRESH_OTSU)
_, contours, hierarchy = cv2.findContours(m2, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
contours_poly = [None]*len(contours)
boundRect = [None]*len(contours)
for i, c in enumerate(contours):
    contours_poly[i] = cv2.approxPolyDP(c, 3, True)
    boundRect[i] = cv2.boundingRect(contours_poly[i])
```

Distance calculation from camera:

Below code will calculate the distance of the image from the camera. First we calculate image depth and then we need a necessary parameter from camera matrix to compute the distance. Camera parameter include image width and height from which we will calculate the focal length. Once we get the focal length we can calculate the distance in cm (centimeters).

```
# Get depth image
cv_depth_image = self.bridge.imgmsg_to_cv2(data.depth, desired_encoding="passthrough")

# Get parameters from camera matrix
camera_matrix = np.array(data.depthCameraInfo.K).reshape([3, 3])
cx = camera_matrix[0, 2]
cy = camera_matrix[1, 2]
fx_inv = 1.0 / camera_matrix[0, 0]
fy_inv = 1.0 / camera_matrix[1, 1]

# Get distance of each point in matrix in cm
dist = np.zeros((cv_depth_image.shape[0], cv_depth_image.shape[1]), np.uint8)
for i in range(cv_depth_image.shape[0]):
    for j in range(cv_depth_image.shape[1]):
        z = cv_depth_image[i, j] / 10.0
        x = z * ((i - cx) * fx_inv)
        y = z * ((j - cy) * fy_inv)
        dist[i, j] = sqrt(x*x + y*y + z*z)
```

E. CNN.py:

In this section, we will do image classification of three chosen object i.e. Scientific Calculator, Apple and Computer Mouse. A separate ROS node is created for Convolutional Neural Network for image classification. Main python library used are NumPy, Matplotlib, Tensorflow and Keras.

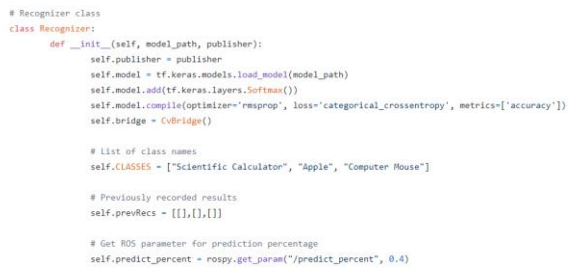
NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+.

Keras is a deep learning framework for Python that provides a convenient way to define and train almost any kind of deep learning model. Keras is a high-level neural networks API, written in Python which is capable of running on top of Tensorflow, Theano and CNTK. It was developed for enabling fast experimentation.

1. Initialization:

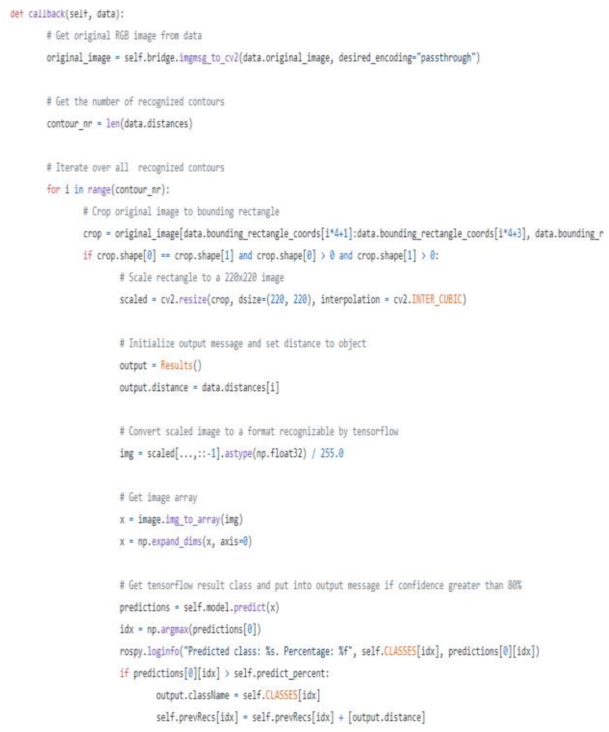
Initialization and declaration is the first step before detection. We need to initialize our training model and add that model to output softmax function. Then, CvBridge() method is used to provide an interface between ROS and OpenCV. CvBridge is a ROS library which is required to convert ROS image into OpenCV format. After that, we have declared the class name of object which we are using in our project i.e. Scientific Calculator, Apple and Computer Mouse.



```
# Recognizer class
class Recognizer:
        def __init__(self, model_path, publisher):
                self.publisher = publisher
                self.model = tf.keras.models.load_model(model_path)
                self.model.add(tf.keras.layers.Softmax())
                self.model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
                self.bridge = CvBridge()

                # List of class names
                self.CLASSES = ["Scientific Calculator", "Apple", "Computer Mouse"]

                # Previously recorded results
                self.prevRecs = [[],[],[]]

                # Get ROS parameter for prediction percentage
                self.predict_percent = rospy.get_param("/predict_percent", 0.4)
```

2. Detection:

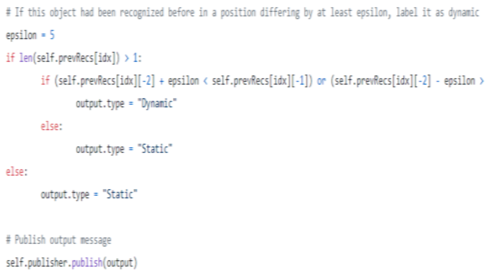
First we will take the original RGB image from the data and then calculate the number of recognized contours. After that we will crop the original image to make a bounding rectangle. Now, we resize the rectangle to 220x220 pixel by using CV2.resize. Next, we have initialized the output to capture result as well as output distance. Now we will do the detection of objects in image. For that, first we need to convert our bounding rectangle image to a format recognized by tensorflow. After that, we have created an array to store that image. Now our image is ready for detection, we will pass that image to train model to detect the class of object. It will detect the object in percentage.



```
def callback(self, data):
        # Get original RGB image from data
        original_image = self.bridge.imgmsg_to_cv2(data.original_image, desired_encoding="passthrough")

        # Get the number of recognized contours
        contour_nr = len(data.distances)

        # Iterate over all  recognized contours
        for i in range(contour_nr):
                # Crop original image to bounding rectangle
                crop = original_image[data.bounding_rectangle_coords[i*4+1]:data.bounding_rectangle_coords[i*4+3], data.bounding_r
                if crop.shape[0] == crop.shape[1] and crop.shape[0] > 0 and crop.shape[1] > 0:
                        # Scale rectangle to a 220x220 image
                        scaled = cv2.resize(crop, dsize=(220, 220), interpolation = cv2.INTER_CUBIC)

                        # Initialize output message and set distance to object
                        output = Results()
                        output.distance = data.distances[i]

                        # Convert scaled image to a format recognizable by tensorflow
                        img = scaled[...,::-1].astype(np.float32) / 255.0

                        # Get image array
                        x = image.img_to_array(img)
                        x = np.expand_dims(x, axis=0)

                        # Get tensorflow result class and put into output message if confidence greater than 80%
                        predictions = self.model.predict(x)
                        idx = np.argmax(predictions[0])
                        rospy.loginfo("Predicted class: %s. Percentage: %f", self.CLASSES[idx], predictions[0][idx])
                        if predictions[0][idx] > self.predict_percent:
                                output.className = self.CLASSES[idx]
                                self.prevRecs[idx] = self.prevRecs[idx] + [output.distance]
```

3. Static or Dynamic object classification.

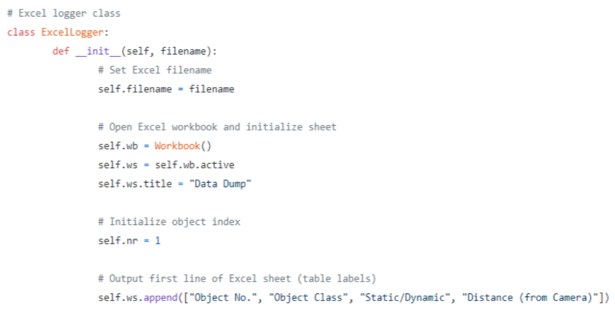
Next parameter to be determined of the object detected could be that it's static or dynamic. To differentiate the type of object i.e. static or dynamic, we will set a flag of epsilon whose value is 5. Therefore, if the previously recognized object has changed its position greater than 5 epsilon then that object is considered to be dynamic else object will be determined as static.



```
# If this object had been recognized before in a position differing by at least epsilon, label it as dynamic
epsilon = 5
if len(self.prevRecs[idx]) > 1:
        if (self.prevRecs[idx][-2] + epsilon < self.prevRecs[idx][-1]) or (self.prevRecs[idx][-2] - epsilon > self.prevRecs[idx][-1]):
                output.type = "Dynamic"
        else:
                output.type = "Static"
else:
        output.type = "Static"

# Publish output message
self.publisher.publish(output)
```

F. excel_logger.py:

Finally the output of the detected objects is the list of objects with their other parameters such as Object Class, if the object is Static or Dynamic and Distance of the object from the camera, is Excel sheet. To create an Excel sheet, we created a different ROS node and named it excel_logger. Here we use Openpyxl library, it is a Python library to read and write Excel 2010 xlsx/xlsm/xltx/xltm files.



```
# Excel logger class
class ExcelLogger:
        def __init__(self, filename):
                # Set Excel filename
                self.filename = filename

                # Open Excel workbook and initialize sheet
                self.wb = Workbook()
                self.ws = self.wb.active
                self.ws.title = "Data Dump"

                # Initialize object index
                self.nr = 1

                # Output first line of Excel sheet (table labels)
                self.ws.append(["Object No.", "Object Class", "Static/Dynamic", "Distance (from Camera)"])
```

This gives us the list of detected objects by our CNN model in the Excel sheet and the Excel file is saved at the desired location.

VI. PROCEDURE OF RUNNING THE PROGRAM

❖ *Following are the requirements:*

- Robot Operating System (ROS)

Project needs to have a working ROS installation. Make sure the full desktop version for either ROS Kinectic or Melodic is installed.

- rtabmap_ros

This is a ROS wrapper for the rtabmap library used to synchronize the Kinect messages. Install it using the following command, replacing <distro> with ROS version installed (kinetic or melodic):

sudo apt-get install ros-<distro>-rtabmap ros-<distro>-rtabmap-ros

- libfreenect2

This library provides drivers for the Microsoft Kinect. Install it if the program intends to use the Kinect sensor from the Xbox One.

First, the libfreenect2 library needs to be cloned from source. Open up a terminal in desired directory and run:

```
git clone https://github.com/OpenKinect/libfreenect2
cd libfreenect2
```

Install all dependencies. Use the following commands if base operating system is using Ubuntu 16.04 (for other systems, relevant commands need to be executed):

```
sudo apt-get install build-essential cmake pkg-config
sudo apt-get install libusb-1.0-0-dev
sudo apt-get install libturbojpeg libjpeg-turbo8-dev
sudo apt-get install libglfw3-dev
sudo apt-get install libopenni2-dev
```

Now make and install the library:

```
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/freenect2 -DENABLE_CXX11=ON
make
make install
```

- Python libraries

Install using pip2 (for Python 2.7):

```
pip2 install --upgrade tensorflow
pip2 install numpy
pip2 install matplotlib
pip2 install openpyxl
pip2 install scikit-image
```

- Catkin Workspace

Create a catkin workspace at desired repository:

```
source /opt/ros/melodic/setup.bash
mkdir -p catkin_ws/src
cd catkin_ws/
catkin_make
source devel/setup.bash
```

If using Kinect v1, clone the freenect_stack repository:

```
cd src
git clone https://github.com/ros-drivers/freenect_stack
```

If using Kinect v2, clone the iai-kinect2 repository:

```
git clone https://github.com/code-iai/iai_kinect2
```

Also, move all the program files (launch files and python files) attached with this paper in this repository.

Make the catkin workspace and source it:

```
cd ..
catkin_make
source devel/setup.bash
```

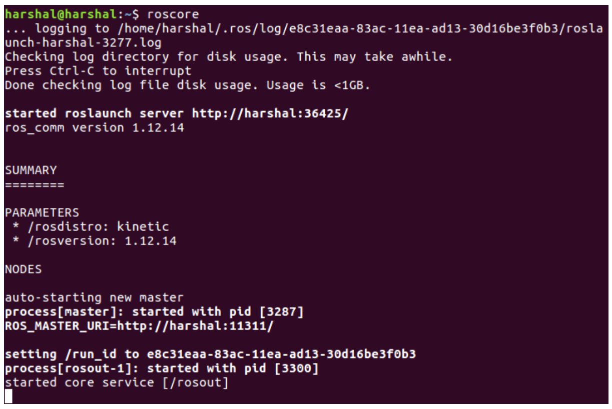
Make all the ROS Python files in the repository executable:

```
find src/desktop_object_detection/src -type f -exec chmod +x {} \;
chmod +x src/desktop_object_detection/test/result_message_stream.py
```

To check if everything is installed perfectly, run following command:

```
roscore
```

This should show the installed ROS version as shown below. If everything is installed correctly, now the recording of data and testing can be carried out. [9]



```
harshal@harshal:~$ roscore
... logging to /home/harshal/.ros/log/e8c31eaa-83ac-11ea-ad13-30d16be3f0b3/rosla
unch-harshal-3277.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://harshal:36425/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [3287]
ROS_MASTER_URI=http://harshal:11311/

setting /run_id to e8c31eaa-83ac-11ea-ad13-30d16be3f0b3
process[rosout-1]: started with pid [3300]
started core service [/rosout]
```

❖ *Record Kinect Data*

Connect the Kinect sensor by USB to the PC running with ROS. Use the following command to test whether the device is recognized:

```
lsusb | grep Xbox
```

This command will show all the Xbox devices connected via USB. Once you have seen them, open a terminal in the root of catkin workspace and run:

```
source devel/setup.bash
```

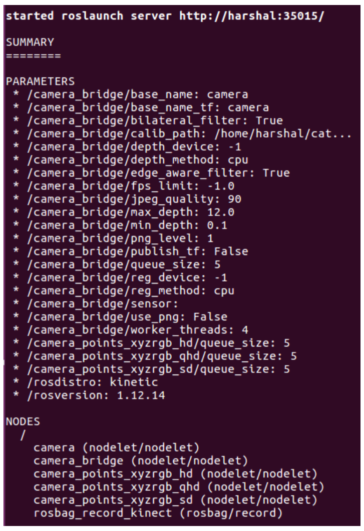
To visualise recorded data open a new terminal and run Rviz by following command:

```
rviz
```

Run the following command by replacing <prefix> with your desired prefix for the bag file that will be saved as absolute_path/name (e.g. /home/user/bag1 where bag1 is name of the bag file) and <using_v1> with false for using the Kinect from Xbox One:

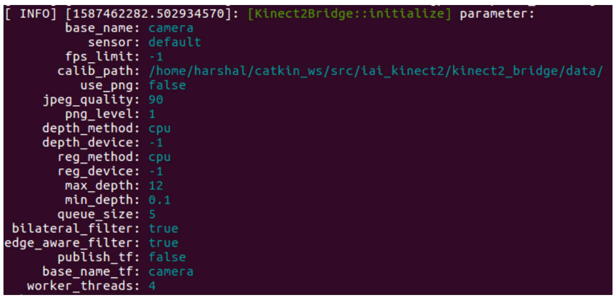
```
roslaunch desktop_object_detection record_bag.launch bag_prefix:=<prefix> kinect_v1:=<using_v1> rviz:= true
```

Kill at any desired moment all the processes to save the bag file by inputting Ctrl+C in the terminal that is running them.



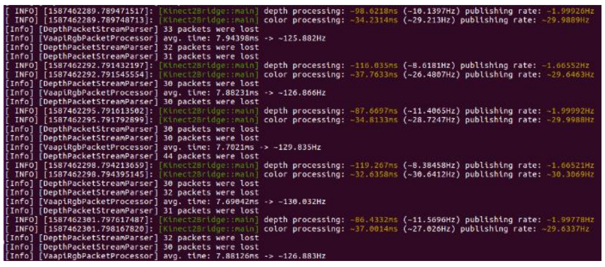
```
started roslaunch server http://harshal:35015/

SUMMARY
========

PARAMETERS
 * /camera_bridge/base_name: camera
 * /camera_bridge/base_name_tf: camera
 * /camera_bridge/bilateral_filter: True
 * /camera_bridge/calib_path: /home/harshal/cat...
 * /camera_bridge/depth_device: -1
 * /camera_bridge/depth_method: cpu
 * /camera_bridge/edge_aware_filter: True
 * /camera_bridge/fps_limit: -1.0
 * /camera_bridge/jpeg_quality: 90
 * /camera_bridge/max_depth: 12.0
 * /camera_bridge/min_depth: 0.1
 * /camera_bridge/png_level: 1
 * /camera_bridge/publish_tf: False
 * /camera_bridge/queue_size: 5
 * /camera_bridge/reg_device: -1
 * /camera_bridge/reg_method: cpu
 * /camera_bridge/sensor:
 * /camera_bridge/use_png: False
 * /camera_bridge/worker_threads: 4
 * /camera_points_xyzrgb_hd/queue_size: 5
 * /camera_points_xyzrgb_qhd/queue_size: 5
 * /camera_points_xyzrgb_sd/queue_size: 5
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    camera (nodelet/nodelet)
    camera_bridge (nodelet/nodelet)
    camera_points_xyzrgb_hd (nodelet/nodelet)
    camera_points_xyzrgb_qhd (nodelet/nodelet)
    camera_points_xyzrgb_sd (nodelet/nodelet)
    rosbag_record_kinect (rosbag/record)
```

record_bag.launch file with its parameters and ROS nodes



```
[ INFO] [1587462282.502934570]: [Kinect2Bridge::initialize] parameter:
        base_name: camera
           sensor: default
        fps_limit: -1
       calib_path: /home/harshal/catkin_ws/src/iai_kinect2/kinect2_bridge/data/
          use_png: false
     jpeg_quality: 90
        png_level: 1
     depth_method: cpu
     depth_device: -1
       reg_method: cpu
       reg_device: -1
        max_depth: 12
        min_depth: 0.1
       queue_size: 5
 bilateral_filter: true
edge_aware_filter: true
       publish_tf: false
     base_name_tf: camera
   worker_threads: 4
```

Kinect Camera Calibration



```
[ INFO] [1587462289.789471517]: [Kinect2Bridge::main] depth processing: ~98.6218ms (~10.1397Hz) publishing rate: ~1.99926Hz
[ INFO] [1587462289.789748713]: [Kinect2Bridge::main] color processing: ~34.2314ms (~29.213Hz) publishing rate: ~29.9889Hz
[Info] [DepthPacketStreamParser] 33 packets were lost
[Info] [VaapiRgbPacketProcessor] avg. time: 7.94398ms -> ~125.882Hz
[Info] [DepthPacketStreamParser] 32 packets were lost
[Info] [DepthPacketStreamParser] 31 packets were lost
[ INFO] [1587462292.791432197]: [Kinect2Bridge::main] depth processing: ~116.035ms (~8.6181Hz) publishing rate: ~1.66552Hz
[ INFO] [1587462292.791545554]: [Kinect2Bridge::main] color processing: ~37.7633ms (~26.4807Hz) publishing rate: ~29.6463Hz
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [VaapiRgbPacketProcessor] avg. time: 7.88231ms -> ~126.866Hz
[Info] [DepthPacketStreamParser] 30 packets were lost
[ INFO] [1587462295.791613502]: [Kinect2Bridge::main] depth processing: ~87.6697ms (~11.4065Hz) publishing rate: ~1.99992Hz
[ INFO] [1587462295.791792899]: [Kinect2Bridge::main] color processing: ~34.8133ms (~28.7247Hz) publishing rate: ~29.9988Hz
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [VaapiRgbPacketProcessor] avg. time: 7.7021ms -> ~129.835Hz
[Info] [DepthPacketStreamParser] 44 packets were lost
[ INFO] [1587462298.794213659]: [Kinect2Bridge::main] depth processing: ~119.267ms (~8.38458Hz) publishing rate: ~1.66521Hz
[ INFO] [1587462298.794395145]: [Kinect2Bridge::main] color processing: ~32.6358ms (~30.6412Hz) publishing rate: ~30.3069Hz
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [DepthPacketStreamParser] 32 packets were lost
[Info] [VaapiRgbPacketProcessor] avg. time: 7.69042ms -> ~130.032Hz
[Info] [DepthPacketStreamParser] 31 packets were lost
[ INFO] [1587462301.797617487]: [Kinect2Bridge::main] depth processing: ~86.4332ms (~11.5696Hz) publishing rate: ~1.99778Hz
[ INFO] [1587462301.798167820]: [Kinect2Bridge::main] color processing: ~37.0014ms (~27.026Hz) publishing rate: ~29.6337Hz
[Info] [DepthPacketStreamParser] 32 packets were lost
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [VaapiRgbPacketProcessor] avg. time: 7.88126ms -> ~126.883Hz
```

While recording bag file, Depth and colour processing being captured.

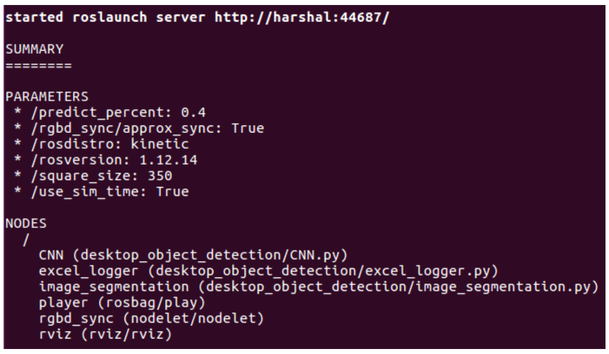
❖ *Play Bag file and Run*

Open a terminal in the root of catkin workspace and run:

```
source devel/setup.bash
```

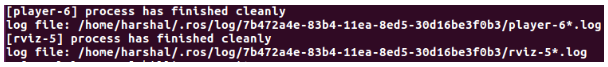
To run all the nodes while playing back the bag file saved using the previous procedure, use the following command by replacing <log_path> with the absolute path of the Excel log file that will be created (use a .xlsx extension), <bag_filename> with the absolute path to the recorded bag file, <model_dir> with the absolute path to the directory of the CNN, and <using_v1> with false for using the Kinect from Xbox One:

```
roslaunch desktop_object_detection run_bag.launch log_filepath:=<log_path> filename:=<bag_filename> model_path:=<model_dir> kinect_v1:=<using_v1> rviz:= true
```



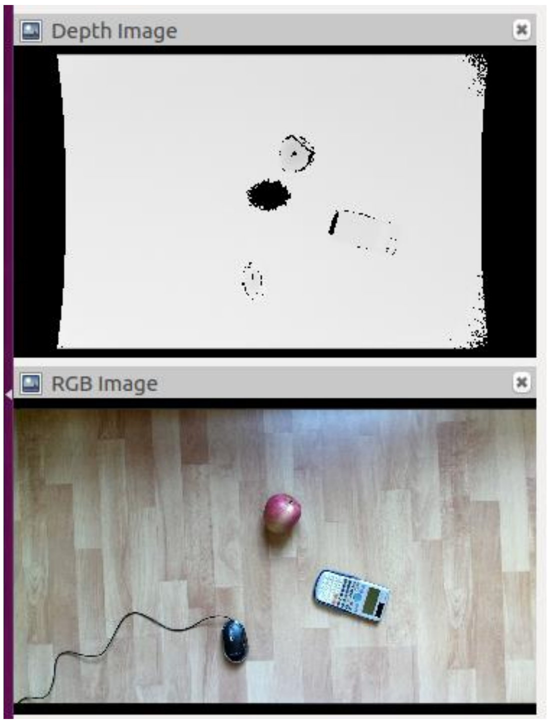
```
started roslaunch server http://harshal:44687/

SUMMARY
========

PARAMETERS
 * /predict_percent: 0.4
 * /rgbd_sync/approx_sync: True
 * /rosdistro: kinetic
 * /rosversion: 1.12.14
 * /square_size: 350
 * /use_sim_time: True

NODES
  /
    CNN (desktop_object_detection/CNN.py)
    excel_logger (desktop_object_detection/excel_logger.py)
    image_segmentation (desktop_object_detection/image_segmentation.py)
    player (rosbag/play)
    rgbd_sync (nodelet/nodelet)
    rviz (rviz/rviz)
```

run_bag.launch file with its parameters and ROS nodes



```
[player-6] process has finished cleanly
log file: /home/harshal/.ros/log/7b472a4e-83b4-11ea-8ed5-30d16be3f0b3/player-6*.log
[rviz-5] process has finished cleanly
log file: /home/harshal/.ros/log/7b472a4e-83b4-11ea-8ed5-30d16be3f0b3/rviz-5*.log
```

Successful test of bag file



Rviz RGB Image and Depth Image visualisation

VII. APPLICATION

The main purpose of this project was to build the object detection and situation awareness for the Kuka robot. We built the ROS packages which can be used to detect the objects and work on the navigation system of the Kuka robot. Since, the CNN model we created is trained on the objects which are more frequently observed on the common desk, we assume this project will serve the requirement of the robot. Moreover, this project is also capable of determining the distance of the object from the camera more precisely.

Other than this, the CNN model can be used for detection of objects in other applications such as Facial recognition, Analyzing documents, etc. Additionally, the model can be trained on new dataset and/or can be used for detection of some other objects as well.

VIII. CONCLUSION

In this project, we have successfully implemented the Image segmentation and Convolutional Neural Network (CNN) for object detection. An analysis of the different approaches for object detection was discussed. One of the approaches we find suitable for our project is to do image segmentation by OpenCV and image classification by CNN. CNN could also perform image segmentation but we have limited the role of CNN to image classification only for better accuracy and precision.

OpenCV functions played a significant role in image segmentation. The raw image was edited as estimated and the bounding boxes were generated around the objects in the image. This in turn become an input to CNN. CNN creates the way we see the world and operate within it for the machines. The bag files recording was challenging part, since Kinect camera has minimum and maximum distance from which it can detect the depth of the image. Initially we recorded bag files by placing objects very close to the camera and ROS packages were not able to calculate the distance of the object. In the end, running the complete project in Robotic operating system was a big challenge but we have successfully created different ROS nodes for image segmentation and image classification.

Our future aim is to expand the project to detect more objects and also there is more space for improving the prediction percentage and accuracy of all objects. The different version of CNN is also evolving, therefore, choosing a suitable CNN could also be a part of future research.

IX. ACKNOWLEDGEMENT

We are profoundly grateful to Prof. Dr. Andreas Pech, Prof. Peter Nauth and S. Sharan for their advices and guidance during this course of work. Additionally, we would like to thank the open-source community especially ROS community for enabling platform for free collaboration provided for ROS, OpenCV libraries and authors and students for publishing their work on an open source platform. And also we would like to thank all my friends who helped me directly or indirectly during this course of work.

X. REFERENCES

[1] "OpenCV," [Online]. Available: https://docs.opencv.org/master/d6/d00/tutorial_py_root.html.

[2] "Medium," [Online]. Available: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.

[3] "ImageNet Classification with Deep Convolutional," Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, University of Toronto.

[4] "Towards Data Science," [Online]. Available:https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

[5] "Towards Data Science," [Online]. Available:https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[6] "Arcticle," [Online]. Available: https://link.springer.com/article/10.1007/s13244-018-0639-9.

[7] "The Science and Information Organization," [Online]. Available: https://thesai.org/Downloads/Volume10No6/Paper_38-Hyperparameter_Optimization_in_Convolutional_Neural_Network.pdf.

[8] "Towards Data Science," [Online]. Available:https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[9] "Robotic Operating System," [Online]. Available: http://wiki.ros.org/.

[10] "Towards Data Science," [Online]. Available:https://towardsdatascience.com/what-why-and-how-of-ros-b2f5ea8be0f3.

[11] "Towards Data Science," [Online]. Available:https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce8.

[12] "Tensorflow," [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn.