Robotics for Artificial Intelligence – KIMROB03
ALICE - A Domestic Service Robot

Group n. 04 — Parth Tiwary and Danny Rogaar

***Abstract*— A domestic service robot should be capable of retrieving known objects from known locations. Here, we practice the situation with the domestic service robot ALICE, focusing on the overall behaviour, navigation, object recognition and grasping. Navigation applied Dijkstra's algorithm to both waypoints and paths. Object recognition uses a convolutional neural network. Both navigation and object recognition performed well during tests. Simulations have shown grasping to be successful although the action failed in practice.**

I. INTRODUCTION

Domestic service robots are to live with people at home, be autonomous to some degree and assist in certain tasks. Such a robot is used, amongst others, in surveillance, cleaning and assisted living and aim to make our lives easier in general. We use the domestic service robot ALICE, that is to do basic household chores such as retrieving objects from some known location. Performing household tasks requires skills in multiple domains be it computer vision, navigation and motor control. With advances in deep learning, both object recognition [Martinez-Martin, 2016] and grasping [Hodson, 2018] have shown improvements, adding to the potential of domestic service robots in the future.

To work on the implementation of ALICE's behaviour we integrate approaches in navigation, object recognition and grasping. The robot is tasked to retrieve boxes from two tables. Hence, we move to the given table(s), see which objects are on it and grasp the ordered boxes. Upon grabbing all objects, the robot returns to a drop-off point. Navigation occurs between location points selected for this, i.e. waypoints, for which Dijkstra's shortest path algorithm [Dijkstra, 1959] is used. The algorithm is implemented using a priority queue data structure to ensure fast execution of planning, which means the robot does not wait too long for a plan. Object recognition is done using the effective convolution neural network [Krizhevsky et al., 2012] approach. The network takes grayscale images from a camera and classifies these into one of the four defined objects. Grasping uses the MoveIt framework [MoveIt, 2019], to which we submit a range of grasps.

In particular, navigation happens on multiple scales. That is, the robot uses the planning algorithm to get the shortest route of waypoints to move to e.g. the next table. Then, the algorithm is also used to plan the shortest valid path between the robot and its next waypoint. To plan between waypoints, they are part of a graph defining routes along which the robot can move. Navigation thus happens hierarchically, which means a graph of waypoints can be easily created for any environment allowing the robot to move there.

Since they were popularised by Krizhevsky in [Krizhevsky et al., 2012], convolutional neural networks (CNN) are state-of-the-art in object recognition [Zhao et al., 2018], [Xie et al., 2016]. Thus, we also use a CNN for object recognition in ALICE. Although state-of-the-art deep learning models are trained with a vast amount of images, few are available to us. A dataset was made available with around 213 images of each box to be recognised. To mitigate the issue with little data, we use data augmentation techniques in order to add more variations of the provided data before training.

For the final demonstration task, our implementation of the service robot is presented with three level of difficulties:

- Alice needs to pick up 1 object from either table 1 or table 2. The specified object can be present with other objects. Alice should return the item to the drop-off point after grasping.
- Alice needs to pick up 2 or more objects from either table 1 or table 2. Objects might or might not be at the locations. Objects should be returned to the drop-off point after grasping.
- Alice needs to grasp all the objects from both the tables in the specified location order. All the objects should be returned to the drop-off point.

In section II we elaborate on specifics of the navigation, object recognition along with data augmentation regime used, which is followed by a discussion on specifics of grasping and, finally, the overall behaviour. Then, section III describes and further analyses results from the performed experiments. Finally, we conclude with a discussion in section IV.

II. METHOD

A. Navigation

The navigation behaviour is to enable ALICE to move between locations. To do so, we implement Dijkstra's algorithm which returns the shortest path between two points in a graph. However, the shortest path sends the

The authors are with Faculty of Science and Engineering, University of Groningen, The Netherlands. {p.d.rogaar,p.p.tiwary}@student.rug.nl

robot very close to obstacles, causing movement to become cautious to avoid hitting anything. Instead, we want the robot to drive along the given waypoints which form paths along the centre of rooms, preventing any slowdowns. The planning algorithm, then, determines which waypoints to follow, and what would be the shortest path between the robots current location and the next waypoint.

For path planning, the robot location and target waypoint are required, as well as the waypoint graph. The environment is a static map (shown in figure 1), obtained by the robot using a laser scanner. From the map, a costmap is created specifying where the robot should not go, that is, into and near obstacles such as tables and closets. The costmap can be seen as a graph where every pixel represents a node that has its neighbouring pixels and diagonally adjacent pixels linked, in an undirected way. Distances are Euclidean so that the shortest path can be retrieved. However, when a costmap node has a cost greater than 0, implying it is near an obstacle, the distance to this node is defined infinite, meaning the robot cannot move near obstacles.
Navigation is facilitated with an understanding of the environment based on the information received from global and local costmaps. The global costmap is generated from the static map and depth camera data. The process involves adding an inflation layer specifying real configuration space for the robot, done by adding padding around objects present in the environment. Effectively, the obstacles in the static map are inflated in order to account for the size of the robot(footprint), producing a global costmap. In the current setup, we also include immediate obstacle data by using the depth camera and laser(sensors), for the global costmap. For the local costmap creation, we only use the obstacle layer and inflation layer on the immediate surroundings. Unlike in case of global costmaps, we do not use static map for the local costmap.
In order for the robot to localise itself in the map, adaptive Monte Carlo localisation is used [Fox et al., 1999], [Fox, 2002]. With the costmaps and the robot localised, a navigation plan can be made. The global planner uses the global costmap for generating a plan. The local planner depends on the local costmap, the local planner breaks the global plan into smaller portions in order to reach the goal. To optimise the run-time of path planning, python's heapq is used. Heaps are binary trees in which every parent has value less than or equal to the child node. We use heappush and heappop operations in order to push all the neighbors of the current node on the heap and extracting the next node with the shortest distance, respectively. Both push and pop here have a time complexity of O(log(n)) which significantly improves the run-time of Dijkstra's algorithm.
Although waypoints provide a nice way to specify paths (a list of waypoints), it is hard to keep track of all possible paths for the robot to move along, especially as the environment scales up to real life situations and the robot tasks involve several locations. To keep track, we define only the connections between each waypoint in a graph. By applying the path planning algorithm to the waypoint graph (for which the positions allow calculating Euclidean distances), we can dynamically obtain a path of waypoints to some target location, instead of having to define it separately. Since our path planner assumed a costmap as input, we convert the waypoint graph by putting the waypoints in an array and redefining the neighbours and distances. The waypoints and their paths which we defined can be seen in figure 1.

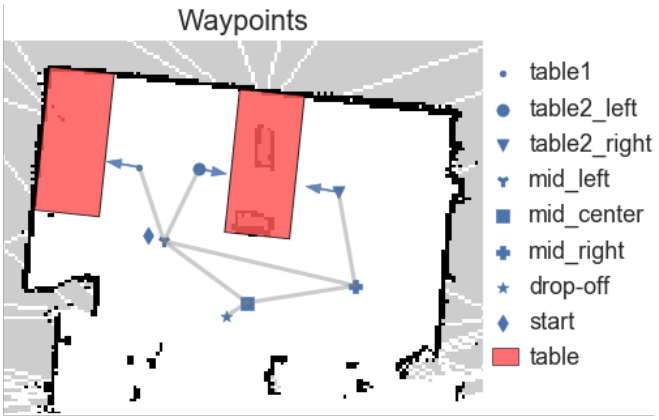


Fig. 1: Navigation waypoints as located in the map. Arrows indicate fixed waypoint orientations. For waypoints without arrows, a direction is determined on the fly. Gray lines show where the robot can move from each waypoint. On start, the robot may only move to mid_center.

Now that we have a route of waypoints specifying the positions along which Alice needs to move, we also need to return an associated orientation with each of the waypoints since it differs depending on the movement direction. We therefore calculate orientations on the fly. Unfortunately, orientations cannot be directly based on some coordinate axis, since the axes depend on how the robot started mapping and can be arbitrary. We solve the issue by constructing a new coordinate system. We calculate basis vectors for a positive x axis using equation 1. In eqn. 1, each waypoint is a pair of x and y coordinates and the mean is taken separately for each spatial dimension.

$$\begin{aligned} x^{+} \text{ axis basis vector} = \text{mean}(& \\ & \text{table2\_right} - \text{table2\_left}, \text{table2\_right} - \text{table1}, \\ & \text{table2\_left} - \text{table1}, \text{mid\_right} - \text{mid\_center}, \\ & \text{mid\_right} - \text{mid\_left}, \text{mid\_center} - \text{mid\_left}) \end{aligned} \tag{1}$$

A basis vector for the y axis is set orthogonal to x and facing in the direction of table 2 seen from mid_center. Now, every waypoint has its coordinates projected onto the new axes giving a system independent of the coordinates used by the map. For the grasping locations we can directly set the facing direction to the positive x axis (table2_left) and negative x axis (table1 and table2_right), as shown with the blue arrows for these locations in fig. 1. For the other waypoints, the orientation is set along the longest axis of

movement.

B. Object Recognition

The robot is equipped with a colour camera to see objects it is tasked to grab. Depth camera data is used to locate the table, and then get the approximate locations and size of the objects in the image. With the object information, the objects are cut from the colour image. The image crops are then converted to grayscale and resized to 128x128 after which a CNN is used to classify the objects. The grayscale format was used to avoid problems with the image channels. Moreover, grayscale provided less information to the CNN which we hope prevents overfitting with little data.
Classification is performed using the architecture shown in fig. 2. The architecture uses blocks of 3x3 convolutions with increasing depth and 2x2 max pooling. After the fifth block, the features are flattened and a fully connected layer is attached with 512 neurons and 50% dropout. A final fully connected classification layer is attached for the four target objects (usbhub, eraserBox, Evergreen and tomato soup). We decided on using multiple convolutional layers with pooling in-between to keep the number of parameters low with respect to shallow neural networks, in order to prevent overfitting. All layers use ReLU [Glorot et al., 2011] activation, besides the final layer that has softmax activation. The network is trained using Adam Optimizer [Kingma and Ba, 2017] with a learning rate of 0.002. After a few epochs the training accuracy reaches approximately 97% and training is stopped after 15 epochs. The high accuracy can be easily achieved given that validation data are unseen augmentations of the original images, similar to augmentations the network is trained on. Unfortunately, our original dataset contained only one instance of each object (though with rotations and slight light variations) such that it is not feasible to construct validation data excluding augmentations. Given the high accuracy, we expect the network to be over-fitted to the training data.

Our data consists of boxes of 4 classes, namely usbhub, eraserbox, Evergreen and tomato soup. All images are around 213 rotations of a single object from each class converted to grayscale.
Given how small our initial dataset is, we desire to enlarge it by data augmentation. By properly augmenting the images, we have more variations to train the network which should result in better object recognition. Moreover, it also allows us to add invariance to common circumstances such as different lighting conditions or viewing angles which can both hinder successful recognition. We add the following augmentations:

- Horizontal Flip: The image is flipped horizontally, which increases the invariance to the viewing angle.
- Brightness: Since the robot views some objects in different light settings, we hope to mitigate problems by changing the image brightness. To do so, the image is converted to HSV and the value is set between 50 and 200 randomly. Then, the image is converted back to RGB
- Image extraction error: Images of objects are imperfectly extracted by the robot. The object may not be completely visible but the network should still classify such images correctly. To handle the problem, we alter how box images are cut from the RGB scene taken by the camera, for training data. Originally, image crops are defined by a bounding box for each object. We generate scalars between plus and minus 25% of the image height. The scalars are used to offset the left, right, top and bottom of the bounding box separately, ensuring the new crop satisfies the image constraints.

The resulting dataset contains around 800 images for each class, in grayscale and finally resized to 128x128. Some of the generated samples can be seen in figure 3.

C. Grasping

For the robot, object recognition and grasping occur at the same locations, so that these are merged in a single action. Thus, after approaching a table, images are extracted for which the CNN detects an object. Once the object is known, pre-defined dimensions of that object are used to construct a collision model, at the approximate location and orientation of the object. With the collision model, a grasp is attempted. Alice is equipped with a Kinova mico arm for grasping, with 6 degrees of freedom and two finger end effectors. To grasp, first, the arm is moved to an intermediate waypoint located away from the robot's own collision model. The arm is then placed above the object and arm poses are defined. After grasping, the intermediate waypoint is used again, before positioning the arm above the robot's body to place boxes. If successful, the object is picked up and placed on the robots back in order to return it to the drop-off point.

To perform a grasp, a few poses are defined such that the arm has a valid trajectory that respects the surrounding objects it can collide with. The grasp consists of, amongst others, an approach before grasping, specifying that the arm should move 15 cm downwards. After the grasp a retreat specifies the arm should move 15 cm up. The posture is specified as having an open gripper before grabbing and a closed gripper during the action. Finally, a grasp pose specifies where the gripper should be to grab the object. To construct these poses, first the object is classified as described in section II-B and the orientation is estimated using an orientation CNN. A predefined collision box (for the now known object) is then placed in an occupancy grid which now contains all information required to grasp the object without collisions. The occupancy grid contains voxel information of the environment detected using the depth camera. With the collision box, we submit 60 positions for the grasp pose, oriented downwards, between the objects centre and twice the object's maximum height. The MoveIt framework picks the most appropriate grasp.

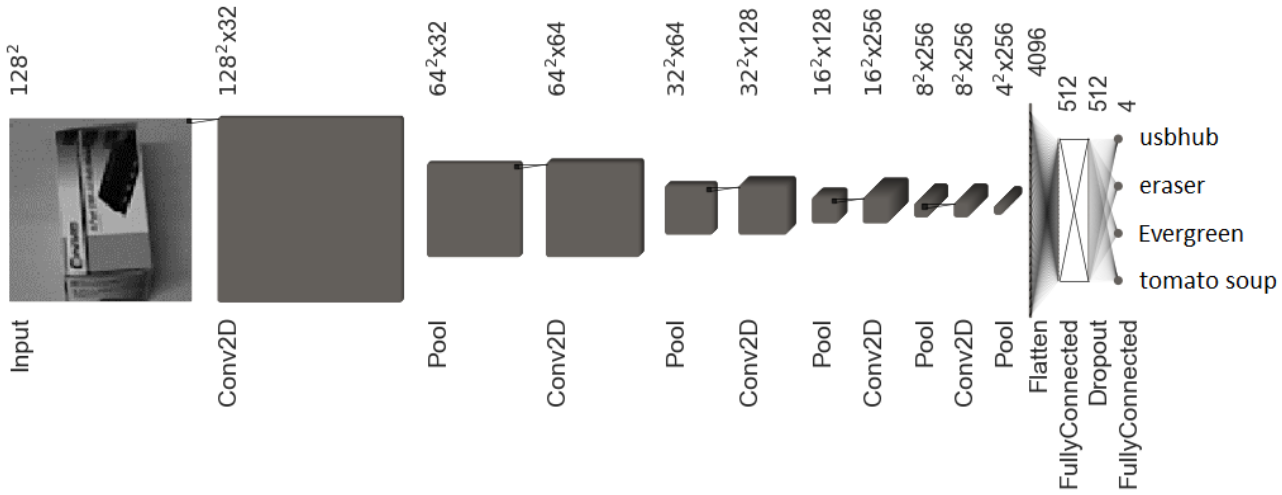


Fig. 2: Convolutional neural network used for box classification. Convolutions use 3x3 kernels and pooling is done with 2x2 kernels for max-pooling. Dropout kept 50% of the activations during training. Layers use ReLU activation although the final layer uses softmax.

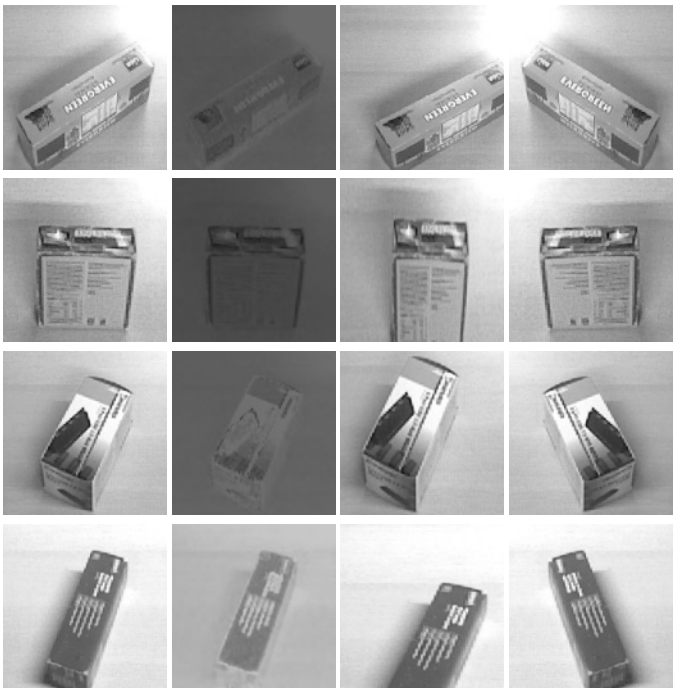


Fig. 3: A subset of the training set, including from left to right: original image, brightness, image extraction and flip augmentation.

D. Behaviour

The complete behaviour cycle of the robot consists of the navigation behaviour, approaching a table, the object detection and grasping action, and finally moving back from a table. The main behaviour uses a local state to keep track of what to do. The local state starts in *waiting_for_order* indicating that an order needs to be sent, specifying objects to be taken from which locations (table 1 or table 2). With an order given, the robot moves to the first table it needs to go to, obtaining the waypoints and starting the navigation behaviour.

Navigation itself uses a few states as seen in figure 4. The behaviour requires a list of waypoints along which to navigate and keeps an index indicating at which waypoint it is. Once the planner initiates, the behaviours state is set to waiting until a path is returned. The robot then moves along the planned path to the next waypoint. In case a plan is not possible or takes too long at any point, the waiting state switches to failed. When successful, however, the robot has reached the next waypoint upon which the state changes to *reached_waypoint*. Then, the behaviour increases the waypoint index and restarts itself. Once the final waypoint is reached the behaviour finishes.

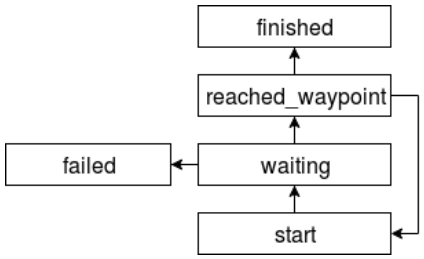


Fig. 4: Navigation subbehaviour states. Navigation to a waypoint happens in the waiting state. *reached_waypoint* denotes a success and restarts the cycle with the next waypoint until none are left (finished). Too much time in the waiting state eventually results in failure.

After successful navigation, the robot is to approach the table, recognise and grab objects in the order, then move back. Repeating the cycle for all locations means looking everywhere for the given object which has the best way to find it. Grabbing every object of the given class also means clearing the tables of it. The behaviour cycle can be seen in fig. 5. Since there are three grasping locations, the behaviour

cycle can be executed three times. However, if only one table is the target, the robot will only execute the behaviour cycle once (table-1) or twice (table-2).
Figure 5 shows the local states used internally, that basically denotes which behaviour or action should be executed and in which cycle the behaviours are. *Approach* indicates the robot is put in proximity of a table and has to drive up to it. The approach action makes the robot look around for objects on the table and if some are seen it will steer the robot in front of them without colliding with the table. After moving in front of the objects, the robot aligns itself with the table. Although the action can be problematic if no objects are seen, we assume this action does not fail. *Graspingloc* is the state used when the robot is directly in front of a table and needs to recognise and grab the present objects, as explained in the previous section. *Reproach* indicates when the robot has finished grasping and needs to move back. The state issues a simple command to the controller which moves the robot back 80cm, to the configuration space such that navigation can take over again.
After moving back from a table, navigation then starts again in the new cycle, navigating to one of the other grasping locations or to the drop-off point (*todropoff*). After reaching drop-off the robot mentions the objects it took and resets the local state to the starting state, *waiting_for_order*.
In figure 5 one also finds that aside from the normal control flow, the navigation behaviour and grasping action may fail. In particular, navigation fails when no plan is found and grasping aborts when any exception occurs in either the neural networks or grasping itself. If either of them fail, we try to rerun the behaviour or action, after re-positioning the robot in an attempt to change to circumstances of error. Re-positioning happens by setting the local state to the state used for navigation, and then adding the waypoints after the current one. As navigation starts, it will move the robot back to the previous waypoint that it did reach, after which the robot returns to the situation of failure (from a different angle).

III. EXPERIMENTS

The robot is tasked to fulfil randomly chosen orders, grabbing up to four objects from the two tables. The robot is placed roughly at the starting point as shown figure 1 with an orientation aiming towards mid_center. Three orders are sent and the robot should be able to execute these without requiring a restart.
Performing the test, we did require a restart as we set the starting local state to *graspingloc1* meaning that the robot started looking for objects after sending the first order. In the starting position of course, no objects were found meaning that the grasping action did not fail and the robot continued driving backwards (from a supposed table). With an obstacle behind ALICE, the action required a mandatory stop and restart. The failure case highlights the necessity of checking the environment for obstacles before approaching or moving back from a table. After this, the local state was appropriately set and the robot executed the orders in sequence correctly. We take a better look at the performance of the individual components.

A. Navigation

We found the navigation behaviour works well both in simulation and during the test. Surprisingly the starting position of the robot was quite robust; if the robot did not match the starting waypoint from fig. 1, it would plan a path to move there. In the real world, we found that waypoints had to be set with obstacles in mind, as the obstacles would add to the inflation layer in the costmap causing planning to fail. With proper waypoints, however, navigation was smooth and ALICE managed to drive from start to drop-off in the last two orders consecutively. With the navigation behaviour used, we did not test the effectiveness of the fallback mechanism as the robot did not get stuck. Previously, errors in waypoint orientations caused the robot to unnecessarily rotate. The issue was fixed by introducing the new coordinate system as discussed in section II-A.

B. Object Recognition

Although our network is designed to classify the given objects, its overfit to the training set with a lot of augmentations, giving a 97% accuracy. The result is that the network tends to be very sure in its predictions even though its accuracy does not match the given confidences. We aimed to reduce the amount of weights in comparison with large pretrained networks, since we do not have enough data to properly train a big CNN and the classification task is not very complex with small images and only 4 classes. During the test, we saved the extracted images showing the boxes seen by the network.
First, we note that objects are not always properly detected. For example, figure 6 shows two images without objects but with artefacts. The most corrupted image (fig. 6b) was classified as usbhub, whereas from another angle and less artefacts the scene is classified as tomato soup (fig. 6a). The artefacts occur in dark areas and may present an issue to the neural network, although we do not observe artefacts on the images containing objects.
Next, we evaluate the performance of the object recognition itself. Images extracted during the test are mostly duplicates. We show unique views of the objects in figure 7. The figure also shows how often correct prediction happened for the shown images. Notably, when the tomato soup is seen from the top, it is misclassified as eraserbox 4 times, which is a reasonable error, given that this angle makes the soup box look small and dark similar to the eraserbox. Overall, the recognition classified 40 images correctly out of 45 detected boxes.

C. Grasping

With successful object recognition during the experiments, grasps were attempted several times. Upon passing the candidate grasps to MoveIt, however, consistently no valid grasps were found. In particular, errors returned specifying a lack of odometric resolution in order to setup the given

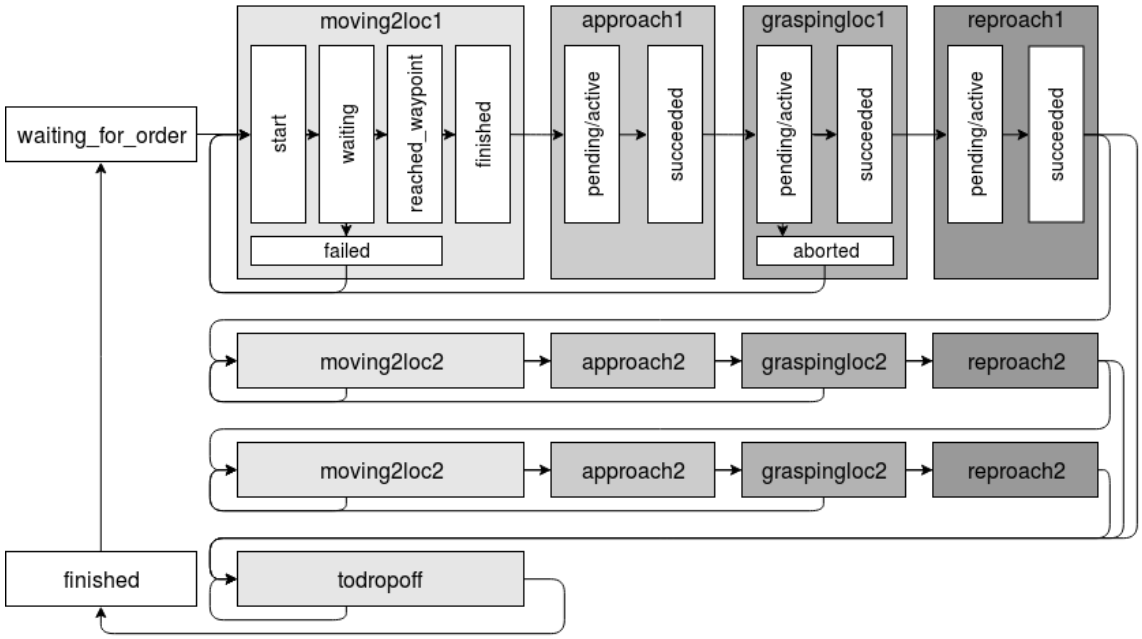
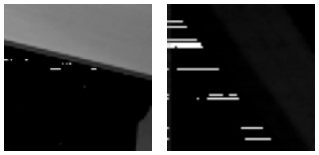
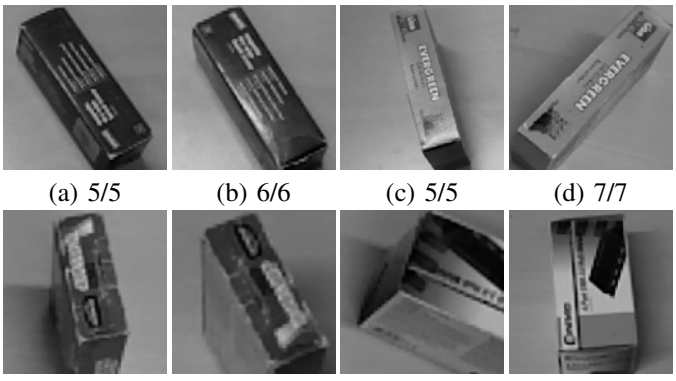


Fig. 5: The local state of our main behaviour. Colours denote which behaviours are used in each state. That is, from light to dark respectively: Navigation sub-behaviour, approach action, grasping action and the backwards movement action. The states *finished* and *waiting_for_order* are handled in the main behaviour for which *waiting_for_order* acts as the starting state.



(a) tomato soup (b) usbhub

Fig. 6: Nine images are wrongly identified as a tomato soup object (6a) and one as usbhub (6b)



(a) 5/5 (b) 6/6 (c) 5/5 (d) 7/7

(f) 1/5 (g) 5/5 (h) 6/7 (i) 5/5

Fig. 7: Unique object views of eraser (a & b), Evergeen (c & d), tomato soup (f & g) and usbhub (h & i). Labels mention how many correct predictions were made in the amount of images taken from this angle.

pose. In simulation, errors did not occur and grasping was successful.

IV. DISCUSSION / CONCLUSION

We implemented a two-layer navigation behaviour for ALICE, working between map locations and between two waypoints. After moving to a table, the robot then uses a CNN trained with augmented images to classify, and another CNN to orient the objects seen. Then, we specify a number of grasps to obtain an optimal one, based on the properties of the object. The object has to be taken along until all ordered boxes are retrieved, after which the robot can return.

When tasked to complete the order and retrieve several boxes from known locations, we observed the performance of each part of the behaviour. Navigation worked very well, and did not fail to move the robot to every desired position. Object detection failed a few times when a table was misidentified as an object. Object identification is based on the depth sensors readings and attempts to locate the table before doing so. The errors are, thus, due to looking in front of the table instead of on top, which misidentifies the table as an object. Object recognition, however, worked well and correctly classified 40/45 images. Four of those errors were due to tomato soup viewed from top which looks like, and is classified as an eraserbox. After recognition, the robot was to grasp the objects. Unfortunately, no valid grasp has been performed throughout the experiments, as sensors readings could not make out the difference between joint poses. With no failure cases in navigation, and no exception in grasping, fallback mechanisms (re-positioning the robot) were not tested. However, the behaviour was overall successful but unable to bring any object required due to grasping failure.

REFERENCES

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271.

[Fox, 2002] Fox, D. (2002). Kld-sampling: Adaptive particle filters. In Dietterich, T. G., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems 14*, pages 713–720. MIT Press.

[Fox et al., 1999] Fox, D., Burgard, W., Dellaert, F., and Thrun, S. (1999). Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the National Conference on Artificial Intelligence*, pages 343–349.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. *Journal of Machine Learning Research*, 15:315–323.

[Hodson, 2018] Hodson, R. (2018). How robots are grasping the art of gripping. *Nature: international journal of science*, 557:s23–s25.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization. *arXiv.org*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*.

[Martinez-Martin, 2016] Martinez-Martin, E. (2016). Object detection and recognition for assistive robots. *ROBOTICS & AUTOMATION MAGAZINE*.

[MoveIt, 2019] MoveIt (2019). https://moveit.ros.org.

[Xie et al., 2016] Xie, S., Girshick, R. B., Dollár, P., Tu, Z., and He, K. (2016). Aggregated residual transformations for deep neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995.

[Zhao et al., 2018] Zhao, Z.-Q., Zheng, P., tao Xu, S., and Wu, X. (2018). Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*.