Total Singulation With Modular Reinforcement Learning

Iason Sarantopoulos, Marios Kiatos, Zoe Doulgeri, and Sotiris Malassiotis

***Abstract*—Prehensile robotic grasping of a target object in clutter is challenging because, in such conditions, the target touches other objects, resulting to the lack of collision free grasp affordances. To address this problem, we propose a modular reinforcement learning method which uses continuous actions to totally singulate the target object from its surrounding clutter. A high level policy selects between pushing primitives, which are learned separately. Prior knowledge is effectively incorporated into learning, through action primitives and feature selection, increasing sample efficiency. Experiments demonstrate that the proposed method considerably outperforms the state-of-the-art methods in the singulation task. Furthermore, although training is performed in simulation the learned policy is robustly transferred to a real environment without a significant drop in success rate. Finally, singulation tasks in different environments are addressed by easily adding a new primitive and by retraining only the high level policy.**

***Index Terms*—Deep learning in grasping and manipulation, perception for grasping and manipulation.**

I. Introduction

Effective robotic grasping in realistic unstructured environments, such as those inhabited by humans e.g. homes and workplaces, is yet to be achieved. Most of these environments are highly cluttered, prohibiting the robotic fingers from reaching and grasping the target object. Dealing with clutter means that the robot should rearrange the obstacles around the target object in order to create enough space for the robotic fingers to perform a prehensile grasp. Prehensile grasp refers to grasping with opposable fingers, as opposed to non-prehensile grasping of an object using a suction cup. Although a plethora of methods [1], [2] have been proposed for prehensile grasp planning and execution, the majority assume a collision free space around the target. Consequently, another body of research emerged focusing on developing methods for singulating the target object, partially or totally, from the surrounding clutter using actions such as pushing, sliding or picking objects preceding grasping. Total singulation refers to the creation of free space around the object (as shown in Fig. 1), while partial singulation refers to the creation of free space along some object sides.

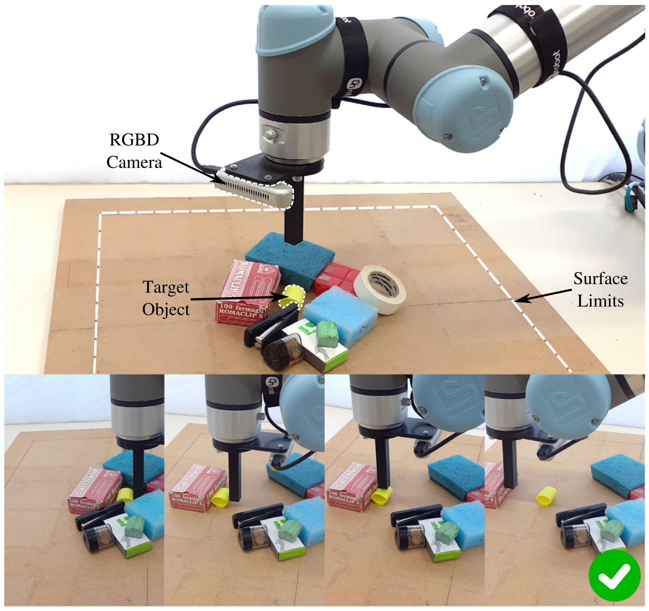


Fig. 1. Total singulation of a target object in our experimental setup.

Due to the variety of existing grasping methods, any singulation method needs to avoid making unnecessary assumptions about the subsequent grasp in order to increase compatibility with as many grasping methods as possible and also increase success rates in realistic conditions. If the singulation method assumes an opposable grasp, then partial singulation of the target is sufficient, but it will fail in cases where different grasping strategies that require free space around the object need to be employed. For example, power grasps [3]–[6] or grasp strategies that exploit contact with the environment [7]–[9] are preferred to compensate for perception errors and require free space around the object. For this reason, our approach deals with total singulation of the target object, i.e. the distance of the target object with the closest obstacle being larger than a threshold to enable a variety of grasps and increase their robustness to errors.

Manuscript received October 15, 2020; accepted February 8, 2021. Date of publication February 25, 2021; date of current version April 8, 2021. This letter was recommended for publication by Associate Editor G. Neumann and Editor M. Vincze upon evaluation of the reviewers' comments. This work was supported by the European Community's Framework Programme Horizon 2020 under Grant 871704, project BACCHUS. *(Iason Sarantopoulos and Marios Kiatos contributed equally to this work.) (Corresponding author: Iason Sarantopoulos.)*

Iason Sarantopoulos and Zoe Doulgeri are with the Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece (e-mail: iasons@auth.gr; doulgeri@ece.auth.gr).

Marios Kiatos is with the Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece, and also with the Information Technologies Institute (ITI) Center of Research and Technology Hellas (CERTH), 57001 Thessaloniki, Greece (e-mail: mkiatos@auth.gr).

Sotiris Malassiotis is with the Information Technologies Institute (ITI) Center of Research and Technology Hellas (CERTH), 57001 Thessaloniki, Greece (e-mail: malasiot@iti.gr).

This article has supplementary downloadable material available at https://doi.org/10.1109/LRA.2021.3062295, provided by the authors.

Digital Object Identifier 10.1109/LRA.2021.3062295

Two main approaches can be used for developing a singulation policy. On the one hand, hand-crafted singulation methods use insights for the problem but tend to not generalize well. On the other hand, learning methods can generalize in different environments, with the cost of large data requirements for learning large state/action spaces. Our approach aims to combine both worlds by incorporating effectively prior knowledge into learning, through action primitives and through state representation selection, to increase the sample efficiency. Specifically, we propose a high level policy that learns to choose between those primitives, which can be either learned or designed. Our results demonstrate that our method can effectively solve the total singulation task in different environments.

In summary, this letter proposes a modular reinforcement learning method which uses continuous actions for singulating effectively a target object from its surrounding clutter to enable prehensile grasping. The proposed method:

1) outperforms considerably the existing works in terms of success rate,
2) allows to learn continuous pushing actions, unlike existing singulation works, which increases the available pushing options,
3) allows to combine different policies, which may be produced independently by means of reinforcement learning, supervised learning or algorithmic design.
4) uses effectively existing knowledge by shaping the action space and reducing the variability of the state representation in order to keep only task relevant information,
5) allows achieving singulation in different environments, like a surface with open limits (table) and a surface with walls on its limits (bin), which traditionally require different manipulation methods.

II. Related Work

Pushing can be used to facilitate prehensile robotic grasping of a target object in cluttered environments. Lynch and Mason [10] pioneered research on analytic models of push mechanics. Model-based approaches assume knowledge of the object pose and geometry in order to choose push actions. Dogar *et al.* [11] proposed a planning framework to reduce grasp uncertainty in cluttered scenes by utilizing push-grasping actions. However, they used known 3D models of objects and estimated the pose of each object in the scene.

Active singulation methods singulate every object in the scene through a sequence of interactions given the segmentation of the scene [12]–[15]. Boularias *et al.* [16] explored the use of reinforcement learning to learn push and grasp actions. However, the agent learns to singulate two specific objects and must be retrained for a new set of objects. In contrast to previous methods, our work is designed to be target oriented and thus more efficient in scenarios related to one specific object. Danielczuk and Kukernkov [17] choose between grasping, suction and pushing actions in order to extract a target object from a pile of unknown objects, given the segmentation of the scene. On the contrary, our approach does not require the segmentation of all objects in the scene but only the identification of the target.

Recent works, use deep reinforcement learning and pushing actions in order to render fully visible an occluded target [18], [19]. In contrast to singulation, achieving full visibility of the target does not always create free space between the target and surrounding obstacles, which is necessary for prehensile grasping. Zeng *et al.* [20] tackled the task of removing every object from a surface by grasping them and demonstrated that pushing actions, prior to grasps, can improve the grasp success rate when dealing with clutter. Specifically, they trained two fully convolutional networks, one for each manipulation primitive, in order to learn synergies between pushing and grasping. Yang *et al.* [21] adapted the work of [20] to extract a target object by learning to select between grasping and pushing actions in clutter, but assume only opposable grasps. On the contrary, we singulate the target object totally and thus enable grasping with a plethora of methods that require free space around the object to increase robustness on object uncertainties and perception errors. Finally, to produce more optimal policies, we take consideration of the workspace limits and their nature (open or constraining walls), which are ignored in [21]. Kiatos *et al.* [22] employ a pushing policy with discrete actions that is learned via Q-learning in order to singulate a target object from each surrounding clutter using depth features that approximate the topography of the scene. However, they made restricted assumptions for the height of the obstacles and the properties of the target object. Sarantopoulos and Kiatos [23] extended the work of [22] by modelling the Q-function with two different networks, one for each primitive action.

In contrast to [22], [23], which use discrete actions to singulate the target object, we use continuous actions which increases the agent's pushing options. Furthermore, [23] uses only a single policy (argmax between the Q-values of the discrete directions), while in the proposed method we have multiple policies: primitive policies and a high level policy which selects between those. The proposed architecture decouples the training of each primitive policy by allowing policies to be trained separately and in different ways. Finally, contrary to [23] we effectively incorporate prior knowledge by shaping the action space and reducing the variability of the state representation.

III. Method

The proposed method uses two pushing primitives, namely the push-target which pushes the target and the push-obstacle which pushes obstacles around the target. We also employ a high level policy that decides between each primitive. The high level policy selects a primitive only if it is valid, which means that it can change the current state of the scene. During training, we extract state representations both from visual observations and from the full state of the simulation for each primitive. During evaluation, we use only the visual state representations for easy transfer to the real world. The proposed architecture allows for modular training of the agent, which means that we train each primitive separately and then we train the high level policy to select between these primitives. This modular approach allows new primitives to be added (learned or handcrafted) with the need to retrain only the high level policy. The latter is demonstrated

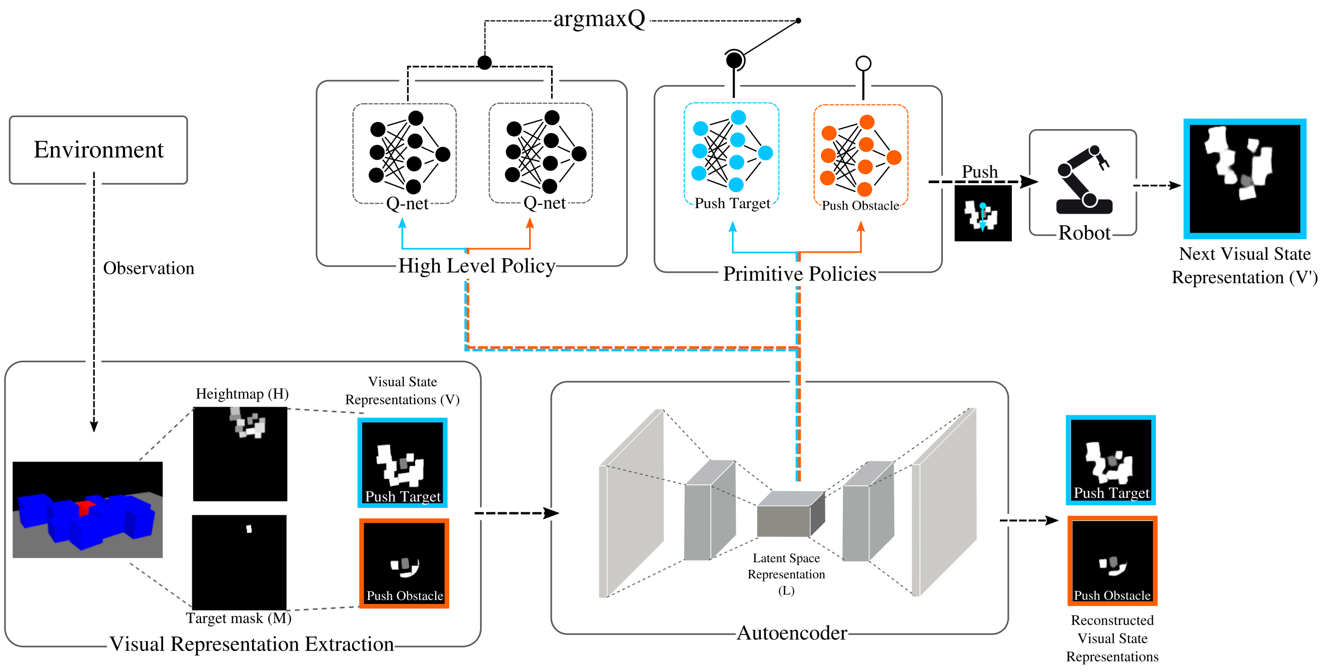


Fig. 2. Overview of the proposed system. The visual state representations V are generated given the scene's heightmap and the target's mask. An autoencoder (AE) is employed to reduce the dimensionality of each observation V. Then, the AE's latent vectors is fed into the high level policy which selects the primitive policy. The selected primitive policy outputs the push action to be executed by the robot for this timestep. Blue color highlights push-target related elements and orange color push-obstacle related elements.

experimentally by adding a sliding primitive for the case of a wall environment. An overview of the proposed method is depicted in Fig. 2.

A. *Problem Formulation*

We formulate the problem as an episodic Markov Decision Process (MDP) with time horizon T. An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$ of continuous states $\mathcal{S}$, continuous actions $\mathcal{A}$, rewards $\mathcal{R}$, an unknown transition function $\mathcal{T}$ and a discount factor γ. Our method focuses on sample efficient reinforcement learning by incorporating prior knowledge for the environment. Prior knowledge can be introduced by shaping the three spaces $\mathcal{A}$, $\mathcal{S}$, $\mathcal{R}$ of the MDP. Thus, instead of performing only reward shaping as most RL methods do, we also constrain the action space and the state representation in a way suitable for each primitive in order to increase the sample efficiency and the quality of the derived policy. The objective is to properly rearrange the cluttered scene in order to totally singulate the target object. Total singulation means that the distance between the target object and the closest obstacle is larger than d_{sing}.

1) Environment: Our environment consists of a rectangular workspace with predefined dimensions, a robotic fingertip to execute pushing actions and a RGBD camera which is able to capture RGB and depth information from a top view (both in simulation and reality). The pose, dimensions and the number of objects are random. Finally, we place the world frame to the center of the table with the z-axis being opposite to the gravity vector. Notice that we assume scenarios of dense clutter on table surfaces in which singulation is meaningful. Other types of clutter, such as a heap of objects can result to scenarios in which the target cannot be totally singulated.

2) Actions: The robot executes push actions by moving the robot finger along a line segment $[\boldsymbol{p}_1, \boldsymbol{p}_2] \in \mathbb{R}^2$ parallel to the support surface. The initial point $\boldsymbol{p}_1$ and the height from the table h, is calculated differently for each primitive as described in the following sections. The final point is determined by the angle θ with respect to x axis and the pushing distance d, which are learnable parameters, i.e. $\boldsymbol{p}_2 = \boldsymbol{p} + [d\cos(\theta), d\sin(\theta)]^T$, with $\boldsymbol{p}$ the position of the target.

3) States and Observations: In our training procedure we use both a full state representation, which is taken from the simulation, and a visual representation of the state, which is based on observations from a camera. Full state representation contains the exact poses of the objects and their corresponding bounding boxes. Regarding the visual representation, we use the mask M of the target object and the 2D heightmap H of the scene (Fig. 2). The mask M is produced by detecting the visible surface of the target object using color detection from an RGB image. This implies that the target can be differentiated from the obstacles by its color, an assumption which can be easily lifted using more advanced object detection techniques which are out of the scope of this work. The heightmap H is generated using the depth information by subtracting from each pixel the maximum depth. Note that we translate the centers of both images to the position of the target in order to produce translation-invariant features. We calculate the position of the target by calculating the centroid of the mask's convex hull, to account for arbitrary target geometries. Finally, we crop the

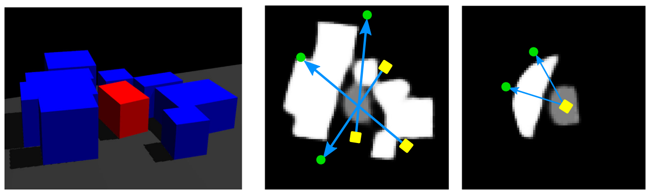


Fig. 3. The reconstructed by the autoencoder visual state representations produced for the same scene for the two primitives. Some possible pushes are depicted, with yellow squares denoting $\boldsymbol{p}_1$, green circles denoting $\boldsymbol{p}_2$ and blue arrows denoting the path of each push. (a) The simulated scene. (b) Push-target is designed to avoid the collision with obstacles in $\boldsymbol{p}_1$. Notice that $\boldsymbol{p}_1$ is not necessarily beside the target (c) Push-obstacle is "blind" to obstacles shorter than the target.

mask and the heightmap to keep only the relevant information for the predefined workspace.

To generate the visual state representation $V(H, M)$, we fuse the mask and the heightmap, according to:

$$V_{ij} = \begin{cases} 1, & \text{if } H_{ij} > 0, M_{ij} = 0, \\ 0.5, & \text{if } M_{ij} = 1, \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

with V_{ij} identifying the element of the matrix V with indices i, j. This means that pixels of V with value 1 belong to obstacles, with value 0.5 to the target object and with value 0 to the support surface.

For a final dimensionality reduction step we train an autoencoder on a large number of observations V (for more details see Appendix A). Then, for each input observation V we use the output of the trained encoder as the final visual feature $\boldsymbol{L}$. Notice that in contrast to [23], this work identifies the target object by its mask on the image plane, posing no constraints on the target object's 3D pose and bounding box or that the target object should never be flipped.

B. Push Target Primitive Policy

1) Action Execution: For the push-target primitive, the action is defined by the vector $a = [\theta, d]^T$, with θ the direction of the push and d the pushing distance, which are the learnable parameters. The initial position $\boldsymbol{p}_1$ is the closest to the target position which is not occupied by obstacles (Fig. 3), along the direction of θ. This can ensure that the finger will avoid undesirable collisions with obstacles (unlike [23]) during reaching the initial position. Nevertheless, during the execution of the pushing action, interaction with obstacles may occur but is not considered undesired. To implement this obstacle avoidance, we are using the visual observation V. Specifically, we check if a patch of pixels, starting from the center of the heightmap and moving along the direction θ, includes only pixels from the table or not. The first position that this happens is the position $\boldsymbol{p}_1$. Note that the size of the patch depends on the size of the finger. The initial height that the push starts is calculated to be halfspace between the table and the top of the target. By hand-engineering the obstacle avoidance process, we significantly reduce the action space and we guide the agent to focus only on the singulation task, which is shown in Section IV-B to improve the overall performance. Finally, notice that the push-target primitive is always valid, because there are no states that are guaranteed to lead to an empty push, as the target object will be always moved.

2) State Representation: For push-target primitive, we use both the visual and the full state representation. The visual observation is extracted as described in Section III-A3. At the end of the latent vector $\boldsymbol{L}$, we append the distances of the target object from the four edges of the support surface (which can be calculated given the position of the target object). This information is essential to learn how to avoid pushing the target off the surface's limits.

Regarding the full state representation $\boldsymbol{s}$, we exploit the access to the full state of the system, since we train the agent in simulation. Specifically, we represent each object i with its pose, which consists of the 3D position $\boldsymbol{t}_i \in \mathbb{R}^3$ defined by the bounding box's centroid and the corresponding orientation defined by the quaternion $\boldsymbol{q}_i \in \mathbb{R}^4$, and its 3D bounding box $\boldsymbol{b}_i \in \mathbb{R}^3$. Thus, the state vector for each object is $\boldsymbol{s}_i = [\boldsymbol{t}_i, \boldsymbol{q}_i, \boldsymbol{b}_i] \in \mathbb{R}^{10}$. Note that we discard the objects that lie outside the cropped area in order to match the visual state representation. Finally, we concatenate the features from each object and we end up with the full state representation $\boldsymbol{s} = [\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n_{max}}] \in \mathbb{R}^{n_{max} \times 10}$, with n_{max} the maximum number of objects. The state vector of the first object correspond always to the target. The rest of the state vectors, corresponding to the obstacles, are sorted with respect to the distance of each obstacle from the target object in order to reduce the variability in state.

3) Training: In contrast to [23] we use continuous actions to solve the singulation task with an actor-critic algorithm [24]. This approach allows learning a more effective singulation policy by reducing the number of push actions required. Inspired by [25], we exploit the fact that we train on the simulation, by using the full state representation to train the critic and the corresponding visual state representation to train the actor. This asymmetry in the inputs of the two networks, allows the critic to approximate the Q-value more accurately and faster, as it learns on the fully observable state and thus the actor's accuracy is positively affected, since it is optimized based on critic's predictions. During testing, Q-value predictions are not needed, so the critic is discarded.

In this letter, we use the Deep Deterministic Policy Gradient (DDPG) [26] as the actor-critic algorithm. At every timestep we perform an explorative action and an image of the scene is rendered from the simulator at the full state in order to estimate the visual state representation as described in the Section III-A3. For each pushing action, we assign a reward of $r = -0.1$ if the push increases the average distance off the target's bounding box from the surrounding bounding boxes and $r = -0.25$ in any other case. In case the target is singulated we reward with 1 and in case the target falls of the table we penalize with -1. Note that only the objects that lie inside the singulation area are taken into consideration in the average distance calculation. We then store the transition to the replay buffer R, which is initialized with 1000 transitions to decrease training time. The episode is considered successful if the target object is singulated and failed if the target falls off the table or a maximum number of actions is reached. Notice that as the size of the finger changes between different robots, the push-target policy may need retraining e.g. when the finger size is larger than the one used during training and the policy results to collisions.

C. Push Obstacle Primitive Policy

1) Action Execution: For the push-obstacle primitive the action is defined from the direction of the push, i.e. $a = \theta$, with the angle θ being the only learnable parameter. For executing this primitive, the initial position $\boldsymbol{p}_1$ is placed above the target object, given its position and height calculated by the heightmap (Fig. 3(c)). The final position $\boldsymbol{p}_2$ is calculated from the angle θ, which is learned, and the fixed pushing distance $d = d_{sing} + b$, where b is the maximum size of the target object. This means that the fingertip can be moved on a disc with radius d placed above the target object. Thus, the robot can interact only with obstacles which intersect this disc, i.e. closer to target than d and taller than the target. We call these obstacles accessible. If no accessible obstacles exist, this primitive will be invalid as it will result to an empty push.

2) State Representation: In order to produce the representation V for the push-obstacle primitive, we ignore obstacles which are not accessible, to reduce unnecessary variability. Firstly, we discard the pixels of the heightmap H with distance from the center of the heightmap, larger than the pushing distance d. Secondly, we produce V as described above, with the only difference that we set the value of each pixel to 1 if the corresponding pixel of H is above the target's height (not just above zero), to exclude from V the pixels corresponding to the shorter obstacles (see Fig. 3(c)). Hence, Eq. (1) becomes:

$$V_{ij} = \begin{cases} 1, & \text{if } M_{ij} = 0, H_{ij} > H_{c_i,c_j}, |p - c| < d, \\ 0.5, & \text{if } M_{ij} = 1, \\ 0, & \text{otherwise} \end{cases}$$

with $c = [c_i, c_j]^T$ the center point of the heightmap and $p = [i, j]^T$.

3) Training: A push-obstacle policy can singulate the target by performing a number of pushes of the surrounding obstacles. However, the sequence to perform these pushes is, in the majority of the cases, not important to accomplish the task, unlike push-target, and thus singulation with push-obstacle is not a sequential decision problem, which means that we can simply train the policy in a supervised manner, instead of RL. The task is to learn a mapping from a latent space vector $\boldsymbol{L}$ to pushing direction θ_g. For each accessible obstacle k of the scene, we calculate the angles $\theta_k = arctan(y_k/x_k)$, where (x_k, y_k) the position of the accessible obstacle acquired from the full state representation from the simulation. Notice that the exact position is needed to ensure that the ground truth policy pushes the obstacle through its center of mass. The ground truth angle is selected as $\theta_g = \min_k \theta_k$. Finally, because we regress on angles we use as the training loss the cosine of the absolute error: $\mathcal{L}_{obs} = 1 - \cos(|\hat{\theta} - \theta_g|)$ where $\hat{\theta}$ is the predicted angle. This loss takes into account that a prediction differing from the ground truth by a multiple of 2π will result to the same push.

D. High Level Policy

The purpose of the high level policy is to combine the above primitives to effectively singulate the target object. For the high level policy we use the SplitDQN algorithm, presented in [23]. The SplitDQN splits the vanilla Q-network (DQN) into several Q-networks, one network per primitive, so as to be more sample efficient than DQN. In our case, each Q-network receives as input the latent vector of the corresponding visual state representation of each primitive, as described above, and outputs the predicted Q value of using this primitive policy. The primitive with the greatest Q value is selected and executed according to the corresponding learned primitive policy from the previous sections. The validity of the primitive is taken into account during exploration and thus the exploration policy chooses between the valid primitives for each scene, which in our case means that if no accessible obstacles exist for the push-obstacle, the agent will select push-target. Exploring only for valid primitives increases further the sample efficiency as the agent stores only experience samples which contain useful information. An empty push-obstacle contains zero information for the policy. More implementation details can be found in Appendix A.

IV. Experiments and Results

To test the proposed method we executed a series of experiments both in simulation and in the real world.[1] The objectives of the experiments are: 1) to investigate if the proposed architecture improves the derived policy compared to the existing methods, 2) to test if the design choices contribute to the increased quality of the final policy, 3) to demonstrate the benefits of the method's modularity and 4) to demonstrate the robust transfer to a real robotic system without unjustifiable drop in the success rate.

A. Comparison to Baselines

We compare the proposed method for total singulation of the target object (**TSO**) with three baseline methods: the SplitDQN (**SDQN**) proposed in [23] which uses similar primitives with discrete actions, the "grasping the invisible" approach proposed in [21] (**GTI**) which coordinates between pushes and grasps and a random baseline (**Random**) which uses the proposed primitives and selects actions randomly. We train and evaluate all three agents in environments simulated with the MuJoCo [27] physics engine. For a fair comparison we evaluated in environments that are similar to those that described in the aforementioned papers. To evaluate the agents we run 1000 episodes, where we measure both the success rate and the number of actions required for a successful run. An episode is considered successful if the target object is singulated or grasped within 15 timesteps. In any other case (end of maximum timesteps or fall out of the surface) the episode is considered as failed.

In particular, first we train and evaluate **TSO** and **SDQN** in a environment similar to the one presented in [23]. Specifically, each scene is randomly generated with the randomized parameters being: the number of obstacles (8-13), the bounding boxes

[1]Source code and videos can be found in: https://robot-clutter.github.io

of the objects ($[1, 1, 0.5]^T$ cm the minimum and $[3, 3, 0.02]$ cm the maximum) and the probability of generating every obstacle with the same height (20 %). **TSO** outperforms the **SDQN** with a success rate of 93.9% with 3.68 average actions before singulation, while **SDQN** accomplishes 83.4% with 3.19 average actions. The difference in success rate can be justified from the fact that **TSO** learns better to avoid throwing the target off the table due to the continuous action space. Implementations details for the training of **TSO** can be found in Appendix A.

Finally, for comparing with **GTI** we simulate a MuJoCo environment similar to the one presented in [21]. This environment has 7 objects, with fixed size obstacles of $[3, 2]^T$ cm and variable height between 0.5 and 2 cm. In [21] the only successful terminal state for **GTI** is grasping the target object. In our case, we have two successful terminal states for **GTI**: grasping the target and total singulation. **TSO** achieves 90.5% with 3.04 average steps and **GTI** 75.4% with 7.66 average actions before a successful terminal state is reached.

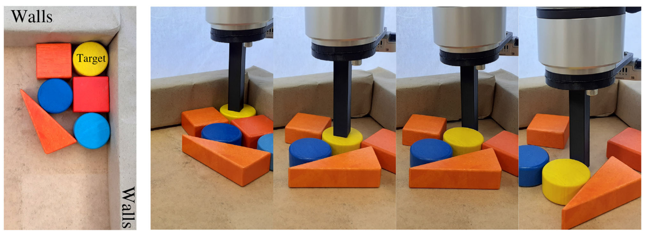
B. Contribution of Design Choices

In order to determine whether certain design choices affect the final policy we run two different experiments. First, we test whether the use of asymmetry in actor-critic (Section III-B3) results to higher success rate. We train the push-target Actor-Critic with and without asymmetry, until convergence. In the first case we use the full state representation as input to the critic network (**Asym**) and in the second the visual state representation (**No-asym**). By evaluating the two policies in 1000 episodes we result in 89.8% success rate for using asymmetry and 72.4% for not using asymmetry, which supports the importance of using the full state representation as input to the critic.

To evaluate the enforcement of collision-free $\boldsymbol{p}_1$ in the push-target primitive, we train the agent by modifying the push-target execution. Specifically, the agent needs to learn the initial distance d_{init} that the push starts, alongside with the angle θ and the pushing distance d, resulting to success rate 44.5%. This means that the agent fails to learn a valuable singulation policy when it also needs to learn to avoid obstacles. This failure can be attributed to the initial sparse rewards that the agent receives because of the high number of collisions during exploration.

C. Effectiveness of the Architectural Modularity

To demonstrate the effectiveness of the proposed architecture's modularity and its effectiveness on different environments, we complicate the problem by adding walls around the surface. This means that there are system states in which both push target and push obstacle primitives are invalid (they are unable to change the scene). This is because the walls operate as fixed obstacles which cannot be moved by the robot. Such states emerge when the target object is stuck alongside a wall as shown in Fig. 4(a). In such cases, using push-target will result either to a collision between the finger and the wall or to pushing the target against the wall. On the other hand, using push-obstacle will result in either an empty push or a collision of the finger with the wall. Hence, we add a new primitive in which the robotic finger exerts tangential forces on top of the target object in order to slide



(a) (b)

Fig. 4. (a) Environment with walls. (b) The sliding primitive. The robot slides the target away from the walls followed by a push towards the center of the surface.

TABLE I
RESULTS

Policy	**Env**	**Success rate %**	**Mean actions**	**Std actions**
TSO		**93.9**	3.68	1.99
SDQN [23]		83.4	3.19	**1.43**
Random	Env-1	13.7	4.64	2.21
Asym		89.8	**2.93**	1.77
No-Asym		72.4	2.96	1.85
TSO	Env-2	**90.5**	**3.04**	**2.30**
GTI [21]		75.4	7.66	3.73
TSO-walls-3	Env-3	**86.2**	3.88	2.87
TSO-walls-2		76.6	**3.12**	**2.10**
TSO	Real	82.5	3.42	2.01

Details for each environment can be found in the Appendix. Bold indicates the best result in each environment.

it. The direction of the primitive is always towards the center of the table, which ensures that the target will be moved away from the walls, creating sufficient space for using a subsequent push target action Fig. 4(b). Finally, this primitive is considered valid only when the target is close to the walls, as there is no need for the robot to attempt a risky sliding motion, unless is absolutely necessary.

We evaluate the pretrained agent of the two primitives in this new environment, resulting to 76.6% success (**TSO-walls-2**). Then, we add the sliding primitive. The approach allows us to add the sliding primitive by training the high level policy without re-training the low level policies in the new environment (**TSO-walls-3**). Training the SplitDQN with a third network in this environment results to 86.2% success rate with 3.88 average actions before singulation. The comparison between TSO-walls-3 and TSO-walls-2 demonstrates the utility of the new simple primitive when the target object lies close to the walls. A summary of the comparisons described above can be found in Table I.

D. Transfer to the Real World

We ran a series of experiments to evaluate our method on a real robot. Our real world setup consists of a UR5e robotic arm with a wrist-mounted Intel RealSense D415 camera (as shown in Fig. 1). The object set consists of toy blocks and novel objects. We performed 40 experiments, 30 with the toys blocks and 10 with the household objects, by placing them in random and adversarial positions. Randomness is achieved by placing the objects on a box, shaking it and emptying it

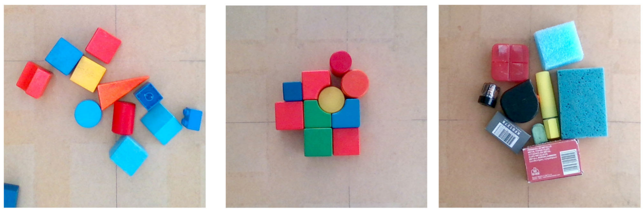

(a) Random scenes (b) Adversarial (c) Novel objects

Fig. 5. Samples of the real scenes in which the method was evaluated.

on the workspace. Examples of the initial configurations can be seen in Fig. 5. The conducted experiments consist of the following steps. At first, the object detection algorithm runs on an acquired RGB-D image, which in our case is a color detector, and the visual features are generated as described in Section III-A3. Given the visual state representations the high level policy selects between the push primitives and the robot executes the action. The episode terminates if the target object is totally singulated, fallen off the workspace limits or a maximum number of timesteps is reached. The learned policy is robustly transferred to a real world environment thanks to robust feature selection, with a 82.5% success rate and 3.42 average actions. The difference in success rate is accounted to the discrepancy in physics between simulation and real environment and also in shapes and sizes between the objects.

V. Conclusions and Future Work

In this letter, we addressed the problem of totally singulating a target object from its surrounding clutter by proposing a modular reinforcement learning framework that incorporates prior knowledge into learning through action primitives and state representation selection. Results demonstrate that the method's modularity can be used to solve singulation tasks by combining the already trained primitive policies with a newly added primitive and that the learned policy can be robustly transferred to a real world environment. Future research targets of this work include the evaluation of the policy for objects of non-convex geometry.

Appendix A
Implementation Details

A. *Autoencoder*

In this section we provide more details on our training setup. Firstly, we train a convolutional autoencoder to reduce the dimensionality of the input state. Specifically, the encoder has four convolution layers, each of which has a bank a 4×4 filters with a stride of 2 and followed by a ReLU activation function. The number of output channels starts with 16 and doubles at each subsequent layer ending up with 128. Thus, we embed each input image to a 256 dimensional latent spaces. The decoder is composed by four up-convolutional layers which are followed by ReLU activation functions. To generate training data we generate scenes in MuJoCo and we randomly execute push-target actions, collecting 10 000 visual state observations V. Then, we crop each observation V to size 256×256 and perform an average pooling step with a kernel $[2, 2]$ and stride 2 to reduce dimensionality. We further perform a data augmentation step by rotating each obserevation 16 times and we end up with 160 K training data. The model is trained using an Adam optimizer with batch 32 for 70 epochs and the learning rate is set to 10^{-4}.

B. *Primitive Policies*

The push-target primitive is trained using the asymmetric trick optimized with the DDPG algorithm. Specifically, the actor and the critic are both 2-layered fully-connected with $400, 300$ hidden units. The actor's output layer uses a tanh activation for enforcing the output in $[-1, 1]$ range. In order to avoid saturation of the the actor's output we add to the actor's loss the following term p, to penalize large outputs outside $[-1, 1]$:

$$p = \begin{cases} 0, & \text{if } |\bar{y}| < 1 \\ 0.05(|\bar{y}| - 1)^2, & \text{otherwise} \end{cases}$$

where $\bar{y}$ the mean output of the layer before the last activation of tanh. The target actor and critic are updated with a polyak averaging of 0.999. Both networks are trained using the Adam optimizer with learning rate 10^{-3}. We use a discount factor of $\gamma = 0.9$. For efficient learning, we normalize the input states i.e. the concatenation of the latent vector and the distances of the target from the table limits. The replay buffer is preloaded with 10^3 transitions, produced by a random policy sampling actions from a uniform distribution. We sample batches consisting of $K = 32$ samples. Finally, the behavioural policy chosen for exploration consists of the output of the actor added with coordinate independent Normal noise with standard deviation equal to 5% of the action range.

Regarding the push obstacle primitive, we collect a dataset of 20 000 pairs of latent vectors/ground truth pushes to train a 2-layered fully-connected with $400, 300$ hidden units, with learning rate 10^{-3}, batch size 8 and loss function $1 - \cos(|\hat{\theta}| - \theta_g)$. Finally, the high level policy is learned with SplitDQN [23] using the learned actors of push-target and push-obstacle as discrete actions, to train two fully connected critics with $400, 300$ hidden units each, learning rate 10^{-3}, batch size 32 and $\gamma = 0.99$.

C. *High Level Policy*

The goal of the high level policy is to choose among the available pushing primitives. To this end, the SplitDQN algorithm [23] is utilized to learn the Q-function for each primitive policy. Specifically, each Q-function is modelled with a feedforward fully connected network, ϕ_τ and ϕ_o, corresponding to each of the pushing primitive policies, push target π_τ and push obstacle π_o. Each network takes as input the associated latent state representation $\boldsymbol{L}_\tau$ and $\boldsymbol{L}_o$ as described in Section III-A along with the action produced by the corresponding learnt primitive policy and outputs a Q-value. The maximum Q-value is given by:

$$\max Q = \max\left(\phi_\tau(\boldsymbol{L}_\tau, \pi_\tau(\boldsymbol{L}_\tau)), \phi_o(\boldsymbol{L}_o, \pi_o(\boldsymbol{L}_o))\right) \quad (2)$$

During training we use two replay buffers (one per primitive). In each timestep, we perform an explorative action u_t and we store the transition to the corresponding replay buffer. Then, we sample a minibatch of K stored transitions $(\boldsymbol{L}_k, r_k, u_k, \boldsymbol{L}_k^{next})$ from this replay buffer and we train the corresponding network by minimizing the mean-squared error loss function

$$\mathcal{L} = \frac{1}{K}\sum_{k=1}^{K}(\phi_k^{\omega}(\boldsymbol{L}_k, \pi_k(\boldsymbol{L}_k)) - y_k)^2$$

$$\text{with: } y_k = r_k + \gamma\phi_k^{-\omega}(\boldsymbol{L}_k, \pi_k(\boldsymbol{L}_k^{next}))$$

where ω are the parameters of the corresponding primitive network for this timestep and ω^- are the parameters of its target network. A similar strategy is followed if a new primitive is added.

D. Environment

For the environment we use singulation distance $d_{sing} = 3$ cm, a spherical fingertip of radius 0.5 cm, support surface 50×50 cm. The pushing distance for the push-target is in the range $[2, 10]$ cm, for the push-obstacle is fixed to 6 cm and for the sliding primitive and finally the sliding motion is 3 cm followed by a 6 cm push. The different environments used in our simulation experiments are listed below:

1) **Env-1**: number of obstacles (8-13), the bounding boxes of the objects ($[1, 1, 0.5]^T$ cm the minimum and $[3, 3, 0.02]$ cm the maximum) and the probability of generating every obstacle with the same height (20 %), without walls.
2) **Env-2**: 7 objects, with fixed size obstacles of $[3, 2]^T$ cm and variable height between 0.5 and 2 cm, without walls.
3) **Env-3**: Env-1 with walls on the surface limits.

References

[1] J. Bohg, A. Morales, T. Asfour, and D. Kragic, “Data-driven grasp synthesis-a survey,” *IEEE Trans. Robot.*, vol. 30, no. 2, pp. 289–309, Apr. 2014.
[2] I. Lenz, H. Lee, and A. Saxena, “Deep learning for detecting robotic grasps,” *Int. J. Robot. Res.*, vol. 34, no. 4-5, pp. 705–724, Apr. 2015.
[3] M. Kiatos and S. Malassiotis, “Grasping unknown objects by exploiting complementarity with robot hand geometry,” in *Proc. Int. Conf. Comput. Vis. Syst*. Berlin, Germany: Springer, 2019, pp. 88–97.
[4] M. Kiatos, S. Malassiotis, and I. Sarantopoulos, “A geometric approach for grasping unknown objects with multifingered hands,” *IEEE Trans. Robot.*, 2020, pp. 1–12, doi: 10.1109/TRO.2020.3033696.
[5] C. Eppner and O. Brock, “Grasping unknown objects by exploiting shape adaptability and environmental constraints,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2013, pp. 4000–4006.
[6] M. A. Roa, M. J. Argus, D. Leidner, C. Borst, and G. Hirzinger, “Power grasp planning for anthropomorphic robot hands,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2012, pp. 563–569.
[7] I. Sarantopoulos and Z. Doulgeri, “Human-inspired robotic grasping of flat objects,” *Robot. Auton. Syst.*, vol. 108, pp. 179–191, Oct. 2018.
[8] C. Eppner, R. Deimel, J. Álvarez-Ruiz, M. Maertens, and O. Brock, “Exploitation of environmental constraints in human and robotic grasping,” *Int. J. Robot. Res.*, vol. 34, no. 7, pp. 1021–1038, Jun. 2015.
[9] F. Angelini, C. Petrocelli, M. G. Catalano, M. Garabini, G. Grioli, and A. Bicchi, “Softhandler: An integrated soft robotic system for handling heterogeneous objects,” *IEEE Robot. Automat. Mag.*, vol. 27, no. 3, pp. 55–72, Sep. 2020.
[10] K. M. Lynch and M. T. Mason, “Stable pushing: Mechanics, controllability, and planning,” *The Int. J. Robot. Res.*, vol. 15, no. 6, pp. 533–556, 1996.
[11] M. R. Dogar and S. S. Srinivasa, “Push-grasping with dexterous hands: Mechanics and a method,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2010, pp. 2123–2130.
[12] T. Hermans, J. M. Rehg, and A. Bobick, “Guided pushing for object singulation,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2012, pp. 4783–4790.
[13] L. Chang, J. R. Smith, and D. Fox, “Interactive singulation of objects from a pile,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2012, pp. 3875–3882.
[14] A. Eitel, N. Hauff, and W. Burgard, “Learning to singulate objects using a push proposal network,” in *Robotics Research*. Berlin, Germany: Springer, 2020, pp. 405–419.
[15] M. Danielczuk, J. Mahler, C. Correa, and K. Goldberg, “Linear push policies to increase grasp access for robot bin picking,” in *Proc. IEEE 14th Int. Conf. Automat. Sci. Eng.*, 2018, pp. 1249–1256.
[16] A. Boularias, J. A. Bagnell, and A. Stentz, “Learning to manipulate unknown objects in clutter by reinforcement,” in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 1336–1342.
[17] M. Danielczuk *et al.*, “Mechanical search: Multi-step retrieval of a target object occluded by clutter,” in *Proc. Int. Conf. Robot. Automat.*, May 2019, pp. 1614–1621.
[18] A. Kurenkov *et al.*, “Visuomotor mechanical search: Learning to retrieve target objects in clutter,” in *Proc. IEEE Int. Conf. Intell. Robots Syst.*, 2020.
[19] T. Novkovic, R. Pautrat, F. Furrer, M. Breyer, R. Siegwart, and J. Nieto, “Object finding in cluttered scenes using interactive perception,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2020, pp. 8338–8344.
[20] A. Zeng, S. Song, S. Welker, J. Lee, A. Rodriguez, and T. Funkhouser, “Learning synergies between pushing and grasping with self-supervised deep reinforcement learning,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, pp. 4238–4245, Oct. 2018.
[21] Y. Yang, H. Liang, and C. Choi, “A deep learning approach to grasping the invisible,” *IEEE Robot. Automat. Lett.*, vol. 5, no. 2, pp. 2232–2239, Apr. 2020.
[22] M. Kiatos and S. Malassiotis, “Robust object grasping in clutter via singulation,” in *Proc. Int. Conf. Robot. Automat.*, May 2019, pp. 1596–1600.
[23] I. Sarantopoulos, M. Kiatos, Z. Doulgeri, and S. Malassiotis, “Split deep q-learning for robust object singulation,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2020, pp. 6225–6231.
[24] V. R. Konda and J. N. Tsitsiklis, “Actor-critic algorithms,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 1008–1014.
[25] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, “Asymmetric actor critic for image-based robot learning,” in *Proc. Robotics: Sci. Syst.*, Pittsburgh, PA, USA, Jun. 2018, doi: 10.15607/RSS.2018.XIV.008.
[26] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,” *Int. Conf. Learn. Representations*, 2016.
[27] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A. physics engine for model-based control,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2012, pp. 5026–5033.