Learning to Cooperate: A Hierarchical Cooperative Dual Robot Arm Approach for Underactuated Pick-And-Placing

Learning to Cooperate: A Hierarchical Cooperative Dual Robot Arm Approach for Underactuated Pick-And-Placing

Sander De Witte[1,2], Thijs Van Hauwermeiren[1,2], Tom Lefebvre[1,2] and Guillaume Crevecoeur[1,2]
[1]Department of Electrical Energy, Metals, Mechanical Constructions & Systems, Ghent University
Tech Lane Ghent Science Park 913, B-9052 Zwijnaarde, Belgium
[2]Core Lab EEDT Decision & Control, Flanders Make Strategic Research Centre for the Manufacturing Industry

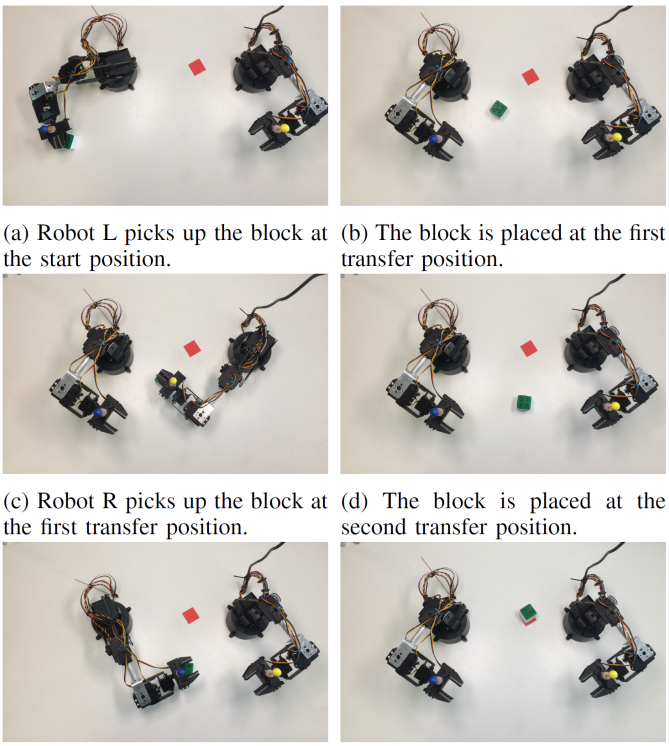
***Abstract*—Cooperative multi-agent manipulation systems allow to extend on the manipulative limitations of individual agents, increasing the complexity of the manipulation tasks the ensemble can handle. Controlling such a system requires meticulous planning of subsequent subtasks, queried to the individual agents, in order to execute the master task successfully. Real-time planning is essential to ensure the task can still be achieved when subtasks execution suffers from uncertainty or when the master task changes intermittently requiring real-time reconfiguration of the plan. In this work we develop a supervisory control architecture tailored to the cooperation of two robotic manipulators equipped with standard pick-and-place facilities in the plane. We consider a toy task description where we control the planar position and orientation of an object. A time-invariant policy function is trained using deep reinforcement learning, which can determine a finite sequence pick-and-place maneuvers to manipulate the object into its desired configuration. A comparison is made between two strategies, with the distinction made based on different treatments of the final step. The more information is given to the policy the easier it trains. In return, it becomes less adaptable and loses some of its generalizability.**

***Index Terms*—cooperation, robotics, reinforcement learning, multi-agent manipulation, pick-and-place**

I. Introduction

DESIGNING a single autonomous robot, adaptable to all circumstances is not attainable or at least not an economical use of resources. Instead, complex tasks are better handled by a combination of multiple standard robots that cooperate [1]. The benefits of cooperation between agents are described by Tan [2] amongst others. For example, manipulation tasks with a higher complexity can be performed by using cooperative multi-agent manipulation systems and thus overcoming the manipulative limitations of individual agents. This trend of cooperation can be seen in a variety of robotic applications, such as multi-UAV control systems [3], heterogeneous multi-robot systems combining aerial and mobile robots [4], manipulation tasks [5] or assembly operations [6]. The cooperation of robots, among themselves and with humans, significantly increases the spectrum of automated tasks. The downside is clearly that the complexity of coordinating and controlling these systems also increases. Meticulous planning of subsequent tasks, queried to the individual agents, is required in order to have a successful execution of the master task.

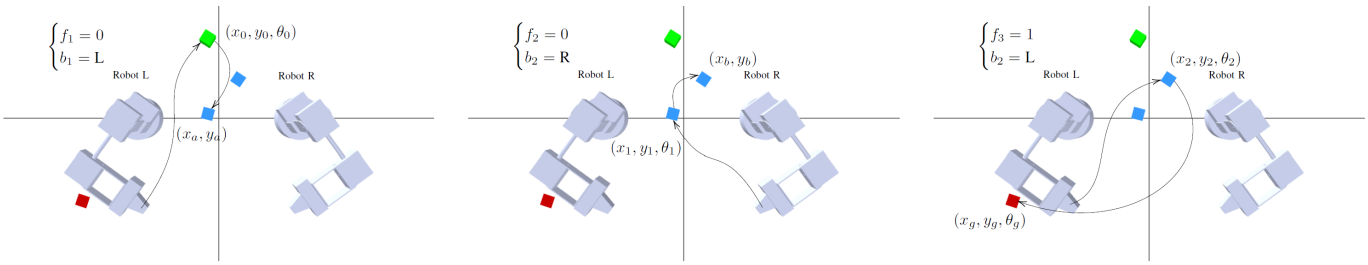
This research received funding from the Flemish Government (AI Research Program) and the Flanders Make (Multi Systems Learning Control).



(a) Robot L picks up the block at the start position.
(b) The block is placed at the first transfer position.
(c) Robot R picks up the block at the first transfer position.
(d) The block is placed at the second transfer position.
(e) Robot L picks up the block at the second transfer position.
(f) Final orientation and position close to the desired goal.

Fig. 1: A successful transfer of the block (green) to the goal configuration (red) is achieved on the experimental set-up.

We consider control and planning of manipulation tasks in a hierarchical sense, with a high level planner that is agnostic to the low level control. Information from each agent is known by the planner, resulting in a centralized control policy which enables collaborative agents. This problem is closely related to task and motion planning (TAMP) [7]. TAMP methods often break up sequential manipulation problems into a high- and a low-level planner. The high-level decision maker describe the pre- and post-conditions of some symbolic abstractions or mo-



(a) First action $a_1 = (0, \mathrm{L}, x_a, y_a)$ (b) Second action $a_2 = (0, \mathrm{R}, x_b, y_b)$ (c) Final action $a_3 = (1, \mathrm{L}, x_g, y_g)$

Fig. 2: An example of a possible action sequence resulting in the final configuration of the object (red) starting from a random initial configuration (green), with the intermediary configurations shown in blue. The low level control is concerned with the motion planning, performed by the individual robots, visualized with the black arrows. The supervisory control determines which action is taken next, revealing the hierarchical control architecture.

tion primitives, e.g. the pick-up and drop-off position of a pick-and-place. The low-level planner is concerned with the motion planning [8]. In [9] a task and motion planning technique is proposed for multi-robot systems, introducing a high-level task planning and a low level motion planning layer. Standard TAMP methods optimize both the action sequence and the motion plan, whereas in this work we are only concerned with optimization of the action sequence. A supervisory control problem should then only query subtasks leaving execution to the individual robots. This control decomposition allows to significantly reduce the associated optimization problem benefiting real-time computation. In that way the system can be reactive to changes in the environment, a change of goal state or simply poorly executed subtasks.

We consider a particular manipulation problem formulation where two individual manipulators share a subset of their individual workspaces. We are motivated by the question of how this shared workspace can be used by the agents to collaborate in such a sense that they are able to execute complex tasks, exceeding their individual capabilities. In this work, a setup consisting of two robotic manipulators tasked with maneuvering an object into an arbitrary goal configuration (position and orientation) in the global workspace is introduced. By working together arbitrary goal configurations can be attained through multiple, collaborative actions in the shared workspace, illustrated in Fig. 2. To this end, a policy is determined that outputs the optimal sequence of action (including which robot to perform the manipulation) to achieve the goal. In order to cope with dynamic changes in the environment (change of goal configuration or poor executed subtasks) in real-time, a time-invariant policy is pursued that depends only on the instantaneous object configuration and the desired goal configuration. Such a policy is obtained using deep reinforcement learning (DRL) [10]. DRL algorithms are applied in a number of robotic applications, to perform motion planning for robotic manipulators [11], to perform complex (dexterous) manipulation [12], [13], to train multiple agents [14], or to train end-to-end policies from an RGB camera image to control [15]. In this work, we apply DRL on the system level to find a policy that determines the next action to be performed by a certain robot.

The policy is trained in simulation and tested on the real setup, indicating a *Sim2Real* transfer. *Sim2Real* transfer is used in a number of manipulation problems, e.g. to solve multi-agent manipulation through locomotion [16] or to learn obstacle avoidance in uncertain environments for robotic manipulators [17]. Combining physical simulation with deep learning techniques shows to be advantageous, since the large quantity of data needed for deep learning can be produced efficiently in simulation. Furthermore, it is safer, takes less time, and it is inexpensive [18], [19]. The validity of our solution is proven on an experimental set-up (see Fig. 1).

The novelty of this paper is the combination of a supervisory control system with learning methods, resulting in a real-time policy able to adapt to changes in a dynamic environment. The method is proven to work on an experimental setup.

II. Problem Statement

In this section, the considered small-scale setup is described together with the associated manipulation problem. As shown in Figures 1, 2 and 6, the set-up consists of a work cell equipped with two symmetrical robotic manipulators and a vision system that can identify and locate objects in the workspace. Both robotic manipulators, henceforth referred to as the *left* (L) and *right* (R) manipulator, have five degrees of freedom (DOFs). Each individual manipulator is equipped with a lower-level open-loop controller and is capable of executing pick-and-place maneuvers in the base XY-plane. We will refer to this plane as the manipulation plane. Two DOFs are used to control the pitch and roll of the end-effector. The remaining three DOFs are used to control the end-effector's position in the manipulation plane, that is its XY-coordinates with a fixed height. As a result it is impossible to control the end-effector's yaw, resulting in an underactuated manipulation problem. Because, the yaw of any manipulated object cannot be controlled, its final orientation will depend on the respective pick-up and drop-off positions and the morphology of the manipulators. We refer to the individual manipulation spaces as $\mathcal{M}_{\mathrm{L}} \subset \mathbb{R}^2$ or $\mathcal{M}_{\mathrm{R}} \subset \mathbb{R}^2$ respectively, which are defined as the controllable set of end-effector configurations in the manipulation plane expressed as Cartesian coordinates with respect to a global frame of reference. Equivalently the manipulation spaces are determined as the intersection of the individual manipulator workspaces and the manipulation

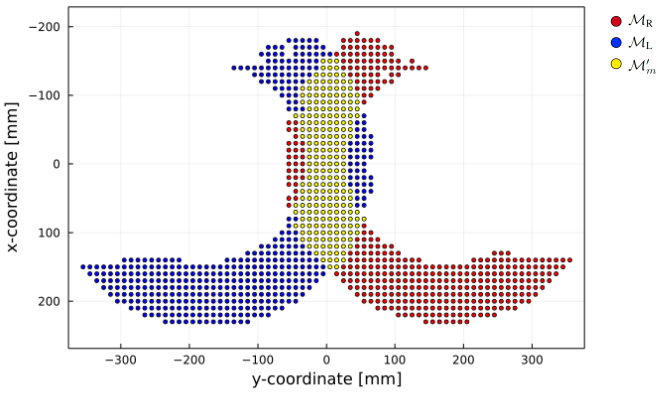


Fig. 3: Illustration of the discretized manipulation spaces ($\mathcal{M}_\mathrm{R}$ and $\mathcal{M}_\mathrm{L}$) with mutual manipulation space as the intersection in yellow ($\mathcal{M}'_m$) for the small-scale setup shown in Fig. 6.

plane. The feasible workspaces are attained as the collection of points that are collision-free, including self collision and collision with the ground. The *global*, $\mathcal{M}$, and *mutual*, $\mathcal{M}_m$, manipulation spaces are defined respectively as the union and intersection of the individual ones (Fig. 3).

$$\mathcal{M} = \mathcal{M}_\mathrm{L} \cup \mathcal{M}_\mathrm{R}$$
$$\mathcal{M}_m = \mathcal{M}_\mathrm{L} \cap \mathcal{M}_\mathrm{R}$$

We consider a pick-and-place task where objects are placed within the global manipulation space $\mathcal{M}$ and need to be repositioned to a new and given goal position, $(x_g, y_g) \in \mathcal{M}$. In our present context a goal orientation, θ_g, of the objects is also specified. Adding the orientation to the manipulation space results in the observation space $\mathcal{S}$, since the orientation is only observable and uncontrollable. The manipulators ought to respect the desired orientation to complete the manipulation task successfully. The morphology and low-level steering of the individual manipulators is such that they cannot execute the pick-and-place task separately but must decide on an optimal cooperation strategy to achieve the desired object configuration

$$\mathbf{s}_g = (x_g, y_g, \theta_g) \in \mathcal{S} = \mathcal{M} \times \mathbb{R}$$

The objective of this work is to find a supervisory control system that is capable of querying pick-and-place maneuvers to the individual manipulators for manipulating objects from an arbitrary starting position and orientation to a given desired configuration. A straightforward solution to overcome the limitations of the individual manipulation space $\mathcal{M}_\mathrm{L}$ and $\mathcal{M}_\mathrm{R}$ and gain access to the global manipulation space $\mathcal{M}$ is to agree upon a fixed intermediate point in the mutual manipulation space $\mathcal{M}_m$ and pass the object from manipulator to manipulator through that point. However, since the orientation cannot be controlled directly, a sequence of different hand-over positions must be determined such that the sum of subsequent relative re-orientations accumulates into the desired orientation. An example is illustrated in Fig. 2.

III. Methodology

To solve this problem of realizing an underactuated pick-and-place under dynamic changes in the environment, we will resort to a learning strategy that learns to cooperate. Before doing so, we establish a mathematical problem formulation.

A. Problem formulation

The state of the system is determined by $\mathbf{s}_t$ which contains the Cartesian coordinates of the *manipulated object* and its orientation. The goal configuration of the object is denoted as $\mathbf{s}_g$ and contains the same information as the state.

$$\mathbf{s}_t = (x_t, y_t, \theta_t) \in \mathcal{S}$$
$$\mathbf{s}_g = (x_g, y_g, \theta_g) \in \mathcal{S}$$

Formally our supervisory control needs to determine which agent to query and whether the current action is final; if not the next drop-off location has to be determined. Since the goal state's $\mathbf{s}_g$ location can be anywhere in the global manipulation space, including the mutual manipulation space $\mathcal{M}_m$, it is not straightforward which manipulator to query. Therefore, these decisions are left to the supervisory control system. They are represented by the boolean $f_t \in \{0, 1\}$ indicating whether the present action is final, and the discrete variable $b_t \in \{\mathrm{L}, \mathrm{R}\}$, indicating whether to query the left or right manipulator. The next drop-off position on the other hand is represented by the actions $(x_{t+1}, y_{t+1}) \in \mathcal{M}$ in Cartesian coordinates. Note that the drop-off position is only relevant when $f_t = 0$, otherwise the drop-off location is equivalent to the goal position by definition so that in practice $(x_{t+1}, y_{t+1}) \in \mathcal{M}_m$ when $f_t = 0$. Resulting in the following action space $\mathcal{A}$

$$\mathbf{a}_t = (f_t, b_t, x_{t+1}, y_{t+1}) \in \mathcal{A} = \{0, 1\} \times \{\mathrm{L}, \mathrm{R}\} \times \mathcal{M}$$

The dynamics of the problem are governed by a nonlinear function that depends on the forward kinematics of the robotic manipulators, the manipulator that is queried and whether the query is final or not. If the query is final, by definition the next position is equal to the goal position. Formally

$$\begin{aligned}\mathbf{s}_{t+1} &= \mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_g) \\ &= \begin{cases} (x_{t+1}, y_{t+1}, \theta(\mathbf{s}_t, \mathbf{a}_t)), & f_t = 0 \\ (x_g, y_g, \theta(\mathbf{s}_t, \mathbf{a}_t)), & f_t = 1 \end{cases}\end{aligned}$$

Our objective is thus to design a time-invariant policy function $\pi : \mathcal{S}^2 \mapsto \mathcal{A}$ so that for any initial state in the global manipulation space, a finite sequence pick-and-place maneuvers is determined that manipulate the object into the desired configuration. To design such a policy we consider a first-exit optimal control problem formulation

$$\pi^* = \arg\min_\pi \mathbb{E}_{\mathbf{s}_0 \sim \mathcal{U}(\mathcal{S})} \left[\sum_{t=0}^{t'} r(\mathbf{s}_t, \mathbf{s}_g, \pi(\mathbf{s}_t, \mathbf{s}_g)) \right]$$

Here $\mathcal{U}(\mathcal{S})$ denotes the uniform distribution on $\mathcal{S}$. The function $r : \mathcal{S}^2 \times \mathcal{A} \mapsto \mathbb{R}$ is defined as the cost rate and can be used to express abstract features of the policy such as maximum accuracy or minimal execution time. The policy can terminate the sequence at any time t' by taking $f_t = 1$. Consequently, the length of each sequence will differ. Taking $f_t = 1$ initiates the final step, which can be determined by the policy itself or externally. The policy depends on the goal state, making it context dependent. Therefore, the same policy can be used even if the context changes, e.g. after a manipulation task is successfully completed or mid-execution.

B. Solution strategy

Here we detail the proposed solution strategy.

*1) **Discretization of** $\mathcal{M}_m$:* In order to downscale the solution space the mutual workspace is discretized. Using the inverse kinematics of the robotic manipulators we verify the feasibility of an arbitrary discrete set of points $\mathcal{D} \in \mathbb{R}^3$. This is done for both manipulators, resulting in two discretized manipulation spaces $\mathcal{M}'_R \subset \mathcal{M}_R$ and $\mathcal{M}'_L \subset \mathcal{M}_L$. Again the discretized mutual manipulation space can be obtained from the intersection of both discretized manipulation spaces, $\mathcal{M}'_m = \mathcal{M}'_{\mathrm{L}} \cap \mathcal{M}'_{\mathrm{R}}$. The resulting discretized mutual manipulation space $\mathcal{M}'_m \subset \mathcal{M}_m$ has a finite cardinality $M = \mathrm{card}(\mathcal{M}'_m)$. A two-dimensional visualization of this mutual workspace is shown in Fig. 3. As a result $\mathcal{A}' = \{0, 1\} \times \{\mathrm{L}, \mathrm{R}\} \times \mathcal{M}'_m$.

*2) **Policy definition**:* Since the problem is deterministic and the action space is now discrete, a possible strategy could be to tackle the problem using a shortest path algorithm such as Dijkstra or A*. As a result of the dimensionality of the state-space and the connectedness of the individual states this would generate a densely connected graph. The supervisory policy ought to search this graph in real-time, since we desire it to be context dependent, making this method infeasible. Therefore, instead we opt to find a parametrized policy $\pi(\cdot, \cdot) \approx \pi(\cdot, \cdot; \boldsymbol{\phi})$ assigning a subtask each discrete time step to one of the manipulators, after which they perform their own low-level motion planning. Next we propose two alternative strategies to determine such a policy. A distinction can be made based on different treatments of the final step. The final step indicates the termination of a sequence and ideally would result in the correct orientation at the goal location. This decision could be made by the policy or externally, e.g. based on the kinematics of the system, resulting in two policy architectures.

a) Final robot fixed (architecture a): The final action is defined by the boolean f_t. It affects the problem in a highly nonlinear fashion, rendering a hard to find optimal solution. This boolean can be eliminated by limiting the goal state to the manipulation space of one of two robotic manipulators, i.e. $(x_g, y_g) \in \mathcal{M}_{b_{\text{final}}} - \mathcal{M}_m$ where $b_{\text{final}} \in \{\mathrm{L}, \mathrm{R}\}$ indicates the robot that has to perform the final action. When a goal state $\mathbf{s}_g$ is queried it is then easily verified whether $b_{\text{final}} = \mathrm{L}$ or $b_{\text{final}} = \mathrm{R}$ which is then used to determine which policy to activate. We must thus use and train *two* policies depending on which manipulator needs to execute the final action. Furthermore, this design choice limits the applicability of the policy, since the goal position is limited to a certain robot's manipulation space.[1] Now as soon the final robot has been identified, we can check for each state $\mathbf{s}_t \in \mathcal{M}_m$ whether it corresponds to the goal configuration $\mathbf{s}_g$ when we would use the final manipulator to maneuver it to the goal position (x_g, y_g). This is implemented by representing the goal configuration as a desired relative orientation between the object and the unique final robot $\theta^{\text{rel}}_{\text{final}}$ which is possible since we use different policies depending on the robot that will be queried last. The relative orientation between the object and the final robot can be calculated by taking the difference between the object's orientation θ_t and the yaw of the robot's end-effector, which is the result of the manipulator's morphology and the pick-up and drop-off position. Note that $\mathbf{g}$ depends on the manipulator.

$$\theta^{\text{rel}}_t = \mathbf{g}_{b_{\text{final}}}(\mathbf{s}_t) = \theta_t - \theta_{b_{\text{final}}}(x_t, y_t)$$

During a pick-and-place the relative orientation does not change, since a rigid grasp is assumed. Hence, if θ^{rel}_t equals $\theta^{\text{rel}}_g = \mathbf{g}(\mathbf{s}_g)$ the object will have the correct orientation when placed at the goal position. The action space reduces to

$$\mathbf{a}_t = (b_t, x_{t+1}, y_{t+1}) \in \mathcal{A}'' = \{\mathrm{L}, \mathrm{R}\} \times \mathcal{M}'_m$$

Now the policy needs to find a sequence of actions resulting in a state in the mutual manipulation space, $\mathcal{M}_m$, that indirectly corresponds to a certain goal state $\mathbf{s}_g \in \mathcal{M} - \mathcal{M}_m$. Whether the step is final is not dictated by the policy, but is determined by an auxiliary routine that is based on θ^{rel}_t. Finally, the previous robot that performed an action is provided to the system as well. This way the policy can reason based on the previous robot to switch between robots, this to change the relative orientation. Resulting in the following augmented state

$$\hat{\mathbf{s}}_t = (x_t, y_t, \theta_t, \theta^{\text{rel}}_g, b_t) \in \mathcal{M} \times \mathbb{R}^2 \times \{\mathrm{L}, \mathrm{R}\}$$

The reward received after acting on the system is sparse. A zero is received when the state, at time t and residing in the mutual workspace, results in the desired relative orientation with the given final robot. Any other step receives minus one.

$$r = \begin{cases} 0, & \theta^{\text{rel}}_t = \theta^{\text{rel}}_g \\ -1, & \theta^{\text{rel}}_t \neq \theta^{\text{rel}}_g \end{cases}$$

b) Learned final step (architecture b): A second approach is a policy that dictates when the final step is taken, resulting in a full action as defined in III-A. The final step is now part of the sequence, in contrast to the previous strategy. The policy will have to dictate from which state $s_t \in \mathcal{S}$ the object needs to be placed at the goal position and with what manipulator. Additionally, it will have to determine a sequence of actions to end up in that state. The goal state is provided in its entirety to the policy and the robot that placed the block at its position as well. The following augmented state is provided to the policy

$$\hat{\mathbf{s}}_t = (x_t, y_t, \theta_t, x_g, y_g, \theta_g, b_t) \in \mathcal{S}^2 \times \{\mathrm{L}, \mathrm{R}\}$$

Introducing this final step in the action space introduces a number of possible terminations of an episode. Similar to the previous strategy taking an action that does not result in the object ending up in the final configuration is punished, since the episode length needs to be minimized. If the policy tries to drop off or pick up an object at a position the manipulator b_t cannot reach, because this position is outside its manipulation space, a large cost is returned. Finally, a sequence can end in two ways, either the object is placed at its goal position with the goal orientation or with a different orientation. The former is rewarded with a large reward, while the latter receives a cost equal to the difference in orientation scaled by a constant. The experimental problem of the pick-and-place task by two robotic manipulators is solved by implementing a heuristic reward. Instead of minimizing a cost rate, as described in III-A,

[1] $\mathcal{M}_m$ could be included by arbitrarily assigning a manipulator as final robot.

a reward function r is maximized. After some reward shaping, the following reward function is received

$$r = \begin{cases} 10, & (x_t, y_t) \equiv (x_g, y_g) \text{ and } |\theta - \theta_g| \leq \epsilon \\ -C \cdot |\theta_t - \theta_g|, & (x_t, y_t) \equiv (x_g, y_g) \text{ and } |\theta - \theta_g| > \epsilon \\ -1, & (x_t, y_t) \in \mathcal{M}_m \\ -25, & (x_t, y_t) \vee (x_{t-1}, y_{t-1}) \notin \mathcal{M}_{b_t} \end{cases}$$

Here $\mathcal{M}_{b_t}$ is the manipulation space of the robot that performs the action, indicated with the boolean $b_t \in \{\mathrm{L}, \mathrm{R}\}$. The final case occurs when the robot tries to pick up or drop off the object from or to a position outside its manipulation space. This can happen in the first step, if the object is outside the reach of the robot indicated by b_t, or in the final step, if the goal position is outside the robot's reach. The more complicated reward function indicates the difference in complexity between both strategies.

*3) **Reinforcement Learning**:* To determine the parametrized policy, deep Reinforcement Learning is used. The use of discrete actions steers us in the direction of Q-learning, where for each action the value function, estimating future rewards, is returned. Mnih et al. [20] proposes the use of a neural network to estimate the value-function. A neural network, called a deep Q-network (DQN), is trained to approximate the optimal action-value function $Q(s, a; \theta) \approx Q^*(s, a)$ with weights $\boldsymbol{\phi}$ and

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

The future discounted return at time t is

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

where γ is the discount factor and determines the horizon.

This neural network is trained off-policy using a stochastic gradient descent. The required parametric policy is defined implicitly using the greedy strategy $\pi(\mathbf{s}) = \max_a Q(s, a; \boldsymbol{\phi})$. During training the actions are selected using an ϵ-greedy strategy. A random action is selected with probability ϵ, while a greedy strategy is used with probability $1 - \epsilon$. The value of ϵ decreases exponentially during training. Q-learning updates are applied to batches of experience $(s, a, r, s') \sim U(\mathcal{D})$, where $U(\mathcal{D})$ denotes a uniform distribution over the set of experiences $\mathcal{D}$. The loss function used by the Q-learning update at iteration i is

$$L_i(\boldsymbol{\phi}_i) = \mathbb{E}_{(s,a,r,s') \sim U(\mathcal{D})}[(r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\phi}_i^-) - Q(s, a; \boldsymbol{\phi}_i))^2]$$

with $\boldsymbol{\phi}_i$ being the parameters of the Q-network at iteration i and $\boldsymbol{\phi}_i^-$ the network parameters used to compute the target at iteration i. The target is only updated every C steps.

The above can help to find the parametrized policies when considering a) a fixed final robot or b) a learned final step. The two different policy architectures, result in a different neural network approximators.

In the former the possible drop-off positions (x_{t+1}, y_{t+1}) can contain all points in the discretized mutual workspace $\mathcal{M}'_m$, i.e. all yellow positions in Fig. 3. The boolean b_t is implemented by doubling the mutual workspace. A function $f : \mathcal{M}'^2_m \mapsto \{0, 1\} \times \mathcal{M}'_m$ translates each position to a position performed by a certain robot, by looking at its index. The first half is performed by the first robot, while the second half by the second robot. The network is visualized in Fig. 4.

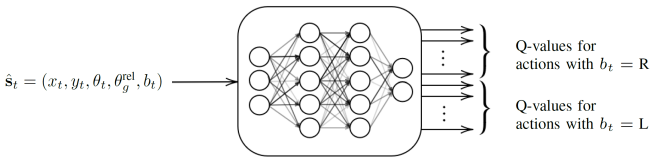


Fig. 4: A visualization of the deep Q-network with the final robot fixed (Architecture a).

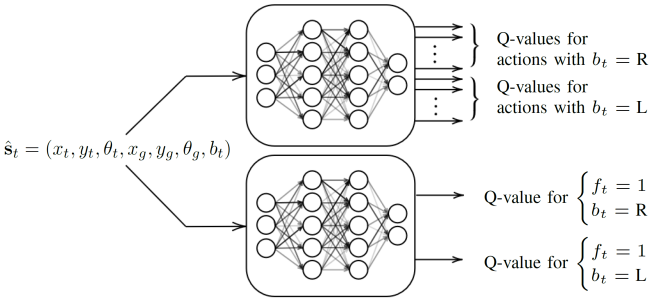


Fig. 5: A visualization of the deep Q-network trained with a learned final step (Architecture b).

For architecture b a final action needs to be introduced. Two additional actions are added, containing the final step done by one of the two robots. When this action is chosen the object is placed at the goal position $(x_g, y_g) \in \mathcal{M}$ and the orientation θ is compared to the goal orientation θ_g. This action is final and terminates the episode. These two actions are visualized in Fig. 5. The reason a parallel network is used to represent the Q-value for the two final actions, is discussed in more detail in IV-B2. Here two methods are introduced to find the policy, corresponding to architecture b.

*4) **Sim2Real**:* A large amount of episodes needs to be played to train the network. This is not practicable on a real setup, since this would take weeks to train. Luckily the deterministic property of the setup and its open loop control make it possible to translate the problem to simulation.

Even more, the policy can be trained without taking the motion planning into account. It is assumed the relative orientation between the object and the robot doesn't change during the motion planning, since the object is picked up with a rigid grasp. Only the robot poses at the drop-off and pick-up locations need to be calculated.

The trained policy in simulation will be executed on the real setup. This translation of simulated environment to real environment depends heavily on the models and the repeatability of the two robotic manipulators.

IV. Experiments

In this section the training results in simulation are discussed first. Secondly the translation to the real setup is considered.

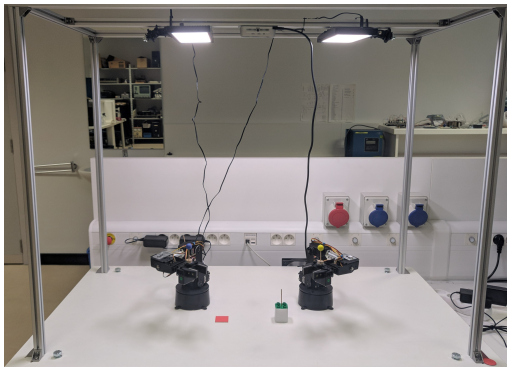


Fig. 6: The complete setup, with the manipulators at the center.

A. Set-up

Two low-cost robotic manipulators are used, two Lynxmotion AL5A robotic arms with five DOFs. The two manipulators can perform pick-and-place maneuvers on a LEGO DUPLO block by grasping the toothpick attached at the center. They are steered in an open-loop fashion from the pick-up position to the drop-off position, using an approximately linear path in the feasible joint space, though keeping the square levelled. A camera, an Intel RealSense Depth Camera D435, is used to perceive information from the system, i.e. to position and orientation of the block and the goal.

A picture of the setup is shown in Fig. 6. Two lights, which can be seen at the top of the frame next to the camera, are mounted in order to provide consistent lighting conditions for the vision. The vision is based on color and edge detection.

B. Implementation

All code is written in Julia [21]. The DQN network is trained using the Julia package ReinforcementLearning.jl [22].

1) ***Discretization****:* A three-dimensional grid of points spaced apart 10 mm in each direction is taken as the set of discrete points $\mathcal{D}$. The feasibility of all points in this set are checked for each robotic manipulator resulting in two point clouds, corresponding to the discrete manipulation spaces. The discretized mutual manipulation space is taken as the intersection of these two point clouds. A two-dimensional section of the calculated discretized manipulation spaces are shown in Fig. 3.

2) ***Networks and training****:* A selection of the used hyperparameters are shown in Table I. The sigmoid function is used as activation function. Three methods are compared. The first method corresponds to the policy architecture a, with a fixed final robot. The last two methods use policy architecture b, and consist of a method without pretraining and one with pretraining.

The latter uses the network proposed in Fig. 5. This is done so that the network controlling the two final actions can be pretrained. Pretraining is done to improve the learning rate and find a viable solution, which will be shown in the next section. The network is first trained on a one-step environment. The goal state corresponds to the start state, such that if the object is placed directly at the goal position with the correct robot, the object will have the goal orientation. This pretraining is done in two steps, first with the parallel network only, with two outputs, and finally with the complete network.

TABLE I: Used hyperparameters for the different trained policies.

Name		**Optimizer**	**Learning rate**	**DNN size**	**Steps**
Architecture a		Adam [23]	0.001	5|64|354	150,000
Architecture b	No pretraining	Adam	0.001	7|512|512|356	500,000
	Pretraining: partial network	Adam	0.001	7|256|256|2	250,000
	Pretraining: full network	Adam	0.001	7|256|256|254 7|256|256|2	150,000
	Final training with pretraining	Adam	0.001	7|256|256|354 7|256|256|2	1,750,000

C. Results

All architectures are trained in simulation and subsequently transferred to the real set-up. No further training on the set-up has been performed.

1) ***Importance of pretraining****:* First we elaborate on the effect of the previously discussed pretraining. The pretraining is to force the network to use the final actions more often, since these encompass the final goal. Since an ϵ-greedy strategy is used to train the network, actions at the start have a higher chance to be chosen at random. Therefore, the two final actions, which are only a small fraction of all possible actions, are only chosen rarely if no pretraining is done. With pretraining the network learns which states correspond to certain final states, resulting in a high reward for the two final actions for these states.

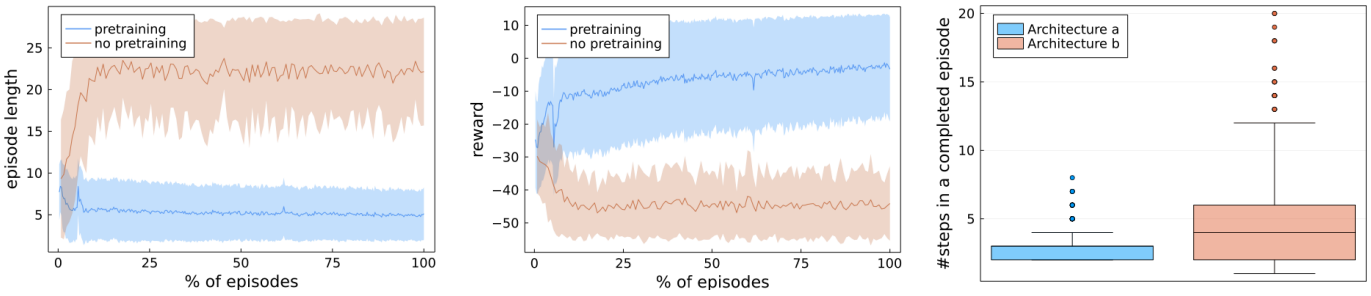
The maximum episode length during training is limited to 25 steps. The episode can end in less steps, if the final action is chosen earlier. The effect of this pretraining is clearly visible in Figs. 7a and 7b, since the method using no pretraining does not find a viable solution. The figures show the mean and variance per 500 episodes. The network trained with pretraining converges to an average episode length of 5 steps, while the network trained without pretraining settles around the maximum episode length. The reward on the other hand is still rising for the method with pretraining, while it converges to a low value for the method using no pretraining.

In the subsequent section only the method with pretraining is used to compare the policy architectures, and thus simply referred to as architecture b from now on.

2) ***Comparison of architecture a and b****:* The two architectures are compared on the basis of two performance indicators: the success rate and the number of steps in an episode. The former is defined as the percentage of episodes for which the goal configuration is reached. The policies are tested on 1000 random start and goal states, the results are shown in Table II.

TABLE II: Performance parameters for both networks.

	Success rate [%]	**Average number of steps**
Architecture a	98.8	2.78
Architecture b	71.4	4.54



(a) Episode lengths during training (b) Rewards (c) Episode lengths after training

Fig. 7: (a)-(b): The reward and episode length during training compared for the methods with and without pretraining. (c): The amount of steps in an episode for 1000 experiments with the trained policies, comparing the performance for both architectures in simulation.

A box plot showing the number of steps in more detail is shown in fig. 7c. The first architecture outperforms the second, with fewer steps and a higher success rate. However, it is less applicable since the goal state is limited to the manipulation space of the final robot, which is fixed.

The final architecture can be looked at in more depth, which is done in 8a. The episode can end four ways: at the goal position with the desired orientation (cr), at the goal position with the wrong orientation (crwa), goal position not reachable by the final robot (wr) or the episode could end without trying to place it at the goal position (nf). It can be seen that in the majority of the cases the robot places the object at the goal position with the correct robot. When placed at the goal position, the correct orientation is achieved most of the time, indicating the policy has found a viable action sequence before calling the (correct) final action.

*3) **Experimental validation**:* Both policies trained in simulation are tested on a real setup. No further training is done on the setup. An example of a sequence performed on the setup is shown in figure 1. The position and the orientation of the block are updated in real time, after each discrete time step. The performance parameters for both robots are found in table III. The error on the orientation is roughly the same for both policy architectures, with an average of 4.56 and 5.15 degrees and a standard deviation of 4.78 and 5.12 degrees respectively for architecture a and b.

TABLE III: Performance parameters for both networks tested for 100 random episodes on the real setup.

	Success rate [%]	**Average number of steps**
Architecture a	90	3.67
Architecture b	72	4.83

Similar to the results in simulation the first policy needs fewer steps than the second strategy to end up with the correct orientation, although the difference becomes smaller. Comparing 8c and 7c shows that the first is less performant on the real setup, while the second shows similar results in simulation as on the real setup. Presumably, the latter translates better to the real world because it does not use the robotic manipulator's model to take a decision on the final step. Therefore, it is less sensitive to errors in placing the block.

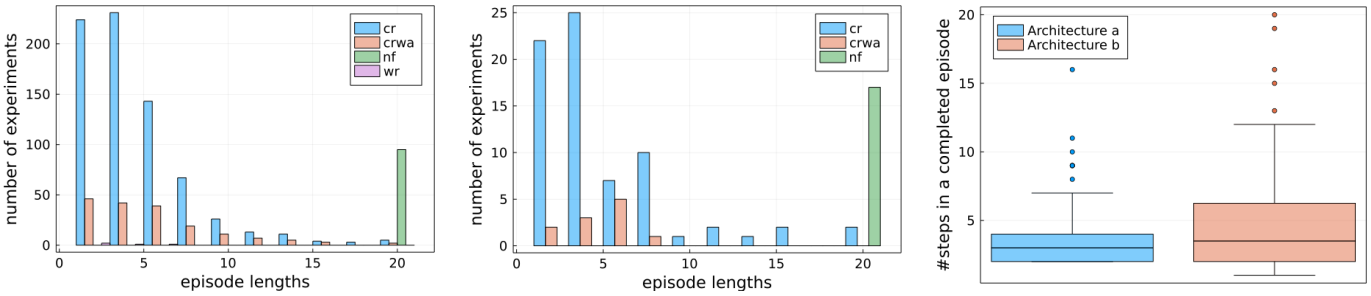
Figure 8b shows the distribution of the episode lengths for architecture b. Since in this no wrong robot is called upon to perform the final step, only three possible cases are indicated. The results are similar in both figures, although the second figure shows less of a normal distribution which could be attributed to the lower number of experiments.

Translating the policy to the real setup seems to work, since similar results are achieved. This could be attributed to the fact that we work on a high level, making the exact model of the system less important. Using the hierarchical control structure is beneficial to react to uncertainties in the environment, as a result of the stochasticity of the real world. Additionally, since a certain error ϵ is allowed both in simulation and in the real world, the model does not need to be exact. Using the policy in a closed loop aids the translation as well, since the previous errors in the model are of no importance. The policy finds a new action based on the new state, which is updated each step.

V. Conclusion

A first step is made to have a multi-robot system that is capable of detecting opportunities to collaborate in an autonomous fashion. Although a successful cooperative multi-agent manipulation system is achieved, it is not yet fully autonomous. The trained hierarchical control system works in real-time and is able to adapt to a dynamic environment. The goal state could be changed mid-execution or the block could be moved in between steps. This is achieved by training a time-invariant parametrized policy in simulation, using reinforcement learning techniques. This principle can be extended to include more robots or to have a larger variety of possible actions. No information of the goal configuration is used in the second policy architecture, it is only checked if the goal configuration is achieved. Therefore, a different type of goal configuration could be considered in future work.

Training in simulation proves to be beneficial by allowing more iterations in less time. With some additional hyper-parameter tuning the performance and learning rate of the second architecture could be improved. Others algorithms could be looked at as well, e.g. the sample efficiency could be improved by using HER [24]. The benefit of training in simulation becomes clear when comparing numbers. Validating 100 episodes on the real setup, with a maximum of 20 steps per episode, took 3.5 hours with an average of around 7.5 steps. Performing the $1,750,000$ steps needed to train the second



(a) Simulation. (b) Real setup. (c) Episode lengths.

Fig. 8: (a)-(b): In depth look for architecture b, both in simulation as on the real setup. (c): Comparison both architectures on the real setup.

method would take 8167 hours or a little more than 340 days non-stop; this is without taking into account the pretraining.

Translating the policy trained in simulation to the real world works, but has its limitations. When translated to the real setup, providing more information to the system works less well. Since the discrepancy between the simulation and the real robotic manipulators are embedded in the system. This possible overfit on deterministic data in simulation can be further explained by the use of partially discrete action and state space. Although the policy expects a certain exact position, the actual position might differ. This actual position is not seen before by the policy, since a discrete set of actions is taken in simulation. Translating the policy to the real setup could be improved by describing the action and state space by continuous variables. Additionally, by using the continuous representation the system could become more adaptable, by e.g. being able to overcome certain constraints in the workspace. The method could be adapted to include time varying constraints as well. An additional method to bridge the *Sim2Real* gap could be to introduce stochasticity in the simulation. This way the real world example will just be an extra variation on the deterministic case.

REFERENCES

[1] Y. Rizk, M. Awad, and E. W. Tunstel, "Cooperative Heterogeneous Multi-Robot Systems," ACM Computing Surveys, vol. 52, no. 2. Association for Computing Machinery (ACM), pp. 1–31, May 31, 2019. doi: 10.1145/3303848.

[2] M. Tan, "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents", in In Proceedings of the Tenth International Conference on Machine Learning, 1993, bll 330–337.

[3] H. Duan and D. Zhang, "A Binary Tree Based Coordination Scheme for Target Enclosing With Micro Aerial Vehicles," in IEEE/ASME Transactions on Mechatronics, vol. 26, no. 1, pp. 458-468, Feb. 2021, doi: 10.1109/TMECH.2020.3028200.

[4] N. Bezzo, B. Griffin, P. Cruz, J. Donahue, R. Fierro and J. Wood, "A Cooperative Heterogeneous Mobile Wireless Mechatronic System," in IEEE/ASME Transactions on Mechatronics, vol. 19, no. 1, pp. 20-31, Feb. 2014, doi: 10.1109/TMECH.2012.2218254.

[5] W. Gueaieb, F. Karray and S. Al-Sharhan, "A Robust Hybrid Intelligent Position/Force Control Scheme for Cooperative Manipulators," in IEEE/ASME Transactions on Mechatronics, vol. 12, no. 2, pp. 109-125, April 2007, doi: 10.1109/TMECH.2007.892820.

[6] Dogar, M., Spielberg, A., Baker, S. et al. Multi-robot grasp planning for sequential assembly operations. Auton Robot 43, 649–664 (2019). https://doi.org/10.1007/s10514-018-9748-z.

[7] C. R. Garrett et al., "Integrated Task and Motion Planning", Annual Review of Control, Robotics, and Autonomous Systems, vol 4, no 1, bll 265–293, 2021.

[8] C. V. Braun, J. Ortiz-Haro, M. Toussaint, en O. S. Oguz, "RHH-LGP: Receding Horizon And Heuristics-Based Logic-Geometric Programming For Task And Motion Planning", arXiv [cs.RO]. 2021.

[9] I. Umay, B. Fidan and W. Melek, "An Integrated Task and Motion Planning Technique for Multi-Robot-Systems," 2019 IEEE International Symposium on Robotic and Sensors Environments (ROSE), 2019, pp. 1-7, doi: 10.1109/ROSE.2019.8790413.

[10] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," in IEEE Signal Processing Magazine, vol. 34, no. 6, pp. 26-38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[11] M. Kim, D.-K. Han, J.-H. Park, and J.-S. Kim, "Motion Planning of Robot Manipulators for a Smoother Path Using a Twin Delayed Deep Deterministic Policy Gradient with Hindsight Experience Replay," Applied Sciences, vol. 10, no. 2, p. 575, Jan. 2020.

[12] H. Nguyen and H. La, "Review of Deep Reinforcement Learning for Robot Manipulation," 2019 Third IEEE International Conference on Robotic Computing (IRC), 2019, pp. 590-595, doi: 10.1109/IRC.2019.00120.

[13] A. Rajeswaran, V. Kumar, A. Gupta, J. Schulman, E. Todorov, en S. Levine, "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations", CoRR, vol abs/1709.10087, 2017.

[14] T. T. Nguyen, N. D. Nguyen and S. Nahavandi, "Deep Reinforcement Learning for Multiagent Systems: A Review of Challenges, Solutions, and Applications," in IEEE Transactions on Cybernetics, vol. 50, no. 9, pp. 3826-3839, Sept. 2020, doi: 10.1109/TCYB.2020.2977374.

[15] A. Singh, L. Yang, K. Hartikainen, C. Finn, en S. Levine, "End-to-End Robotic Reinforcement Learning without Reward Engineering", CoRR, vol abs/1904.07854, 2019.

[16] O. Nachum, M. Ahn, H. Ponte, S. Gu, en V. Kumar, "Multi-Agent Manipulation via Locomotion using Hierarchical Sim2Real", CoRR, vol abs/1908.05224, 2019.

[17] T. Zhang, K. Zhang, J. Lin, W. -Y. G. Louie and H. Huang, "Sim2real Learning of Obstacle Avoidance for Robotic Manipulators in Uncertain Environments," in IEEE Robotics and Automation Letters, vol. 7, no. 1, pp. 65-72, Jan. 2022, doi: 10.1109/LRA.2021.3116700.

[18] S. Höfer et al., "Sim2Real in Robotics and Automation: Applications and Challenges," in IEEE Transactions on Automation Science and Engineering, vol. 18, no. 2, pp. 398-400, April 2021, doi: 10.1109/TASE.2021.3064065.

[19] W. Zhao, J. P. Queralta and T. Westerlund, "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey," 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020, pp. 737-744, doi: 10.1109/SSCI47803.2020.9308468.

[20] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540. Springer Science and Business Media LLC, pp. 529–533, Feb. 25, 2015. doi: 10.1038/nature14236.

[21] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing", SIAM review, vol 59, no 1, bll 65–98, 2017.

[22] J. Tian and other Contributors, "ReinforcementLearning.jl: A Reinforcement Learning Package for the Julia Programming Language". 2020.

[23] D. P. Kingma en J. Ba, "Adam: A Method for Stochastic Optimization", arXiv [cs.LG]. 2017.

[24] M. Andrychowicz et al., "Hindsight Experience Replay", CoRR, vol abs/1707.01495, 2017.