Architectural Assembly with Tactile Skills: Simulation and Optimization

Jan Schneider Tim Schneider Boris Belousov Georgia Chalvatzaki Samuele Tosatto Bastian Wibranek

***Abstract*— Construction is an industry that could benefit significantly from automation, yet still relies heavily on manual human labor. Thus, we investigate how a robotic arm can be used to assemble a structure from predefined building blocks autonomously. Since assembling structures is a challenging task that involves complex contact dynamics, we propose to use a combination of reinforcement learning and planning for this task. In this work we take a first step towards autonomous construction by training a controller to place a single building block in simulation. Our evaluations show that TD3 can be used on this task to achieve placement errors of around 1cm on average. We conclude that this precision - albeit not being perfect - validates our approach and gives reason to move on to more complex assembly tasks which include structures with multiple blocks.**

I. Introduction

In modern society, automation is omnipresent in large parts of the world-wide economy. Yet construction - despite being one of the biggest industries - is still relying heavily on manual human labor. Due to the versatility and complexity of the work performed on construction sites, human work force is vital and cannot easily be replaced by machines.

However, working in the construction industry is linked to a series of health and safety problems. For example in the UK, 27% of fatal workplace injuries happen in the construction industry [1]. Heavy lifting and working in cramped positions for long durations, which is typical in the construction industry, also increases the risk of musculoskeletal disorders [1]. Therefore, construction workers are often forced to quit their job early due to health problems. Furthermore, there are construction tasks in locations that are inherently inaccessible or unsafe to human workers, such as areas of recent natural disasters, deep sea, or outer space [2]. Many of these problems can be avoided by using robots instead of human workers to fulfill harmful and dangerous tasks on construction sites.

In this work we aim to tackle the problem of constructing modular designs in simulation using a UR10 [3] robot arm with a Robotis RH-P12-RN [4] gripper. The designs we construct consist of predefined building blocks, which we will call *parts* for the remainder of this report. A predefined construction plan defines the target positions and orientations of each part as well as the order in which the parts have to be placed. For a visualization of the described setup, please refer to Figure 1.

An additional benefit of simulating the construction process is that the target structure can already be optimized for automated assembly during the design phase. Such a *design-for-robotic-assembly* [5] process would allow for modifications in the target structure that avoid instabilities during the construction process or increase the overall efficiency of the automated construction.

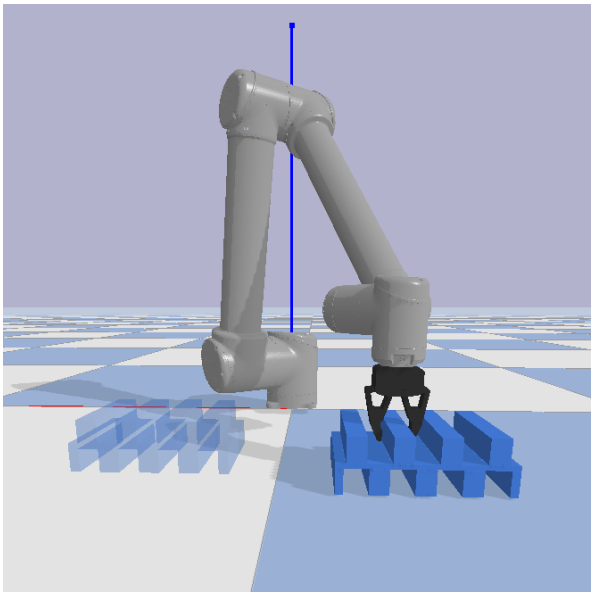

Fig. 1. UR10 robot with a Robotis RH-P12-RN gripper manipulating a part in a `PyBullet` simulation. The objective is to move the grasped part to the target location on the left. For reference: the robot arm has a length of 130cm, the building block is 10cm high, 30cm wide and 20cm long.

During construction, accumulating inaccuracies in the placement of parts, incorrectly placed grasps, and slippage make it necessary for the controller adapt to feedback from sensors. Typically, an RGB or depth camera is used to provide sensor input to the controller in such scenarios. Cameras, however, have the disadvantage that crucial parts of the scene can be occluded by the robot or by the object that is manipulated by the robot [6]. Additionally, in contact-rich manipulation tasks, very small offsets of a couple of millimeters can already have a significant impact on the task performance. For example, a small error in the estimated part position can make the difference between placing the part correctly and hitting and potentially destroying the present structure.

Since it is hard to consistently obtain such levels of accuracy from a camera alone, we compliment the input from an Intel RealSense LiDAR L515 [7] depth camera with a DIGIT [8] vision-based tactile sensor on each fingertip of the robot gripper. Tactile sensors have been shown to be capable of detecting slippage [9, 10] and have been used for a variety of manipulation tasks in the past [11, 12, 13]. The DIGIT

sensor was chosen for this task since it is provides high-resolution feedback, is comparably inexpensive, and small enough to be attached to the fingers of the Robotis RH-P12-RN gripper.

Using such high-dimensional sensors comes with the major drawback that the sensor readings are typically hard to interpret. Therefore, we approach the task with a combination of reinforcement learning and planning as further described in subsection III-D. In section V we evaluate our method on a task where a single part has to be placed in a predefined pose in an otherwise empty scene and show that our method achieves a mean placement error of around 1cm. We conclude that our results validate our method and give reason to move on to more complex assembly tasks consisting of multiple parts.

II. Related Work

Similar to our work, Hartmann et al. [5] also investigate the problem of building structures from a set of given blocks with autonomous robots. They approach the task by applying task-and-motion planning to calculate a path in the configuration space of the robots. However, the authors neglect the actual control problem and instead assume that the robots are able to follow the plan perfectly without any deviations and that the parts can be grasped without any slippage, which limits the applicability of this approach.

Sartoretti et al. [14] tackle the problem of building a structure with many tiny autonomous robots. Reinforcement learning is used to learn a common collaborative policy for all robots. The fact that the policy is shared across all robots allows to vary the number of robots without much additional effort.

Utilizing tactile information for robotic manipulation tasks is an area of active research [15, 16]. Since information from tactile sensors is typically high-dimensional and hard to interpret, oftentimes learning is applied to create controllers that use this kind of information. For example in [11] a model is learned that predicts whether a future grasp will be stable, based on input from a GelSight [17] tactile sensor and the actions to be executed. The controller then searches the action space for a actions that will result in a stable grasp. In [12] reinforcement learning is used to stabilize an object with a robot arm based on tactile feedback from a BioTac [18] tactile sensor. Tian et al. [13] use reinforcement learning to move and rotate small objects like marbles and dice based on sensor information from a modified GelSight sensor.

III. Methodology

In the following section we describe the construction task to be solved, how the task is simulated, and which techniques are used to solve the task.

A. Task Definition

The objective of this work is to use a robot to build modular structures out of different types of building blocks (*parts*) according to a predefined construction plan. This construction plan defines not only the target position and orientation of each part in the structure but also the order in which the parts have to be placed. While it would be possible to include finding the optimal placement order as part of the task, we decided that solving this discrete optimization problem would require intensive research that goes beyond the scope of this project. Thus, we model the construction plan P as a list of steps $s_i = (t_i, \mathbf{p}_i, \mathbf{q}_i)$, where $t_i \in \mathbb{N}$ is a part type identifier, $\mathbf{p}_i \in \mathbb{R}^3$ is the target part position and $\mathbf{q}_i \in \mathbb{R}^4$ is the quaternion that describes the target part orientation.

B. Simulation Environment

Since training reinforcement learning algorithms on real robots is time and labor intensive, we use a simulator to enable rapid training. Later, the learned controllers will be transferred to the real robot and fine-tuned. While our first choice of a simulator was `CoppeliaSim` [19], we later decided to perform our experiments using `PyBullet` [20] instead, as the lower communication overhead of the latter reduced our runtime to roughly 25% of the original runtime.

As visible in Figure 1, we use a simulated version of the UR10 [3] robot arm with the Robotis RH-P12-RN [4] gripper. The UR10 arm has 6 motorized joints, which we control in velocity mode, while the RH-P12-RN gripper has a single motor to operate 4 joints, which we control in position mode. On the fingertips of the gripper, we attached simulated DIGIT sensors, as further described in subsection III-C. The robot controller is executed at a rate of 20Hz which is realistic for a real system as well. However, contrary to a real system the simulator is fully synchronized and hence will not produce any delayed signals and any action issued by the controller will be applied immediately.

C. Simulating the DIGIT sensors

DIGIT [8] is a high-dimensional vision-based tactile sensor, similar to GelSight [17]. It consists of an opaque layer of gel, which is illuminated from the inside of the sensor by three differently colored LEDs. These LEDs are placed inside the casing so that they illuminate the gel from different directions. The reflections of the LEDs' light from the gel is captured by an RGB camera. If the sensor touches an object, the gel deforms according to the geometry of the object. This deformation changes how the light of the LEDs reflects from the gel and thus the geometry of the touching object can be seen in the image captured by the camera (see Figure 2).

To train the agent in simulation, it is necessary to simulate the DIGIT sensor. `PyBullet` is one of the few simulators that features a deformable object simulation. However, this feature is still experimental and thus often yields not very realistic behavior, as noted by Matas et al. [21]. We found that the deformable object simulation is not realistic enough to simulate the gel of the sensor properly.

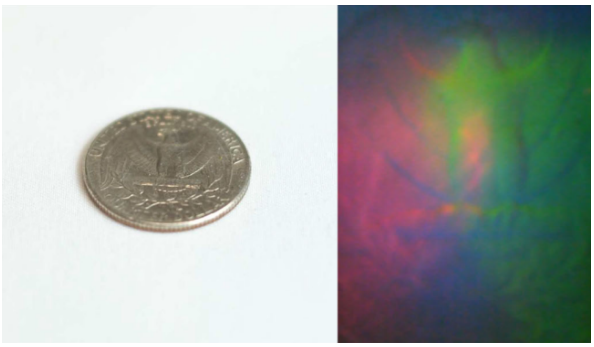

Fig. 2. Output of the real DIGIT sensor (image taken from [8]). Touching the coin with the sensor causes the gel to deform according to the coin's surface geometry. Fine details of the coin are clearly visible in the reflections of the LEDs' light from the gel.

Hence, instead of relying on deformable object simulation, we adapted the work of Gomes et al. [6]. Gomes et al. simulate the GelSight [17] tactile sensor with the help of a simulated depth camera. They do not simulate the layer of gel at all and instead use the depth camera to sense the geometry of objects that are close enough so that they would touch the gel layer. The authors use Phong shading [22] to calculate how the light of the LEDs is reflected from the gel if it is deformed by the sensed object geometry. For an exemplary output of the simulated DIGIT sensor, please refer to Figure 3.

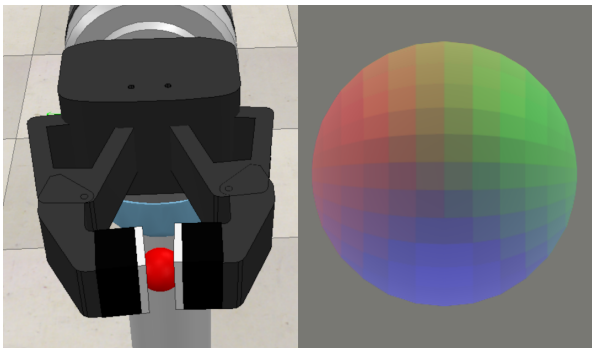

Fig. 3. Output of the simulated DIGIT sensor. One sensor is mounted on each fingertip of the gripper as it grasps a small sphere.

Since the resolution of the sensor is fairly high (640×480 pixels), the simulation is computationally expensive. Initially, one measurement took 136.9 milliseconds[1], which is quite limiting since measurements are required at every simulation step during training. By using the GPU to accelerate the sensor simulation, we were able to reduce the computation time for one measurement to 8.5 milliseconds[1]. This corresponds to a speedup of factor 16.

D. Combining Planning and Reinforcement Learning

To complete a single step of the construction process, the robot has to perform three steps: *grasping* the part at the pickup location, *transporting* the part to the vicinity of the target location, and *placing* the part. One option is to use reinforcement learning and learn to perform all three steps at once. However, this would cause fairly long training sequences, which are generally hard to learn, especially if the rewards are sparse. A general way of dealing with such problems is reward shaping [23], where expert knowledge about the solution is incorporated into the reward function to guide the learner towards a good solution. Yet, methods that require reward shaping are usually less general as the reward function is solution specific and they might result in a lower final performance as the agent is prevented from exploiting strategies that the expert did not consider [23].

Hence, in this work we chose a different approach. Instead of applying reinforcement learning on all steps at once, we only learn to perform the *placing* step and use open-loop trajectory planning for the *grasping* and *transporting* steps. Using trajectory planning for the *transporting* step is reasonable, as the way the part is transported is unlikely to influence the quality of the final placement if the part was grasped properly. Algorithms for collision-free trajectory planning have been developed for years and we found them to be robust enough for this application [24].

Using planning for the *grasping* step on the other hand comes at two major disadvantages. Firstly, for each type of part, a grasping sequence has to be created manually, which limits the generality of this method. Secondly, depending on the target position of the part, different grasps might be appropriate. While most parts can probably be placed by grasping them from the top, there might be some configurations where this is not possible. As an example, consider a part that has to be placed on the very top of a structure, barely in reach of the robot arm. Grasping that part from the side or below would increase the effective workspace of the robot and maybe make a placement possible that would be impossible if the part was grasped from the top. However, due to the limited scope of this project, we leave reinforcement learning for the *grasping* step as future work.

E. Learning to Place Parts

In the *placing* step, the robot starts with the part grasped and close to the target location.

a) Optimization Objective: The objective is to learn a policy $\pi_\theta(\mathbf{a} \mid \mathbf{o})$ that minimizes the expected cost of an episode $\tau = (\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{a}_{T-1}, \mathbf{s}_T)$:

$$\min_\theta \mathbb{E}_{\tau \sim p(\tau \mid \pi_\theta)} \left[\phi_f(\mathbf{s}_T) + \frac{1}{T} \sum_{t=1}^{T-1} \phi_i(\mathbf{s}_t, \mathbf{a}_t) \right]$$

where $\mathbf{s}_t$ is the full system state at time step t, $\mathbf{o}_t$ is the observation the policy gets of $\mathbf{s}_t$, $\mathbf{a}_t$ is the chosen action, $\phi_f(\mathbf{s}_T)$ is the final cost, $\phi_i(\mathbf{s}_t, \mathbf{a}_t)$ is the intermediate cost at step t, and

$$p(\tau \mid \pi_\theta) = p(\mathbf{s}_1) \prod_{t=1}^{T-1} p(\mathbf{s}_{t+1} \mid \mathbf{a}_t, \mathbf{s}_t)\, \pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)\, p(\mathbf{o}_t \mid \mathbf{s}_t)$$

is the trajectory distribution induced by policy π_θ. Here, T is not fixed but we terminate the episode when the robot

[1]Measurements taken on an AMD Ryzen 5 3600 CPU with an Nvidia GeForce RTX 2070 GPU

releases the part or when a maximum number of steps $T_{\max}$ has been reached.

The action vector is composed of two components: $\mathbf{a} = (\mathbf{a}_{arm}, a_{gripper})$, where the entries of $\mathbf{a}_{arm} \in \mathbb{R}^6$ are the target joint velocities of the 6 arm joints and $a_{gripper} \in [0, 1]$ is the target closure state of the gripper. Concerning the gripper control signal, a 1 indicates that the gripper shall be fully closed, while a 0 indicates that the gripper shall be fully opened. The signal is realized by a position controller on the gripper motor.

Our main objective is that all parts in the structure are placed properly and in a stable manner. Hence, we chose our final cost function to be a linear combination of two functions to cover both aspects of that objective:

$$\phi_f(\mathbf{s}_T) = \alpha_{pose}\phi_{pose}(\mathbf{s}_T) + \alpha_{vel}\phi_{vel}(\mathbf{s}_T)$$

where $\alpha_{pose}, \alpha_{vel} \in \mathbb{R}$ are weighting factors and ϕ_{pose} and ϕ_{vel} are cost functions for the part poses and velocities, further explained below.

ϕ_{pose} is a penalty for the pose error of any part in the scene. To ensure that the robot is not displacing any previously placed parts, this penalty includes the pose errors of previously placed parts. Note that "pose" here refers to a combination of position and orientation. Since it is challenging to define a distance metric on poses directly, as it is not straightforward to weight position and orientation error up in a useful manner, we chose a different approach to measuring pose error. To measure the error in pose, we compute the average positional error of the 8 vertices of the bounding box of each part:

$$d_k^2 = \frac{1}{8}\sum_{j=1}^{8} \|\mathbf{m}_{kj} - \tilde{\mathbf{m}}_{kj}\|_2^2$$

where $\mathbf{m}_{kj}$ is the target position of vertex j of part k at the final time step and $\tilde{\mathbf{m}}_{kj}$ is the actual position, respectively. The final pose cost is then computed as

$$\phi_{pose}(\mathbf{s}_T) = \min\left\{\frac{\frac{1}{P}\sum_{k=1}^{P}\delta_{\log}\left(d_k^2\right)}{\phi_{pose}^{\max}}, 1\right\}$$

where P is the number of parts of the structure, including the part to be placed and $\phi_{pose}^{\max}$ is a hyperparameter that controls the clamping of this term. The intuition behind the clamping is that if the part is placed to far away from the target location, we consider the task as failed and simply assign a maximum cost. Hence, the reward becomes more sparse and the Q-function in regions far away from the target state (e.g. if the robot threw the part away from itself) becomes easier to learn. The scaling of this term by $\frac{1}{\phi_{pose}^{\max}}$ simply ensures that $\phi_{pose}(\mathbf{s}_T) \in [0, 1]$, which makes it easier to choose the weights α later, since all cost terms will share the same domain. Further, $\delta_{\log}$ is defined as follows:

$$\delta_{\log}\left(d^2\right) = d^2 + 0.01\left(\ln\left(d^2 + 10^{-5}\right) - \ln\left(10^{-5}\right)\right) \quad (1)$$

While it would be possible to use the mean of the distances d_k^2 directly as a cost function, this approach comes at a major drawback: the closer a part is to its target pose, the flatter the squared penalty becomes and the less incentive the learner has to place the part even more precisely. Yet, as argued before, in contact-rich manipulation tasks, even sub-millimeter accuracy can be crucial for the successful completion of a task. The function in Equation 1 on the other hand has a concave shape near the optimal position while being similar to a quadratic cost term further away, as visible in Figure 4. We follow prior work with this type of cost function [25].

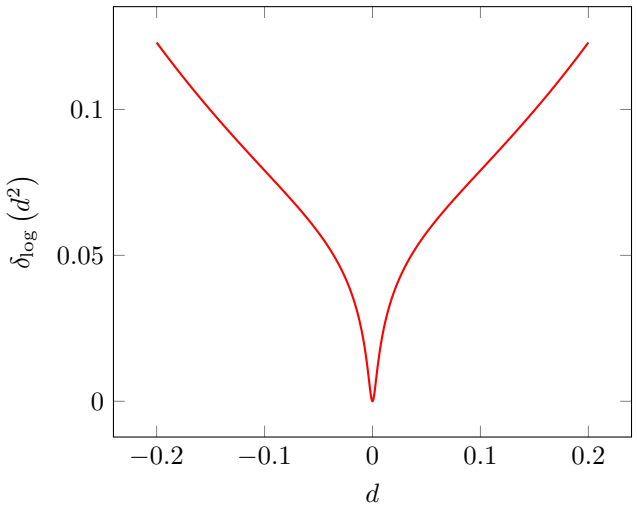

Fig. 4. Plot of the distance term used in the cost function. The concave shape introduced by the logarithmic term encourages the optimization to reach the target position as precisely as possible.

ϕ_{vel} is a penalty for part velocity at the final time step for any part in the scene. By including this term in the cost function, we ensure that the structure is stable after the part has been placed. We formulate this term as a quadratic penalty on the linear and angular velocity of parts:

$$\phi_{vel}(\mathbf{s}_T) = \min\left\{\frac{\frac{1}{P}\sum_{k=1}^{P}\left\|\mathbf{v}_{lin}^k\right\|_2^2 + \left\|\mathbf{v}_{ang}^k\right\|_2^2}{\phi_{vel}^{\max}}, 1\right\}$$

where $\mathbf{v}_{lin}^k$ is the linear velocity of the center of part k at the final time step, $\mathbf{v}_{ang}^k$ the angular velocity, respectively, and $\phi_{vel}^{\max}$ is again a hyperparameter.

The intermediate cost $\phi_i(\mathbf{s}_t, \mathbf{a}_t)$ consists of a constant term and penalty on the acceleration of the end-effector:

$$\phi_i(\mathbf{s}_t, \mathbf{a}_t) = \alpha_{time} + \alpha_{acc}\phi_{acc}(\mathbf{s}_t, \mathbf{a}_t)$$

The first term encourages the learner to complete the episode quickly, while the second term is defined as follows:

$$\phi_{acc}(\mathbf{s}_t, \mathbf{a}_t) = \frac{1}{\phi_{acc}^{\max}}\left(\left\|\left(\mathbf{v}_{lin}^{ee}\right)_t - \left(\mathbf{v}_{lin}^{ee}\right)_{t-1}\right\|_2^2 + \left\|\left(\mathbf{v}_{ang}^{ee}\right)_t - \left(\mathbf{v}_{ang}^{ee}\right)_{t-1}\right\|_2^2\right) \quad (2)$$

where $\left(\mathbf{v}_{lin}^{ee}\right)_t$ and $\left(\mathbf{v}_{ang}^{ee}\right)_t$ are the linear and angular velocities of the end-effector at time t, respectively. The purpose of this term is to encourage smooth movements in order to reduce energy consumption and decrease

the likelihood of the part being accidentally dropped. Additionally, we found the contact simulation between the part and the gripper to be less stable when the grippers movement is jagged.

b) Observation space: The observation $\mathbf{o}$ of the agent consists of the joint angles and velocities of both the robot arm and the gripper and the end-effector pose and velocity in cartesian space. While the observations eventually would also include the images from the LiDAR camera and DIGIT sensors, we simplify the task for now by giving the agent access to the true poses of the current part and the k parts closest to the target position. Since the target pose of the part is different for every part, we also include it in the observation and hence train a goal-conditioned policy.

While it is straightforward to find vector representations of all the state attributes mentioned above, some of them need a bit of tweaking to work well with neural network policies. Firstly, the joint positions of the robot could for example be represented by the joint angle in radians. However, then they either would not be unique as angles are 2π periodic or if they are restricted to lie within e.g. $[-\pi, \pi]$, there would be a discontinuity in the representation between $-\pi$ and π. We deal with this problem by representing each angle φ in a continuous 2-dimensional fashion, namely as $(\sin(\varphi), \cos(\varphi))^T$. While this increases the observation dimensionality, we found it to lead to a more stable training.

Finally, a lot of the observed properties are poses of some kind (e.g. part poses or the end-effector pose), which consist of a 3D position and orientation each. There are many ways to represent orientations in 3D, the most popular of which are probably Euler angles and quaternions. Euler angles suffer from the disadvantage that they are neither unique nor is their mapping to the space of 3 dimensional rotations continuous [26]. While quaternions are almost unique (there are always two quaternions that describe the same rotation), their mapping to the rotation space is also not continuous [26] and they are hence not well suited to be used as input for neural networks. Instead, we follow Zhou et al. [26] and use rotation matrices to represent orientations. Since rotation matrices are orthogonal, their entries are highly redundant and could in theory be reduced from 9 to 5 without loosing information. Zhou et al. found, however, that using two full columns of the rotation matrix yielded the best results. Hence, instead of 3 or 4 dimensions, we use a continuous 6 dimensional representation of orientations in 3 dimensional space.

F. Algorithm

We decided to use the model-free reinforcement learning algorithm Twin Delayed DDPG (TD3) [27] to learn the controller since there already exists a stable implementation as part of the `stable-baselines3` [28] project and TD3 is said to require relatively little hyperparameter tuning [29]. TD3 is a successor of the well-known algorithm Deep Deterministic Policy Gradients (DDPG) [30].

Note that the following section is formulated with rewards r instead of costs ϕ to match the formulation in the original paper [27]. The use of TD3 with our cost formulation from subsection III-E is straightforward since costs are nothing but negative rewards: $r = -\phi$.

TD3 learns two Q-functions Q_{θ_1} and Q_{θ_2} and a policy π_ψ, which are parameterized by neural networks. To make learning more stable, TD3 uses separate target Q-functions $Q_{\theta'_1}$ and $Q_{\theta'_2}$ and a target policy $\pi_{\psi'}$ to update the current Q-functions Q_{θ_1} and Q_{θ_2}. The parameters of the target networks are updated to slowly track the parameters of the current networks:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
$$\psi' \leftarrow \tau\psi + (1-\tau)\psi'$$

with $\tau \in (0, 1)$ being a hyperparameter.

TD3 maintains a replay buffer, in which it stores previously seen transitions as tuples (s, a, r, s'). At each update step TD3 samples a random minibatch of N transitions (s_j, a_j, r_j, s'_j) from the replay buffer. Both Q-functions Q_{θ_1} and Q_{θ_2} are then trained to minimize the error of the Bellman equation. To counter the overestimation bias that is present in DDPG, both target Q-functions $Q_{\theta'_1}$ and $Q_{\theta'_2}$ are evaluated and the smaller of the two values is used as target. This results in the following loss function:

$$L(\theta_i) = \frac{1}{N}\sum_j (y_j - Q_{\theta_i}(s_j, a_j))^2$$
$$y_j = r_j + \gamma \min_{i=1,2} Q_{\theta'_i}(s'_j, \tilde{a}_j)$$

where $\tilde{a}_j$ denotes the action chosen by the target policy $\pi_{\psi'}$ plus Gaussian noise:

$$\tilde{a}_j = \pi_{\psi'}(s'_j) + \epsilon$$
$$\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -\epsilon_{max}, \epsilon_{max})$$

The noise is added to enforce the notion that in a given state similar actions should result in similar values.

The policy π_ψ is trained to maximize the agent's performance J:

$$J(\psi) = \mathbb{E}\left[\sum_{t=1}^{T} \gamma^t r(s_t, \pi_\psi(s_t))\right]$$

This optimization is done by estimating the policy gradient with Q-function Q_{θ_1}:

$$\nabla_\psi J(\psi) \approx \frac{1}{N}\sum_j \nabla_a Q_{\theta_1}(s, a)|_{s=s_j, a=\pi_\psi(s_j)} \nabla_\psi \pi_\psi(s)|_{s=s_j}$$

IV. Rhino 6 Integration

This project is part of a cooperation between IAS and the Digital Design Unit (DDU) of the architecture department, with the long term goal of automating design and construction tasks. Currently, the process of designing buildings and other structures is typically done manually in a 3D-CAD tool, like `Rhino 6` [31]. A team of the DDU is working on the challenging task of automating the design of structures using reinforcement learning. Since it is possible to create arbitrarily complex structures in modern 3D-CAD programs,

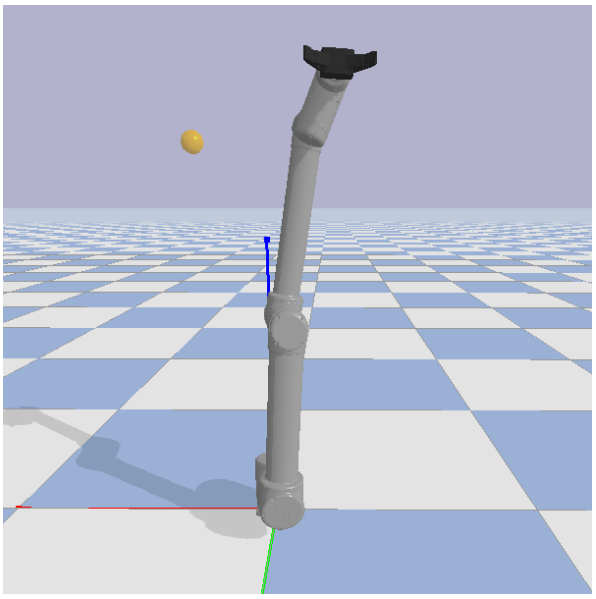

Fig. 5. The *path finding* task in simulation: the robot starts in a randomly perturbed upright position and has to reach the target position (marked with a yellow sphere) with its gripper.

they simplified the problem to creating structures out of a set of predefined modules (parts). The objective is then to create a construction plan consisting of the pose of each part and the order of placement that minimizes some cost function. These construction plans can then be used to construct the structure automatically and potentially report information about the constructability back to the architect. It is also thinkable to directly use constructability as a cost measure for the reinforcement learning algorithm creating the design in the future.

One of the key factors in rating a structure's quality is stability, which can be evaluated using a physics simulator like `CoppeliaSim` or `PyBullet`. While it is possible to try to construct each structure with the robot in order to assess its stability, the resource and time requirements would be high. Especially since most potential structures are already unstable without any kind of robot intervention, it is reasonable to filter those out without going through the effort of trying to construct them.

`Rhino` itself does not offer any kind of physics simulation off the shelf but it allows to write extensions in Python using the `Grasshopper` [32] plugin. `Grasshopper` allowed us to contribute to the DDU team's efforts by integrating the `PyBullet` simulator into `Rhino`. Specifically, this integration allows to simulate a modular structure for a fixed amount of seconds and visualize the resulting poses of all parts. The results of this simulation can either be used to manually refine a structure until it is stable or as a cost measure for a structure generating reinforcement learning algorithm.

V. Evaluation

We evaluated our approach on two different tasks: *path finding* and *single part construction*. The *path finding* task is a toy-task we used to check the sanity of our environment and the algorithms used. As further described in subsection V-A, the objective here is to move the gripper to a varying target position. In the *single part construction* task, further described in subsection V-B, the robot has to place a single part in an otherwise empty environment. Hence, this task can be seen as a first step towards autonomously constructing complex modular structures. We approach both tasks using the `stable-baselines3` [28] implementation of TD3. The same hyperparameters are used for both problems. The values of all these hyperparameters are listed in appendix A.

A. Path Finding

In this task, the robot arm always starts in a slightly randomly perturbed upright position as visible in Figure 5. The objective is to move the end-effector as close as possible towards a target position within a time window of 2 seconds. This target position is drawn uniformly randomly from a sphere with 30cm radius placed around 1.15m above the ground. As a cost function we use a linear combination of the end-effector acceleration penalty introduced in Equation 2 and the distance function of Equation 1 on the Euclidean distance to the target position:

$$\phi_{path}\left(\tau\right) = \frac{1}{T}\sum_{t=1}^{T} 0.95 \cdot \phi_{ee}\left(\mathbf{s}_t\right) + 0.05 \cdot \phi_{acc}\left(\mathbf{s}_t, \mathbf{a}_t\right)$$

where $T = 40$ is the number of steps. We define the gripper distance cost ϕ_{ee} as follows:

$$\phi_{ee}(\mathbf{s}_t) = \frac{\delta_{\log}\left(\|\mathbf{p}^{tg} - \mathbf{p}_t^{ee}\|_2^2\right)}{\phi_{ee}^{\max}}$$

where $\mathbf{p}^{tg}, \mathbf{p}_t^{ee} \in \mathbb{R}^3$ are the target and current end-effector positions, respectively and set $\phi_{ee}^{\max} = 0.4$.

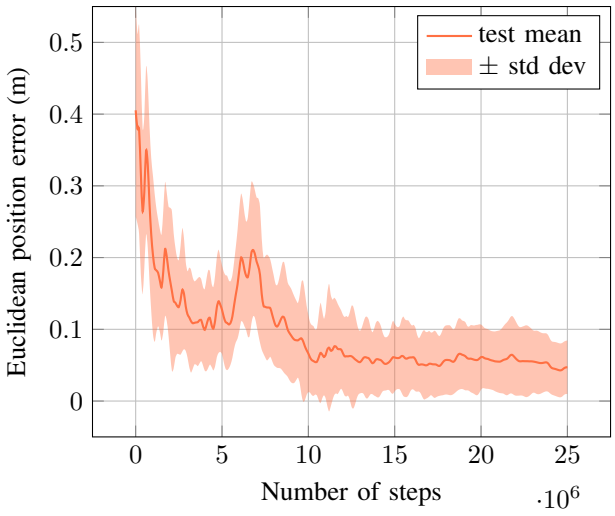

Fig. 6. *Path Finding* task: evolution of the gripper position error in the final time step of an episode (2s) over the course of the training using TD3. *Number of steps* refers to the total number of simulator steps made at the respective point in the training. The data was obtained by evaluating the model on 100 unseen test configurations every 100,000 steps. We started with a learning rate of 10^{-4} and multiplied it by a factor of $\frac{1}{\sqrt{10}}$ at 50%, 75%, 90%, and 95% of the time step limit (25 million).

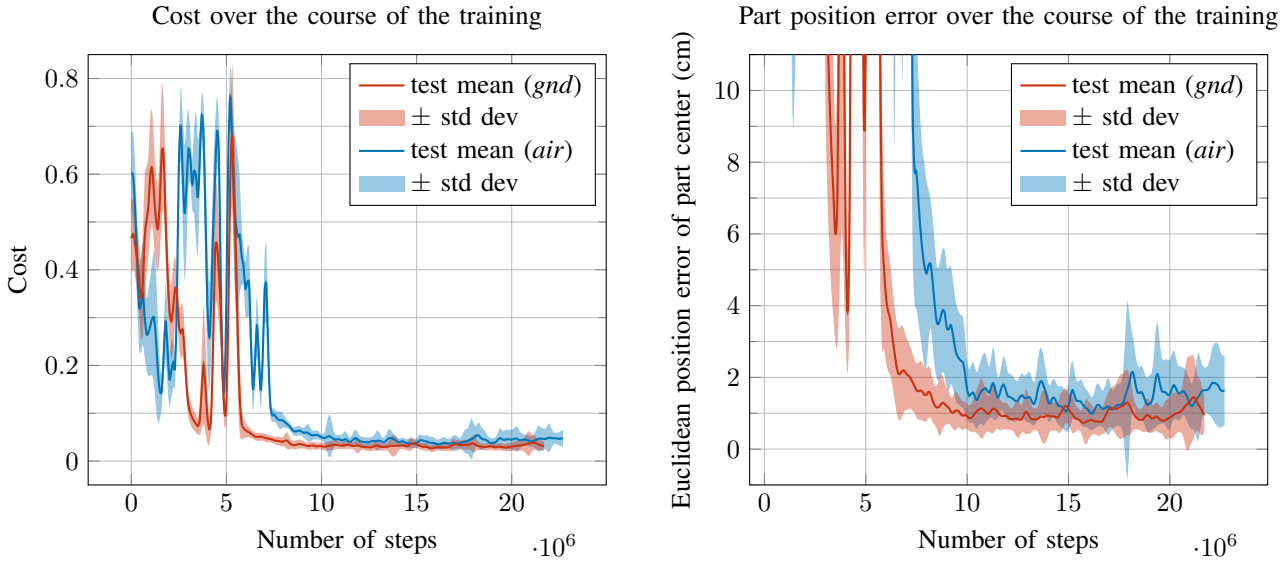

Fig. 7. *Single Part Construction* task: evolution of the cost function and the part center position error in the final time step of an episode (5s) over the course of the training using TD3. Visible in the plot are two runs: *gnd*, where the part is on the ground at the start of each episode and *air*, where the part starts 30cm above the ground. *Number of steps* refers to the total number of simulator steps made at the respective point in the training. The data was obtained by evaluating the model on 100 unseen test configurations every 100,000 steps. We started with a learning rate of 10^{-4} and multiplied it with a factor of $\frac{1}{\sqrt{10}}$ every 5 million steps, which likely explains the sudden convergence at the 5 million step mark. The reason the training stopped at different step counts for each run was due to the time limit of 72h being reached before the maximum step count of 25 million could be reached.

Figure 6 shows the results of TD3 on the *path finding* task. As visible in the graph, TD3 is able to reduce the mean displacement of the gripper to its target location to 5cm at the end of an episode. The slight descent at the end of the graph also seems to indicate that further improvement is possible by continuing the training with a smaller learning rate. However, as this is only a toy task and our simulations are generally fairly time and computation intensive, we decided not to continue the training and rather focus our resources on learning manipulation skills.

B. Single Part Construction

In this task, the robot has to perform the *placing* step on a randomly generated structure consisting of a single part, measuring $30 \times 20 \times 10$ cm. Since there is only one part, the only difference between the structures is the pose of that part. We generate each training structure by placing the part uniformly randomly on the bottom a cube of 60cm side length while keeping the orientation constant. The robot starts with the part already grasped in the correct orientation around 50cm from its target location. As described in subsection III-E, the objective of this task is to place the given part as precisely in its target pose as possible, in a way that it does not move at the end of the episode.

Figure 7 shows the course of the training for two different instances: in the first the part starts laying on the floor (marked as *gnd*) and in the second the part is around 30cm above the ground when the episode starts (marked as *air*). The starting configuration of the former instance is visualized in Figure 1, where the transparent part marks the target location of the part. In both instances we use $\alpha_{pose} = 0.65$, $\alpha_{vel} = 0.3$, $\alpha_{acc} = 0.04$, and $\alpha_{time} = 0.01$, as well as $\phi_{pose}^{\max} = 0.5$, $\phi_{vel}^{\max} = 1 + \pi$, and $\phi_{acc}^{\max} = 100$. 10,000 randomly generated structures serve as training data while 100 unseen structures are used for evaluation every 100,000 steps.

As visible in the graph, the mean final part position error is around 1cm for the *gnd* configuration and around 1.7cm for the *air* configuration. While a placement accuracy of 1–2cm is not perfect yet and could likely be improved by hyperparameter tuning, we believe that these results validate the feasibility of our approach and give reason to move on to constructing structures consisting of multiple parts.

VI. Conclusion and Future Work

In this work we tackled the problem of autonomous construction of modular structures in simulation. We separated the problem of placing a part into 3 stages: *grasping*, *transporting*, and *placing* and focused on the latter step. During our evaluation we showed that reinforcement learning can be used to learn to place parts in an otherwise empty structure with reasonable accuracy. While the accuracy presented in Figure 7 can likely be improved with more hyperparameter tuning, our main focus will lie on constructing structures with multiple parts in the future.

One issue of our implementation is the fairly high training time required in each run. The training presented in subsection V-B was stopped after it reached the time limit of 72h before reaching the target step count. Considering that a higher number of parts in the scene not only increases the task complexity but also the computational effort of the simulator at each time step, much higher training times are to be expected when learning to construct more complex structures. While increasing the training time limit is the most straightforward way of dealing with this problem, it

comes at the drawback that hyperparameter tuning becomes prohibitively expensive in terms of time. A better, yet more complex solution would be to optimize the learning procedure in order to increase the number of training steps per second. Since the bottleneck of the training are the simulator rollouts, which are currently drawn in a sequential way, we will focus on extending our implementation to allow parallel drawing of rollouts from multiple simulator instances. As we are training on a computer cluster with many CPU cores, we expect a substantial increase in performance from parallelization.

Model-based reinforcement learning algorithms often tend to require less training samples than model-free algorithms [33]. Thus, a way to reduce the training time could also be to switch from the model-free algorithm TD3 to a model-based algorithm, such as PlaNet [34]. PlaNet has been shown to reach similar performance as model-free algorithms on several continuous control tasks, while needing 200 times less training samples on average. In future work, we will apply PlaNet to our task to investigate whether it is more sample-efficient than TD3 for this task.

Furthermore, we currently only evaluate the construction of structures consisting of a single part. While our cost function is already designed to work with multiple parts, there are other challenges that need to be tackled in order to construct larger structures. One major challenge is the inclusion of the previously placed parts into the robot's perception. Since every structure is different, the robot has to have some way of sensing where other parts in the structure are in order to place the next part without collision. One way of adding this information is to include depth cameras in the simulation and use a CNN in the policy to evaluate those. However, adding cameras introduces a series of new challenges, as some parts might be occluded and the robot now additionally has to learn to interpret visual signals. While eventually the real robot is going to use a camera to sense the existing structure, we will start by providing information about other parts in the scene directly to the policy in terms of positions and orientations for now. The challenge here is to deal with different numbers of parts in the scene and achieve invariance to permutations. Both of these challenges could be tackled by utilizing Graph Convolutional Networks [35].

Additionally, we plan to learn a controller also for the *grasping* step in the future. Learning a controller for grasping allows the agent to adjust its grasp to the part type and target position. Adjusting the grasp is crucial to be able to place parts at the boundary of the workspace or in areas where the grasp from above might be blocked by already placed parts.

Finally, the long term goal of this project is to be able to construct structures using a real robot. While the real robot is already operational, there are a number of challenges that need to be tackled when transferring our method to the real world. First of all, especially when contact is happening, simulators can be fairly inaccurate, which makes transferring the policy to the real system challenging as it first has to be fine-tuned on the new system dynamics. Furthermore, some measurements that are easy to obtain in simulation are not directly available in the real system. One example is the pose of the parts, which is used in the cost function and as sensor input to our robot and can only be obtained by evaluating the images of our depth camera. Finally, the simulation of the DIGIT sensors is likely to be inaccurate, as the deformation of the gel is not simulated in our approach. While we did some experiments with soft body simulation, we found the results to be very unstable and inaccurate and hence decided to refrain from simulating deformation.

REFERENCES

[1] S. Eaves, D. E. Gyi, and A. G. Gibb, "Building healthy construction workers: Their views on health, wellbeing and better workplace design," *Applied ergonomics*, vol. 54, pp. 10–18, 2016.

[2] H. Ardiny, S. Witwicki, and F. Mondada, "Are autonomous mobile robots able to take over construction? a review," 2015.

[3] "Universal Robot UR10," https://www.universal-robots.com/products/ur10-robot, Accessed: 2020-08-30.

[4] "Robotis hand RH-P12-RN," http://www.robotis.us/robotis-hand-rh-p12-rn, Accessed: 2020-08-30.

[5] V. N. Hartmann, O. S. Oguz, D. Driess, M. Toussaint, and A. Menges, "Robust task and motion planning for long-horizon architectural construction planning," *arXiv preprint arXiv:2003.07754*, 2020.

[6] D. F. Gomes, A. Wilson, and S. Luo, "Gelsight simulation for sim2real learning," in *ICRA ViTac Workshop*, 2019.

[7] "Intel RealSense LiDAR camera L515," https://www.intelrealsense.com/lidar-camera-l515, Accessed: 2020-08-30.

[8] M. Lambeta, P.-W. Chou, S. Tian, B. Yang, B. Maloon, V. R. Most, D. Stroud, R. Santos, A. Byagowi, G. Kammerer *et al.*, "DIGIT: A novel design for a low-cost compact high-resolution tactile sensor with application to in-hand manipulation," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 3838–3845, 2020.

[9] C. Melchiorri, "Slip detection and control using tactile and force sensors," *IEEE/ASME Transactions on Mechatronics*, vol. 5, no. 3, pp. 235–243, 2000.

[10] F. Veiga, H. van Hoof, J. Peters, and T. Hermans, "Stabilizing novel objects by learning to predict tactile slip," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 5065–5072.

[11] R. Calandra, A. Owens, D. Jayaraman, J. Lin, W. Yuan, J. Malik, E. H. Adelson, and S. Levine, "More than a feeling: Learning to grasp and regrasp using vision and touch," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3300–3307, 2018.

[12] H. Van Hoof, N. Chen, M. Karl, P. van der Smagt, and J. Peters, "Stable reinforcement learning with autoencoders for tactile and visual data," in *2016 IEEE/RSJ*

international conference on intelligent robots and systems (IROS). IEEE, 2016, pp. 3928–3934.

[13] S. Tian, F. Ebert, D. Jayaraman, M. Mudigonda, C. Finn, R. Calandra, and S. Levine, "Manipulation by feel: Touch-based control with deep predictive models," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 818–824.

[14] G. Sartoretti, Y. Wu, W. Paivine, T. S. Kumar, S. Koenig, and H. Choset, "Distributed reinforcement learning for multi-robot decentralized collective construction," in *Distributed Autonomous Robotic Systems*. Springer, 2019, pp. 35–49.

[15] Z. Kappassov, J.-A. Corrales, and V. Perdereau, "Tactile sensing in dexterous robot hands," *Robotics and Autonomous Systems*, vol. 74, pp. 195–220, 2015.

[16] H. Yousef, M. Boukallel, and K. Althoefer, "Tactile sensing for dexterous in-hand manipulation in robotics—a review," *Sensors and Actuators A: physical*, vol. 167, no. 2, pp. 171–187, 2011.

[17] W. Yuan, S. Dong, and E. H. Adelson, "Gelsight: High-resolution robot tactile sensors for estimating geometry and force," *Sensors*, vol. 17, no. 12, p. 2762, 2017.

[18] J. A. Fishel and G. E. Loeb, "Sensing tactile microvibrations with the biotac—comparison with human sensitivity," in *2012 4th IEEE RAS & EMBS international conference on biomedical robotics and biomechatronics (BioRob)*. IEEE, 2012, pp. 1122–1127.

[19] E. Rohmer, S. P. N. Singh, and M. Freese, "CoppeliaSim (formerly V-REP): a versatile and scalable robot simulation framework," in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013, www.coppeliarobotics.com.

[20] "Bullet real-time physics simulation," https://pybullet.org, Accessed: 2020-09-03.

[21] J. Matas, S. James, and A. J. Davison, "Sim-to-real reinforcement learning for deformable object manipulation," *arXiv preprint arXiv:1806.07851*, 2018.

[22] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, 1975.

[23] A. D. Laud, "Theory and application of reward shaping in reinforcement learning," Tech. Rep., 2004.

[24] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, https://ompl.kavrakilab.org.

[25] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," *CoRR*, vol. abs/1501.05611, 2015. [Online]. Available: http://arxiv.org/abs/1501.05611

[26] Y. Zhou, C. Barnes, J. Lu, J. Yang, and H. Li, "On the continuity of rotation representations in neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[27] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[28] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[29] "Reinforcement learning tips and tricks," https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html, Accessed: 2020-09-13.

[30] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[31] "Rhinoceros," https://www.rhino3d.com, Accessed: 2020-09-07.

[32] "Grasshopper - algorithmic modeling for rhino," https://www.grasshopper3d.com, Accessed: 2020-09-07.

[33] C. G. Atkeson and J. C. Santamaria, "A comparison of direct and model-based reinforcement learning," in *Proceedings of international conference on robotics and automation*, vol. 4. IEEE, 1997, pp. 3557–3564.

[34] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2555–2565.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

APPENDIX

A. Stable-baselines3 model hyperparameters

In Table I the hyperparameters that were passed to the TD3 implementation in `stable-baselines3` are given.

Parameter name	Value
gamma	0.99
buffer_size	500000
learning_starts	4096
batch_size	256
train_freq	1024
gradient_steps	100

TABLE I

PARAMETERS USED FOR THE TD3 ALGORITHM FROM STABLE-BASELINES3

For the policy and Q-functions we used a two-layer fully-connected network with ReLU activations and 400 and 300 neurons per layer. The action noise is distributed normally with $\mu = 0$ and $\sigma = 0.01$. The parameters that are not mentioned above were left at the default value.