MOTION PLANNING AND CONTROL

RoboBallet: Planning for multirobot reaching with graph neural networks and reinforcement learning

Matthew Lai[1,2]*, Keegan Go[3], Zhibin Li[2], Torsten Kröger[3], Stefan Schaal[3], Kelsey Allen[1]†, Jonathan Scholz[1]†

Modern robotic manufacturing requires collision-free coordination of multiple robots to complete numerous tasks in shared, obstacle-rich workspaces. Although individual tasks may be simple in isolation, automated joint task allocation, scheduling, and motion planning under spatiotemporal constraints remain computationally intractable for classical methods at real-world scales. Existing multiarm systems deployed in industry rely on human intuition and experience to design feasible trajectories manually in a labor-intensive process. To address this challenge, we propose a reinforcement learning (RL) framework to achieve automated task and motion planning, tested in an obstacle-rich environment with eight robots performing 40 reaching tasks in a shared workspace, where any robot can perform any task in any order. Our approach builds on a graph neural network (GNN) policy trained via RL on procedurally generated environments with diverse obstacle layouts, robot configurations, and task distributions. It uses a graph representation of scenes and a graph policy neural network trained through RL to generate trajectories of multiple robots, jointly solving the subproblems of task allocation, scheduling, and motion planning. Trained on large randomly generated task sets in simulation, our policy generalizes zero-shot to unseen settings with varying robot placements, obstacle geometries, and task poses. We further demonstrate that the high-speed capability of our solution enables its use in workcell layout optimization, improving solution times. The speed and scalability of our planner also open the door to capabilities such as fault-tolerant planning and online perception-based replanning, where rapid adaptation to dynamic task sets is required.

INTRODUCTION

In robotics applications such as welding (*1*), assembly (*2*), painting (*3*), construction (*4*), and object manipulation (*5*), robotic arms are programmed to perform many tasks in a shared workspace. To reduce the overall execution time and increase throughput, more robots can be placed in the same cell to perform tasks concurrently. Such high-density setups have the potential for reduced execution time with minimal increases in cost and space compared with duplicating entire workcells.

However, coordinating multiple robots efficiently in confined spaces is both theoretically and technically difficult because the planner must solve the task assignment, scheduling, and collision-free motion planning problems jointly, all of which have high combinatorial complexity, often in environments with complex obstacle geometries. This work addresses this challenge in a setting where multiple robots can perform all tasks in any order, which represents the most challenging scenario in terms of the size of the search space.

Finding a collision-free path in joint space that moves a robot end effector to a pose target is the motion planning aspect of the problem and has been conventionally addressed using sampling-based approaches such as rapidly exploring random trees (RRT) and its variants (*6*, *7*). However, although these sampling-based algorithms are designed to tackle high-dimensional configuration spaces and offer probabilistic completeness, their runtime scales exponentially in the dimensionality of the configuration space and the complexity of the obstacles (*8*). In practice, these techniques are suitable only for relatively small configuration spaces [two or fewer robots (*4*)] when attempting to plan with all robotics at the same time. One potential solution is to plan for each robot sequentially; however, this formulation introduces its own challenges because other robots would have to be treated as moving obstacles, which sampling-based motion planners also struggle with, and would require techniques such as converting paths of other robots into swept volumes, which then unduly limits the search space of the planner and might lead to suboptimal solutions or planning failures.

Similarly to the path planning problem, the task scheduling problem (to decide the order in which the tasks should be done) is also polynomial space (PSPACE) complete (*9*, *10*), formalized as the traveling salesman problem (TSP). Although exact solutions to the TSP are not computationally feasible for real-world problem sizes, many approximate solutions exist with varying optimality/efficiency trade-offs. The main challenge in adapting these solutions to task scheduling in our context is that, instead of having a fixed cost to move between each pair of tasks, as is the case in the classically formulated TSP, the joint-space trajectories required to move from task to task can be modified depending on the trajectories of other robots, and each potential trajectory has a different cost. Furthermore, there is often more than one inverse kinematics (IK) solution per task, either because the task is not fully constrained or the robot has more degrees of freedom (DoFs) than the task. In the case of seven-DoF robot arms, each six-DoF task pose has up to an infinite number of IK solutions. A good path planning algorithm must take the entire sequence of tasks into account when selecting the best IK solutions to use to solve each task.

The task allocation problem adds yet another layer of complexity. Finding an allocation of tasks to robots that minimizes the total execution time is conceptually similar to a multicontainer variant of the classical knapsack problem (*11*), except that we do not know the marginal cost of allocating an additional task to a robot without

[1]Google DeepMind, London, UK. [2]University College London, London, UK. [3]Intrinsic, Mountain View, CA, USA.
*Corresponding author. Email: matthewlai@google.com
†These authors contributed equally to this work.

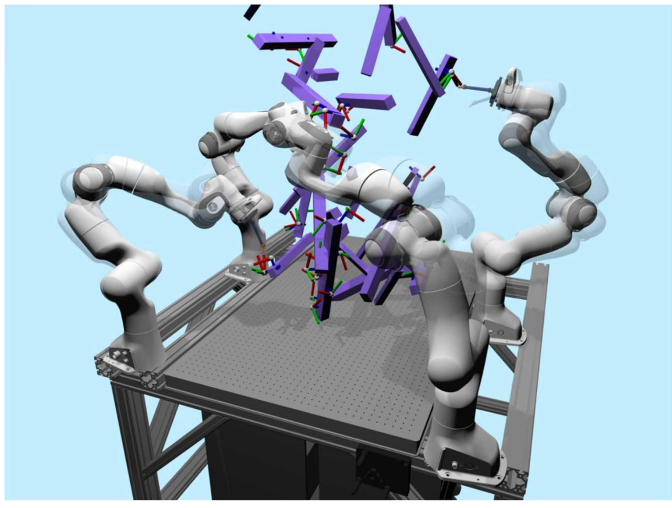
solving the entire planning problem, nor are the costs independent of other tasks being solved by the robots. Therefore, algorithmic solutions to the knapsack problem are not practically helpful in this setting.

As a result of the high asymptotic complexity of each of the subproblems, it has not been computationally feasible to combine existing algorithms with completeness guarantees to jointly solve high-density multirobot planning on real-world problem sizes because of their exponential scaling in the number of robots, number of tasks, and complexity of obstacles (*12*). Instead, existing approaches focus on breaking the problem down into manageable subproblems, solving them one at a time, often iteratively, to generate solutions with realistic compute requirements (*4*). However, this approach sacrifices potential solution optimality for computational feasibility and still does not scale to environmental complexities of real-world applications, given that the current state of the art reported scalability to only five robots and 10 tasks by iterating between a high-level task planner and a low-level motion planner, with some heuristics based on reachability and availability (*13*). By necessity, industry currently relies instead on manual trajectory planning with complex schedules of interlocks to ensure collision-free operation. These trajectories often take hundreds or thousands of hours to design by hand (*1*), and any change to the tasks or the environment requires time-consuming manual replanning.

To tackle this challenge, we present RoboBallet, a learned heuristics-based approach for a task and motion planning on randomly generated environments, capable of planning multiarm reaching trajectories in environments different from those seen during training, with arbitrary obstacle geometries, task poses, and robot placements. Task and motion planning is a large class of problems that require both high-level discrete scheduling and low-level continuous planning to solve. In this work, we focus on the reaching subclass for simplicity. Specifically, we address problems where scalability and efficient task scheduling is the main challenge, with no implicit or explicit task interdependencies and where each individual task is simple to accomplish. For extensions to manipulation tasks or problems with ordering/compatibility constraints, see the Limitations and future work section.

Our approach tackles the task allocation, scheduling, and motion planning problems jointly with no hand-designed simplifications common in existing approaches, such as presampling a fixed number of IK solutions, preallocating tasks to robots on the basis of heuristics, or planning subtrajectories for one robot at a time. Our key insight is to adopt a deep reinforcement learning (deep RL) approach that trains an agent to control the arms at each time step. This shifts expensive online computations to an offline phase, amortizing and shifting the compute requirement from the planning phase to the training phase, benefiting applications that require planning solutions on many similar setups or a setup with many variations. By training on configuration variations that we wish to support, for example, robot positioning and morphology, task types, or obstacle types, RoboBallet can quickly generate plans for a wide range of environments and novel configurations at test time without further training (Movie 1). After training, we can either simply execute these controllers in the real world, or, for safety, we can generate the solution trajectory by unrolling the policy step-by-step with a forward kinematic simulator so it can be inspected and potentially processed further.

However, naive applications of deep RL to multiarm task and motion planning would also fail to scale beyond trivial scenes, given



Movie 1. RoboBallet: Planning for multirobot reaching with GNNs and RL. We trained the model on environments with randomized robot placement, randomly generated obstacles, and randomly generated tasks. These randomized environments are generated per episode in training. Throughout the training process, the model interacts with about 1 million of these randomized environments. Each RGB (red, green, blue) coordinate frame on the obstacles represents an end-effector pose the robots need to reach. For testing, the evaluation setup used a fixed robot placement, real-world obstacles, and presampled tasks. Despite having never seen the evaluation robot placements, obstacles, or tasks during training, the model, trained on the randomized training environment, is able to transfer without any additional training to the evaluation cell and generates time-efficient and collision-free trajectories.

that the scene complexity still scales combinatorially and this leads to exponential scaling in the required size of the underlying deep neural network representing the agent's policy. The main contribution of this work is in the use of graphs to represent the state and the use of graph neural networks (GNNs) for RL policy and state-action value estimation. GNNs use weight sharing to allow scaling of the graph size without scaling the model complexity, similarly to how convolutional neural networks (CNNs) (*14*) have fixed kernel sizes independent of image sizes, and inference time complexity is quadratic in the number of robots—refer to the Discussion section for more details. The state graphs are constructed such that graph nodes represent robots, obstacles, and tasks and graph edges connect each task and each obstacle to each robot, and each robot to each other, supporting coordination among the robots and allowing each node to receive sufficient information about the environment to plan a collision-free path to tasks (Fig. 1).

With this formulation, RoboBallet directly and simultaneously controls multiple (here up to eight) seven-DoF robots, coordinates an up to 56-dimensional configuration space, and handles up to 40 shared tasks, with completely unconstrained robot allocation and task scheduling. The model decides at each 100-ms time step the target joint velocities of all joints in all of the robots, and our environment setup simply applies velocity and acceleration clamping, stops robots from colliding, and integrates the resulting velocities to generate joint configuration waypoints.

RoboBallet adopts an approach that prioritizes scalability and trades off theoretical guarantees to enable real-world deployment on a much larger scale of problems (test with up to eight robots in dense environments) while demonstrating empirically high-quality solutions. RoboBallet opens up exciting possibilities for many industrial

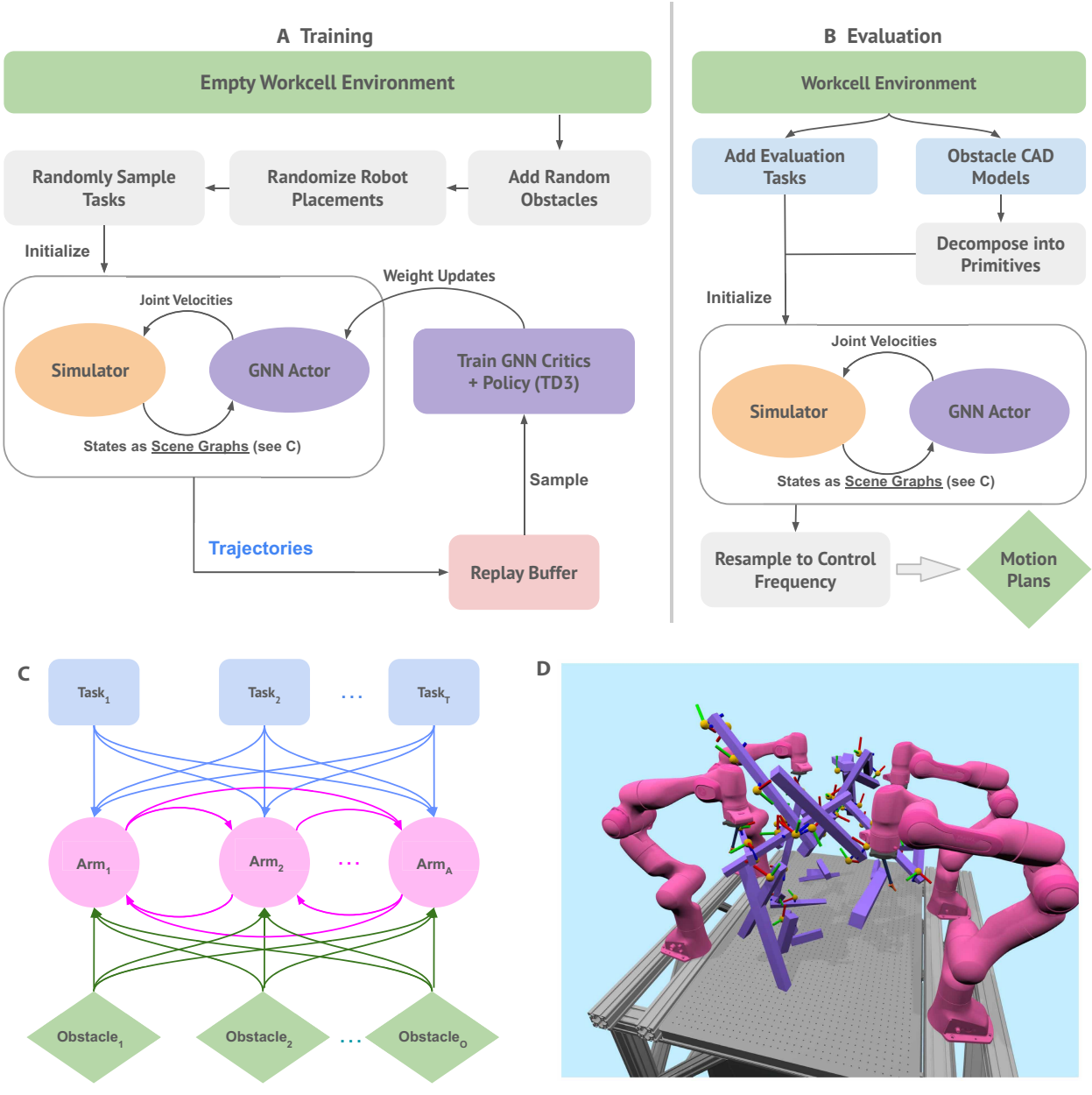


Fig. 1. Training and evaluation pipeline. (**A**) Training pipeline. Each episode starts with an empty table, upon which we add random cuboid obstacles, randomize robot placements, and randomly sample end-effector tasks. The GNN actor interacts with the kinematic simulator to try to complete the tasks. The trajectories are stored in a replay buffer and used to train a critic and policy function using TD3. (**B**) The evaluation pipeline, where externally defined tasks, together with obstacles, decomposed into collision primitives, are used to construct the simulated environment, with which the GNN interacts in the same way as in training. (**C**) Topographical view of the scene graph, containing robot, task, and obstacle nodes, with bidirectional edges between robots and unidirectional edges from each task and obstacle primitive to each robot. (**D**) Simplified training cell with a reduced number of obstacles and tasks. Each task is specified and visualized as a coordinate frame that the end effector of the robots must match, where the blue/*z* axis is in line with the radial axis of the end effector.

applications, such as workcell layout optimization, fast replanning in case one robot fails, and handling run-to-run variations in workpiece pose using image-based pose estimation to capture the workpiece pose for the planner, as well as environment changes using three-dimensional (3D) environment reconstruction with cameras or LIDAR (light detection and ranging) systems. RoboBallet is an enabling technology that opens the door to these important capabilities that have thus far not been possible because of the high cost and latency of manual task and motion planning in the multirobot manufacturing setting.

RESULTS

To demonstrate the capability of RoboBallet, we trained the model over several days on randomly generated workcells of different sizes in simulation and validated the performance of trained models in a

simulated workcell with eight robot arms, a workpiece constructed out of aluminum struts, and 40 tasks (Movie 1). To demonstrate the feasibility of real-world execution of our generated trajectories, we trained a model with four robot arms using exactly the same framework, and we validated the generated trajectories on a real workcell with minimal postprocessing. We observed that our models generated high-quality trajectories in seconds and can effectively use many robots to shorten execution time even on the largest settings tested (eight robots, 40 tasks, 30 obstacle primitives), on a single GPU (graphics processing unit), and in both a real and a simulated multiarm workcell.

To showcase one of the most important capabilities enabled by a fast automated planner, we used RoboBallet, combined with a black box optimizer, to perform workcell layout optimization (optimizing placement of robots given fixed sets of tasks and obstacles). We saw that this optimization setup improved task execution times by up to 33% compared with nonoptimized robot placements in our test cell on our fixed test set of tasks.

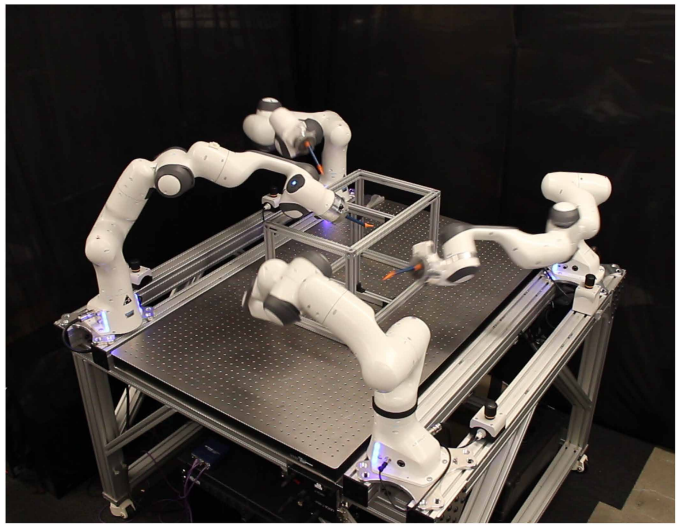
The training environments had either four or eight Franka Panda seven-DoF robots: four mounted on a table and an additional four mounted on the ceiling mirroring the four on the table in the case of eight robots (Movie 1). At the beginning of each episode, we first randomized the location of the robots along virtual 1.6-m-long rails parallel to the side of the table, only rejecting configurations in collision at the start. Then, we sampled and placed 30 cuboid obstacles with randomly generated positions and orientations, only rejecting obstacles that would be in collision with the robots in their start configurations. Next, we uniformly sampled tasks on the surfaces of the obstacles with a small offset away from the surfaces, where each task is a position and an orientation that a robot end effector must reach. We then tried to generate IK solutions from each robot to the task pose and rejected the task if it did not have at least one collision-free IK solution from at least one robot. Information about the IK solutions and reachability was not given to the model.

In the motion planning literature, the execution time (the durations of trajectories) is the primary metric to be optimized for. However, because no published approach exists that can scale to our problem sizes in the multiarm task and motion planning setting with realistic execution time, we provide a comparison against a baseline method for trajectory optimality in a simplified setup. This is given in the "Optimality of generated trajectories" section below. The rest of the Results section reports on the scalability and generalizability of the algorithm on the full problem. For a qualitative evaluation, please refer to Movie 2 and movies S1 to S4 showing generated trajectories.

Optimality of generated trajectories

In both real-world applications and academic benchmarks, trajectories are most commonly evaluated by their time cost (duration). In this section, we tested our approach on a set of smaller and simplified problems where we could generate near-optimal trajectory solutions through exhaustive search and preassignments of tasks to robots. The preassignment is necessary because exhaustively testing all possible allocations in addition to schedules is not computationally feasible for all but the smallest problems.

Specifically, we generated 10 sets of 20 tasks, where each set contains five tasks for each of the four robots. We filtered the sampled tasks to ensure that each can be reached by only one robot. To generate solutions that are as close as possible to optimal to compare against, we needed to tackle the same four challenges that our model needs to tackle simultaneously—task allocation, task scheduling, IK solution selection, and motion planning. Optimal task allocation is ensured by problem design (only one allocation is possible). Optimal task scheduling is done by exhaustively testing all possible task schedules. In the case of seven-DoF robot arms, each six-DoF task pose has up to an infinite number of IK solutions when poses are within the feasible reachability workspace, and choosing good IK solutions in the context of each trajectory is also important in minimizing total trajectory cost. We present multiple minimum costs based on different numbers of IK solutions sampled (and exhaustively tested) to compare against. Last, because a computationally feasible optimal motion planning algorithm for seven DoFs does not exist, we used RRT-Connect (*15*) with shortcutting, which is a widely used motion planning algorithm known for generating low-cost trajectories in practice. We first planned for each robot separately and then, for each task set, took the cost of the longest single-robot subtrajectory as the cost of the multirobot trajectory.



Movie 2. Four Franka Panda robots executing a plan generated by RoboBallet.

In Fig. 2, we first showed that, with a random task schedule for each robot, the trajectory costs were much higher than with optimal task scheduling, demonstrating the importance of good task scheduling even in this simplified setting. Then, we showed that, by increasing the number of IK solutions sampled (and selecting the best IK solution for each task in each trajectory), the trajectory costs also decreased substantially, demonstrating the importance of choosing good IK solutions. Last, we showed that RoboBallet generated trajectories with costs that are competitive with the baseline approach at eight IK samples per task. These results showed that, although an RL-based approach cannot offer the probabilistic completeness that sampling-based methods do, the costs of trajectories generated in our problem settings are comparable to our baseline results in practice. The baseline also uses orders of magnitude more computation and relies on several simplifying assumptions, unlike RoboBallet.

There are two notable differences between the baseline and RoboBallet setups that need to be taken into account when interpreting these results. First, the RRT-Connect algorithm builds a tree from both the start and the finish poses, generating a valid path that puts the end effector at exactly the goal pose, whereas our approach uses

a tolerance of 2.5 cm and 15°. In a real-world application, our approach would integrate a linear driver to achieve final docking. This would be implemented as part of the environment, where the RL agent only receives a reward if the final docking is successful, ensuring that the agent chooses solution poses where the final docking is feasible. This integration is excluded from the current study for simplicity, and its inclusion would result in a marginal increase in trajectory costs for RoboBallet. Second, the RRT-Connect results are generated with each robot in isolation (other robots removed), whereas RoboBallet moves all robots at the same time. This means that the RRT-Connect results here ignore potential collisions between robots during task execution. If collisions did occur in practice when all robot trajectories are executed simultaneously, this would require adjustment of task sequences to create a collision-free multirobot trajectory.

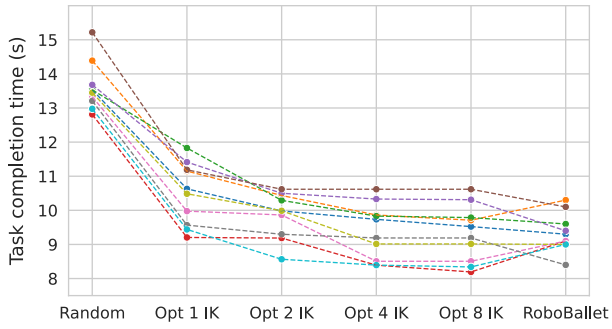


Fig. 2. Comparison against RRT-Connect plus exhaustive task scheduling baseline in a simplified setting. Each color represents one task set with 20 tasks (five per robot). All columns except "RoboBallet" use RRT-Connect and shortcutting for motion planning. The "Random" column averages over all possible task schedules. The "Opt [*N*] IK" columns use *N* IK solutions per task and the optimal task schedule and IK solution selection by exhaustive search.

Training time scaling in the number of robots and tasks

As discussed in the Introduction, classical approaches scale exponentially in the number of robots and number of tasks, and this prevents them from being applicable to large settings. In Fig. 3, we demonstrated that our approach scales much more favorably at training time. Quadrupling the number of tasks only required marginally more training steps to converge, despite the 4× difference in the number of tasks and therefore in the difficulty of the scheduling problem. This does not correspond to equal amounts of computation. The number of edges in our scene graph is linear in the number of tasks, and therefore, each training step takes up to four times more computation in the 40-task case compared with the 10-task case.

The second important aspect of computational scalability pertains to the number of robots. Similarly to the scaling with number of tasks, we saw that the training curves converged at a similar number of training steps. The theoretical asymptotic time complexity is $O(N_{\text{robots}}^2)$ for N_{robots} number of robots, given that the robot nodes are fully connected with each other for coordination. In practice, quadratic scaling here is not a problem because there is an inherent limit in the density of placement of robots, beyond which there would be no benefit in execution time given that it becomes more

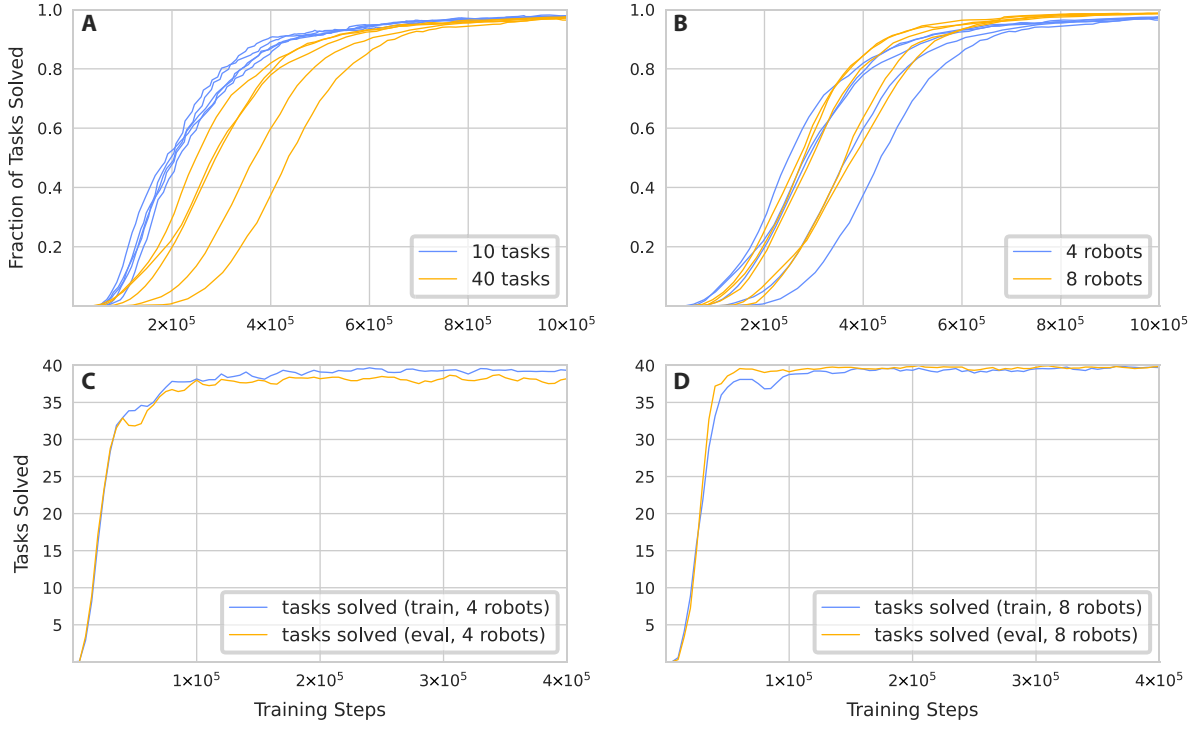


Fig. 3. Training curves for scalability and generalizability. (**A**) Training curves for 10 versus 40 tasks, four robots. (**B**) Training curves for four versus eight robots, 40 tasks. The number of training steps required to reach convergence does not scale exponentially. However, in terms of asymptotic time complexity, the amount of computation required is linear in the number of tasks and quadratic in the number of robots. (**C**) Results on randomly generated training cells versus hand-designed evaluation cell over training, four robots. (**D**) Results with eight robots.

and more difficult for the robots to avoid one another. For very large workcells with more than a dozen robots, the robots would be positioned such that they would not all have overlapping work volumes. If two robots do not have overlapping work volumes, then they do not need to coordinate with each other because they cannot complete the same tasks or collide with each other. Therefore, in the asymptotic case, assuming that we have a limiting density in placement of robots, and add more robots by expanding the workcell without connecting robots that do not have overlapping work volumes, the algorithm's asymptotic time complexity falls back to $O(N_{robots})$.

In practice, because of the performance characteristics of GPUs, the runtime differences are even smaller than suggested by the asymptotic complexity. In our case, each training step took 33 ms for the four-robot–and–10-task case, 52 ms for the four-robot–and–40-task case, and 80-ms for the eight-robot–and–40-task case, with a batch size of 128 state transitions per training step. Figure 3 shows the training curves with three different configurations—four robots with 10 versus 40 tasks and 40 tasks with eight robots.

Inference time scaling

At inference time, even the largest setup with eight robots and 40 tasks only took about 0.3 ms per planning step on an NVIDIA A100 GPU, allowing for greater than 300× real-time planning at 10-Hz time steps. On a single Intel Cascade Lake CPU core, each step takes about 30 ms, which is still three times faster than real time with 10-Hz time steps. The planning process consists of one inference and one collision check for the entire scene at each time step.

Efficient use of multiple robots

Another important aspect of scalability is the reduction in task completion time as the number of robots increases. The ability to control more robots is only meaningful if the controller is able to solve tasks more quickly when given more robots to use. For this experiment, we first randomly sampled 10 sets of robot placements, each with four robots—two on either side of the table and two on either rail on the ceiling. For each set of placements, we sampled a set of 20 tasks that can be solved using those four robots in those randomly sampled placements. These sets of tasks made up the evaluation configurations, each of which is a task set that is guaranteed to be solvable using four robots. Then, we performed experiments to find the shortest trajectory lengths we can generate for each evaluation configuration using four, five, six, seven, and eight robots. For each number of robots and evaluation task set, we performed placement optimization (see the "Workcell layout optimization" section) to find the best placements for the given number of robots and set of tasks and recorded the minimum trajectory length achieved. Figure 4 shows the best execution time achieved using four to eight robots with optimized placements for each evaluation configuration. The plot shows strong execution time scaling from four to eight robots, with the average execution time reducing from ~7.5 s using four robots to 4.5 s using eight robots (a reduction of 60%). This is approximately optimal scaling, given that in the four-robot case, with uniform task distribution, each robot needed to perform five tasks, whereas in the eight-robot case, four of the robots needed to perform two tasks and the other four needed to perform three tasks.

Generalizability to hand-designed environments

Our models were trained with randomized robot placements, randomized obstacles, and randomized tasks to enable maximum flexibility at evaluation time. Movie 1 shows the evaluation setup, where the obstacle is a hand-designed shape made out of aluminum struts, robot placements are fixed to the locations in our real test workcell, and the tasks are randomly sampled on the faces of the obstacle. We generated 10 sets of 40 tasks for evaluation, which were not used in training. Figure 3 (C and D) shows performance on these evaluation configurations compared with that in the training environment over the training process. Our results show that the model, trained on randomly generated environments, can generalize very well to an unseen hand-designed evaluation cell.

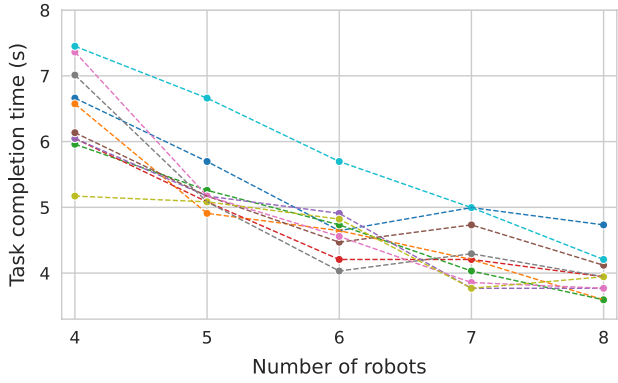


Fig. 4. Task completion time scaling with the number of robots (optimized placements). Shortest trajectory execution times achieved with the 10 evaluation configurations using optimized placements of four to eight robots. Each color represents one evaluation configuration.

Workcell layout optimization

A fast task and motion planner enables use cases that are not currently possible with manual planning. One such use case is workcell layout optimization. In this experiment, we trained the model on fully randomized environments as described above and used the Google Vizier (*16*) black box optimizer with a modified Gaussian process bandit algorithm (*17*) to find the best placement of robots to achieve the shortest execution time on each of the 10 sets of fixed evaluation tasks. Figure 5 shows the execution time improvements from the optimized placements compared with the common default placements with one robot at each corner of the table, as well as the breakdown of time spent on each task by each robot, both with motion plans generated using RoboBallet. We saw that there is universal improvement in trajectory execution time over all evaluation configurations and that the improvement comes primarily from the more even redistribution of tasks to robots and secondarily from reduced motion time to reach the task poses. In setups where the task completion time is long relative to the motion time, and therefore the even distribution of tasks is more crucial, the benefit of placement optimization would be even more pronounced. We chose to use a dwell time of 0.5 s (simulating task completion time) at each task here to show the result on a more balanced setup where both motion time and task completion time contribute substantially to the total solution length.

DISCUSSION

Our results demonstrate that GNNs combined with RL enable efficient solutions to the joint problem of task allocation, scheduling,

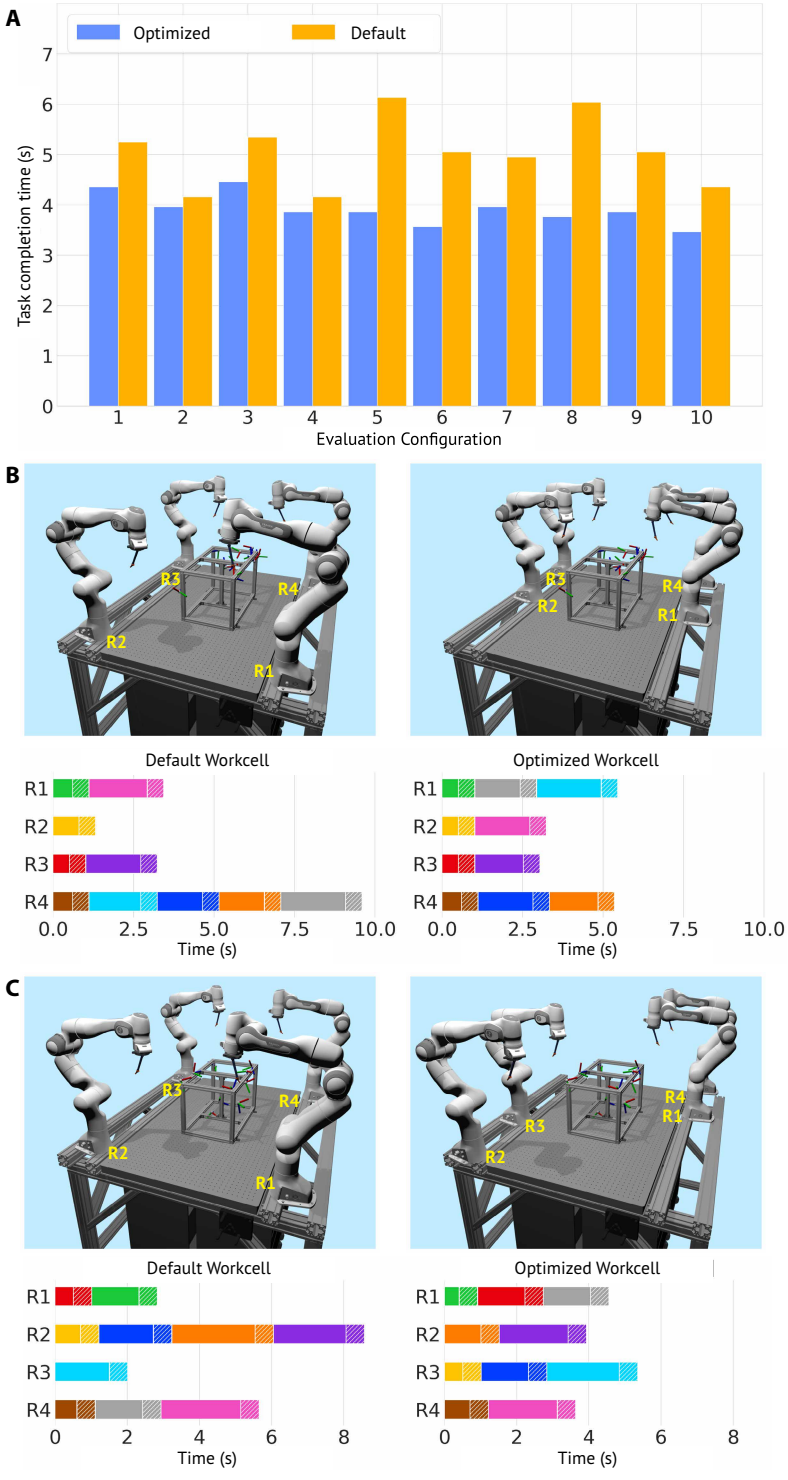


Fig. 5. Layout optimization of the workcell. (**A**) Execution time comparison between optimized and default layouts on the 10 evaluation configurations (each with 10 tasks). (**B**) Execution time breakdown of configuration #5, the configuration with the biggest improvements. Each bar represents the tasks done by a robot in the trajectory, and each block represents the motion time and dwell time (simulating time needed to perform work at the site, e.g., a spot welding task). Each task has a unique color so that the redistribution can be observed directly. (**C**) Same for configuration #8.

and motion planning. Although the conventional sampling-based approaches offer probabilistic completeness, a valuable property in certain contexts, our approach is able to scale to much larger real-world problem sizes, delivering high-quality solutions with minimal computation requirements, especially at evaluation time. This trade-off between theoretical completeness and scalability is critical for practical applications in large-scale, dynamic environments.

The high scalability of our method comes from the fact that in terms of model complexity of our learned functions, the network only has to learn the interactions between different types of nodes (tasks to robots, obstacles to robots, and robots to robots) once. This results in O(1) scaling in model complexity (model size) with respect to the number of robots, number of tasks, and complexity of obstacles. This means that, theoretically, the amount of training data needed does not scale with the size of the environment, and we observed that to be broadly true in our experiments. In terms of asymptotic time complexity, computation is dominated by edge processing, which is proportional to the number of edges. We have $N_{\text{robot}} \times N_{\text{robot}}$ robot-to-robot edges, $N_{\text{robot}} \times N_{\text{robot}}$ task-to-robot edges, and $N_{\text{robot}} \times N_{\text{obstacle}}$ robot-to-obstacle edges, for a total time complexity of $O({N_{\text{robot}}}^2 + N_{\text{robot}}N_{\text{task}} + N_{\text{robot}}N_{\text{obstacle}})$. This complexity is linear in both number of tasks and number of obstacles and quadratic in the number of robots, which is manageable because of the inherent limit in how many robots can be usefully placed in the same space.

A fast and automated task and motion planner enables many use cases that are previously not known to be practical because of the speed of existing planners. As an example that we explored in this work, we used the planner to design the layout of new robot workcells by running a black box optimizer on top of the planner to find the best robot placement for each set of tasks. The cost of robots and floor space is one of the primary limiting factors in many manufacturing setups, and our results show that we are able to substantially reduce the execution time of each of our evaluation configurations by performing layout optimization. With our RL setup, the reward function can easily be modified to incorporate different optimization objectives with different levels of importance. As an example, for our demonstrations on real robots, we trained the model with an additional reward term for returning to the start position at the end of the episode using the Euclidean (L2) norm in configuration space from the start configuration, only activated once all tasks are solved. We also used a weak penalty term based on squared joint space acceleration to discourage superfluous movements and unnecessarily extreme accelerations. Robotic applications often have unique requirements, and with an RL formulation, we only need to design a cost function to reflect them, which is often much easier than designing behaviors, making RL-based systems more easily adaptable to different requirements compared with both hand-coded systems and supervised learning–based systems learning from either an algorithmic planner or human demonstrations.

As another example use case, in situations where assembly line halts due to robot failures are not acceptable, the only current strategy in the industry is to create fail-safe motion plans with each robot removed, which is extremely costly in the manual labor required to design these plans and not scalable to more than one robot failure per cell. With our approach, the trained model can either generate the backup trajectories ahead of time, or as the failures happen, with minimal downtime. In case the remaining robots cannot continue to cover all of the tasks, but at least some of the robots can be moved (e.g., along linear rails), the model can also be used, with layout optimization, to quickly determine the optimal new placements for the remaining robots.

In terms of real-world execution of the generated trajectories, although the models already respect joint velocity and acceleration constraints, it would be prudent to perform postprocessing on the generated trajectory for smoothing and for compliance with more complex constraints such as jerk and torque. Movie 2 shows trajectories generated by the model postprocessed by fifth-order spline interpolation to 1 kHz running on Franka Panda robots. In this case, the network has also been trained to respect a 1.0-cm collision margin for additional safety.

High-density robotic task and motion planning for reaching is a problem that expert humans can manually solve at great time investment and manpower costs, using human experience and intuition to navigate the huge combinatorial search space. However, this problem has proven to be very challenging for automated algorithmic solutions. Machine learning provides methods to imitate human intuition that is hard to express as hand-coded heuristics. However, a naive machine learning–based solution would suffer the same fate of combinatorially increasing cost if the model architecture does not take into account the relational inductive biases that can be exploited to reduce the required learning efforts and improve generalization. Our combination of a structured state representation and a model learning approach that exploits the structure mirrors how expert humans solve this problem using their intuitive understanding of the construction of the problem.

The physical world is inherently structured, and one of the most intuitive computational representations of those structures is as graphs. In our case, we chose to represent the scene as a graph made out of robot, task, and obstacle nodes, exploiting the fact that each type of interaction (e.g., driving a robot to a task or avoiding an obstacle) only needs to be learned once, regardless of the complexity of the scene. This gives us better data efficiency and fewer parameters to learn compared with naive network architectures that do not exploit these inductive biases. The fact that we can learn from every interaction of the same type (e.g., robot to task) to train one fixed-size function turns the challenge of combinatorial complexity into an advantage and gives us the ability to efficiently train small and high-quality models that do not overfit to the training data.

Limitations and future work

In practical applications, the key challenge we identified in task and motion planning for dense robotics is to achieve scalability, and it is the main factor preventing existing solutions from being applicable to real-world applications. Our approach is demonstrated in a general reaching setting and a formulation where any robot can complete any task in any order within a shared workspace. To specifically focus on the scalability challenge, the configuration in this paper represents the largest search space, allowing us to tackle the fundamental scalability issues before introducing additional constraints. In practical/industrial applications, there are often constraints, such as intertask dependencies and heterogeneous robot capabilities, that are not considered in this work. Our approach could be extended to handle these constraints, for example, by disabling edges between incompatible robots and tasks or adding edges between tasks to signal dependencies. The resulting problem is theoretically simpler to solve than the fully unconstrained problem, but this is to be demonstrated in future work. Similarly, on the motion

planning level, we chose the abstract setting where the tasks are fully constrained end-effector poses, which makes motion planning the most constrained and difficult. In many settings such as spot welding or painting, the task poses are only semiconstrained, e.g., some rotational constraints of the end effector in spot welding are soft constraints. We believe that this can be accomplished by augmentation of the reward function, and that is also left to be demonstrated in future work.

The favorable scaling properties demonstrated in this paper are sufficient to enable many applications and industrial use cases, but if scalability to even larger settings is required, it would be computationally more efficient to apply some heuristics to prune graph connectivity. For example, a preprocessing step can determine reachability between all robot and task pairs, and edges are only added where a robot can reach a task. This takes $O(N_{robot}N_{task})$ time and would not increase the time complexity of the overall algorithm. Similarly, where a robot's work volume does not overlap with an obstacle, the robot can never collide with the obstacle, and the pair would not need a connecting edge. We did not implement these simple optimizations because we wanted to explore the limit of machine learning and keep the algorithm as simple and generally applicable as possible, but in a real-world application, it would be prudent to include them. A graph attention mechanism (*18*) may also improve training in very large cells to automatically reduce the relevant pairs of entities the network has to consider.

For simplicity, we chose a minimal representation of obstacles—cuboids. Although this representation works well for our evaluation environments, it may not be suitable for environments where most obstacles cannot be decomposed into a reasonable number of cuboid shapes. In that case, and especially if the robots must get very close to the obstacles (e.g., for spot welding), it may be beneficial to add more primitive types and pick the best primitive type for each decomposed obstacle component. This approach would require an adjustment in training to randomly place obstacles of all primitive types. Another possibility is to support arbitrary polyhedrons by representing them as graphs themselves and using another GNN to summarize their properties into fixed-length embeddings to be used in the main neural networks. The embedding and policy/value networks can be trained end-to-end to ensure that the embeddings capture useful information for motion planning.

To illustrate the core capabilities of our approach, we focus on reaching tasks. However, the algorithm can be readily extended to more complex operations, such as pick-and-place tasks. One possible formulation is to factor pick-and-place tasks into separate picking and placing task poses, adding and removing task dependencies and compatibility as required using edge features in the state graph, and have the simulator attach and detach objects from end effectors where necessary. In this formulation, a separate controller or a grasp pose predictor is required to generate feasible pose targets for picking up arbitrary objects, both at training time and at inference time. Extension to manipulation tasks presents further challenges, including object collisions, robot-environment constraints, and occlusion handling in multirobot coordination, requiring future work on both algorithmic development and experimental validation.

Going beyond existing capabilities into potential future applications, although our method is more than fast enough at inference time for online planning, online planning comes with other challenges such as perception and robot trajectory tracking accuracy without postprocessing. Solving these issues, together with our fast planner, would enable industrial capabilities that are not currently available and would be a valuable extension to this work.

As an extension to our workcell layout optimization example, using RoboBallet combined with a black box optimizer as the outer loop, a designer can use it to quickly find the optimal number, types, and positioning of robots in a workcell. For example, they may need to make a decision between using a six-DoF robot mounted on a linear rail versus a seven-DoF robot or maybe a heterogeneous team of robots. Here, we present a simplified version where only the placements are optimized. If the choice of robots is also to be optimized, the model would be trained with robot morphology randomization at training time, in addition to the other axes of randomization, so that the resulting model would be compatible with multiple robot morphologies.

Last, bimanual manipulation is the most natural way for humans to interact with the environment, and it enables capabilities that are out of reach of single manipulators. One of the main challenges in multiarm manipulation is the combinatorial complexity in performing motion planning in such a high-dimensional action space. Our approach shows that this scalability can be achieved using a structured network architecture, which can provide more insights and pave the way for developing future technologies in bimanual and multimanual task settings, extending the nature of multiarm coordination from collision avoidance, which is the focus of our work, to cooperative task completion in many other domains of robot manipulation.

In this work, we present a method for multirobot task and motion planning (demonstrated on but not fundamentally restricted to reaching) that is able to perform automated planning on industrial-scale workcells, a task that is currently performed manually at great cost and great latency. We showed that a fast planner like ours not only allows for acceleration of this process but also makes previously unknown capabilities such as workcell optimization and online planning feasible.

MATERIALS AND METHODS

Our goal is to train a neural network–based controller that, given the states of all robots, tasks, and obstacle primitives, will generate a set of joint velocities for each robot such that, when followed over time, produces the most efficient trajectory to solve all of the tasks. To that end, we designed an RL formulation, where the observations are represented as graphs in which nodes represent robots, tasks, and obstacle primitives and edges represent the relationships between them (e.g., relative pose). The trained policy network is a GNN (*19*) that takes the observation graphs as input and produces commanded joint velocities for all robots at each time step. This in effect allows the agent to reason relationally and combinatorially while only taking in the raw state as inputs. The joint velocities are integrated to produce next robot configurations, except where the commanded velocities would cause a collision, in which case the joint velocities for the robot are zeroed. The agent is rewarded for solving tasks and avoiding collisions. We used a slightly modified asynchronous version of the TD3 (*20*) RL algorithm to train the policy network and two auxiliary discounted state-action return prediction Q networks, also known as critics. The overall training and evaluation setup is shown in Fig. 1.

RL setup

Environment

The environment is implemented using a kinematic simulator. Each action applied advances the simulation by 0.1 s, and each action consists of desired joint velocities for all joints of all robots. We apply acceleration and velocity limiting on the basis of robot kinematic limits. A task is considered solved if the end effector of a robot moves to within a Euclidean distance of 2.5 cm and an angular distance of 15° of the task pose. Although this may seem coarse, there is a trade-off between tolerance and learning speed, and in practice, 2.5 cm is close enough to generate good plans that can be postprocessed to arbitrary precision. When a task is solved, the solving robot is frozen for 0.5 s of dwell time to simulate performing some work at the site and to encourage even task distribution between robot arms. If the joint velocities generated for any of the robots cause it to go into collision with itself, another robot, or the environment, the joint velocities for the robot in question are zeroed, and thus the robot stands still for that time step. Therefore, the environment does not allow for collisions. This is a desirable property for transfer to real-world setups, where collisions must be avoided at all costs. We use a collision penalty to discourage the policy from generating joint velocities that would cause collisions (see the "Reward" section below for more details).

Observation

We represent environment states as graphs, with robots, tasks, and obstacle primitives as nodes. The primary benefit of using this structured representation rather than simply flattening the state information into a vector is that it provides a way to incorporate relational information between entities in a reusable way and allows the network to reason about them relationally. For example, the model only has to learn to reason about relative pose between a robot and a task once, and it automatically transfers via the graph to all combinations of robots and tasks. By contrast, conventional architectures would be forced to discover these relationships for every combination of entities, leading to exponential compute and memory scaling as well as poor data efficiency.

In GNNs, edges can be directional—we used bidirectional edges between robot nodes to allow for collision avoidance and coordination, whereas unidirectional edges from the task nodes and obstacle nodes to robot nodes give the necessary information to robot nodes for motion planning. The directionality of edges determines how information flows during the graph inference step, and our choice was designed to propagate information to the robot nodes to allow reading out of the actions for each robot.

Obstacle representation

Our goal was to train a generalized planner that can plan in environments it has not seen during training and support obstacles with arbitrary geometries. That requires breaking obstacles down to primitives, where each primitive can be added to the graph as a node. During training, we generated cuboid obstacles of random sizes, positions, and orientations to add to the graphs as obstacle nodes, filtered only to exclude collisions between them and the robots in their initial poses.

At evaluation time, with externally defined meshes as obstacles, the obstacle meshes are decomposed into approximate convex hulls, and the oriented bounding boxes of those convex hulls are used as obstacle nodes (see Fig. 6). The approximate convex decomposition is performed using the V-HACD library (*21*), whereas the oriented bounding box search uses our implementation of the algorithm described in (*22*), which consists of a combination of genetic and Nelder-Mead (*23*) algorithms to efficiently find the best rotation for minimum-volume bounding boxes through unconstrained optimization on the 3D rotation group $SO(3)$. Because each cuboid has four equivalent representations that can be created by exchanging axes and making the corresponding change in pose, we randomly selected between the representations at each time step as a simple augmentation.

Task sampling

At training time, the tasks (desired end-effector poses) were sampled uniformly in position from the surfaces of the obstacle cuboids and in orientation randomly such that the desired orientation corresponding to the end-effector direction was maximally 22.5° from the normal direction of the face (that is, to complete the task, the end effector pointed approximately into the obstacle). At evaluation time, the tasks were generated in the same way on the externally defined obstacle geometry. We checked that each task pose had at least one collision-free IK solution from one robot. For efficiency, we were not verifying that there was a collision-free path from the start position to the task pose, only that a collision-free IK solution existed. Therefore, some of the tasks may not have had a valid path solution. The IK solutions were not stored, and no information from this task-filtering process was used in the training process.

Node features

The environment observations are in the form of graphs, where robot and task nodes have different features, whereas obstacle nodes have no node features. The node features for different types of nodes are then concatenated in a block-diagonal fashion to create the combined node features for the scene graph $\mathbf{G}_n$.

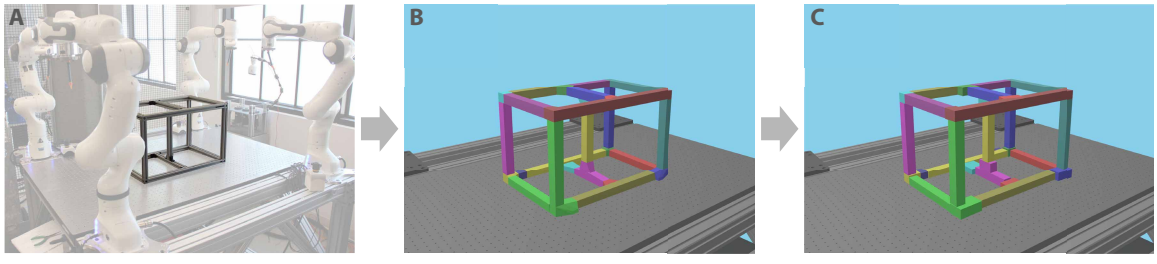


Fig. 6. Obstacle mesh decomposition pipeline. (**A**) Starting from the computer-aided design (CAD) model of the obstacle. (**B**) Result of decomposition into convex polyhedrons using V-HACD. (**C**) Oriented bounding boxes computed from the convex polyhedrons using (*22*).

For each robot, the robot node consists of the joint configuration for each of the seven DoFs of the robot, joint velocity for each DoF, and dwell time remaining, which is nonzero if the robot has reached a task pose and is executing the task (we froze the robot in the task pose for 0.5 s of dwell time to simulate executing some task at the site). Each task node consists of a single feature, task status, which records whether a task has been solved.

Edge features

Edge features encode relative information between each pair of nodes where an edge exists. Similarly to the node features, we have different edge features for each edge type, concatenated in a block-diagonal fashion to create the combined edge features for the scene graph $\mathbf{G}_e$. In a graph net, edges are directional and represent direction of information flow, and we used "sender" and "receiver" to denote the vertices of an edge relative to the direction of information flow. All poses were encoded using a 3D translation and a 6D representation of the rotation (*24*).

The robot → robot edges encode the sending robot base pose relative to the receiving robot tip pose. The task → robot edges similarly encode the task end-effector pose relative to the receiving robot tip pose. Last, the obstacle → robot edges have the following features: centroid translation of the obstacle primitive relative to the receiving robot tip pose and rotation matrix of the obstacle primitive relative to the receiving robot tip pose, with each basis vector scaled by the corresponding span of the primitive in that direction.

Global features

The global features $\mathbf{G}_n$ of the graph consist of the following: episode time normalized to timeout, the episode time at which we terminated the episode if the tasks had not been completed, and the current score as defined in the "Reward" section.

Reward

To calculate the reward for a step, we first calculated a score $S(\mathbf{s})$ for the state $\mathbf{s}$, which is simply the fraction of tasks N_{total} that have already been solved N_{done} (Eq. 1)

$$S(\mathbf{s}) = \frac{N_{\text{done}}(\mathbf{s})}{N_{\text{total}}(\mathbf{s})} \tag{1}$$

The reward R consists of two components. The first component R_{score} is the difference in score between the new state (state $\mathbf{s}$ with action $\mathbf{a}$ applied) and the previous state (Eq. 2)

$$R_{\text{score}}(\mathbf{s}, \mathbf{a}) = S(\mathbf{s}+\mathbf{a}) - S(\mathbf{s}) \tag{2}$$

The second component $R_{\text{collision}}$ is a penalty for actions that would have put the robots in collision (if it was not stopped by the environment). This is the number of robots that would be put in collision $N_{\text{robots\_colliding}}$, scaled by a coefficient C_{col} (Eq. 3). That said, in theory, we do not need to add an additional penalty for collisions because the increase in execution time, combined with the discounting, would encourage the agent not to command joint velocities that would cause collisions, which are not optimal. However, in practice, adding an explicit penalty for collisions is effective to both reduce collisions and improve solution execution time

$$R_{\text{collision}}(\mathbf{s}, \mathbf{a}) = C_{\text{col}} N_{\text{robots\_colliding}}(\mathbf{s}, \mathbf{a}) \tag{3}$$

The total reward is the sum of the two components (Eq. 4)

$$R(\mathbf{s}, \mathbf{a}) = R_{\text{score}}(\mathbf{s}, \mathbf{a}) + R_{\text{collision}}(\mathbf{s}, \mathbf{a}) \tag{4}$$

Hindsight experience replay

Given that the reward as specified above is sparse, it is unlikely that an RL agent early on during the training process would be able to find good rewards through random exploration. The conventional solution to this problem is to engineer a shaped reward by giving partial credit when the agent moves robots closer to completing tasks. However, designing a good shaped reward that encourages fast and reliable learning is not trivial and is also specific to the type of task being performed. Hindsight experience replay (HER), as proposed by Andrychowicz *et al.* (*25*), provides another solution to the sparse reward problem. With HER, instead of providing partial credits to aid learning when the agent is not yet reaching tasks, with each failed episode, we also generated a corresponding imaginary episode where we moved a randomly selected subset of tasks to randomly selected points of the robot trajectories within the episode, therefore providing positive training episodes that, in essence, show the learner that it would have been a successful episode if the tasks were actually along the robot trajectories the agent generated. This provides rich training signals with a healthy mix of successes and failures for efficient learning. Refer to Andrychowicz *et al.* (*25*) for more details on the general concept of HER. In our experiments, we found that HER performs approximately as well as even a well-tuned shaped reward, so we relied entirely on HER for the experiments above for simplicity.

Modified TD3

Twin-delayed deep deterministic policy gradient (TD3) (*20*) is an RL algorithm based on the deterministic policy gradient algorithm (*26*) in an actor-critic framework, where two networks are trained simultaneously—a policy network $\pi(\mathbf{s})$ that predicts the best action to take from a state and a critic network $Q(\mathbf{s},\mathbf{a})$ that predicts the discounted total future return (Q value) given a state and an action taken in that state. The policy network is trained to produce actions that result in a high Q value

$$\pi(\mathbf{s}) \Rightarrow \text{argmax}_a Q(\mathbf{s}, \mathbf{a}) \tag{5}$$

and the critic network is trained to predict the sum of the immediate reward from applying action $\mathbf{a}$ in state $\mathbf{s}$ and the discounted Q value from the resulting state $\mathbf{s}'$ with the best action in that state. Because we do not know the best action in state $\mathbf{s}'$, we use the output of the policy network $\pi(\mathbf{s})$ as an approximation

$$Q(\mathbf{s}, \mathbf{a}) \Rightarrow r(\mathbf{s}, \mathbf{a}) + \lambda Q\left(\mathbf{s}', \pi\left(\mathbf{s}'\right)\right) \tag{6}$$

where $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ is a transition from applying action $\mathbf{a}$ to state $\mathbf{s}$, arriving at state $\mathbf{s}'$ with reward r, and λ is the discount factor.

Deterministic policy gradient algorithms are a newer class of RL algorithms where the policy network predicts an action directly as opposed to a distribution over actions that would be predicted by policy networks in earlier stochastic policy algorithms. They are more efficient to train but are prone to get stuck in local minima. TD3 is the current state of the art of a lineage of deterministic policy gradient algorithms that each added some tweaks to stabilize training. We refer the reader to Fujimoto *et al.* (*20*) for a thorough description of the algorithm.

In our implementation of TD3, we removed the delayed target network updates for simplicity because, similarly to what the authors reported, we found the stability contribution of the delayed updates to be minimal, and the benefit is mainly a computational one. However, because our network architecture is much more

complex, the target network updates do not constitute a notable part of the compute requirement.

We also decoupled critic and policy Polyak averaging (*27*) constants so they can be tuned independently. On the actor side, we introduced exponential decay in exploration variance to encourage strong exploration in the beginning and more accurate task solutions later on in the training process. Similarly, we introduced two exponential decays—λ_{η_π} and λ_{η_Q} for the policy and Q function learning rates, respectively.

Network architecture

The system consists of three learned functions—two Q value estimators and a policy function. All three functions use a GNN at their core, with the same architecture, but initialized and trained independently without weight sharing.

GNN core

The core GNN takes as input graph G with nodes N with features matrix $\mathbf{G}_n$, edges E with features matrix $\mathbf{G}_e$, and graph-global feature vector $\mathbf{G}_g$ and produces an output graph $\mathbf{G}'$ similarly with node, edge, and global features $\mathbf{G}'_n$, $\mathbf{G}'_e$, and $\mathbf{G}'_g$. The GNN contains three learned update functions—a node update function U_n, an edge update function U_e, and a global update function U_g. In all of the following equations, $\bigoplus$ denotes concatenation.

The GNN first gathers, for each edge $e \in E$, the sending and receiving node features $\mathbf{G}_n[\mathbf{n}_s]$ and $\mathbf{G}_n[\mathbf{n}_r]$ (where $\mathbf{n}_s$ is the sending node and $\mathbf{n}_r$ is the receiving node), edge features $\mathbf{G}_e[\mathbf{n}_s \rightarrow \mathbf{n}_r]$, and global features $\mathbf{G}_g$ and concatenates them for input into the edge update function U_e, which generates new edge features $\mathbf{G}'_e$ for the output graph (Eq. 7)

$$\mathbf{G}'_e = U_e\left(\mathbf{G}_n[\mathbf{n}_s] \oplus \mathbf{G}_n[\mathbf{n}_r] \oplus \mathbf{G}_e[\mathbf{n}_s \rightarrow \mathbf{n}_r] \oplus \mathbf{G}_g\right) \forall e(\mathbf{n}_s \rightarrow \mathbf{n}_r) \in E \tag{7}$$

Then, for each node $n \in N$, all of the edge output features for edges coming into the node are element-wise aggregated by summation to generate an aggregated view of information arriving into the node. This is concatenated with the node input features and the global input features and passed as input into the node update function U_n, which generates node output features $\mathbf{G}'_n$ (Eq. 8)

$$\mathbf{G}'_n = U_n\left(\left(\sum_{e \in E(^* \rightarrow n)} \mathbf{G}'_e[e]\right) \oplus \mathbf{G}_n[n] \oplus \mathbf{G}_g\right) \forall n \in N \tag{8}$$

Last, the global update function U_g takes as input the aggregated node embeddings and edge embeddings, as well as the input global features $\mathbf{G}_g$, and generates new global embeddings $\mathbf{G}'_g$ (Eq. 9)

$$\mathbf{G}'_g = U_g\left(\sum_{n \in N} \mathbf{G}'_n[n] \oplus \sum_{e \in E} \mathbf{G}'_e[e] \oplus \mathbf{G}_g\right) \tag{9}$$

In the discussion below, we use $\mathrm{Core}(\mathbf{G}_n, \mathbf{G}_e, \mathbf{G}_g)$ to denote the operations in Eqs. 7 to 9 above for brevity (Eq. 10)

$$\mathbf{G}'_n, \mathbf{G}'_e, \mathbf{G}'_g = \mathrm{Core}(\mathbf{G}_n, \mathbf{G}_e, \mathbf{G}_g) \tag{10}$$

For a more thorough introduction to GNNs, please refer to Battaglia *et al*. (*19*).

Critic networks (Q value estimators)

The two state-value estimation networks Q have identical architecture but independent weight initialization. Given a state $\mathbf{s}$ with features encoded in graph tuple $(\mathbf{G}_n, \mathbf{G}_e, \mathbf{G}_g)$ and an action $\mathbf{a}$, each Q network first concatenates the action (commanded joint velocities) with the corresponding robot node features, and the new node features are embedded into a higher-dimensional embedding space using the node embedding function E_n to produce the node embedding inputs to the core GNN $\mathbf{L}_n$. The edge features and global features are also embedded into a higher-dimensional embedding space using the learned edge embedding function E_e and global embedding function E_g, into embeddings $\mathbf{L}_e$ and $\mathbf{L}_g$, respectively (Eq. 11)

$$\mathbf{L}_n, \mathbf{L}_e, \mathbf{L}_g = E_n(\mathbf{G}_n \oplus \mathbf{a}), E_e(\mathbf{G}_e), E_g(\mathbf{G}_g) \tag{11}$$

Then, the resulting graph embedding made up of the feature embeddings is passed through the core GNN for processing, and the output Q value is predicted from the global features of the output graph with a scalar prediction function P_Q (Eq. 12)

$$Q(\mathbf{s}, \mathbf{a}) = P_Q\left(\mathrm{Core}(\mathbf{L}_n, \mathbf{L}_e, \mathbf{L}_g)_g\right) \tag{12}$$

Policy network

The policy network uses a similar embedding process as the Q networks but without the action concatenation (Eq. 13)

$$\mathbf{L}_n, \mathbf{L}_e, \mathbf{L}_g = E_n(\mathbf{G}_n), E_e(\mathbf{G}_e), E_g(\mathbf{G}_g) \tag{13}$$

After passing through the core GNN, the output actions are predicted from the node features corresponding to the robot nodes N_{robots}, using another prediction function P_π (Eq. 14)

$$\pi(\mathbf{s}) = P_\pi\left(\mathrm{Core}(\mathbf{L}_n, \mathbf{L}_e, \mathbf{L}_g)_n[\mathbf{n}_r]\right) \forall \mathbf{n}_r \in N_{\text{robots}} \tag{14}$$

All of the learned functions (three embedding functions and three update functions) have the same architecture. Each is a fully connected network where each layer consists of a matrix multiplication followed by a layer normalization operation (*28*) and Gaussian error linear unit (GELU) activation (*29*).

Both the value (P_Q) and policy (P_π) prediction networks are similar in architecture to the main learned update functions but much smaller in size. The experiments were conducted with JAX (*30*), with models built in the Jraph library (*31*) and the Flax library (*32*). Refer to table S1 for the model shapes.

Training

During training, 64 actors were used to continuously and asynchronously generate training episodes for insertion into a first-in–first-out replay buffer implemented using the Reverb library (*33*), updating their local copies of the policy network from the learner after each episode. The learner samples uniformly from the replay buffer. The replay buffer enforces a fixed sample-to-insert ratio by blocking insertion or sampling as necessary. The learner uses a single NVIDIA A100 GPU, and each actor uses an Intel Ice Lake Xeon CPU core. The training process takes 6 to 24 hours depending on problem size to achieve a 95% task solution rate, and the quality of trajectories continues to improve (solving more tasks, reducing execution time, and reducing number of collisions) for an additional 2 to 6 days. Please refer to table S1 for the main hyperparameters used in the experiments. These hyperparameters were tuned with the Google Vizier (*16*) black box optimizer.

Supplementary Materials

The PDF file includes:
Table S1

Other Supplementary Material for this manuscript includes the following:
Movies S1 to S4

REFERENCES AND NOTES

1. S. Pellegrinelli, N. Pedrocchi, L. M. Tosatti, A. Fischer, T. Tolio, Multi-robot spot-welding cells for car-body assembly: Design and motion planning. *Robot. Comput. Integr. Manuf.* **44**, 97–116 (2017).
2. Y. Yamada, S. Nagamatsu, Y. Sato, "Development of multi-arm robots for automobile assembly" in *Proceedings of 1995 IEEE International Conference on Robotics and Automation* (IEEE, 1995), vol. 3, pp. 2224–2229.
3. H. Chen, T. Fuhlbrigge, X. Li, "Automated industrial robot path planning for spray painting process: A review" in *2008 IEEE International Conference on Automation Science and Engineering* (IEEE, 2008), pp. 522–527.
4. V. N. Hartmann, A. Orthey, D. Driess, O. S. Oguz, M. Toussaint, Long-horizon multi-robot rearrangement planning for construction assembly. *IEEE Trans. Robot.* **39**, 239 (2022).
5. Z. Xian, P. Lertkultanon, Q.-C. Pham, Closed-chain manipulation of large objects by multi-arm robotic systems. *IEEE Robot. Autom. Lett.* **2**, 1832–1839 (2017).
6. S. LaValle, Rapidly-exploring random trees: A new tool for path planning (Tech. Rep. 98-11, Iowa State University, 1998).
7. S. M. LaValle, J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects" in *Algorithmic and Computational Robotics* (Taylor-Francis, 2001), pp. 303–307.
8. S. Karaman, E. Frazzoli, Sampling-based algorithms for optimal motion planning. *Int. J. Robot. Res.* **30**, 846–894 (2011).
9. J. Canny, *The Complexity of Robot Motion Planning* (MIT Press, 1988).
10. W. Vega-Brown, N. Roy, "Task and motion planning is pspace-complete" in *Proceedings of the AAAI Conference on Artificial Intelligence* (AAAI, 2020), vol. 34, pp. 10385–10392.
11. G. B. Mathews, On the partition of numbers. *Proc. Lond. Math. Soc.* **s1-28**, 486–490 (1896).
12. H. Ha, J. Xu, S. Song, Learning a decentralized multi-arm motion planner. arXiv:2011.02608 [cs.RO] (2020).
13. T. Pan, A. M. Wells, R. Shome, L. E. Kavraki, "A general task and motion planning framework for multiple manipulators" *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (IEEE, 2021), pp. 3168–3174.
14. Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, L. D. Jackel, "Handwritten digit recognition with a back-propagation network" in *Advances in Neural Information Processing Systems* (MIT Press, 1989), pp. 396–404.
15. J. J. Kuffner, S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning" in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)* (IEEE, 2000), vol. 2, pp. 995–1001.
16. D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, D. Sculley, "Google Vizier: A service for black-box optimization" in *KDD '17: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery, 2017), pp. 1487–1495.
17. X. Song, Q. Zhang, C. Lee, E. Fertig, T.-K. Huang, L. Belenki, G. Kochanski, S. Ariafar, S. Vasudevan, S. Perel, D. Golovin, The Vizier gaussian process bandit algorithm. arXiv:2408.11527 [cs.LG] (2024).
18. P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph attention networks. *stat* **1050**, 10 (2017).
19. P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, R. Pascanu, Relational inductive biases, deep learning, and graph networks. arXiv:1806.01261 [cs.LG] (2018).
20. S. Fujimoto, H. Hoof, D. Meger, "Addressing function approximation error in actor-critic methods" in *Proceedings of the 35th International Conference on Machine Learning* (PMLR, 2018), pp. 1587–1596.
21. J. W. Ratcliff, K. Mamou, Voxelized hierarchical convex decomposition - V-HACD, version 4, GitHub (2023); https://github.com/kmammou/v-hacd.
22. C.-T. Chang, B. Gorissen, S. Melchior, Fast oriented bounding box optimization on the rotation group *SO*(3, ℝ). *ACM Trans. Graph.* **30**, 1–16 (2011).
23. D. M. Olsson, L. S. Nelson, The Nelder-Mead simplex procedure for function minimization. *Technometrics* **17**, 45–51 (1975).
24. Y. Zhou, C. Barnes, J. Lu, J. Yang, H. Li, "On the continuity of rotation representations in neural networks" in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (IEEE, 2019), pp. 5745–5753.
25. M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, W. Zaremba, "Hindsight experience replay" in *Advances in Neural Information Processing Systems* (Curran Associates, 2017), pp. 5048–5058.
26. D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller, "Deterministic policy gradient algorithms" in *Proceedings of the 31st International Conference on Machine Learning* (MLResearchPress, 2014), pp. 387–395.
27. B. T. Polyak, A new method of stochastic approximation type. *Avtom. Telemek.* **7**, 98–107 (1990).
28. J. L. Ba, J. R. Kiros, G. E. Hinton, Layer normalization. arXiv:1607.06450 [stat.ML] (2016).
29. D. Hendrycks, K. Gimpel, Gaussian error linear units (GELUs). arXiv:1606.08415 [cs.LG] (2016).
30. J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: Composable transformations of Python+NumPy programs, version 0.3.13, GitHub (2018); https://github.com/jax-ml/jax.
31. J. Godwin, T. Keck, P. Battaglia, V. Bapst, T. Kipf, Y. Li, K. Stachenfeld, P. Veličković, A. Sanchez-Gonzalez, Jraph: A library for graph neural networks in jax, version 0.0.1.dev, GitHub (2020); http://github.com/google-deepmind/jraph.
32. J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, M. van Zee, Flax: A neural network library and ecosystem for JAX, version 0.11.1, GitHub (2023); http://github.com/google/flax.
33. A. Cassirer, G. Barth-Maron, E. Brevdo, S. Ramos, T. Boyd, T. Sottiaux, M. Kroiss, Reverb: A framework for experience replay. arXiv:2102.04736 [cs.LG] (2021).
34. E. Todorov, T. Erez, Y. Tassa, "Mujoco: A physics engine for model-based control" in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2012), pp. 5026–5033.

Acknowledgments
Funding: This work was funded by Google DeepMind and Intrinsic. **Author contributions:** M.L. and K.A. conceptualized, designed, and conducted the experiments. M.L., K.A., and J.S. developed the core algorithmic framework. M.L. and K.G. developed and implemented the experimental infrastructure. M.L., K.A., K.G., J.S., and S.S. proposed improvements to the system. M.L., K.A., S.S., and J.S. prepared and authored the paper. J.S., K.A., and Z.L. provided research guidance and planning. K.A., J.S., S.S., and Z.L. edited and revised the paper. T.K. managed the project and provided organizational support and funding. **Competing interests:** Google DeepMind has submitted US and WO patent applications for the techniques described in this paper. All authors affiliated with Google DeepMind and Intrinsic own Alphabet shares as part of their employment. All authors declare that they have no other competing interests. **Data and materials availability:** All data needed to support the conclusions of this manuscript are included in the main text or Supplementary Materials. The open sourced version of the system described in this paper, which is implemented with the MuJoCo physics simulator (*34*), is available at https://doi.org/10.5281/zenodo.16546403.

Submitted 4 November 2024
Accepted 5 August 2025
Published 3 September 2025
10.1126/scirobotics.ads1204