CONTROL WHAT YOU CAN: INTRINSICALLY MOTIVATED REINFORCEMENT LEARNER WITH TASK PLANNING STRUCTURE

Anonymous authors
Submission under double-blind review

ABSTRACT

We present a hierarchical reinforcement learning agent that is intrinsically motivated to learn how to control its observation space in the fastest possible manner by optimizing learning progress. Our agent learns what can be controlled, how to allocate time and attention, and the relations between objects in the environment. We show the effectiveness in multi-stage object manipulation tasks. In a nutshell, our work combines several task-level planning ideas (e.g., backtracking search on task graph, probabilistic road-maps, allocation of search efforts, etc.), in the form of structured prior, with deep RL control and relational reasoning to learn from scratch.

1 INTRODUCTION

This paper studies *how to make an autonomous agent learn to gain maximal control of its environment* under little external reward. To answer this question, we turn to the young experts among us and ask: what would children do? They play, often with any objects within their reach. The purpose may not be immediately clear to us. But to play is to manipulate, to *gain control*. In the same spirit, we specifically design a Reinforcement learning (RL) agent that is 1) intrinsically motivated by gaining control of the environment 2) capable of learning its own curriculum and reasoning about object relations.

As a motivational example, consider an environment with a heavy object that cannot be moved without using a tool such as a forklift, as depicted in Fig. 1(a,b). The agent needs to be able to control itself and the tool, and use it to move the heavy object. In the beginning, we do not assume the agent has knowledge of the tool, object, or physics. It needs to learn *from scratch*.

Without external rewards, an agent may be driven by intrinsic motivation (IM) to gain control over its own internal representation of the world, which includes itself and objects in the environment. It often faces a decision of what to attempt to learn with limited time and attention: if there are several objects that can be manipulated, which one should be dealt with first? In our approach the scheduling is solved by an automatic curriculum that aims at improving *learning progress*. The learning progress, as detailed in Sec. 2, may have its unique advantage over other quantities such as prediction error (curiousity): it renders unsolvable tasks uninteresting as soon as progress stalls.

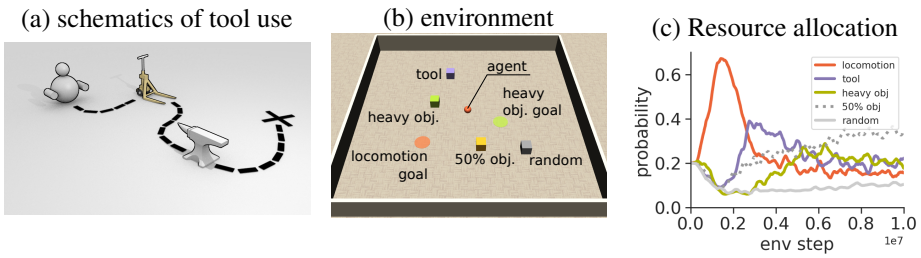

Figure 1: Basic object manipulation environment illustrated in (a) with a screenshot in (b). During learning it allocates resources wisely (c).

Table 1: A list of recent methods in intrinsically motivated RL and their differences.

	Intrinsic motivation	Computational methods
CWYC Ours	learning progress + prediction error	HRL, SAC, relational attention
h-DQN Kulkarni et al. (2016)	reaching subgoals	HRL, DQN
IMGEP Forestier et al. (2017)	learning progress	memory-based
CURIOUS Colas et al. (2018)	learning progress	DDPG, HER, E-UVFA
SAC-X Riedmiller et al. (2018)	auxiliary task	HRL, (DDPG-like) PI
Relational RL Zambaldi et al. (2018)	-	relation net, IMPALA
ICM Pathak et al. (2017)	prediction error	A3C, ICM
Goal GAN Florensa et al. (2018)	adversarial goal	GAN, TRPO
Asymmetric self-play Sukhbaatar et al. (2017)	self-play	Alice/Bob, TRPO, REINFORCE

Fig. 1(d) demonstrates that our agent, motivated by its learning progress, is able to allocate resources wisely to gain maximal control over the environment.

Inspired by the exciting progress of task-level planning from the robotics and AI planning communities, we model the RL agent using *temporal abstraction* in the form of chained subtasks. In practice, this is modeled as a task graph as in Fig. 1(c). In order to perform manipulation, the agent and the tool need to be in a specific relation. Our agent learns relationships by an attention mechanism which generalizes to new environments. Instead of following an end-to-end approach, we provide a core *reasoning structure* about tasks and subgoals.

Our main contributions are:

1. We propose to combine several task-level planning ideas (e.g., backtracking search on task graph/goal regression, probabilistic road-maps, allocation of search efforts, etc.), in the form of structured prior, with deep RL control to achieve task completion and skill acquisition from scratch.
2. We demonstrate maximizing controlablity and learning progress are an effective form of intrinsic motivation for RL.

Due to the large volume of recent intrinsically motivated RL studies, we list a few approaches in Table 1 to help readers clearly understand the relation of our work and the existing methods.

2 METHOD

This study proposes to use gaining controllability of the agent's environment as the driving force for an autonomous agent. We call the method *Control What You Can* (CWYC). The goal is to make an agent learn to control itself and objects in its environment, or more generically, to control the components of its internal representation. This idea originates from the controllability/reachability concept in control theory Kalman et al. (1960). We consider the control of each part of the representation space (e. g. objects) as one task. During the learning/development phase, the agent can decide which task/object to attempt to control. This requires the learning algorithm to spend available resources wisely on tasks where the agent can make progress and inferring potential dependencies between tasks. In order to express the capability of controlling a certain object, we consider goal-reaching tasks with randomly selected goals. If the agent can successfully bring the object to arbitrary goal positions, then it has achieved control of the object.

Our approach contains several components as illustrated in Fig. 2, namely **tasks** [1], **intrinsic motivation** [2], **task selector** [3], **task dependencies** [4],[5], **subgoal generator** [6], **task policies** [7], and **history buffers and forward models** [8]. The highlevel interplay of the components is described in the figure caption of Fig. 2.

All components are trained concurrently and without supervision. Prior knowledge enters only in the form of specifying the goal spaces (groups of coordinates of the state space). The environment allows the agent to select which task to do next and generates a random arrangement with a random goal.

Further details abaout all the introduced componanent can be found in Suppl. A

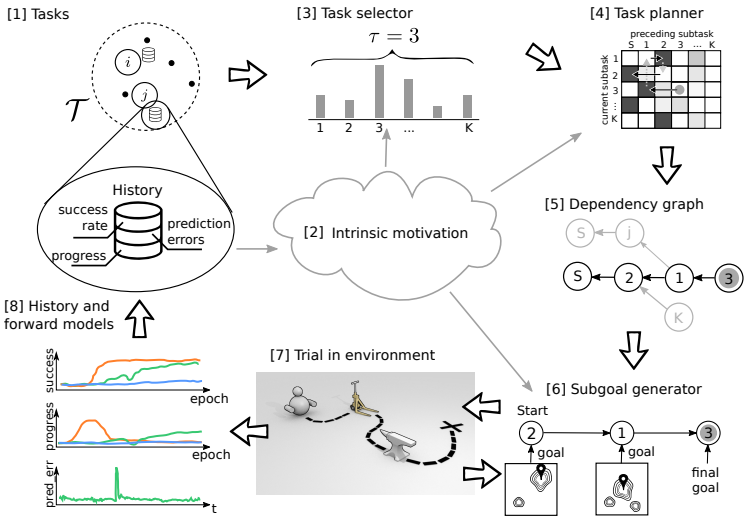

Figure 2: Overview of CWYC method. [1] For each task of all tasks $\mathcal{T}$, the agent stores a history to keep track of training progress. [2] An intrinsic motivation module computes the rewards and target signals for 3,4, and 6 based on learning progress and prediction errors. [3] A task selector (bandit) is used to select a self-imposed task (final task) maximizing expected learning progress. [4] Given a final task, the task planner computes a viable subtask sequence (bold) from a learned task graph [5]. [6] The subgoal generators (attention networks) create goals in each subtask. [7] After one rollout different quantities measuring the training progress are computed and stored in the per task history buffer [8].

3 Experimental Results

In this section, we present the experiment in a continuous environment for controlling objects. We want to demonstrate that the agent with CWYC learns efficiently to gain control over the environment.

In all our experiments we use our CWYC agent and the following baselines: In the *SAC* baseline the low-level controllers for each individual task (controlling the coordinates) try to solve them independently and spend resources on all tasks with equal probability. We also add an upper baseline which is our algorithm, but with handcrafted task planner (B) and subgoal generator (G) denoted as *crafted*, see Suppl. F. The task selector Π [3] is, however, learned to spend resources wisely.

3.1 Basic object manipulation

The basic object manipulation environment has continuous state and action spaces. It is implemented in the MuJoCo physics simulator (Todorov et al., 2012). The agent is a modeled as a physical object with two degrees of freedom which can move in the xy-plane. The environment has walls and several objects. The observation vector for d objects is structured as follows $(x, y, o^1_x, o^1_y, \ldots, o^d_x, o^d_y, \dot{x}, \dot{y}, p^1, \ldots, p^d)$, where (x, y) is the position of the agent, (o^i_x, o^i_y) is the position of the i-th object and p^i indicates whether the agent is in possession of the i-th object. The goal spaces are the coordinates of the agent (x, y) and the coordinates of each object (o^i_x, o^i_y).

In what follows, we consider the *Tool-Use* case with 4 objects: 1. the *tool*, that can be picked up easily; 2. the *heavy object* that needs the tool to be moved; 3. an unreliable object denoted as *50% object*, that does not respond to control during 50% of the rollouts; 4. a *random object* that moves around randomly and cannot be manipulated by the agent, see Fig. 1(b). The detail of the physics in this environment can be found in Suppl. C.

Figure 3 presents the training progress for our algorithm and the two baselines, *SAC* and *crafted*. Each task accounts for 20% of the total overall competence. In this setting an average maximum of 70% can be achieved, due to the "random object" and "50% object". The *SAC* baseline attempts to solve each task independently and spends resources equally between tasks. It only succeeds in the locomotion task. It cannot learn to pick up any of the other objects. As a remark, the arena is

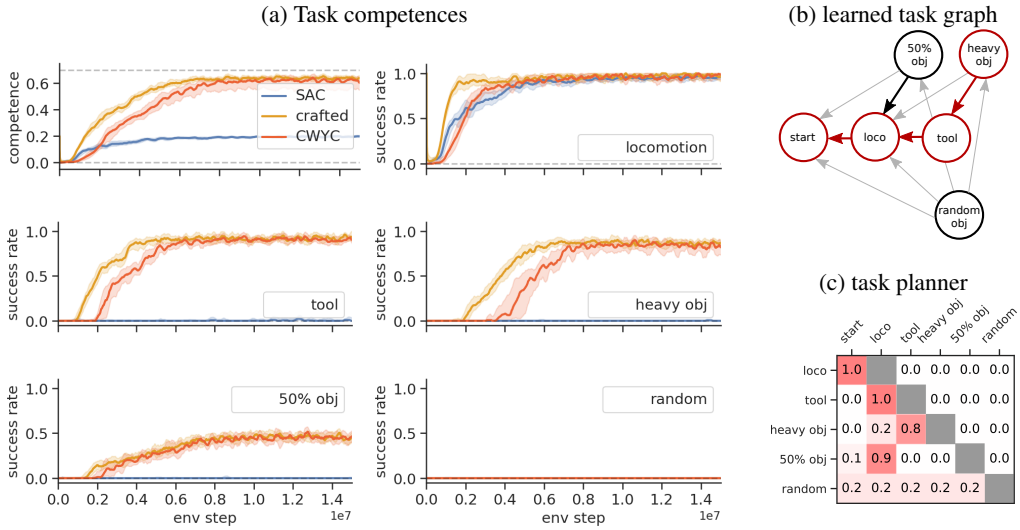

Figure 3: (a) Task competence of the agents in solving all 5 tasks in the tool-use setting. Overall performance (top left) can be maximal 70%. Individual tasks competences follow in the remaining panels. Learned task graph (a) and its matrix representation (b) in the tool-use setting. The arrows point to the preceding task, which corresponds to the planning direction. Line strength corresponds to the probability of using this transition. The probabilities of selecting task i (row) before task j (column) is shown in (b). Self-loops (gray) are not permitted. Every sub-task sequence begins in the *start* state.

relatively large such that random encounters are not very likely. The results show that our method is able to quickly gain control over the environment. After 10^7 steps, the agent can control what is controllable.

Analysis of gaining control. How does the agent gain control of the environment? Let us inspect the overall resource allocation as shown in Fig. 1(c). Starting from all tasks are uniformly selected, the agent quickly spends most time on the locomotion, because it can be directly controlled and is the easiest among all tasks. After locomotion can be solved well enough, to be able to make progress on the immediately movable objects, the agent starts to concentrate on them (tool and the "50% object"). Afterwards, the heavy object becomes controllable due to the competence in the tool task (at about $3 \cdot 10^6$ steps) and gets a higher share. The task selector produces the expected result that simple tasks are solved first and stop getting attention as soon as they cannot be improved more than other tasks. This is in contrast to approaches that are solely based on curiosity/prediction error. As a remark, the learned resource allocation of the *crafted* agent is similar to that of CWYC.

Understanding the structure. The resource allocation alone is not the solution to the problem of gaining control. The deep RL agent (*SAC*) cannot solve the more complicated tasks. One reason is that informative states are visited rarely. Another reason is that relations between objects are important but this structure is difficult to pick up directly.

The dependencies between tasks is learned by the task planner B, see Fig. 2[4]. Initially the dependencies between the subtasks are unknown such that $B(i, j) = 0$ resulting in a $1/\kappa$ probability of selecting a certain preceding subtask. After learning, the CWYC agent has found which task needs to be executed before which other one, as displayed in Fig. 3(c). The underlying policy of selecting task j before task i is shown in Fig. 3(b). The agent has found that locomotion can be done directly (probability ≈ 1). It also deduced from the experience, that the tool needs the locomotion task to be moved (second row) and that the heavy object needs the tool (third row).

Knowing that locomotion is needed to move the tool is very helpful, but by itself not sufficient. Where should the agent move? This is where the object relations learned by the subgoal generators (Fig. 2[6]) come in. The subgoal generators $G_{i,j}$ learn initially from surprising events and attempt to learn the relation among the components of the observation vector. For instance, every time

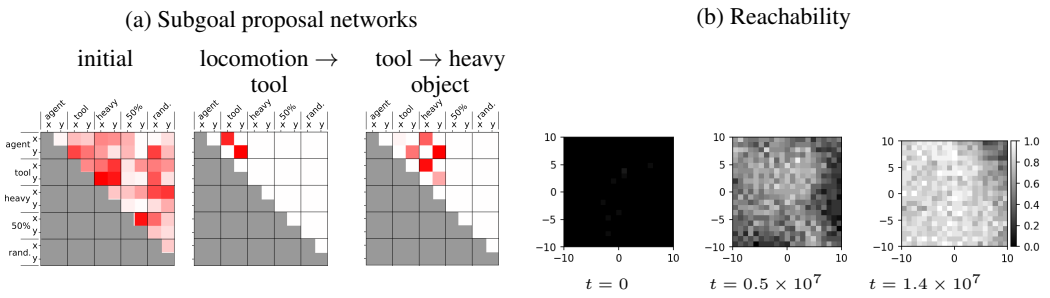

Figure 4: Subgoal proposal networks and the learned object relationships. (b) Reachability of the *heavy object*. Plotted is the success-rate in xy-plane of the arena. The color of each point reflects the probability of reaching it with the heavy box from 20 random starting states.

the tool is moved, the agent's location is close to that of the tool. Figure 4 displays the learned relationships for the subgoal generation of locomotion → tool transition and for tool → heavy object transition. For visualization purposes we present a reduced parametrization, namely $\min(|w^1|, |w^2|)$. A non-zero value indicates that the corresponding components are involved in the relationship. The full parametrization is visualized and explained in Suppl. E. However, in general G can also model different relationships, such that fixed offsets between objects or fixed locations.

At the beginning of each subtask the goal proposal network computes the goal with the highest value. In case of the tool-moving task the goal for the locomotion policy is computed to be at the location of the tool (first row in Fig. 4(a)(middle)). For moving the heavy object, both the agent and the tool have to be at the heavy box for it to be successfully moved. This is learned by the corresponding G, see Fig. 4(a)(right). This process is fully general and automatic. Figure 5 illustrates the function value and the proposed goals.

Reachability in space: We want to quantify how well the agent can move the heavy box to any location in space, i. e. the reachability. We consider a fine grid spanning the arena. For each point on the grid we measure the success-rate of reaching it with the heavy box from 20 random starting states of tool and heavy object. Figure 4(b) visualizes this success-rate for different snapshots of the system at the beginning, an intermediate state, and after mastering all tasks. It can be seen that the reachability grows with time and reaches almost full coverage.

4 Discussion

We present *control what you can* (CWYC) that makes an autonomous agent learn to control its observation space effectively. We impose a structure prior that is suitable for task planning while all components, except the state representation, are learned from scratch. CWYC shows superior performance in learning speed of controlling difficult parts of the environment, where the baseline fails. Our method uses learning progress as the driver for an automatic curriculum. which allows the agent to not invest resources in uncontrollable objects, nor try unproportionally often to improve its performance on not fully solvable tasks. This is different from approaches solely based on curiosity.

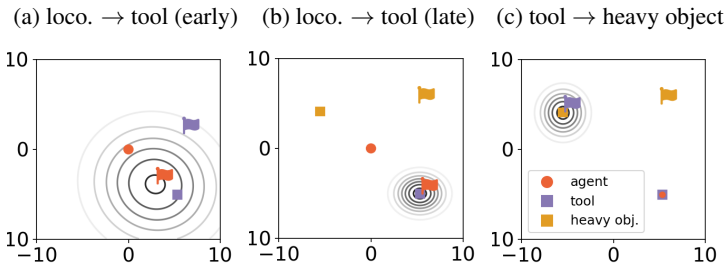

Figure 5: Visualized function value of the goal proposal networks. The concentric circles show the equi-value lines.

REFERENCES

Cédric Colas, Pierre Fournier, Olivier Sigaud, and Pierre-Yves Oudeyer. Curious: Intrinsically motivated multi-task, multi-goal reinforcement learning, 2018. arXiv preprint `https://arxiv.org/abs/1810.06284`.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning*, pp. 1514–1523, 2018.

Sébastien Forestier, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *arXiv preprint arXiv:1708.02190*, 2017.

Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Bol. Soc. Mat. Mexicana*, 5(2):102–119, 1960.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pp. 3675–3683, 2016.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, volume 2017, 2017.

Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing-solving sparse reward tasks from scratch. *arXiv preprint arXiv:1802.10567*, 2018.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, Oct 2012. doi: 10.1109/IROS.2012.6386109.

Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

SUPPLEMENTARY MATERIAL TO CONTROL WHAT YOU CAN: INTRINSICALLY MOTIVATED REINFORCEMENT LEARNER WITH TASK PLANNING STRUCTURE

Anonymous authors
Submission under double-blind review

The supplementary information is structured as follows. We start with a detailed explanation of all the components introduced in the methods section. We give the algorithmic details in the next section. We provide further information on the environment in Sec. C and on the training procedure in Sec. D. Sec. E elaborates on the goal proposal networks.

A METHOD DETAILS

In order to explain the architecture let us use a concrete example, already mentioned in the introduction. Assume, the environment contains two objects, a tool (forklift) which is easy to carry/transport and a heavy object which can only be moved when in possession of the tool.

We assume the state space $\mathcal{S}$ is partitioned into potentially controllable coordinates called goal spaces. These can be e. g. the agents position, object positions and so forth, for instance:

$$s = (\underbrace{x_{\text{self}}, y_{\text{self}}}_{\text{goal space 1}}, \underbrace{x_{\text{o1}}, y_{\text{o1}}}_{\text{goal space 2}}, \underbrace{x_{\text{o2}}, y_{\text{o2}}}_{\text{goal space 3}}, \dot{x}_{\text{self}}, \dot{y}_{\text{self}}, \dots), \tag{S1}$$

however, the semantics are unknown to the agent. Generally, some goal spaces might be hard to control, which will make the corresponding task difficult to achieve directly. In our example, the heavy object needs the tool that is somewhere in the environment. The agent should ideally discover relations between the subtasks/goal-spaces and how to optimally chain tasks. We propose an intrinsically motivated hierarchical learning framework based on a probabilistic graph and relational learning on top of a multi-goal reinforcement learning algorithm employed in each task.

A.1 TASKS AND GOAL SPACES

In our setting, a task $\tau \in \mathcal{T} = \{1, 2, \dots, K\}$ consists of reaching a goal g_i in a certain subspace of the observation space. The corresponding coordinates in s at denoted by m_i as in Eq. S1. For instance, if task 1 has its goal-space along the coordinates 3 and 4 then $m_1 = (3, 4)$ and s_{m_1} is the state values in that goal-space. The agent computes the reward for the low-level controller as the negative distance to the goal as $r_i = -\|s_{m_i} - g_i\|$ and declares success as:

$$\text{succ}_i = \begin{cases} 1 & \|s_{m_i} - g_i\| \leq \delta_i \\ 0 & \text{otherwise} \end{cases} \tag{S2}$$

where δ_i is a precision threshold.

We assume there is a certain fixed dependency among tasks. For instance, moving a heavy object needs the subtask of getting the fork-lift and this requires the subtask of locomotion before. Task dependencies and the task ordering are initially unknown to the learner and need to be inferred in an unsupervised manner.

A.2 INTRINSIC MOTIVATIONS AND FORWARD MODELS

In general, our agent is motivated to learn as fast as possible, i. e. to have the highest possible learning progress, and to be as successful as possible in each task. We use different measures to express intrinsic motivations:

Success rate $\mathrm{sr}_i = \mathbb{E}(\mathrm{succ}_i)$ this is estimated by the sample running mean of the last attempts.

Learning progress $\rho_i = \frac{\mathrm{d}\,\mathrm{sr}_i}{\mathrm{d}\,t}$ is the time derivative of the success rate, quantifying whether the agent gets better at task i.

However, initially, any success signals might be very sparse such that learning is slow because of uninformed exploration. Hence, we employ a proxy that guides the agents attention to tasks and states that might be *interesting*.

Prediction error $e_i(t)$ of an adaptive forward model in goal space i. We train a forward model $f\colon [\mathcal{S} \times \mathcal{A}] \to \mathcal{S}$ to learn the state changes induces by one step using the usual square-loss. The error is

$$e(t) = \|(f(s^t, a^t) + s^t) - s^{t+1}\|^2 \tag{S3}$$

and $e_i = e_{m_i}$ refers to error in the goal space of task i.

Surprising events $\mathrm{surprise}_i(t) \in [0, 1]$ is 1 if the time derivative of the prediction error $\dot{e}_{m_i}(t)$ in task i exceeds a confidence interval computed independently on each rollout, 0 otherwise (**?**).

To understand why surprising events can be informative, let us consider again our example: Assume the agent just knows how to move itself. It will move around and will not be able to manipulate other parts of its state-space, i. e. it can neither move the box nor the tool. Whenever it accidentally hits the tool, the tool moves and creates a surprise signal in the coordinates of the tool task. Thus, it is likely that this particular situation is a good starting point for solving the tool task. In addition, the tool task might be worth exploring.

A.3 Final task selector

The task selector Π [3] models the learning progress when attempting to solve a task and is implemented as a multi-armed bandit. While no learning progress is available, the surprise signal is used as a proxy. Thus, the internal reward signal for the bandit per rollout is

$$r^{\Pi}(i) = |\rho_i| + \beta^{\Pi} \max_t(\mathrm{surprise}_i(t)) \tag{S4}$$

with $\beta^{\Pi} \ll 1$. We use the absolute value of the learning progress $|\rho|$ because the system should both learn when it can improve, but also if performance degrades **?**. Initially, the surprise term dominates the quantity. As soon as actual progress can be made ρ takes the leading role. The reward is non-stationary and the action-value is updated according to

$$Q^{\Pi}(i) = Q^{\Pi}(i) + \alpha^{\Pi}(r^{\Pi}(i) - Q^{\Pi}(i)) \tag{S5}$$

with learning rate α^{Π}. The task selector is to choose the (final) task for each rollout relative to their value accordingly. We want to maintain exploration, such that we opt for a stochastic policy with $p^{\Pi}(\tau = i) = Q^{\Pi}(i) / \sum_i Q^{\Pi}(i)$. In our setup, the (final) goal within this task is determined by the environment (in a random fashion).

A.4 Task planner

Given the final task selected by the task selector, the agent has to decide which subtasks to perform. This is taken care of by the task planner, the higher level control of the hierachical reinforcement learning agent. The task planner models how well/quick the task i can be solved when doing task j directly before in terms of the time $T_{i,j}$ needed to solve task i. If the task cannot be solved $T_{i,j} = T_{\mathrm{T}}$, where T_{T} is the maximum number time steps in the rollout, there is no success signal. As before we use the surprising events as a proxy signal for potential future success. The values of each task transition are captured by $B(i, j)$, where $i \in [1, \ldots, K]$ and $j \in [S, 1, , \ldots, K]$ with S representing the start. Each entry of the B matrix is learned with a multi-armed bandit with action-values $Q^B(i, \cdot)$:

$$B(i, \cdot) = \text{normalize}(Q^B(i, \cdot)) \tag{S6}$$

$$Q^B(i, j) = \frac{T_{\mathrm{T}} - \langle T_{i,j} \rangle}{T_{\mathrm{T}}} + \beta^B \langle \max_t(\mathrm{surprise}_i(t)) \rangle \tag{S7}$$

where $\langle\cdot\rangle$ denotes a running average and $T_{i,j}$ is the runtime for solving task i by doing task j before.

Similarly to Eq. S4, this quantity is initially dominated by the surprise signals and later by the actual success values (nonzero if the time is shorter than the maximal time T_T). Thereby, $T_{i,j}$ not only reflects the competence of the low-level policy in solving the task, i. e. reaching a certain goal state, but also the quality of the goal proposal networks (discussed in the following section). Only if the reached goal state using policy j actually helps solving task i a success can be achieved.

The B matrix represents the adjacency matrix of the task graph, see Fig. 2[5]. It is used to construct a sequence of subtasks κ in the following way: Starting from the final task τ we draw the previous subtask (option) with an ϵ-greedy policy using $B(\tau, \cdot)$. Then this is repeated for the next subtask, until S (start) is sampled, see also Fig. 2[4] and [5].

A.5 Goal proposal: relational attention network

Each (sub)task is itself a goal-reaching problem. In order to decide which subgoals need to be chosen we employ an attention network for each task transition, i. e. $G_{i,j}$ for the transition from task j to task i. As before, the aim of the goal proposal network $G_{i,j}$ is to maximize the success rate of solving task i when using the proposed goal in task j before. In the example, in order to pick up the tool, the goal of the preceding locomotion task should be the location of the tool. An attention network that can learn relations between observations is required. We use an architecture that models local pairwise distance relationships. It associates a value/attention to each point in the goal-space of the preceding task as a function of the state s: $G_{i,j} : \mathcal{S} \to \mathbb{R}$: (omitting index $_{i,j}$)

$$G(s) = e^{-\gamma \sum_{k=1}^{n} \sum_{l=k+1}^{n} \|w^1_{kl} s_k + w^2_{kl} s_l + w^3_{kl}\|^2} \tag{S8}$$

where w^1, w^2, w^3, and γ are trainable parameters. The network is trained using a square-loss with the following target signal $r^G_{i,j}(s_t) \in [0, 1]$:

$$r^G_{i,j}(s_t) = \min(1, \mathrm{succ}_i \cdot \Gamma_{i,j}(s_t) + \mathrm{surprise}_i(t)) \tag{S9}$$

for all s_t that occurred during task j where $\Gamma_{i,j}(s)$ is 1 if the switching state from task j to task i occurred in state s and zero otherwise. To get an intuition about the parametrization, consider a particular pair of coordinates (k, l), say agent's and tool's x-coordinate. The model can express with $w^1_{k,l} = -w^2_{k,l} \neq 0$ that both have to be at distance zero for r^G to be 1. However, with w^3 the system can also model offsets, global reference points and other relationships. Details on the architecture and training can be found in E. We observe that the goal proposal network can learn a relationship after a few examples (in the order of 10), possibly due to the restricted model class. The goal proposal network can be thought of as a relational network **?**.

A.6 Subgoal sampling

For each subtask the goal is selected with the maximal value in the attention map. However, coordinates of tasks that are still to be solved in the task-chain are fixed, because they can likely not be controlled by the current policy. Formally:

$$\begin{aligned} s^* = \arg\max_{s'} \quad & G_{i,j}(s') \\ \text{subject to } \; & s'_{m_k} = s_{m_k}, \; \forall k \in \kappa(i+) \end{aligned} \tag{S10}$$

where κ is the task-chain and $\kappa(i+)$ denotes all tasks after i and including i. The goal for subtask j is then $\mathrm{goal}_j(s) = s^*_{m_j}$. Thanks to the parametrization, the solution of Eq. S10 can be computed analytically.

A.7 Low-level control

Each task i has its own policy π_i which is trained separately using an off-policy deep RL algorithm. We use soft actor critic (SAC) **?**, where the policy and the critic networks are parametrized by the goal (UVFA (**?**)).

A.8 TRAINING / OVERALL PROCEDURE

All components are initialized randomly. A rollout starts with the (final) task determined by the task selector. The task planner constructs the task chain κ. For the first task in κ a subgoal computed by the goal proposal network. Given the subgoal the goal-parametric policy of that task is used. Whenever the goal is reached (up to a certain precision) a switch to the next task occurs. Again the goal proposal network is employed to select a goal in this task, unless it is the final task where the final goal is obviously used. If a goal cannot be reached the task ends after T_{T} steps. In practice we run 5 rollouts in parallel. Then all components are trained using the collected data. For the task selector and task planner we use Eq. S5 and Eq. S7, respectively. Forward model and Gs are trained using square-loss and Adam (**?**). The policies are trained according to SAC. Pseudo-code and implementation details can be found in the Supplementary Material (B, D).

B PSEUDOCODE

Algorithm 1 CWYC

for epoch in epochs **do**
  for cycle in cycles **do**
    sample main task $\tau_{\text{final}} \sim \Pi$
    sample main goal $g_{\tau_{\text{final}}}$ from environment
    compute task chain κ from $B(\tau_{\text{final}})$
    // κ contains list task indices
    $i = 1$
    while $t < T$ and no success in τ_{final} **do**
      $\tau = \tau_{\kappa[i]}$
      if $\tau \neq \tau_{\text{final}}$ **then**
        sample goal g_τ from $G(\kappa[i], \kappa[i+1])$
      end if
      try to reach g_τ with policy$_\tau$
      if success$(\kappa[i])$ **then**
        $i = i + 1$ *// next task in task chain*
      end if
    end while
    store episode in history buffer
    calculate statistics based on history
    train policies for each task
    train B *// Sec. A.4*
    train all G *// Sec. A.5*
    train Π *// Sec. A.3*
  end for
end for

C ENVIRONMENT

The environment that we use is depicted in Fig. 1 and is simulated by the physics engine MuJoCo. The agent is modeled by a ball that is controlled by applying force in the x and y axis, so the agent's action corresponds to a 2-dimensional vector:

$$a = (F_x, F_y) \tag{S11}$$

The motion of the agent is subject to the laws of motion with the application of friction from the environment which makes it non-trivial to control. Other than the agent, the environment contains objects with different dynamics. The positions of the objects are part of the observation space of the agent along with a flag that specifies if the object has been picked up by the agent. We are dealing with a fully observable environment.

We define the goal spaces of the tasks as corresponding to the position of the individual objects. Some objects are harder to move than others and have other objects as dependencies. This means that the agent has to find this relation between them in order to successfully master the environment.

The types of objects that are used in the experiments are the following:

- *Static objects* cannot be moved
- *Random objects* move randomly in the environment, but cannot be moved by the agent
- *50% light objects* can be moved in 50% of the rollouts
- *Tool* can be moved and used to move the heavy object
- *Heavy objects* can be moved when using the tool

D TRAINING DETAILS AND PARAMETERS

- **Training:**

# parallel rollout workers:	5
# rollouts (per cycle):	5 (1 per worker)
# cycles (per epoch):	10

- **Environment:**

arena size:	20×20
T_{T}:	1600

- **SAC:**

lr:	3×10^{-4}
batch size:	64
policy type:	gaussian
discount:	0.99
reward scale:	5
target update interval:	1
tau (soft update)	5×10^{-3}
action prior:	uniform
reg:	1×10^{-3}
layer size (π, q, v):	256
# layers (π, q, v):	2
# train iterations:	200
buffer size:	1×10^{6}

- **Forward model:**

lr:	10^{-4}
batch size:	64
input:	(o_{t-1}, u_{t-1})
confidence interval:	5
network type:	MLP
layer size:	64
# layers:	2
# train iterations:	100

- **Final task selector:**
  β^{Π}: 10^{-2}
  α^{Π}: 10^{-1}
- **Task planner:**
  β^{B}: 10^{-1} avg. window size: 100
- **Goal proposal network:**

lr:	10^{-3}
batch size:	64
L1 reg.:	10^{-5}
L2 reg.:	10^{-5}
γ init:	1.0
γ trainable:	False
# train iterations:	30

E TRAINING DETAILS OF THE GOAL PROPOSAL NETWORK

In an ever-changing environment as the ones presented in this paper, the goal proposal networks are a critical component of our framework that aim to learn relations between entities in the world. Transitions observed in the environment are labeled by the agent in interesting and undetermined transitions. Interesting transitions are those, in which a surprising event (high prediction error) occurs or which lead to an success in task i given some other task j was solved before, see Eq. S9. All other transitions are labeled as undetermined, since they might contain transition which are similar to those that are labeled interesting but didn't spark high interest. Coming back to our running example: bumping into, hence suddenly moving, the tool might spark interest in the tool because of a suddenly jump in prediction error. In general, the behaviour of an object after the surprising event is unknown and label for these transitions is not clear.

Conclusively, we discard all undetermined transition within a rollout that come after a transition with positive label. After discarding these transition from the unlabeled data, there is still data that is either very similar to positively labeled data but did not spark interest, e.g., all the transitions where the agent is really close to the object but does not touch it yet. We reduce the impact of this data by classifying the undetermined data in a training batch with the latest version of the network and discard 20% of the data that is most similar to the positive labeled data. This is inspired by techniques in PU learning [Li & Liu, "Learning from Positive and Unlabeled Examples with Different Data Distributions.", ECML 2005].

After removing all data that might prevent the goal proposal networks from learning the right relations it remains the problem that positive events are rare compared to the massive body of undetermined data. Hence, we balance the training data in each batch during training.

To make efficient use of the few positive samples we collect in the beginning of the training we impose a structural prior on the goal proposal network given by Eq. S8. The weight matrices are depicted in Fig. S1. This particular structure restricts the hypothesis space of the component to positional relations between components in the observation space that contains entities in the environment. In the main text, Figure 4 shows a compact representation of the initial and final weight matrices for different tasks that are computed by taking the minimum over w^1 (left column) and w^2 (middle column) in Fig. S1.

To understand the parametrization, consider to model that two components (k, l) of s should have the same value for a possitive signal, then $w^1_{kl} \approx -w^2_{kl}$ should be nonzero and $w^3_{kl|} = 0$. In this case the corresponding term in the exponent of Eq. S9 is zero if $s_l = s_k$. We see that in the case of the learned G in Fig. S1 this relationship is true for the relevant components (position of agent, tool and object).

F HAND-CRAFTED UPPER BASELINE

To assess the maximum performance of CWYC in the described settings, we crafted an upper baseline in which all learned components, except for the final task selector Π, are fixed and set to their optimal value.

In the distractor setting, every task is solved by first doing the locomotion task. The goal proposal network $G_{i,j}(s)$ returns always the state value s_{m_i}, reflecting the ground truth relation we try to learn.

In the tool-use setting, the task graph depicted in Figure S2 is used. As in the distractor setting, $G_{i,j}(s)$ returns always the state value s_{m_i}.

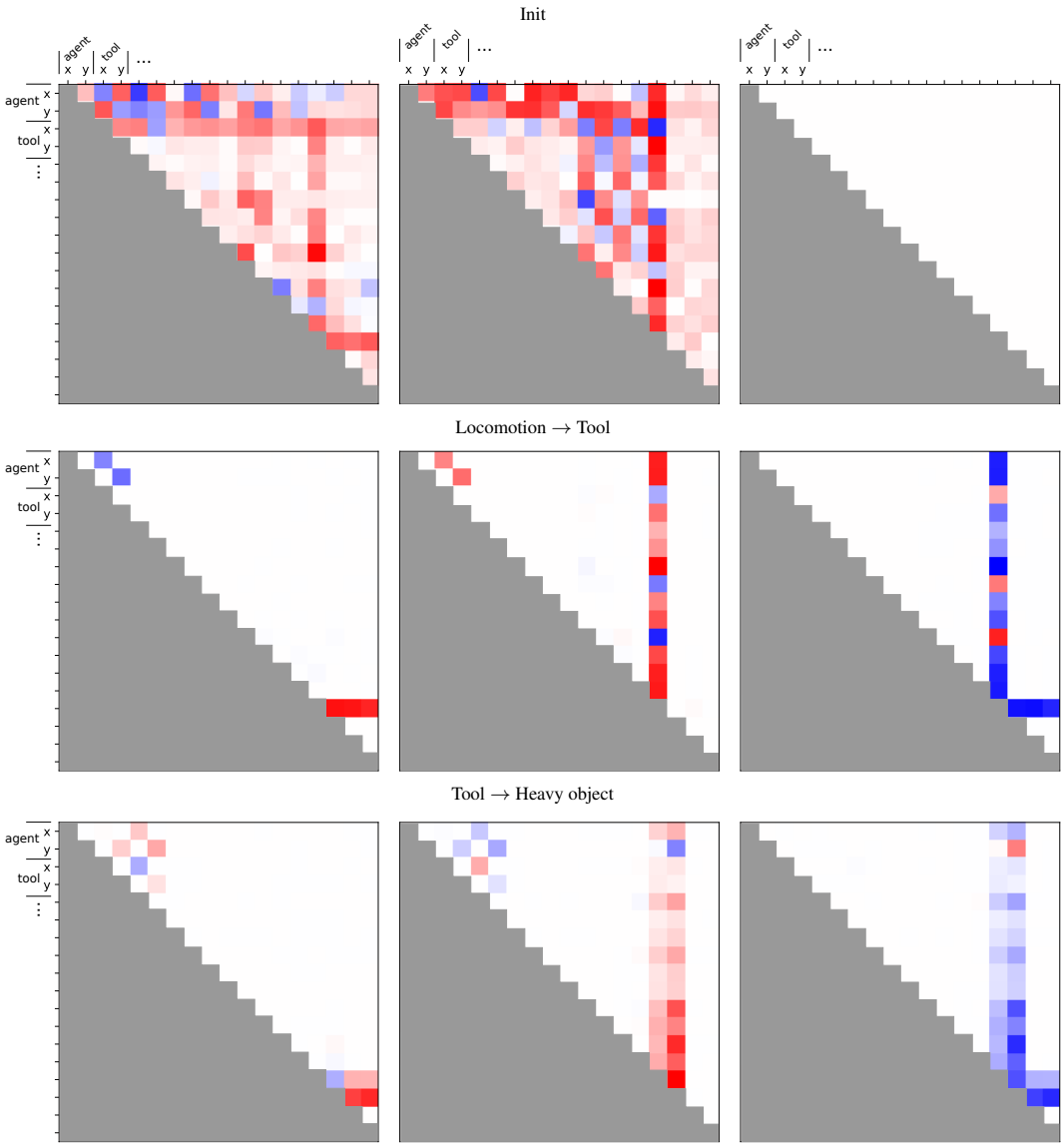

Figure S1: Weights learned by goal proposal networks for different task transitions. The left column shows the weights of w^1, the middle column of w^2 and the right column of w^3 (see Eq. S8).

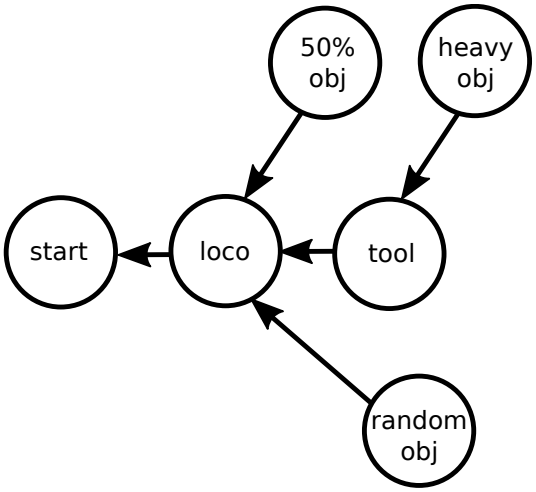

Figure S2: Crafted dependency graph.

G INTRINSIC MOTIVATIONS

For computing the success rate we use a running mean of the last $Z = 50$ attempts of the particular task:

$$\mathrm{sr}_i(\text{cycle}) = {}^1\!/_Z \sum_{z=0}^{Z} \mathrm{succ}_i(-z) \tag{S12}$$

where $\mathrm{succ}_i(-z)$ denotes the z-th last rollout where task i was attempted to be solved.

The learning progress ρ_i is then given as the finite difference of sr_i between subsequent attempts of task i.

To compute the surprise signal $\mathrm{surprise}_i$ we compute the statistics of the prediction error derivative over one rollout, i. e. we assume

$$(e_i(t) - e_i(t-1)) \sim \mathcal{N}(\mu_i, \sigma_i^2) \tag{S13}$$

and compute the empirical μ and σ. Denoting the finite difference by $\dot{e}_i$, surprise within one rollout is then defined as

$$\mathrm{surprise}_i(t) = \begin{cases} 1 & \text{if } |\dot{e}_i(t)| > \mu_i + 5.0\sigma_i \\ 0 & \text{otherwise.} \end{cases} \tag{S14}$$