Supplementary Material for Learning Temporally Extended Skills in Continuous Domains as Symbolic Actions for Planning

Jan Achterhold **Markus Krimmel** **Joerg Stueckler**
Embodied Vision Group, Max Planck Institute for Intelligent Systems, Tübingen, Germany
{jan.achterhold,markus.krimmel,joerg.stueckler}@tuebingen.mpg.de

S.1 Introduction

In the following we provide supplementary details and analysis of our approach. We show visualisations of the learned skills in Sec. S.2 and provide an analysis for solution depths > 5 in Sec. S.3. In Sec. S.4 we introduce additional embedded *LightsOut* environments, featuring a 3D board and larger board sizes, to evaluate the exploration capabilities and limitations of our method. In Sec. S.5 we give details on the robot experiment, and provide additional ablation results in Sec. S.6. Architectural and implementation details are given in Sec. S.8 for `SEADS`, in Sec. S.9 for the SAC baseline and in Sec. S.10 for the HAC baseline. In Sec. S.11 we compare `SEADS` to a variant which uses a skill discriminator instead of a forward model as in "Variational Intrinsic Control" (VIC, Gregor et al. [1]). In Sec. S.12 we detail how we define training and test splits on our proposed environments. Finally, we present detailed results for hyperparameter search on the SAC and HAC baselines in Sec. S.13.

S.2 Learned skills

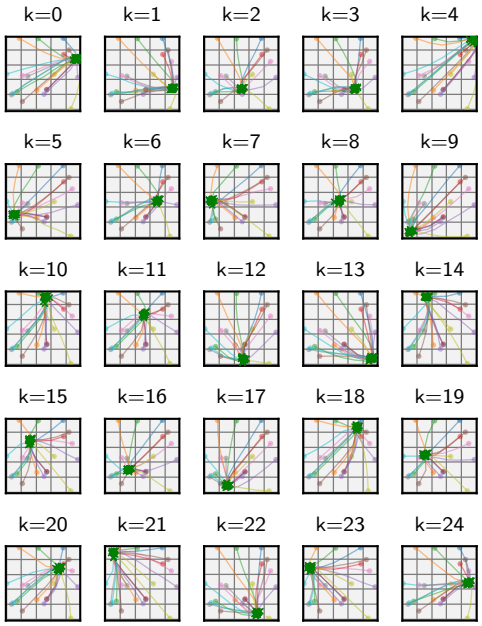
We provide additional visualizations on the behaviour of `SEADS` in Fig. S.1, showing that `SEADS` learns to assign skills to pushing individual fields on the game boards.

(a) Contact points of *Jaco* end effector.

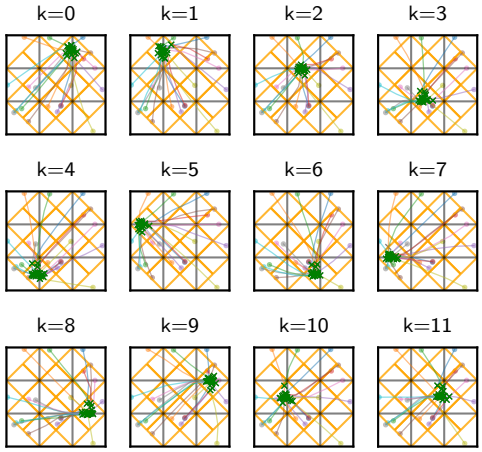
(b) Contact points of *Reacher* end effector.

Figure S.1: Contact points of the *Jaco* (*Reacher*) end effector in the `LightsOutCursor` (`LightsOutReacher`) environments when executing skill $k \in \{1, ..., 25\}$ on 20 different initializations of the environment. Each skill is assigned a unique color/marker combination. We show the agent performance after 1×10^7 environment steps. We observe that the `SEADS` agent learns to push individual fields as skills.

6th Conference on Robot Learning (CoRL 2022), Auckland, New Zealand.



(a) Skill trajectories on `LightsOutCursor`.



(b) Skill trajectories on `TileSwapCursor`.

Figure S.2: Trajectories in *Cursor*-embedded environments for skills k on 20 different environment initializations. Colored lines show the x, y-coordinates of the Cursor, with the circular marker indicating the start position of the skill. Black markers indicate locations where a "push" was executed, and green markers where the push has caused a change in the symbolic state.

S.2.1 Skill trajectories

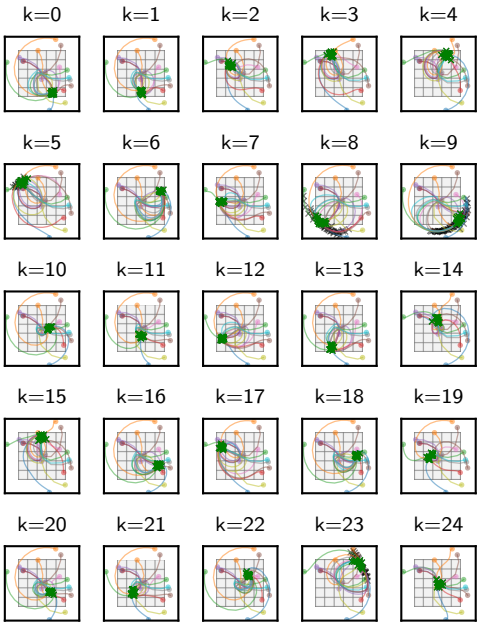
In Figs. S.2, S.3 and S.4 we provide visualizations of trajectories executed by the learned skills on the *Cursor*, *Reacher* and *Jaco* environments.

S.2.2 Solution length

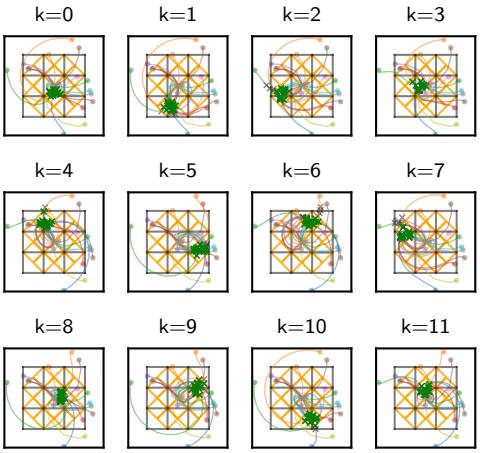
In Fig. S.5 we provide an analysis how many low-level environment steps (i.e., manipulator actions) are executed to solve instances of the presented physically embedded board games. We follow the evaluation procedure of the main paper (Fig. 3) and show results for 10 trained agents and 20 initial board configurations for each solution depth in $\{1, ..., 5\}$. We only report results on board configurations which are successfully solved by `SEADS`. We observe that even for a solution depth of 5, for the Cursor environments, only relatively few environment steps are required in total to solve the board game (≈ 15 for `LightsOutCursor`, ≈ 10 for `TileSwapCursor`). In contrast, the more complex *Reacher* and *Jaco* environments require significantly more interactions to be solved, with up to 400 steps executed on `TileSwapJaco` for a solution depth of 5 and enabled re-planning.

S.2.3 Skill length

Following the evaluation procedure from Sec. S.2.2 we report the distribution of skill lengths (i.e., number of actions applied to the manipulator per skill) in Fig. S.6. Again, we only include problem instances which were successfully solved by `SEADS`. While skill executions on the *Cursor* environments are typically short (median 3/2 manipulator actions for `LightsOutCursor`/`TileSwapCursor`), the *Reacher* and *Jaco* environments require a higher number of manipulator actions per skill (median 11/12/9/13 for `LightsOutReacher`/`LightsOutJaco`/`TileSwapReacher`/`TileSwapJaco`).

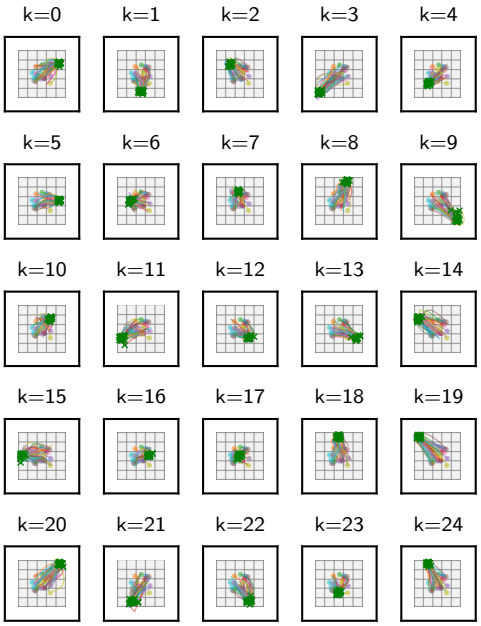


(a) Skill trajectories on `LightsOutReacher`.

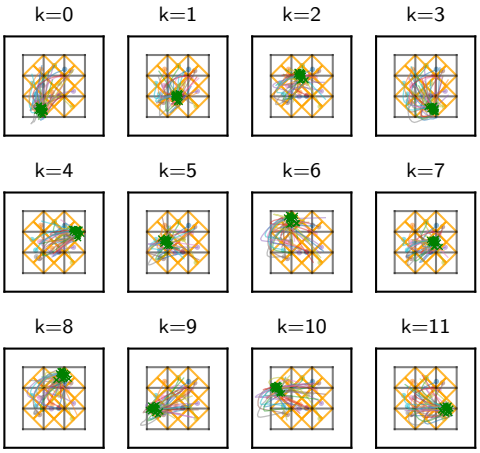


(b) Skill trajectories on `TileSwapReacher`.

Figure S.3: Trajectories in *Reacher*-embedded environments for skills k on 20 different environment initializations. Colored lines show the x, y-coordinates of the Reacher end-effector, with the circular marker indicating the start position of the skill. Black markers indicate locations where a "push" was executed, and green markers where the push has caused a change in the symbolic state.



(a) Skill trajectories on `LightsOutJaco`.



(b) Skill trajectories on `TileSwapJaco`.

Figure S.4: Trajectories in *Jaco*-embedded environments for skills k on 20 different environment initializations. Colored lines show the x, y-coordinates of the Jaco hand, with the circular marker indicating the start position of the skill. Green markers indicate contact locations with the board.

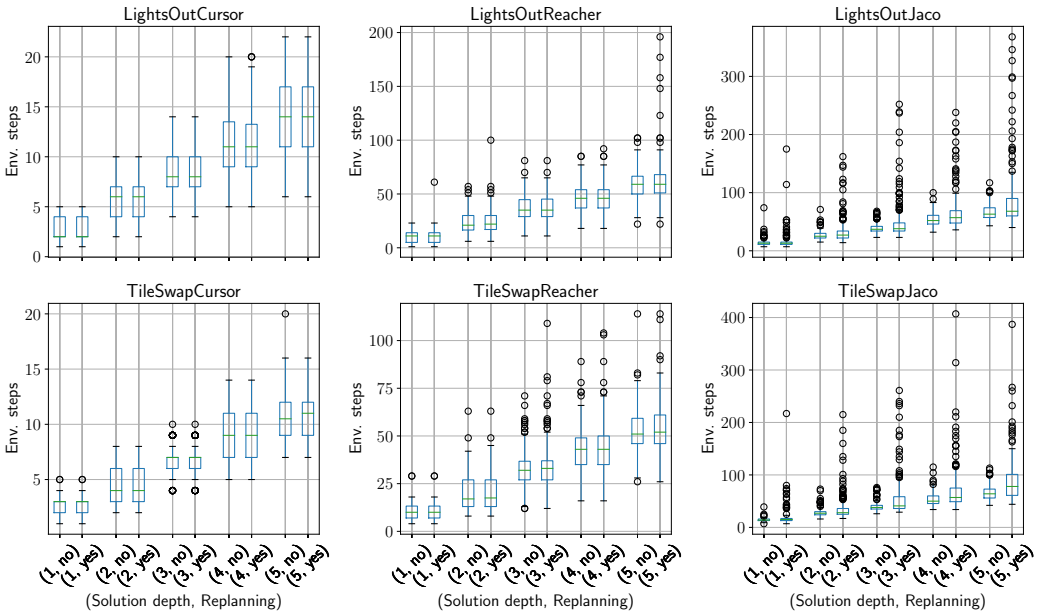


Figure S.5: Analysis of solution lengths (total number of manipulator steps required to solve the physically embedded board games) for different environments, solution depths and planning with/without re-planning. Box-plots show the 25%/75% quartiles and median (green line). Whiskers extend to the farthest datapoint within the 1.5-fold interquartile range. Outliers are plotted as circles.

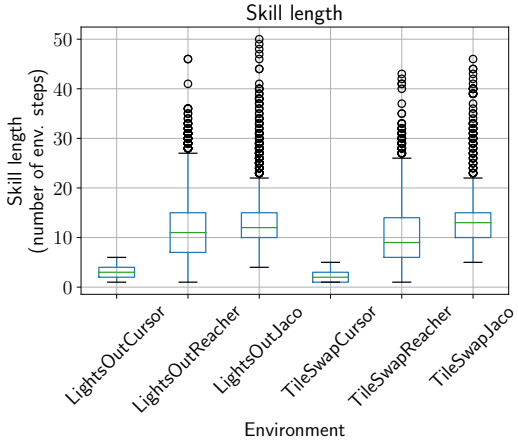


Figure S.6: Analysis of skill lengths (number of manipulator steps executed within a single skill) for different environments. Box-plots show the 25%/75% quartiles and median (green line). Whiskers extend to the farthest datapoint within the 1.5-fold interquartile range. Outliers are plotted as circles.

S.3 Large solution depth analysis

In Fig. S.7 we present an analysis for solving *LightsOut* tasks with solution depths > 5 using the learned `SEADS` agent on `LightsOutCursor`. We observe a high mean success rate of $\geq 98\%$ for solution depths ≤ 8. However, the time required for the breadth-first search planner to find a feasible plan increases from $\approx 2.02s$ for solution depth 5 to $\approx 301.8s$ for solution depth 9. We abort BFS planning if the list of nodes to expand exceeds a size of ≈ 16 GB memory usage. All experiments were conducted on Intel® Xeon® Gold 5220 CPUs with a clock rate of 2.20 GHz.

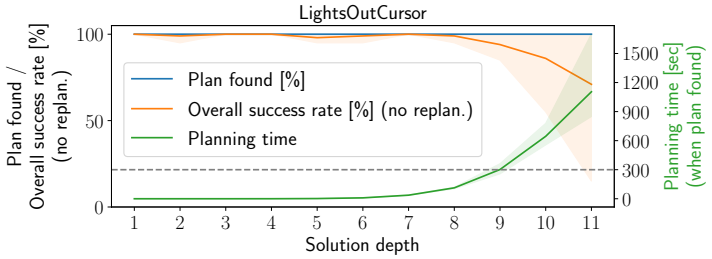


Figure S.7: Analysis of the `LightsOutCursor` environment for solution depths > 5. "Solution depth" is the number of game moves required to solve the *LightsOut* instance. "Plan found" refers to the ratio of problem instances where BFS finds a feasible plan. "Overall success rate" quantifies the ratio of problem instances for which a feasible plan was found and, in addition, was successfully executed by the low-level policy. The "Planning time" refers to the wall-time the breadth-first search planner runs, for the cases in which it finds a feasible plan. We report results for 20 problem instances for each solution depth for 5 independently trained agents. We refer to sec. S.3 for details. The solid line refers to the mean, shaded area to min/max over 5 independently trained agents.

S.4 Additional LightsOut variants

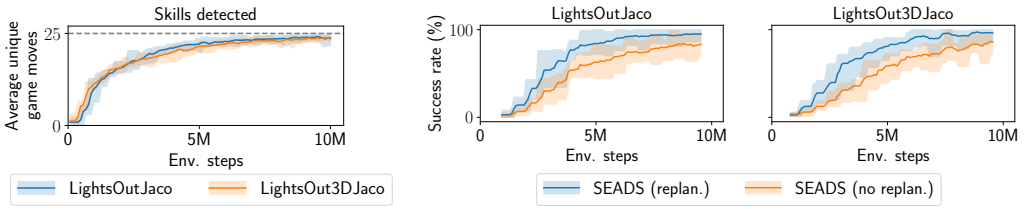
S.4.1 LightsOut3DJaco

In addition to the `LightsOutJaco` environment presented in the main paper we introduce an additional environment `LightsOut3DJaco`. While in `LightsOutJaco` the LightsOut board is a flat plane, in `LightsOut3DJaco` the fields are elevated/recessed depending on their distance to the board's center (see Fig. S.8). This poses an additional challenge to the agent, as it has to avoid to push fields with its fingers accidentally during skill execution. Despite the increased complexity of `LightsOut3DJaco` over `LightsOutJaco` we observe similar results in terms of detected game moves (Fig. S.9a) and task performance (Fig. S.9b). Both environments can be solved with a high success rate of 94.9% (`LightsOutJaco`) and 96.3% (`LightsOut3DJaco`). For the experiments, we follow the evaluation protocols from the main paper.

Figure S.8: `LightsOut3DJaco`: A variant of the `LightsOutJaco` environment with elevated fields. In comparison to the `LightsOutJaco` environment this environment poses an additional challenge to the agent, as it has to avoid to push fields with its fingers accidentally during skill execution. We show the execution of a skill which has learned to push the center field for $T = \{0, 4, 8, 12, 14\}$.

S.4.2 Larger LightsOut boards / spacing between fields

In this experiment we investigate how well `SEADS` performs on environments in which a very large number of skills has to be learned and where noisy executions of already learned skills uncover



(a) Average number of unique game moves detected on `LightsOutJaco` and `LightsOut3DJaco`.

(b) `SEADS` task performance on `LightsOutJaco` and `LightsOut3DJaco`, with and without replanning.

Figure S.9: Evaluation on number of (a) detected game moves and (b) task performance on `LightsOut3DJaco` (see sec. S.4.1 for details).

new skills with low probability. To this end, we modified the `LightsOutCursor` environment to have more fields (and thereby, more skills to be learned), and introduced a spacing between the tiles, which makes detecting new skills more challenging. For a fair comparison, we keep the total actionable area in all environments constant, which introduces an empty area either around the board or around the tiles (see Fig. S.10). We make two main observations: (i) As presumed, learning skills in the `LightsOutCursor` environment with spacing between tiles requires more environment interactions than for adjacent tiles (see Fig. S.11a) (ii) For boards up to size 9×9 a large majority of skills is found after 1.5 million environment steps of training ($5 \times 5 : 24.9/25$, $7 \times 7 : 48.6/49$, $9 \times 9 : 76.8/81$). The `LightsOutCursor` environment with boardsize 13×13 poses a challenge to `SEADS` with $119.3/169$ skills detected after 1.5 million environment steps (see Fig. S.11). The numbers reported are averages over 5 independently trained agents.

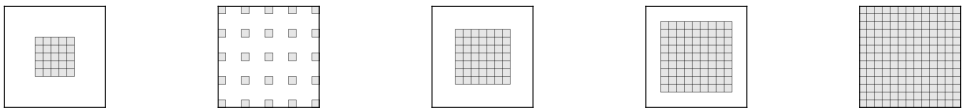
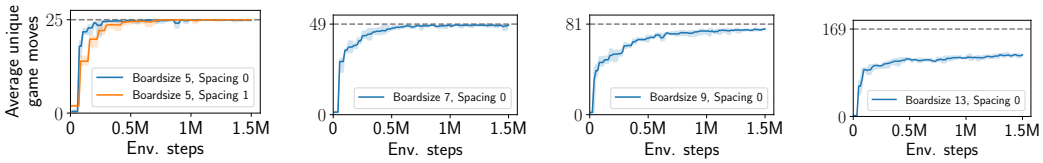


Figure S.10: Variants of `LightsOutCursor` for different board sizes (5×5, 7×7, 9×9, 13×13) and spacing introduced between fields (second panel from left).



(a) Boardsize 5×5 (b) Boardsize 7×7 (c) Boardsize 9×9 (d) Boardsize 13×13

Figure S.11: Detected average unique game moves (skills) on `LightsOutCursor` environment for different board sizes (5×5, 7×7, 9×9, 13×13) and spacing. The introduced spacing slows down skill learning (a). While for boards until size 9×9 nearly all skills are found (a-c), the 13×13 environment poses a challenge to `SEADS` (d). Solid line: mean / Shaded area: min/max over 5 independently trained agents.

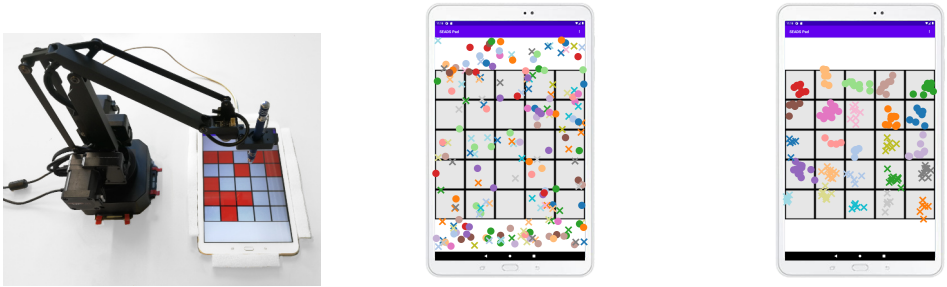
S.5 Real Robot Experiment

For the real robot experiment we use a uArm Swift Pro robotic arm that interacts with a *LightsOut* board game. The board game runs on a Samsung Galaxy Tab A6 Android tablet with a screen size of 10.1 inches. We mapped the screen plane excluding the system status bar and action bar of the app (blue bar) to normalized coordinates $(x, y) \in [0, 1]$. A third $z \in [0, 1]$ coordinate measures the perpendicular distance to the screen plane, with $z = 1$ approximately corresponding to a distance

of $10\,\mathrm{cm}$ to the screen. To control the robot arm, we use the Python SDK from [2], which allows to steer the end effector to $\overrightarrow{X} = (X, Y, Z)$ target locations in a coordinate frame relative to the robot's base. As the robot's base is not perfectly aligned with the tablet's surface, e.g. due to the rear camera, we employed a calibration procedure. We measured the location of the four screen corners in (X, Y, Z) coordinates using the SDK's `get_position` method (by placing the end effector holding the capacitive pen on the particular corners) and fitted a plane to these points minimizing the squared distance. We reproject the measured points onto the plane and compute a perspective transform by pairing the reprojected points with normalized coordinates $(x, y) \in \{0, 1\} \times \{0, 1\}$. To obtain robot coordinates (X, Y, Z) from normalized coordinates (x, y, z) we first apply the perspective transform on (x, y), yielding $\hat{X} = (X, Y, Z = 0)$. We subsequently add the plane's normal to (X, Y, Z) scaled by z and an additional factor which controls the distance to the tablet's surface for $z = 1$. The state of the board is communicated to the host machine running `SEADS` via USB through the logging functionality of the Android Debug Bridge. The whole system including robotic arm and Android tablet is interfaced as an `OpenAI Gym` [3] environment.

S.5.1 Training on robot with absolute push position actions

In a first variant, the action space of the robotic environment is 2-dimensional, comprising a normalized pushing coordinate $(x, y) \in [0, 1]$ which is translated into a sequence of three commands sent to the robot, setting the position of the end effector to $(x, y, z = 0.2)$, $(x, y, z = 0)$, $(x, y, z = 0.2)$ subsequently. To simulate a more realistic gameplay, we do not resample the LightsOut board state or the robots' pose at the beginning of an episode. We show the setup and behaviour during training in Fig. S.12. After 5000 environment interactions (corresponding to ≈ 7.5 hours total training time) we evaluated the SEADS agent's performance on 100 board configurations (20 per solution depth in $\{1, ..., 5\}$) and found all of them to be successfully solved by the agent.



(a) Robotic arm (uArm Swift Pro) interacting with a LightsOut game running on an Android tablet.

(b) Contact points of 200 skill executions with randomly sampled skills $k \in \{1, ..., 25\}$ at the beginning of training.

(c) Contact points of the last 200 skill executions with randomly sampled skills $k \in \{1, ..., 25\}$ when training `SEADS` on 5000 environment interactions.

Figure S.12: Real-world setup with a robotic arm (a), on which `SEADS` learns a symbolic forward model on the LightsOut board state and associated low-level skills which relate to pushing locations on the tablet's surface. In (b) and (c) we depict the first 200 and last 200 pushing locations of 5000 pushes used for training in total. While pushing locations are randomly scattered at the beginning of training (b), in the last 200 of 5000 training interactions skills relate to pushing particular fields on the game board (c).

S.5.2 Training on robot with positional displacement actions

In this experiment the action space of the environment is 3-dimensional $a = (\Delta x, \Delta y, p)$, with the first two actions being positional displacement actions $\Delta x, \Delta y \in [-0.2, 0.2]$ and the third action $p \in [-1, 1]$ indicating whether a "push" should be executed. The displacement actions represent incremental changes to the robotic arm's end effector position. In normalized coordinates (see sec. S.5) the end effector is commanded to steer to $(\mathrm{clip}(x + \Delta x, 0, 1), \mathrm{clip}(y + \Delta y, 0, 1), z = 0.3)$, where (x, y) are the current coordinates of the end effector. If the push action p exceeds a threshold $p > 0.6$, first the end effector is displaced, followed by a push, which is performed by sending the

target coordinates $(x, y, z = 0), (x, y, z = 0.3)$ to the arm subsequently. In contrast to the first variant in which the SEADS agent sends a push location to the agent directly, here the SEADS agent has to learn temporally extended skills which first locate the end effector above a particular board field and then execute the push. Therefore, to reach a high success rate on the *LightsOut* task, significantly more environment interactions are required compared to the first variant. We observe that a test set of 25 *LightsOut* instances (5 per solution depth in $\{1, ..., 5\}$) is solved with a success rate of 100% after $\approx 165k$ environment interactions, taking in total ≈ 43.5 hours wall-time to train. We refer to Fig. S.13 for a visualization of the success rate of SEADS over the course of training and to Fig. S.14 for a visualization of skills learned after $\approx 220k$ environment interactions. *We refer to the video on the real robot experiment provided with the supplemental materials.*

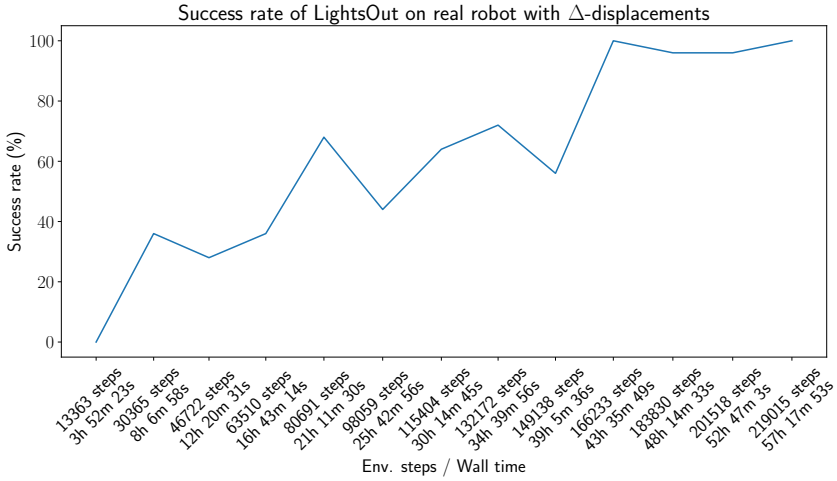


Figure S.13: Task success rate during the course of training SEADS on the *LightsOut* game embedded into a physical manipulation scenario with the uArm Swift Pro robotic arm. The arm is controlled by SEADS with positional displacement actions (see S.5.2). We evaluate the task performance as success rate on 25 board configurations for each reported step (5 instances per solution depth in $\{1, ..., 5\}$).

S.6 Extended ablations

In this section we present an extended ablation analysis. We refer to sec. 4 of the main paper for a description of the evaluation protocol.

S.6.1 Relabelling

In this section we provide additional ablations for the episode relabelling. In addition to the "No relabelling" ablation considered in the main paper we investigate not relabelling episodes for the SAC agent only (No SAC relabelling) and not relabelling episodes for training the forward model (No forw. model relabelling). All evaluations are performed on 10 individually trained agents. We refer to Fig. S.15 for a visualization of the results. The results demonstrate that relabelling both for the forward model and SAC agent training are important for the performance of SEADS.

S.6.2 Numerical results

In Table 1 we present the number of average unique game moves executed as skills by SEADS and its ablations. On the simpler Cursor environments, the ablations perform similarly well as SEADS in finding almost all skills. On the Reacher environments, most important for the performance of SEADS is to perform relabelling. On LightsOutJaco, all ablated design choices are important for the high performance of SEADS.

We also observe that the "More skills" variant (equivalent to SEADS, but with $K = 30$ for *LightsOut*, $K = 15$ for *TileSwap*) yields a similar number of executed unique game moves as SEADS, which is

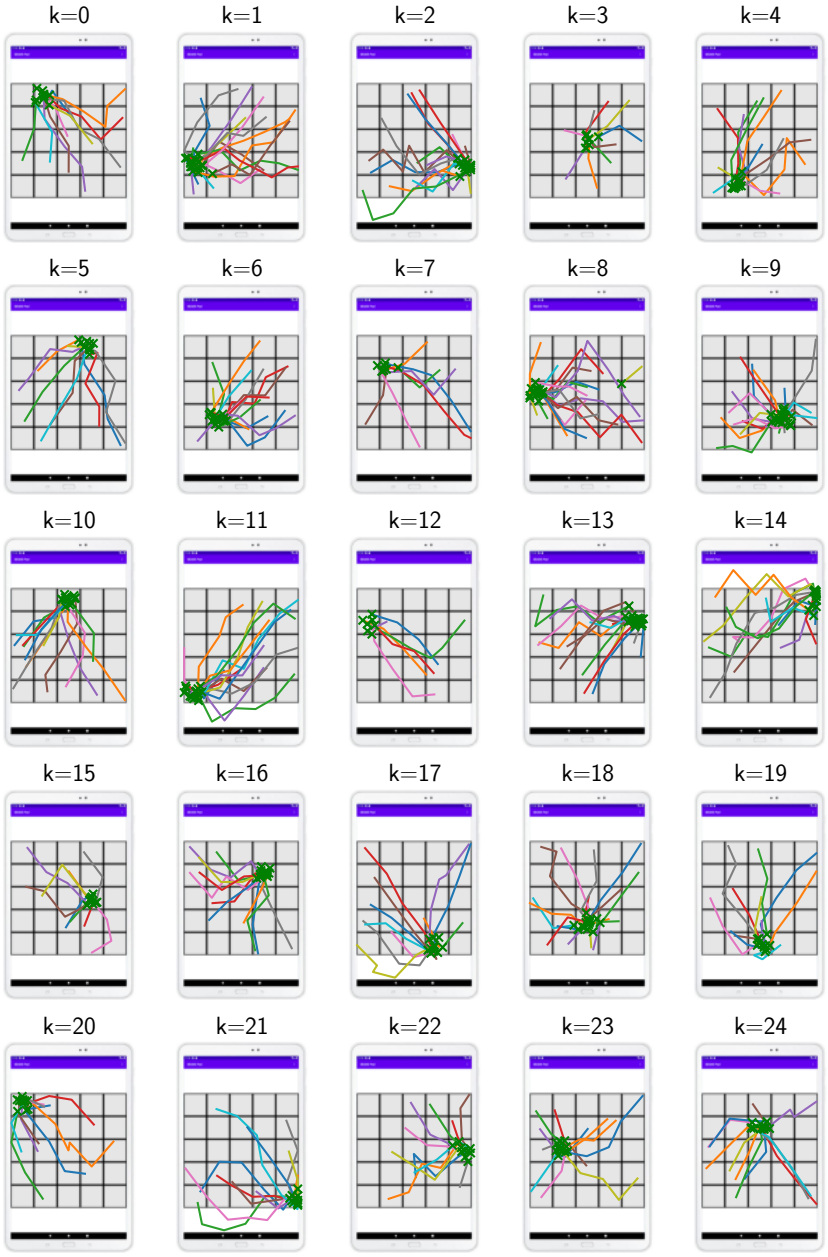


Figure S.14: Visualization of 320 trajectories executed on the uArm Swift robotic arm with positional displacement actions, after the agent has been trained for $\approx 220k$ environment interactions. Each subpanel shows trajectories for a specific symbolic action k. Green markers indicate push locations. Similar to the other environments, SEADS has learned to push individual fields as skills.

an encouraging result, justifying to over-estimate the number of skills K in situations where it is unknown.

We perform one-sided Mann-Whitney U tests [4] to conclude about significance of our results. On each environment, for each of the 10 independently trained SEADS agents, we obtain a set of 10 samples on the average number of detected skills. Analogously, we obtain such a set of 10 samples for every ablation. We aim at finding ablations which either (i) detect significantly more skills than SEADS or (ii) detect significantly less skills than SEADS on a particular environment. We reject null hypotheses for $p < 0.01$. For (i), we first set up the null hypothesis that the distribution

underlying the SEADS samples is stochastically greater or equal to the distribution underlying the ablation samples. For ablations on which this null hypothesis can be rejected it holds that they detect significantly more skills than SEADS. *As we cannot reject the null hypothesis for any ablation, no ablation exists which detects significantly more skills than SEADS.* For (ii), we set up the null hypothesis that the distribution underlying the SEADS samples is stochastically less or equal to the distribution underlying the ablation samples. *For all ablations there exists at least one environment in which we can reject the null hypothesis to (ii), indicating that all ablations contribute significantly to the performance of SEADS on at least one environment*. The results of the significance test are highlighted in Table 1.

	Cursor		Reacher		Jaco	
	LightsOut	TileSwap	LightsOut	TileSwap	LightsOut	TileSwap
SEADS	24.94 ± 0.06	11.99 ± 0.02	24.3 ± 0.28	11.81 ± 0.13	23.64 ± 1.04	11.54 ± 0.27
No sec.-best norm.	24.74 ± 0.42	11.98 ± 0.03	24.42 ± 0.32	11.78 ± 0.11	22.28 ± 0.92	9.26 ± 1.17
No nov. bonus	24.83 ± 0.32	11.99 ± 0.03	23.75 ± 0.71	11.81 ± 0.09	20.24 ± 1.46	11.58 ± 0.29
No relab.	24.72 ± 0.39	12.0 ± 0.02	16.58 ± 4.62	3.62 ± 3.2	19.26 ± 1.0	11.53 ± 0.12
No forw. mod. relab.	24.9 ± 0.07	11.99 ± 0.02	22.73 ± 0.94	11.77 ± 0.28	11.5 ± 3.47	8.64 ± 1.83
No SAC relabelling	24.88 ± 0.09	11.98 ± 0.03	22.94 ± 1.51	11.76 ± 0.15	17.32 ± 2.54	11.05 ± 0.48
More skills	24.97 ± 0.03	11.99 ± 0.02	24.36 ± 0.36	11.8 ± 0.12	23.9 ± 0.47	11.47 ± 0.41

Table 1: Number of average unique game moves executed as skills by SEADS and its ablations after training on 5×10^5 (*Cursor*) / 1×10^7 (*Reacher*, *Jaco*) environment interactions. Best performance on each environment is marked by underline, worst in red. The ablated design choices contribute to the performance of SEADS especially on the more difficult *Reacher* and *Jaco* environments. Overestimating the number of skills does not decrease performance ("More skills"). We report mean and standard deviation on 10 independently trained agents. Ablations which perform significantly (one-sided Mann-Whitey U [4], $p < 0.01$) worse than SEADS are colored red. We did not find any ablation to perform significantly better than SEADS (including the "More skills" ablation). We refer to sec. S.6.2 for details.

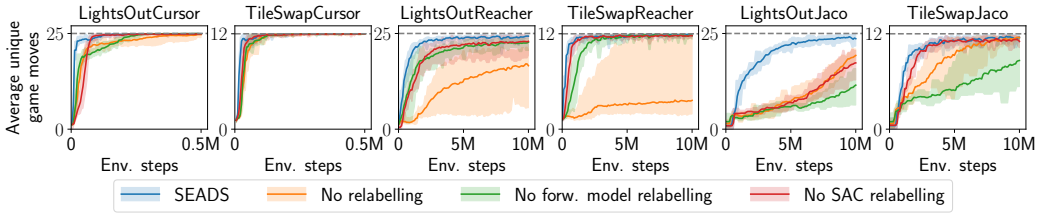


Figure S.15: Extended relabelling ablation analysis with 10 trained agents per variant. The results demonstrate that relabelling both for the forward model and SAC agent training are important for the performance of SEADS. See sec. S.6 for details.

S.7 Environment details

S.7.1 Cursor environments

At the beginning of an episode, the position of the cursor is reset to $x \sim \text{Uniform}(0, 1)$, $y \sim \text{Uniform}(0, 1)$. After a game move, the cursor is not reset. Both *LightsOut* and *TileSwap* board extents are $(x, y) \in [0, 1] \times [0, 1]$.

S.7.2 Reacher environments

At the beginning of an episode, the two joints of the Reacher are set to random angles $\theta \sim \text{Uniform}(0, 2\pi)$, $\phi \sim \text{Uniform}(0, 2\pi)$. After a game move, the joints are not reset. Both *LightsOut* and *TileSwap* board extents are $(x, y) \in [-0.15, 0.15] \times [-0.15, 0.15]$. The control simulation timestep is $0.02s$, and we use an action repeat of 2.

Algorithm 1 Task solution (without replanning). $\mathrm{bfs\_plan}$ denotes breadth-first search over a sequence of skills to transition $z_{(0)}$ to z^*, leveraging the symbolic forward model q_θ. Nodes are expanded in BFS via the function $\mathrm{successor}_{q_\theta} : \mathcal{Z} \times \mathcal{K} \to \mathcal{Z}$.

Input: environment E, skill-conditioned policy π, symbolic forward model q_θ, initial state $s_{(0)} = s_0$, symbolic goal z^*, symbolic mapping function Φ
Output: boolean success
$N, [k_1, ..., k_N] = \mathrm{bfs\_plan}(q_\theta, z_{(0)} = \Phi(s_{(0)}), z^*)$
for $n = 1$ **to** N **do**
  $s_{(n)} = \mathrm{apply}(E, \pi, s_{(n-1)}, k_i)$
end for
$\mathrm{success} = (\Phi(s_{(N)}) == z^*)$

S.7.3 Jaco environments

At the beginning of an episode and after a game move the Jaco arm is randomly reset above the board. The tool's (end effector) center point is randomly initialized to $x \sim \mathrm{Uniform}(-0.1, 0.1)$, $y \sim \mathrm{Uniform}(-0.1, 0.1)$, $z \sim \mathrm{Uniform}(0.2, 0.4)$ with random rotation $\theta \sim \mathrm{Uniform}(-\pi, \pi)$. The *LightsOut* board extent is $(x, y) \in [-0.25, 0.25] \times [-0.25, 0.25]$, the *TileSwap* board extent $(x, y) \in [-0.15, 0.15] \times [-0.15, 0.15]$. We use a control timestep of $0.1s$ in the simulation.

S.8 SEADS details

In the following, we will present algorithmic and architectural details of our approach.

S.8.1 Task solution with and without replanning

One main idea of the proposed SEADS agent is to use separate phases of symbolic planning (using the symbolic forward model $q_\theta(z_T \mid z_0, k)$) and low-level control (using the skill policies $\pi(a \mid s, k)$) for solving tasks. In algorithm 1 and algorithm 2 we present pseudocode of task solution using planning and skill execution, with and without intermittent replanning.

Algorithm 2 Task solution (with replanning). $\mathrm{bfs\_plan}$ denotes breadth-first search over a sequence of skills to transition $z_{(0)}$ to z^*, leveraging the symbolic forward model q_θ. Nodes are expanded in BFS via the function $\mathrm{successor}_{q_\theta} : \mathcal{Z} \times \mathcal{K} \to \mathcal{Z}$.

Input: environment E, skill-conditioned policy π, symbolic forward model q_θ, initial state $s_{(0)} = s_0$, symbolic goal z^*, symbolic mapping function Φ
$n \leftarrow 0$
repeat
  $N, [k_1, ..., k_N] \leftarrow \mathrm{bfs\_plan}(q_\theta, z_{(n)} = \Phi(s_{(n)}), z^*)$
  for $i \in \{1, \ldots, N\}$ **do**
    $n \leftarrow n + 1$
    $\hat{z}_{(n)} \leftarrow \mathrm{successor}_{q_\theta}(\Phi(s_{(n-1)}), k_i)$ {Compute the expected symbolic state after applying skill k_i}
    $s_{(n)} \leftarrow \mathrm{apply}(E, \pi, s_{(n-1)}, k_i)$
    $z_{(n)} \leftarrow \Phi(s_{(n)})$
    if $z_{(n)} \neq \hat{z}_{(n)}$ **then**
      Break {If the actual symbolic state differs from the predicted state, we replan}
    end if
  end for
until $z_{(n)} == z^*$

S.8.2 Training procedure

The main training loop of our proposed SEADS agent consists of intermittent episode collection and re-labelling for training the skill-conditioned policy π using soft actor-critic (SAC, [5]) and

symbolic forward model (see Algorithm 3). For episode collection we first sample a skill from a uniform distribution over skills $k \sim \text{Uniform}\{1, ..., K\}$ and then collect the corresponding episode. In our experiments we collect 32 episodes per epoch (i.e., $N_{\text{episodes}} = 32$) for all experiments except the real robot experiment with absolute push positions (sec. S.5.1), where we collect 4 episodes per epoch. We maintain two replay buffers of episodes for short-term ($\text{Episodes}_{\text{recent}}$) and long-term storage ($\text{Episodes}_{\text{buffer}}$), in which we keep the $N_{\text{buffer}} = 2048$ / $N_{\text{recent}} = 256$ most recent episodes. For training the SAC agent and skill model we combine a sample of 256 episodes from the long-term buffer and all 256 episodes from the short-term buffer, comprising the episodes Episodes. These episodes are subsequently passed to the relabelling module. On these relabelled episodes the SAC agent and skill model are trained. Please see the following subsections for details on episode collection, relabelling, skill model training and policy training.

Algorithm 3 SEADS training loop

Input: Environment E
Number of epochs N_{epochs}
Number of new episodes per epoch N_{episodes}
Episode buffer size N_{buffer}
Result: Trained skill-conditioned policy $\pi(a \mid s, k)$ and forward model $q_\theta(z_T \mid z_0, k)$
$\text{Episodes}_{\text{buffer}} = []$, $\text{Episodes}_{\text{recent}} = []$
for $n_{\text{epoch}} = 1$ **to** N_{epochs} **do**
  for $n_{\text{episode}} = 1$ **to** N_{episodes} **do**
    Sample $k \sim p(k)$
    $s_0 = E.\text{reset}()$
    $\text{Ep} = \text{collect\_episode}(E, \pi, s_0, k)$
    $\text{Episodes}_{\text{buffer}}.\text{append}(\text{Ep})$, $\text{Episodes}_{\text{recent}}.\text{append}(\text{Ep})$
  end for
  $\text{Episodes}_{\text{buffer}} \leftarrow \text{Episodes}_{\text{buffer}}[\text{-}N_{\text{buffer}}:]$ {Keep N_{buffer} most recent episodes}
  $\text{Episodes}_{\text{recent}} \leftarrow \text{Episodes}_{\text{recent}}[\text{-}N_{\text{recent}}:]$
  $\text{Episodes} = \text{sample}(\text{Episodes}_{\text{buffer}}, N = 256) \cup \text{Episodes}_{\text{recent}}$
  $\text{Episodes}_{\text{SM}} \leftarrow \text{relabel}(\text{Episodes}, p = 1.0)$
  $\text{update\_skill\_model}(\text{Episodes}_{\text{SM}})$
  $\text{Episodes} = \text{sample}(\text{Episodes}_{\text{buffer}}, N = 256) \cup \text{Episodes}_{\text{recent}}$
  $\text{Episodes}_{\text{SAC}} \leftarrow \text{relabel}(\text{Episodes}, p = 0.5)$
  $\text{update\_sac}(\text{Episodes}_{\text{SAC}})$
end for

S.8.2.1 Episode collection (collect_episode)

The operator collect_episode works similar to the apply operator defined in the main paper (see sec. 3). It applies the skill policy $\pi(a_t \mid s_t, k)$ iteratively until termination. However, the operator returns all intermediate states $s_0, ..., s_T$ and actions $a_0, ..., a_{T-1}$ to be stored in the episode replay buffers.

S.8.2.2 Relabelling (relabel)

For relabelling we sample a Bernoulli variable with success probability of p for each episode in the Episodes buffer, indicating whether it may be relabeled. For training the forward model we relabel all episodes ($p = 1$), while for the SAC agent we only allow half of the episodes to be relabeled ($p = 0.5$). The idea is to train the SAC agent also on *negative* examples of skill executions with small rewards. Episodes in which the symbolic observation did not change are excluded from relabelling for the SAC agent, as for those the agent receives a constant negative reward. All episodes which should be relabeled are passed to the relabelling module as described in sec. 3. The union of these relabelled episodes and the episodes which were not relabeled form the updated buffer which is returned by the relabel operator.

S.8.2.3 SAC agent update (update_sac)

In each epoch, we fill a transition buffer using all transitions from the episodes in the $\text{Episodes}_{\text{SAC}}$ buffer. Each transition tuple is of the form $([s_t^i, k^i], a_t^i, [s_{t+1}^i, k^i], r_{t+1}^i)$ where s are environment

observations, a low-level actions, k a one-hot representation of the skill and r the intrinsic reward. $[\cdot]$ denotes the concatenation operation. The low-level SAC agent is trained for 16 steps per epoch on batches comprising 128 randomly sampled transitions from the transition buffer. We train the actor and critic networks using Adam [6]. For architectural details of the SAC agent, see sec. S.8.3.1.

S.8.2.4 Skill model update (update_skill_model)

From the episode buffer $\text{Episodes}_{\text{SM}}$ we sample transition tuples $(z_0^i, k_i, z_{T^i}^i)$. The skill model is trained to minimize an expected loss

$$\mathcal{L} = \mathbb{E}_{\mathcal{B}} \left[\sum_{i \in \mathcal{B}} L(z_0^i, k_i, z_{T^i}^i) \right] \tag{1}$$

for randomly sampled batches $\mathcal{B}$ of transition tuples. We optimize the skill model parameters θ using the Adam [6] optimizer for 4 steps per epoch on batches of size 32. We use a learning rate of 10^{-3}. The instance-wise loss L to be minimized corresponds to the negative log-likelihood $L = -\log q_\theta(z_{T^i}^i \mid z_0^i, k)$ for symbolic forward models or $L = -\log q_\theta(k \mid z_0^i, z_{T^i}^i)$ for the VIC ablation (see sec. S.11).

S.8.3 Architectural details

S.8.3.1 SAC agent

We use an open-source soft actor-critic implementation [7] in our SEADS agent. Policy and critic networks are modeled by a multilayer perceptron with two hidden layers with ReLU activations. For hyperparameters, please see the table below.

{*LightsOut*, *TileSwap* }-	*Cursor*	*Reacher*	*Jaco*	Robot (LightsOut)
Learning rate	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Target smoothing coeff. τ	0.005	0.005	0.005	0.005
Discount factor γ	0.99	0.99	0.99	0.99
Hidden dim.	512	512	512	512
Entropy target α	0.1	0.01	0.01	0.1
Automatic entropy tuning	no	no	no	no
Distribution over actions	Gaussian	Gaussian	Gaussian	Gaussian

S.8.3.2 Symbolic forward model

Our symbolic forward model models the distribution over the terminal symbolic observation z_T given the initial symbolic observation z_0 and skill k. It factorizes over the symbolic observation as $q_\theta(z_T \mid k, z_0) = \prod_{d=1}^{D} q_\theta([z_T]_i \mid k, z_0) = \prod_{d=1}^{D} \text{Bernoulli}([\alpha_T(z_0, k)]_i)$ where D is the dimensionality of the symbolic observation. We assume $z \in \mathcal{Z}$ to be a binary vector with $\mathcal{Z} = \{0, 1\}^D$. The Bernoulli probabilities $\alpha_T(z_0, k)$ are predicted by a learnable neural component. We use a neural network f to parameterize the probability of the binary state in z_0 to flip $p_{\text{flip}} = f_\theta(z_0, k)$, which simplifies learning if the *change* in symbolic state only depends on k and is independent of the current state. Let α_T be the probability for the binary state to be `True`, then $\alpha_T = (1 - z_0) \cdot p_{\text{flip}} + z_0 \cdot (1 - p_{\text{flip}})$. The input to the neural network is the concatenation $[z_0, \text{onehot}(k)]$. We use a multilayer perceptron with two hidden layers with ReLU nonlinearities and 256 hidden units.

S.9 SAC baseline

We train the SAC baseline in a task-specific way by giving a reward of 1 to the agent if the board state has reached its target configuration and 0 otherwise. At the beginning of each episode we first sample the difficulty of the current episode which corresponds to the number of moves required to solve the game (solution depth S). For all environments S is uniformly sampled from $\{1, ..., 5\}$. For all *Cursor* environments we impose a step limit $T_{\text{lim}} = 10 \cdot S$, for *Reacher* and *Jaco* $T_{\text{lim}} = 50 \cdot S$. This corresponds to the number of steps a single skill can make in SEADS multiplied by S. We use a replay buffer which holds the most recent 1 million transitions and train the agent with a batchsize of

256. The remaining hyperparameters (see table below) are identical to the SAC component in SEADS; except for an increased number of hidden units and an additional hidden layer (i.e., three hidden layers) in the actor and critc networks to account for the planning the policy has to perform. In each epoch of training we collect 8 samples from each environment which we store in the replay buffer. We performed a hyperparameter search on the number of agent updates performed in each epoch N and entropy target values α. We also experimented with skipping updates, i.e., collecting 16 (for $N = 0.5$) or 32 (for $N = 0.25$) environment samples before performing a single update. We found that performing too many updates leads to unstable training (e.g., $N = 4$ for `LightsOutCursor`). For all results and optimal settings per environment, we refer to sec. S.13.1. For the SAC baseline we use the same SAC implementation from [7] which we use for SEADS.

{*LightsOut*, *TileSwap* }-	*Cursor*	*Reacher*	*Jaco*
Learning rate	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Target smoothing coeff. τ	0.005	0.005	0.005
Discount factor γ	0.99	0.99	0.99
Hidden dim.	512	512	512
Entropy target α	*tuned* (see sec. S.13.1)		
Automatic entropy tuning	no	no	no
Distribution over actions	Gaussian	Gaussian	Gaussian

S.10 HAC baseline

For the HAC baseline we adapt the official code release [8]. We modify the architecture to allow for a two-layer hierarchy of policies in which the higher-level policy commands the lower-level policy with *discrete* subgoals (which correspond to the symbolic observations z in our case). This requires the higher-level policy to act on a discrete action space $\mathcal{A}_{\text{high}} = \mathcal{Z}$. The lower-level policy acts on the continuous actions space $\mathcal{A}_{\text{low}} = \mathcal{A}$ of the respective manipulator (*Cursor*, *Reacher*, *Jaco*). To this end, we use a discrete-action SAC agent for the higher-level policy and a continuous-action SAC agent for the lower-level policy. For the higher-level discrete SAC agent we parameterize the distribution over actions as a factorized reparametrizable RelaxedBernoulli distribution, which is a special case of the Concrete [9] / Gumbel-Softmax [10] distribution. We use an open-source SAC implementation [7] for the SAC agent on both levels and extend it by a RelaxedBernoulli distribution over actions for the higher-level policy.

S.10.1 Hyperparameter search

We performed an extensive hyperparameter search on all 6 environments ([*LightsOut*, *TileSwap*] $\times$ [*Cursor*, *Reacher*, *Jaco*]) for the HAC baseline. We investigated a base set of entropy target values $\alpha_{\text{low}}, \alpha_{\text{high}} \in \{0.1, 0.01, 0.001, 0.0001\}$ for both layers separately. On the *Cursor* environments we refined these sets in regions of high success rates. We performed a hyperparameter search on the temperature parameter τ of the RelaxedBernoulli distribution on the *Cursor* environments with $\tau \in \{0.01, 0.05, 0.1, 0.5\}$ and found $\tau = 0.1$ to yield the best results. For experiments on the *Reacher* and *Jaco* environments we then fixed the parameter $\tau = 0.1$. We report results on parameter sets with highest average success rate on 5 individually trained agents after 5×10^5 (*Cursor*) / 1×10^7 (*Reacher*, *Jaco*) environment interactions. We refer to sec. S.13.2 for a visualization of all hyperparameter search results.

Parameters for high-level policy

	all environments
Learning rate	$3 \cdot 10^{-4}$
Target smoothing coeff. τ	0.005
Discount factor γ	0.99
Hidden layers for actor/critic	2
Hidden dim.	512
Entropy target α_{high}	*tuned* (see sec. S.13.2)
Automatic entropy tuning	no
Distribution over actions	RelaxedBernoulli
RelaxedBernoulli temperature τ	*tuned* (see sec. S.13.2)

Parameters for low-level policy

	all environments
Learning rate	$3 \cdot 10^{-4}$
Target smoothing coeff. τ	0.005
Discount factor γ	0.99
Hidden layers for actor/critic	2
Hidden dim.	512
Entropy target α_{high}	*tuned* (see sec. S.13.2)
Automatic entropy tuning	no
Distribution over actions	Gaussian

S.11 VIC baseline

We compare to Variational Intrinsic Control (VIC, Gregor et al. [1]) as a baseline method of unsupervised skill discovery. It is conceptually similar to our method as it aims to find skills such that the mutual information $\mathcal{I}(s_T, k \mid s_0)$ between the skill termination state s_T and skill k is maximized given the skill initiation state s_0. To this end it jointly learns a skill policy $\pi(s_t \mid a_t, k)$ and *skill discriminator* $q_\theta(k \mid s_0, s_T)$. We adopt this idea and pose a baseline to our approach in which we model $q_\theta(k \mid z_0, z_T)$ *directly* with a neural network, instead of modelling $q_\theta(k \mid z_0, z_T)$ indirectly through a forward model $q_\theta(z_T \mid z_0, k)$. The rest of the training process including its hyperparameters is identical to `SEADS`. We implement $q_\theta(k \mid z_0, z_T)$ by a neural network which outputs the parameters of a categorical distribution and is trained by maximizing the log-likelihood $\log q_\theta(k \mid z_0^i, z_{T^i}^i)$ on transition tuples $(z_0^i, k_i, z_{T^i}^i)$ (see sec. S.8.2.4). We experimented with different variants of passing $(z_0^i, z_{T^i}^i)$ to the network: (i) concatenation $[z_0^i, z_{T^i}^i]$ and (ii) concatenation with XOR $[z_0^i, z_{T^i}^i, z_0^i \text{ XOR } z_{T^i}^i]$. We only found the latter to show success during training. The neural network model contains two hidden layers of size 256 with ReLU activations (similar to the forward model). We also evaluate variants of VIC which are extended by our proposed *relabelling scheme* and *second-best reward normalization*. In contrast to VIC, our `SEADS` agent discovers all possible game moves reliably in both `LightsOutCursor` and `TileSwapCursor` environments, see Fig. S.16 for details. Our proposed second-best normalization scheme (*+SBN*, sec. 3) slightly improves performance of VIC in terms of convergence speed (`LightsOutCursor`) and variance in number of skills detected (`TileSwapCursor`). The proposed relabelling scheme (*+RL*, sec. 3) does not improve (`LightsOutCursor`) or degrades (`TileSwapCursor`) the number of detected skills.

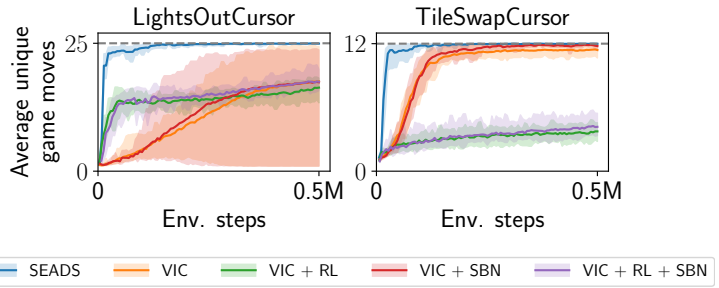


Figure S.16: Number of discovered skills on the `LightsOutCursor`, `TileSwapCursor` environments for the `SEADS` agent and variants of VIC [1]. Only `SEADS` discovers all skills reliably on both environments. See sec. S.11 for details.

	solution depth							
	1	2	3	4	5	6	7	8
LightsOut	25	300	2300	12650	53130	176176	467104	982335
TileSwap	12	88	470	1978	6658	18081	38936	65246
	solution depth							
	9	10	11	12	13	14	15	16
LightsOut	1596279	1935294	1684446	1004934	383670	82614	7350	0
TileSwap	83000	76688	48316	18975	4024	382	24	1

Table 2: Number of feasible board configurations for varying solution depths. No feasible board configurations exist with solution depth > 16.

S.12 Environment details

S.12.1 Train-/Test-split

In order to ensure disjointness of board configurations in train and test split we label each board configuration based on a hash remainder. For the hashing algorithm we first represent the current board configuration as comma-separated string, e.g. $s = "1, 1, 0, ..., 0"$ for *LightsOut* and $s = "1, 0, 2, ..., 8"$ for *TileSwap*. Then, this string is passed through a CRC32 hashing function, yielding the split based on an integer division remainder

$$\text{split} = \begin{cases} \text{train} & \text{CRC32}(s) \mod 3 = 0 \\ \text{test} & \text{CRC32}(s) \mod 3 \in \{1, 2\} \end{cases} \tag{2}$$

S.12.2 Board initialization

We quantify the difficulty of a particular board configuration by its *solution depth*, i.e., the minimal number of game moves required to solve the board.

We employ a breadth-first search (BFS) beginning from the goal board configuration (all fields *off* in *LightsOut*, ordered fields in *TileSwap*). Nodes (board configurations) are expanded through applying feasible actions (12 for *TileSwap*, 25 for *LightsOut*). Once a new board configuration is observed for the first time, its solution depth corresponds to the current BFS step.

By this, we find all feasible board configurations for *LightsOut* and *TileSwap* and their corresponding solution depths (see table 2). In table 3 we show the sizes of the training and test split for *LightsOut* and *TileSwap* environments for solution depths in $\{1, ..., 5\}$.

		solution depth				
		1	2	3	4	5
LightsOut	train	7	99	785	4200	17849
	test	18	201	1515	8450	35281
	total	25	300	2300	12650	53130
TileSwap	train	7	31	179	683	2237
	test	5	57	291	1295	4421
	total	12	88	470	1978	6658

Table 3: Number of initial board configurations for varying solution depths and dataset splits (train / test).

S.13 Results of hyperparameter search on HAC and SAC baselines

S.13.1 Results of SAC hyperparameter search

Please see figures S.17, S.18, S.19, S.20, S.21, S.22 for a visualization of the SAC hyperparameter search results.

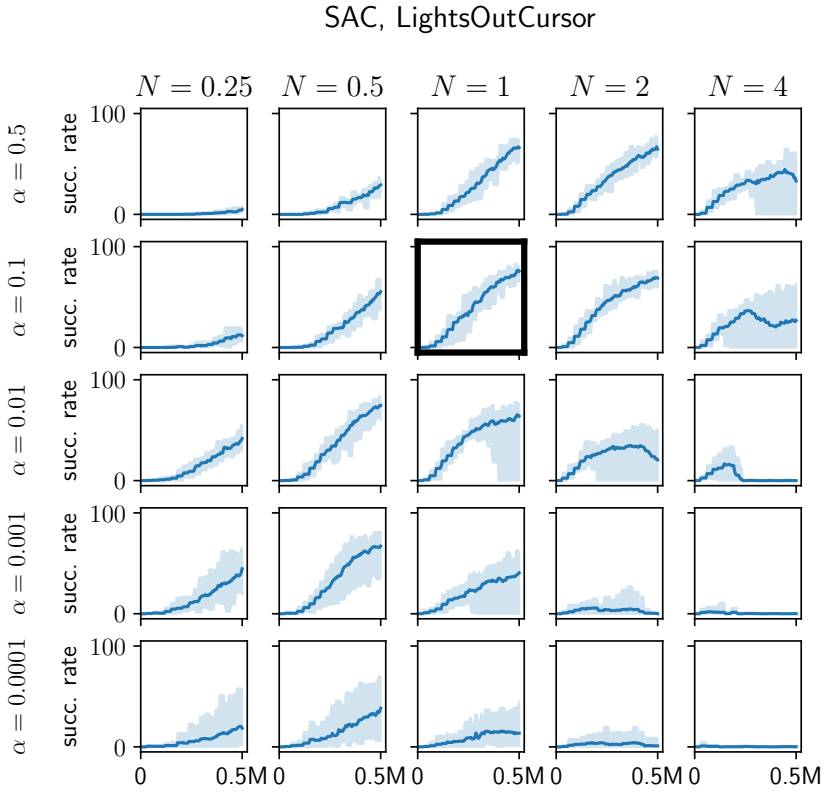


Figure S.17: Test performance of SAC agents on the `LightsOutCursor` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 1$, $\alpha = 0.1$).

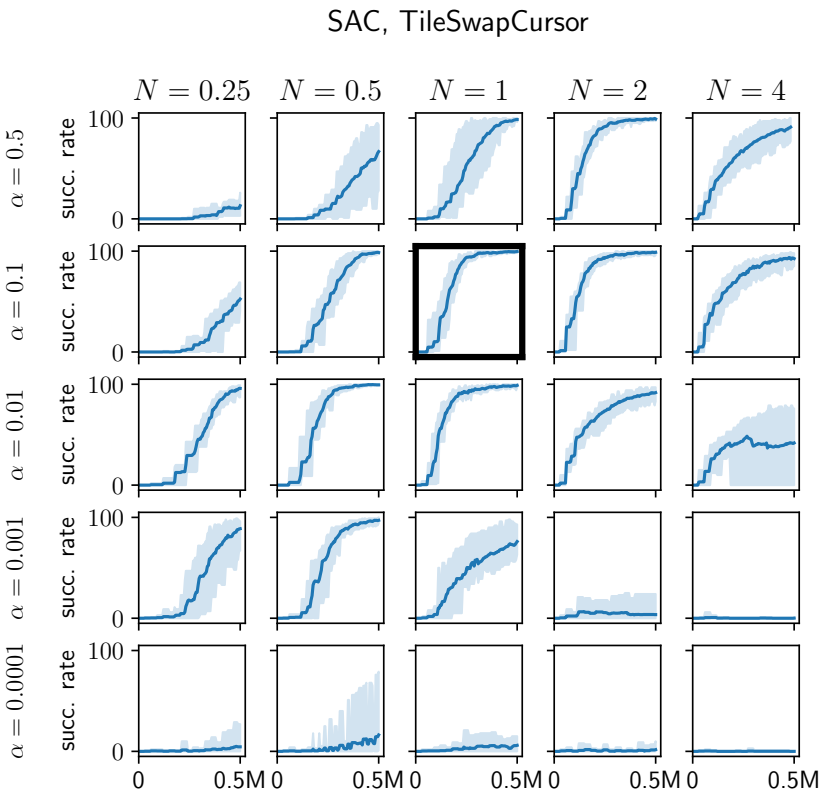


Figure S.18: Test performance of SAC agents on the `TileSwapCursor` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 1$, $\alpha = 0.1$).

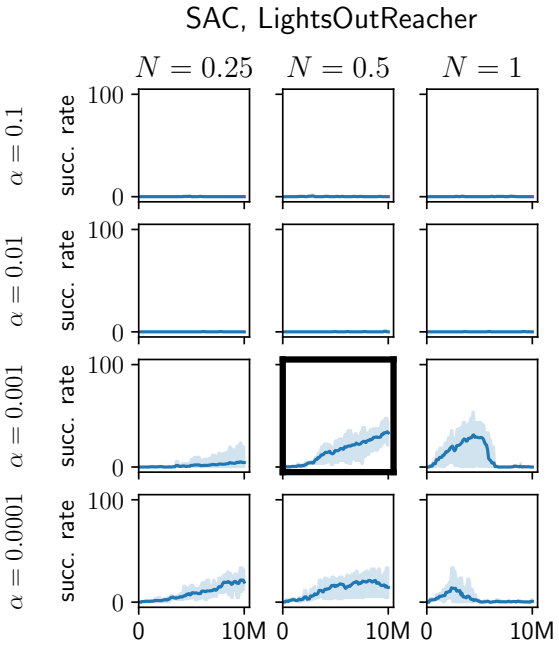


Figure S.19: Test performance of SAC agents on the `LightsOutReacher` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 0.5$, $\alpha = 0.001$).

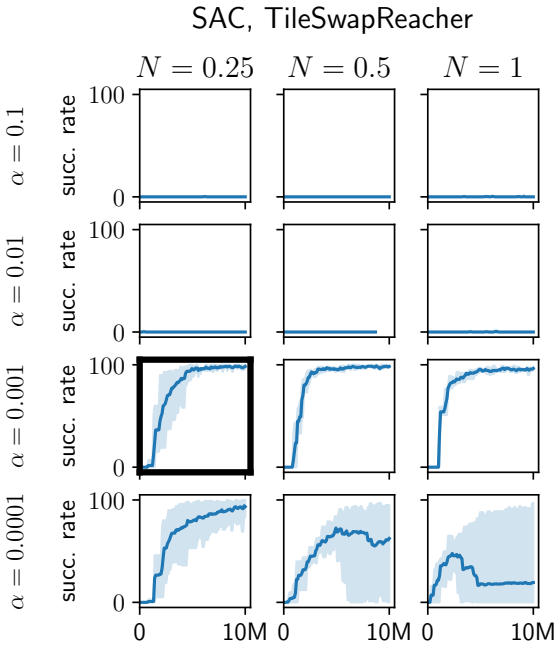


Figure S.20: Test performance of SAC agents on the `TileSwapReacher` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 0.25$, $\alpha = 0.001$).

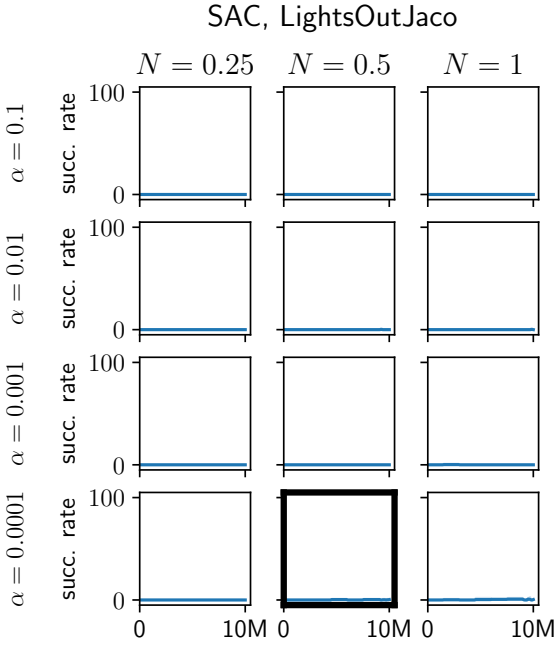


Figure S.21: Test performance of SAC agents on the `LightsOutJaco` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 0.5$, $\alpha = 0.0001$).

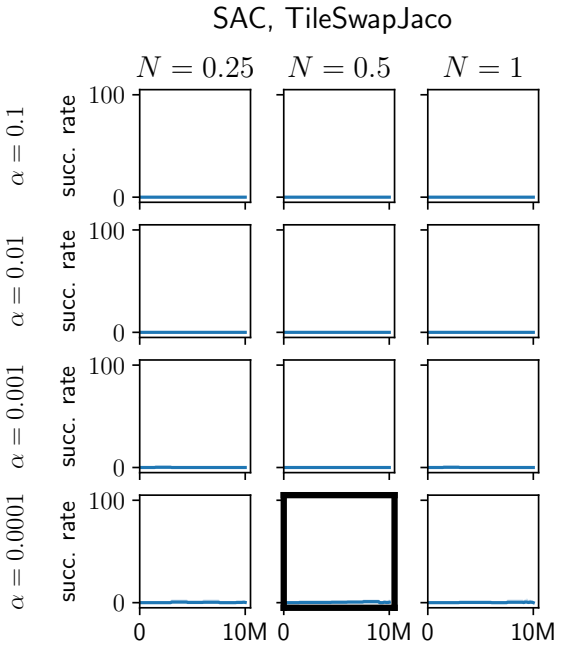
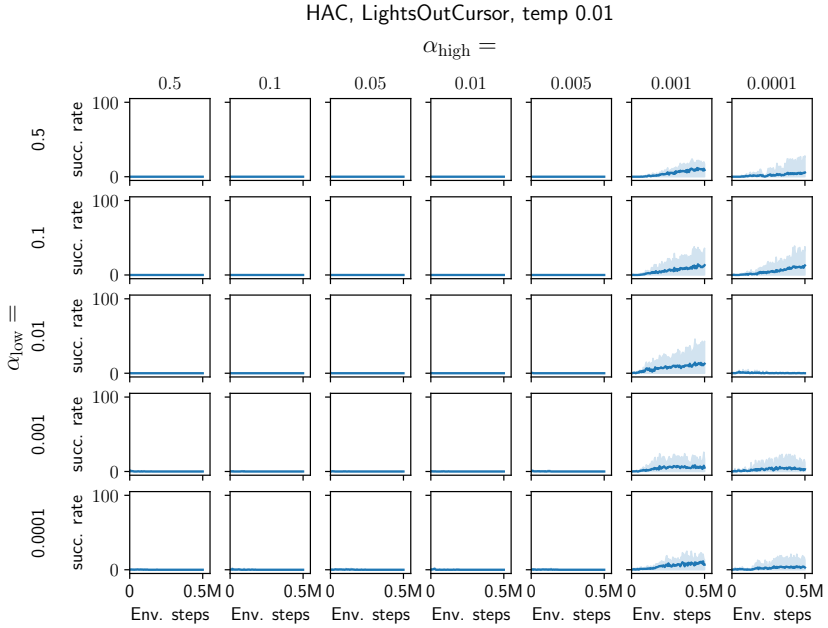
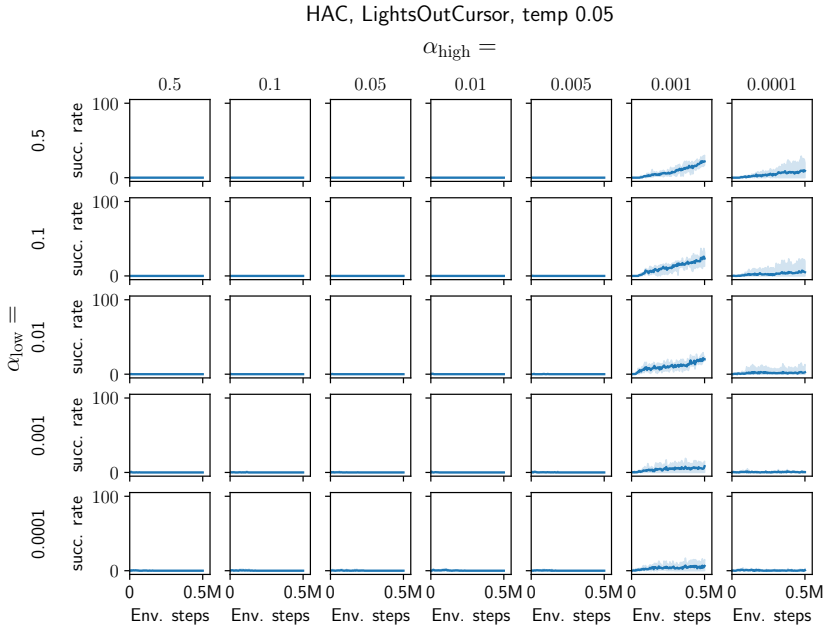


Figure S.22: Test performance of SAC agents on the `TileSwapJaco` environment for varying number of update steps per epoch (N) and parameters α. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($N = 0.5$, $\alpha = 0.0001$).

S.13.2 Results of HAC hyperparameter search

Please see figures S.22, S.22, S.23, S.24, S.25, S.26 for a visualization of the HAC hyperparameter search results.





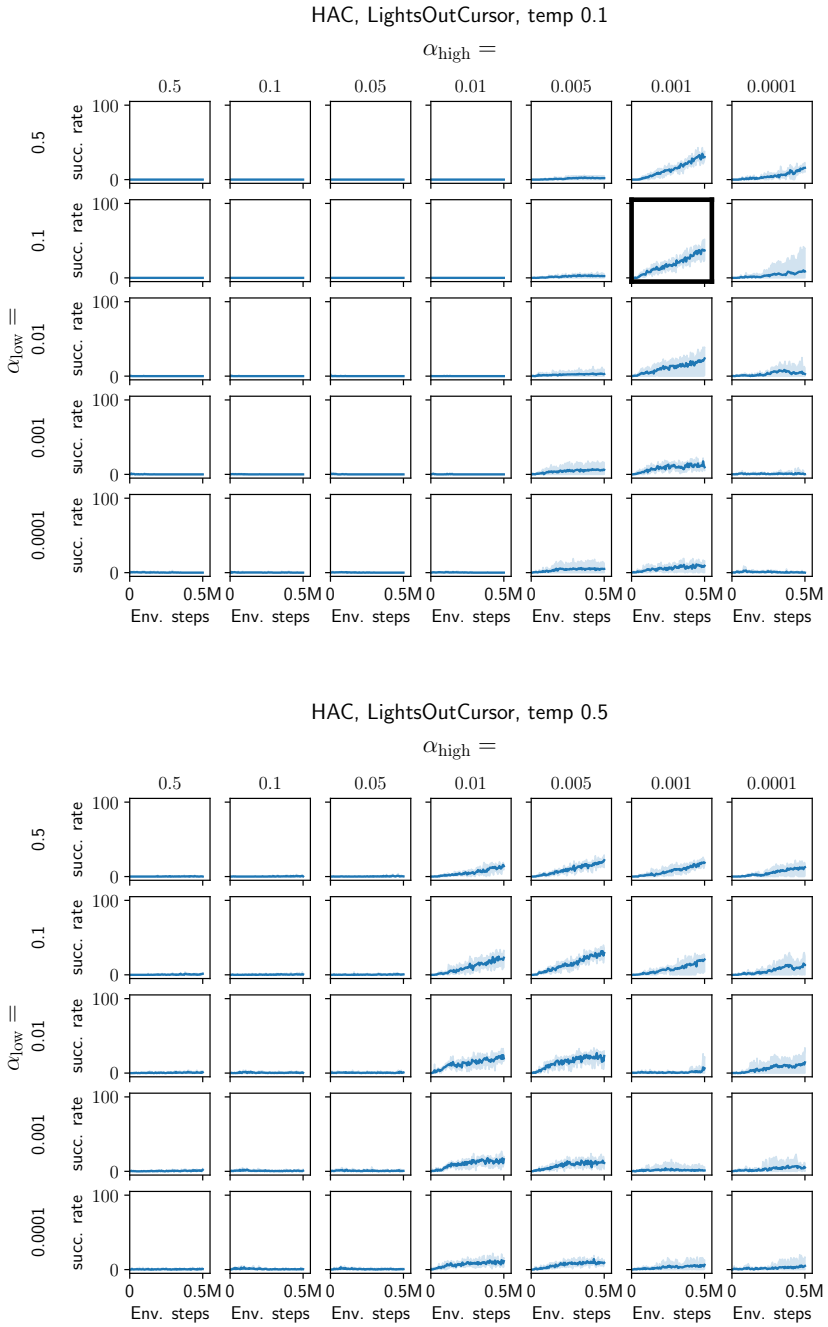
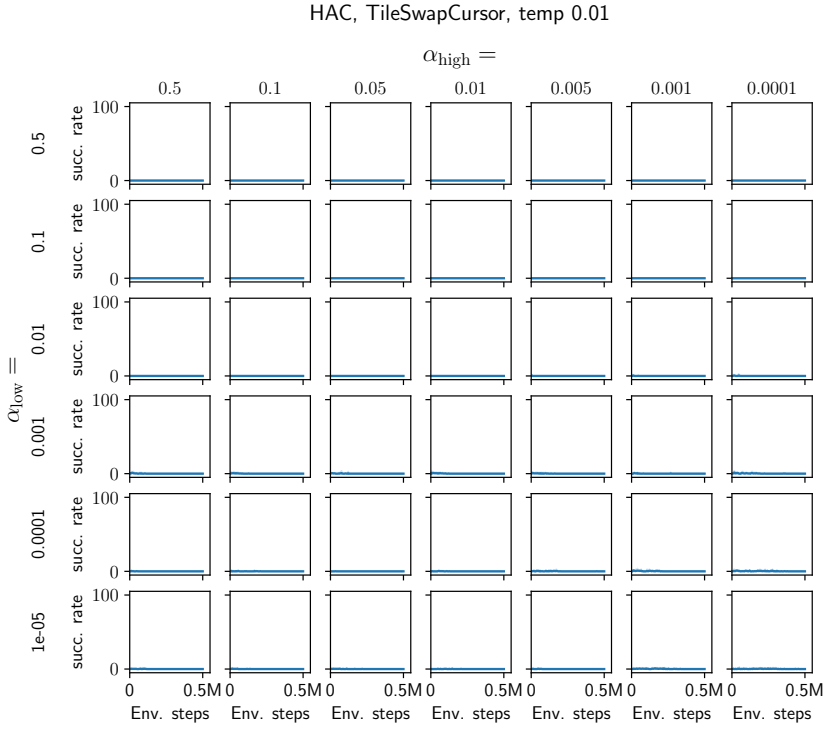
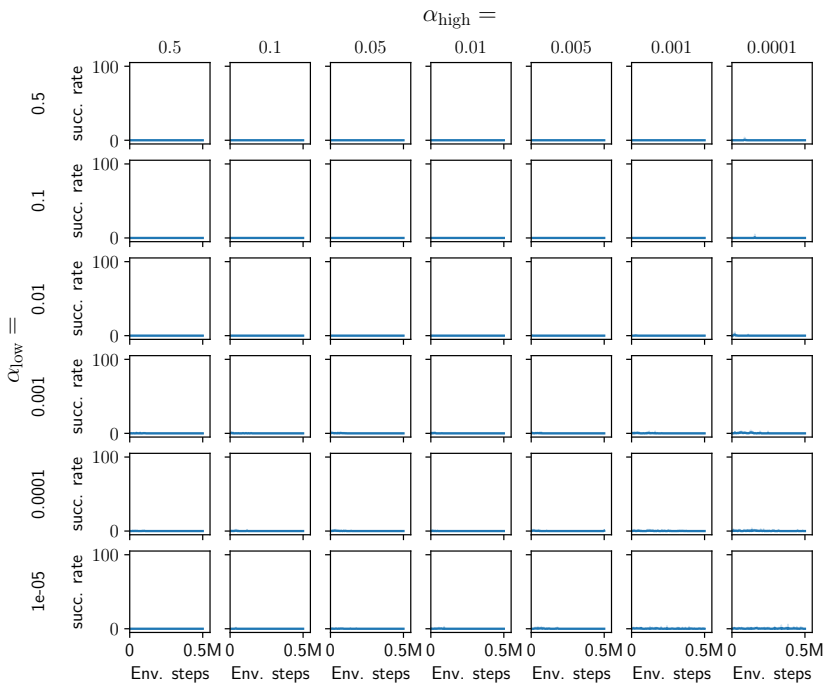


Figure S.22: Test performance of HAC agents on the `LightsOutCursor` environment for varying values for RelaxedBernoulli temperature τ and entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.1$, $\alpha_{\text{low}} = 0.1$, $\alpha_{\text{high}} = 0.001$).


HAC, TileSwapCursor, temp 0.01
$\alpha_{\text{high}} =$
0.5
0.1
0.05
0.01
0.005
0.001
0.0001
$\alpha_{\text{low}} =$
succ. rate
100
0
0
0.5M
Env. steps


HAC, TileSwapCursor, temp 0.05
$\alpha_{\text{high}} =$
0.5
0.1
0.05
0.01
0.005
0.001
0.0001
$\alpha_{\text{low}} =$
succ. rate
100
0
0
0.5M
Env. steps

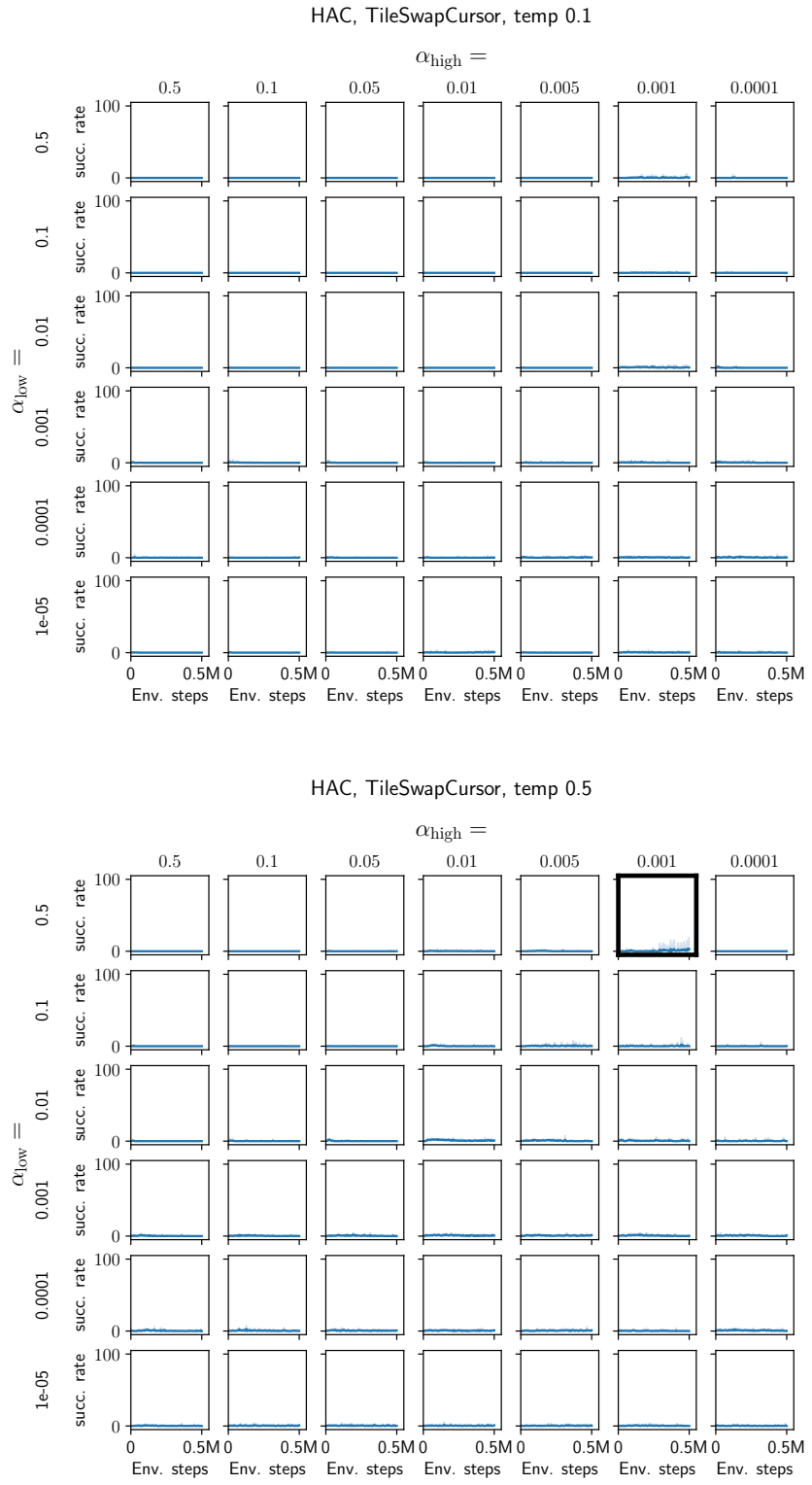


Figure S.22: Test performance of HAC agents on the `TileSwapCursor` environment for varying values for RelaxedBernoulli temperature τ and entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.5$, $\alpha_{\text{low}} = 0.5$, $\alpha_{\text{high}} = 0.001$).

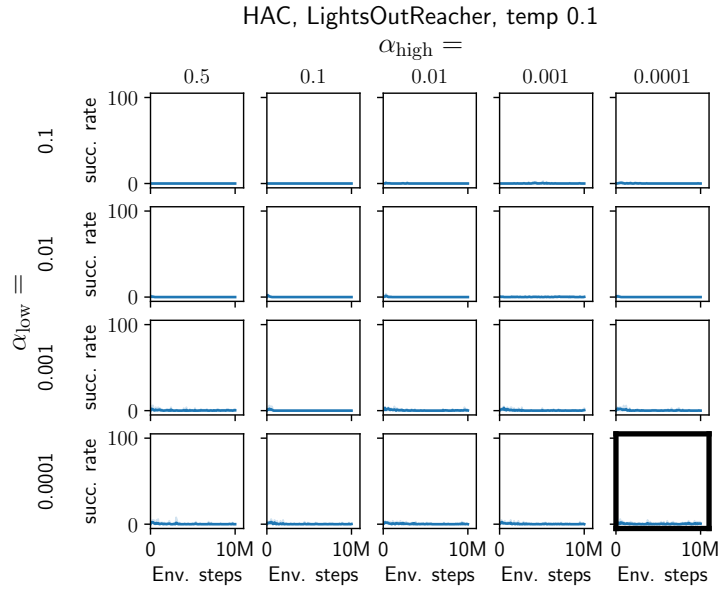


Figure S.23: Test performance of HAC agents on the `LightsOutReacher` environment for varying values for the entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$ and fixed RelaxedBernoulli temperature $\tau = 0.1$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.1$, $\alpha_{\text{low}} = 0.0001$, $\alpha_{\text{high}} = 0.0001$).

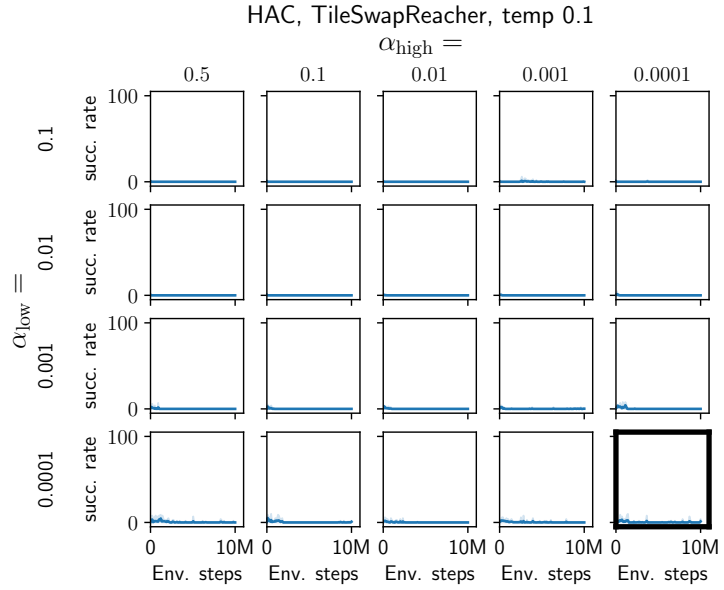


Figure S.24: Test performance of HAC agents on the `TileSwapReacher` environment for varying values for the entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$ and fixed RelaxedBernoulli temperature $\tau = 0.1$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.1$, $\alpha_{\text{low}} = 0.0001$, $\alpha_{\text{high}} = 0.0001$).

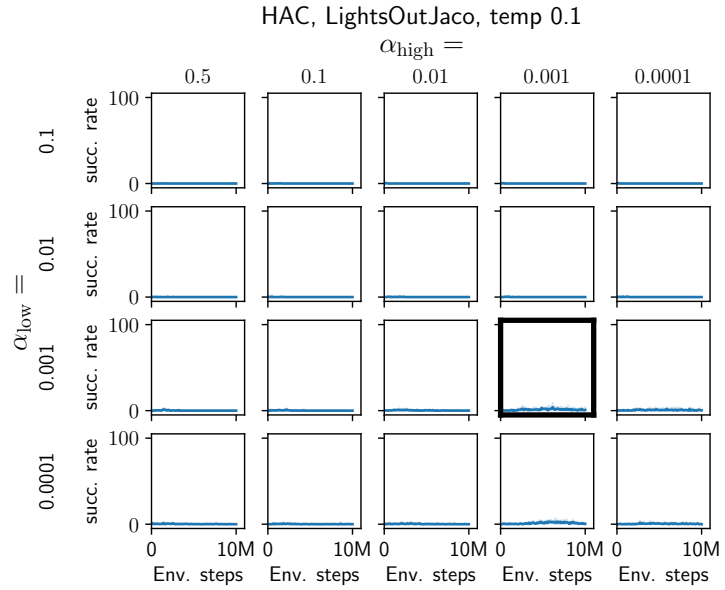


Figure S.25: Test performance of HAC agents on the `LightsOutJaco` environment for varying values for the entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$ and fixed RelaxedBernoulli temperature $\tau = 0.1$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.1$, $\alpha_{\text{low}} = 0.001$, $\alpha_{\text{high}} = 0.001$).

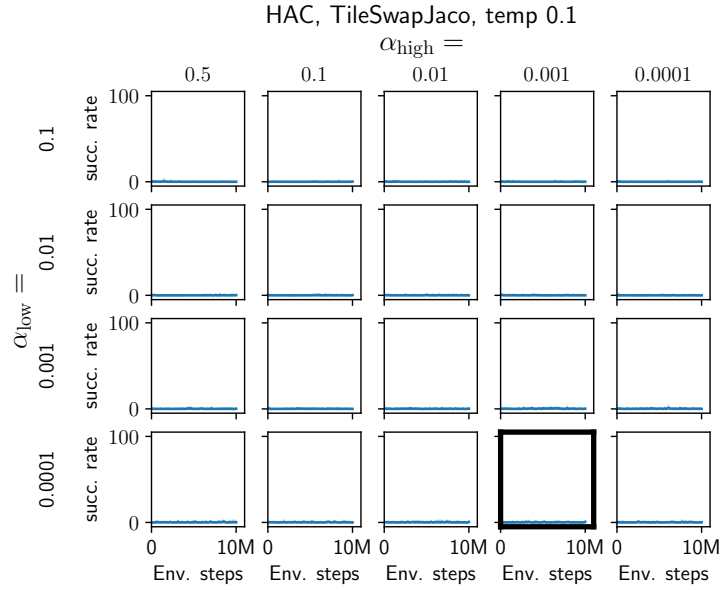


Figure S.26: Test performance of HAC agents on the `TileSwapJaco` environment for varying values for the entropy targets $\alpha_{\text{high}}, \alpha_{\text{low}}$ and fixed RelaxedBernoulli temperature $\tau = 0.1$. We evaluate 5 individual agents per configuration. The best configuration is marked in **bold** ($\tau = 0.1$, $\alpha_{\text{low}} = 0.0001$, $\alpha_{\text{high}} = 0.001$).

References

[1] K. Gregor, D. J. Rezende, and D. Wierstra. Variational intrinsic control. In *International Conference on Learning Representations (ICLR), Workshop Track Proceedings*, 2017. URL https://openreview.net/forum?id=Skc-Fo4Yg.

[2] uArm Developer. uarm-python-sdk. https://github.com/uArm-Developer/uArm-Python-SDK, 2021.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[4] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947. ISSN 00034851. URL http://www.jstor.org/stable/2236101.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018. URL http://proceedings.mlr.press/v80/haarnoja18b.html.

[6] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[7] P. Tandon. pytorch-soft-actor-critic. https://github.com/pranz24/pytorch-soft-actor-critic, 2021.

[8] A. Levy. Hierarchical-actor-critc-hac-. https://github.com/andrew-j-levy/Hierarchical-Actor-Critc-HAC-, 2020.

[9] C. J. Maddison, A. Mnih, and Y. W. Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations (ICLR)*, 2017. URL https://openreview.net/forum?id=S1jE5L5gl.

[10] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations (ICLR)*, 2017. URL https://openreview.net/forum?id=rkE3y85ee.