First return, then explore

Adrien Ecoffet[*,1,2], Joost Huizinga[*,1,2], Joel Lehman[1,2], Kenneth O. Stanley[1,2] & Jeff Clune[1,2]

[1] *Uber AI Labs, San Francisco, CA, USA*

[2] *OpenAI, San Francisco, CA, USA*

[*] *These authors contributed equally to this work*

Correspondence should be addressed to Adrien Ecoffet (email: adrienecoffet@gmail.com), Joost Huizinga (email: joost.hui@gmail.com), and Jeff Clune (email: jclune@gmail.com).

The promise of reinforcement learning is to solve complex sequential decision problems autonomously by specifying a high-level reward function only. However, reinforcement learning algorithms struggle when, as is often the case, simple and intuitive rewards provide sparse[1] and deceptive[2] feedback. Avoiding these pitfalls requires thoroughly exploring the environment, but creating algorithms that can do so remains one of the central challenges of the field. We hypothesise that the main impediment to effective exploration originates from algorithms forgetting how to reach previously visited states ("detachment") and from failing to first return to a state before exploring from it ("derailment"). We introduce Go-Explore, a family of algorithms that addresses these two challenges directly through the simple principles of explicitly remembering promising states and first returning to such states before intentionally exploring. Go-Explore solves all heretofore unsolved Atari games and surpasses the state of the art on all hard-exploration games[1], with orders of magnitude improvements on the grand challenges *Montezuma's Revenge* and *Pitfall*. We also demonstrate the practical potential of Go-Explore on a sparse-reward pick-and-place robotics task. Additionally, we show that adding a goal-conditioned policy can further improve Go-Explore's exploration efficiency and enable it to handle stochasticity throughout training. The substantial performance gains from Go-Explore suggest that the simple principles of remembering states, returning to them, and exploring from them are a powerful and general approach to exploration, an insight that may prove critical to the creation of truly intelligent learning agents.

Recent years have yielded impressive achievements in Reinforcement Learning (RL), including world-champion level performance in Go[3], Starcraft II[4], and Dota II[5], as well as autonomous learning of robotic skills such as running, jumping, and grasping[6,7]. Many of these successes were enabled by carefully-designed, highly-informative reward functions. However, for many practical problems, defining a good reward function is non-trivial; to guide a robot to a refrigerator, one might provide a reward only when the refrigerator is reached, but doing so makes the reward *sparse* if many actions are required to reach the refrigerator. Unfortunately, a denser reward (e.g. the Euclidean distance to the refrigerator), can be *deceptive*; naively following the reward function may lead the robot into a dead end and can also produce unintended (and potentially unsafe) behaviour (e.g. the robot not detouring around obstacles like pets)[8–10].

These challenges motivate designing RL algorithms that better handle sparsity and deception. A key observation is that sufficient *exploration* of the state space enables discovering sparse rewards and avoiding deceptive local optima[11,12]. We argue that two major issues have hindered the ability of previous algorithms to explore. The first is *detachment*, wherein the algorithm prematurely stops returning to certain areas of the state space despite having evidence that those areas are promising (SI "Detachment"). Detachment is especially likely when (as is common) there are multiple areas to explore because the algorithm may partially explore one area, switch to a second area, and forget how to visit the first area. The second is *derailment*, wherein the exploratory mechanisms of the algorithm prevent it from returning to previously visited states, preventing exploration directly and/or forcing practitioners to make exploratory mechanisms so minimal that effective exploration does not occur (SI "Derailment"). For example, if a long string of correct actions is required to reach a particular area, a high probability of exploratory actions prevents the area from being reached while a low probability of exploratory actions results in little exploration in general. We present Go-Explore, a family of algorithms designed to explicitly avoid detachment and derailment, and demonstrate that it thoroughly explores environments. Go-Explore surpasses human performance on (solves) all previously unsolved Atari games*, which has been posited as a major milestone in previous work[13,15,16]. It also surpasses the state of the art on all hard-exploration Atari games (i.e. where obtaining rewards requires long sequences of correct actions, meaning randomly sampling actions rarely produces rewards and thus more intelligent *exploration* is needed). Additionally,

---

*Concurrent work[13] similarly reached this milestone (SI "Comparing Go-Explore and Agent57"), but under easier, mostly deterministic conditions that do not meet community-defined standards[14] for evaluation on Atari. Our descriptions of prior results include only evaluations meeting these standards, unless explicitly mentioned (Methods "State of the art on Atari").

we demonstrate that it can solve a practical simulated robotics problem with an extremely sparse reward. Finally, we show that its performance can be greatly increased by incorporating minimal domain knowledge and examine how harnessing learned skills during exploration can improve exploration efficiency, highlighting the versatility of the Go-Explore family.
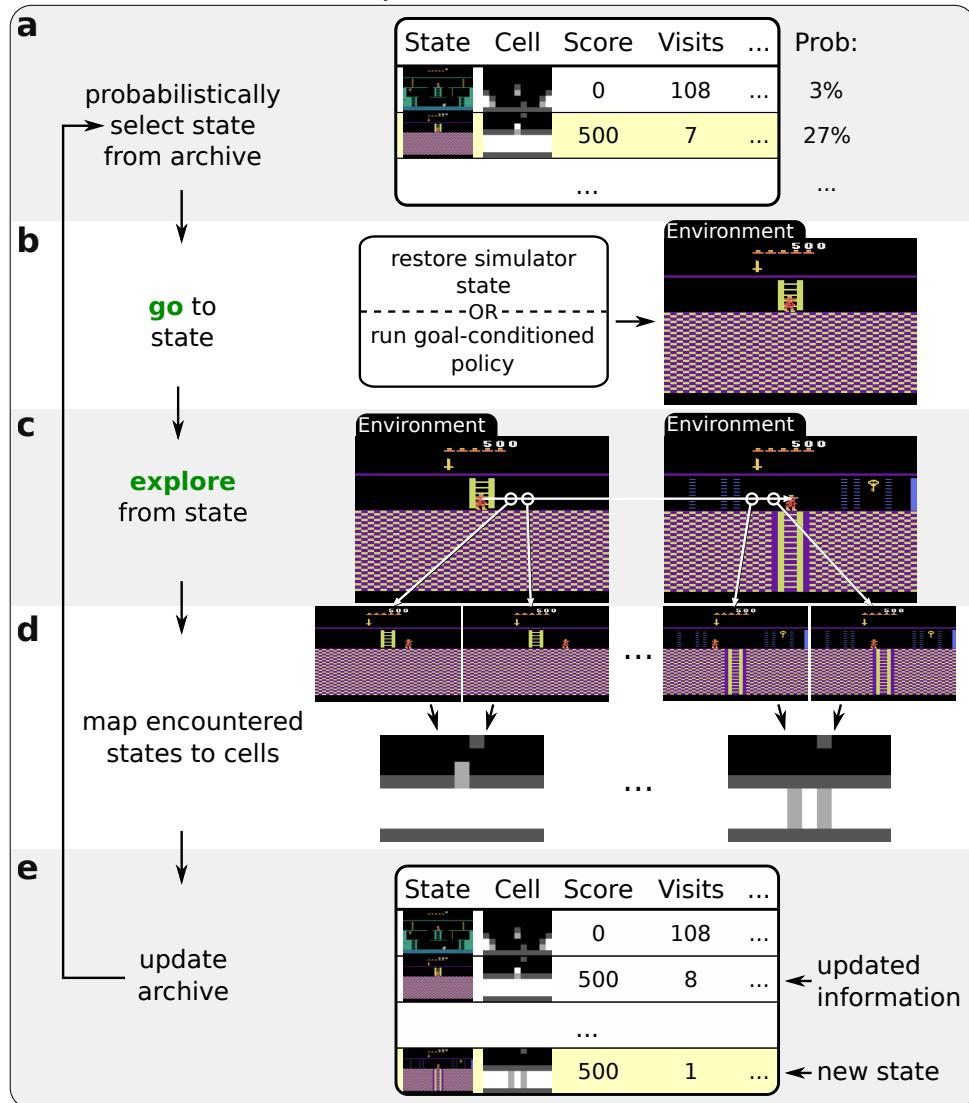
The Go-Explore family of algorithms

To avoid detachment, Go-Explore builds an *archive* of the different states it has visited in the environment, thus ensuring that states cannot be forgotten. Starting from an archive containing only the initial state, it builds this archive iteratively: first, it probabilistically selects a state to return to from the archive (Fig. 1a), returns to that state (the "go" step; Fig. 1b), then explores from that state (the "explore" step; Fig. 1c) and updates the archive with all novel states encountered (Fig. 1e). The overall process is reminiscent of classical planning algorithms (e.g. the archive can be considered a frontier, the "explore" step represents expanding a node, etc.), the potential of which have been relatively unappreciated within deep RL. However, for problems focused on by the RL community (like hard-exploration Atari games), which are high-dimensional with sparse rewards and/or stochasticity, no known planning method works[14,17]. Among other reasons (SI "Go-Explore, Planning, and Model-based RL"), such state spaces are too large to search exhaustively (requiring hard-to-invent heuristics to prune search) and stochastic transitions make it impossible to know whether a node has been fully expanded. Go-Explore can be seen as porting the principles of planning algorithms to these challenging problems.

Previous RL algorithms do not separate returning from exploring, and instead mix in exploration throughout an episode, usually by adding random actions a fraction of the time[15,18] or by sampling from a stochastic *policy* (a function that decides which action to take in each state, often a neural network)[19,20]. By first returning before exploring, Go-Explore avoids derailment by minimising exploration when returning (thus minimising failure to return) after which it can purely focus on exploration.

Because non-trivial environments have too many states to store explicitly, Go-Explore groups similar states into *cells*, and states are only considered novel if they are in a cell that does not yet exist in the archive (Fig. 1d). The archive stores one state per cell, and to maximise performance, if a state maps to an already known cell, but is associated with a better trajectory (higher performing or shorter; Methods), that state and its associated trajectory will replace the state and trajectory currently associated with that cell. Go-Explore selects states to return to (Fig. 1a) proportionally
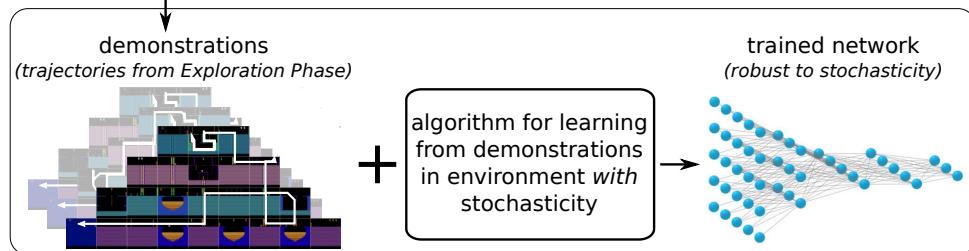
3

Figure 1: **Overview of Go-Explore.** (a) Probabilistically select a state from the archive, preferring states associated with promising cells. (b) Return to the selected state, such as by restoring simulator state or by running a goal-conditioned policy. (c) Explore from that state by taking random actions or sampling from a trained policy. (d) Map every state encountered during returning and exploring to a low-dimensional *cell* representation. (e) Add states that map to new cells to the archive and update other archive entries.

4

to weights it assigns to their associated cells in the archive (Methods).

While returning to a previously found state can be done with a trained policy (demonstrated in Sec. "Policy-based Go-Explore"), Go-Explore provides a unique opportunity to leverage the availability and widespread use of *simulators* in RL tasks[7,21–23]. Simulators are *restorable environments* because previous states can be saved and instantly returned to, thus completely negating derailment.

When exploiting this property of restorable environments, Go-Explore thoroughly explores the environment during its *exploration phase* by continually restoring (and subsequently taking exploratory actions from) one of the states in its archive (Fig. 1). It eventually returns the highest-scoring trajectory (sequence of actions) it found. Such trajectories are not robust to stochasticity or unexpected outcomes (e.g. a robot may slip and miss a crucial turn, invalidating the entire trajectory). To resolve this issue, Go-Explore trains a robust policy by *Learning from Demonstrations* (LfD)[24], where the exploration phase trajectories replace the usual human expert demonstrations (similar to Guo *et al.* (2014)[25]), in a variant of the environment featuring sufficient stochasticity to ensure robustness. The exploration-phase trajectories will be informative in the stochastic environment as long as following a close approximation to the example trajectory still leads to a high cumulative reward (SI "Go-Explore and Stochasticity"). Because it produces robust policies from open-loop (i.e. predetermined) trajectories, we call this LfD process the *robustification phase* (Fig. 1 "Robustification Phase").

Learning Atari with state restoration

The Atari benchmark suite[26], a prominent benchmark for RL algorithms[15,27,28], is an appropriate test-bed for Go-Explore because it contains a diverse set of games with varying levels of reward sparsity and deceptiveness. The following experiment highlights the benefit of a "go" step that restores the simulator state directly. In this experiment, the "explore" step happens through random actions, meaning the exploration phase operates entirely without a trained policy, which assumes that random actions have a sufficiently high probability of discovering new cells; more complex problems may require policy-based exploration (explored below). The state-to-cell mapping for Go-Explore's archive consists of *downscaling* the current game frame from the original 210×160 colour frame to a much smaller grayscale image, which (in contrast to most RL pre-processing that reduces dimensionality to save compute while minimising conflation[15]) aggregates similar-looking frames into the same cell (Fig. 1d). This mapping does not require game-specific knowledge and
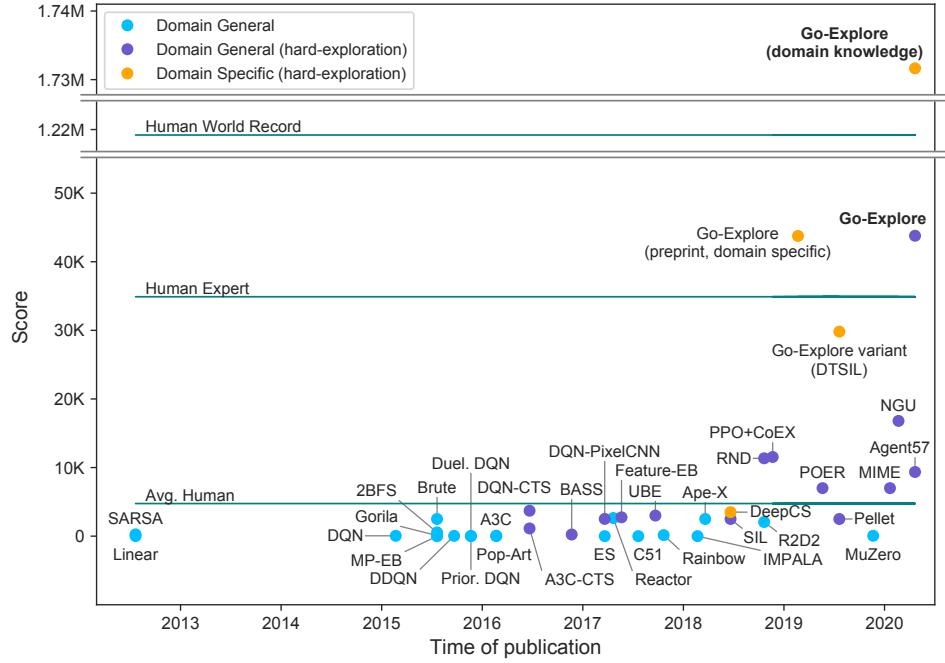
proves to be efficient across the entire Atari benchmark, though more complex environments may require more sophisticated (e.g. learned) representations. Good state-to-cell-mapping parameters result in a representation that strikes a balance between two extremes: lack of aggregation (e.g. one cell for every frame, which is computationally inefficient) and excessive aggregation (e.g. assigning all frames to a single cell, which prevents exploration). Because appropriate downscaling parameters (width, height, and number of possible grayscale values) vary across Atari games (SI "Exploration phase without dynamic representations") as well as when exploration progresses within a given game, these parameters are optimised *dynamically* at regular intervals (Methods "Downscaling on Atari"). The hyperparameters of this optimisation procedure are robust and generalise (SI "Ablations").

Here, the robustification phase consists of a modified version of the "backward algorithm"[29], currently the highest performing LfD algorithm on *Montezuma's Revenge*. Due to the large computational expense of the robustification process, this work focuses on the set of eleven games that have been considered hard-exploration challenges by the community[1] or for which the state-of-the-art performance was still below human performance (Methods "State of the art on Atari"). To ensure the trained policy becomes robust to environmental perturbations, during robustification stochasticity is added to these environments following current community standards[14]. The demonstrations provided by the exploration phase provide enough information about available rewards to allow Go-Explore to eschew standard reward clipping, which overemphasises small rewards[30], in favour of automatically scaling rewards to an appropriate range (Methods).

At test time, the mean performance of Go-Explore is both superhuman and surpasses the state of the art in all eleven games (except in Freeway where both Go-Explore and the state of the art reach the maximum score; Fig. 2b). These games include the grand challenges of *Montezuma's Revenge*, where Go-Explore quadruples the state-of-the-art score, and *Pitfall*, where Go-Explore surpasses average human performance while previous algorithms were unable to score any points. The number of frames processed in these experiments is 30 billion (Extended Data Fig. 2 and 4), similar to that of recent distributed RL algorithms[13,27,31]. While older algorithms often processed fewer frames, many of them show signs of convergence (meaning no further progress is expected), and it is often unclear whether these algorithms would be able to process billions of frames in a reasonable amount of time.

The ability of the exploration phase to find high-performing trajectories is not limited to hard-exploration problems; it finds trajectories with superhuman scores for all of the 55 Atari games

6

a Historical progress on Montezuma's Revenge.

b Performance on the 11 focus games of the Go-Explore variant that uses the downscaled representation during the exploration phase.

Game	Exploration Phase	Robustification Phase	SOTA	Avg. Human
Berzerk	131,216	**197,376**	1,383	2,630
Bowling	247	**260**	69	160
Centipede	613,815	**1,422,628**	10,166	12,017
Freeway	34	**34**	**34**	30
Gravitar	13,385	**7,588**	3,906	3,351
Montezuma's Revenge	24,758	**43,791**	11,618	4,753
Pitfall	6,945	**6,954**	0	6,463
Private Eye	60,529	**95,756**	26,364	69,571
Skiing	-4,242	**-3,660**	-10,386	-4,336
Solaris	20,306	**19,671**	3,282	12,326
Venture	3,074	**2,281**	1,916	1,187

Figure 2: **Performance of robustified Go-Explore on Atari games.** (a) Go-Explore produces substantial improvements over previous methods on Montezuma's Revenge, a grand challenge which has been the primary focus of hard-exploration research for many years. Different methods use different amounts of computation. Go-Explore processed a similar number of frames (30B) as other distributed RL algorithms like Ape-X (22B) and NGU (35B). (b) It exceeds the average human score in each of the 11 hard-exploration and unsolved games in the Atari suite, and matches or beats (often by a factor of 2 or more) the state of the art in each of these games. Bold indicates the best scores with stochastic evaluation. Score differences between the exploration and robustification phases are discussed in SI "Robustification scores analysis". A video of high performing runs can be found at https://youtu.be/e_aqRq59-Ns.
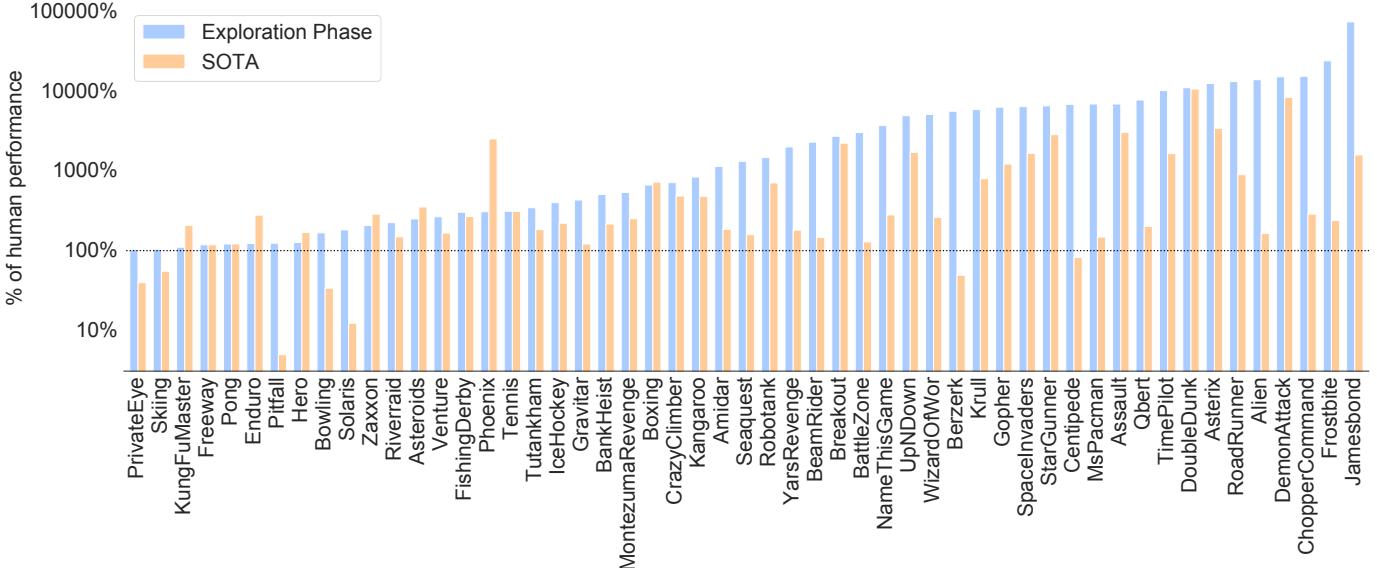
Figure 3: **Human-normalised performance of the exploration phase and state-of-the-art algorithms on all Atari games.** The exploration phase of Go-Explore exceeds human performance in every game, often by orders of magnitude, and outperforms the prior state of the art in most games (details in Extended Data Fig. 2a and Extended Data Table 3).

provided by OpenAI gym[32], a feat that has not been performed before (save concurrent work[13]). In 85.5% of these games the trajectories reach scores higher than those achieved by state-of-the-art RL algorithms (Fig. 3). Go-Explore's performance also exceeds that of planning algorithms (which similarly restore simulator states) that were evaluated on Atari[14,17].

In practical applications, it is often possible to define helpful features based on domain knowledge. Go-Explore can harness such easy-to-provide domain knowledge to substantially boost performance by constructing a cell representation (for the archive, not policy inputs) that only contains features relevant for exploration. The domain-knowledge features are the discretised position of the agent and relevant items held (Methods). With this domain knowledge cell representation, Go-Explore produces robustified policies that achieve a mean score of over 1.7 million on Montezuma's Revenge, surpassing the state of the art by a factor of 150 as well as the human world record of 1.2 million[33] (Fig. 2a). On Pitfall, the addition of domain knowledge produces robustified policies with a mean score of 102,571, close to the maximum possible of 112,000 and far above the state of the art of 0. The exploration phase explores both games extensively (Extended Data Fig. 3b), discovering effectively every unique location in each game (SI "Exploration in Atari"). Previous work suggests that intrinsic motivation algorithms benefit far less from domain knowl-

edge; a count-based exploration algorithm with the same domain knowledge representation scores 12,240 on Montezuma's Revenge[34].
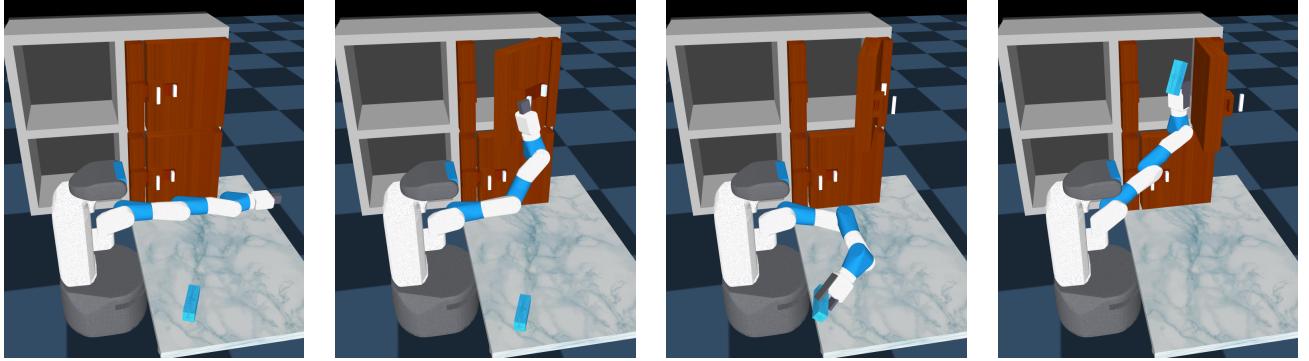
A hard-exploration robotics environment

While robotics is a promising application for RL and it is often easy to define the high-level goal of a robotics task (e.g. to put a cup in a cupboard), it is much more difficult to define a sufficiently dense reward function[10] (e.g. reward all of the low-level motor commands to move toward the cup, grasp it, etc.). Go-Explore enables forgoing such a dense reward function in favour of a sparse reward function that only considers the high-level task. Additionally, robot policies are usually trained in simulation before being transferred to the real world[7,21–23], making robotics a natural domain to demonstrate the usefulness of harnessing the ability to restore simulator state.
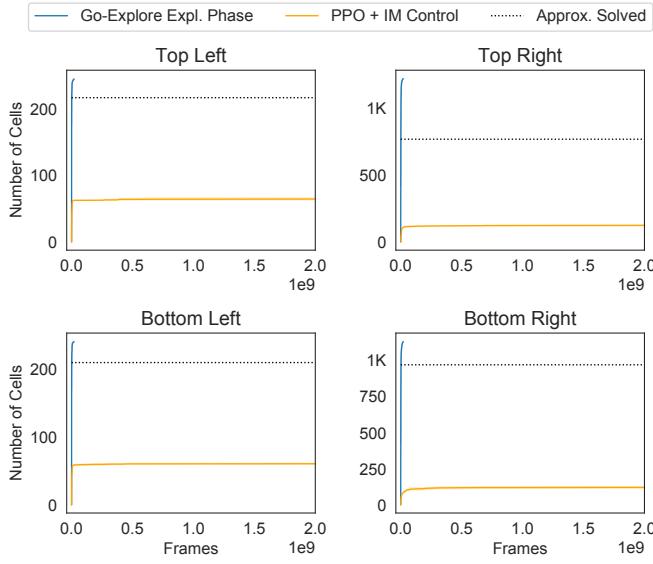
The following experiment, featuring a realistic simulation of a real-world robot[35], demonstrates that Go-Explore can solve a practical hard-exploration task where a robot arm must pick up an object and put it inside of one of four shelves, two of which are behind latched doors (Fig. 4a). A reward is given *only* when the object is put into a specified target shelf. A state-of-the-art RL algorithm for continuous control (PPO[20]) does not encounter a single reward after training in this environment for a billion frames, showcasing the hard-exploration nature of this problem. Go-Explore's "explore" step takes random actions and states are assigned to cells with an easy-to-provide domain-knowledge-based mapping (Methods).

The exploration phase quickly and reliably discovers trajectories for putting the object in each of the four shelves (Fig. 4b and Extended Data Fig. 5a). Go-Explore succeeds because it thoroughly explores its environment without suffering from detachment (e.g. once each cupboard is opened, Go-Explore never forgets about those states) or derailment (Go-Explore can directly restore to difficult-to-reach states like grasping). In contrast, a count-based intrinsic motivation algorithm with the same representation as the exploration phase is incapable of discovering any reward (Fig. 4c), and discovers only a fraction of the cells discovered by the exploration phase after 2 billion frames of training, 100 times more than the exploration phase (Fig. 4b). Despite receiving intrinsic rewards for touching the object, this control was incapable of learning to reliably grasp objects. Evidence suggests that this failure to grasp is due to the problem of *derailment* (SI "Derailment in robotics"), which Go-Explore is specifically designed to solve. Robustifying the trajectories found by Go-Explore produces robust policies in 99% of cases (Fig. 4c).

9

a Solving the top right shelf of the robotics environment.

b Cells discovered by the exploration phase and an intrinsic motivation control.

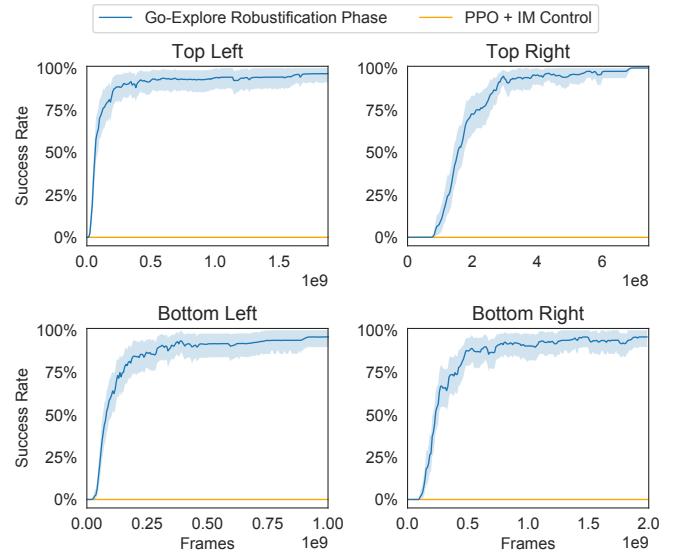**c** Robustification progress per target shelf.

Figure 4: **Go-Explore can Solve a Challenging, Sparse-Reward, Simulated Robotics Task.** (a) A simulated Fetch robot needs to grasp an object and put it in one of four shelves. (b) The exploration phase significantly outperforms an intrinsic motivation control using the same cell representation. (c) For each of four different target locations, including the two with a door, the robot is able to learn to pick the object up and place it on the shelf in 99% of trials. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

Policy-based Go-Explore

Leveraging the ability of simulators to restore states increases Go-Explore's efficiency, but is not a requirement. When returning, instead of restoring simulator state, it is possible to execute a policy conditioned on (i.e. told to go to) the cell to return to, which we call *policy-based Go-Explore*. There are advantages to doing so. First, it enables sampling from the policy during the "explore" step, which can substantially increase exploration efficiency vs. taking random actions because the policy can generalise to new situations (e.g. it need only learn to overcome a type of obstacle once instead of solving that problem again each time via random actions). To test this hypothesis, our implementation commits with equal probability to either taking random actions or sampling from the policy for the duration of the "explore" step, making it possible to compare random and policy-based exploration (Methods). Second, training a policy in the exploration phase obviates the need for robustification and thus removes its associated additional complexity, hyperparameters, and overhead. Finally, policy-based Go-Explore can explore directly in a stochastic environment (which we do in our experiments) and can potentially handle forms of stochasticity not explored in our experiments (e.g. stochastic rewards; SI "Policy-based Go-Explore and Stochasticity").

The goal-conditioned policy is trained during the exploration phase with a common RL algorithm (PPO[20]). Because goal-conditioned policies often struggle to reach distant states[36] (SI "Ablations"), the policy is guided towards the selected state by being presented with intermediate goals along the best trajectory that previously led to the selected state (Methods). Policy-based Go-Explore includes additional innovations to promote exploration and stabilise learning, the most important of which are Self-Imitation Learning[37] (SI "Ablations"), dynamic entropy increase, soft-trajectories, and dynamic episode limits, all discussed in detail in Methods.

Policy-based Go-Explore was tested on Montezuma's Revenge and Pitfall with the domain-knowledge cell representation for the archive (and to represent the goal to the policy; the game state is input as pixels). It beats the state of the art and human performance with a mean reward of 97,728 points on Montezuma's Revenge and 20,093 points on Pitfall (Fig. 5), demonstrating that Go-Explore's performance is not merely a result of its ability to leverage simulator restorability, but is a function of its overall design. Policy-based Go-Explore also outperforms a concurrently developed, similar algorithm[34] in terms of performance and sample efficiency (SI "Comparing Policy-based Go-Explore and DTSIL"). Furthermore, confirming our hypothesis, sampling from the policy is more effective at discovering new cells than taking random actions, and becomes increasingly effective across training because the policy gains new, generally useful skills, ultimately resulting
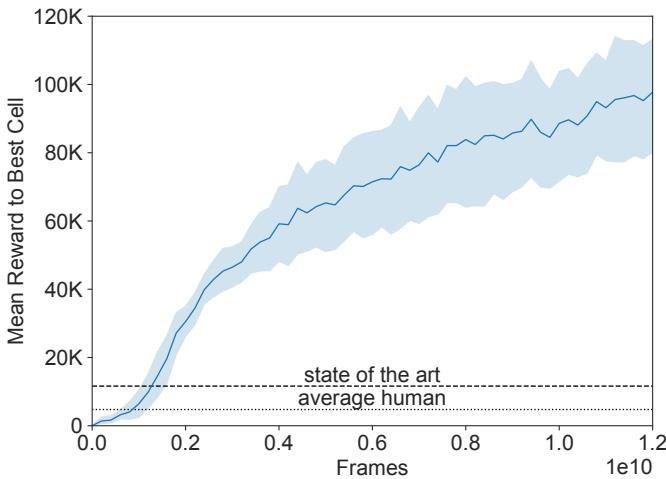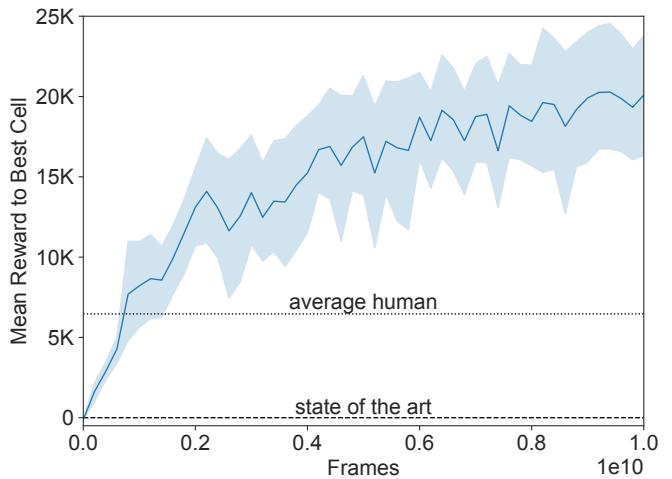
11

a Montezuma's Revenge   **b** Pitfall

Figure 5: **Policy-based Go-Explore with domain knowledge outperforms state-of-the-art and human performance in Montezuma's Revenge and Pitfall.** On both Montezuma's Revenge **(a)** and Pitfall **(b)**, performance increases throughout the run, suggesting even higher performance is possible with additional training time. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

in the discovery of over four times more cells than random actions on both Montezuma's Revenge and Pitfall (Extended Data Fig. 7), highlighting the potential of goal-conditioned, policy-based exploration over the usual random actions used in RL.

Conclusion

The effectiveness of the Go-Explore family of algorithms presented in this work suggests that it will enable progress in many domains that can be framed as sequential decision making problems, including robotics[7,21–23], language understanding[38], and drug design[39]. However, these instantiations represent only a fraction of the possible ways in which the Go-Explore paradigm can be implemented, opening up many exciting possibilities for future research. A key direction for future work is to learn cell representations, such as through compression-based methods[40,41], contrastive-predictive encodings[42] or auxiliary tasks[43], which would allow Go-Explore to generalise to even more complex domains. Other future extensions could learn to choose which cells to return to, learn which cells to try to reach during the exploration step, learn a specialised policy for exploration in the "explore" step, learn to explore safely in the real world by mining diverse catastrophes in simulation, maintain a continuous density-based archive rather than a discrete cell-based one,

improve sample efficiency by leveraging multiple trajectories (or even all transitions) from a single exploration-phase run or improve the robustification phase to work from a single demonstration, and so on. Furthermore, the planning-like nature of the Go-Explore exploration phase highlights the potential of porting other powerful planning algorithms like MCTS[44], RRT[45], A*[46], or conformant planning[47] to high-dimensional state-spaces. These new directions offer rich possibilities to improve the generality, performance, robustness, and efficiency of algorithms inspired by Go-Explore. Finally, the insights presented in this work extend broadly; the simple decomposition of remembering previously found states, returning to them, and then exploring from them appears to be especially powerful, suggesting it may be a fundamental feature of learning in general. Harnessing these insights, either within or outside of the context of Go-Explore, may be essential to improve our ability to create generally intelligent agents.

References

1. Bellemare, M. *et al. Unifying count-based exploration and intrinsic motivation* in *NIPS* (2016), 1471–1479.

2. Lehman, J. & Stanley, K. O. *Novelty Search and the Problem with Objectives* in *Genetic Programming Theory and Practice IX (GPTP 2011)* (2011).

3. Silver, D. *et al.* Mastering the game of Go without human knowledge. *Nature* **550,** 354–359 (2017).

4. Vinyals, O. *et al.* Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **575,** 350–354 (2019).

5. OpenAI *et al.* Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680* (2019).

6. Merel, J. *et al.* Hierarchical visuomotor control of humanoids. *arXiv preprint arXiv:1811.09656* (2018).

7. OpenAI *et al.* Learning dexterous in-hand manipulation. *The International Journal of Robotics Research* **39,** 3–20 (2020).

8. Lehman, J. *et al.* The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities. *CoRR* **abs/1803.03453** (2018).

9. Amodei, D. *et al.* Concrete Problems in AI Safety. *CoRR* **abs/1606.06565** (2016).

10. Smart, W. D. & Kaelbling, L. P. *Effective reinforcement learning for mobile robots* in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)* **4** (2002), 3404–3410.

11. Lehman, J. & Stanley, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* **19,** 189–223 (2011).

12. Conti, E. *et al. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents* in *Advances in neural information processing systems* (2018), 5027–5038.

13. Puigdomènech Badia, A. *et al.* Agent57: Outperforming the Atari Human Benchmark. *arXiv preprint arXiv:2003.13350* (2020).

14. Machado, M. C. *et al.* Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *J. Artif. Intell. Res.* **61,** 523–562 (2018).

15. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015).

16. Aytar, Y. *et al.* Playing hard exploration games by watching YouTube. *arXiv preprint arXiv:1805.11592* (2018).

17. Lipovetzky, N., Ramírez, M. & Geffner, H. *Classical Planning with Simulators: Results on the Atari Video Games* in *IJCAI* (2015).

18. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (Bradford, 1998).

19. Mnih, V. *et al. Asynchronous methods for deep reinforcement learning* in *International conference on machine learning* (2016), 1928–1937.

20. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal Policy Optimization Algorithms. *CoRR* **abs/1707.06347** (2017).

21. Cully, A., Clune, J., Tarapore, D. & Mouret, J.-B. Robots that can adapt like animals. *Nature* **521,** 503–507 (2015).

22. Peng, X. B., Andrychowicz, M., Zaremba, W. & Abbeel, P. *Sim-to-real transfer of robotic control with dynamics randomization* in *2018 IEEE international conference on robotics and automation (ICRA)* (2018), 1–8.

23. Tan, J. *et al.* Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332* (2018).

24. Hester, T. *et al. Deep Q-learning From Demonstrations* in *AAAI* (2018).

25. Guo, X., Singh, S. P., Lee, H., Lewis, R. L. & Wang, X. *Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning* in *NIPS* (2014).

26. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)* **47,** 253–279 (2013).

27. Horgan, D. *et al.* Distributed Prioritized Experience Replay. *CoRR* **abs/1803.00933** (2018).

28. Espeholt, L. *et al. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures* in *ICML* (2018).

29. Salimans, T. & Chen, R. Learning Montezuma's Revenge from a Single Demonstration. *arXiv preprint arXiv:1812.03381* (2018).

30. Van Hasselt, H. P., Guez, A., Hessel, M., Mnih, V. & Silver, D. *Learning values across many orders of magnitude* in *Advances in Neural Information Processing Systems* (2016), 4287–4295.

31. Puigdomènech Badia, A. *et al.* Never Give Up: Learning Directed Exploration Strategies. *arXiv* (2020).

32. Brockman, G. *et al.* OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).

33. *ATARI VCS/2600 Scoreboard* 2018. `http://www.ataricompendium.com/game_library/high_scores/high_scores.html`.

34. Guo, Y. *et al.* Efficient Exploration with Self-Imitation Learning via Trajectory-Conditioned Policy. *arXiv preprint arXiv:1907.10247* (2019).

35. Wise, M., Ferguson, M., King, D., Diehr, E. & Dymesich, D. *Fetch and freight: Standard platforms for service robot applications* in *Workshop on autonomous mobile service robots* (2016).

36. Eysenbach, B., Salakhutdinov, R. R. & Levine, S. *Search on the replay buffer: Bridging planning and reinforcement learning* in *Advances in Neural Information Processing Systems* (2019), 15220–15231.

37. Oh, J., Guo, Y., Singh, S. & Lee, H. *Self-Imitation Learning* in *ICML* (2018).

38. Madotto, A. *et al.* Exploration Based Language Learning for Text-Based Games. *arXiv preprint arXiv:2001.08868* (2020).

39. Popova, M., Isayev, O. & Tropsha, A. Deep reinforcement learning for de novo drug design. *Science advances* **4,** eaap7885 (2018).

40. Alvernaz, S. & Togelius, J. *Autoencoder-augmented neuroevolution for visual doom playing* in *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (2017), 1–8.

41. Cuccu, G., Togelius, J. & Cudré-Mauroux, P. Playing atari with six neurons. *arXiv preprint arXiv:1806.01363* (2018).

42. Oord, A. v. d., Li, Y. & Vinyals, O. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

43. Jaderberg, M. *et al.* Reinforcement Learning with Unsupervised Auxiliary Tasks. *CoRR* **abs/1611.05397** (2016).

44. Chaslot, G., Bakkes, S., Szita, I. & Spronck, P. *Monte-Carlo Tree Search: A New Framework for Game AI* in *AIIDE* (2008).

45. Lavalle, S. M. *Rapidly-Exploring Random Trees: A New Tool for Path Planning* tech. rep. (Iowa State University, 1998).

46. Hart, P. E., Nilsson, N. J. & Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* **4,** 100–107. ISSN: 0536-1567 (July 1968).

47. Smith, D. E. & Weld, D. S. *Conformant graphplan* in *AAAI/IAAI* (1998), 889–896.

Methods

State of the art on Atari. With new work on RL for Atari being published on a regular basis, and with reporting methods often varying significantly, it can be difficult to establish the state-of-the-art score for each Atari game. At the same time, it is important to compare new algorithms with previous ones to evaluate progress.

For determining the state-of-the-art score for each game, we considered a set of notable, recently published papers that cover at least the particular subset of games this paper focuses on, namely hard-exploration games. Community guidelines advocate *sticky actions*, which approximate the minor lack of precision in control that a human might have (e.g. continuing to tilt the joystick for a fraction of a second longer than intended), as a way to evaluate agents on Atari[14]. There is substantial evidence to show that sticky actions can decrease performance substantially compared to the now deprecated *no-ops* evaluation strategy[14,48,49]. As a result, we exclude work which was only evaluated with no-ops from our definition of state of the art. Fig. 2a includes works tested only with no-ops as they help bring context to the amount of effort expended by the community on solving Montezuma's Revenge. We did not include work that does not provide individualised scores for each game. To avoid cherry-picking lucky rollouts that can bias scores upward substantially, we also exclude work that only provided the maximum score achieved in an entire run as opposed to the average score achieved by a particular instance of the agent.

In total, state of the art results were extracted from the following papers: Burda *et al.* (2018)[50], Castro *et al.* (2018)[48], Choi *et al.* (2018)[51], Fedus *et al.* (2019)[52], Taiga *et al.* (2020)[53], Tang *et al.* (2020)[54], and Toromanoff *et al.* (2019)[49]. Because these works themselves report scores for several algorithms and variants, including reproductions of previous algorithms, a total of 23 algorithms and variants were included in the state of the art assessment. For each game, the state-of-the-art score was the highest score achieved across all algorithms.

Downscaling on Atari. In the first variant of Go-Explore presented in this work (Sec. "Learning Atari with state restoration"), the cell representation is a downscaled version of the original game frame, which can be applied in any domain where the state is a visual observation (SI "Generality of downscaling").

To obtain the downscaled representation, (1) the original frame is converted to grayscale, (2) its resolution is reduced with pixel area relation interpolation to a width $w \leq 160$ and a height $h \leq 210$, and (3) the pixel depth is reduced to $d \leq 255$ using the formula $\lfloor \frac{d \cdot p}{255} \rfloor$, where p is

the value of the pixel after step (2). A fixed set of values for the parameters w, h and d would not generalise across games because visuals (e.g. the amount of detail shown on screen and how much it varies across frames) vary substantially between games (SI "Ablations"). Therefore these parameters are updated dynamically by proposing different values for each, calculating how a sample of recent frames would be grouped into cells under these proposed parameters, and then selecting the values that result in the best cell distribution (as determined by the objective function defined below).

The objective function for candidate downscaling parameters is calculated based on a target number of cells T (where T is a fixed fraction of the number of cells in the sample, 12.5% in our experiment, though the algorithm is relatively robust to different values, see SI "Downscaling target proportion"), the actual number of cells produced by the parameters currently considered n, and the distribution of sample frames over cells $\mathbf{p}$. Its general form is

$$O(\mathbf{p}, n) = \frac{H_n(\mathbf{p})}{L(n, T)} \tag{1}$$

$L(n, T)$ measures the discrepancy between the number of cells under the current parameters n and the target number of cells T. It prevents the representation that is discovered from aggregating too many frames together, which would result in low exploration, or from aggregating too few frames together, which would result in an intractable time and memory complexity, and is defined as

$$L(n, T) = \sqrt{\left| \frac{n}{T} - 1 \right| + 1} \tag{2}$$

$H_n(\mathbf{p})$ is the ratio of the entropy of how frames were distributed across cells to the entropy of the discrete uniform distribution of size n, i.e. the normalised entropy. In this way, the loss encourages frames to be distributed as uniformly as possible across cells, which is important because highly non-uniform distributions may suffer from the same lack of exploration that excessive aggregation can produce or the same intractability that lack of aggregation can produce. Unlike unnormalized entropy, normalised entropy is comparable across different numbers of cells, allowing the number of cells to be controlled solely by $L(n, T)$. Its form is

$$H_n(\mathbf{p}) = -\sum_{i=1}^{n} \frac{p_i \log(p_i)}{\log(n)} \tag{3}$$

18

At each step of the randomised search, new values of each parameter w, h and d are proposed by sampling from a geometric distribution whose mean is the current best known value of the given parameter. If the current best known value is lower than a minimum mean (set to approximately $1/20^{\text{th}}$ of the maximum value of each parameter: 8 for w, 10.5 for h and 12 for d), the minimum mean is used as the mean of the geometric distribution (SI "Downscaling distribution minimum means" shows that the algorithm is not overly sensitive to the particular setting of the minimum means). New parameter values are re-sampled if they fall outside of the valid range for that parameter. In our implementation, the randomised search runs for 3,000 iterations.

The recent frames that constitute the sample over which parameter search is done are obtained by maintaining a set of recently seen sample frames as Go-Explore runs: each time a frame not already in the set is seen during the explore step, it is added to the running set with a probability of 1%, ensuring that the set contains a diverse set of frames rather than just the most recent frames (SI "Downscaling sampling rate" shows that the algorithm is not overly sensitive to the value of this parameter). If the resulting set contains more than 10,000 frames, the oldest frame it contains is removed. This set is reminiscent of a FIFO replay buffer, except that, because it is not used for training the network, it only stores individual frames, rather than complete state transitions.

The first downscaling parameters are computed after running with a single-cell representation for 40,000 frames. To handle changes in frame distribution as exploration progresses and to avoid being stuck with a bad representation, the search for a new representation is performed every 40 million frames. To avoid excessive memory usage, the representation is also recomputed if the number of cells in the archive exceeds 50,000. When switching to a new representation, a new archive is created and initialised by converting all previous archives to the new representation using the frames corresponding to each state in the previous archives (the number of cells in the archive on Atari over time can be seen in Extended Data Fig. 3a).

Hyperparameter values were found by an initial randomised sweep on Montezuma's Revenge, with the 10 best combinations then tested on Gravitar to ensure their generalisability (it is the norm in hard-exploration work to include Montezuma's Revenge as part of the tuning set[1,13,31,34,50,51,53,55–57], though which other games are included, if any, varies). Aside from the hyperparameters examined in SI "Ablations" (the target proportion, minimum means and buffer sampling rate), the hyperparameters control the tradeoff between computational and memory efficiency and the quality of downscaling parameters obtained (e.g. increasing the number of search iterations is likely to produce better parameters at the cost of more time spent searching for parameters), and should thus

be set according to the computational constraints of the user. In our experiments, approximately 25% of the computation spent on the exploration phase was spent searching for new downscaling parameters.

Domain knowledge representations. The domain knowledge representation for Pitfall consists of the current room (out of 255) the agent is currently located in, as well as the discretised x, y position of the agent. In Go-Explore without a return policy, the x, y position is discretised in 8 by 16 pixel cells. Policy-based Go-Explore uses the coarser-grained 18 by 18 pixel cell representation from Guo *et al.* (2019)[34], which reduces training time (there are fewer cells the policy needs to learn how to reach) without hindering exploration. In Montezuma's Revenge, the representation also includes the keys currently held by the agent (including which room they were found in) as well as the current level. Most of these features (level, room and x, y position) serve to specify the location of the agent, capturing the intuition that exploration requires discovering the different available locations within a space, while the keys held by the agent are important affordances that allow the agent to reach new locations. Although this information can in principle be extracted from RAM, in this work it was extracted from pixels through small hand-written classifiers, showing that domain knowledge representations need not require access to the inner state of a simulator. For practical applications, the features that help with exploration are often easier to identify and obtain than the features that are necessary for a policy to successfully execute a task. For example, in a task where a robot has to pick up an object, it is clear that the robot should explore different positions for its end effector in order to find a good grip on the object and the end effector position is generally easy to obtain[58,59], but a policy executing such a task will also need to recognise the object itself under a wide range of circumstances, which may require advanced image processing that benefits from being learned[60].

In robotics, the domain knowledge representation is extracted from the internal state of the MuJoCo[61] simulator. However, similar information has been extracted from raw camera footage for real robots by previous work[7]. It consists of the current 3D position of the robot's gripper, discretised in voxels with sides of length 0.5 meters, whether the robot is currently touching (with a single grip) or grasping (touching with both grips) the object, and whether the object is currently in the target shelf. In the case of the two target shelves with doors, the positions of the door and its latch are also included. The discretization for latches and doors follows the following formula, given that d is the distance of the latch/door from its starting position in meters: $\lfloor \frac{d+0.195}{0.2} \rfloor$.

Exploration phase. During the exploration phase (SI Algorithm 1), the selection probability of a cell at each step is proportional to its *selection weight*, which unless otherwise specified is calculated as:

$$W = \frac{1}{\sqrt{C_{\text{seen}} + 1}} \tag{4}$$

where C_{seen} is the number of exploration steps in which that cell is visited (i.e. the C_{seen} count of a cell is increased by one when it is visited in the exploration step, even if the cell was visited multiple times in that step). This reciprocal square root weight is similar to the exploration bonus used in algorithms such as UCT[62] and count-based intrinsic motivation algorithms[1,63].

One advantage of introducing domain knowledge into cell representations is that we can leverage our semantic understanding of domain features to improve cell selection. We demonstrate this advantage on Montezuma's Revenge with domain knowledge but without a return policy, where we define the cell selection weight based on: (1) The number of horizontal neighbours to the cell present in the archive (h); (2) A key bonus: for each location (defined by level, room, and x,y position), the cell with the largest number of keys at that location gets a bonus of $k = 1$ ($k = 0$ for other cells); (3) the current level. The first two values contribute to the location weight

$$W_{\text{location}} = \frac{2 - h}{10} + k. \tag{5}$$

This value captures the intuition that a cell that *lacks* neighbours in the archive is likely to be at the current frontier of search (vertical neighbours do not have the same effect as it is often more difficult to move from one vertical level to another, requiring e.g. a ladder to be present), and that an agent has more exploration capacity (i.e. affordances) if it is holding more keys. W_{location} is then combined with W above as well as the level of the given cell l and the maximum level in the archive L to obtain the final weight for Montezuma's Revenge with domain knowledge:

$$W_{\text{mont\_domain}} = 0.1^{L-l} \left(W + W_{\text{location}} \right). \tag{6}$$

This level-weighing puts a much stronger weight on cells in the highest level reached so far, thus focusing exploration on the frontier of search. These domain knowledge features substantially improve sample complexity in Montezuma's Revenge relative to the default selection weight W defined above, but Go-Explore with the default selection weight is still able to get to the end of level 3, and thus still finds trajectories that traverse Montezuma's Revenge in its entirety (SI "Ablations"). While it is possible to produce an analogous domain knowledge cell selection weight

for Pitfall with domain knowledge, no such weight produced any substantial improvement over W alone.

Unless otherwise specified, once a cell is returned to, exploration proceeds with random actions for a number of steps (100 in Atari, 30 in robotics), or until the end of episode signal is received from the environment. In Atari, where the action set is discrete, actions are chosen uniformly at random. In robotics, each of the 9 continuous-valued components of the action is sampled independently and uniformly from the interval from -1 to 1. To help explore in a consistent direction, the probability of repeating the previous action is 95% for Atari and 90% for robotics. The effect of action repetition is investigated in SI "Ablations".

For increased efficiency, the exploration phase is processed in parallel by selecting a batch of return cells and exploring from each one of them across multiple processes. In all runs without a return policy, the batch size is 100.

All reported experiments, except those involving policy-based Go-Explore, return by directly restoring simulator state. This method of returning is available whenever a simulator is available, which is the case for most RL experiments; due to the large number of training trials current RL algorithms require, as well as the safety concerns that arise when running RL directly in the real world, simulators have played a key role in training the most compelling applications of RL, and will likely continue to be harnessed for the foreseeable future.

The backward algorithm. The "backward algorithm"[29] places the agent close to the end of the trajectory and runs PPO (SI "PPO and SIL") until the performance of the agent matches that of the demonstration. Once that is achieved, the agent's starting point is moved closer to the trajectory's beginning and the process is repeated.

The algorithm was modified to support multiple (10, in our experiments) demonstrations by selecting a demonstration uniformly at random at the start of each episode, which stabilises learning. The demonstrations can be obtained cheaply by running the exploration phase multiple times. In Atari, the agent may be able to find rewards from the starting position before it has worked backwards all the way to the start in a way that matches the demonstration performance. To track such partial progress, a virtual "demonstration" corresponding to starting the agent at the true starting point was added (SI "Multiple demonstrations"). This process was not performed in the robotics environment as there is only one point to score, making partial success impossible. Self-Imitation Learning (SIL)[37] was performed on the demonstrations provided to the backward algorithm (SI

"PPO and SIL"). In Atari, we normalise the rewards based on the mean absolute returns found in the demonstrations to allow a single set of hyperparameters to be used across all games, including those with widely varying reward magnitudes (SI "Reward scaling"). Pseudo-code, which includes the modifications above, is shown in SI Algorithm 2, and the neural-network architectures that were trained are shown in Extended Data Fig. 1.

Evaluation. In Atari, the score of an exploration phase run is measured as the highest score ever achieved at episode end (SI "Score tracking in the exploration phase"). For the 11 focus games, exploration phase scores are averaged across 50 exploration phase runs. For the other games, they are averaged across 5 runs. For domain knowledge, they are averaged across 100 runs.

On Atari, only the 11 focus games are robustified and evaluated in a stochastic setting. Modern RL algorithms are already able to adequately solve the games not included in the 11 focus games in this work, as demonstrated by previous work (Extended Data Table 3). Thus, because robustifying these already solved games would have been prohibitively expensive, we did not perform robustification experiments for these 44 games.

During robustification, a checkpoint is produced every 100 training iterations (13,926,400 frames). A subset of checkpoints corresponding to points during which the rolling average of scores seen during training was at its highest are tested by averaging their scores across 100 test episodes. Then the highest scoring checkpoint found is retested with 1,000 new test episodes to eliminate selection bias. For the downscaled representation, robustification scores are averaged over 5 runs. For domain knowledge, they are averaged across 10 runs. All testing is performed with sticky actions (see "State of the art on Atari"). To accurately compare against the human world record of 1.2 million[33], we patched an ALE bug that prevents the score from exceeding 1 million (SI "ALE issues").

The exploration phase for robotics was evaluated across 50 runs per target shelf, for a total of 200 runs. The reported metric is the proportion of runs that discovered a successful trajectory. Because the outcome of a robotics episode is binary (success or failure), there is no reason to continue robustification once the agent is reliably successful (unlike with Atari where it is usually possible to further improve the score). Thus, robustification runs for robotics are terminated once they keep a success rate greater than 98.5% for over 150 training iterations (19,660,800 frames), and the runs are then considered successful. To ensure that the agent learns to keep the object inside of the shelf, a penalty of -1 is given for taking the object outside of the shelf, and during robustification the agent is given up to 54 additional steps after successfully putting the object in

the shelf (see "Extra frame coef" in Extended Data Table 1a), forcing it to ensure the object doesn't leave the shelf. Out of 200 runs (50 per target shelf), 2 runs did not succeed after running for over 3 billion frames (whereas all other runs succeeded in fewer than 2 billion) and were thus considered unsuccessful (one for the bottom left shelf and the other for the bottom right shelf), resulting in a 99% overall success rate.

The robotics results are compared to two controls. First, to confirm the hard-exploration nature of the environment, 5 runs per target shelf of ordinary PPO[20] with no exploration mechanism were run for 1 billion frames. At no point during these runs were any rewards found, confirming that the robotics problem in this paper constitutes a hard-exploration challenge. Secondly, we ran 10 runs per target shelf for 2 billion frames of ordinary PPO augmented with count-based intrinsic rewards, one of the best modern versions of intrinsic motivation[1,53,63,64] designed to deal with hard-exploration challenges. The representation for this control is identical to the one used in the exploration phase, so as to provide a fair comparison. Similar to the exploration phase, the counts for each cell are incremented each time the agent enters a cell for the first time in an episode, and the intrinsic reward is given by $\frac{1}{\sqrt{n}}$, similar to W. Because it is possible (though rare) for the agent to place the object out of reach, a per-episode time limit is necessary to ensure that not too many training frames are wasted on such unrecoverable states. In robustification, the time limit is implicitly given by the length of the demonstration combined with the additional time described above and in Extended Data Table 1a. For the controls, a limit of 300 time steps was given as it provides ample time to solve the environment (Extended Data Fig. 5b), while ensuring that the object is almost always in range of the robot arm throughout training. As shown in Fig. 4b, this control was unable to find anywhere near the number of cells found by the exploration phase, despite of running for significantly longer, and as shown in Fig. 4c, it also was unable to find any rewards in spite of running for longer than any successful Go-Explore run (counting both the exploration phase and robustification phase combined).

Hyperparameters. Hyperparameters are reported in Extended Data Table 1. Extended Data Table 1b reports the hyperparameters specific to the Atari environment. Of note are the use of sticky actions as recommended by Machado *et al.* (2018)[14], and the fact that the agent acts every 4 frames, as is typical in RL for Atari[15]. In this work, sample complexity is always reported in terms of raw Atari frames, so that the number of actions can be obtained by dividing by 4. In robotics, the agent acts 12.5 times per second. Each action is simulated with a timestep granularity of 0.001 seconds, corresponding to 80 simulator steps for every action taken.

24

While the robustification algorithm originates from Salimans & Chen (2018)[29], it was modified in various ways (Methods "The backward algorithm"). Extended Data Table 1a shows the hyper-parameters for this algorithm used in this work, to the extent that they are different from those in the original paper, or were added due to the modifications in this work. Extended Data Tables 2a and 2b show the state representation for robotics robustification.

With the downscaled representation on Atari, the exploration phase was run for 2 billion frames prior to extracting demonstrations for robustification. Because exploration phase performance was slightly below human on Pitfall, Skiing and Private Eye, the exploration phase was allowed to run longer on these three games (5 billion for Pitfall and Skiing, 15 billion for Private Eye) to demonstrate that it can exceed human performance on all Atari games. The demonstrations used to robustify these three games were still extracted after 2 billion frames, and the robustified policies still exceeded average human performance thanks to the ability of robustification to improve upon demonstration performance. With the domain knowledge representation on Atari, the exploration phase ran for 1 billion frames. Robustification ran for 10 billion frames on all Atari games except Solaris (20 billion) and Pitfall when using domain knowledge demonstrations (15 billion). On robotics, the exploration phase ran for 20 million frames and details for the robustification phase are given in the Evaluation section.

Policy-based Go-Explore. The idea in policy-based Go-Explore is to learn how to return (rather than to restore archived simulator states to return). The algorithm builds off the popular PPO algorithm[20] (SI "PPO and SIL") and pseudo-code for the algorithm is shown in SI Algorithm 3. At the heart of policy-based Go-Explore lies a goal-conditioned policy $\pi_\theta(a|s,g)$ (Extended Data Fig. 1c), parameterised by θ, that takes a state s and a goal g and defines a probability distribution over actions a. Policy-based Go-Explore includes all PPO loss functions described in SI "PPO and SIL", except that instances of the state s are replaced with the state-goal tuple (s,g). The total reward r_t is the sum of the trajectory reward r_t^τ (defined below) and the environment reward r_t^e, where r_t^e is clipped to the $[-2,2]$ range. Because most rewards in Atari have an absolute value greater than 2, this clip range effectively sets the magnitude of in-game rewards to 2. Given that trajectory rewards are 1 (see below), this clipping implements the intuition that in-game rewards should be more important than following the trajectory. We implement this intuition in the form of clipping so as to not increase the importance of the smallest Atari rewards. Policy-based Go-Explore also includes self-imitation learning (SIL)[37] (SI "PPO and SIL"), where SIL actors follow the same procedure as regular actors, except that they replay the trajectory associated with the cell

25

they select from the archive. Hyperparameters are listed in Extended Data Table 1a.

To fit the batch-oriented paradigm, policy-based Go-Explore updates its archive after every mini-batch (Extended Data Fig. 6). In addition, the "go" step now involves executing actions in the environment (as explained below), and each actor independently tracks whether it is in the "go" step or the "explore" step of the algorithm. For the purpose of updating the archive, no distinction is made between data gathered during the "go" step and data gathered during the "explore" step, meaning policy-based Go-Explore can discover new cells or update existing cells while returning.

For the experiments presented in this paper, data is gathered in episodes. Whenever an actor starts a new episode, it selects a state from the archive with a cell selection weight of:

$$W = \frac{1}{0.5C_{\text{steps}} + 1} \tag{7}$$

where C_{steps} is the total number of steps the agent spend in the cell. This equation is different from the one in the exploration-phase without a policy ($0.5C_{\text{steps}}$ grows much faster than $\sqrt{C_{\text{seen}}}$) because policy-based Go-Explore benefits from focusing more strongly on the most recently discovered cells for two reasons: (1) after a new cell is discovered in policy-based Go-Explore the policy may first need to learn how to return there reliably; focusing on new cells helps the agent collect the necessary experience to do so, and (2) policy-based Go-Explore will visit many cells along the way to a target cell, allowing it to explore from those intermediate cells without selecting them explicitly (Extended Data Fig. 7). After a cell is selected, policy-based Go-Explore runs its goal-conditioned policy to reach the selected state, which enables it to be applied without assuming access to a deterministic or restorable environment during the exploration phase. It is exceedingly difficult and practically unnecessary to reach a particular state exactly, so instead, the policy is conditioned to reach the *cell* associated with this state, referred to as the *goal cell*, provided to the policy in the form of a concatenated one-hot encoding for every attribute characterizing the cell. Directly providing the goal cell to the goal-conditioned policy did not perform well (SI "Ablations"), presumably because goal-conditioned policies tend to falter when goals become distant[36]. Instead, the actor is iteratively conditioned on the successive cells traversed by the archived trajectory that leads to the goal cell.

Here, we allow the agent to follow the archived trajectory in a soft-order, a method similar to the one described in Guo *et al.* (2019)[34]. To prevent the soft-trajectory from being affected by the time an agent spends in a cell, the algorithm first constructs a trajectory of non-repeated cells,

collapsing any consecutive sequence of identical cells into a single cell. Then, given a window size $N_w = 10$, if the agent is supposed to reach a specific goal cell in this trajectory and it reaches that or any of the subsequent 9 cells in this trajectory, the goal is considered met. When a goal is met, the agent receives a trajectory reward r_t^τ of 1 and the subsequent goal in the non-repeated trajectory (i.e. the goal that comes after the cell that was actually reached) is set as the next goal. When the cell that was reached occurs multiple times in the window (indicating cycles) the next goal is the one that follows the last occurrence of this repeated goal cell.

When an agent reaches the last cell in the trajectory, it receives a trajectory reward r_t^τ of 3, which is higher than the intermediate trajectory reward of 1 to implement the general practice of having a higher reward for reaching a desired final state than for completing any intermediate objectives[65,66]: this practice improved performance (SI "Ablations"). Then the agent executes the "explore" step, either through *policy exploration* or *random exploration*. With policy exploration, the agent will select a goal for the policy according to one of three rules: (1) with 10% probability, randomly select an adjacent cell (see "Exploration phase") not in the archive, (2) with 22.5% probability, select any adjacent cell, whether already in the archive or not, and (3), in the remaining 67.5% of cases, select a cell from the archive according to the standard cell-selection weights. If the first rule does not apply because all adjacent cells are already in the archive, rules 2 and 3 are selected with proportionally scaled probabilities. Note that, in the exploration step, the agent is presented directly with the goal, rather than with a trajectory. Whenever the current exploration goal is reached, or if the goal is not reached for some number of steps (here 100), a new exploration goal is chosen. With random exploration, the agent takes random actions according to the random exploration procedure described in Methods "Exploration phase". All gathered data is ignored with respect to calculating the loss of the policy.

While following a trajectory or during exploration, it is possible for the agent to fail to make progress towards the current goal cell because the policy has converged towards putting all its probability mass on a small set of actions, meaning the policy performs insufficient exploration to discover the goal and observe its reward. To alleviate this issue, in addition to having the entropy bonus $\mathcal{L}^{ENT}$, the policy is extended with an entropy term e_t that divides the logits of the policy right before the softmax activation function is applied. If the agent fails to reach the current goal for some number of steps e_t^T (defined below), this entropy term is increased following:

$$e_t(\hat{t}) = 1 + (\max(0, \hat{t} - e_t^T) \cdot e_f)^{e_p} \tag{8}$$

where $\hat{t}$ is the number of steps the agent has taken since it last reached a goal (for returning) or discovered a new cell (for exploring), $e_f = 0.01$ is the entropy increase factor and $e_p = 2$ is the entropy increase power. While executing the "explore" step, the threshold e_t^T has a fixed value of 50. While returning, the threshold e_t^T equals the number of actions that the followed trajectory required to move from the previously reached goal cell to the current goal cell. Here, the previously reached goal cell refers to the first cell in the soft-trajectory window that matched the cell occupied by the agent at the time the previous goal was considered met.

Lastly, to prevent actors from spending many time steps without making any progress (possibly because the agent reached a state from which further progress is impossible), we terminate the episode early if the current goal is not reached within 1,000 steps after we have started to increase entropy (while returning), or if no new cells are discovered for 1,000 steps (while exploring). For Montezuma's Revenge with policy-based Go-Explore only, we also terminate the episode on death to deal with an ALE bug (details in Supplemental Information).

Robotics environment. The environment, from https://github.com/vikashplus/fetch, features a realistic model[67] of the Fetch Mobile Manipulator[35] and was minimally modified to implement a sparse-reward pick-and-place task. The modified environment is included with the Go-Explore code.

References
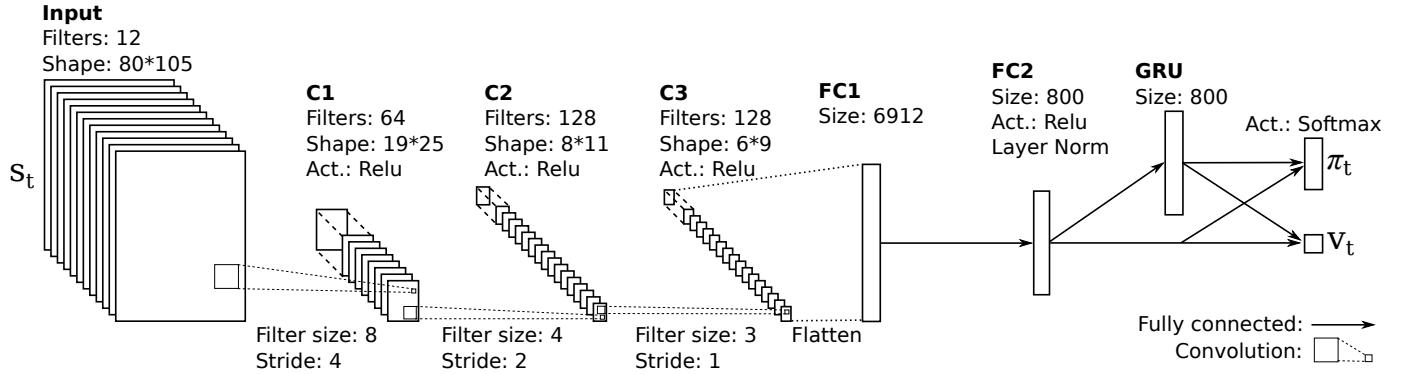
48. Castro, P. S., Moitra, S., Gelada, C., Kumar, S. & Bellemare, M. G. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110* (2018).

49. Toromanoff, M., Wirbel, E. & Moutarde, F. Is Deep Reinforcement Learning Really Superhuman on Atari? *arXiv preprint arXiv:1908.04683* (2019).

50. Burda, Y., Edwards, H., Storkey, A. & Klimov, O. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894* (2018).

51. Choi, J. *et al.* Contingency-Aware Exploration in Reinforcement Learning. *CoRR* **abs/1811.01483** (2018).

52. Fedus, W., Gelada, C., Bengio, Y., Bellemare, M. G. & Larochelle, H. Hyperbolic discounting and learning over multiple horizons. *arXiv preprint arXiv:1902.06865* (2019).

53. Taiga, A. A., Fedus, W., Machado, M. C., Courville, A. & Bellemare, M. G. *On Bonus Based Exploration Methods In The Arcade Learning Environment* in *International Conference on Learning Representations* (2020).

54. Tang, Y., Valko, M. & Munos, R. Taylor expansion policy optimization. *arXiv preprint arXiv:2003.06259* (2020).

55. Ostrovski, G., Bellemare, M. G., van den Oord, A. & Munos, R. *Count-Based Exploration with Neural Density Models* in *ICML* (2017).

56. Martin, J., Sasikumar, S. N., Everitt, T. & Hutter, M. *Count-Based Exploration in Feature Space for Reinforcement Learning* in *IJCAI* (2017).

57. O'Donoghue, B., Osband, I., Munos, R. & Mnih, V. *The Uncertainty Bellman Equation and Exploration* in *ICML* (2018).

58. Goldenberg, A., Benhabib, B. & Fenton, R. A complete generalized solution to the inverse kinematics of robots. *IEEE Journal on Robotics and Automation* **1,** 14–20 (1985).

59. Spong, M. W., Hutchinson, S., Vidyasagar, M., *et al. Robot modeling and control* (2006).

60. Zhao, Z.-Q., Zheng, P., Xu, S.-t. & Wu, X. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems* **30,** 3212–3232 (2019).

61. Todorov, E., Erez, T. & Tassa, Y. *MuJoCo: A physics engine for model-based control* in *IROS* (2012), 5026–5033.

62. Kocsis, L. & Szepesvári, C. *Bandit Based Monte-Carlo Planning* in *ECML* (2006).

63. Strehl, A. L. & Littman, M. L. An analysis of model-based interval estimation for Markov decision processes. *Journal of Computer and System Sciences* **74,** 1309–1331 (2008).

64. Tang, H. *et al. # Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning* in *NIPS* (2017), 2750–2759.

65. Ng, A. Y., Harada, D. & Russell, S. *Policy invariance under reward transformations: Theory and application to reward shaping* in *ICML* **99** (1999), 278–287.

66. Hussein, A., Gaber, M. M., Elyan, E. & Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)* **50,** 1–35 (2017).

67. Plappert, M. *et al.* Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464* (2018).

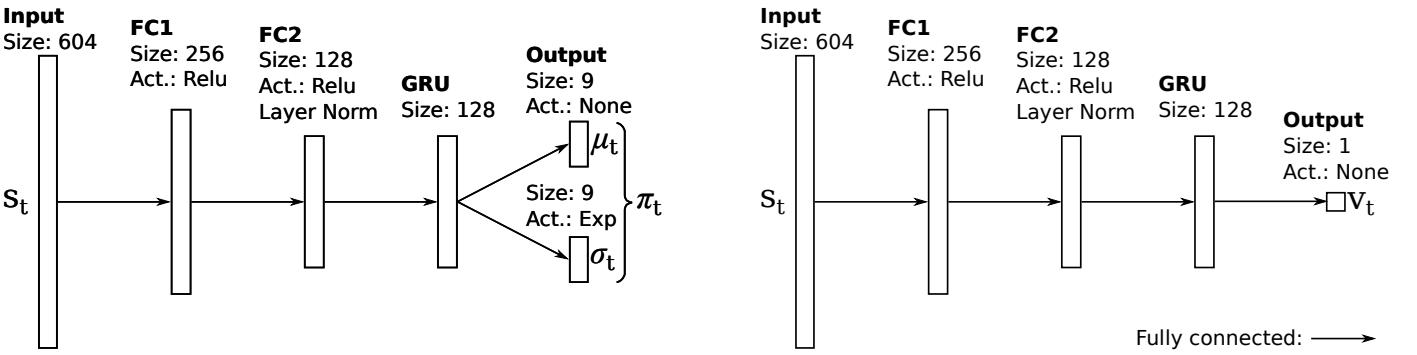**Data availability.** The data that support the findings of this study (including the raw data for all figures and tables in the manuscript, extended data and SI, as well as the demonstration trajectories used in robustification) are available from the corresponding authors upon reasonable request.

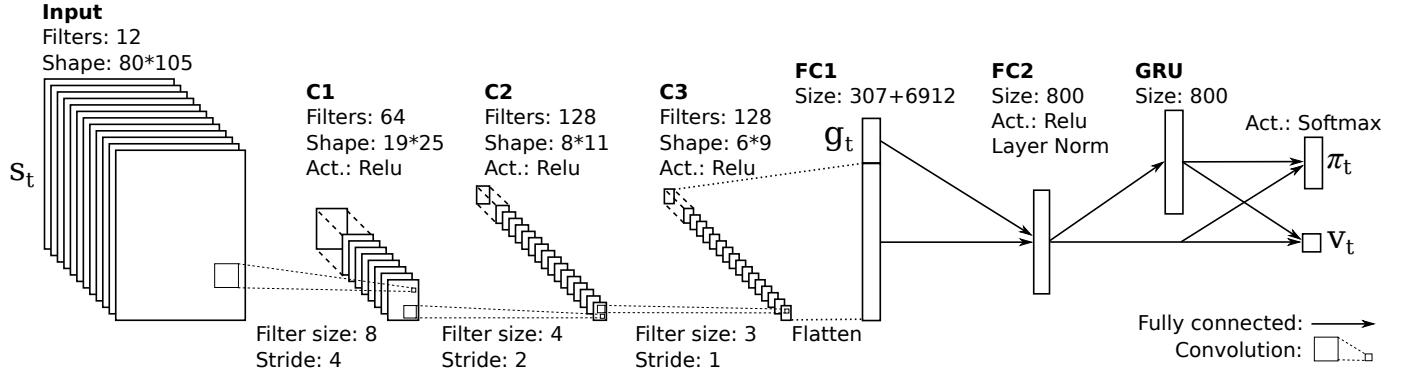**Code availability.** The Go-Explore code is available at: https://github.com/uber-research/go-explore.

a Atari architecture.

Input
Filters: 12
Shape: 80*105

S_t

C1
Filters: 64
Shape: 19*25
Act.: Relu

C2
Filters: 128
Shape: 8*11
Act.: Relu

C3
Filters: 128
Shape: 6*9
Act.: Relu

FC1
Size: 6912

FC2
Size: 800
Act.: Relu
Layer Norm

GRU
Size: 800

Act.: Softmax

π_t

V_t

Filter size: 8
Stride: 4

Filter size: 4
Stride: 2

Filter size: 3
Stride: 1

Flatten

Fully connected: ⟶
Convolution: ⬚

b Robotics architecture.

Input
Size: 604

S_t

FC1
Size: 256
Act.: Relu

FC2
Size: 128
Act.: Relu
Layer Norm

GRU
Size: 128

Output
Size: 9
Act.: None

μ_t

Size: 9
Act.: Exp

σ_t

π_t

Input
Size: 604

S_t

FC1
Size: 256
Act.: Relu

FC2
Size: 128
Act.: Relu
Layer Norm

GRU
Size: 128

Output
Size: 1
Act.: None

V_t

Fully connected: ⟶

c Policy-based Go-Explore architecture.

Input
Filters: 12
Shape: 80*105

S_t

C1
Filters: 64
Shape: 19*25
Act.: Relu

C2
Filters: 128
Shape: 8*11
Act.: Relu

C3
Filters: 128
Shape: 6*9
Act.: Relu

FC1
Size: 307+6912

g_t

FC2
Size: 800
Act.: Relu
Layer Norm

GRU
Size: 800

Act.: Softmax

π_t

V_t

Filter size: 8
Stride: 4

Filter size: 4
Stride: 2

Filter size: 3
Stride: 1

Flatten

Fully connected: ⟶
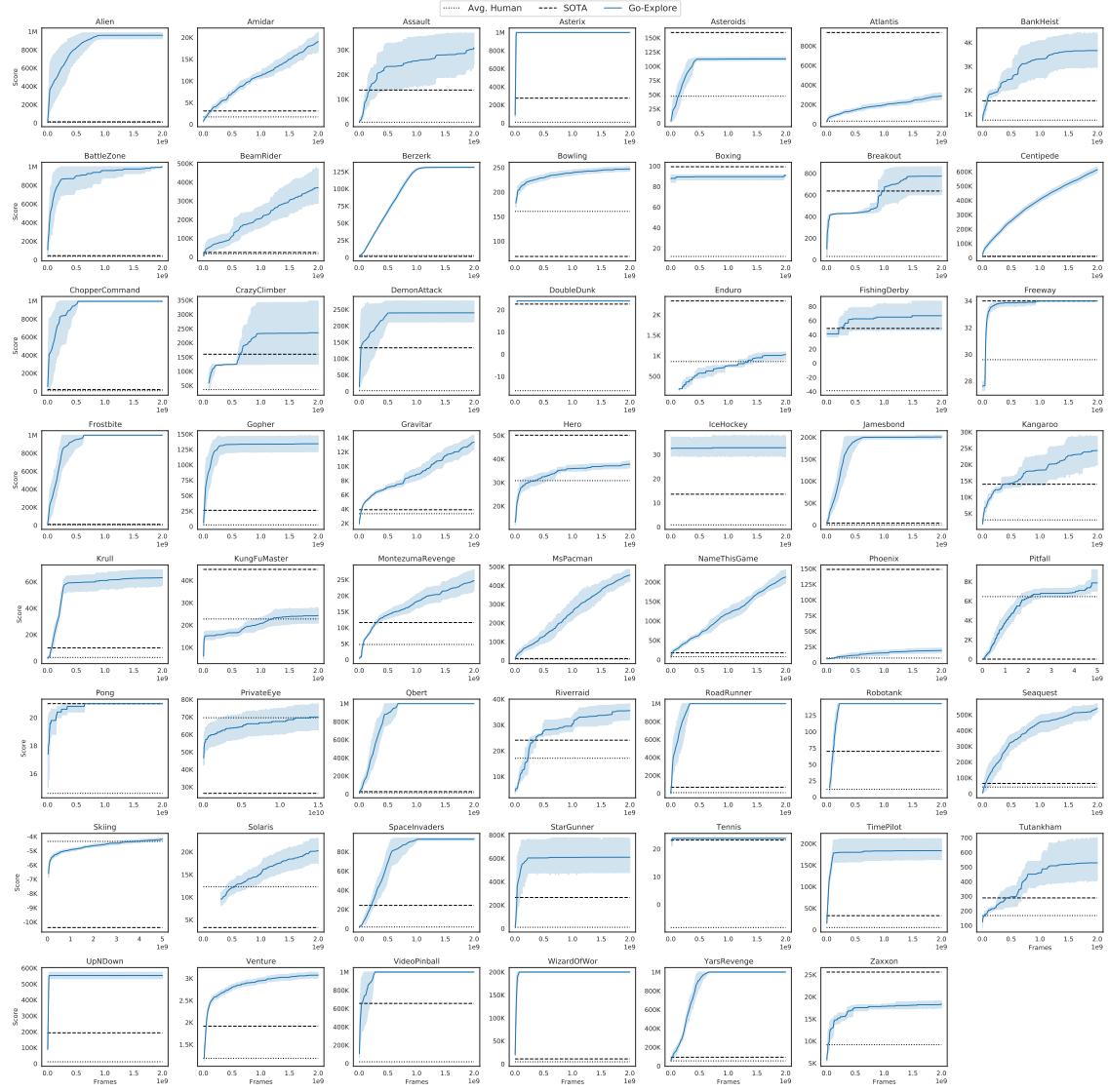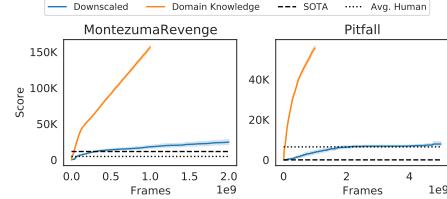Convolution: ⬚

Extended Data Figure 1: **Neural network architectures.** **(a)** The Atari architecture is based on the architecture provided with the backward algorithm implementation. The input consists of the RGB channels of the last four frames (re-scaled to 80 by 105 pixels) concatenated, resulting in 12 input channels. The network consists of 3 convolutional layers, 2 fully connected layers, and a layer of Gated Recurrent Units (GRUs)[68]. The network has a policy head $\pi(s|a)$ and a value head $V(s)$. **(b)** For the robotics problem, the architecture consists of two separate networks, each with 2 fully connected layers and a GRU layer. One network specifies the policy $\pi(s|a)$ by returning a mean μ and variance σ for the actuator torques of the arm and the desired position of each of the two fingers of the gripper (gripper fingers are implemented as Mujoco position actuators[61] with $kp = 10^4$ and a control range of $[0, 0.05]$). The other network implements the value function $V(s)$. **(c)** The architecture for policy-based Go-Explore is identical to the Atari architecture, except that the goal representation g is concatenated with the input of the first fully connected layer.

31.
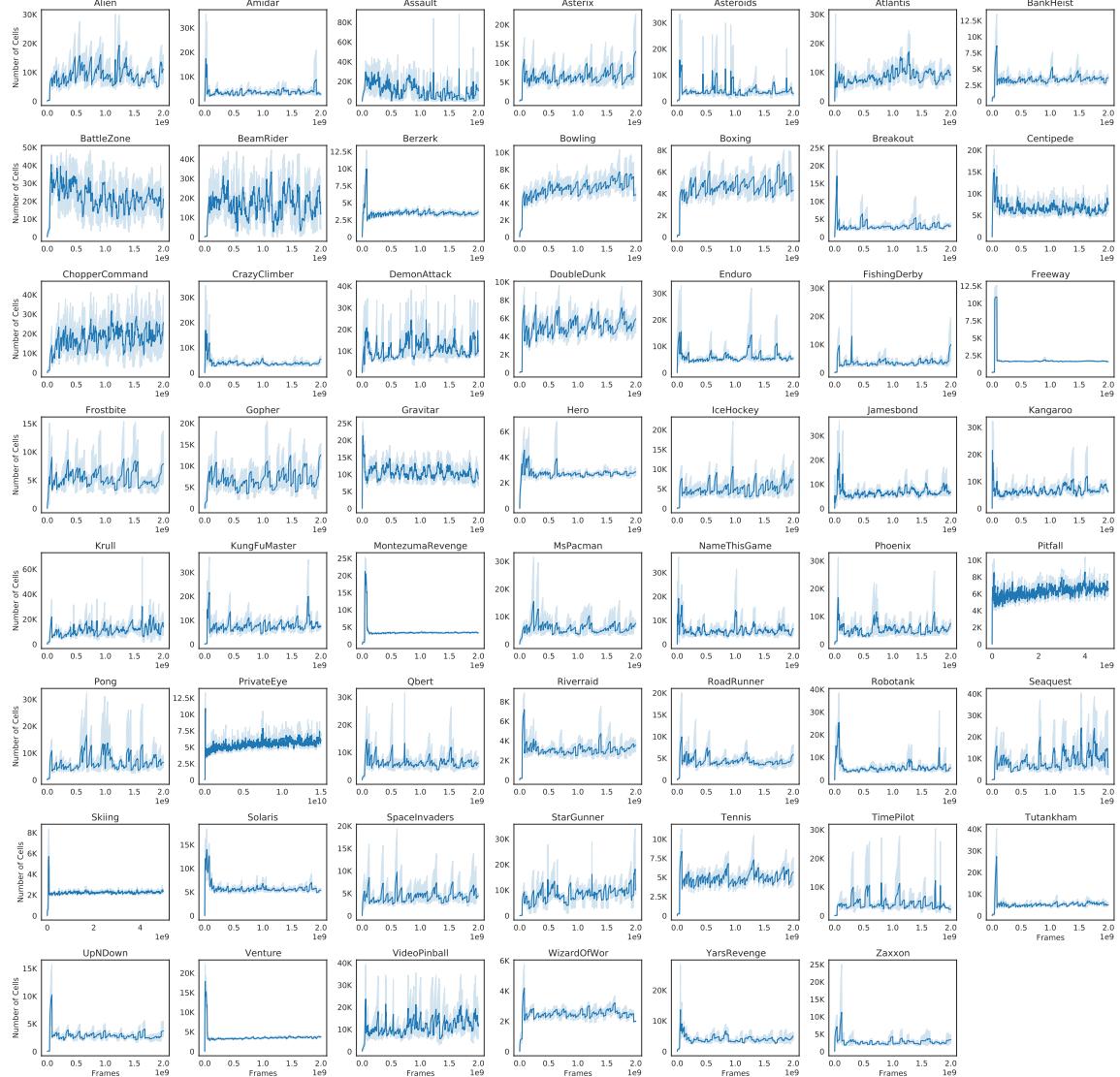
a Exploration phase without domain knowledge.

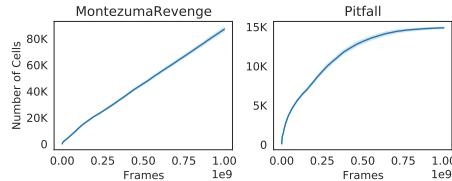**b** Exploration phase with domain knowledge (compared to downscaled).

Extended Data Figure 2: **Maximum end-of-episode score found by the exploration phase on Atari.** Because only scores achieved at the episode end are reported, the plots for some games (e.g. Solaris) begin after the start of the run, when the episode end is first reached. In (a), averaging is over 50 runs for the 11 focus games and 5 runs for other games. In (b), averaging is over 100 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

32

a Exploration phase without domain knowledge.

b Exploration phase with domain knowledge.

Extended Data Figure 3: **Number of cells in archive during the exploration phase on Atari.** In (a), archive size can decrease when the representation is recomputed. Previous archives are converted to the new format when the representation is recomputed, possibly leading to an archive larger than 50K. In this case, one iteration of the exploration phase runs and the representation is recomputed again. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.
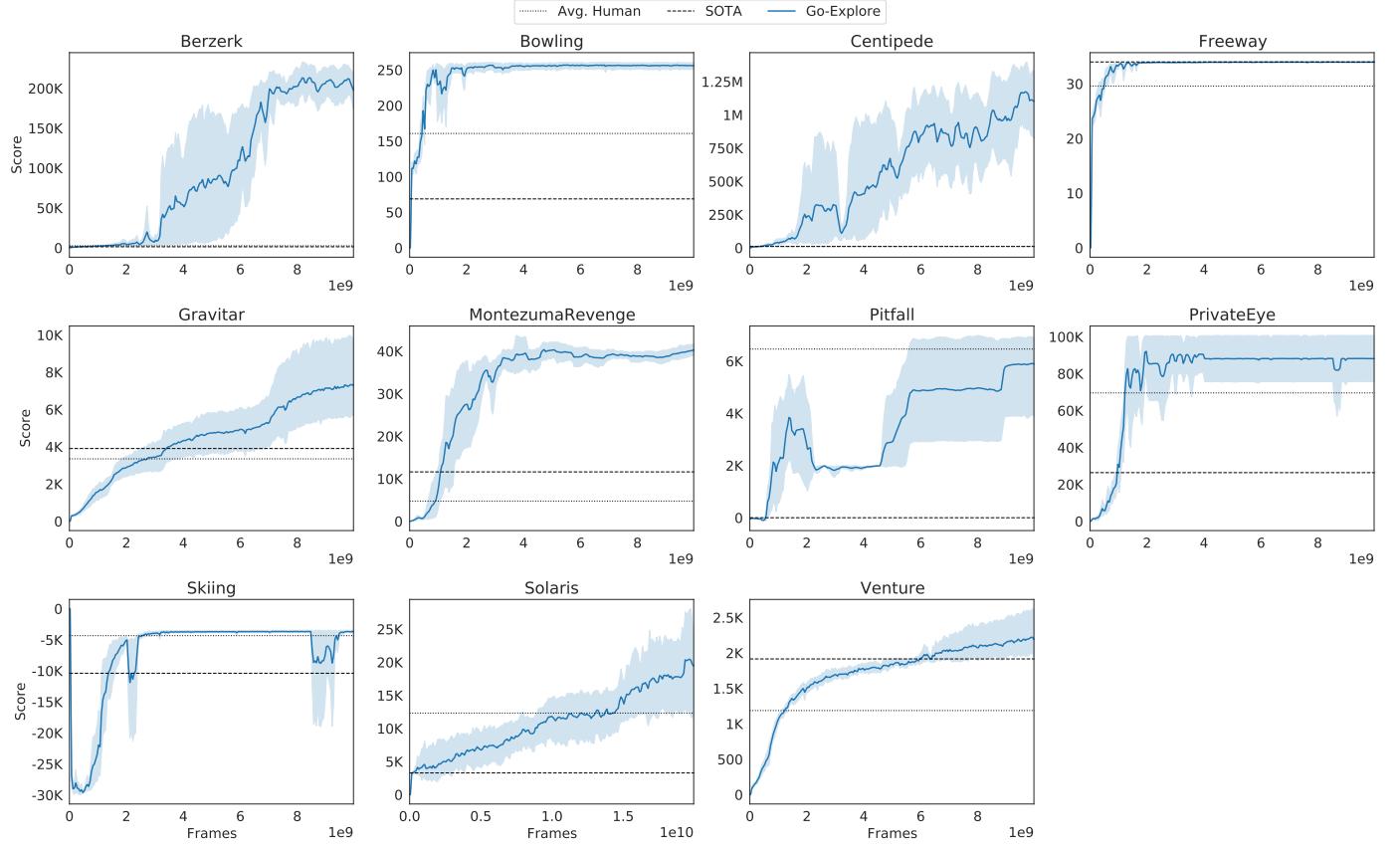
33

a Exploration phase without domain knowledge.

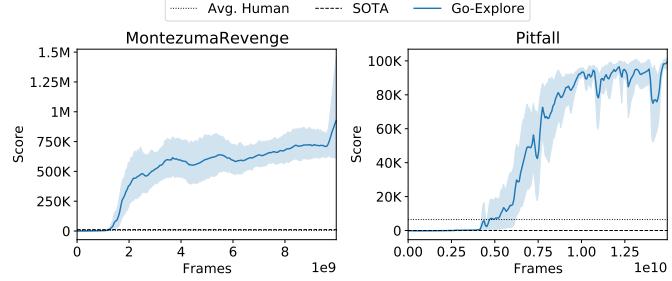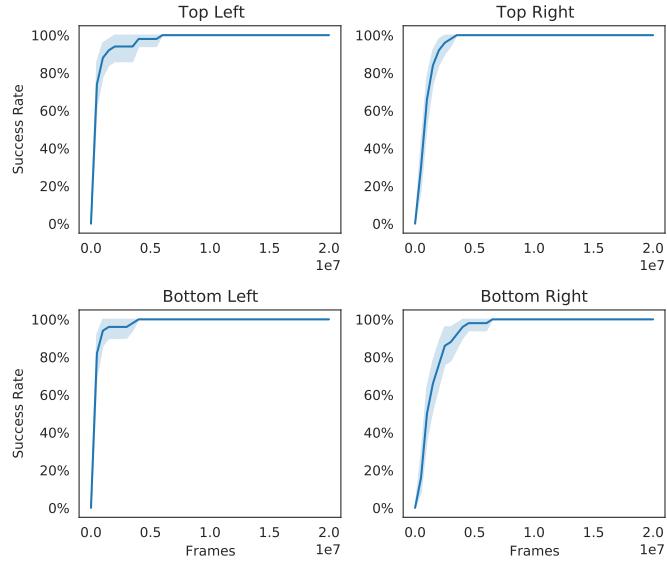

b Exploration phase with domain knowledge.
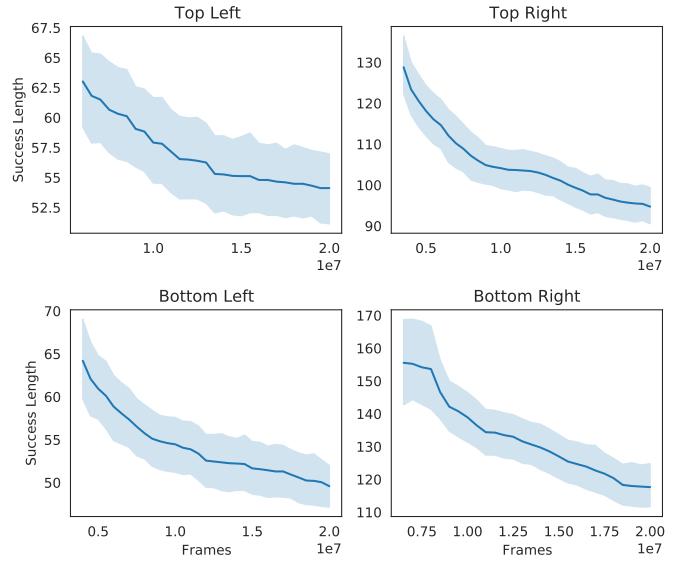
Extended Data Figure 4: **Progress of robustification phase on Atari.** Shown are the scores achieved by robustifying agents across training time for the exploration phase (a) with representations informed by domain knowledge, and (b) representations without domain knowledge. In particular, the rolling mean is shown of performance across the past 100 episodes when starting from the virtual demonstration (which corresponds to the domain's traditional starting state). Note that in (a) averaging is over 5 independent runs, while in (b) averaging is over 10 runs. Because the final performance is obtained by testing the highest-performing network checkpoint for each run over 1,000 additional episodes, rather than directly extracted from the curves above, the performance reported in Fig. 2b does not necessarily match any particular point along these curves (Methods). Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

34
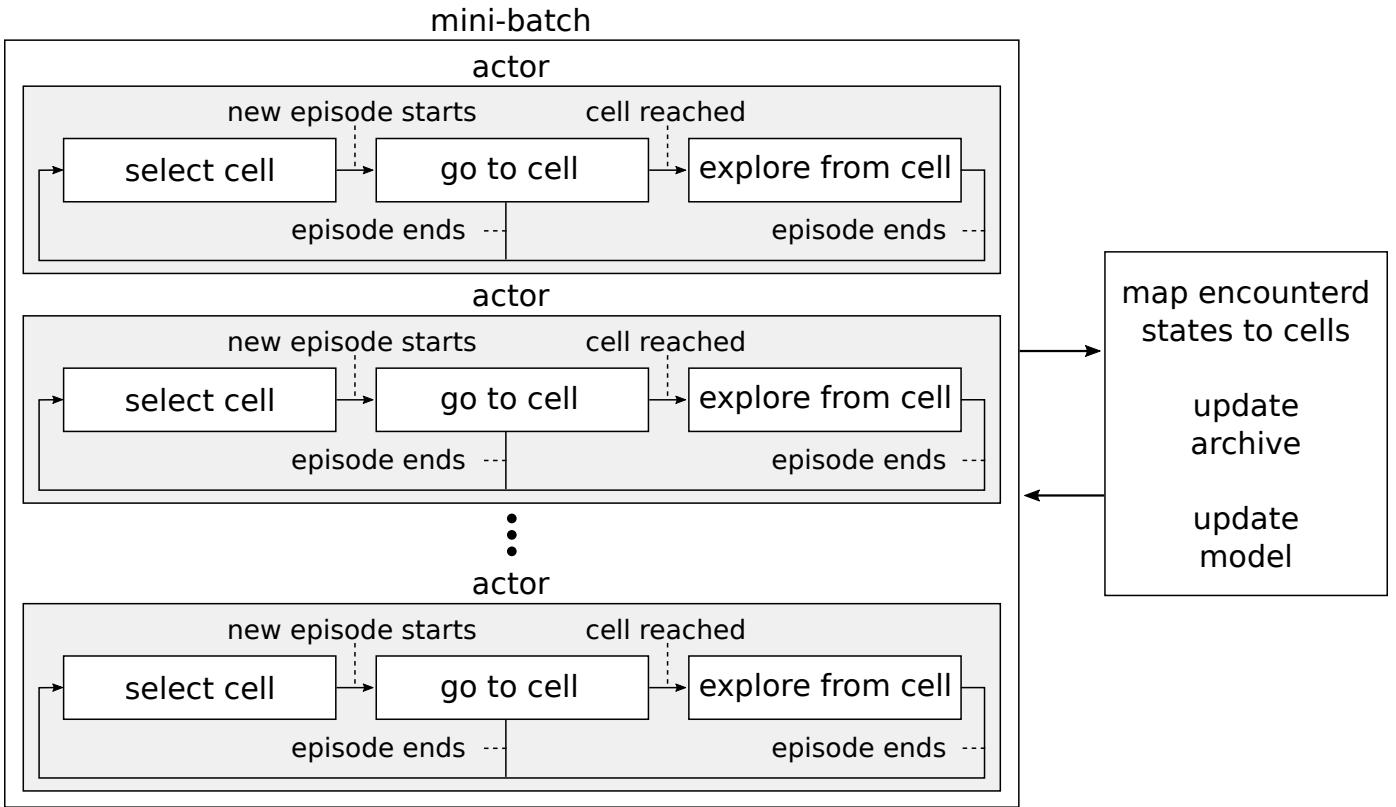
a Runs with successful trajectories.

b Length of the shortest successful trajectory
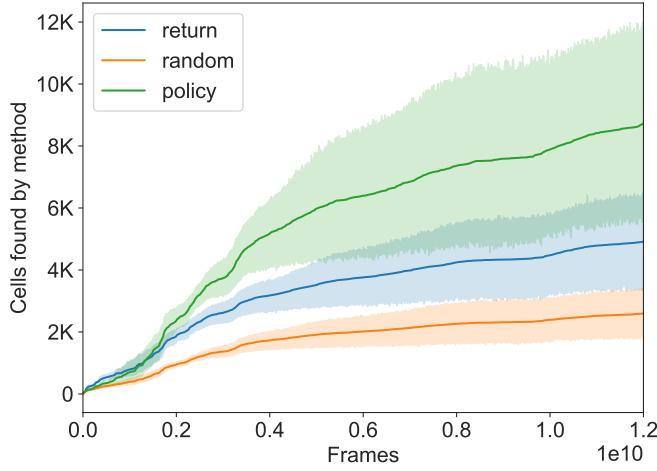
Extended Data Figure 5: **Progress of the exploration phase in the robotics environment.** In (a), the exploration phase quickly achieves 100% success rate for all shelves in the robotics environment. However, (b) shows that while success is achieved quickly, it is useful to keep the exploration phase running longer to reduce the length of the successful trajectories, thus making robustification easier. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.
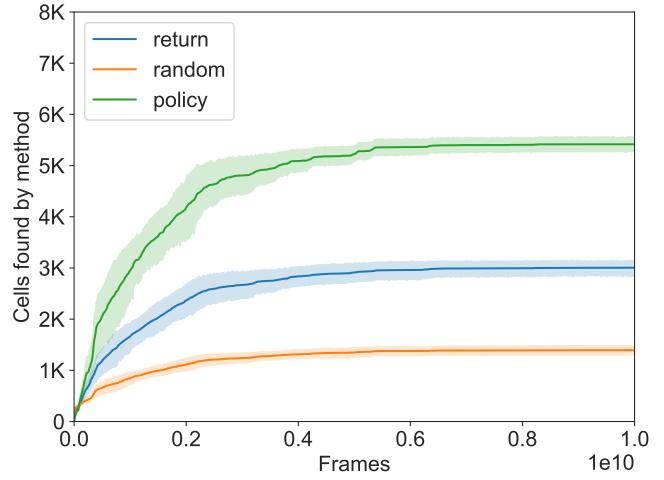
Extended Data Figure 6: **Policy-based Go-Explore overview.** With respect to their practical implementation, the main difference between policy-based Go-Explore and Go-Explore when restoring simulator state is that in policy-based Go-Explore there exist separate actors that each have an internal loop switching between the "select", "go", and "explore" steps, rather than one outer loop in which the "select", "go", and "explore" steps are executed in synchronised batches. This structure allows policy-based Go-Explore to be easily combined with popular RL algorithms like A3C[19], PPO[20] or DQN[15], which already divide data gathering over many actors.

a Montezuma's Revenge

b Pitfall

Extended Data Figure 7: **Method by which cells are found.** In both **(a)** Montezuma's Revenge and **(b)** Pitfall, sampling from the goal-conditioned policy results in the discovery of roughly four times more cells than when taking random actions. At the start of training there is effectively no difference between random actions and sampling from the policy, supporting the intuition that sampling from the policy only becomes more efficient than random actions after the policy has acquired the basic skills for moving towards the indicated goal. Lastly, the number of cells that are discovered while returning is about twice that of the cells discovered when taking random actions after returning, indicating that the frames spent while returning to a previously visited cell are not just overhead required for moving towards the frontier of yet undiscovered states and training the policy network, but actually provide a substantial contribution towards exploration as well. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

a Parameters for PPO, SIL, and the backward algorithm.

	Robustification		Policy-Based
Parameter	*Atari*	*Robotics*	*Atari*
Discount factor (γ)	0.999	0.99	0.99
N-step return factor (λ)	0.95	0.95	0.95
Num. workers	8	8	16
Num. actors per worker	32 + 2 SIL	120 + 8 SIL	15 + 1 SIL
Num. actors (N)	256 + 16 SIL	960 + 64 SIL	240 + 16 SIL
Steps per batch (T)	128	128	128
PPO Clip (ϵ)	0.1	0.1	0.1
PPO Epochs	4	4	4
Value coef. (w_{VF})	0.5	0.5	0.5
Ent. coef. (w_{ENT})	10^{-5}	10^{-5}	10^{-4}
L2 coef. (w_{L2})	10^{-7}	10^{-7}	10^{-7}
SIL coef. (w_{SIL})	0.1	0.1	0.1
SIL ent. coef. (w_{SIL_ENT})	10^{-5}	10^{-5}	0
SIL value coef. (w_{SIL_VF})	0.01	0.1	0.01
Allowed lag	50	10	-
Extra frame coef	7	4	-
Move threshold	0.1	0.1	-
Num. demonstrations	10 + 1 virtual	10	-
SIL from start prob.	0.3	0	-
Window size (frames)	160	40	-

b Atari environment parameters

Parameter	Value
Sticky actions	True
Length limit	400K frames[*]
End of episode	All lives lost[†]
Action repeat (i.e. frame skip)	4
Frame max pool	2 or 4[‡]

Extended Data Table 1: **Hyperparameters.** (a) Parameters above the dividing line are applicable to PPO with SIL, while parameters below the line are specific to the backward algorithm. "Allowed lag" is the number of frames the agent may lag the demonstration before being considered unsuccessful. When the agent matches the demonstration, it runs for additional frames, controlled by "Extra frame coef" c: $\lfloor e^{cX} \rfloor$ ($X \sim U(0, 1)$). Window size is the number of starting points below the maximum starting point of the demonstration that the algorithm may start from. (b) For the exploration phase when restoring simulator state, only "Max episode length", "End of episode", and "Action repeat" apply. (*) OpenAI Gym default. (†) Except for Montezuma's Revenge with a return policy (Methods "Policy-based Go-Explore"). (‡) 4 for Gravitar and Venture.

a Position and velocity objects	b Collision and bounding box objects
Object	**Object**
door1	DoorLR
door	DoorUR
elbow_flex_link	Shelf
forearm_roll_link	Table
gripper_link	door1
head_camera_link	door
head_pan_link	frameL1
head_tilt_link	frameL
l_gripper_finger_link	frameR1
latch1	frameR
latch	gripper_link
obj0	l_gripper_finger_link
r_gripper_finger_link	latch1
shoulder_lift_link	latch
shoulder_pan_link	obj0
upperarm_roll_link	r_gripper_finger_link
wrist_flex_link	world
wrist_roll_link	

Extended Data Table 2: **Robotics state representation.** Position and velocities of the objects in (a) are included in the state representation for robotics. Collisions between any two objects in (b) as well as whether each object is currently inside the bounding boxes for the table and shelves are also included in the state representation. Objects are given by their MuJoCo[61] entity names in the source code for the environment. Door-related objects ending with a 1 correspond to the lower door while door-related objects not ending with anything correspond to the upper door. The frame objects are the unmovable wooden blocks situated on either side of the movable part of the door. "L" and "R" correspond to "left" and "right" whereas "L" and "U" correspond to "lower" and "upper". The difference between "door" and "DoorUR" as well as "door1" and "DoorLR" is that in each case the latter object corresponds to the entire door structure, including the frames, while the former corresponds only to the movable part of the door. A link to the original source code for the MuJoCo description files defining these entities is given in "Acknowledgements", and a link to the Go-Explore codebase containing our modified version is provided in Methods "Code availability".

Game	Expl. Phase	Robust. Phase	SOTA	Avg. Human	Agent57
Alien	959,312		11,358	7,128	297,638
Amidar	19,083		3,092	1,720	29,660
Assault	30,773		13,759	742	67,213
Asterix	999,500		274,491	8,503	991,384
Asteroids	112,952		159,426	47,389	150,855
Atlantis	286,460		937,558	29,028	1,528,842
BankHeist	3,668		1,563	753	23,072
BattleZone	998,800		45,610	37,188	934,135
BeamRider	371,723		24,031	16,927	300,510
Berzerk	131,417	**197,376**	1,383	2,630	61,508
Bowling	247	**260**	69	161	251
Boxing	91		99	12	100
Breakout	774		637	31	790
Centipede	613,815	**1,422,628**	10,166	12,017	412,848
ChopperCommand	996,220		19,256	7,388	999,900
CrazyClimber	235,600		160,161	35,829	565,910
DemonAttack	239,895		133,030	1,971	143,161
DoubleDunk	24		23	-16	24
Enduro	1,031		2,338	861	2,368
FishingDerby	67		49	-39	87
Freeway	34	**34**	**34**	30	33
Frostbite	999,990		10,003	4,335	541,281
Gopher	134,244		26,123	2,413	117,777
Gravitar	13,385	**7,588**	3,906	3,351	19,214
Hero	37,783		50,142	30,826	114,736
IceHockey	33		14	1	64
Jamesbond	200,810		4,303	303	135,785
Kangaroo	24,300		13,982	3,035	24,034
Krull	63,149		9,971	2,666	251,997
KungFuMaster	24,320		44,920	22,736	206,846
MontezumaRevenge	24,758	**43,791**	11,618	4,753	9,352
MsPacman	456,123		9,901	6,952	63,994
NameThisGame	212,824		18,084	8,049	54,387
Phoenix	19,200		148,840	7,243	908,264
Pitfall	7,875	**6,954**	0	6,464	18,756
Pong	21		21	15	21
PrivateEye	69,976	**95,756**	26,364	69,571	79,716
Qbert	999,975		26,172	13,455	580,328
Riverraid	35,588		24,116	17,118	63,319
RoadRunner	999,900		67,962	7,845	243,026
Robotank	143		70	12	127
Seaquest	539,456		64,985	42,055	999,998
Skiing	-4,185	**-3,660**	-10,386	-4,337	-4,203
Solaris	20,306	**19,671**	3,282	12,327	44,200
SpaceInvaders	93,147		24,183	1,669	48,681
StarGunner	609,580		265,480	10,250	839,574
Tennis	24		23	-8	24
TimePilot	183,620		32,813	5,229	405,425
Tutankham	528		288	168	2,355
UpNDown	553,718		193,520	11,693	623,806
Venture	3,074	**2,281**	1,916	1,188	2,624
VideoPinball	999,999		656,572	17,668	992,341
WizardOfWor	199,900		10,980	4,757	157,306
YarsRevenge	999,998		93,680	54,577	998,532
Zaxxon	18,340		25,603	9,173	249,809

Extended Data Table 3: **Full scores on Atari.** Go-Explore outperforms SOTA on all focus games (Freeway's score is at its maximum). The exploration phase similarly finds trajectories that frequently exceed SOTA scores. Finally, Go-Explore outperforms Agent57 on 7 of the 11 focus games, despite Go-Explore being evaluated in a harder environment. Agent57 was included because it is the only other algorithm that has achieved superhuman scores on all unsolved and hard-exploration games, but it is listed separately because it was evaluated under easier, mostly deterministic conditions (Methods "State of the art on Atari").

Supplementary Information

1 Algorithms This section presents pseudo-code for the algorithms presented in the main text.

Algorithm 1 Exploration Phase with Simulator Restoration.

1: **Input**: The archive sampling batch size K, the rollout length L, the starting state startingState, the initial representation parameters rParams

2: **(Optional) Input for dynamic representations**: probability of adding frame to recent frames sample p_s, max archive size M, initial number of frames before recomputing representations F, interval between recomputing representations I

3: Initialise archive $\mathcal{A} \leftarrow \{\}$ ▷ $\{\}$ represents an empty dictionary
4: $\mathcal{A}[\text{REPRESENTATION}(\text{startingState}, \text{rParams})] \leftarrow \text{startingState}$
5: Initialise seen counts $\mathcal{C} \leftarrow \{\}$
6: Initialise recent frame sample set $\mathcal{S} \leftarrow \{\}$
7: frameCount $\leftarrow 0$
8: **while** true **do**
9:      **if** using dynamic representation and ($|\mathcal{A}| > M$ or frameCount $> F$) **then**
10:         rParams $\leftarrow$ RECOMPUTEREPRESENTATION($\mathcal{S}$)   ▷ Compute representation parameters based on the sample
11:         $\mathcal{S} \leftarrow \{\}$
12:         $F \leftarrow$ frameCount $+ I$
13:     **end if**
14:     Sample batch of starting cells $\mathcal{B} \leftarrow$ WEIGHTEDSAMPLE($\mathcal{A}, \mathcal{C}, K$)
15:     **for** Cell s in $\mathcal{B}$ **do**
16:         Restore environment state to s
17:         seen $\leftarrow \{\}$
18:         **for** i in 0,...,L or until DONE **do**
19:             $a \leftarrow$ RANDOMACTION()   ▷ Random action implements action repetition
20:             Take action a and receive state s_i
21:             frameCount $\leftarrow$ frameCount $+ 1$
22:             repr $\leftarrow$ REPRESENTATION(s_i, rParams)   ▷ If s_i is a done state, returns $DONE$
23:             **if** repr not in $\mathcal{A}$ or ISBETTER(s_i, $\mathcal{A}[\text{repr}]$) **then**   ▷ ISBETTER checks if s_i has a better score or an equal score but shorter trajectory
24:                 $\mathcal{A}[\text{repr}] \leftarrow s_i$
25:             **end if**
26:             **if** repr not in seen **then**   ▷ $\mathcal{C}[\text{repr}]$ is the number of rollouts in which repr was seen, so it can only increase once per rollout
27:                 $\mathcal{C}[\text{repr}] \leftarrow \mathcal{C}[\text{repr}] + 1$
28:                 Add repr to seen
29:             **end if**
30:             **if** dynamic representation and RAND() $< p_s$ **then**
31:                 Add s_i's frame to $\mathcal{S}$ ▷ When $\mathcal{S}$ is at max size, oldest frame is evicted
32:             **end if**
33:         **end for**
34:     **end for**
35: **end while**

Algorithm 2 Robustification Phase Rollout Worker.

1: **Input**: A set of demonstrations $\mathcal{D}$, each a list of tuples $(s_t, a_t, r_t, \text{done}_t)$; current starting point S_{demo} for each demonstration demo; the number of steps to initialise the RNN state K; the allowed lag of the agent compared to the demonstration L; the extra frames coefficient C; the current policy π, which is updated by the PPO optimiser process in the background

2: $\text{frameCountsDemo} \leftarrow [\,]$ $\triangleright$ List of number of frames in episodes in which a demo was used
3: $\text{frameCountsVirtual} \leftarrow [\,]$ $\triangleright$ List of number of frames processed in episodes with the virtual (NULL) demo
4: $\text{done} \leftarrow \text{true}$
5: **while** true **do**
6:     **if** done **then**
7:        $w_d \leftarrow |\mathcal{D}|/\text{MEAN}(\text{frameCountsDemo})$ **if** frameCountsDemo is not empty **else** 1
8:        $w_v \leftarrow 1/\text{MEAN}(\text{frameCountsVirtual})$ **if** frameCountsVirtual is not empty **else** 1
9:        $p_v \leftarrow \frac{w_v}{w_d + w_v}$
10:        **if** using virtual demonstration and $\text{RAND}() < p_v$ **then**
11:           $\text{demo} \leftarrow \text{NULL}$
12:           $i \leftarrow 0$
13:           $s_i \leftarrow \text{RESETENV}()$ $\triangleright$ Performs no-ops if appropriate
14:        **else**
15:           $\text{demo} \leftarrow \text{RANDCHOICE}(\mathcal{D})$
16:           $i \leftarrow \max(0, S_{\text{demo}} - K)$ $\triangleright$ Set the current timestep to the demo starting point minus the number of RNN initialisation steps
17:           $s_i \leftarrow \text{RESTOREDEMO}(\text{demo}, i)$ $\triangleright$ Restore the environment to step i in the demo and return the corresponding state
18:           $\text{score} \leftarrow \text{DEMOSCORE}(\text{demo}, i)$
19:           $\text{remainingFrameCount} \leftarrow |\text{demo}| - i + e^{C \cdot \text{RAND}()}$ $\triangleright$ Give $e^{C \cdot \text{RAND}()}$ extra training frames when the policy matches the demo performance
20:        **end if**
21:     **end if**
22:     **if** demo is NULL or $i \geq S_{\text{demo}}$ **then**
23:        Sample $a_i \sim \pi(s_i, \theta)$
24:        $m_i \leftarrow \text{true}$ $\triangleright$ The transition can be used in training
25:     **else**
26:        $a_i \leftarrow \text{DEMOACTION}(d, i)$ $\triangleright$ Replaying demo data to warm-up the RNN policy
27:        $m_i \leftarrow \text{false}$ $\triangleright$ The transition should be masked out in training
28:     **end if**
29:     $s_i, r_i, s_{i+1}, \text{done} \leftarrow \text{STEP}(a_i, \text{stochastic}=m_i)$ $\triangleright$ Replay transitions cannot be stochastic
30:     $\text{score} \leftarrow \text{score} + r_i$
31:     Send the transition $(s_i, a_i, r_i, s_{i+1}, \text{done}, m_i)$ to the optimizer
32:     **if** demo is not NULL and not done **then**
33:        $\text{remainingFrameCount} \leftarrow \text{remainingFrameCount} - 1$
         $\triangleright$ With negative rewards we must consider all demo scores from frame $i - L$ to $i + L$ to check for lag
34:        $\text{lagging} \leftarrow \text{score} < \min(\text{DEMOSCORES}(\text{demo}, i - L, i + L))$
35:        $\text{done} \leftarrow \text{remainingFrameCount} < 0$ or lagging
36:     **end if**
37:     $i \leftarrow i + 1$
38:     **if** done and demo is not NULL **then**
39:        Append $i - \max(0, S_{\text{demo}} - K)$ to frameCountsDemo
         $\triangleright$ The optimiser modifies S_{demo} based on the success rate following the logic in Salimans & Chen (2018)
40:        Report a success for demo at S_{demo} if $\text{score} \geq \text{DEMOSCORE}(\text{demo}, |\text{demo}|)$ or a failure otherwise
41:     **else if** done **then**
42:        Append i to frameCountsVirtual
43:     **end if**
44: **end while**

Algorithm 3 Policy-based Go-Explore.

1: **Input**: The environment E, initial state startingState, the reward for reaching the final goal r^G, the reward for reaching an intermediate goal r^g, the maximum steps without progress $\hat{T}$, and initial policy parameters θ

2: Initialise archive $\mathcal{A} \leftarrow \{\}$ ▷ $\{\}$ represents an empty dictionary
3: $\mathcal{A}[\text{REPRESENTATION}(\text{startingState})] \leftarrow (\text{reward} = 0, \text{visits} = 0, \text{trajectory} = [\,])$
4: **while** true **do**
5:     $G \leftarrow \text{SELECTCELL}(\mathcal{A})$ ▷ Selects final goal from archive according to visit counts
6:     $\tau \leftarrow \mathcal{A}[G].\text{trajectory}$
7:     $\text{mode} \leftarrow \text{return}$
8:     $s \leftarrow \text{RESET}(E)$ ▷ Resets the environment and obtains the initial state
9:     $c \leftarrow \text{REPRESENTATION}(s)$ ▷ Extracts cell representation from state
10:     $\text{done} \leftarrow \text{False}$
11:     $\tau' \leftarrow [\,]$ ▷ τ' tracks the cell trajectory for this episode
12:     $R \leftarrow 0$ ▷ R tracks the total undiscounted reward for this episode
13:     $e \leftarrow 1$ ▷ Initialise entropy to 1
14:     $\hat{t} \leftarrow 0$ ▷ Initialise time-out counter to 0
15:     **while** not done **do**
16:        **if** mode = return **then**
17:           $g \leftarrow \text{GETNEXTGOAL}(\tau, c)$ ▷ Determines next goal based on soft-trajectory
18:        **else if** mode = exploreFromPolicy **then**
19:           $g \leftarrow \text{GETEXPLORATIONGOAL}(\mathcal{A}, c, g)$ ▷ Goal updates when reached or after 100 steps
20:        **end if**
21:        **if** mode = exploreRandom **then**
22:           $a \leftarrow \text{RANDOMACTION}()$ ▷ Random action implements action repetition
23:        **else**
24:           Sample $a \sim \pi_\theta(s, g, e)$ ▷ Logits of $\pi_\theta(s, g)$ are divided by entropy e before softmax is applied
25:        **end if**
26:        $s, r^e, \text{done} \leftarrow \text{STEP}(E, a)$ ▷ Returns new state, environment reward, and episode end flag
27:        $c \leftarrow \text{REPRESENTATION}(s); \hat{t} \leftarrow \hat{t} + 1$
28:        $r^\tau \leftarrow 0$ ▷ r^τ is the trajectory reward
29:        **if** mode = return **then**
30:           **if** $c = G$ **then**   ▷ The final goal was reached
31:              $r^\tau \leftarrow r^G; \hat{t} \leftarrow 0$
32:              $\text{mode} \leftarrow \text{RANDSELECT}(\text{exploreRandom}, \text{exploreFromPolicy})$
33:           **else if** $c = g$ **then**   ▷ The current sub-goal was reached
34:              $r^\tau \leftarrow r^g; \hat{t} \leftarrow 0$
35:           **end if**
36:        **else if** $c \notin \mathcal{A}$ **then**
37:           $\hat{t} \leftarrow 0$
38:        **end if**
39:        $e \leftarrow \text{UPDATEENTROPY}(\hat{t})$ ▷ See equation 8
40:        $\text{done} \leftarrow \text{done} \vee \hat{t} \geq \hat{T}$ ▷ Terminate early if no progress is made for too long
41:        $\tau' \leftarrow \text{APPEND}(\tau', c); R \leftarrow R + r^e$
       ▷ In practice, the steps below are performed in batches to enable parallelization
42:        $\theta \leftarrow \text{UPDATEMODEL}(\theta, s, a, r^e, r^\tau)$ ▷ θ updated following standard PPO procedure
43:        **if** $c \notin \mathcal{A}$ or $\text{BETTER}((R, \tau'), \mathcal{A}[c])$ **then**   ▷ Cell is added to archive if new, higher reward, or shorter
44:           $\mathcal{A}[c].\text{reward} \leftarrow R; \mathcal{A}[c].\text{trajectory} \leftarrow \tau'$
45:        **end if**
46:        $\mathcal{A}[c].\text{visits} \leftarrow \mathcal{A}[c].\text{visits} + 1$
47:     **end while**
48: **end while**

2 Prior work on Montezuma's Revenge Supplementary Table 1 provides the referenced list of algorithms shown in Figure 2a of the main text. In cases where the work introducing the algorithm was first released as a pre-print and later formally published, the date of the pre-print is used (to better convey the historical sequencing), but the published version is given in the references.

3 Ablations For the ablations and parameter analyses presented in this section, data comes from the main experiment where available. Other than the parameter or setting being varied, the only

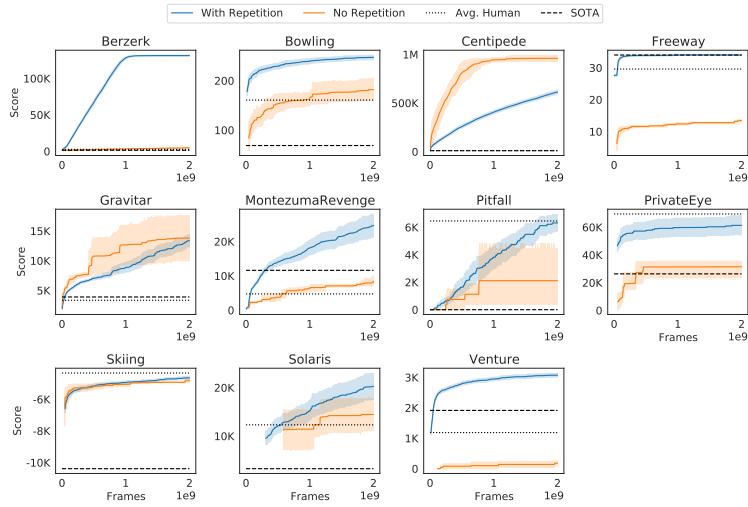| Algorithm	Time of publication	Score
2BFS[17]	Jul 2015	540
A3C-CTS[1]	Jun 2016	1,127
A3C[19]	Feb 2016	67
Agent57[13]	Apr 2020	9,352
Ape-X[27]	Mar 2018	2,500
BASS[64]	Nov 2016	238
Brute[69]	Jul 2015	2,500
C51[70]	Jul 2017	0
DDQN[71]	Sep 2015	42
DeepCS[72]	Jun 2018	3,500
DQN-CTS[1]	Apr 2017	4,638
DQN-PixelCNN[55]	Jun 2016	3,705
DQN[15]	Mar 2017	2,514
DTSIL[34]	Feb 2015	50
Duel. DQN[73]	Nov 2015	22
ES[74]	Mar 2017	0
Feature-EB[56]	May 2017	2,745
Gorila[75]	Jul 2015	84
IMPALA[28]	Feb 2018	0
Linear[26]	Jul 2012	10.7
MIME[76]	Jan 2020	7,000*
MP-EB[77]	Jul 2015	0
MuZero[78]	Nov 2019	57
NGU[79]	Feb 2020	16,800*
Pellet[80]	Jul 2019	2500
POER[81]	May 2019	7,000*
Pop-Art[30]	Feb 2016	0
PPO+CoEX[51]	Nov 2018	11,540
Prior. DQN[82]	Nov 2015	13
R2D2[83]	Oct 2018	2,061
Rainbow[84]	Oct 2017	154
Reactor[85]	Apr 2017	2,643.5
RND[50]	Oct 2018	11,347
SARSA[86]	Jul 2012	259
SIL[37]	Jun 2018	2,500
UBE[87]	Sep 2017	2,750*

Supplementary Table 1: **Scores on Montezuma's Revenge for the algorithms shown in Figure 2a of the main paper.** Scores marked with an asterisk were estimated from a graphical representation.

difference between the main experiment and its ablations is that the main experiment sometimes includes a larger number of independent runs. The number of runs for each treatment is specified in the caption of the corresponding figure.
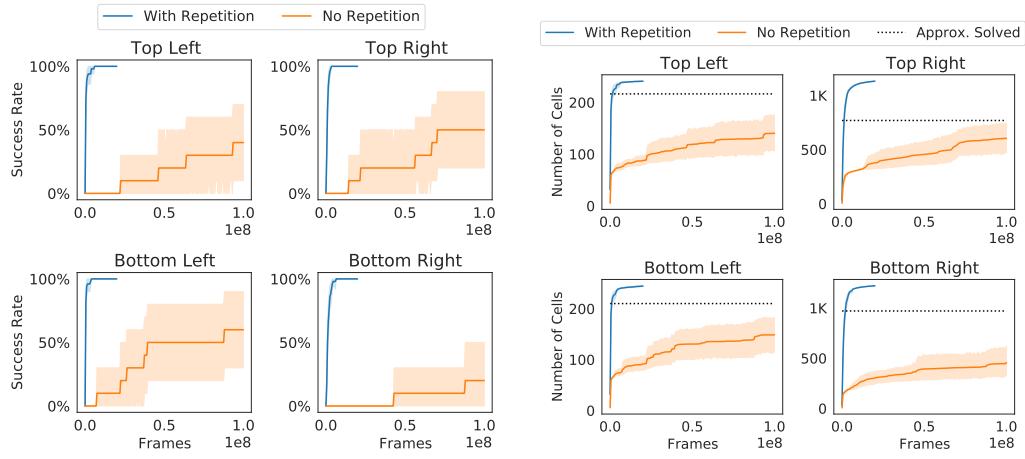
3.1   Exploration phase without action repetition   As explained in Methods "Exploration phase", actions are repeated with high probability during the exploration step. Action repetition allows agents to explore in a well-defined direction instead of dithering in place, analogously to existing temporally correlated exploration methods[88,89]. Indeed, nearly all deep reinforcement learning methods perform such directed exploration by default due to the correlation of neural network outputs across nearby frames. Supplementary Fig. 1 shows that both Atari (in most games) and robotics benefit significantly from action repetition, but that in both cases Go-Explore compares favourably to the state of the art and the PPO control without action repetition.

3.2   Exploration phase without dynamic representations   As detailed in Methods "Downscaling on Atari", the downscaling parameters for the variant of Go-Explore without domain knowledge are updated dynamically (i.e. automatically) throughout exploration for each game by Go-Explore. Supplementary Fig. 2 compares dynamically discovered representations to a fixed representation optimised for Montezuma's Revenge. The fixed representation performs well on Montezuma's Revenge, Centipede and possibly Private Eye (where the drop in performance is not statistically significant), but fails to generalise to other games, often catastrophically so. This reduction in performance is due to the two pathologies described in Methods "Downscaling on Atari": producing an excessive number of cells (Berzerk and Solaris) or producing too few cells (Bowling, Freeway, Gravitar, Pitfall and Venture). It may not be obvious from Supplementary Fig. 2b that the fixed representation in Pitfall produces too few cells, but the regular changes in representation in the dynamic variant increase the effective number of cells over time beyond the number at any given time.

Per the evaluation method (Methods "Evaluation"), scores in Supplementary Fig. 2a are recorded at the end of episodes. In Solaris with fixed downscaling, so many cells are produced close to the starting point that exploration focuses on the start of the game and never produces trajectories long enough to reach the end of episode (i.e. the agent never makes it to a place in the game where it can die), resulting in no line being shown. The maximum score regardless of episode end averages to 744 (CI: 640 – 888), far less than with dynamic downscaling. In Venture with fixed downscaling, so few cells are created and rewards are so sparse that trajectories have difficulty being extended (as longer trajectories are only produced if they lead to a new cell or a higher total score), so that
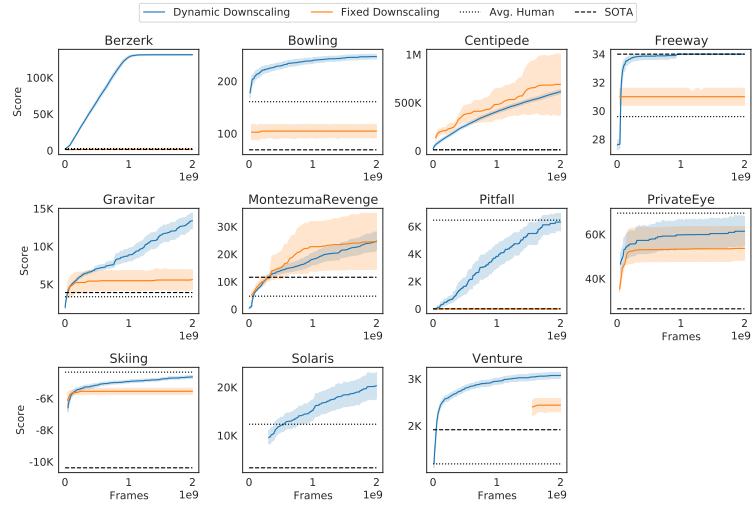
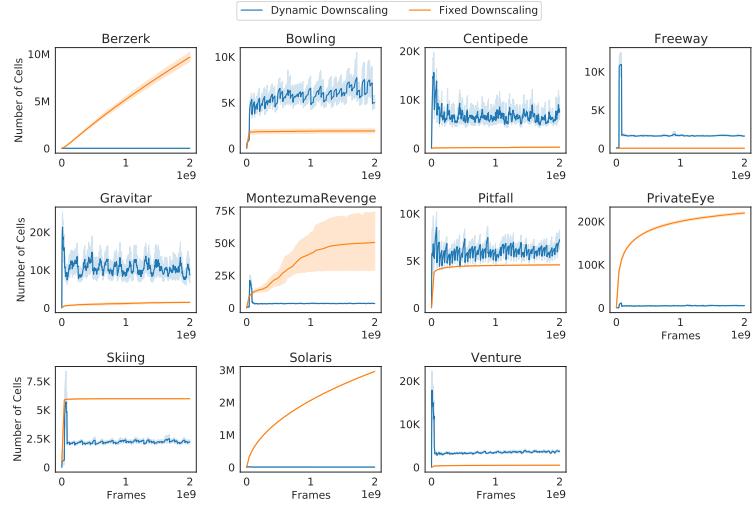(a) Exploration phase scores in the 11 Atari focus games.

(b) Exploration phase success rates in the robotics environment.

(c) Cells discovered by the exploration phase in the robotics environment.

Supplementary Figure 1: **Exploration phase performance with and without action repetition**. In Atari, action repetition is generally helpful or neutral, with the notable exception of Centipede, in which it significantly hurts exploration phase performance (though performance on Centipede is very high even with action repetition). In the robotics environment, the benefit of action repetition is large, as the variant without action repetition does not reach a 100% success rate after 100 million frames, 5 times more than was given to the variant with repetition. Though action repetition has a large positive effect on many Atari games and in robotics, even without repetition the exploration phase surpasses state-of-the-art performance in most of the Atari games included in this ablation experiment and greatly outperforms the 0% success rate of the intrinsically motivated PPO control in robotics (main text Fig. 4c). Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. In (a), averaging is over 50 runs per game with repetition and 5 runs per game without repetition. In (b) and (c), averaging is over 100 runs per target shelf with repetition and 10 runs per target shelf without repetition.

(a) Exploration phase scores in the 11 Atari focus games. In Pitfall, the score with fixed downscaling remains at 0, overlapping with the SOTA line. The peculiarities of the Solaris and Venture plots are explained in SI "Exploration phase without dynamic representations".

(b) Cells discovered by the exploration phase in the 11 Atari focus games.

Supplementary Figure 2: **Exploration phase performance with a fixed vs. dynamically discovered representation.** The fixed representation was optimised for Montezuma's Revenge. (a) The use of a fixed representation greatly hinders performance in the vast majority of the tested games for which the representation was not optimised. (b) The two pathologies of fixed representations can clearly be seen: producing an excessive number of cells (Berzerk and Solaris) and producing too few cells (Bowling, Freeway, Gravitar and Venture). Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over 50 runs per game with dynamic downscaling and 5 runs per game without dynamic downscaling.
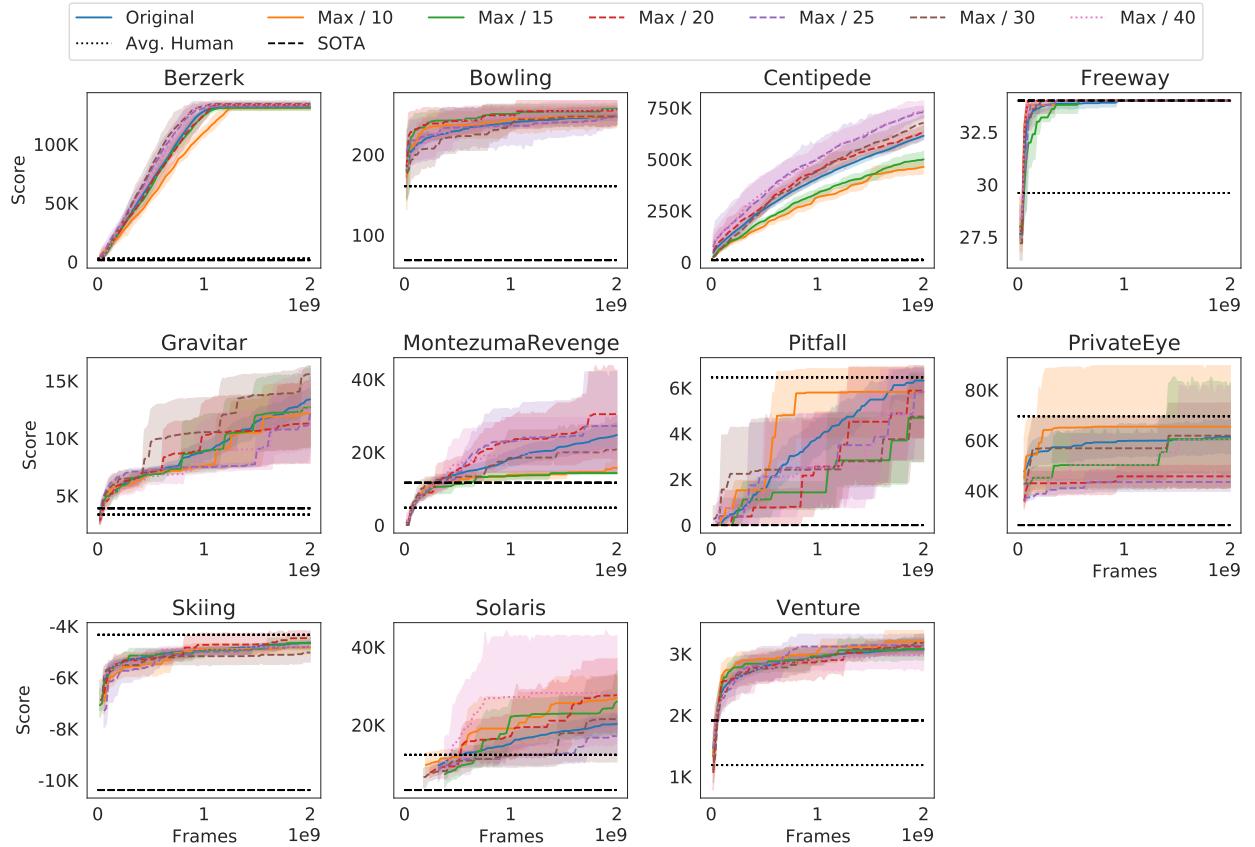
the end of an episode (i.e. the agent dying) is only reached after around 1.6 billion frames have been processed.

3.3   Downscaling distribution minimum means   During the randomised search for downscaling parameters (Methods "Downscaling on Atari"), parameters are sampled from a geometric distribution whose mean is the current best value of the parameter. The use of the geometric distribution captures the intuition that lower parameter values need to be over-sampled compared to larger values because they produce representations that are more different from each other, i.e. a downscaling with a width of 100 is unlikely to produce an aggregation that is very different from one with a width of 101, while representations with width 1 and 2 are likely to be very different from each other. The geometric distribution was thus chosen as a way to sample low values with higher probabilities. If the current best known value for a parameter is very low, however, it may become virtually impossible for larger values to ever be sampled. As a result, we define a minimum mean for each parameter. These are set to approximately $1/20^{th}$ of the maximum value the parameter can take (8 for width, 10.5 for height, and 12 for depth).
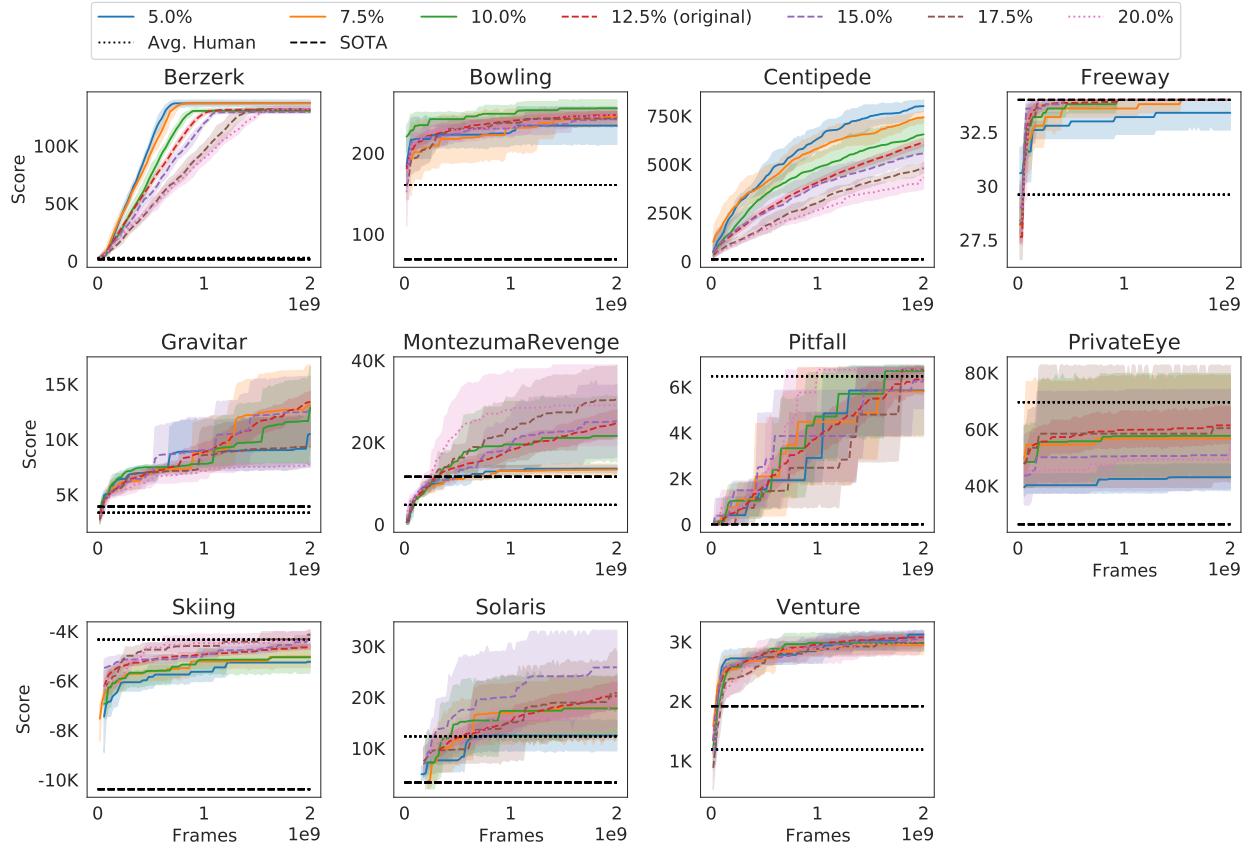
It is reasonable to wonder whether the minimum means have a strong effect on the performance of the exploration phase of Go-Explore, or whether they merely serve to avoid the collapse to low values described above and can be set to a wide range of values. In Supplementary Fig. 3, we show that the algorithm is robust to values ranging from $1/40^{th}$ to $1/10^{th}$ of the maximum, with no value producing qualitatively different results from the original relative to average human and state-of-the-art performance.

3.4   Downscaling target proportion   The search for downscaling parameters aims to produce a target number of cells given a buffer of sample frames (Methods "Downscaling on Atari"). This target number of cells is a fraction of the number of frames in the buffer, set to 12.5% in our implementation (i.e. targeting one cell for every 8 frames in the buffer). Because the target fraction affects the number of cells created in the archive, which could affect exploratory behaviour, we tested how sensitive the exploratory process is to its value.

In Supplementary Fig. 4, we show that good performance is achieved with values ranging from 5% to 20%, and thus that this parameter does not require excessive fine-tuning. With the exception of Freeway with 5%, all values achieve the same or better qualitative results as the original in terms of performance relative to average human and state-of-the-art performance.

49

Supplementary Figure 3: **Exploration phase performance with different values of the minimum means.** The effect of these values is small and statistically insignificant for the vast majority of values and games, with rare exceptions (most notably "Max / 10" and "Max / 15" get significantly worse results on Montezuma's Revenge and Centipede; $p < 0.05$ according to a two-sample empirical bootstrap test with 10,000 samples). Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over 50 runs per game for the original minimum means and 5 runs per game for other values.

Supplementary Figure 4: **Exploration phase performance with different target cell proportions.** High-quality results are achieved across all values for all games (except Freeway at 5%). While results within any target cell proportion are high-performing, sparse reward games such as Freeway, Montezuma, PrivateEye, and Solaris tend to benefit more from a larger target cell proportion, while dense reward games like Berzerk and Centipede tend to benefit from a smaller target proportion. These results suggest that producing more cells favours exploration, while producing fewer cells favours exploitation, perhaps because more time is spent expanding high-scoring trajectories rather than newly-found cells. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over 50 runs per game for the original 12.5% value and 5 runs per game for other values.
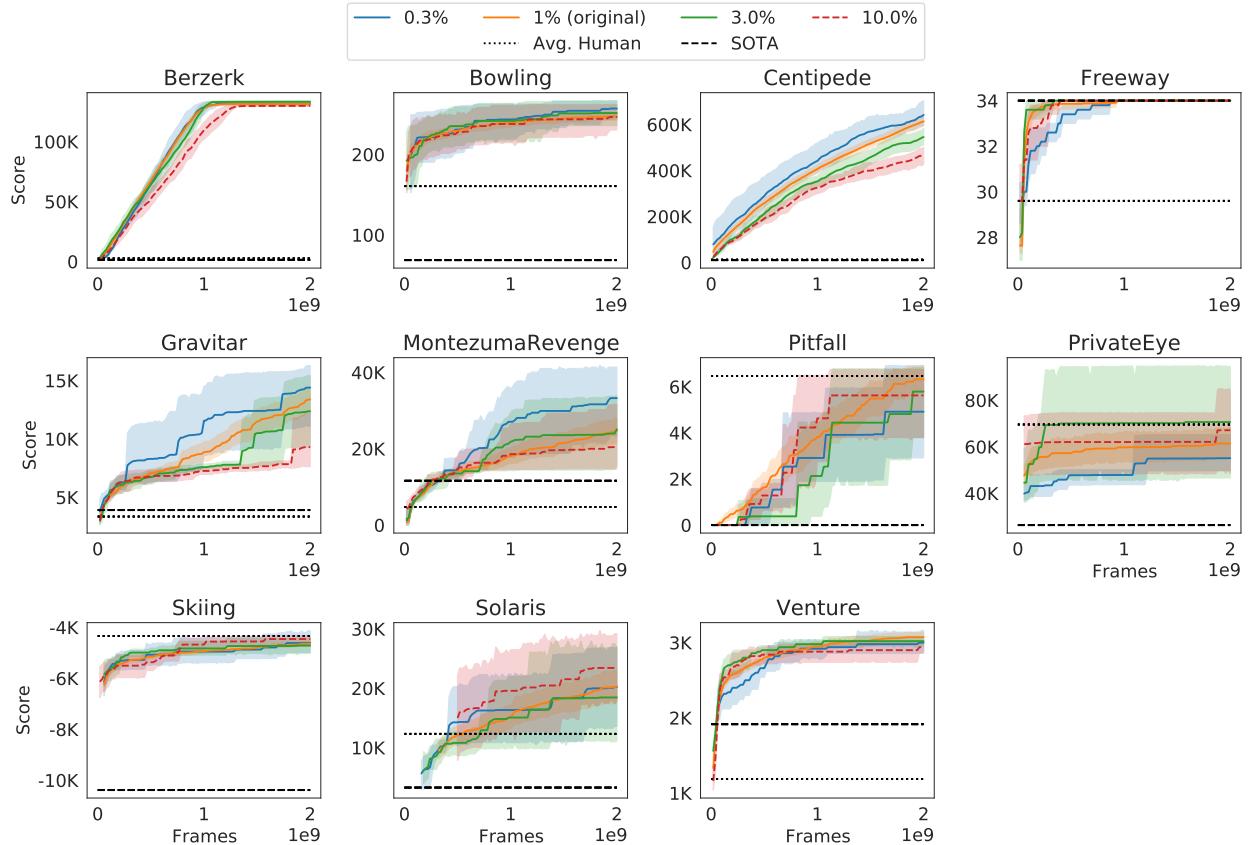
3.5 Downscaling sampling rate Downscaling hyperparameters are found through a search process over a sample of frames discovered during exploration. Because the buffer has a limited size and to ensure that it contains diverse frames, frames are added to the buffer probabilistically according to a sampling rate of 1% (Methods "Downscaling on Atari"). The sampling rate effectively controls the tradeoff between having a more diverse set of frames in the buffer (if it is low) or a set of more recent frames (if it is high). In this experiment we investigate whether the performance of the exploration phase is strongly sensitive to this hyperparameter.

In Supplementary Fig. 5, we analyse sampling rates ranging from 0.3% (because the buffer is cleared every 10 million actions – the frequency at which the representation is recomputed – and contains 10,000 frames, frequencies less than or equal to 0.1% are likely to not always fill the buffer completely) to 10%. We find that the sampling rate generally does not have a large effect on results, though the higher sampling rate of 10% does tend to produce lower performance relative to the original value of 1% (the difference is statistically significant in Berzerk, Centipede, Gravitar and Venture; $p < 0.05$ according to a two-sample empirical bootstrap test with 10,000 samples), supporting the usefulness of increasing buffer diversity through sampling.
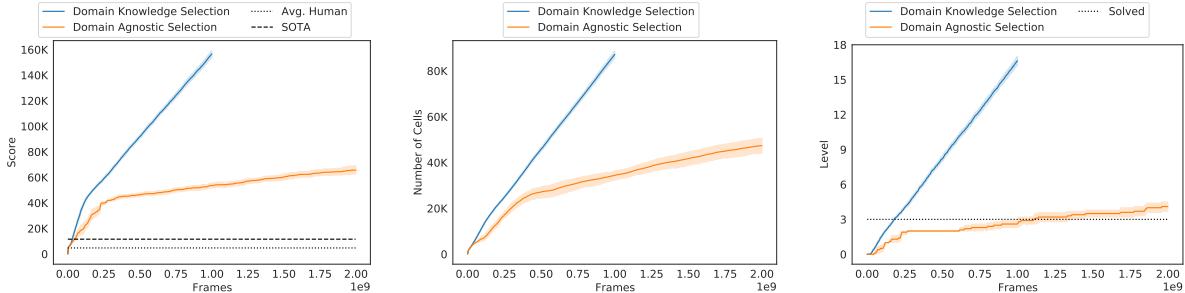
3.6 Domain-agnostic selection probabilities in Montezuma's Revenge As explained in Methods "Exploration phase", a custom cell selection probability that makes use of domain knowledge is used for Montezuma's Revenge with domain knowledge. Supplementary Fig. 6 shows that this selection probability greatly speeds up the exploration phase in that context, but that even with the generic selection probability, Go-Explore still solves the entire game when using a domain knowledge cell representation.

3.7 Robustification without imitation learning loss As mentioned in Sec. "PPO and SIL", a loss inspired by self-imitation learning[37] (the "SIL loss") is added during the robustification phase of Go-Explore. Supplementary Fig. 7 shows the effect of the SIL loss on the Atari games Montezuma's Revenge and Pitfall as well as on the robotics environment. SIL provides an early lift in robustifying Montezuma's Revenge and may provide a slight overall boost in both games, though its effect by the end of training is not statistically significant at the 95% level in either Montezuma's Revenge or Pitfall, according to a two-sample empirical bootstrap test. In robotics, the effect of the SIL loss is drastic: without SIL, no robustification run was able to succeed after 1 billion frames, whereas the success rate with SIL at 1 billion frames across all target shelves is 96.5%.

3.8 Policy-based Go-Explore without imitation learning loss Similar to the robustification phase, policy-based Go-Explore implements a Self-Imitation Learning (SIL) loss. Removing the SIL loss

Supplementary Figure 5: **Exploration phase performance with different frame sampling rates.**
While the sampling rate does not have a large effect overall (and does not change the qualitative
result of Go-Explore producing advances over humans and state-of-the-art algorithms), as a gen-
eral trend, larger values (and in particular the largest value in this experiment, 10%) tend to have
reduced performance, highlighting the importance of diversity rather than recency in the sample
buffer. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over
50 runs per game for the original 1% value and 5 runs per game for other values.

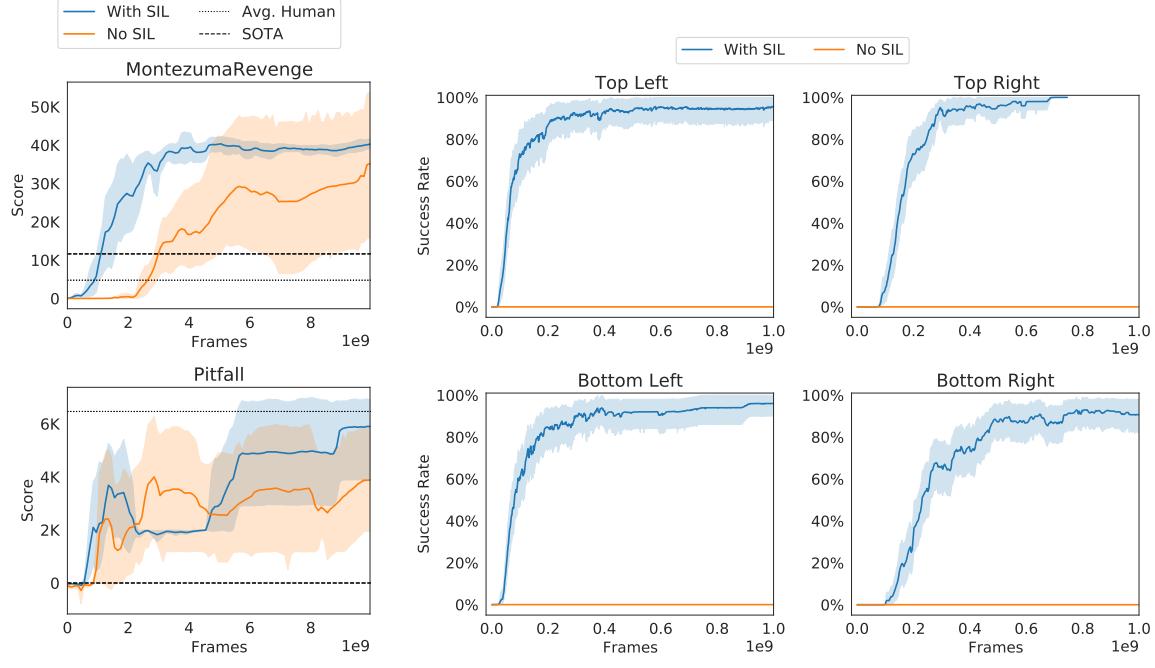(a) Exploration phase score in Montezuma's Revenge with domain knowledge.

(b) Cells discovered by the exploration phase in Montezuma's Revenge with domain knowledge.

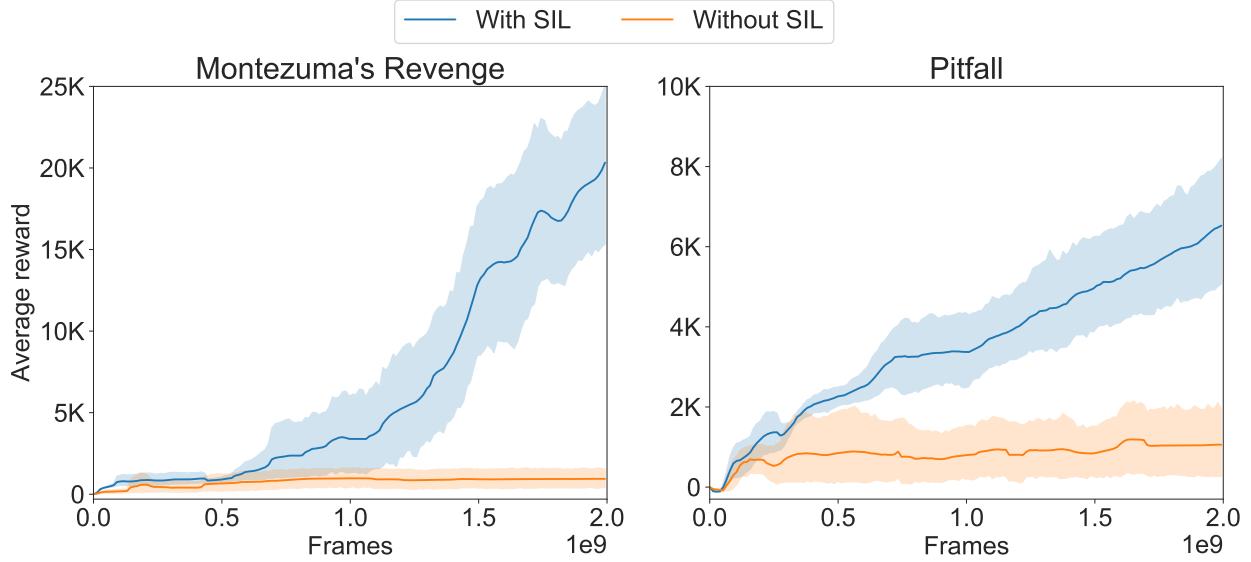(c) Levels solved by the exploration phase in Montezuma's Revenge with domain knowledge.

Supplementary Figure 6: **Exploration phase on Montezuma's Revenge with domain knowledge with and without a cell selection probability that makes use of domain knowledge.** Making use of domain knowledge in cell selection probability greatly speeds up exploration, especially in the later stages. However, even without a game-specific cell selection probability, the exploration phase quickly exceeds the state of the art, makes consistent (though slower) progress, and eventually solves level 3 and thus the entire game. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over 100 runs with domain knowledge selection and 10 runs with domain agnostic selection.

substantially reduces the performance of policy-based Go-Explore on both Montezuma's Revenge and Pitfall, at least over the first 2 billion frames (Supplementary Fig. 8). It is unclear why the SIL loss has such a clear benefit for policy-based Go-Explore on Atari while it does not have a clear benefit when robustifying Atari. One possible explanation is that the SIL loss mostly helps when learning how to reach difficult to reach states. Policy-based Go-Explore benefits because, without SIL, it only obtains experience on how to reach a difficult to reach state when the policy is actually able to return to such a state. In robustification, on the other hand, the agent is regularly started near these difficult to reach states, meaning it is much more likely to gather new experience that reaches these states, and thus obtains less benefit from imitating the experience from the demonstration.

3.9   Policy-based Go-Explore without a cell trajectory   A prominent feature of policy-based Go-Explore is that the goal-conditioned policy is provided with a cell-by-cell trajectory towards the target cell. In principle, it should be possible to train a goal-conditioned policy to move directly towards the target cell, without providing the intermediate trajectory. However, the ablation that removes this cell trajectory demonstrates that policy-based Go-Explore performs substantially worse without the trajectory on both Montezuma's Revenge and Pitfall, even to the extent that policy-

(a) Robustification score for Montezuma's Revenge and Pitfall.

(b) Robustification success rate for robotics.
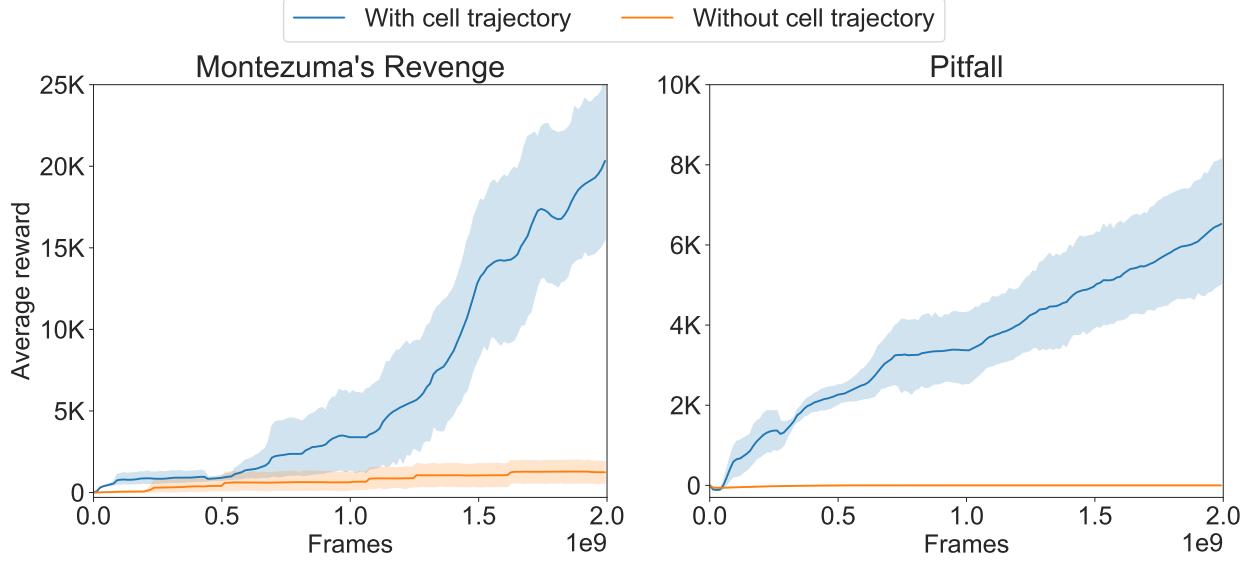
Supplementary Figure 7: **Robustification with and without the Self Imitation Learning (SIL) loss.** (a) In Montezuma's Revenge, the SIL loss provides an early lift. In Pitfall, the SIL loss appears to provide some benefit, though in neither case is the improvement statistically significant. (b) In robotics, the benefit of the SIL loss is very large: without SIL no robustification process was able to succeed after 1 billion frames, while the overwhelming majority of runs succeed in the same amount of time when the SIL loss is included. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. In (a) averaging is over 5 runs per game and per variant. In (b), averaging is over 50 runs per target shelf with SIL, and 5 runs per target shelf without SIL.

Supplementary Figure 8: **Policy-based Go-Explore with and without the Self Imitation Learning (SIL) loss.** The SIL loss provides substantial increases in terms of average score on both Montezuma's Revenge and Pitfall. Shaded areas show 95% bootstrapped confidence intervals of the mean with 1,000 samples. Each line is averaged over 10 runs with independent seeds.

based Go-Explore is unable to find any reward in Pitfall (Supplementary Fig. 9). The reason is that, without the intermediate trajectory, returning to a far-away cell is itself a sparse reward problem. Initially, the agent will visit cells near the starting position purely by random exploration. Over time, the agent will learn how to visit those cells intentionally when they are provided to the goal-conditioned policy as a target. From there, the agent will discover new cells that are farther away from the starting point. However, when being trained to return to these farther away cells, the agent has to find those cells from the start, and is not provided with any gradient towards those cells. Imagine that the agent has mostly explored the first room in Montezuma's Revenge and now discovers its first cell in the next room. The representation of this new cell has $[room = 2, x = 0)]$, but it was discovered from a cell with $[room = 1, x = 20]$. At this point, there is no way for the policy to know that, in order to reach the target with $[room = 2, x = 0]$, it has to first execute the policy towards $[room = 1, x = 20]$ (in fact, the policy is much more likely to execute the actions towards $[room = 1, x = 0]$ instead, because room was never a relevant feature before). As such, the agent has to rely on the occasional random actions from the stochastic policy to rediscover the cell in the next room and obtain the experience necessary to learn how to reach it. Both the SIL loss (which replays the experience of discovering the cell) and the cell trajectory help in alleviating the problem, and their impact on performance is similar (compare Supplementary Fig. 8 and Supple-

mentary Fig. 9), but both are required to really improve performance. Other solutions could be to develop techniques that explicitly teach the agent to generalise to new cells or to provide rewards for following the trajectory, even if the policy does not get to observe the intermediate cells.
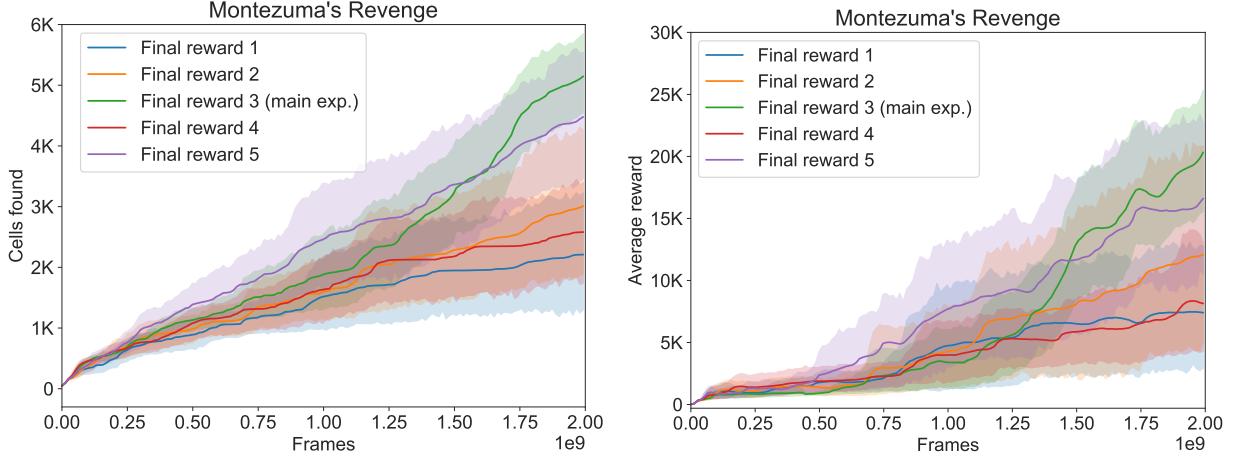
Supplementary Figure 9: **Policy-based Go-Explore with and without a trajectory of cells.** The results suggest that the trajectory of cells is essential for obtaining high average cumulative rewards in both Montezuma's Revenge and Pitfall. Shaded areas show 95% bootstrapped confidence intervals of the mean with 1,000 samples. Each line is averaged over 10 runs with independent seeds.

3.10 Policy-based Go-Explore final-cell reward To implement the general practice of having a higher reward for reaching a desired final state than for completing any intermediate objectives[65,66], we provide a higher reward for reaching the final cell in a trajectory than for reaching any of the intermediate cells. Doing so encourages the goal-conditioned policy to shorten its trajectory towards the final cell of a trajectory, thus resulting in more efficient behaviour. To demonstrate the effect of this increased *final reward*, we ran experiments for different values of the final reward, ranging from 1 (equal to the intermediate cell reward) through 5 (5 times higher than the intermediate cell reward). The experiments were performed on Montezuma's Revenge for 2 billion frames, with 10 random seeds for each parameter value.

The best values for the final reward (i.e. 3 and 5) result in increased performance, but all choices result in the continuous discovery of new cells and increasing performance over time (Supplementary Fig. 10). It is unclear why certain values perform better than others. It is possible that the final
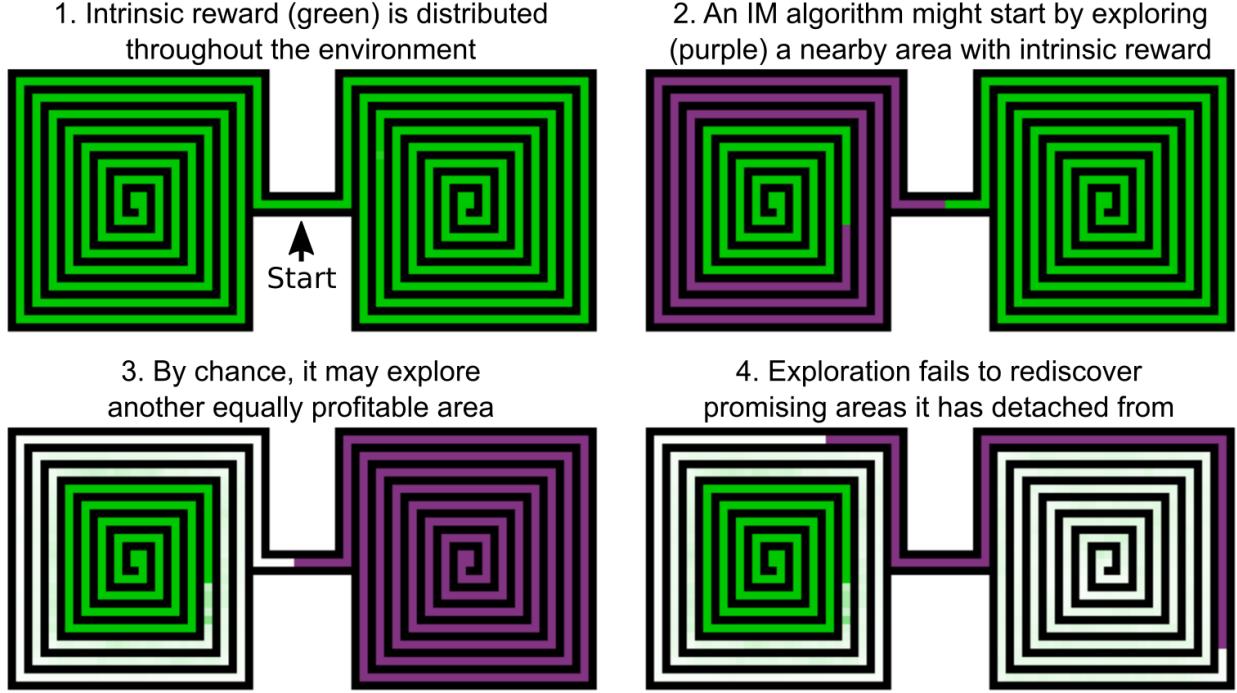
reward influences which in-game rewards are collected by the agent, thus affecting both exploration and average reward, or that the results are statistical noise due to the relatively small number of 10 samples.

Supplementary Figure 10: **All tested values for final rewards lead to the continuous discovery of new cells over the first 2 billion frames (left).** Similarly, the average reward continuously increases for all treatments (right). Shaded areas show 95% bootstrapped confidence intervals of the mean with 1,000 samples. Each line is averaged over 10 runs with independent seeds.

4   Detachment and derailment Environment exploration has long been a central topic in the field of reinforcement learning[1,34,50,51,64]. Despite the extensive research on exploration in reinforcement learning, we hypothesise that many previous algorithms have been affected by two major issues which we call *detachment* and *derailment*.

4.1   Detachment We define detachment as losing track of interesting areas to explore from. Here, "interesting areas to explore from" refers to states for which we have evidence (e.g. a low number of visits) that they could lead to the discovery of new areas of the environment. "Losing track" means that the algorithm stops trying to visit those areas prematurely, despite the fact that they are not thoroughly explored yet. For reinforcement learning algorithms that only optimise the expected return, detachment is almost guaranteed, as these algorithms do not attempt to promote exploration explicitly. Unless external rewards are aligned with interesting areas to explore from, such algorithms will stop visiting under-explored areas in favour of areas with a high return. However, algorithms that reward the agent for exploring new states can still suffer from detachment. In intrinsic-motivation (IM) algorithms, for example, detachment can happen because the intrinsic reward is lowered each time a state is visited, so that, eventually, they in effect provide no incen-

1. Intrinsic reward (green) is distributed throughout the environment

2. An IM algorithm might start by exploring (purple) a nearby area with intrinsic reward

3. By chance, it may explore another equally profitable area

4. Exploration fails to rediscover promising areas it has detached from

Supplementary Figure 11: **Example of detachment with intrinsic reward.** Green areas indicate intrinsic reward, white indicates areas where no intrinsic reward remains, and purple areas indicate where the algorithm is currently exploring.

tive for an agent to return to them. This issue can be especially prominent when there are multiple frontiers, as the agent may now, due to stochastic exploration, stop visiting one of those frontiers long enough to forget how to return to it (Supplementary Fig. 11). When this happens, the agent has to effectively relearn how to reach the frontier from scratch, but this time there is no intrinsic reward anymore to guide the agent. If intrinsic rewards were required to find the frontier in the first place, meaning that an algorithm without intrinsic rewards would fail to find this frontier, the IM algorithm will similarly fail to rediscover the frontier as well, meaning it has detached from the frontier. One might think that allowing intrinsic motivation to regrow after a time would solve the issue, but in that case the same dynamic can just play out over and over again endlessly.

4.2 Derailment We define derailment as when the exploratory mechanisms of the algorithm prevent it from returning to previously visited states. Returning to previously visited states is important because many RL environments, and especially hard-exploration environments, contain a large number of states that are far way from a starting state and can not be easily reached from such a starting state through random actions, even when exploring for hundreds of billions of frames. To discover these states, many algorithms rely on the policy learning to take actions that lead to states
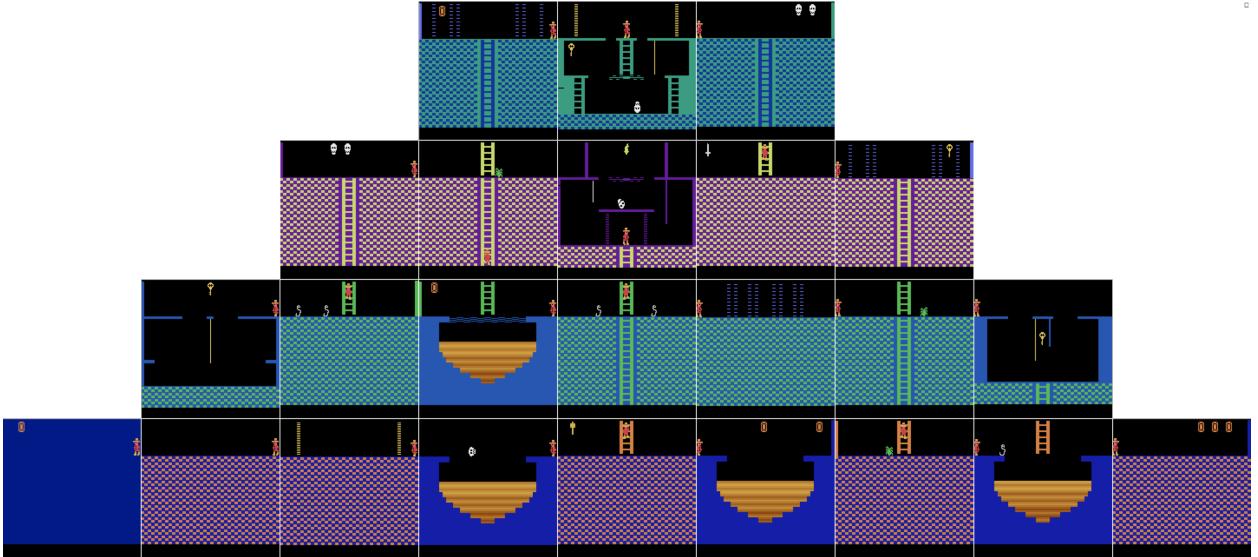
59

that are increasingly further away, either because the policy discovered external rewards leading towards these far away states, or because the algorithm provides intrinsic rewards that lead the policy towards infrequently visited states. However, as the policy needs to take an increasingly large number of correct actions to reach unexplored areas of the environment, it becomes increasingly likely that a state-agnostic exploration mechanism (i.e. any exploration mechanism that explores the same amount in all states, regardless of whether the agent is in a novel or a well-explored area), such as ϵ-greedy exploration, will cause the policy to take one or more exploratory actions that prevent it from reaching the distant state it sought to return to, thus stifling exploration. Two common strategies to prevent derailment are to (1) set the exploration probability (e.g. ϵ in ϵ-greedy exploration) to be small or (2) to start with a high exploration probability, but reduce it over training iterations[18]. Working with a fixed, low exploration probability throughout training reduces the effective derailment throughout training, but it also means that very little exploration will happen once a new state is reached. Annealing exploration over training iterations will initially lead to a lot of exploration at the cost of heavy derailment. Unfortunately, if there exists a far away state that requires precise actions to reach, that state will not be reached until the exploration probability has been reduced sufficiently to avoid derailment. This low exploration probability means that, once the agent is finally able reliably reach this far away state, very little exploration will be performed after reaching it.

The solution we propose to avoid derailment is to have an algorithm exhibit separate exploration probabilities depending on whether it is in a well-known area of the environment, meaning the probability of exploratory actions should be low, or in an unknown area of the environment, meaning the probabilities of exploratory actions should be high. While doing so is not a feature of state-agnostic exploratory mechanisms like ϵ-greedy exploration, one could hope that it is a property of algorithms that explore by sampling from a stochastic policy, because stochastic policies could learn to be low entropy in familiar states (i.e. they are certain about the correct action) while remaining high entropy in new states (i.e. where they should be uncertain about the correct action). However, deep learning struggles to remain well-calibrated when provided out-of-distribution input data. In image classification, for example, when networks classify images far out of distribution, it would be helpful if these networks returned a uniform distribution of probability across classes to properly indicate uncertainty, but networks instead are often surprisingly overconfident[90,91]. In the context of RL, this result means that we can expect trained policies to be highly confident in their actions (i.e. low entropy), even in areas that they have never observed before, thus

resulting in a lack of exploration. To remedy this issue, stochastic-policy RL algorithms generally add an entropy bonus to the loss function, encouraging the policy to assign more equal probability to all actions. However, because this entropy bonus applies equally throughout the trajectory of a policy, it is difficult to tune the entropy bonus in a way that guarantees effective exploration without sacrificing the network's ability to return; if the entropy bonus is too high, the policy will frequently take exploratory actions that prevent it from returning, thus causing derailment, but if the entropy bonus is too small, the policy will not explore sufficiently when a new area is reached.

5 Exploration in Atari While scores provide one measure of exploration in hard-exploration games, additional insights can be gained by examining game-specific metrics, especially for games which have received at lot of attention in past research, such as Montezuma's Revenge and Pitfall. Both Montezuma's Revenge and Pitfall are recognised in the field of RL for being exceptionally difficult to solve because they have extremely sparse rewards[1,50,51,55–57,64,72,85,92]. Montezuma's Revenge has become an important benchmark for exploration algorithms (including intrinsic motivation algorithms) because precise sequences of hundreds of actions must be taken in between receiving rewards. Pitfall is even harder because its rewards are sparser and because many actions yield small negative rewards that dissuade RL algorithms from exploring the environment. As such, these games often received special attention in previous works, with many papers reporting game specific metrics in addition to raw scores[31,34,50,51,56,72,93].

One particular metric indicative of exploration in both Montezuma's Revenge and Pitfall is the number of *rooms* the algorithm discovers. In both of these games the world is broken down into rooms, with each room containing different obstacles, enemies, and items. These rooms are spatially organised: Montezuma's Revenge features 24 rooms organised in the shape of a pyramid (Supplementary Fig. 12) and Pitfall has 255 rooms that are horizontally organised, with the last room being connected to the first room. The agent can travel between rooms by moving to the edge of the screen, after which it will be moved to the adjacent room. Montezuma's Revenge also has 3 unique *levels*, where each level features a new set of 24 rooms, thus resulting in a total of 72 unique rooms. To reach the next level in Montezuma's Revenge, the agent has to navigate to the bottom-left room in the pyramid (called the treasure room), after which it will be teleported (after a short delay) to the middle room at top of the pyramid in the next level. While Montezuma's Revenge has only 3 levels, the final level repeats indefinitely (with some stochastic events due to sticky actions), meaning the agent can play this level an arbitrary number of additional times to increase its score (provided it is robust enough to handle the stochastic events).

61

Supplementary Figure 12: **The pyramid-like room layout of Montezuma's Revenge (level 1).**

While we did not track levels or rooms in the experiments with a downscaled representation (because such information is domain dependent and our downscaled representation is not), it is part of our domain knowledge representation, which we tested on Montezuma's Revenge and Pit-fall. These metrics indicate that Go-Explore with a domain-knowledge representation thoroughly explores these games. In Pitfall, Go-Explore finds all 255 rooms in 96 of the 100 runs, with the remaining runs never finding fewer than 183 rooms. In Montezuma's Revenge, Go-Explore finds all 72 unique rooms in 97 out of 100 runs (and 71 rooms in the remaining 3). Go-Explore also completely solves all 3 unique levels in Montezuma's Revenge in all of the 100 runs, which is what allows the robustified policies to obtain almost arbitrarily high scores, as reported in the main paper. Policy-based Go-Explore demonstrates similar exploratory abilities. On Pitfall, it finds all 255 rooms in 8 out of 10 runs, with the lowest number of rooms discovered among the other 2 runs being 236. On Montezuma's Revenge, policy-based Go-Explore finds all 72 unique rooms and solves all 3 unique levels in 9 out of 10 runs, with the remaining run finding 70 unique rooms and solving the first 2 levels.

Other games worth mentioning are Private Eye and Skiing. Private Eye is a sparse reward game and is also a common benchmark in work focused on exploration in RL[1,13,31,51,53–55]. The goal in Private Eye is to collect evidence and apprehend a criminal by navigating a maze-like "city". Similar to Montezuma's Revenge and Pitfall, the environment in Private Eye contains very few rewards and is full of hazards that need to be avoided. In addition, the objectives need to be

completed in a specific order (all evidence needs to be collected and delivered to specific locations before the criminal can be apprehended), making it particularly difficult to complete all objectives, as is evidenced by the fact that state-of-the-art performance is 26,364[53], far below the roughly 100,000 points that can be obtained by completing all main objectives. Despite these obstacles, the robustified policies produced by Go-Explore with a downscaled representation are able to reliably achieve more than 100,000 points in 4 out 5 runs, thus having discovered and learned to perform all objectives in the game.

Skiing is a notoriously difficult game for RL agents, with state-of-the-art performance (-10,386)[54] far below that of average human performance (-4,336). In contrast to hard-exploration games like Montezuma's Revenge and Pitfall, Skiing is difficult to learn because of its reward structure. The goal of the game is to reach the end of a slope as fast as possible while passing through all the gates that appear along the way down, with each frame spend resulting in a small (-1 or -2) negative reward and each gate missed resulting in a large (-500) negative reward. However, while the time-spent reward is provided immediately, the negative reward for gates missed is only provided at the end of the slope, thus resulting in a difficult credit assignment problem with delayed reward (e.g. the reason for a negative reward at the final time step may be the result of missing the very first gate). The scale difference in rewards also means that reward clipping generally results in the agent ignoring the gates completely, because no matter how many are missed, the gates account for only a single reward instance, while the time-based reward is received many times throughout an episode. Go-Explore, however, is able to find demonstrations that pass through many gates (as it keeps track of the highest performing trajectory that reaches the end of the slope) and allows rewards to be normalised appropriately (as it learns about the magnitude of the game rewards during the exploration phase), rather than clipped. As a result, the robustified policies produced by Go-Explore with a downscaled cell representation receive a mean score of -3,660, outperforming human performance (Fig. 2b in the main paper).

6 Generality of downscaling Downscaling is a simple method for aggregating states into cells that can potentially be applied in any domain where the state is a visual observation (our experiments demonstrate that it is effective on all games in the Atari benchmark), though it is possible that complex environments with rich visuals may produce visual changes that are irrelevant to exploration, yet result in different downscaled frames, which can hinder exploration and may require more sophisticated (e.g. learned) representations.

7 Derailment in robotics As shown in the main text of this paper, a count-based intrinsic motivation control completely fails to discover any rewards in the robotics environment even though it is given the same domain knowledge state representation as Go-Explore's exploration phase and when given a comparable budget of frames to Go-Explore's exploration and robustification phases combined. Evidence from the experiments suggests that this failure is primarily due to the problem of *derailment*, specifically to the difficulty that the IM control has of learning to reliably *grasp* the object.

Grasping is widely considered an extremely difficult task to learn in robotics[94,95]. The overwhelming majority of undiscovered cells are those that require grasping the object and lifting it to reach. The claim that the failure to explore the environment is due to derailment when grasping the object necessitates that grasping is *discovered*, but cannot be reliably reproduced by the policy due to its excessive exploratory mechanisms. We separate the discovery of grasping into three steps: touching the object with one of the two grippers (the "touch" step), touching the object with both grippers (the "grasp" step), and finally lifting the object (the "lift" step). An analysis of the cells and counts discovered by 20 control runs (5 per target shelf) shows that all runs discover the "touch" and "grasp" step, but in 18 (90%) of these runs, the count associated with the "grasp" step is at least 10x smaller than that associated with the "touch" step, indicating difficulty (and thus possible derailment) in learning to go from the "touch" step to the "grasp" step. In the 2 (10%) remaining runs, the "grasp" step count is closer to the "touch" step count, but, in one case, lifting is never discovered, and in the other, lifting is discovered, but the count for the "lift" step is again over 10x smaller than that of the "grasp" step, indicating possible derailment in between those two steps. It is thus apparent that the IM control has difficulty returning to the grasping stepping stones that it discovers, in spite of these cells often having amongst the lowest counts of any cell discovered, and thus the highest intrinsic rewards, thereby providing evidence of derailment.

8 Go-Explore and Quality-Diversity Preserving and exploring from stepping stones in an archive is reminiscent of the MAP-Elites algorithm[96], and quality diversity algorithms more broadly[97,98]. However, Go-Explore applies these insights in a novel way: while previous QD algorithms focus on exploring the space of behaviours by randomly perturbing the current archive of policies (in effect departing from a stepping stone in policy space rather than in state space), Go-Explore explicitly explores the state space by departing to explore anew from precisely where a previous exploration left off. In effect, Go-Explore offers significantly more controlled exploration of the state space than other QD methods by ensuring that the scope of exploration is cumulative through

the state space as each new exploratory trajectory departs from the endpoint of a previous one.

9 Go-Explore, Planning, and Model-based RL The way in which Go-Explore explores a search space is reminiscent of classical planning algorithms such as breadth-first search, depth-first search, or A*[46]. These planning algorithms often explore a search space starting with a set of unexplored nodes called the *frontier* and then iteratively: (1) select a node from the frontier, (2) gather the nodes that can be reached from the selected node (called *expanding the node*), and (3) add the gathered nodes to the frontier so that they can be selected and expanded in the future. Within the formalism of Markov decision processes that is the basis of RL, states are the natural equivalent of planning nodes and in order to fully expand a state it would be necessary to take every possible action from that state (up to an infinite amount of times if the environment features unknown stochastic transitions). Small environments with limited stochasticity may be explored in their entirety this way, but many practical RL environments feature state-spaces that are much too large to fit in memory, large or continuous actions spaces in which it is intractable to try every action in every state, and ubiquitous stochasticity in transitions that makes it impossible to know when a state has been fully explored. Go-Explore demonstrates one way in which we can transfer the principles from planning algorithms and overcome the aforementioned challenges. When considering Go-Explore from the perspective of a planning algorithm, the archive is analogous to the frontier, selecting a state from the archive is analogous to selecting a node from the frontier, exploring from a state is analogous to expanding a node, and adding new states to the archive is analogous to adding the gathered nodes to the frontier. However, Go-Explore innovates relative to classic planning algorithms in two ways: (1) by aggregating similar states into cells, Go-Explore can be applied to domains with high-dimensional state spaces (more on this issue in the next paragraph), and (2) by running a learning-from-demonstrations algorithm on the trajectories found in the exploration phase, Go-Explore is able to train closed-loop policies that are able to deal with environmental uncertainty by generalising to never-seen-before states, which can even result in a policy that can outperform the plans they were trained on.

Porting the principles behind planning algorithms to high-dimensional state spaces by aggregating similar states into cells is a non-trivial technical challenge. One problem is that, in the aggregated space, it is unknown whether edges exist between the different nodes, meaning that an algorithm has to empirically discover the existence of an edge between two nodes, for example by executing a sequence of actions to try to reach one node from another. Thus, nodes can never fully be marked as "closed". Go-Explore addresses this issue by never marking any cells as closed, but

instead reducing the relative probability of selecting a cell when it is explored from. This way, if a cell has been explored many times, meaning it probably should be considered "closed", it is indeed selected only infrequently. However, if at some point all cells in the archive have been explored many times, the selection probabilities will equalise, thus ensuring that old cells will be explored from again, thus preventing the algorithm from getting stuck. Another challenge is that two different trajectories towards the same aggregated cell can actually lead to two very different states (e.g. in one state the agent may be in the process of jumping over a gap while in the other state the agent may be falling into that gap). This difference in states makes it difficult to substitute one path to a particular cell with a new path to that same cell, even if the new path is shorter or higher scoring, because, while the two paths may lead to the same cell, they may actually lead to very different states within that cell. Go-Explore side-steps this challenge by never performing path substitution, but instead by re-exploring whenever a better path to an existing cell is found. Overall, Go-Explore motivates porting the techniques from classic planning algorithms to challenging problems with high-dimensional search spaces and suggests some methods for how this may be achieved.

Go-Explore also exhibits important similarities with Rapidly-exploring Random Trees (RRT)[45], a popular planning algorithm in robotics domains, as both algorithms keep track of an archive of states and trajectories to those states. However, there are some crucial differences, including: (1) RRT proceeds by first sampling a goal to attempt to reach, which can be impractical in environments where reachable states are not known a priori (and which is particularly pernicious in high-dimensional state spaces, such as pixels or even learned encodings, where most randomly selected goals are unreachable), and (2) RRT does not have the concept of aggregation of states that is present in Go-Explore and thus RRT can add many very similar states to its archive that do little to help the algorithm reach meaningfully different unexplored areas of the search space. As with the classic planning algorithms, Go-Explore motivates porting techniques like RRT to challenging problems with high-dimensional search spaces.

Finally, Go-Explore is reminiscent in some ways of model-based RL algorithms. For example, when Go-Explore returns to previously visited states by restoring simulator state, the simulator effectively operates as the model for Go-Explore. Many model-based RL algorithms perform relatively shallow runs of stochastic planning algorithms, like MCTS[99] or UCT[62,100], to decide which action to take every time a decision has to be made. Go-Explore with state restoration, on the other hand, is more similar to algorithms like Dyna[101] and AHCON-M[102], which are hybrids

between model-based and model-free RL algorithms because they use a model during training, but eventually take actions with a model-free policy. However, Go-Explore with state restoration differentiates itself from those algorithms by deeply exploring a pre-existing model to find high-performing solutions which it robustifies into a model-free policy only after this exploration process has finished. Another difference is that model-based RL is generally focused on learning a model, but the two variants of Go-Explore presented in this paper do not try to learn a model; the version of Go-Explore that restores simulator state assumes that an appropriate simulator is already available (and it frequently is in many practical domains), while policy-based Go-Explore trains a goal-conditioned but model-free policy to navigate the environment. However, there is no obvious obstacle that precludes a variant of Go-Explore that takes exploratory steps in the real environment in order to learn a model, and then performs deep exploration within that model to determine the most promising location to explore next. Doing so is a promising area for future work.

10   Go-Explore and Stochasticity   One of the goals in the field of reinforcement learning is to develop agents that can operate "in the real world," which refers to applications ranging from having a robot navigate a house to a virtual assistant that could help accomplish tasks online such as booking travel. Many of these real world applications feature events that are unpredictable, but which can be modelled as being stochastic. For example, a gust of wind may be the result of pressure differences in the air around us, but predicting it based on what can be observed locally is difficult, meaning that it often makes more sense to consider gusts of wind as stochastic events that can happen with some probability. As such, in order to operate in the real world, agents will have to be able to deal with these kinds of perceived stochasticity in a robust and reliable way.

When exploring by restoring simulator state, Go-Explore relies on the robustification process to train a policy that is capable of dealing with stochasticity and this robustification process generally takes place in the simulator. Regardless of whether the simulator was deterministic or stochastic during the exploration process, it is desirable that the simulator includes some form of stochasticity during the robustification process in order to encourage robustness in the trained policy (e.g. the sticky actions in our experiments).

In order for the Go-Explore trajectories found in the exploration phase to be informative during the robustification phase, there are some limits to the amount and kind of stochasticity that can be in the environment during robustification. For robustification to be possible, the following two conditions should hold: (1) it has to be possible to roughly follow the trajectories found during the exploration phase and (2) doing so should result in a high expected cumulative reward.

67

The first condition is met if the transitions in the sample trajectory are sufficiently probable or if there is way to recover from, or compensate for, some of the transitions not leading to the desired next state. With 25% sticky actions, for example, there is a 25% chance that a transition does not lead to the desired next state whenever the previous action differs from the current action, virtually guaranteeing significant drift from the original trajectory, which is frequently thousands of steps long. However, in Atari, it is often possible to recover from an undesired transition. In Pitfall, an undesirable transition may occur when the agent transitions from moving left (requiring the "left" action) to climbing a ladder (requiring the "up" action). A sticky action here can cause the agent to overshoot the ladder, at which point the "up" action is no longer effective, but the agent can recover by taking a "right" action in order to move back to the ladder. Note that the first condition specifies that the agent only needs to be able to "roughly" follow the example trajectory. Extending the example above, because Pitfall has a time limit, the recovered state is not the exact same state that the agent would be in if it did not overshoot the ladder, because it now has fewer frames left to obtain the highest possible reward. However, the recovered state is sufficiently similar, in the sense that the agent does not have to adopt a different policy in order to obtain a high expected reward from this state. As such, the agent can still "roughly" follow the example trajectory, even though it is no longer possible to visit states that are identical to those found in the example trajectory.

The second condition mostly puts constraints on the stochasticity of the rewards. This condition can be broken if one or more of the transitions in the example trajectory are associated with rewards that are substantially higher than the expected reward for those transitions in the stochastic environment. For example, the highest scoring trajectory returned by the Go-Explore exploration phase could contain a transition in which the agent plays the lottery and wins (i.e. receives a high reward), even though the expected reward for playing the lottery may actually be negative. This condition can similarly be broken if even minor differences in state can have a large effect on the expected reward. One example would be if the agent had to operate under a very strict time limit such that any kind of deviation would result in the agent running out of time before being able to collect some final reward.

Overall, we do not believe that these constraints are overly restrictive in most practical scenarios. Robotics problems, for example, generally have to deal with stochasticity in the form of small inaccuracies in the actuators, rather than with lottery tickets. That said, it is likely that it is possible to improve Go-Explore to be more robust towards stochasticity in transitions and rewards (in fact, we suggest some improvements in SI "Policy-based Go-Explore and Stochasticity") and we believe

that this is a promising direction for future work.

11   Policy-based Go-Explore and Stochasticity Restoring simulator state is a highly efficient method for returning to previously visited states, as it both removes the need to replay the trajectory towards a previously visited state as well as the need to train a policy capable of doing so reliably. That said, doing so also has the potential drawback that some of the trajectories found by restoring simulator state can be hard to robustify if that trajectory is not representative of realistic policies that can succeed in the stochastic testing environment. For example, imagine a robot with the goal of crossing a busy highway. The cars on the highway are stochastic, meaning that their position, speed, and reactions will differ in every episode, but the highway is always busy. As such, we assume that there is no safe way to reliably cross the busy highway directly. The highway has an overpass that allows the robot to easily and reliably cross the highway safely, but it is located some distance away from the robot, meaning that it is not the shortest method to cross the highway. However, when restoring simulator state, it is likely that Go-Explore will find a way directly across the highway in this particular scenario, because there probably exists some static sequence of lucky actions that brings the agent from one cell on the highway to the next cell, and once the next cell is reached, that progress is saved in the form of the simulator state. Once the opposite side of the highway is reached, this shorter trajectory will overwrite any longer trajectories that go over the overpass, and the final trajectory returned will go directly over the highway. Training a policy that can reliably follow this trajectory may be impossible because the stochasticity of the cars on the highway (i.e. the stochasticity of the environment) can make it such that a sufficiently reliable policy simply does not exist. That is, each new random busy highway situation requires its own lucky set of actions that may not derive in any systematic way from the agent's observations.

Policy-based Go-Explore can alleviate this situations in two ways. First, because its progress along the highway is not saved and because each trial is in a stochastic environment, policy-based Go-Explore must attempt to return manually to each cell across the highway in different conditions. If there does not exist a reliable policy that can do so, it is unlikely that policy-based Go-Explore will ever cross the highway this way. As a result, policy-based Go-Explore is much more likely to learn a policy that reliably navigates the overpass instead.
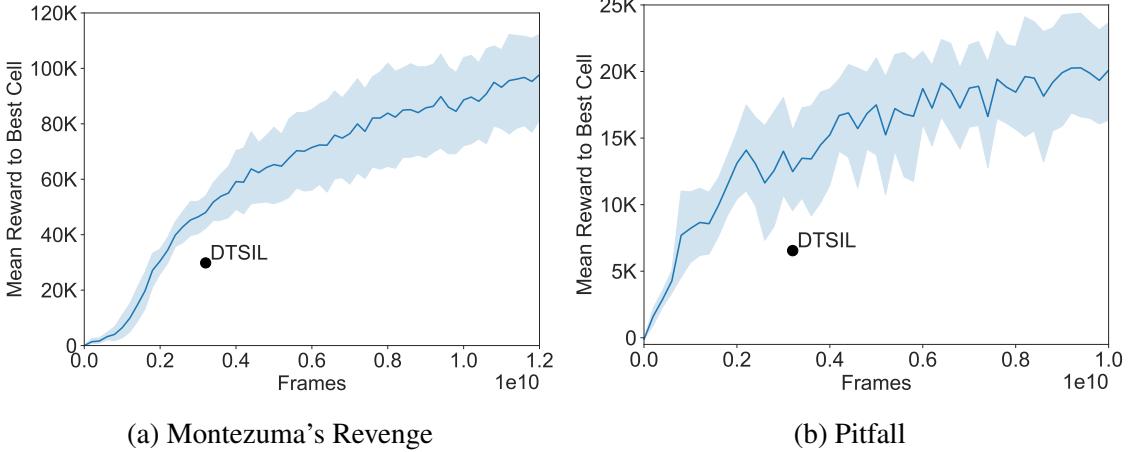
Second, because policy-based Go-Explore needs to return to cells in the presence of stochasticity, it can keep track of the success rate towards each cell in the archive. As such, even if policy-based Go-Explore is sometimes able to cross the highway, it is possible to not overwrite cells on the other side of the highway until the policy has learned to return to those cells reliably.

Doing so prevents the shorter, but unreliable trajectories from overwriting the longer but more reliable trajectories that take the safe overpass. A similar mechanic could be implemented to deal with stochasticity in rewards, as policy-based Go-Explore makes it possible to track the average reward when attempting to reach a particular state. Because it was not necessary to implement such mechanics for the games of Montezuma's Revenge and Pitfall, studying the effectiveness and exact implementation details of such a mechanic is a topic for future research.

Last, it may be possible to resolve these issues even when Go-Explore is allowed to restore simulator state. For example, it is possible to run the Go-Explore exploration phase many times with different random seeds, thus making it possible to estimate which trajectories are and which trajectories are not reliable. Recognising that trajectories with only slight differences in the states visited still represent effectively the same solution could be handled by the aggregation that occurs in the cell representation, meaning that solutions that visit the same *cells* in order (instead of the same states in order) could be considered the same. It thus seems possible to produce a version of Go-Explore that is able to estimate the reliability of different trajectories while still gaining the advantages of restoring simulator state. Identifying the exact form of that algorithm and experimentally validating it is a fruitful area of future research.

12 Comparing Policy-based Go-Explore and DTSIL After a pre-print paper describing Go-Explore[103] (but not policy-based Go-Explore) was published, and after our work on policy-based Go-Explore was long underway, another research team independently developed and published the Diverse Trajectory-conditioned Self-Imitation Learning algorithm (DTSIL)[34], which is similar to policy-based Go-Explore in many ways, as detailed below. Policy-based Go-Explore outperforms DTSIL on both Montezuma's Revenge and Pitfall after 3.2 billion frames, despite the fact that policy-based Go-Explore was tested on a harder problem, i.e. with sticky actions (Supplementary Fig. 13). In addition, the score of policy-based Go-Explore keeps increasing, eventually achieving a score of 97,728 on Montezuma's Revenge and 20,093 on Pitfall after 12 billion and 10 billion frames, respectively. It is possible that the performance of DTSIL would also improve with additional frames, but those results were not reported.

DTSIL is similar to policy-based Go-Explore in that it follows the methodology described in the Go-Explore pre-print in the following ways: (1) Like the original Go-Explore, DTSIL explicitly keeps track of an archive of many different states and trajectories to those states, (2) DTSIL first moves the agent to one of these states before performing random exploration, (3) DTSIL determines whether to add a state to the archive with the help of a domain-knowledge based state

| (a) Montezuma's Revenge | (b) Pitfall |

Supplementary Figure 13: **Policy-based Go-Explore test performance over time compared against final performance of DTSIL.** The final performance of DTSIL is indicated by the black dot and is positioned at 3.2 billion frames, the number of frames for which the DTSIL agent was trained. **(a)** On Montezuma's Revenge policy-based Go-Explore outperforms DTSIL after 3.2 billion frames by roughly 18,000 points. **(b)** On Pitfall policy-based Go-Explore outperforms DTSIL after 3.2 billion frames by roughly 6,000 points. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

embedding, where similar embeddings are grouped into a single cluster (i.e. a cell representation), and (4) DTSIL selects trajectories to follow (i.e. states to return to) by selecting them probabilistically based on the number of times particular clusters have been visited, though DTSIL did not consider any domain knowledge specific information for the purpose of this selection procedure.

Similar to policy-based Go-Explore, and as was recommended as a profitable future direction in our pre-print[103], DTSIL is a method that returns to previously visited cells with the help of a goal-conditioned policy (referred to as a trajectory-conditioned policy in the DTSIL paper because the policy is provided with a sequence of the next few goals, as explained below). Also similar to policy-based Go-Explore, DTSIL follows a trajectory of intermediate sub-goals towards a particular goal cell, rather than conditioning the policy directly on the state to return to, and DTSIL includes self-imitation learning to make training the policy more sample efficient. While working on policy-based Go-Explore, we independently invented the technique of following a trajectory of sub-goals and harnessing self-imitation learning.

One major difference between policy-based Go-Explore and DTSIL is that, when following a trajectory, DTSIL aims to provide the entire trajectory as input to the policy, rather than just the next cell in the trajectory. As a result, the DTSIL network architecture requires some method to

deal with variable length trajectories (resolved with an attention layer), while the policy-based Go-Explore architecture only requires the next (sub) goal cell as an input. Note that, in the DTSIL paper, most trajectories that are followed were found to be too long to provide to the network in their entirety, meaning the trajectory is instead provided to the network in small chunks, resulting in a dynamic that is similar to providing only the next goal.

Another major difference is that policy-based Go-Explore also samples from the policy while exploring, rather than just relying on random actions. As mentioned before (Extended Data Fig. 7), sampling from the policy results in the discovery of many more cells, potentially explaining the large performance advantage Go-Explore exhibits vs. DTSIL.

A third major difference is the way in which DTSIL and policy-based Go-Explore transition from exploration to exploitation. DTSIL transitions from exploration to exploitation by either slowly annealing across training iterations from selecting promising cells for exploration to selecting the highest scoring cells, or by making such a transition abruptly once a particular score threshold is reached. Policy-based Go-Explore, on the other hand, only focuses on exploration during training. Interestingly, despite being asked to return to all cells, rather than spending a good number of training iterations on just the highest performing ones (which is what DTSIL does), we found that policy-based Go-Explore tends to be able to reliably return to the highest scoring cell in the archive at test time.

A fourth major difference is our introduction of increasing entropy when the agent takes too long to reach the next cell (see Methods "Policy-based Go-Explore"). This entropy increases the exploration performed by policy-based Go-Explore only when necessary, thus largely avoiding the problem of derailment. Because derailment can severely lower performance, we expect that this innovation contributes substantially to the performance advantage of our implementation of policy-based Go-Explore relative to DTSIL.

A last major difference is with respect to the experiments that were performed. DTSIL was tested on Montezuma's Revenge and Pitfall without sticky actions. In preliminary experiments with policy-based Go-Explore, we found that removing sticky actions greatly simplified the problem, and testing policy-based Go-Explore without sticky actions would have increased its performance. As a result, and as explained in methods, we did not include DTSIL in our comparison with the state of the art, but we did provide a comparison at the beginning of this section.

Besides these major differences, there are many smaller differences between the two algorithms,

including differences in cell selection probabilities, SIL equations, reward clipping, maximum episode length, and hyperparameters. For a full overview of these differences, we recommend comparing the methods explained in this paper directly with the methods described in the DTSIL paper[34].

Because the algorithm described in the DTSIL paper is similar to policy-based Go-Explore, in preliminary experiments, we tested whether some of the hyperparameters described in the DTSIL paper would improve the performance of policy-based Go-Explore. In these preliminary experiments, we found that the grid size of their cell representation (determining the granularity for the x and y coordinates of the agent) of 9x9[†] and their learning rate of $2.5 \cdot 10^{-4}$ did indeed perform better than the hyperparameters we were testing at that time, and we adopted these hyperparameters from the DTSIL paper instead.

The training performance of DTSIL on Montezuma's Revenge and Pitfall was reported by Guo *et al.* (2019)[34] for experiments that ran for 3.2 billion frames. However, the performance-over-time graph presented in the DTSIL paper is not directly comparable with the performance-over-time graph shown in this paper because they represent results for different selection strategies. For policy-based Go-Explore, we always report the average score achieved when returning to the highest scoring cell in the archive (obtained after training is completed by loading stored checkpoints and testing the policy 100 times). In contrast, the DTSIL graph shows a rolling average score during training, which means that its average includes returning to low-scoring cells as the algorithm attempts to explore the environment. That said, the final performance of DTSIL after 3.2 billion frames is measured over only the highest scoring trajectories, and can thus be reasonably compared with the testing performance of policy-based Go-Explore after that many frames. For Montezuma's Revenge, we compare policy-based Go-Explore against the results that were reported in the supplementary information of the DTSIL paper for a version of DTSIL that implemented the same cell representation as the one used in the policy-based Go-Explore experiments (note, the Montezuma's Revenge results reported in the main DTSIL paper were lower). For Pitfall, we compare against the only reported results, which were obtained with a slightly different cell representation than the one used by policy-based Go-Explore. Specifically, the DTSIL cell representation includes the cumulative positive reward achieved; the representations are otherwise the same.

[†]Note that, while an Atari frame is 160x210 pixels, the top 50 rows of pixels are unreachable in our test games and were ignored, meaning that a discretization of 18x18 pixels does result in a 9x9 grid.

13   No-ops and sticky actions   The Atari benchmark has been accepted as a common RL benchmark because of the large variety of independent environments it provides[26]. One downside of the games available in this benchmark is that they are inherently deterministic, making it possible to achieve high scores by simply memorising state-action mappings or a fixed sequence of actions, rather than learning a policy able to generalise to the much larger number of states that are available in each game. As the community is interested in learning general policies rather than open-loop solutions, many have suggested approaches to improve the benchmark, usually by adding some form of stochasticity to the game[14].

One of the first of these approaches was to start each game with a random number (up to 30) of *no-op* (i.e. do nothing) actions[15]. Executing a random number of no-ops causes the game to start in a slightly different state each episode, as many game entities like enemies and items move in response to time. While no-ops add some stochasticity at the start of an episode, the game dynamics themselves are still deterministic, allowing for frame-perfect strategies that would be impossible for a human player to reproduce reliably. In addition, with only 30 different possible starting states, memorisation is more difficult, but still possible.

Because of the downsides of no-ops, an alternative approach called *sticky actions* was recommended by the community[14]. Sticky actions mean that, at any time-step greater than 0, there exists a 25% chance that the current action of the agent is ignored, and the previous action is executed instead. In a way, sticky actions simulate the fact that it is difficult for a human player to provide the desired input at the exact right frame; often a button is pressed a little bit too early, a little bit too late, or held down for a little too long. Given that Atari games have been designed for human play, this means that human competitive scores should be achievable despite the stochasticity introduced by sticky actions.

While sticky actions have been recommended by the community, no-ops are still widely employed in many recent papers[13,31]. As it is possible that no-ops add some challenges not encountered with just sticky actions, we ensure that Go-Explore is evaluated under conditions that are at least as difficult as those presented in recent papers by evaluating Go-Explore with *both* sticky actions and no-ops. All Go-Explore scores in this paper come from evaluations with both of these forms of stochasticity combined.

14   PPO and SIL   Both the robustification "backward" algorithm and the implementation of policy-based Go-Explore are based on the actor-critic-style PPO algorithm from Schulman *et al.* (2017)[20], wherein N parallel actors collect data in mini-batches of T timesteps, and policy

updates are performed after each batch. In all PPO-based algorithms presented here, the loss of the policy and value function (both parameterized by θ) is defined as

$$\mathcal{L}(\theta) = \mathcal{L}^{PG}(\theta) + w_{VF}\mathcal{L}^{VF}(\theta) + w_{ENT}\mathcal{L}^{ENT}(\theta) + w_{L2}\mathcal{L}^{L2} + w_{SIL}\mathcal{L}^{SIL}(\theta) \tag{9}$$

$\mathcal{L}^{PG}(\theta)$ is the policy gradient loss with PPO clipping, defined as

$$\mathcal{L}^{PG}(\theta) = \mathbb{E}_{s,a \sim \pi_\theta}[\max(-A_t r_t^\pi(\theta), -A_t \text{clip}(r_t^\pi(\theta), 1 - \epsilon, 1 + \epsilon))] \tag{10}$$

$$r_t^\pi(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{11}$$

where s is a state, a is an action sampled from the policy π_θ, r_t is the reward obtained at time-step t, and ϵ is a hyperparameter limiting how much the policy can be changed at each epoch. A_t is a truncated version of the generalised advantage estimation:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{12}$$

$$\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \tag{13}$$

where γ is the discount factor, λ interpolates between a 1-step return and a T-step return, V_θ is the value function parameterised by θ, and s_t and r_t are the state and reward at time step t, respectively. Similar to the policy gradient loss, the value function loss $\mathcal{L}^{VF}(\theta)$ implemented here also includes PPO-based clipping, and is defined as

$$\mathcal{L}^{VF}(\theta) = \mathbb{E}_{s,a \sim \pi_\theta}[\max((V_\theta(s_t) - \hat{R}_t)^2, (\text{clip}(V_\theta(s_t) - V_{\theta_{old}}(s_t), -\epsilon, \epsilon) - \hat{A}_t)^2)] \tag{14}$$

$$\hat{R}_t = V_{\theta_{old}}(s_t, g_t) + \hat{A}_t \tag{15}$$

The algorithms further include entropy regularization[104] $\mathcal{L}^{ENT}(\theta)$ and an L2 regularization penalty $\mathcal{L}^{L2}(\theta)$.

Finally, both the robustification algorithm and policy-based Go-Explore include a self-imitation learning (SIL) loss[37], $\mathcal{L}^{SIL}(\theta)$, which is calculated over previously collected data $\mathcal{D}$. While the source of $\mathcal{D}$ differs between the robustification algorithm and policy-based Go-Explore (see SI "Multiple demonstrations" and Methods "Policy-based Go-Explore" for details), in both cases $\mathcal{D}$ comes from previously collected trajectories τ and consists of tuples (s, a, R), where s is a state

encountered in one of these rollouts, a is the action that was taken in that state, and R is the discounted reward that was collected from that state. With SIL, a small number (N_{SIL}) of PPO's actors are assigned to be SIL actors. Each of these SIL actors, instead of taking actions in the environment, replays one of these trajectories τ, and at each iteration the data collected by these SIL actors forms the data set $\mathcal{D}$. The $\mathcal{L}^{SIL}(\theta)$ loss is then calculated as

$$\mathcal{L}^{SIL}(\theta) = \mathcal{L}^{SIL\_PG}(\theta) + w_{SIL\_VF}\mathcal{L}^{SIL\_VF}(\theta) + w_{SIL\_ENT}\mathcal{L}^{SIL\_ENT}(\theta) \qquad (16)$$

$$\mathcal{L}^{SIL\_PG}(\theta) = \mathbb{E}_{s,a,R\in\mathcal{D}}[-\log\pi_\theta(a|s) \cdot \max(0, R - V_{\theta_{old}}(s))] \qquad (17)$$

$$\mathcal{L}^{SIL\_VF}(\theta) = \mathbb{E}_{s,a,r\in\mathcal{D}}\left[\frac{1}{2}\max(0, R - V_\theta(s))^2\right] \qquad (18)$$

Here, $\mathcal{L}^{SIL\_ENT}(\theta)$ is the entropy regularization term[104] calculated by evaluating the current policy over $\mathcal{D}$. An investigation of the effect of the SIL loss is provided in Sec. "Ablations".

All architectures that feature recurrent units are updated with a variant of the truncated back-propagation-through-time algorithm called BPTT$(h; h')$[105], which is suitable when data is gathered in mini-batches. In BPTT$(h; h')$, the network is updated every h' timesteps (here $h' = T$, the number of timesteps in the mini-batch), but it is unfolded for $h \geq h'$ timesteps (here $h = 1.5T$), meaning that gradients can flow beyond the boundaries of the mini-batch. To facilitate BPTT$(h; h')$, the first mini-batch of each run consists of $1.5T$ timesteps.

15   Backward algorithm details This section explains the details about how we use multiple demonstrations and reward scaling in the backward algorithm.

15.1   Multiple demonstrations The original version of the backward algorithm relied on a single demonstration due to the assumption that obtaining human demonstrations was expensive. In our case, however, obtaining multiple demonstrations is easy and cheap by simply re-running the exploration phase, which is why we modified the algorithm to utilise multiple demonstrations: at the start of each episode, a demonstration is chosen at random to provide the starting point for the agent. For Atari, 10 demonstrations from different runs of the exploration phase were used for each robustification. In Atari, the demonstration was extracted by finding all trajectories that reached an end-of-episode state (to prevent selection of length 0 trajectories in games with exclusively negative rewards, see "Score tracking in the exploration phase"), and extracting the shortest one among those with the highest score. For robotics, because it was possible to extract diverse demonstrations from a single run of the exploration phase (this was not possible in Atari because high-scoring

trajectories within a given Atari run tended to share most of their actions), 10 demonstrations from the same runs were used for robustification. In robotics, the first demonstration corresponds to the shortest successful trajectory (i.e. the shortest trajectory that puts the object in the shelf), while each subsequent demonstration corresponds to the successful trajectory with the highest mean difference from all previously selected trajectories, where the difference between two trajectories is given by $\frac{\sum_{i=1}^{L} I(\tau_i^a \neq \tau_i^b)}{L}$ (τ^a and τ^b are the list of actions being compared, $L = \min(|\tau^a|, |\tau^b|)$, I is the indicator function). Because actions are continuous, meaning that it is exceedingly unlikely for two independently sampled actions to be the same, $\tau_i^a = \tau_i^b$ only for parts where the trajectories are identical because they were branched from the same intermediate trajectory. As such, this metric effectively measures to what degree the two trajectories have a shared history, and prefers trajectories that share as little history as possible. In both cases, SIL was also performed on the set of demonstrations provided to the backward algorithm (see the "PPO and SIL" section). Each robustification run used demonstrations from different, non-overlapping exploration phase runs (10 exploration runs for Atari, 1 for robotics).

On Atari, the score that can be obtained by starting the algorithm from the start of the environment is tracked throughout the run by adding a virtual demonstration of length 0, i.e. traditional training that executes the current policy from the domain's traditional starting state. This addition makes it possible to occasionally obtain superhuman policies even when the backward algorithm has not yet reached the starting point of any of the non-virtual demonstrations it was provided. During training, the time limit of an episode is the remaining length of the demonstration (which is generally much shorter than the environment time limit, especially at the beginning of training since we start at the end of demonstrations and move backwards) plus a few extra frames (Extended Data Table 1a). When the virtual demonstration is selected, however, the time limit is that of the underlying environment (Extended Data Table 1b). As a result, training episodes in which the virtual demonstration was selected often require many more frames to complete than those corresponding to an exploration phase demonstration. To balance the number of frames allocated to the virtual demonstration, the average number of steps in an episode corresponding to the virtual demonstration (l_v) is tracked as well as the average number of steps corresponding to starting from any other demonstration (l_d), and the selection probability of the virtual demonstration is then $\frac{1}{11}\frac{l_d}{l_v}$, where 11 is the total number of demonstrations (10 from the exploration phase runs and 1 virtual demonstration). In cases where the virtual demonstration was not stochastically chosen, one of the exploration phase demonstrations was chosen uniformly at random.

15.2 Reward scaling A key difficulty in implementing an RL algorithm that can perform well across all Atari games with identical hyperparameters is the significant variations in reward scales within the Atari benchmarks, with some games having an average reward of 1 and others with average rewards of over 1,000. Traditionally, this challenge has been addressed with reward *clipping*, in which all rewards are clipped to either -1 or +1, but such an approach is problematic in games (e.g. Pitfall and Skiing) in which the scale of rewards within the game is relevant because it tells the agent the relative importance of different rewarded (or punished) actions. In this work, we take advantage of the fact that the deterministic exploration phase is unaffected by reward scale and can provide us with a sense of the scale of scores achievable in each game. We are thus able to use reward *scaling* in the robustification phase: at the start of the robustification phase, the rewards of the demonstrations are used to produce a reward multiplier that will result in every game having approximately the same value function scale. This reward multiplier is given by

$$m = \frac{C}{\mu_V} \tag{19}$$

where C is a constant representing the target average absolute value of the value function when following the demonstration (in all our experiments, $C = 10$), and μ_V is defined as

$$\mu_V = \frac{1}{\sum_{d=1}^{D} T_d} \sum_{d=1}^{D} \sum_{t=1}^{T_d} |V_d(t)| \tag{20}$$

where D is the number of demonstrations, T_d is the number of steps in each demonstration, and $V_d(t)$ is the sum of discounted rewards in demonstration d starting from step t.

16 Score tracking in the exploration phase In Atari, the score of an exploration phase run is measured as the highest score ever achieved at episode end. In our implementation, this score is tracked by maintaining a virtual cell corresponding to the end of the episode. An alternative approach would be to track the maximum score across all cells in the archive, regardless of whether they correspond to the end of episode, but this approach fails in games where rewards can be negative (e.g. Skiing): in these cases, it is possible that the maximum scoring cell in the archive inevitably leads to future negative rewards, and is therefore not a good representation of the maximum score achievable in the game. In practice, for games in which rewards are non-negative, the maximum score at end of episode is usually equal or close to the maximum score achieved in the entire archive.

17 Robustification scores analysis Since the robustification phase is intended to train a robust policy from exploration phase trajectories, it might be expected that it would produce policies that approximately replicate the performance of the original trajectories. While this is observed in the robotics environment and in several Atari games (Bowling, Freeway, Solaris, and Pitfall without domain knowledge), there are both positive and negative score differences on some other games on Atari. Negative differences (Gravitar and Venture) occur when PPO struggles to match the performance of the original demonstration at a particular point in the trajectory, resulting in the backward algorithm failing to move the training starting point to the beginning of the trajectory. In spite of these difficulties, the exposure to a larger part of the state space as well as the SIL frames allow the robustification phase to exceed human and state-of-the-art performance even in such cases of partial failure. Interestingly, there are also several cases (Berzerk, Centipede, Montezuma's Revenge, Private Eye, Skiing, and Pitfall with domain knowledge) where the robustified policies obtain substantially higher scores than the trajectories discovered during the exploration phase, demonstrating that the robustification process can have benefits that go beyond merely providing robustness to stochasticity. There are three reasons for this: First, while the exploration phase does optimise for score by updating trajectories to higher scoring ones, the robustification phase is more effective at fine-grained optimisation due to its underlying use of an RL algorithm. For example, if the exploration phase wasted a few frames by bumping into a wall while going from one cell to another, PPO will easily be able to optimise that path, which can impact the final score in time-based games. Second, the robustified policy may generalise patterns that are discovered in the exploration phase. An extreme example is found in Montezuma's Revenge with domain knowledge exploration. Because the exploration phase is able to reach the end of level 3, after which Montezuma's Revenge keeps repeating this level indefinitely (with some stochastic events due to sticky actions), the robustified policy learns to achieve arbitrarily high scores by repeatedly solving this level. Finally we provide the backward algorithm with demonstrations from multiple (10) runs of the exploration phase, thus allowing it to follow the best trajectory in the sample while still benefiting from the data contained in worse trajectories.

18 Comparing Go-Explore and Agent57 The creation of the Atari benchmark started the search for reinforcement learning algorithms capable of achieving super-human performance on all games in this benchmark[26]. For a majority of these games, super-human performance was reached quickly through early deep reinforcement learning techniques now considered standard[15], but for a small set of games super-human performance remained out of reach. The work presented in this

paper achieved this historic feat concurrently with an algorithm called Agent57[13]. Go-Explore and Agent57 accomplish this milestone via very different methods, offering the scientific community a diversity of promising tools to use and build upon going forward.

Agent57 is built upon the Never Give Up (NGU) algorithm[31]. NGU was able to achieve super-human performance on the majority of the Atari games by combining within-episode and across-episode intrinsic motivation, tracking many Q-functions that each maintain a different trade-off between intrinsic and extrinsic motivation, and implementing efficient parallelization of data collection. Agent57 elevated the performance of NGU to superhuman on all games by dynamically learning which of its many Q-functions provides the highest cumulative reward, stabilizing the learning of those Q-functions, and by running the algorithm for an impressive 100 billion frames. So, while Agent57 achieved the milestone of superhuman performance on the last few remaining games at the same time as Go-Explore, its method is vastly different from Go-Explore.

With respect to results, it is first of all important to reiterate that Go-Explore was evaluated in an environment with sticky actions (i.e. following community standards, see SI "No-ops and sticky actions") while Agent57 was evaluated in an environment without sticky actions. Sticky actions make the games substantially harder to play well, which is why Agent57 was not considered for direct comparison with Go-Explore in the main paper (see Methods "State of the art on Atari").

Despite the fact that Go-Explore solutions were evaluated under more difficult conditions, Go-Explore still outperforms Agent57 on 7 out of the 11 games that we tested (Table 2). It is also worth noting that the Go-Explore results reported here were obtained after a total of 30 billion (or 40 billion for Solaris) frames of training data, while Agent57 was trained for 100 billion frames. While Go-Explore does "skip" frames by reloading simulator state, we argue that these frames would also be skipped in almost any scenario where Go-Explore is practically applied, as it should be possible to save and restore the state of a modern simulator. That said, the relative sample efficiency of policy-based Go-Explore (e.g. 97,728 points on Montezuma's Revenge after 12 billion frames) suggests that policy-based Go-Explore could be more sample efficient than Agent57 even if states can not be restored, though the fact that policy-based Go-Explore was only tested with domain knowledge makes it impossible to provide a fair comparison at this time.

19   ALE issues   While the Arcade Learning Environment (ALE)[26], which is the underlying back-end of OpenAI Gym, is the the standard way to interface with Atari games in RL, the library comes with a couple issues that needed to be addressed in our work.

Game	Go-Explore	Agent57
Berzerk	**197,376**	61,508
Bowling	**260**	251
Centipede	**1,422,628**	412,848
Freeway	**34**	33
Gravitar	7,588	**19,214**
MontezumaRevenge	**43,791**	9,352
Pitfall	6,954	**18,756**
PrivateEye	**95,756**	79,717
Skiing	**-3,660**	-4,203
Solaris	19,671	**44,200**
Venture	2,281	**2,628**

Supplementary Table 2: **Go-Explore outperforms Agent57 on 7 out of the 11 games that we tested.** Here we show the results of the Go-Explore variant where the exploration phase was performed without domain knowledge and with restoration of simulator state. The Go-Explore results were obtained by re-evaluating the final agent 1,000 times on the environment with sticky actions and no-ops.

First, the score on Montezuma's Revenge rolls over (i.e. is subject to numerical overflow) when it exceeds 1 million, which is incorrectly interpreted by the ALE as a negative reward of -1 million. We patched the environment to remove this bug and thereby make it possible for algorithms to learn to produce scores higher than 1 million. In addition, we removed an arbitrary time limit of 400,000 frames, imposed by OpenAI Gym, that is not inherent to the game. This enabled us to learn that Go-Explore can substantially outperform the human world record of 1.2 million[33], with one agent frequently reaching a score of over 40 million after 12.5 million frames (the equivalent of about 58 hours of continuous game play). On Montezuma's Revenge, no previous work had achieved scores anywhere near high enough to trigger this bug. It is similarly unlikely that the performance of previous work was limited by the OpenAI Gym time limit on Montezuma's Revenge.

Second, the implementation of Montezuma's Revenge in the ALE library includes a bug that prevents the agent from progressing to the next level when the agent is on its last life, which is clearly unintended behaviour that does not occur in the original game. Because there are no penalties for losing a life, policy-based Go-Explore learns to sacrifice lives in order to bypass hazards or to return to the entrance of a room more quickly. As a result, policy-based Go-Explore frequently reaches the treasure room without any lives remaining, preventing further progress. As such, for policy-based Go-Explore only, we terminate the episode on first death, which avoids this bug without simplifying the game.

20 Infrastructure In terms of infrastructure, each exploration phase run was performed on a single worker machine equipped with 44 CPUs and 96GB of RAM, though memory usage is

substantially lower for most games. Each robustification run was parallelized across 8 worker machines each equipped with 11 CPUs, 24GB of RAM, and 1 GPU. Policy-based Go-Explore was parallelized across 16 worker machines each equipped with 11 CPUs, 10GB of RAM, and 1 GPU.

References

68. Cho, K., Van Merriënboer, B., Bahdanau, D. & Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

69. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The Arcade Learning Environment: An Evaluation Platform for General Agents (Extended Abstract). *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI),* 4148–4152 (2015).

70. Bellemare, M. G., Dabney, W. & Munos, R. *A Distributional Perspective on Reinforcement Learning* in *ICML* (2017).

71. Van Hasselt, H., Guez, A. & Silver, D. *Deep Reinforcement Learning with Double Q-Learning.* in *AAAI* **2** (2016), 5.

72. Stanton, C. & Clune, J. Deep Curiosity Search: Intra-Life Exploration Improves Performance on Challenging Deep Reinforcement Learning Problems. *CoRR* **abs/1806.00553** (2018).

73. Wang, Z., de Freitas, N. & Lanctot, M. *Dueling Network Architectures for Deep Reinforcement Learning* in *ICML* (2016).

74. Salimans, T., Ho, J., Chen, X., Sidor, S. & Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

75. Nair, A. *et al.* Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296* (2015).

76. Xu, H., McCane, B., Szymanski, L. & Atkinson, C. MIME: Mutual Information Minimisation Exploration. *ArXiv* **abs/2001.05636** (2020).

77. Stadie, B. C., Levine, S. & Abbeel, P. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814* (2015).

78. Schrittwieser, J. *et al.* Mastering atari, go, chess and shogi by planning with a learned model. *Nature* **588,** 604–609 (2020).

79. Badia, A. P. *et al.* Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038* (2020).

80. Dann, M., Zambetta, F. & Thangarajah, J. Deriving Subgoals Autonomously to Accelerate Learning in Sparse Reward Domains. *Proceedings of the AAAI Conference on Artificial Intelligence* **33,** 881–889. `https://ojs.aaai.org/index.php/AAAI/article/view/3876` (July 2019).

81. Sovrano, F. *Combining experience replay with exploration by random network distillation* in *2019 IEEE Conference on Games (CoG)* (2019), 1–8.

82. Schaul, T., Quan, J., Antonoglou, I. & Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

83. Kapturowski, S., Ostrovski, G., Quan, J., Munos, R. & Dabney, W. *Recurrent experience replay in distributed reinforcement learning* in *International conference on learning representations* (2018).

84. Hessel, M. *et al. Rainbow: Combining Improvements in Deep Reinforcement Learning* in *AAAI* (2018).

85. Gruslys, A., Azar, M. G., Bellemare, M. G. & Munos, R. The Reactor: A sample-efficient actor-critic architecture. *arXiv preprint arXiv:1704.04651* (2017).

86. Bellemare, M. G., Veness, J. & Bowling, M. H. *Investigating Contingency Awareness Using Atari 2600 Games* in *AAAI* (2012).

87. Kahn, G., Villaflor, A., Pong, V., Abbeel, P. & Levine, S. Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182* (2017).

88. Lillicrap, T. P. *et al.* Continuous control with deep reinforcement learning. *CoRR* **abs/1509.02971** (2015).

89. Fortunato, M. *et al.* Noisy networks for exploration. *arXiv preprint arXiv:1706.10295* (2017).

90. Nguyen, A., Yosinski, J. & Clune, J. *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), 427–436.

91. Szegedy, C. *et al.* Intriguing properties of neural networks. *ArXiv e-prints* **abs/1312.6199** (2013).

92. Garriga Alonso, A. *Solving Montezuma's Revenge with Planning and Reinforcement Learning* 2017.

93. Dann, M., Zambetta, F. & Thangarajah, J. *Deriving Subgoals Autonomously to Accelerate Learning in Sparse Reward Domains* in *Proceedings of the AAAI Conference on Artificial Intelligence* **33** (2019), 881–889.

94. Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W. & Abbeel, P. *Overcoming exploration in reinforcement learning with demonstrations* in *2018 IEEE International Conference on Robotics and Automation (ICRA)* (2018), 6292–6299.

95. Kraft, D. *et al.* Development of object and grasping knowledge by robot exploration. *IEEE Transactions on Autonomous Mental Development* **2,** 368–383 (2010).

96. Mouret, J.-B. & Clune, J. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).

97. Lehman, J. & Stanley, K. O. *Evolving a diversity of virtual creatures through novelty search and local competition* in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation* (2011), 211–218.

98. Pugh, J. K., Soros, L. B. & Stanley, K. O. Quality Diversity: A New Frontier for Evolutionary Computation. *Front. Robotics and AI* **3.** ISSN: 2296-9144 (2016).

99. Browne, C. *et al.* A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* **4,** 1–43 (2012).

100. Kocsis, L., Szepesvári, C. & Willemson, J. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep* **1** (2006).

101. Sutton, R. S. in *Machine learning proceedings 1990* 216–224 (Elsevier, 1990).

102. Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* **8,** 293–321 (1992).

103. Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O. & Clune, J. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995* (2019).

104. O'Donoghue, B., Munos, R., Kavukcuoglu, K. & Mnih, V. Combining policy gradient and Q-learning. *arXiv preprint arXiv:1611.01626* (2016).

105. Williams, R. J. & Peng, J. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation* **2,** 490–501 (1990).