Article

# First return, then explore

https://doi.org/10.1038/s41586-020-03157-9

Received: 13 May 2020

Accepted: 22 December 2020

Published online: 24 February 2021

Check for updates

Adrien Ecoffet[1,2,3]✉, Joost Huizinga[1,2,3]✉, Joel Lehman[1,2], Kenneth O. Stanley[1,2] & Jeff Clune[1,2]✉

Reinforcement learning promises 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 a thorough exploration of the environment, but creating algorithms that can do so remains one of the central challenges of the field. Here we hypothesize that the main impediment to effective exploration originates from algorithms forgetting how to reach previously visited states (detachment) and 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 returning to such states before intentionally exploring. Go-Explore solves all previously 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 of 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, 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 (for example, 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 (for example, the robot not detouring around obstacles like pets)[8–10].

These challenges motivate designing reinforcement learning 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 (Supplementary Information section 4.1). 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 (Supplementary Information section 4.2). 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 games in the Atari 2600 benchmark provided by the Arcade Learning Environment[13] (ALE), which has been posited as a major milestone in previous work[14–16]. Concurrent work[14] similarly reached this milestone (Supplementary Information section 18), but under easier, mostly deterministic conditions that do not meet community-defined standards[17] for evaluation on Atari. Our descriptions of prior results include only evaluations meeting these standards, unless explicitly mentioned (Methods section 'State-of-the-art performance on Atari'). Go-Explore also surpasses the state of the art on all hard-exploration Atari games (that is, where obtaining rewards requires long sequences of correct actions, meaning randomly sampling actions rarely produces rewards

[1]Uber AI Labs, San Francisco, CA, USA. [2]OpenAI, San Francisco, CA, USA. [3]These authors contributed equally: Adrien Ecoffet, Joost Huizinga. ✉e-mail: adrienecoffet@gmail.com; joost.hui@gmail.com; jclune@gmail.com

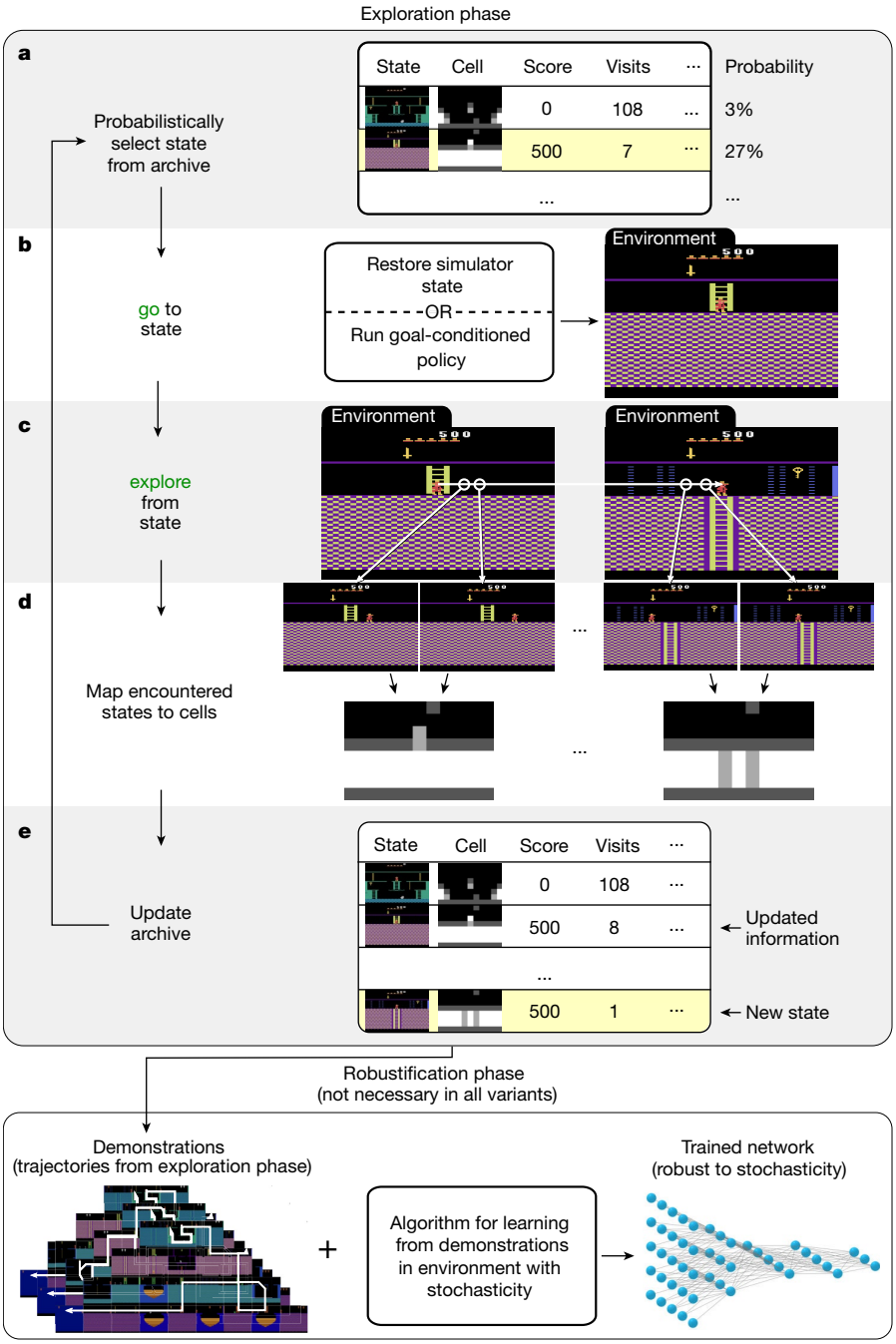
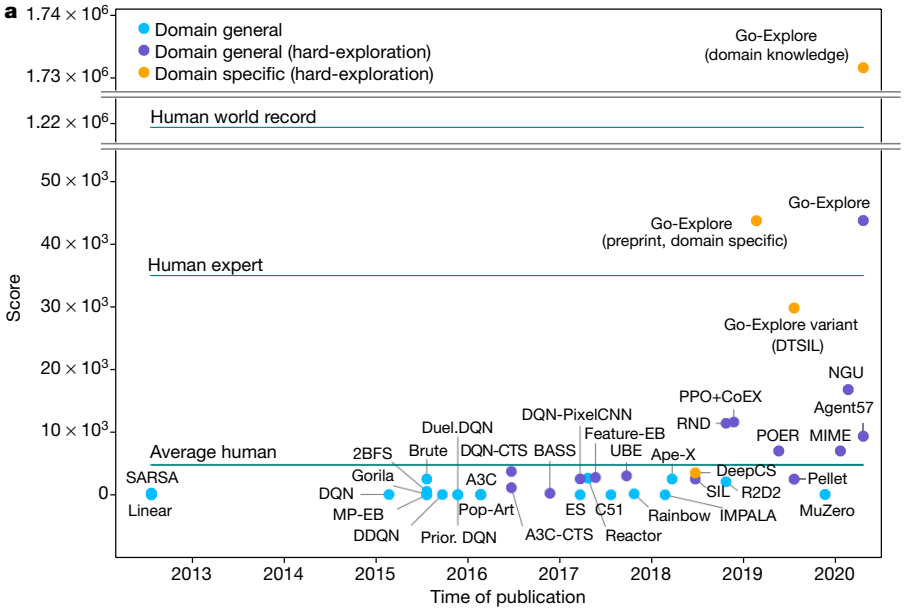



**Fig. 1 | Overview of Go-Explore. a**, Probabilistically select a state from the archive, preferring states associated with promising cells. **b**, Return to the selected state, such as by restoring a simulator state or by running a goal-conditioned policy. **c**, Explore from that state by taking random 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.

and thus more-intelligent 'exploration' is needed). Additionally, 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 (for example, the archive can be considered a frontier, the 'explore' step represents expanding a node, and so on), the potential of which have been relatively unappreciated within deep reinforcement learning research. However, for problems focused on by the reinforcement learning community (such as hard-exploration Atari games), which are high-dimensional with sparse rewards and/or stochasticity, no




b

| Game | Exploration phase | Robustification phase | State-of-the-art performance | Average 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 |

**Fig. 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 that has been the primary focus of hard-exploration research for many years. Different methods use different amounts of computing power. Go-Explore processed a similar number of frames (30 billion) as other distributed reinforcement learning algorithms like Ape-X (22 billion) and NGU (35 billion). **b**, Go-Explore 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 two 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 Supplementary Information section 17. A video of high-performing runs can be found at https://youtu.be/u6_Ng2oFzEY. For citations to the listed algorithms, see Supplementary Information section 2.

known planning method works[17,18]. Among other reasons (Supplementary Information section 9), such state spaces are too large to search exhaustively (requiring hard-to-define 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 reinforcement learning 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[15,19] or by sampling from a stochastic 'policy'—a function that decides which action to take in each state, often a neural network[20,21]. By first returning before exploring, Go-Explore avoids derailment by minimizing exploration when returning (thus minimizing failure to return) after which it can focus purely 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, e). The archive stores one state per cell, and to maximize 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 to weights that it assigns to their associated cells in the archive (Methods).

Although returning to a previously found state can be done with a trained policy (demonstrated in 'Policy-based Go-Explore'), Go-Explore provides a unique opportunity to leverage the availability and widespread use of simulators in reinforcement learning tasks[7,22–24]. 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 (for example, 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)[25], where the exploration phase trajectories replace the usual human expert demonstrations (similar to a previous work)[26], in a variant of the environment featuring sufficient stochasticity to ensure robustness.

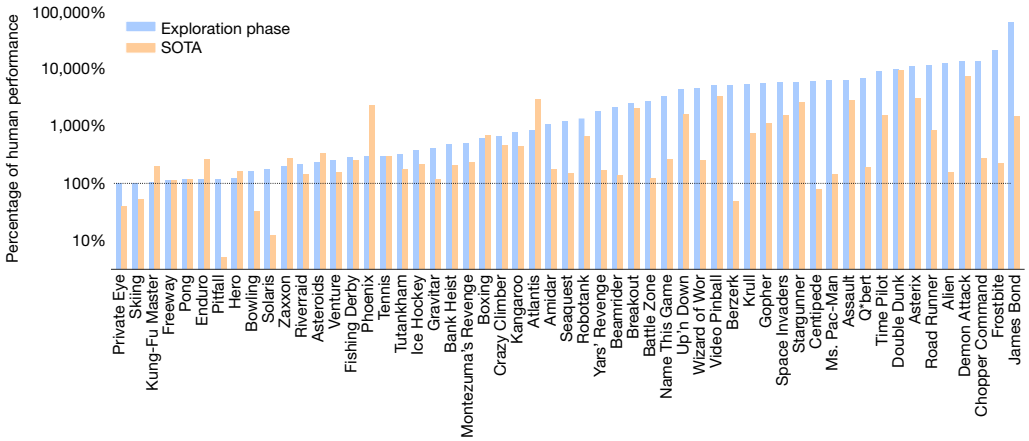



**Fig. 3 | Human-normalized performance of the exploration phase and state-of-the-art algorithms on all Atari games.** The exploration phase of Go-Explore exceeds average human performance in every game, often by orders of magnitude, and outperforms the prior state of the art in most games (details in Extended Data Fig. 2a, Extended Data Table 3).

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 (Supplementary Information section 10). Because it produces robust policies from open-loop (that is, predetermined) trajectories, we call this LFD process the 'robustification phase' (Fig. 1).

## Learning Atari with state restoration

The Atari benchmark suite[13], a prominent benchmark for reinforcement learning 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 directly restores the state of the simulator. In this experiment, the 'explore' step happens through random actions, meaning that 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 greyscale image, which—in contrast to most reinforcement learning preprocessing that reduces dimensionality to save computational resources while minimizing 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 (for example, learned) representations. Good state-to-cell-mapping parameters result in a representation that strikes a balance between two extremes: lack of aggregation (for example, one cell for every frame, which is computationally inefficient) and excessive aggregation (for example, assigning all frames to a single cell, which prevents exploration). Because appropriate downscaling parameters (width, height, and number of possible greyscale values) vary across Atari games (Supplementary Information section 3.2) as well as when exploration progresses within a given game, these parameters are optimized dynamically at regular intervals (Methods section 'Downscaling on Atari'). The hyperparameters of this optimisation procedure are robust and generalize to unseen games (Supplementary Information section 3).

Here the robustification phase consists of a modified version of the 'backward algorithm'[29] that is currently the highest-performing LFD algorithm on Montezuma's Revenge. Owing to the large computational expense of the robustification process, this work focuses on the set of 11 games that have been considered hard-exploration challenges by the community[1] or for which the state-of-the-art performance was still below average human performance (Methods section 'State-of-the-art performance on Atari'). To ensure the trained policy becomes robust to environmental perturbations, during robustification stochasticity is added to these environments following current community standards[17]. The demonstrations provided by the exploration phase provide enough information about available rewards to allow Go-Explore to eschew standard reward clipping—which overemphasizes 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 11 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 the average human performance, whereas previous algorithms were unable to score any points. The number of frames processed in these experiments is 30 billion (Extended Data Figs. 2, 4), similar to that of recent distributed reinforcement learning algorithms[14,27,31]. Although older algorithms often processed fewer frames, many of them show signs of convergence (meaning no further progress is expected), and for many of these algorithms, it is 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 provided by OpenAI gym[32], a feat that has not been performed before (save concurrent work)[14]. In 85.5% of these games the trajectories reach scores higher than those achieved by state-of-the-art reinforcement learning algorithms (Fig. 3). Go-Explore's performance also exceeds that of planning algorithms (which similarly restore simulator states) that were evaluated on Atari[17,18].

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 contains only features relevant for exploration. The domain-knowledge features are the discretized 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, and also surpassing the human

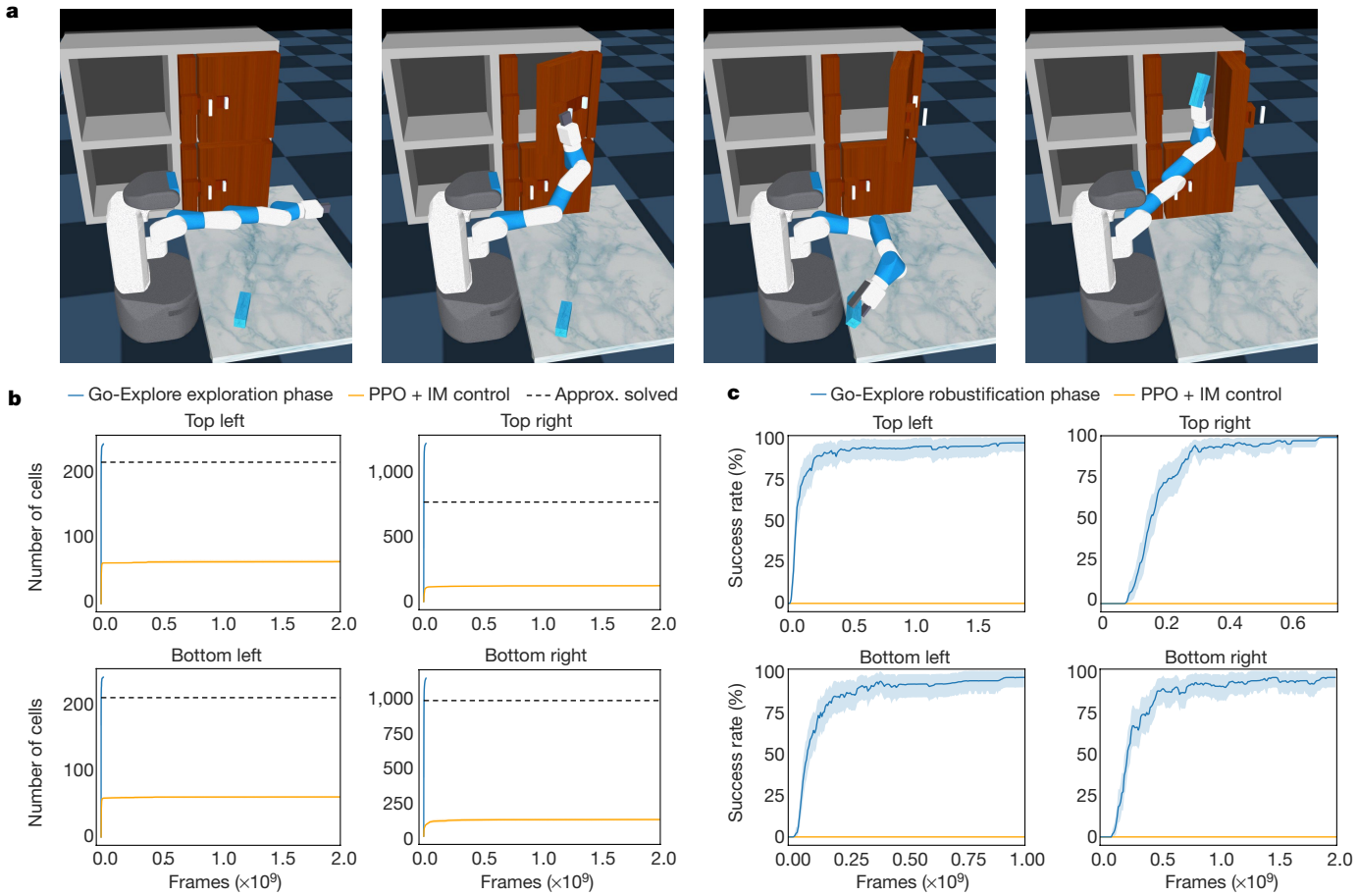



**Fig. 4 | Go-Explore can solve a challenging, sparse-reward, simulated robotics task. a**, A simulated Fetch robot needs to grasp an object and put it in one of four shelves. **b**, The exploration phase substantially outperforms an intrinsic motivation (IM) control using the same cell representation. The 'Approx. solved' line gives a rough indication of the number of cells that need to be discovered to find a successful trajectory. It corresponds to the mean number of cells contained in the archive of the exploration phase runs when they first found a successful trajectory. **c**, Across the 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 (Methods section 'Evaluation'). Lines show the mean over 50 runs for Go-Explore and 10 runs for the PPO + IM control. Shaded areas show 95% bootstrap confidence intervals (CIs) of the mean with 1,000 samples.

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), in effect discovering every unique location in each game (Supplementary Information section 5). Previous work suggests that intrinsic motivation algorithms benefit far less from domain knowledge; a count-based exploration algorithm with the same domain-knowledge representation scores 12,240 on Montezuma's Revenge[34].

## A hard-exploration robotics environment

Although robotics is a promising application for reinforcement learning and it is often easy to define the high-level goal of a robotics task (for example, to put a cup in a cupboard), it is much more difficult to define a sufficiently dense reward function[10] (for example, reward all of the low-level motor commands to move towards the cup, grasp it, and so forth). 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[7,22–24], making robotics a natural domain to demonstrate the usefulness of harnessing the ability to restore simulator states.

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 reinforcement learning algorithm for continuous control (proximal policy optimization, PPO)[21] 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, Extended Data Fig. 5a). Go-Explore succeeds because it thoroughly explores its environment without suffering from detachment (for example, 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). By 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 two billion frames of training, 100 times more than the exploration phase (Fig. 4b). Despite receiving intrinsic rewards for touching the object, this control

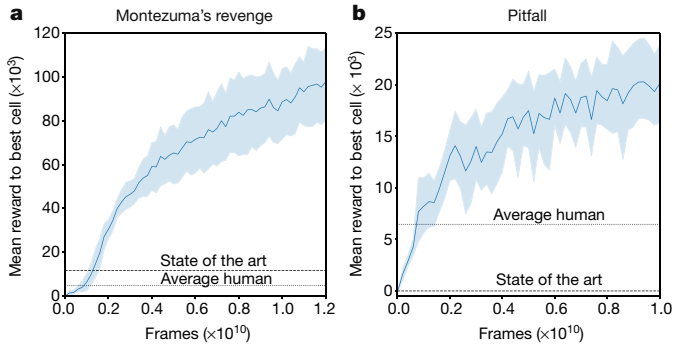



**Fig. 5 | Policy-based Go-Explore with domain knowledge outperforms state-of-the-art and average human performance in Montezuma's Revenge and Pitfall. a**, **b**, On both Montezuma's Revenge (**a**) and Pitfall (**b**), Go-Explore's performance increases throughout the run, suggesting even higher performance is possible with additional training time. Lines show the mean over 10 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

was incapable of learning to reliably grasp objects. Evidence suggests that this failure to grasp is due to the problem of 'derailment' (Supplementary Information section 7), which Go-Explore is specifically designed to solve. Robustifying the trajectories found by Go-Explore produces robust policies in 99% of cases (Fig. 4c).

## Policy-based Go-Explore

Leveraging the ability of simulators to restore to states increases Go-Explore's efficiency, but it is not a requirement. When returning, instead of restoring a simulator state, it is possible to execute a policy conditioned on (that is, 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 versus taking random actions, because the policy can generalize to new situations—for example, 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 (for example, stochastic rewards; Supplementary Information section 11).

The goal-conditioned policy is trained during the exploration phase with a common reinforcement learning algorithm (PPO)[21]. Because goal-conditioned policies often struggle to reach distant states[36] (Supplementary Information section 3.9), 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 stabilize learning, the most important of which are self-imitation learning[37] (Supplementary Information section 3.8), 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 average 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 (Supplementary Information section 12). 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 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 reinforcement learning.

## 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[7,22–24], 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 generalize 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 specialized 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.

## Online content

Any methods, additional references, Nature Research reporting summaries, source data, extended data, supplementary information, acknowledgements, peer review information; details of author contributions and competing interests; and statements of data and code availability are available at https://doi.org/10.1038/s41586-020-03157-9.


1. Bellemare, M. et al. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems 29 (NIPS 2016)* (eds Lee, D. et al.) 1471–1479 (2016).
2. Lehman, J. & Stanley, K. O. Novelty search and the problem with objectives. In *Genetic Programming Theory and Practice IX* (eds Riolo, R. et al.) 37–56 (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. Open AI. Dota 2 with large-scale deep reinforcement learning. Preprint at https://arxiv.org/abs/1912.06680 (2019).

6. Merel, J. et al. Hierarchical visuomotor control of humanoids. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=BJfYvo09Y7 (2019).
7. Open AI. Learning dexterous in-hand manipulation. *Int. J. Robot. Res.* **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. *Artif. Life* **26**, 274–306 (2020).
9. Amodei, D. et al. Concrete problems in AI safety. Preprint https://arxiv.org/abs/1606.06565 (2016).
10. Smart, W. D. & Kaelbling, L. P. Effective reinforcement learning for mobile robots. In *Proc. 2002 IEEE Int. Conf. Robotics and Automation* 3404–3410 (IEEE, 2002).
11. Lehman, J. & Stanley, K. O. Abandoning objectives: evolution through the search for novelty alone. *Evol. Comput*. **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 31 (NeurIPS 2018)* (eds Bengio S. et al.) 5027–5038 (2018).
13. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The Arcade Learning Environment: an evaluation platform for general agents. *J. Artif. Intell. Res.* **47**, 253–279 (2013).
14. Puigdomènech Badia, A. et al. Agent57: outperforming the Atari human benchmark. In *Int. Conf. Machine Learning* 507–517 (PMLR, 2020).
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. In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)* (eds Bengio, S. et al.) 2930–2941 (2018).
17. 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).
18. Lipovetzky, N., Ramirez, M. & Geffner, H. Classical planning with simulators: results on the Atari video games. In *IJCAI'15 Proc. 24th Int. Conf. Artificial Intelligence* (eds Yang, Q. & Woolridge, M.) 1610–1616 (2015).
19. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* (Bradford, 1998).
20. Mnih, V. et al. Asynchronous methods for deep reinforcement learning. In *Proc. 33rd Int. Conf. Machine Learning* (eds Balcan, M. F. & Weinberger, K. Q.) 1928–1937 (2016).
21. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal policy optimization algorithms. Preprint at https://arxiv.org/abs/1707.06347 (2017).
22. Cully, A., Clune, J., Tarapore, D. & Mouret, J.-B. Robots that can adapt like animals. *Nature* **521**, 503–507 (2015).
23. Peng, X. B., Andrychowicz, M., Zaremba, W. & Abbeel, P. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE Int. Conf. Robotics and Automation (ICRA)* (ed. Lynch, K.) 3803–3817 (IEEE, 2018).
24. Tan, J. et al. Sim-to-real: learning agile locomotion for quadruped robots. In *Proc. Robotics: Science and Systems* (eds Kress-Gazit, H. et al.) https://doi.org/10.15607/RSS.2018.XIV.010 (2018).
25. Hester, T. et al. Deep Q-learning from demonstrations. In *Thirty-Second AAAI Conf. Artificial Intelligence* 3223–3230 (2018).
26. 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 *Advances in Neural Information Processing Systems 27 (NIPS 2014)* (eds Ghahramani, Z. et al.) 3338–3346 (2014).
27. Horgan, D. et al. Distributed prioritized experience replay. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=H1Dy---0Z (2018).
28. Espeholt, L. et al. IMPALA: scalable distributed deep-RL with importance weighted actor-learner architectures. In *Proc. 35th Int. Conf. Machine Learning* (eds Dy, J. & Krause, A.) 1407–1416 (2018).
29. Salimans, T. & Chen, R. Learning Montezuma's Revenge from a single demonstration. Preprint at https://arxiv.org/abs/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 29 (NIPS 2016)* (eds Lee, D. et al.) 4287–4295 (2016).
31. Puigdomènech Badia, A. et al. Never give up: learning directed exploration strategies. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=Sye57xStvB (2020).
32. Brockman, G. et al. OpenAI gym. Preprint at https://arxiv.org/abs/1606.01540 (2016).
33. ATARI VCS/2600 Scoreboard. *Atari Compendium* http://www.ataricompendium.com/game_library/high_scores/high_scores.html (accessed 6 January 2020).
34. Guo, Y. et al. Efficient exploration with self-imitation learning via trajectory-conditioned policy. Preprint at https://arxiv.org/abs/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 of the Intl Joint Conf. Artificial Intelligence* (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 32 (NeurIPS 2019)* (eds Wallach, H. et al.) 15220–15231 (2019).
37. Oh, J., Guo, Y., Singh, S. & Lee, H. Self-imitation learning. In *Proc. 35th Int. Conf. Machine Learning* (eds Dy, J. & Krause, A.) 3878–3887 (2018).
38. Madotto, A. et al. Exploration-based language learning for text-based games. Preprint at https://arxiv.org/abs/2001.08868 (2020).
39. Popova, M., Isayev, O. & Tropsha, A. Deep reinforcement learning for de novo drug design. *Sci. Adv.* **4**, eaap7885 (2018).
40. Alvernaz, S. & Togelius, J. Autoencoder-augmented neuroevolution for visual Doom playing. In *2017 IEEE Conf. Computational Intelligence and Games (CIG)* 1–8 (IEEE, 2017).
41. Cuccu, G., Togelius, J. & Cudré-Mauroux, P. Playing Atari with six neurons. In *Proc. 18th Intl Conf. Autonomous Agents and MultiAgent Systems* 998–1006 (2019).
42. Oord, A. d., Li, Y. & Vinyals, O. Representation learning with contrastive predictive coding. Preprint at https://arxiv.org/abs/1807.03748 (2018).
43. Jaderberg, M. et al. Reinforcement learning with unsupervised auxiliary tasks. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=SJ6yPD5xg (2017).
44. Chaslot, G., Bakkes, S., Szita, I. & Spronck, P. Monte-Carlo tree search: a new framework for game AI. In *AIIDE'08: Proc. Fourth AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment* (eds Darken, C. & Mateas, M.) 216–217 (2008).
45. Lavalle, S. M. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Technical Report No. 98-11 (Iowa State Univ., 1998).
46. Hart, P. E., Nilsson, N. J. & Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* **4**, 100–107 (1968).
47. Smith, D. E. & Weld, D. S. Conformant Graphplan. In *AAAI '98/IAAI '98: Proc. 15th Natl/10th Conf. Artificial Intelligence/Innovative Applications of Artificial Intelligence* (eds Mostow, J. et al.) 889–896 (1998).


**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature Limited 2021

# Methods

### State-of-the-art performance on Atari

With new work on reinforcement learning for Atari being published on a regular basis, and with reporting methods often varying greatly, 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 (for example, continuing to tilt the joystick for a fraction of a second longer than intended), as a way to evaluate agents on Atari[17]. There is substantial evidence to show that sticky actions can decrease performance substantially compared to the now deprecated 'no-ops' (no-operations) evaluation strategy[17,48,49]. As a result, we exclude work that was only evaluated with no-ops from our definition of state of the art. Figure 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 individualized scores for each game. To avoid cherry-picking lucky rollouts that can substantially bias scores upward, 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.[50], Castro et al.[48], Choi et al.[51], Fedus et al.[52], Taiga et al.[53], Tang et al.[54] and Toromanoff et al.[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 ('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 (Supplementary Information section 6).

To obtain the downscaled representation, (1) the original frame is converted to greyscale, (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 $\lfloor dp/255 \rfloor$, 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 generalize across games because visuals (for example, the amount of detail shown on screen and how much it varies across frames) vary substantially between games (Supplementary Information section 3). 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 on the basis of 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, although the algorithm is robust to different values of $T$, see Supplementary Information section 3.4), 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

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

$L(n, T)$ 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

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

$H_n(\mathbf{p})$ is the ratio of the entropy of how frames were distributed across cells to the entropy of the discrete uniform distribution of size $n$, that is, the normalized 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, normalized entropy is comparable across different numbers of cells, allowing the number of cells to be controlled solely by $L(n, T)$. Its form is

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

At each step of the randomized 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 1/20th 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 (Supplementary Information section 3.3 shows that the algorithm is not overly sensitive to the particular setting of the minimum means). New parameter values are resampled if they fall outside of the valid range for that parameter. In our implementation, the randomized 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 (Supplementary Information section 3.5 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 first-in, first-out 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 initialized 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 randomized sweep on Montezuma's Revenge, with the 10 best combinations then tested on Gravitar to ensure their generalizability (it is the norm in hard-exploration work to include Montezuma's Revenge as part of the tuning set[1,14,31,34,50,51,53,55–57], although which other games are included, if any, varies). Aside from the hyperparameters examined in Supplementary Information section 3 (the target proportion, minimum

means and buffer sampling rate), the hyperparameters control the trade-off between computational and memory efficiency and the quality of downscaling parameters obtained (for example, 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 discretized $x, y$ position of the agent. In Go-Explore without a return policy, the $x, y$ position is discretized in 8 by 16 pixel cells. Policy-based Go-Explore uses the coarser-grained 18 by 18 pixel cell representation from Guo et al.[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, and 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 the Atari RAM (random-access memory), 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 recognize 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 three-dimensional position of the robot's gripper, discretized in voxels with sides of length 0.5 m, 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 metres: $\lfloor (d + 0.195)/0.2 \rfloor$.

### Exploration phase

During the exploration phase (Supplementary 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 $C_{\text{seen}}$ is the number of exploration steps in which that cell is visited (that is, the $C_{\text{seen}}$ 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 intuitive notion 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 for example, a ladder to be present), and that an agent has more exploration capacity (that is, affordances) if it is holding more keys. $W_{\text{location}}$ 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:

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

This level-weighting 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 (Supplementary Information section 3.6). Although 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 nine 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 Supplementary Information section 3.1.

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 a simulator state. This method of returning is available whenever a simulator is available, which is the case for most reinforcement learning experiments; owing to the large number of training trials current reinforcement learning algorithms require, as well as the safety concerns that arise when running reinforcement learning directly in the real world, simulators have played a key role in training the most compelling applications of reinforcement learning, 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 (Supplementary Information section 14) 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 stabilizes 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 (Supplementary Information section 15.1). This process was not performed in the robotics environment as there is only one point to score, making partial success impossible. Self-imitation learning[37] was performed on the demonstrations provided to the backward algorithm (Supplementary Information section 14). In Atari, we normalize 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 (Supplementary Information section 15.2). The pseudocode that includes the modifications above is shown in Supplementary 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 (Supplementary Information section 16). For the 11 focus games, exploration-phase scores are averaged across 50 exploration-phase runs. For the other games, they are averaged across five 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 reinforcement learning 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 five runs. For domain knowledge, they are averaged across 10 runs. All testing is performed with sticky actions (see Methods section 'State-of-the-art performance 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 (Supplementary Information section 19).

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 the compartment, a penalty of −1 is given for removing the object from the compartment, and during robustification the agent is given up to 54 additional steps after successfully putting the object in the shelf ('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), two 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, five runs per target shelf of ordinary PPO[21] with no exploration mechanism were run for 1 billion frames. At no point during these runs was any reward found, confirming that the robotics problem in this paper constitutes a hard-exploration challenge. Second, 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 $1/\sqrt{n}$, 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 considerably 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.[17], and the fact that the agent acts every four frames, as is typical in reinforcement learning 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 four. In robotics, the agent acts 12.5 times per second. Each action is simulated with a timestep granularity of 0.001 s, corresponding to 80 simulator steps for every action taken.

Although the robustification algorithm originates from Salimans & Chen[29], it was modified in various ways (Methods section 'The backward algorithm'). Extended Data Table 1a shows the hyperparameters 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 Table 2a, b shows the state representation for robotics robustification.

With the downscaled representation on Atari, the exploration phase was run for 2 billion frames before extracting demonstrations for robustification. Because exploration-phase performance was slightly below average human performance 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). In the robotics experiment, the exploration phase ran for

20 million frames and details for the robustification phase are given in Methods section 'Evaluation'.

### Policy-based Go-Explore details

The idea of 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[21] (Supplementary Information section 14) and pseudocode for the algorithm is shown in Supplementary Algorithm 3. At the heart of policy-based Go-Explore lies a goal-conditioned policy $\pi_\theta(a|s,g)$ (Extended Data Fig. 1c), parameterized by $\theta$, 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 Supplementary Information section 14, except that instances of the state $s$ are replaced with the state-goal tuple $(s,g)$. The total reward $r_t$ at time $t$ is the sum of the trajectory ($\tau$) reward $r_t^\tau$ (defined below) and the environment (e) 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[37] (Supplementary Information section 14), where self-imitation learning actors follow the same procedure as regular actors, except that they replay the trajectory associated with the cell 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 are 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 $C_{\text{steps}}$ is the total number of steps the agent has spent 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, enabling 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 in practice 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', which is 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 (Supplementary Information section 3.9), 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.[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 nine cells in this trajectory, the goal is considered met. When a goal is met, the agent receives a trajectory reward $r_t^\tau$ of 1 and the subsequent goal in the non-repeated trajectory (that is, 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 $r_t^\tau$ 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 (Supplementary Information section 3.10). 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 Methods section '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 section 'Exploration phase'. All gathered data are 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 $\mathcal{L}^{\text{ENT}}$, the policy is extended with an entropy term $e_t$ that divides the logits (inputs to the softmax activation function) of the policy. 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:

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

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 upon death to deal with an ALE bug (Supplementary Information section 19).

### Robotics environment

The robotics 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.

## 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, Supplementary Information, 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.


48. Castro, P. S., Moitra, S., Gelada, C., Kumar, S. & Bellemare, M. G. Dopamine: a research framework for deep reinforcement learning. Preprint at https://arxiv.org/abs/1812.06110 (2018).
49. Toromanoff, M., Wirbel, E. & Moutarde, F. Is deep reinforcement learning really superhuman on Atari? In *Deep Reinforcement Learning Workshop of 39th Conf. Neural Information Processing Systems (NeurIPS 2019)* (2019).
50. Burda, Y., Edwards, H., Storkey, A. & Klimov, O. Exploration by random network distillation. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=H1lJJnR5Ym (2019).
51. Choi, J. et al. Contingency-aware exploration in reinforcement learning. In *Int. Conf. Learning Representations* https://openreview.net/forum?id=HyxGB2AcY7 (2019).
52. Fedus, W., Gelada, C., Bengio, Y., Bellemare, M. G. & Larochelle, H. Hyperbolic discounting and learning over multiple horizons. Preprint at https://arxiv.org/abs/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 *Int. Conf. Learning Representations* https://openreview.net/forum?id=BJewlyStDr (2020).
54. Tang, Y., Valko, M. & Munos, R. Taylor expansion policy optimization. In *Proc. 37th Int. Conf. Machine Learning* (eds Daumé III, H. & Singh, A.) 9397–9406 (2020).
55. Ostrovski, G., Bellemare, M. G., van den Oord, A. & Munos, R. Count-based exploration with neural density models. In *Proc. 34th Int. Conf. Machine Learning* (eds Precup, D. & Teh, Y. W.) 2721–2730 (2017).
56. Martin, J., Sasikumar, S. N., Everitt, T. & Hutter, M. Count-based exploration in feature space for reinforcement learning. In *IJCAI'17: Proc. 26th Int. Joint Conf. Artificial Intelligence* (ed. Sierra, C.) 2471–2478 (2017).
57. O'Donoghue, B., Osband, I., Munos, R. & Mnih, V. The uncertainty Bellman equation and exploration. In *Proc. 35th Int. Conf. Machine Learning* (eds Dy, J. & Krause, A.) 3839–3848 (2018).
58. Goldenberg, A., Benhabib, B. & Fenton, R. A complete generalized solution to the inverse kinematics of robots. *IEEE J. Robot. Autom.* **1**, 14–20 (1985).
59. Spong, M. W., Hutchinson, S., Vidyasagar, M. *Robot Modeling and Control* (Wiley, 2006).
60. Zhao, Z.-Q., Zheng, P., Xu, S.-t. & Wu, X. Object detection with deep learning: a review. *IEEE Trans. Neural Netw. Learn. Syst.* **30**, 3212–3232 (2019).
61. Todorov, E., Erez, T. & Tassa, Y. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ Int. Conf. Intelligent Robots and Systems* 5026–5033 (IEEE, 2012).
62. Kocsis, L. & Szepesvári, C. Bandit-based Monte Carlo planning. In *European Conf. Machine Learning ECML 2006* (eds Fürnkranz, J. et al.) 282–293 (Springer, 2006).
63. Strehl, A. L. & Littman, M. L. An analysis of model-based interval estimation for Markov decision processes. *J. Comput. Syst. Sci.* **74**, 1309–1331 (2008).
64. Tang, H. et al. #Exploration: a study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)* (eds Guyon, I. et al.) 2750–2759 (2017).
65. Ng, A. Y., Harada, D. & Russell, S. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th Int. Conf. Machine Learning* (eds Bratko, I. & Džeroski, S.) 278–287 (1999).
66. Hussein, A., Gaber, M. M., Elyan, E. & Jayne, C. Imitation learning: a survey of learning methods. *ACM Comput. Surv.* **50**, 21 (2017).
67. Plappert, M. et al. Multi-goal reinforcement learning: challenging robotics environments and request for research. Preprint at https://arxiv.org/abs/1802.09464 (2018).
68. Cho, K., Van Merriënboer, B., Bahdanau, D. & Bengio, Y. On the properties of neural machine translation: encoder-decoder approaches. In *Proc. SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation* 103–111 (Association for Computational Linguistics, 2014).



**Acknowledgements** We thank A. Edwards, S. Kapoor, F. Petroski Such and J. Zhi for their ideas, feedback, technical support and work on aspects of Go-Explore not presented in this work. We are grateful to the Colorado Data Center and OpusStack Teams at Uber for providing our computing platform. We thank V. Kumar for creating the MuJoCo files that served as the basis for our robotics environment (https://github.com/vikashplus/fetch).


**Author contributions** A.E. and J.H. contributed equally and are responsible for the technical work (J.H. focused primarily on policy-based Go-Explore and A.E. on most other technical contributions) as well as the initial draft of the paper. J.C. and K.O.S. led the team. All authors (A.E., J.H., J.L., K.O.S. and J.C.) significantly contributed to ideation, experimental design, analysing data, strategic decisions, developing the philosophical motivation for the algorithm and editing the paper.

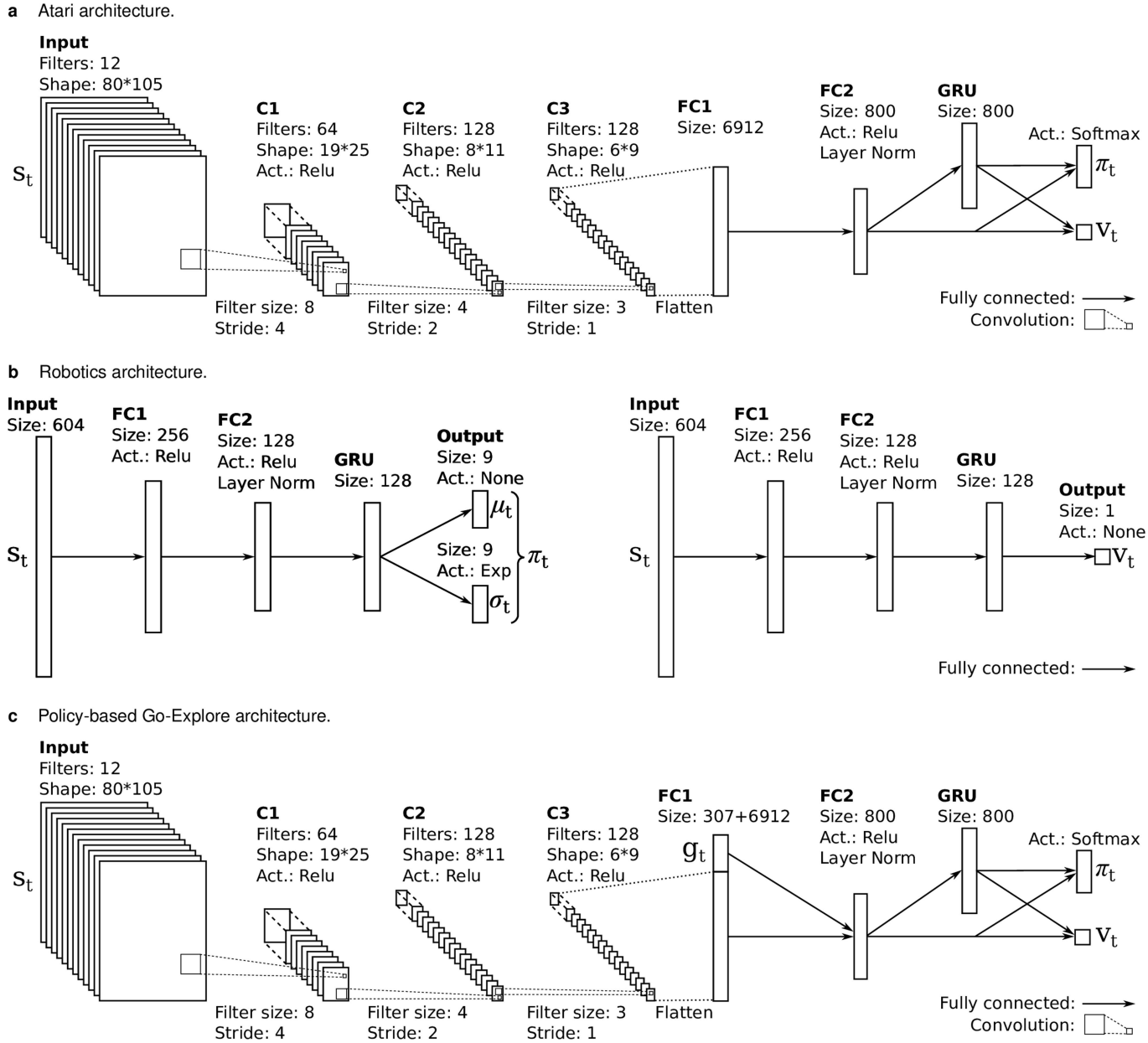
**Competing interests** Uber Technologies, Inc. has filed a publicly available provisional patent application 16/696,893 about some Go-Explore variants featuring a deep reinforcement learning model, with all authors (A.E., J.H., J.L, K.O.S. and J.C.) listed as inventors.


**Additional information**
**Supplementary information** The online version contains supplementary material available at https://doi.org/10.1038/s41586-020-03157-9.


**Correspondence and requests for materials** should be addressed to A.E., J.H. or J.C.

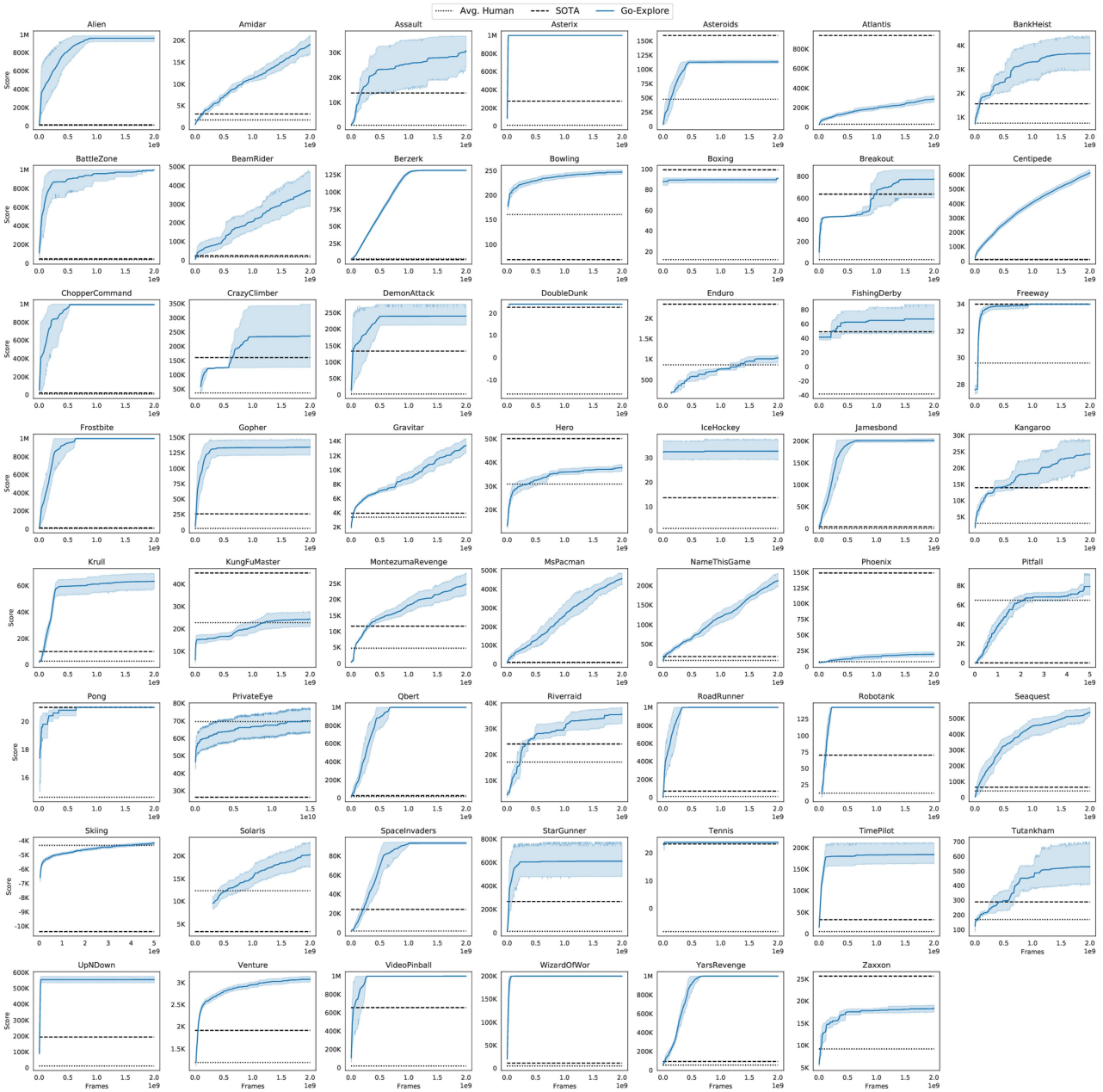

**Peer review information** *Nature* thanks Julian Togelius and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.


**Reprints and permissions information** is available at http://www.nature.com/reprints.

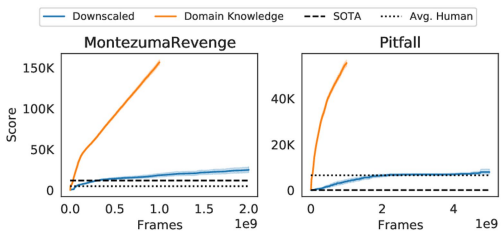



**Extended Data Fig. 1 | Neural network architectures. a**, The Atari architecture is based on the architecture provided with the backward algorithm implementation. The input consists of the RGB channels of the last four frames (rescaled to 80 by 105 pixels) concatenated, resulting in 12 input channels. The network consists of three convolutional layers (C), two fully connected layers (FC), and a layer of gated recurrent units (GRUs)[68]. The network has a policy head $\pi_t(s_t|a_t)$ and a value head $V_t(s_t)$. **b**, For the robotics problem, the architecture consists of two separate networks, each with two fully connected layers and a GRU layer. One network specifies the policy $\pi_t(s_t|a_t)$ by returning a mean $\mu_t$ and variance $\sigma_t$ 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 $V_t(s_t)$. **c**, The architecture for policy-based Go-Explore is identical to the Atari architecture, except that the goal representation $g_t$ is concatenated with the input of the first fully connected layer. Activation functions (Act.) are: the rectified-linear unit (Relu), the exponential function (Exp) and the softmax function (Softmax). Layers can also include layer normalization (Layer norm), which transforms the output of the layer by subtracting the mean and dividing by the standard deviation of the layer.

**a** Exploration phase without domain knowledge.

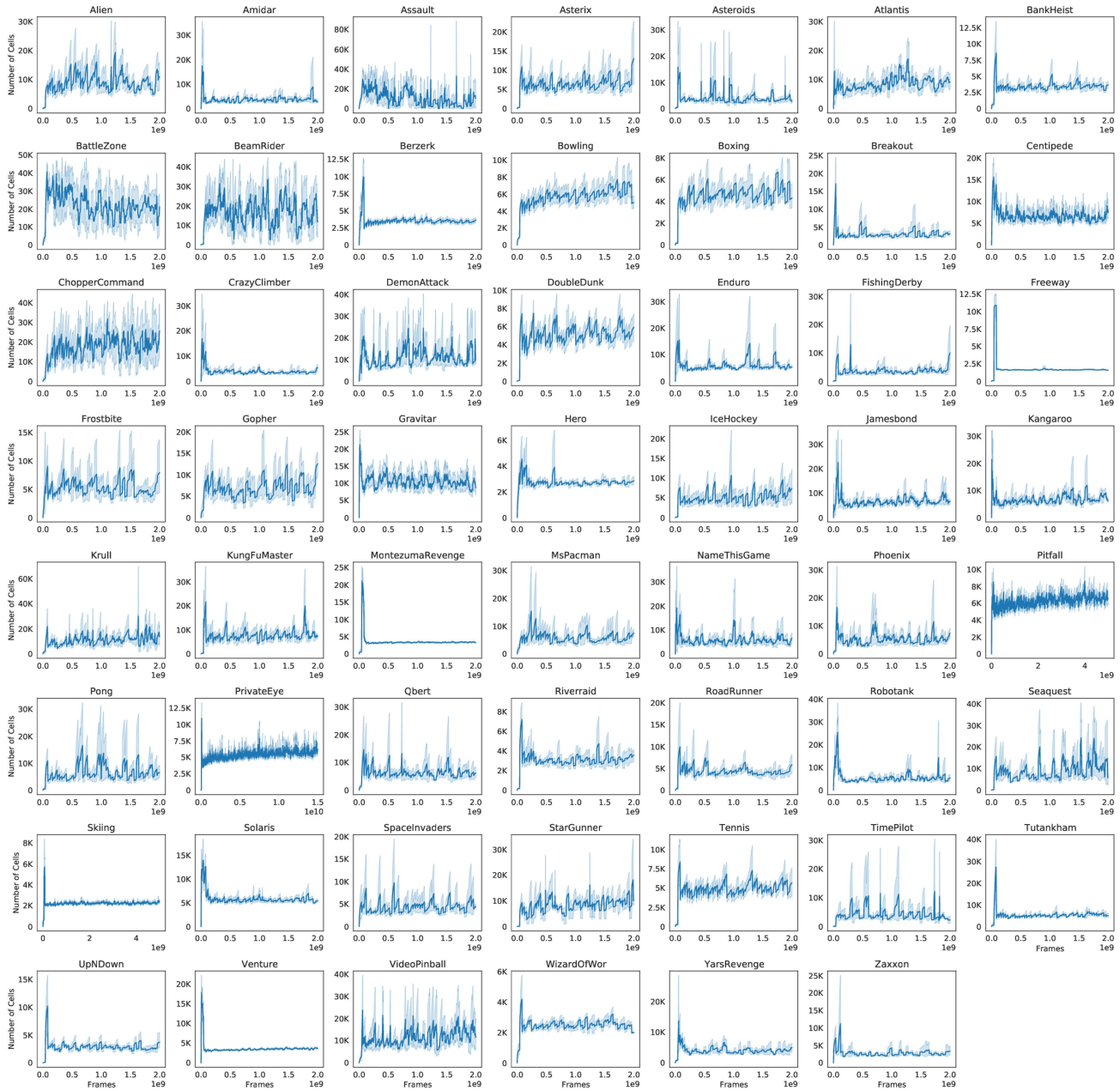



**b** Exploration phase with domain knowledge (compared to downscaled).

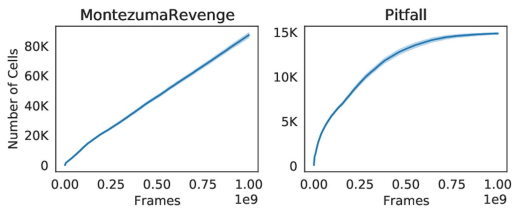



**Extended Data Fig. 2 | Maximum end-of-episode score found by the exploration phase on Atari. a**, Exploration phase without domain knowledge. **b**, Exploration phase with domain knowledge, compared to downscaled. Because only scores achieved at the episode end are reported, the plots for some games (for example, 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 five runs for other games. In **b**, averaging is over 100 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Avg. Human, average human performance; SOTA, state-of-the-art performance; M, $\times 10^6$; K, $\times 10^3$.

**a** Exploration phase without domain knowledge.

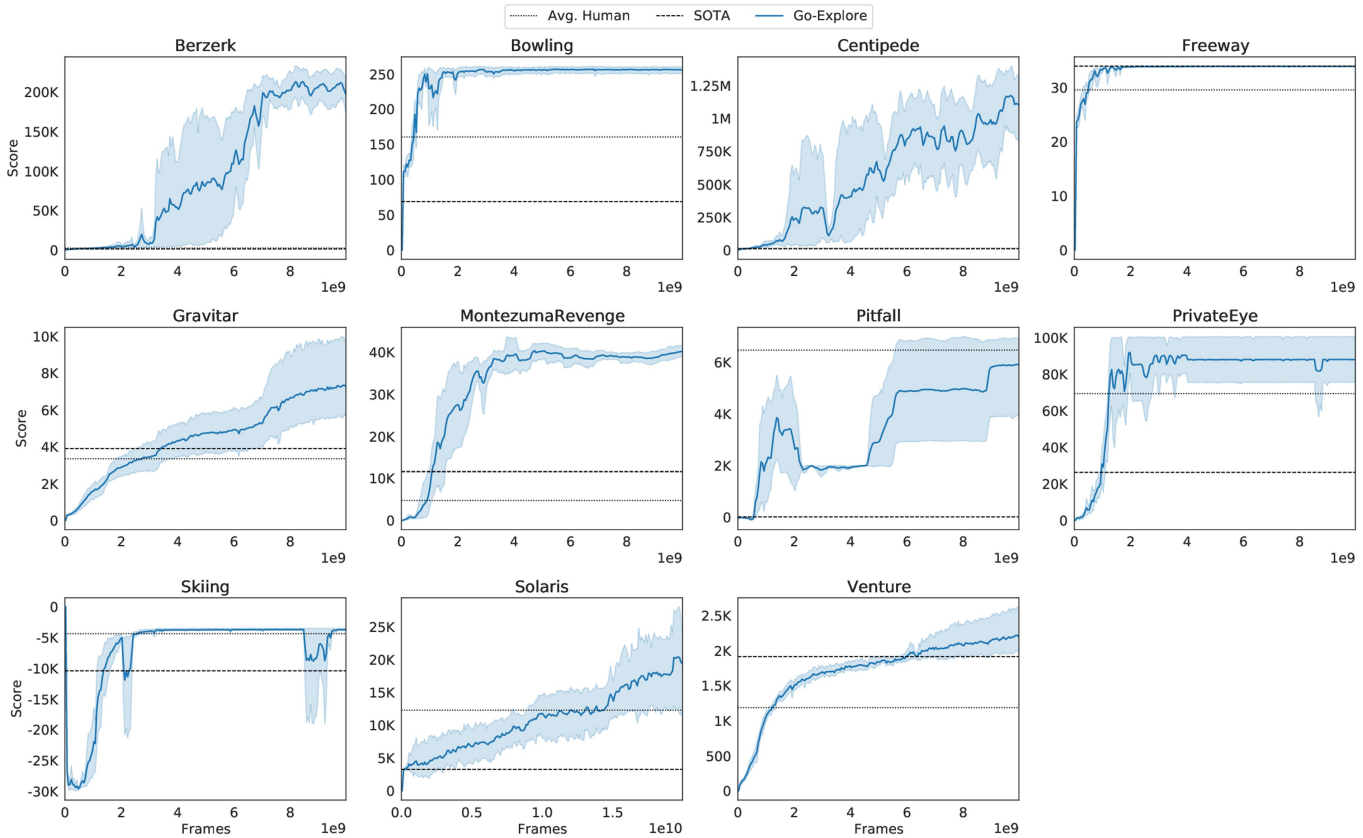



**b** Exploration phase with domain knowledge.

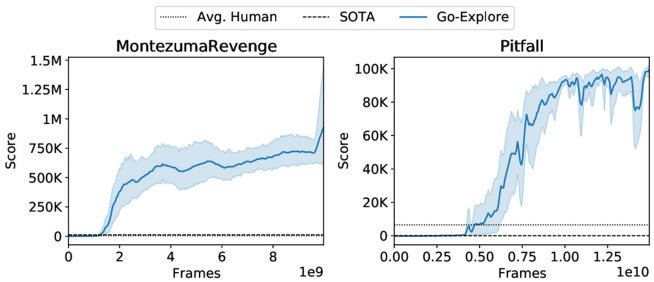



**Extended Data Fig. 3 | Number of cells in archive during the exploration phase on Atari. a**, Exploration phase without domain knowledge. **b**, Exploration phase with domain knowledge. In **a**, archive size can decrease when the representation is recomputed. Previous archives are converted to the new format when the representation is recomputed, possibly leading to an archive with a size larger than 50,000. In this case, one iteration of the exploration phase runs and the representation is recomputed again. In **a**, averaging is over 50 runs for the 11 focus games and five runs for other games. In **b**, averaging is over 100 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

**a** Exploration phase without domain knowledge.

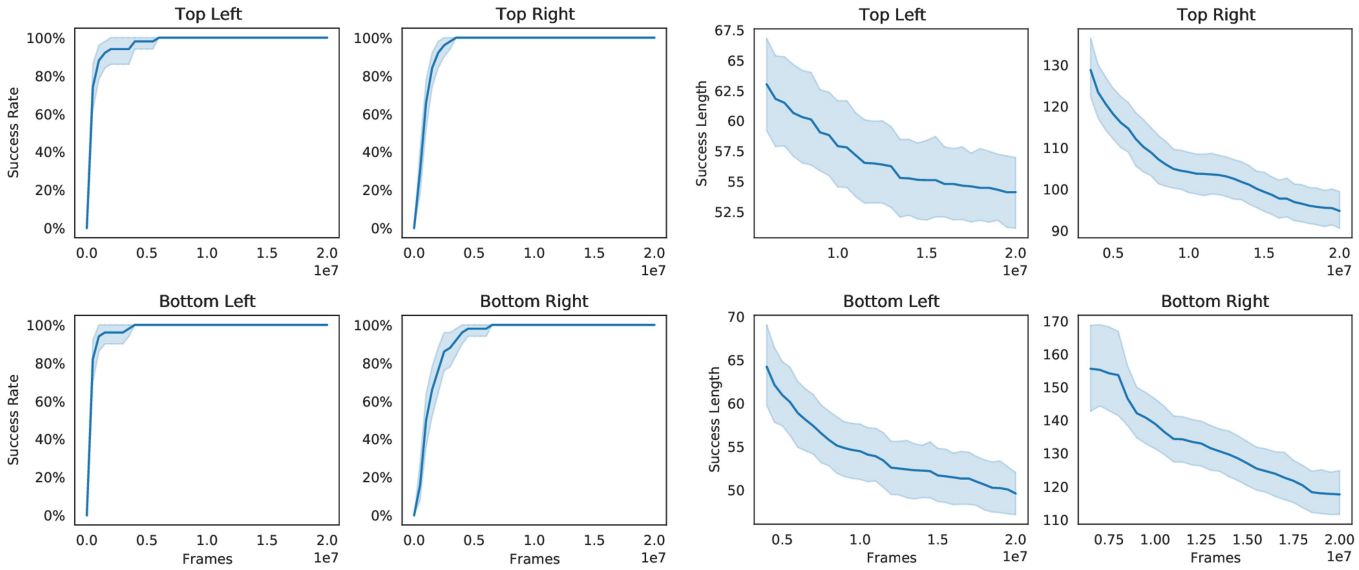



**b** Exploration phase with domain knowledge.

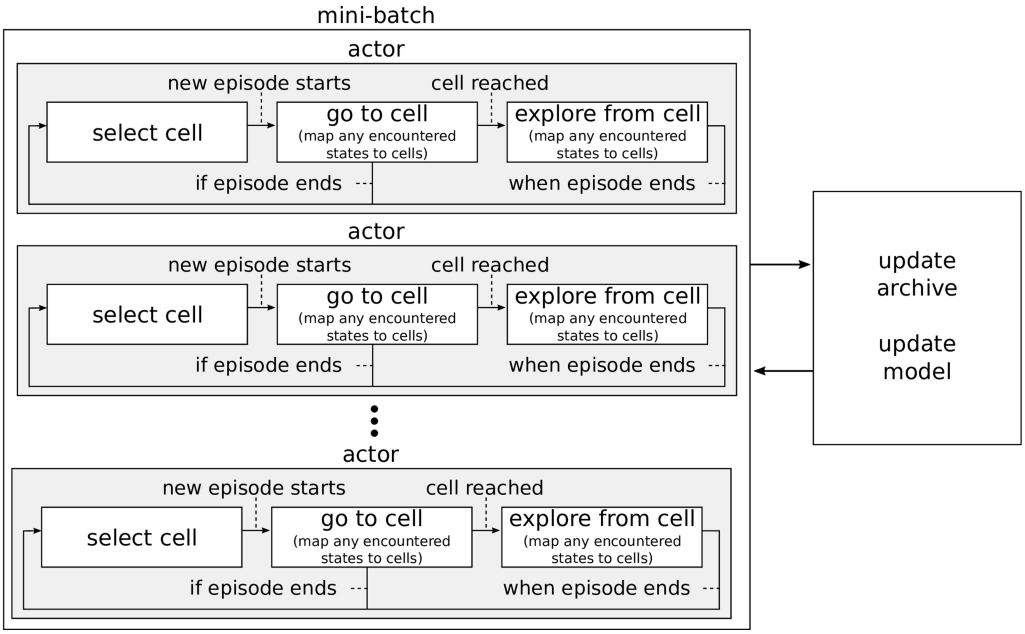



**Extended Data Fig. 4 | Progress of robustification phase on Atari.** **a**, Exploration phase without domain knowledge. **b**, Exploration phase with domain knowledge. Shown are the scores achieved by robustifying agents across training time for the exploration phase without domain-knowledge representations (**a**) and with representations informed by domain knowledge (**b**). In particular, the rolling mean is shown for 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 five independent runs, whereas 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.

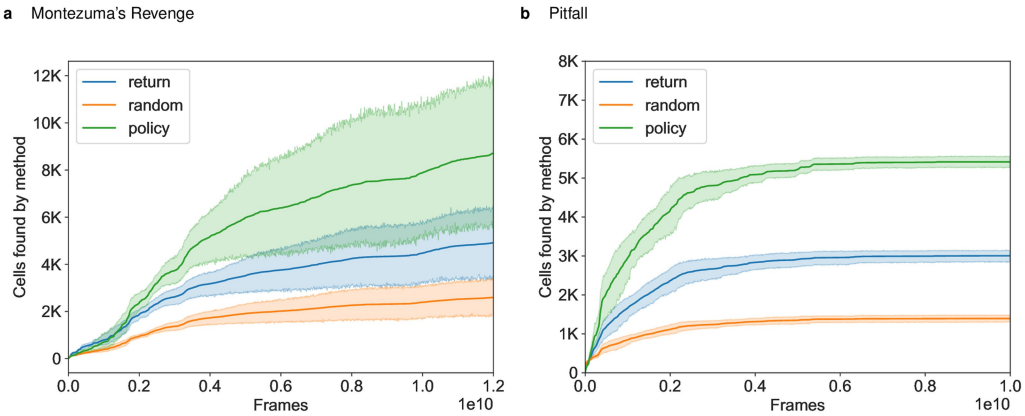
**a** Runs with successful trajectories. **b** Length of the shortest successful trajectory




**Extended Data Fig. 5 | Progress of the exploration phase in the robotics environment. a**, Runs with successful trajectories. **b**, Length of the shortest successful trajectory. In **a**, the exploration phase quickly achieves 100% success rate for all shelves in the robotics environment. However, **b** shows that although 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. Lines show the mean over 50 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.




**Extended Data Fig. 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 a 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 synchronized batches. This structure allows policy-based Go-Explore to be easily combined with popular reinforcement learning algorithms like A3C[20], PPO[21] or DQN[15], which already divide data-gathering over many actors.




**Extended Data Fig. 7 | Method by which cells are found. a**, **b**, In both Montezuma's Revenge (**a**) and Pitfall (**b**), 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. Lines show the mean over 10 runs. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples.

**Extended Data Table 1 | Hyperparameters**

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

| Parameter | Robustification | | Policy-Based |
|---|---|---|---|
| | *Atari* | *Robotics* | *Atari* |
| Discount factor ($\gamma$) | 0.999 | 0.99 | 0.99 |
| N-step return factor ($\lambda$) | 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 ($\epsilon$) | 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‡ |

**a**, Parameters above the dividing line are applicable to PPO with self-imitation learning, whereas 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', $c$: $\lfloor e^{cX} \rfloor$ ($X \sim U(0, 1)$), where e is Euler's number and $U$ refers to the uniform distribution. 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 a simulator state, only 'Length limit', 'End of episode', and 'Action repeat' apply.

*OpenAI Gym default.

†Except for Montezuma's Revenge with a return policy (Methods section 'Policy-based Go-Explore details').

‡4 for Gravitar and Venture.

SIL, self-imitation learning; Num., number; coef., coefficient, max., maximum.

**Extended Data Table 2 | Robotics state representation**

**a** Position and velocity objects

| Object |
|---|
| door1 |
| door |
| elbow_flex_link |
| forearm_roll_link |
| gripper_link |
| head_camera_link |
| head_pan_link |
| head_tilt_link |
| l_gripper_finger_link |
| latch1 |
| latch |
| obj0 |
| r_gripper_finger_link |
| shoulder_lift_link |
| shoulder_pan_link |
| upperarm_roll_link |
| wrist_flex_link |
| wrist_roll_link |

**b** Collision and bounding box objects

| Object |
|---|
| DoorLR |
| DoorUR |
| Shelf |
| Table |
| door1 |
| door |
| frameL1 |
| frameL |
| frameR1 |
| frameR |
| gripper_link |
| l_gripper_finger_link |
| latch1 |
| latch |
| obj0 |
| r_gripper_finger_link |
| world |

**a**, Position and velocities of the objects in **a** are included in the state representation for robotics. **b**, 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 section 'Code availability'.

**Extended Data Table 3 | Full scores on Atari**

| 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 |

Go-Explore outperforms the state of the art on all focus games (Freeway's score is at its maximum). The exploration phase similarly finds trajectories that frequently exceed state-of-the-art scores. Finally, Go-Explore outperforms Agent57 on seven 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 section 'State-of-the-art performance on Atari'). Expl. phase, Exploration phase; Robust. phase, Robustification phase; Avg. human, Average human.