nature portfolio

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

Supplementary information

First return, then explore

Supplementary Information for: First return then explore

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

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

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

* *These authors contributed equally to this work*

1 Algorithms

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

Supplementary Algorithm 1 Exploration Phase with Simulator Restoration.

```
1: Input: The archive sampling batch size $K$, the rollout length $L$, the starting state $\mathrm{startingState}$, the initial representation parameters $\mathrm{rParams}$
2: (Optional) Input for dynamic representations: probability of adding frame to recent frames sample $p_s$, max archive size $M$, initial number of frames before recomputing representations $F$, interval between recomputing representations $I$

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

Supplementary Algorithm 2 Robustification Phase Rollout Worker.

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

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

Supplementary Algorithm 3 Policy-based Go-Explore.

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

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

2 Prior work on Montezuma's Revenge

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

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

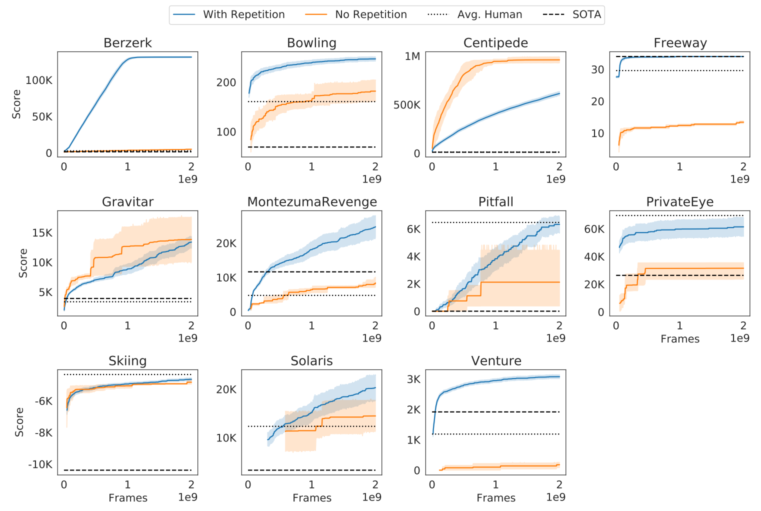
Supplementary Table 1: **Scores on Montezuma's Revenge for the algorithms shown in Figure 2a of the main paper.** Scores marked with an asterisk were estimated from a graphical representation.

3 Ablations

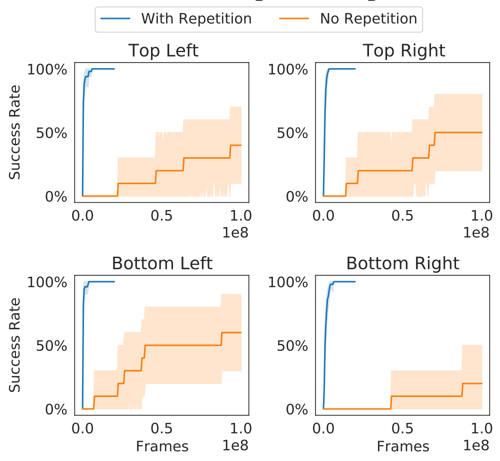
For the ablations and parameter analyses presented in this section, data comes from the main experiment where available. Other than the parameter or setting being varied, the only difference between the main experiment and its ablations is that the main experiment sometimes includes a larger number of independent runs. The number of runs for each treatment is specified in the caption of the corresponding figure.

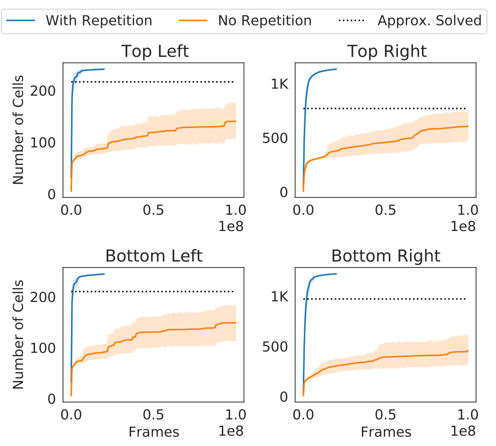
3.1 Exploration phase without action repetition. As explained in Methods "Exploration phase", actions are repeated with high probability during the exploration step. Action repetition allows agents to explore in a well-defined direction instead of dithering in place, analogously to existing temporally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

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



(a) Exploration phase scores in the 11 Atari focus games.

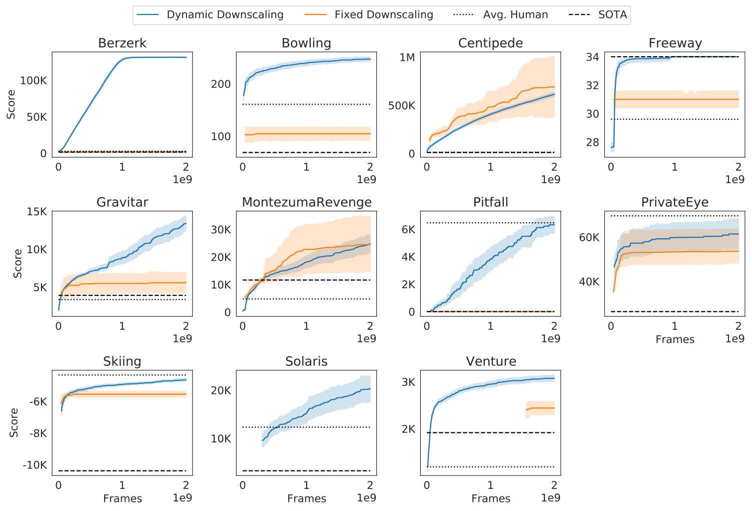




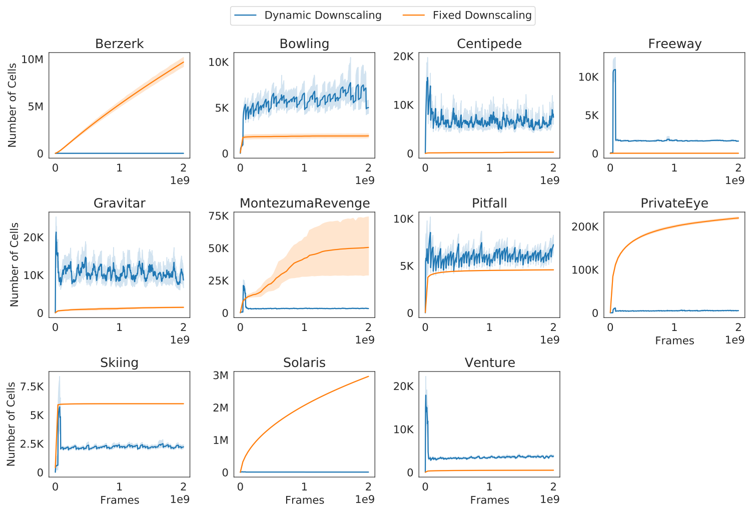
(b) Exploration phase success rates in the robotics environment.

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

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



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



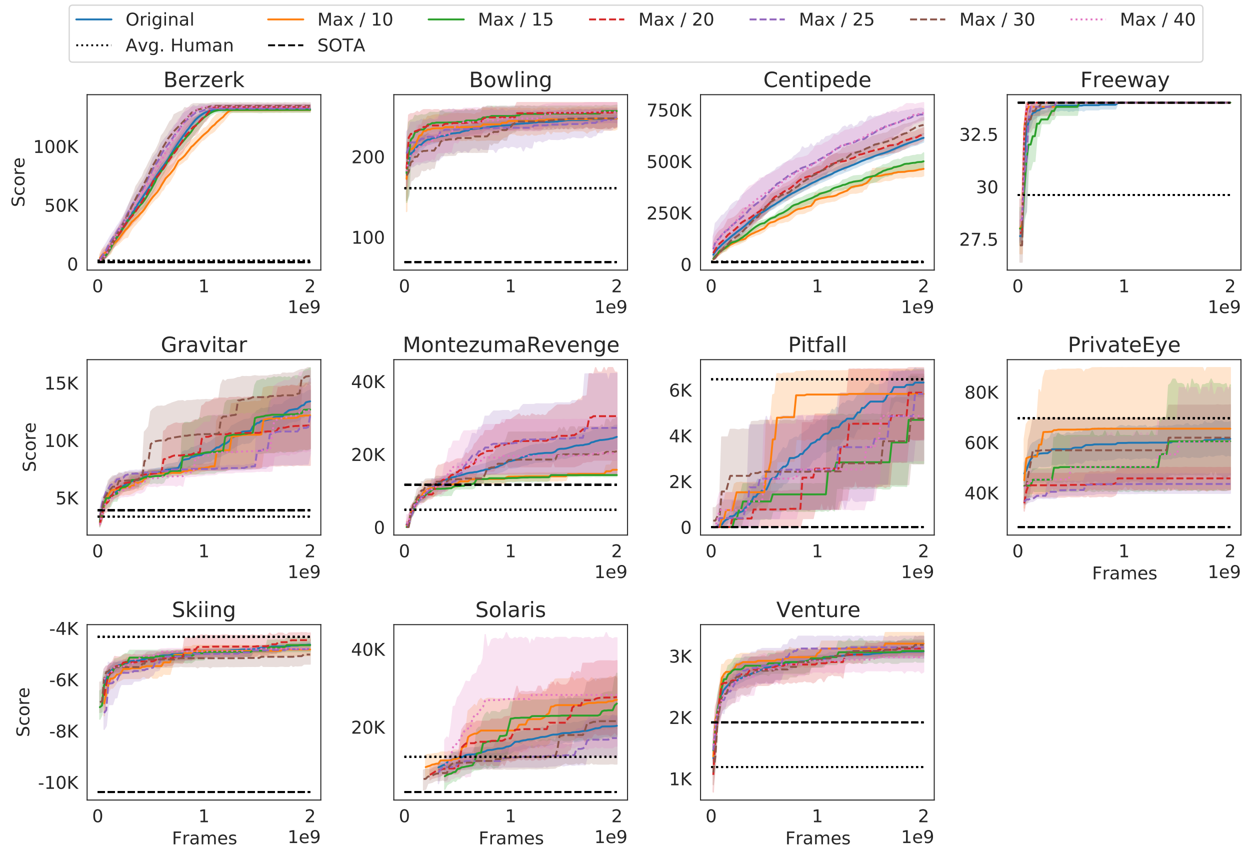
(b) Cells discovered by the exploration phase in the 11 Atari focus games.

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

Per the evaluation method (Methods "Evaluation"), scores in Supplementary Fig. 2a are recorded at the end of episodes. In Solaris with fixed downscaling, so many cells are produced close to the starting point that exploration focuses on the start of the game and never produces trajectories long enough to reach the end of episode (i.e. the agent never makes it to a place in the game where it can die), resulting in no line being shown. The maximum score regardless of episode end averages to 744 (CI: 640 – 888), far less than with dynamic downscaling. In Venture with fixed downscaling, so few cells are created and rewards are so sparse that trajectories have difficulty being extended (as longer trajectories are only produced if they lead to a new cell or a higher total score), so that the end of an episode (i.e. the agent dying) is only reached after around 1.6 billion frames have been processed.

3.3 Downscaling distribution minimum means. During the randomised search for downscaling parameters (Methods "Downscaling on Atari"), parameters are sampled from a geometric distribution whose mean is the current best value of the parameter. The use of the geometric distribution captures the intuition that lower parameter values need to be over-sampled compared to larger values because they produce representations that are more different from each other, i.e. a downscaling with a width of 100 is unlikely to produce an aggregation that is very different from one with a width of 101, while representations with width 1 and 2 are likely to be very different from each other. The geometric distribution was thus chosen as a way to sample low values with higher probabilities. If the current best known value for a parameter is very low, however, it may become virtually impossible for larger values to ever be sampled. As a result, we define a minimum mean for each parameter. These are set to approximately 1/20th of the maximum value the parameter can take (8 for width, 10.5 for height, and 12 for depth).

It is reasonable to wonder whether the minimum means have a strong effect on the performance of the exploration phase of Go-Explore, or whether they merely serve to avoid the collapse to low values described above and can be set to a wide range of values. In Supplementary Fig. 3, we show that the algorithm is robust to values ranging from 1/40th to 1/10th of the maximum, with no value producing qualitatively different results from the original relative to average human and state-of-the-art performance.



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

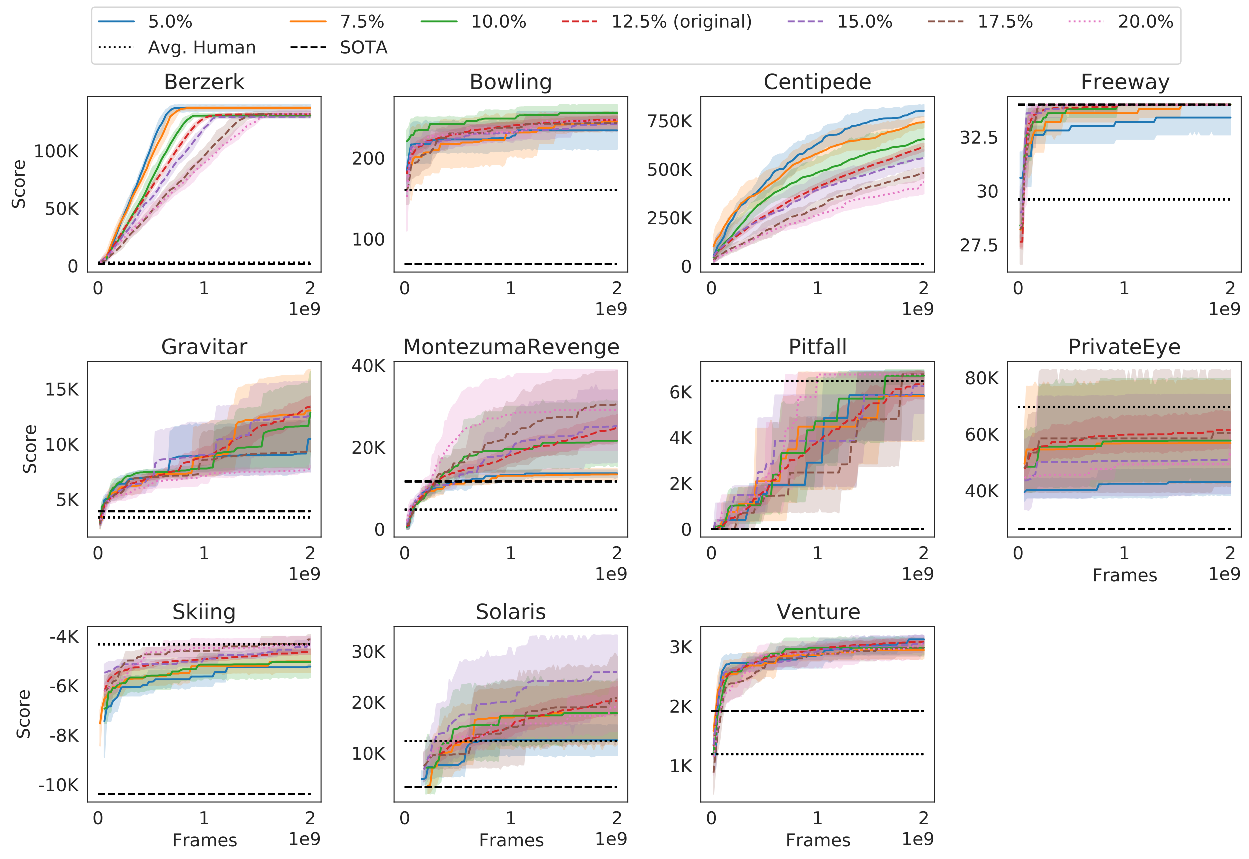
3.4 Downscaling target proportion. The search for downscaling parameters aims to produce a target number of cells given a buffer of sample frames (Methods "Downscaling on Atari"). This target number of cells is a fraction of the number of frames in the buffer, set to 12.5% in our implementation (i.e.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

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

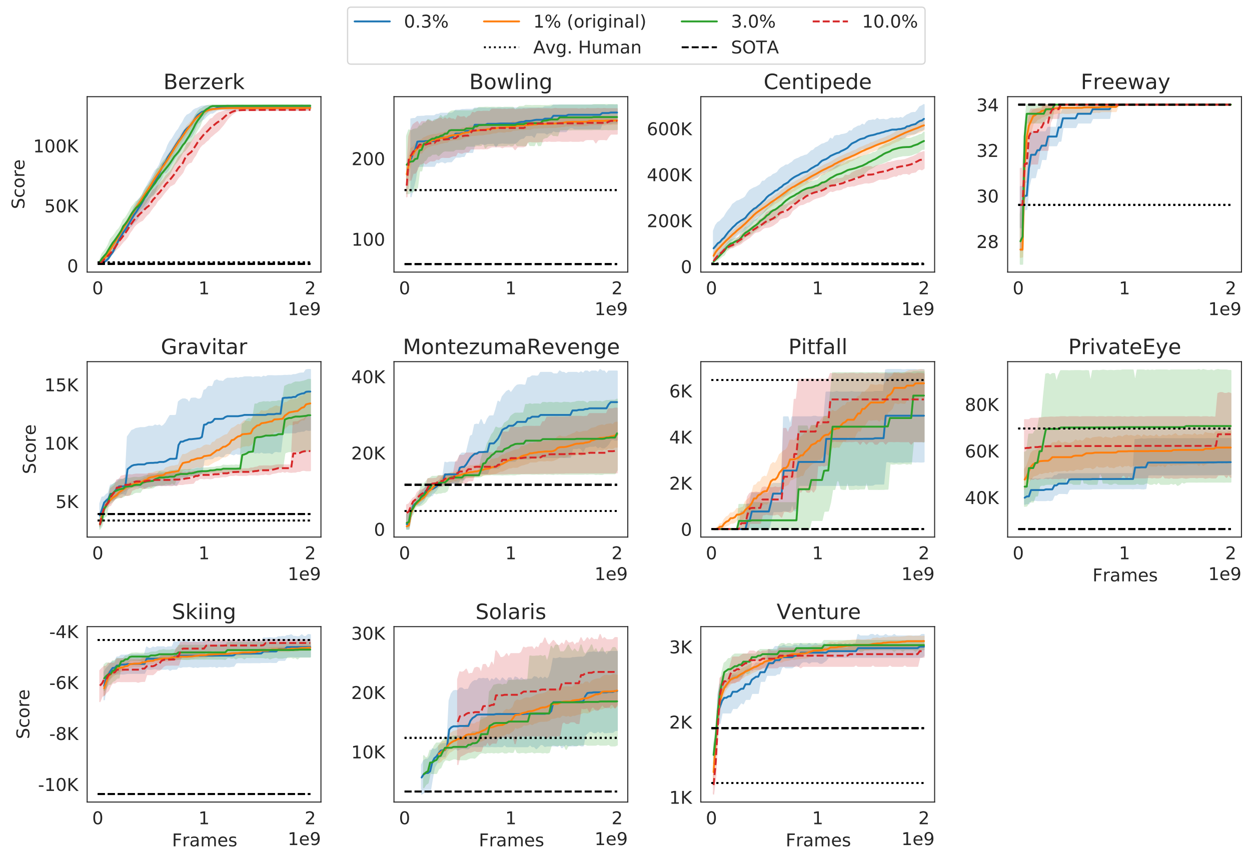
3.5 Downscaling sampling rate. Downscaling hyperparameters are found through a search process over a sample of frames discovered during exploration. Because the buffer has a limited size and to ensure that it contains diverse frames, frames are added to the buffer probabilistically according to a sampling rate of 1% (Methods "Downscaling on Atari"). The sampling rate effectively controls the tradeoff between having a more diverse set of frames in the buffer (if it is low) or a set of more recent frames (if it is high). In this experiment we investigate whether the performance of the exploration phase is strongly sensitive to this hyperparameter.

In Supplementary Fig. 5, we analyse sampling rates ranging from 0.3% (because the buffer is cleared every 10 million actions – the frequency at which the representation is recomputed – and contains 10,000 frames, frequencies less than or equal to 0.1% are likely to not always fill the buffer completely) to 10%. We find that the sampling rate generally does not have a large effect on results, though the higher sampling rate of 10% does tend to produce lower performance relative to the original value of 1% (the difference is statistically significant in Berzerk, Centipede, Gravitar and Venture; $p < 0.05$ according to a two-sample empirical bootstrap test with 10,000 samples), supporting the usefulness of increasing buffer diversity through sampling.

3.6 Domain-agnostic selection probabilities in Montezuma's Revenge. As explained in Methods "Exploration phase", a custom cell selection probability that makes use of domain

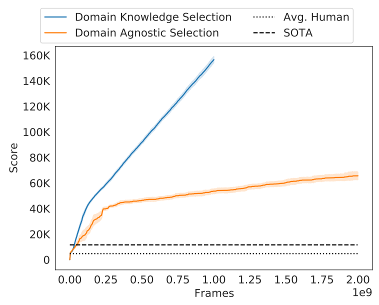


Supplementary Figure 4: **Exploration phase performance with different target cell proportions.** High-quality results are achieved across all values for all games (except Freeway at 5%). While results within any target cell proportion are high-performing, sparse reward games such as Freeway, Montezuma, PrivateEye, and Solaris tend to benefit more from a larger target cell proportion, while dense reward games like Berzerk and Centipede tend to benefit from a smaller target proportion. These results suggest that producing more cells favours exploration, while producing fewer cells favours exploitation, perhaps because more time is spent expanding high-scoring trajectories rather than newly-found cells. Shaded areas show 95% bootstrap CIs of the mean with 1,000 samples. Averaging is over 50 runs per game for the original 12.5% value and 5 runs per game for other values.

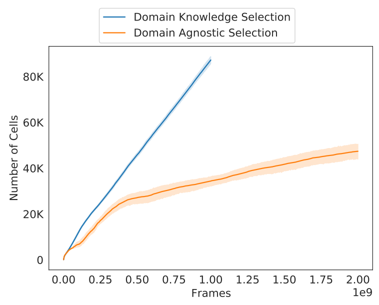


Supplementary Figure 5: **Exploration phase performance with different frame sampling rates.** While the sampling rate does not have a large effect overall (and does not change the qualitative result of Go-Explore producing advances over humans and state-of-the-art algorithms), as a general trend, larger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

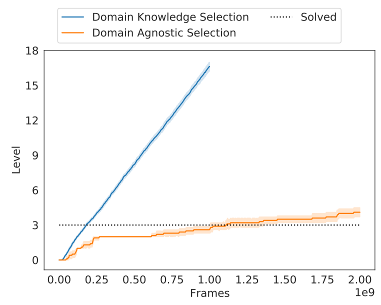
knowledge is used for Montezuma's Revenge with domain knowledge. Supplementary Fig. 6 shows that this selection probability greatly speeds up the exploration phase in that context, but that even with the generic selection probability, Go-Explore still solves the entire game when using a domain knowledge cell representation.



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



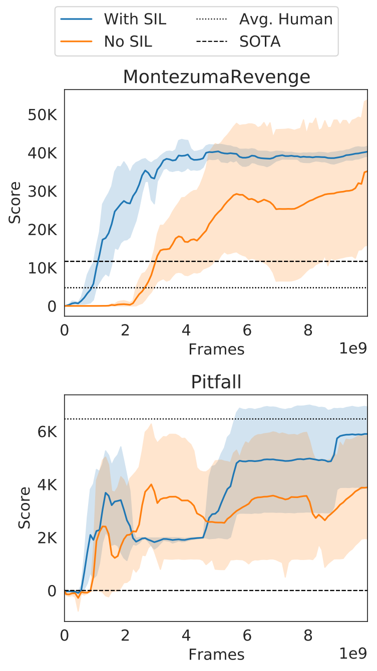
(b) Cells discovered by the exploration phase in Montezuma's Revenge with domain knowledge.



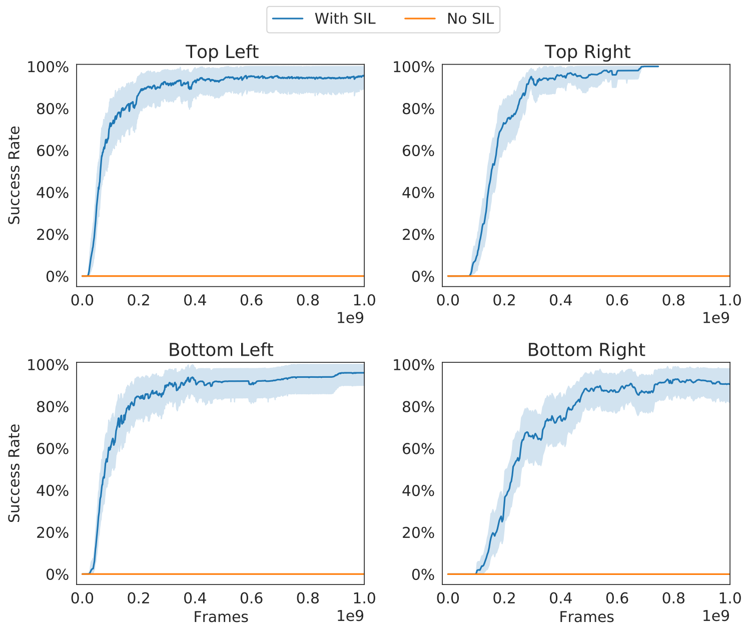
(c) Levels solved by the exploration phase in Montezuma's Revenge with domain knowledge.

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

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



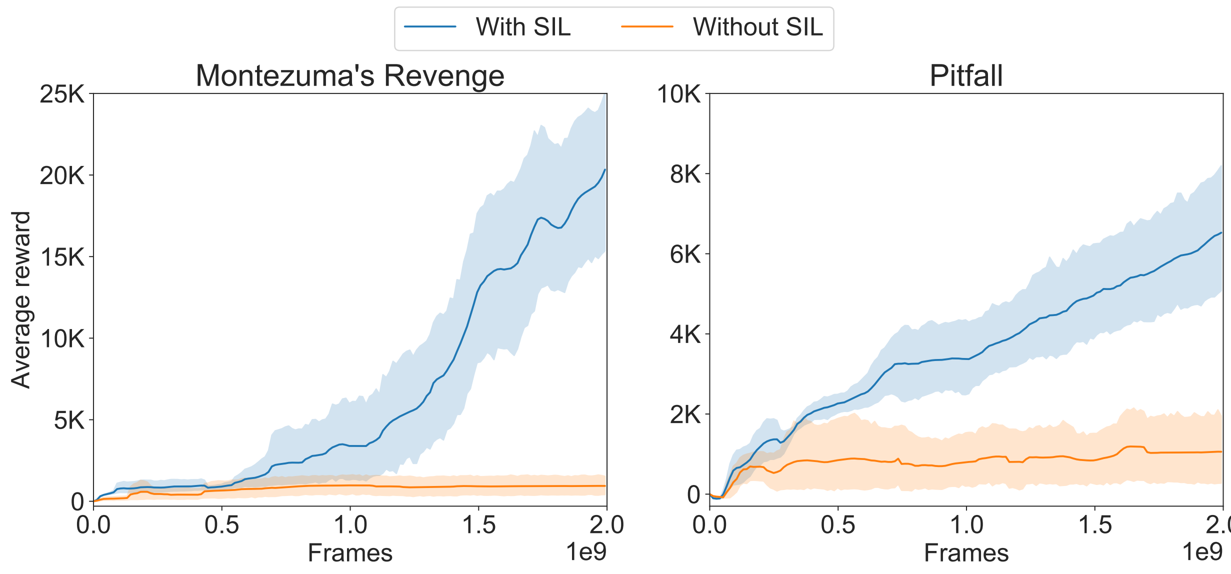
(a) Robustification score for Montezuma's Revenge and Pitfall.



(b) Robustification success rate for robotics.

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

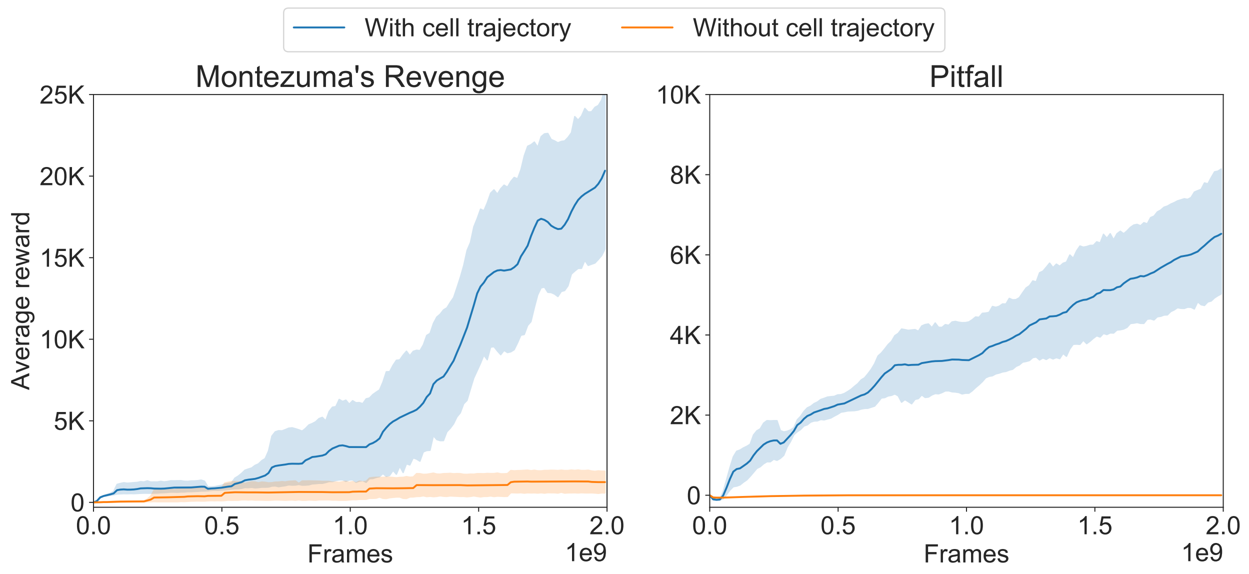
3.8 Policy-based Go-Explore without imitation learning loss. Similar to the robustification phase, policy-based Go-Explore implements a Self-Imitation Learning (SIL) loss. Removing the SIL loss substantially reduces the performance of policy-based Go-Explore on both Montezuma's Revenge and Pitfall, at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



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

3.9 Policy-based Go-Explore without a cell trajectory. A prominent feature of policy-based Go-Explore is that the goal-conditioned policy is provided with a cell-by-cell trajectory towards the target cell. In principle, it should be possible to train a goal-conditioned policy to move directly towards the target cell, without providing the intermediate trajectory. However, the ablation that removes this cell trajectory demonstrates that policy-based Go-Explore performs substantially worse without the trajectory on both Montezuma's Revenge and Pitfall, even to the extent that policy-based Go-Explore is unable to find any reward in Pitfall (Supplementary Fig. 9). The reason is that, without the intermediate trajectory, returning to a far-away cell is itself a sparse reward problem. Initially, the agent will visit cells near the starting position purely by random exploration. Over time, the agent will learn how to visit those cells intentionally when they are provided to the goal-conditioned policy as a target. From there, the agent will discover new cells that are farther away from the starting point. However, when being trained to return to these farther away cells, the agent has to find those cells from the start, and is not provided with any gradient towards those cells. Imagine that the agent has mostly explored the first room in Montezuma's Revenge and now discovers its first cell in the next room. The representation of this new cell has [$room = 2, x = 0$)], but it was discovered from a cell with [$room = 1, x = 20$]. At this point, there is no way for the policy to know that, in order to reach the target with [$room = 2, x = 0$], it has to first execute the policy towards [$room = 1, x = 20$] (in fact, the policy is much more likely to execute the actions towards [$room = 1, x = 0$] instead, because room was never a relevant feature before). As such, the agent has to rely on the occasional random actions from the stochastic policy to rediscover the cell in the next room and obtain the experience necessary to learn how to reach it. Both the SIL loss (which replays the experience of discovering the cell) and the cell trajectory help in alleviating the problem, and their impact on performance is similar (compare Supplementary Fig. 8 and Supplementary Fig. 9), but both are required to really improve performance. Other solutions could be to develop techniques that explicitly teach the agent to generalise to new cells or to provide rewards for following the trajectory, even if the policy does not get to observe the intermediate cells.

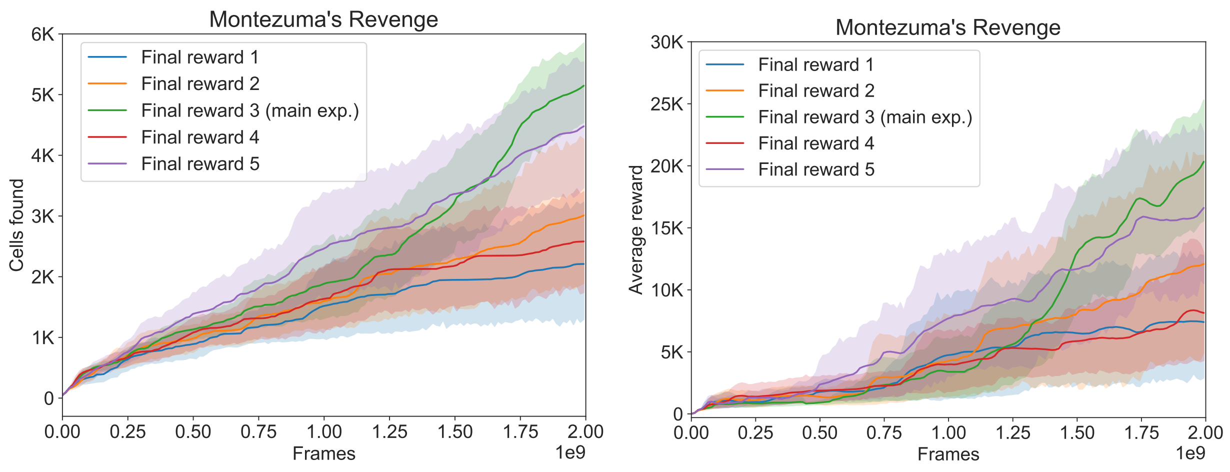
3.10 Policy-based Go-Explore final-cell reward. To implement the general practice of having a higher reward for reaching a desired final state than for completing any intermediate objectives[65,66], we provide a higher reward for reaching the final cell in a trajectory than for reaching any of the intermediate cells. Doing so encourages the goal-conditioned policy to shorten its trajectory towards



Supplementary Figure 9: **Policy-based Go-Explore with and without a trajectory of cells.** The results suggest that the trajectory of cells is essential for obtaining high average cumulative rewards in both Montezuma's Revenge and Pitfall. Shaded areas show 95% bootstrapped confidence intervals of the mean with 1,000 samples. Each line is averaged over 10 runs with independent seeds.

the final cell of a trajectory, thus resulting in more efficient behaviour. To demonstrate the effect of this increased *final reward*, we ran experiments for different values of the final reward, ranging from 1 (equal to the intermediate cell reward) through 5 (5 times higher than the intermediate cell reward). The experiments were performed on Montezuma's Revenge for 2 billion frames, with 10 random seeds for each parameter value.

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

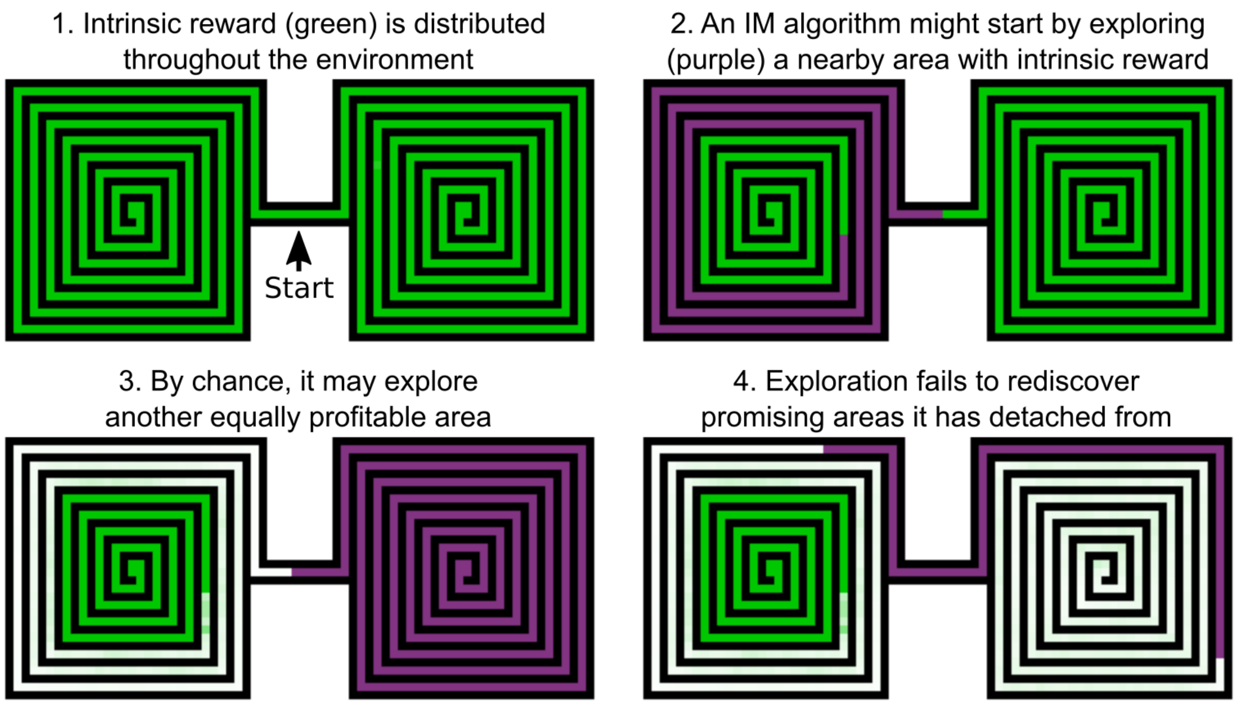


Supplementary Figure 10: **All tested values for final rewards lead to the continuous discovery of new cells over the first 2 billion frames (left).** Similarly, the average reward continuously increases for all treatments (right). Shaded areas show 95% bootstrapped confidence intervals of the mean with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

4 Detachment and derailment

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

4.1 Detachment. We define detachment as losing track of interesting areas to explore from. Here, "interesting areas to explore from" refers to states for which we have evidence (e.g. a low number of visits) that they could lead to the discovery of new areas of the environment. "Losing track" means that the algorithm stops trying to visit those areas prematurely, despite the fact that they are not thoroughly explored yet. For reinforcement learning algorithms that only optimise the expected return, detachment is almost guaranteed, as these algorithms do not attempt to promote exploration explicitly. Unless external rewards are aligned with interesting areas to explore from, such algorithms will stop visiting under-explored areas in favour of areas with a high return. However, algorithms that reward the agent for exploring new states can still suffer from detachment. In intrinsic-motivation (IM) algorithms, for example, detachment can happen because the intrinsic


1. Intrinsic reward (green) is distributed throughout the environment
Start
2. An IM algorithm might start by exploring (purple) a nearby area with intrinsic reward
3. By chance, it may explore another equally profitable area
4. Exploration fails to rediscover promising areas it has detached from

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

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

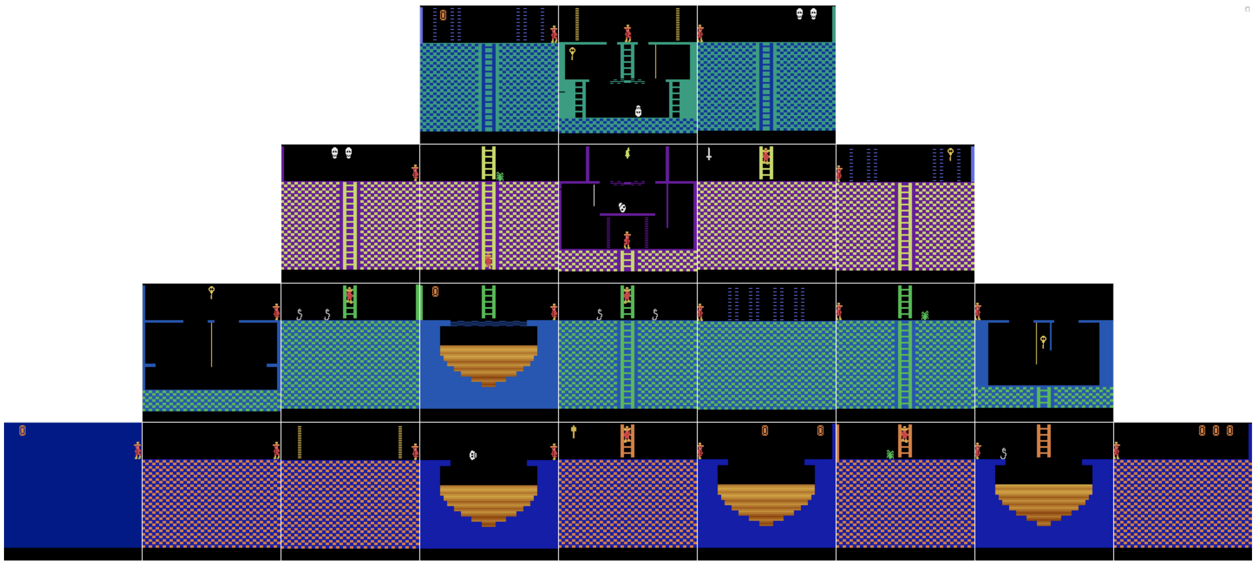
4.2 Derailment. We define derailment as when the exploratory mechanisms of the algorithm prevent it from returning to previously visited states. Returning to previously visited states is important because many RL environments, and especially hard-exploration environments, contain a large number of states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

The solution we propose to avoid derailment is to have an algorithm exhibit separate exploration probabilities depending on whether it is in a well-known area of the environment, meaning the probability of exploratory actions should be low, or in an unknown area of the environment, meaning the probabilities of exploratory actions should be high. While doing so is not a feature of state-agnostic exploratory mechanisms like ϵ-greedy exploration, one could hope that it is a property of algorithms that explore by sampling from a stochastic policy, because stochastic policies could learn to be low entropy in familiar states (i.e. they are certain about the correct action) while remaining high entropy in new states (i.e. where they should be uncertain about the correct action). However, deep learning struggles to remain well-calibrated when provided out-of-distribution input data. In image classification, for example, when networks classify images far out of distribution, it would be helpful if these networks returned a uniform distribution of probability across classes to properly indicate uncertainty, but networks instead are often surprisingly overconfident[90,91]. In the context of RL, this result means that we can expect trained policies to be highly confident in their actions (i.e. low entropy), even in areas that they have never observed before, thus resulting in a lack of exploration. To remedy this issue, stochastic-policy RL algorithms generally add an entropy bonus to the loss function, encouraging the policy to assign more equal probability to all actions. However, because this entropy bonus applies equally throughout the trajectory of a policy, it is difficult to tune the entropy bonus in a way that guarantees effective exploration without sacrificing the network's ability to return; if the entropy bonus is too high, the policy will frequently take exploratory actions that prevent it from returning, thus causing derailment, but if the entropy bonus is too small, the policy will not explore sufficiently when a new area is reached.

5 Exploration in Atari

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

One particular metric indicative of exploration in both Montezuma's Revenge and Pitfall is



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

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

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

Other games worth mentioning are Private Eye and Skiing. Private Eye is a sparse reward game and is also a common benchmark in work focused on exploration in RL[1,13,31,51,53–55]. The goal in Private Eye is to collect evidence and apprehend a criminal by navigating a maze-like "city". Similar to Montezuma's Revenge and Pitfall, the environment in Private Eye contains very few rewards and is full of hazards that need to be avoided. In addition, the objectives need to be completed in a specific order (all evidence needs to be collected and delivered to specific locations before the criminal can be apprehended), making it particularly difficult to complete all objectives, as is evidenced by the fact that state-of-the-art performance is 26,364[53], far below the roughly 100,000 points that can be obtained by completing all main objectives. Despite these obstacles, the robustified policies produced by Go-Explore with a downscaled representation are able to reliably achieve more than 100,000 points in 4 out 5 runs, thus having discovered and learned to perform all objectives in the game.

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

6 Generality of downscaling

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

7 Derailment in robotics

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

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

8 Go-Explore and Quality-Diversity

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

9 Go-Explore, Planning, and Model-based RL

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

Porting the principles behind planning algorithms to high-dimensional state spaces by aggregating similar states into cells is a non-trivial technical challenge. One problem is that, in the aggregated space, it is unknown whether edges exist between the different nodes, meaning that an algorithm has to empirically discover the existence of an edge between two nodes, for example by executing a sequence of actions to try to reach one node from another. Thus, nodes can never fully be marked as "closed". Go-Explore addresses this issue by never marking any cells as closed, but instead reducing the relative probability of selecting a cell when it is explored from. This way, if a cell has been explored many times, meaning it probably should be considered "closed", it is indeed selected only infrequently. However, if at some point all cells in the archive have been explored many times, the selection probabilities will equalise, thus ensuring that old cells will be explored from again, thus preventing the algorithm from getting stuck. Another challenge is that two different trajectories towards the same aggregated cell can actually lead to two very different states (e.g. in one state the agent may be in the process of jumping over a gap while in the other state the agent may be falling into that gap). This difference in states makes it difficult to substitute one path to a particular cell with a new path to that same cell, even if the new path is shorter or higher scoring, because, while the two paths may lead to the same cell, they may actually lead to very different states within that cell. Go-Explore side-steps this challenge by never performing path substitution, but instead by re-exploring whenever a better path to an existing cell is found. Overall, Go-Explore motivates porting the techniques from classic planning algorithms to challenging problems with high-dimensional search spaces and suggests some methods for how this may be achieved.

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

Finally, Go-Explore is reminiscent in some ways of model-based RL algorithms. For example, when Go-Explore returns to previously visited states by restoring simulator state, the simulator effectively operates as the model for Go-Explore. Many model-based RL algorithms perform relatively shallow runs of stochastic planning algorithms, like MCTS[99] or UCT[62,100], to decide which action to take every time a decision has to be made. Go-Explore with state restoration, on the other hand, is more similar to algorithms like Dyna[101] and AHCON-M[102], which are hybrids between model-based and model-free RL algorithms because they use a model during training, but eventually take actions with a model-free policy. However, Go-Explore with state restoration differentiates itself from those algorithms by deeply exploring a pre-existing model to find high-performing solutions which it robustifies into a model-free policy only after this exploration process has finished. Another difference is that model-based RL is generally focused on learning a model, but the two variants of Go-Explore presented in this paper do not try to learn a model; the version of Go-Explore that restores simulator state assumes that an appropriate simulator is already available (and it frequently is in many practical domains), while policy-based Go-Explore trains a goal-conditioned but model-free policy to navigate the environment. However, there is no obvious obstacle that precludes a variant of Go-Explore that takes exploratory steps in the real environment in order to learn a model, and then performs deep exploration within that model to determine the most promising location to explore next. Doing so is a promising area for future work.

10 Go-Explore and Stochasticity

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

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

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

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

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

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

11 Policy-based Go-Explore and Stochasticity

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

Policy-based Go-Explore can alleviate this situations in two ways. First, because its progress along the highway is not saved and because each trial is in a stochastic environment, policy-based Go-Explore must attempt to return manually to each cell across the highway in different conditions. If there does not exist a reliable policy that can do so, it is unlikely that policy-based Go-Explore will ever cross the highway this way. As a result, policy-based Go-Explore is much more likely to learn a policy that reliably navigates the overpass instead.

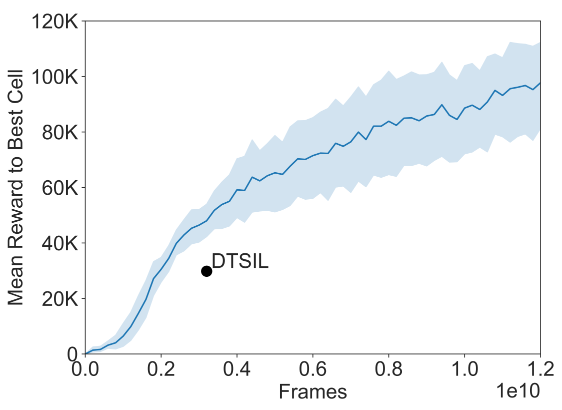
Second, because policy-based Go-Explore needs to return to cells in the presence of stochasticity, it can keep track of the success rate towards each cell in the archive. As such, even if policy-based Go-Explore is sometimes able to cross the highway, it is possible to not overwrite cells on the other side of the highway until the policy has learned to return to those cells reliably. Doing so prevents the shorter, but unreliable trajectories from overwriting the longer but more reliable trajectories that take the safe overpass. A similar mechanic could be implemented to deal with stochasticity in rewards, as policy-based Go-Explore makes it possible to track the average reward when attempting to reach a particular state. Because it was not necessary to implement such mechanics for the games of Montezuma's Revenge and Pitfall, studying the effectiveness and exact implementation details of such a mechanic is a topic for future research.

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

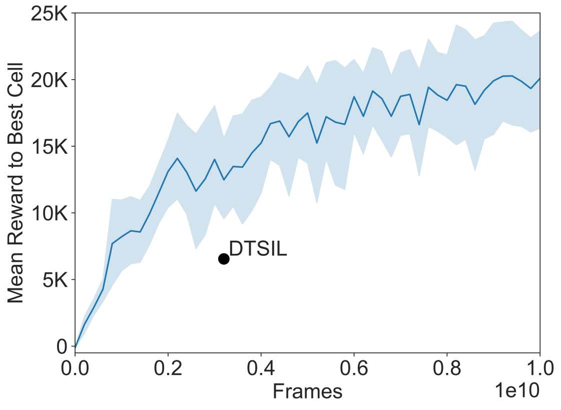
12 Comparing Policy-based Go-Explore and DTSIL

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

DTSIL is similar to policy-based Go-Explore in that it follows the methodology described in the Go-Explore pre-print in the following ways: (1) Like the original Go-Explore, DTSIL explic-



(a) Montezuma's Revenge



(b) Pitfall

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

itly keeps track of an archive of many different states and trajectories to those states, (2) DTSIL first moves the agent to one of these states before performing random exploration, (3) DTSIL determines whether to add a state to the archive with the help of a domain-knowledge based state embedding, where similar embeddings are grouped into a single cluster (i.e. a cell representation), and (4) DTSIL selects trajectories to follow (i.e. states to return to) by selecting them probabilistically based on the number of times particular clusters have been visited, though DTSIL did not consider any domain knowledge specific information for the purpose of this selection procedure.

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

One major difference between policy-based Go-Explore and DTSIL is that, when following a trajectory, DTSIL aims to provide the entire trajectory as input to the policy, rather than just the next cell in the trajectory. As a result, the DTSIL network architecture requires some method to deal with variable length trajectories (resolved with an attention layer), while the policy-based Go-Explore architecture only requires the next (sub) goal cell as an input. Note that, in the DTSIL paper, most trajectories that are followed were found to be too long to provide to the network in their entirety, meaning the trajectory is instead provided to the network in small chunks, resulting in a dynamic that is similar to providing only the next goal.

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

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

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

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

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

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

The training performance of DTSIL on Montezuma's Revenge and Pitfall was reported by Guo *et al.* (2019)[34] for experiments that ran for 3.2 billion frames. However, the performance-over-time graph presented in the DTSIL paper is not directly comparable with the performance-over-time graph shown in this paper because they represent results for different selection strategies. For policy-based Go-Explore, we always report the average score achieved when returning to the high-

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

est scoring cell in the archive (obtained after training is completed by loading stored checkpoints and testing the policy 100 times). In contrast, the DTSIL graph shows a rolling average score during training, which means that its average includes returning to low-scoring cells as the algorithm attempts to explore the environment. That said, the final performance of DTSIL after 3.2 billion frames is measured over only the highest scoring trajectories, and can thus be reasonably compared with the testing performance of policy-based Go-Explore after that many frames. For Montezuma's Revenge, we compare policy-based Go-Explore against the results that were reported in the supplementary information of the DTSIL paper for a version of DTSIL that implemented the same cell representation as the one used in the policy-based Go-Explore experiments (note, the Montezuma's Revenge results reported in the main DTSIL paper were lower). For Pitfall, we compare against the only reported results, which were obtained with a slightly different cell representation than the one used by policy-based Go-Explore. Specifically, the DTSIL cell representation includes the cumulative positive reward achieved; the representations are otherwise the same.

13 No-ops and sticky actions

The Atari benchmark has been accepted as a common RL benchmark because of the large variety of independent environments it provides[26]. One downside of the games available in this benchmark is that they are inherently deterministic, making it possible to achieve high scores by simply memorising state-action mappings or a fixed sequence of actions, rather than learning a policy able to generalise to the much larger number of states that are available in each game. As the community is interested in learning general policies rather than open-loop solutions, many have suggested approaches to improve the benchmark, usually by adding some form of stochasticity to the game[14].

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

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

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

14 PPO and SIL

Both the robustification "backward" algorithm and the implementation of policy-based Go-Explore are based on the actor-critic-style PPO algorithm from Schulman *et al.* (2017)[20], wherein N parallel actors collect data in mini-batches of T timesteps, and policy updates are performed after each batch. In all PPO-based algorithms presented here, the loss of the policy and value function (both parameterized by θ) is defined as

$$\mathcal{L}(\theta) = \mathcal{L}^{PG}(\theta) + w_{VF}\mathcal{L}^{VF}(\theta) + w_{ENT}\mathcal{L}^{ENT}(\theta) + w_{L2}\mathcal{L}^{L2} + w_{SIL}\mathcal{L}^{SIL}(\theta) \tag{1}$$

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

$$\mathcal{L}^{PG}(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}[\max(-A_t r_t^\pi(\theta), -A_t \text{clip}(r_t^\pi(\theta), 1-\epsilon, 1+\epsilon))] \tag{2}$$

$$r_t^{\pi}(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3}$$

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

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{4}$$

$$\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \tag{5}$$

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

$$\mathcal{L}^{VF}(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}[\max((V_\theta(s_t) - \hat{R}_t)^2, (\text{clip}(V_\theta(s_t) - V_{\theta_{old}}(s_t), -\epsilon, \epsilon) - \hat{A}_t)^2)] \tag{6}$$

$$\hat{R}_t = V_{\theta_{old}}(s_t, g_t) + \hat{A}_t \tag{7}$$

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

Finally, both the robustification algorithm and policy-based Go-Explore include a self-imitation learning (SIL) loss[37], $\mathcal{L}^{SIL}(\theta)$, which is calculated over previously collected data $\mathcal{D}$. While the source of $\mathcal{D}$ differs between the robustification algorithm and policy-based Go-Explore (see SI "Multiple demonstrations" and Methods "Policy-based Go-Explore" for details), in both cases $\mathcal{D}$ comes from previously collected trajectories τ and consists of tuples (s, a, R), where s is a state encountered in one of these rollouts, a is the action that was taken in that state, and R is the discounted reward that was collected from that state. With SIL, a small number (N_{SIL}) of PPO's actors are assigned to be SIL actors. Each of these SIL actors, instead of taking actions in the environment, replays one of these trajectories τ, and at each iteration the data collected by these

SIL actors forms the data set $\mathcal{D}$. The $\mathcal{L}^{SIL}(\theta)$ loss is then calculated as

$$\mathcal{L}^{SIL}(\theta) = \mathcal{L}^{SIL\_PG}(\theta) + w_{SIL\_VF}\mathcal{L}^{SIL\_VF}(\theta) + w_{SIL\_ENT}\mathcal{L}^{SIL\_ENT}(\theta) \tag{8}$$

$$\mathcal{L}^{SIL\_PG}(\theta) = \mathbb{E}_{s,a,R\in\mathcal{D}}[-\log\pi_\theta(a|s) \cdot \max(0, R - V_{\theta_{old}}(s))] \tag{9}$$

$$\mathcal{L}^{SIL\_VF}(\theta) = \mathbb{E}_{s,a,r\in\mathcal{D}}\left[\frac{1}{2}\max(0, R - V_\theta(s))^2\right] \tag{10}$$

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

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

15 Backward algorithm details

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

15.1 Multiple demonstrations. The original version of the backward algorithm relied on a single demonstration due to the assumption that obtaining human demonstrations was expensive. In our case, however, obtaining multiple demonstrations is easy and cheap by simply re-running the exploration phase, which is why we modified the algorithm to utilise multiple demonstrations: at the start of each episode, a demonstration is chosen at random to provide the starting point for the agent. For Atari, 10 demonstrations from different runs of the exploration phase were used for each robustification. In Atari, the demonstration was extracted by finding all trajectories that reached an end-of-episode state (to prevent selection of length 0 trajectories in games with exclusively negative rewards, see "Score tracking in the exploration phase"), and extracting the shortest one among those with the highest score. For robotics, because it was possible to extract diverse demonstrations from a single run of the exploration phase (this was not possible in Atari because high-scoring trajectories within a given Atari run tended to share most of their actions), 10 demonstrations from the same runs were used for robustification. In robotics, the first demonstration corresponds to the shortest successful trajectory (i.e. the shortest trajectory that puts the object in the shelf), while each subsequent demonstration corresponds to the successful trajectory with the highest mean difference from all previously selected trajectories, where the difference between two trajectories is given by $\frac{\sum_{i=1}^{L} I(\tau_i^a \neq \tau_i^b)}{L}$ (τ^a and τ^b are the list of actions being compared, $L = \min(|\tau^a|, |\tau^b|)$, I is the indicator function). Because actions are continuous, meaning that it is exceedingly unlikely for two independently sampled actions to be the same, $\tau_i^a = \tau_i^b$ only for parts where the trajectories are identical because they were branched from the same intermediate trajectory. As such, this metric effectively measures to what degree the two trajectories have a shared history, and prefers trajectories that share as little history as possible. In both cases, SIL was also performed on the set of demonstrations provided to the backward algorithm (see the "PPO and SIL" section). Each robustification run used demonstrations from different, non-overlapping exploration phase runs (10 exploration runs for Atari, 1 for robotics).

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

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

$$m = \frac{C}{\mu_V} \tag{11}$$

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

$$\mu_V = \frac{1}{\sum_{d=1}^{D} T_d} \sum_{d=1}^{D} \sum_{t=1}^{T_d} |V_d(t)| \tag{12}$$

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

16 Score tracking in the exploration phase

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

17 Robustification scores analysis

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

18 Comparing Go-Explore and Agent57

The creation of the Atari benchmark started the search for reinforcement learning algorithms capable of achieving super-human performance on all games in this benchmark[26]. For a majority of these games, super-human performance was reached quickly through early deep reinforcement learning techniques now considered standard[15], but for a small set of games super-human performance remained out of reach. The work presented in this paper achieved this historic feat concurrently with an algorithm called Agent57[13]. Go-Explore and Agent57 accomplish this milestone via very different methods, offering the scientific community a diversity of promising tools to use and build upon going forward.

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

So, while Agent57 achieved the milestone of superhuman performance on the last few remaining games at the same time as Go-Explore, its method is vastly different from Go-Explore.

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

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

19 ALE issues

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

First, the score on Montezuma's Revenge rolls over (i.e. is subject to numerical overflow) when it exceeds 1 million, which is incorrectly interpreted by the ALE as a negative reward of -1 million. We patched the environment to remove this bug and thereby make it possible for algorithms to learn to produce scores higher than 1 million. In addition, we removed an arbitrary time limit of 400,000

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

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

frames, imposed by OpenAI Gym, that is not inherent to the game. This enabled us to learn that Go-Explore can substantially outperform the human world record of 1.2 million[33], with one agent frequently reaching a score of over 40 million after 12.5 million frames (the equivalent of about 58 hours of continuous game play). On Montezuma's Revenge, no previous work had achieved scores anywhere near high enough to trigger this bug. It is similarly unlikely that the performance of previous work was limited by the OpenAI Gym time limit on Montezuma's Revenge.

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

20 Infrastructure

In terms of infrastructure, each exploration phase run was performed on a single worker machine equipped with 44 CPUs and 96GB of RAM, though memory usage is substantially lower for most games. Each robustification run was parallelized across 8 worker machines each equipped with 11 CPUs, 24GB of RAM, and 1 GPU. Policy-based Go-Explore was parallelized across 16 worker machines each equipped with 11 CPUs, 10GB of RAM, and 1 GPU.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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