A. Implementation details

A.1. Replay buffer

In addition to training our policy on the goals that were generated in the current iteration, we also save a list ("regularized replay buffer") of goals that were generated during previous iterations (`update_replay`). These goals are also used to train our policy, so that our policy does not forget how to achieve goals that it has previously learned. When we generate goals for our policy to train on, we sample two thirds of the goals from the Goal GAN and we sample the one third of the goals uniformly from the replay buffer. To prevent the replay buffer from concentrating in a small portion of goal space, we only insert new goals that are further away than ϵ from the goals already in the buffer, where we chose the goal-space metric and ϵ to be the same as the ones introduced in Section 3.1.

A.2. Goal GAN Initialization

In order to begin our training procedure, we need to initialize our goal generator to produce an initial set of goals (`initialize_GAN`). If we initialize the goal generator randomly (or if we initialize it to sample uniformly from the goal space), it is likely that, for most (or all) of the sampled goals, our initial policy would receives no reward due to the sparsity of the reward function. Thus we might have that all of our initial goals g have $\bar{R}^g(\pi_0) < R_{\min}$, leading to very slow training.

To avoid this problem, we initialize our goal generator to output a set of goals that our initial policy is likely to be able to achieve with $\bar{R}^g(\pi_i) \geq R_{\min}$. To accomplish this, we run our initial policy $\pi_0(a_t \mid s_t, g)$ with goals sampled uniformly from the goal space. We then observe the set of states S^v that are visited by our initial policy. These are states that can be easily achieved with the initial policy, π_0, so the goals corresponding to such states will likely be contained within S_0^I. We then train the goal generator to produce goals that match the state-visitation distribution $p_v(g)$, defined as the uniform distribution over the set $f(S^v)$. We can achieve this through traditional GAN training, with $p_{\text{data}}(g) = p_v(g)$. This initialization of the generator allows us to bootstrap the Goal GAN training process, and our policy is able to quickly improve its performance.

B. Experimental details

B.1. Ant specifications

The ant is a quadruped with 8 actuated joints, 2 for each leg. The environment is implemented in Mujoco (Todorov et al., 2012). Besides the coordinates of the center of mass, the joint angles and joint velocities are also included in the observation of the agent. The high degrees of freedom make navigation a quite complex task requiring motor coordination. More details can be found in Duan et al. (2016), and the only difference is that in our goal-oriented version of the Ant we append the observation with the goal, the vector from the CoM to the goal and the distance to the goal. For the Free Ant experiments the objective is to reach any point in the square $[-5m, 5m]^2$ on command. The maximum time-steps given to reach the current goal are 500.

B.2. Ant Maze Environment

The agent is constrained to move within the maze environment, which has dimensions of 6m x 6m. The full state-space has an area of size 10 m x 10 m, within which the maze is centered. To compute the coverage objective, goals are sampled from within the maze according to a uniform grid on the maze interior. The maximum time-steps given to reach the current goal are 500.

B.3. Point-mass specifications

For the N-dim point mass of Section 5.3, in each episode (rollout) the point-mass has 400 timesteps to reach the goal, where each timestep is 0.02 seconds. The agent can accelerate in up to a rate of 5 m/s^2 in each dimension ($N = 2$ for the maze). The observations of the agent are $2N$ dimensional, including position and velocity of the point-mass.

B.4. Goal GAN design and training

After the generator generates goals, we add noise to each dimension of the goal sampled from a normal distribution with zero mean and unit variance. At each step of the algorithm, we train the policy for 5 iterations, each of which consists of 100 episodes. After 5 policy iterations, we then train the GAN for 200 iterations, each of which consists of 1 iteration of training the discriminator and 1 iteration of training the generator. The generator receives as input 4 dimensional noise sampled from the standard normal distribution. The goal generator consists of two hidden layers with 128 nodes, and the goal discriminator consists of two hidden layers with 256 nodes, with relu nonlinearities.

B.5. Policy and optimization

The policy is defined by a neural network which receives as input the goal appended to the agent observations described above. The inputs are sent to two hidden layers of size 32 with tanh nonlinearities. The final hidden layer is followed by a linear N-dimensional output, corresponding to accelerations in the N dimensions. For policy optimization, we use a discount factor of 0.998 and a GAE lambda of 0.995. The policy is trained with TRPO with Generalized Advantage Estimation implemented in rllab (Schulman et al., 2015a;b; Duan et al., 2016). Every "update_policy" consists of 5

iterations of this algorithm.

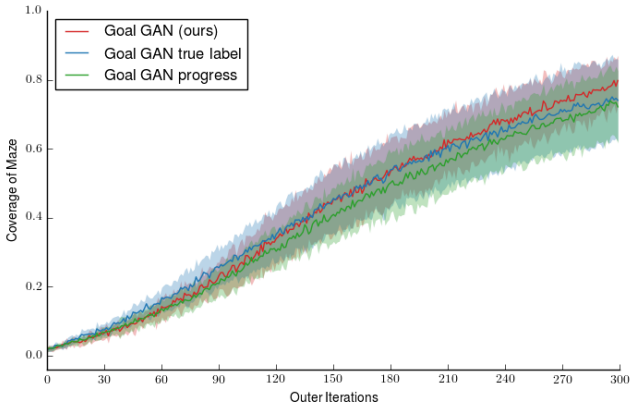
C. Study of GoalGAN goals

To label a given goal (Section 4.1), we could empirically estimate the expected return for this goal $\bar{R}^g(\pi_i)$ by performing rollouts of our current policy π_i. The label for this goal is then set to $y_g = \mathbb{1}\left\{R_{\min} \leq \bar{R}^g(\pi_i) \leq R_{\max}\right\}$. Nevertheless, having to execute additional rollouts just for labeling is not sample efficient. Therefore, we instead use the rollouts that were used for the most recent policy update. This is an approximation as the rollouts where performed under π_{i-1}, but as we show in Figs. 8a-8b, this small "delay" does not affect learning significantly. Indeed, using the true label (estimated with three new rollouts from π_i) yields the *Goal GAN true label* curves that are only slightly better than what our method does. Furthermore, no matter what labeling technique is used, the success rate of most goals is computed as an average of at most four attempts. Therefore, the statement $R_{\min} \leq \bar{R}^g(\pi_i)$ will be unchanged for any value of $R_{\min} \in (0, 0.25)$. Same for $\bar{R}^g(\pi_i) \leq R_{\max}$ and $R_{\max} \in (0.75, 1)$. This implies that the labels estimates (and hence our automatic curriculum generation algorithm) is almost invariant for any value of the hypermparameters $R_{\min}$ and $R_{\max}$ in these ranges.

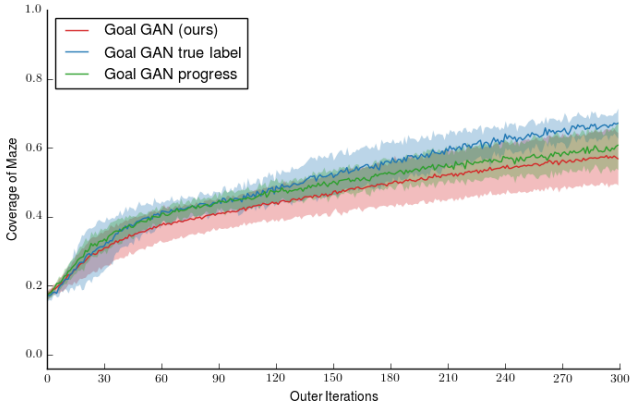
In the same plots we also study another criteria to choose the goals to train on that has been previously used in the literature: learning progress (Baranes & Oudeyer, 2013b; Graves et al., 2017). Given that we work in a continuous goal-space, estimating the learning progress of a single goal requires estimating the performance of the policy on that goal before the policy update and after the policy update (potentially being able to replace one of these estimations with the rollouts from the policy optimization, but not both). Therefore the method does require more samples, but we deemed interesting to compare how well the metrics allow to automatically build a curriculum. We see in the Figs. 8a-8b that the two metrics yield a very similar learning, at least in the case of Ant navigation tasks with sparse rewards.

D. Goal Generation for Free Ant

Similar to the experiments in Figures 3 and 4, here we show the goals that were generated for the Free Ant experiment in which a robotic quadruped must learn to move to all points in free space. Figures 9 and 10 show the results. As shown, our method produces a growing circle around the origin; as the policy learns to move the ant to nearby points, the generator learns to generate goals at increasingly distant positions.



(a) Free Ant - Variants



(b) Maze Ant - Variants

Figure 8. Learning curves comparing the training efficiency of our method and different variants. All plots are an average over 10 random seeds.

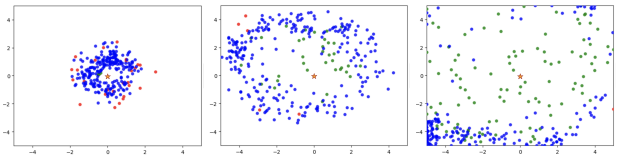
E. Learning for Multi-path point-mass

To clearly observe that our GoalGAN approach is capable of fitting multimodal distributions, we have plotted in Fig. 11 only the samples coming from the GoalGAN (i.e. no samples from the replay buffer). Also, in this environment there are several ways of reaching every part of the maze. This is not a problem for our algorithm, as can be seen in the full learning curves in Fig.12, where we see that all runs of the algorithm reliably reaches full coverage of the multi-path maze.

F. Comparisons with other methods

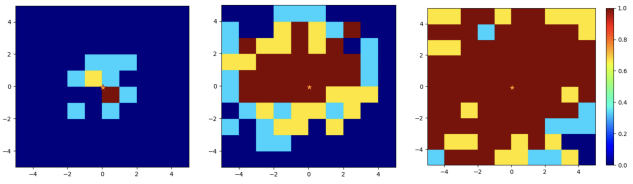
F.1. Asymmetric self-play (Sukhbaatar et al., 2017)

Although not specifically designed for the problem presented in this paper, it is straight forward to apply the method proposed by Sukhbaatar et al. (2017) to our problem. An interesting study of its limitations in a similar setting can be found in (Florensa et al., 2017b).



(a) Iteration 10 (b) Iteration 100 (c) Iterartion 300

Figure 9. Goals that our algorithm trains on (200 sampled from the Goal GAN, 100 from the replay). "High rewards" (green) are goals with $\bar{R}^g(\pi_i) \geq R_{\max}$; $GOID_i$ (blue) have appropriate difficulty for the current policy $R_{\min} \leq \bar{R}^g(\pi_i) \leq R_{\max}$. The red ones have $R_{\min} \geq \bar{R}^g(\pi_i)$



(a) Iteration 10: Coverage = 0.037 (b) Iteration 100: Coverage = 0.4 (c) Iteration 300: Coverage = 0.86

Figure 10. Visualization of the policy performance for different parts of the state space (same policy training as in Fig. 9). For illustration purposes, the feasible state-space is divided into a grid, and a goal location is selected from the center of each grid cell. Each grid cell is colored according to the expected return achieved on this goal: Red indicates 100% success; blue indicates 0% success.

F.2. SAGG-RIAC (Baranes & Oudeyer, 2013b)

In our implementation of this method, we use TRPO as the "Low-Level Goal-Directed Exploration with Evolving Context". We therefore implement the method as batch: at every iteration, we sample N_{new} new goals $\{y_i\}_{i=0...N_{new}}$, then we collect rollouts of t_{max} steps trying to reach them, and perform the optimization of the parameters using all the collected data. The detailed algorithm is given in the following pseudo-code.

UpdateRegions(R, y_f, Γ_{y_f}) is exactly the Algorithm 2 described in the original paper, and **Self-generate** is the "Active Goal Self-Generation (high-level)" also described in the paper (Section 2.4.4 and Algorithm 1), but it's repeated N_{new} times to produce a batch of N_{new} goals jointly. As for the competence Γ_{y_g}, we use the same formula as in their section 2.4.1 (use highest competence if reached close enough to the goal) and $C(y_g, y_f)$ is computed with their equation (7). The `collect_rollout` function resets the state $s_0 = s_{reset}$ and then applies actions following the goal-conditioned policy $\pi_\theta(\cdot, y_g)$ until it reaches the goal or the maximum number of steps t_{max} has been taken. The final state, transformed in goal space, y_f is returned.

As hyperparameters, we have used the recommended ones in the paper, when available: $p_1 = 0.7,\ p_2 = 0.2,\ p_3 = 0.1$.

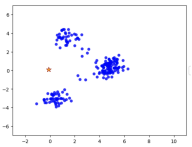


Figure 11. Iteration 10 Goal GAN samples (Fig. 5b without replay samples)

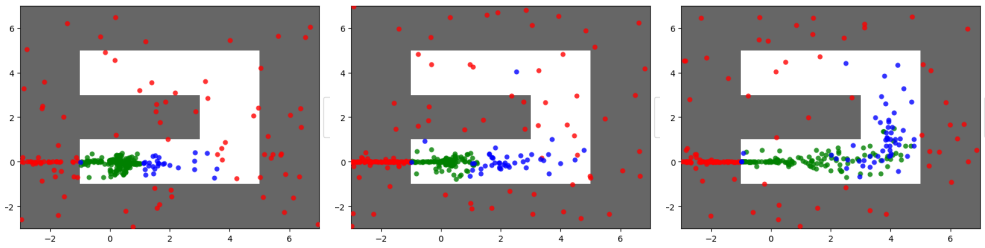
Figure 12. Learning curves of our algorithm on Multi-path Point-mass Maze, consistently achieving full coverage

Algorithm 2 Generative Goal with Sagg-RIAC

Hyperparameters: window size ζ, tolerance threshold ϵ_{max}, competence threshold ϵ_C, maximum time horizon t_{max}, number of new goals N_{new}, maximum number of goals g_{max}, mode proportions (p_1, p_2, p_3)
Input: Policy $\pi_{\theta_0}(s_{start}, y_g)$, goal bounds B_Y, reset position s_{rest}
Output: Policy $\pi_{\theta_N}(s_{start}, y_g)$
R $\leftarrow \{(R_0, \Gamma_{R_0})\}$ where $R_0 = Region(B_Y), \Gamma_{R_0} = 0$
for $i \leftarrow 1$ **to** N **do**
  $goals \leftarrow$ **Self-generate** N_{new} goals: $\{y_j\}_{j=0...N_{new}}$
  $paths = [\,]$
  while $number\_steps\_in(paths) < batch\_size$ **do**
    Reset $s_0 \leftarrow s_{rest}$
    $y_g \leftarrow \text{Uniform}(goals)$
    $y_f, \Gamma_{y_g}, path \leftarrow$ `collect_rollout`$(\pi_{\theta_i}(\cdot, y_g), s_{reset})$
    $paths$`.append`$(path)$
    UpdateRegions(R, $y_f, 0)$
    UpdateRegions(R, $y_g, \Gamma_{y_g})$
  end while
  $\pi_{\theta_{i+1}} \leftarrow$ train π_{θ_i} with TRPO on collected $paths$
end for

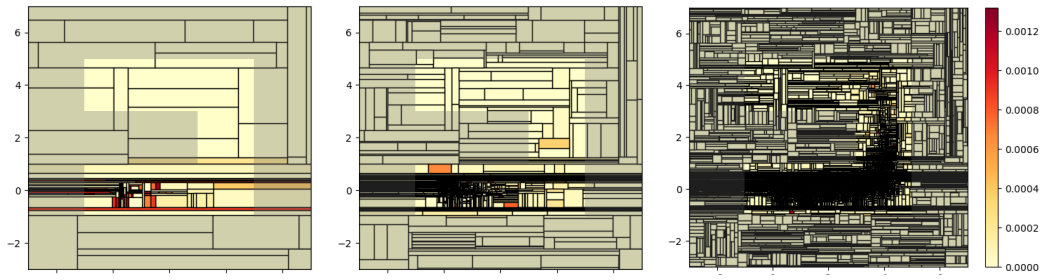
For the rest, the best performance in an hyperparameter sweep yields: $\zeta = 100$, $g_{max} = 100$. The noise for mode(3) is chosen to be Gaussian with variance 0.1, the same as the tolerance threshold ϵ_{max} and the competence threshold ϵ_C.

As other details, in our tasks there are no constraints to penalize for, so $\rho = \emptyset$. Also, there are no sub-goals. The reset value r is 1 as we reset to s_{start} after every reaching attempt. The number of explorative movements $q \in \mathbb{N}$ has a less clear equivalence as we use a policy gradient update with a stochastic policy π_θ instead of a SSA-type algorithm.



(a) Iteration 2 (b) Iteration 20 (c) Iterartion 300

Figure 13. Goals sampled by SAGG-RIAC (same policy training as in Fig. 14). "High rewards" (in green) are goals with $\bar{R}^g(\pi_i) \geq R_{\max}$; $GOID_i$ (in blue) are those with the appropriate level of difficulty for the current policy ($R_{\min} \leq \bar{R}^g(\pi_i) \leq R_{\max}$). The red ones have $R_{\min} \geq \bar{R}^g(\pi_i)$



(a) Iteration 2:
Num. of Regions = 54

(b) Iteration 100:
Num. of Regions = 1442

(c) Iteration 300:
Num. of Regions = 15420

Figure 14. Visualization of the regions generated by the SAGG-RIAC algorithm