A. Comparison of Functions Used by Multi-Goal Agents

	SELECT	REWARD	RELABEL
HER (Andrychowicz et al., 2017)	Samples from environment	Sparse environment reward	`future`
SAGG-RIAC (Baranes & Oudeyer, 2013)	Chooses goals in areas where absolute learning progress is greatest	Uses "competence", defined as the normalized negative Euclidean distance between final achieved goal and goal.	N/A
Goal GAN (Florensa et al., 2018)	Samples from a GAN trained to generate goals of intermediate difficulty	Sparse environment reward	N/A
RIG (Nair et al., 2018b)	Samples from the prior of a generative model (VAE) fitted to past achieved goals	$r(s,g) = -\|e(s) - e(g)\|$, Negative Euclidean distance in latent space	50% `future`, 50% samples from generative model
DISCERN (Warde-Farley et al., 2019)	Samples uniformly from a diversified buffer of past achieved goals	$r_T = \max(0, l_g)$, where l_g is dot product between L2-normalized embeddings of the state and goal, and $r_t = 0$ otherwise for $t = 0, \ldots, T-1$	Samples $g' \in \{g'_{T-H}, \ldots, g'_T\}$ and sets $r_T = 1$
CURIOUS (Colas et al., 2018)	Samples from one of several environments in which absolute learning progress is greatest	Sparse environment reward depending on the current environment (module/task)	`future`
CHER (Fang et al., 2019)	Samples from environment	Sparse environment reward	Select based on sum of diversity score of selected goals and proximity score to desired goals
Skew-Fit (Pong et al., 2019)	Samples from a generative model that is skewed to be approximately uniform over past achieved goals	$r(s,g) = -\|e(s) - e(g)\|$, Negative Euclidean distance in latent space	50% `future`, 50% samples from generative model
DDLUS (Hartikainen et al., 2020)	Selects maximum distance goals according to a learned distance function	$r(s,g) = -d^{\pi}(s,g)$, Negative expected number of time steps for a policy π reach goal g from state s	N/A
MEGA (Ours, 2020)	Selects low density goals according to a learned density model	Sparse environment reward	`rfaab` (Appendix C)

Table 1. High level summary of SELECT, REWARD, RELABEL functions used by various multi-goal algorithms for Algorithm 1.

B. Worked Propositions

Proposition 1 (Discrete Entropy Gain). *Given buffer $\mathcal{B}$ with $\eta = \frac{1}{|\mathcal{B}|}$, maximizing expected next step entropy is equivalent to maximizing expected point-wise entropy gain $\Delta H(g')$:*

$$\begin{aligned}\hat{g}^* &= \arg\max_{\hat{g}\in\mathcal{B}} \mathbb{E}_{g'\sim q(g' \mid \hat{g})} H[p_{ag \mid g'}] \\ &= \arg\max_{\hat{g}\in\mathcal{B}} \mathbb{E}_{g'\sim q(g' \mid \hat{g})} \Delta H(g'),\end{aligned} \tag{9}$$

where $\Delta H(g') = p_{ag}(g') \log p_{ag}(g') - (p_{ag}(g') + \eta) \log(p_{ag}(g') + \eta)$.

Proof. Expanding the expression for maximizing the expected next step entropy:

$$\hat{g}^* = \arg\max_{\hat{g}\in\mathcal{B}} \mathbb{E}_{g'\sim q(g'|\hat{g})} H[p_{ag \mid g'}] \tag{10}$$

$$= \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g}) \sum_{g} -p_{g'}(g) \log p_{g'}(g) \tag{11}$$

We write the new empirical achieved goal distribution $p_{ag \mid g'}(g)$ after observing achieved goal g' in terms of $\eta = \frac{1}{|\mathcal{B}|}$ and the original achieved goal distribution $p_{ag}(g)$:

$$p_{ag \mid g'}(g) = \frac{p_{ag}(g) + \eta \mathbb{1}[g = g']}{1 + \eta} \tag{12}$$

Substituting the expression of $p_{ag \mid g'}(g)$ into above, we can ignore several constants when taking the $\arg\max$ operation:

$$\hat{g}^* = \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g}) \sum_{g} -\frac{p_{ag}(g) + \eta\mathbb{1}[g=g']}{1+\eta} \log \frac{p_{ag}(g) + \eta\mathbb{1}[g=g']}{1+\eta} \tag{13}$$

$$= \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g}) \sum_{g} -(p_{ag}(g) + \eta\mathbb{1}[g=g']) \log(p_{ag}(g) + \eta\mathbb{1}[g=g']) \tag{14}$$

We can break down the summation for the new entropy $H[p_{ag \mid g'}] = H[p_{ag}] + \Delta H(g')$ as the difference between the original entropy $H[p_{ag}]$ and the difference term $\Delta H(g')$. Then we can discard the original entropy term which is constant with respect to the sampled achieved goal g':

$$\hat{g}^* = \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g})[H[p_{ag}] + \Delta H(g')] \tag{15}$$

$$= \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g}) \big[\big(-\sum_{g} p_{ag}(g)\log p_{ag}(g)\big) + \big(p_{ag}(g')\log p_{ag}(g') - (p_{ag}(g')+\eta)\log(p_{ag}(g')+\eta)\big)\big] \tag{16}$$

$$= \arg\max_{\hat{g}\in\mathcal{B}} \sum_{g'} q(g'|\hat{g}) \big[p_{ag}(g')\log p_{ag}(g') - (p_{ag}(g')+\eta)\log(p_{ag}(g')+\eta)\big] \tag{17}$$

$$= \arg\max_{\hat{g}\in\mathcal{B}} \mathbb{E}_{g'\sim q(g'|\hat{g})} \big[p_{ag}(g')\log p_{ag}(g') - (p_{ag}(g')+\eta)\log(p_{ag}(g')+\eta)\big] \tag{18}$$

Therefore, we have reduced the complexity for computing $\hat{g}^*$ with Equation (18) to $O(d^2)$ from $O(d^3)$ in Equation (10), where d denotes the size of the support.

□

Proposition 2 (Discrete Entropy Gradient)**.**

$$\begin{aligned}\lim_{\eta\to 0} \hat{g}^* &= \arg\max_{\hat{g}\in\mathcal{B}} \langle \nabla_{p_{ag}} H[p_{ag}], q(g' \mid \hat{g}) - p_{ag}\rangle \\ &= \arg\max_{\hat{g}\in\mathcal{B}} \mathrm{D}_{\mathrm{KL}}(q(g' \mid \hat{g}) \parallel p_{ag}) + H[q(g' \mid \hat{g})]\end{aligned} \tag{19}$$

Proof. Putting $p = p_{ag}(g')$ and dividing by η, we have:

$$\begin{aligned}\lim_{\eta\to 0}\frac{\Delta H(g')}{\eta} &= \lim_{\eta\to 0}\frac{p}{\alpha}\log p - \frac{1}{\alpha}(p+\eta)\log(p+\eta) \\ &= \lim_{\eta\to 0} -\log\left(\frac{p+\eta}{p}\right)^{p/\eta} - \log(p+\eta) \\ &= -\log e - \log p \\ &= -1 - \log p.\end{aligned} \tag{20}$$

This is the same as $\nabla_{p_{ag}} H[p_{ag}]$, so that:

$$\lim_{\alpha\to 0} \hat{g}^* = \arg\max_{\hat{g}\in\mathcal{B}} \mathbb{E}_{g'\sim q(g' \mid \hat{g})} \nabla_{p_{ag}} H[p_{ag}](g') = \arg\max_{\hat{g}\in\mathcal{B}} \langle \nabla_{p_{ag}} H[p_{ag}], q(g' \mid \hat{g})\rangle.$$

Then, to get the first equality, we subtract p_{ag} from $q(g' \mid \hat{g})$ inside the inner product since it does not depend on $\hat{g}$, and results in a directional derivative that respects the constraint $\int p = 1$. The second equality follows easily after substituting $\nabla_{p_{ag}} H[p_{ag}] = -1 - \log p$ into $\langle \nabla_{p_{ag}} H[p_{ag}], q(g' \mid \hat{g}) - p_{ag}\rangle$. □

Let us now assume that the empirical p_{ag} is used to induce (abusing notation) a density p_{ag} with full support. This can be done by assuming that achieved goal observations are noisy (note: this is not the same thing as relaxing the Markov state assumption) and using p_{ag} to represent our posterior of the goals that were actually achieved. Then, Proposition 2 extends to the continuous case by taking the functional derivative of differential entropy with respect to the variation $\eta(g) = q(g' \mid \hat{g})(g) - p_{ag}(g)$. We consider only the univariate case $G = \mathbb{R}$ below.

Proposition 2* (Differential Entropy Gradient)**.**

$$\delta H(p_{ag}, \eta) \triangleq \lim_{\epsilon \to 0} \frac{H(p_{ag}(g) + \epsilon\eta(g)) - H(p_{ag}(g))}{\epsilon} = \mathrm{D_{KL}}(q(g' \mid \hat{g}) \parallel p_{ag}) + H[q(g' \mid \hat{g})] - H[p_{ag}] \tag{21}$$

Proof. Since the derivative of $f(x) = x \log x$ is $1 + \log x$, the variation $\delta H(p_{ag}, \eta)$ with respect to η is:

$$\begin{aligned}
\delta H(p_{ag}, \eta) &= -\lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_{-\infty}^{\infty} (p_{ag} + \epsilon\eta) \log(p_{ag} + \epsilon\eta) - (p_{ag} \log p_{ag}) dx \\
&= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_{-\infty}^{\infty} (-1 - \log p_{ag})\epsilon\eta + O((\epsilon\eta)^2) dx \\
&= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_{-\infty}^{\infty} (-1 - \log p_{ag})\epsilon\eta dx + \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_{-\infty}^{\infty} O((\epsilon\eta)^2) dx \\
&= \int_{-\infty}^{\infty} (-1 - \log p_{ag})(q(g' \mid \hat{g}) - p_{ag}) dx \\
&= -\mathbb{E}_{q(g' \mid \hat{g})} \log p_{ag}(x) + \mathbb{E}_{p_{ag}} \log p_{ag}(x) \\
&= \mathrm{D_{KL}}(q(g' \mid \hat{g}) \parallel p_{ag}) + H[q(g' \mid \hat{g})] - H[p_{ag}]
\end{aligned} \tag{22}$$

where the second equality uses Taylor's theorem, and the third equality is justified when the limits exist. As in Proposition 2, this functional derivative is maximized by choosing $q(g' \mid \hat{g})$ to maximize $\mathrm{D_{KL}}(q(g' \mid \hat{g}) \parallel p_{ag}) + H[q(g' \mid \hat{g})]$. □

Proposition 3. *If $q(g'|\hat{g}) = \mathbb{1}[g' = \hat{g}]$, the discrete entropy gradient objective simplifies to a minimum density objective:*

$$\begin{aligned}
\hat{g}^* &= \arg\max_{\hat{g} \in \mathcal{B}} -\log[p_{ag}(\hat{g})] \\
&= \arg\min_{\hat{g} \in \mathcal{B}} p_{ag}(\hat{g})
\end{aligned} \tag{23}$$

Proof. We substitute the the case where $q(g'|\hat{g}) = \mathbb{1}[g' = \hat{g}]$ into the discrete entropy objective and simplify:

$$\lim_{\eta \to 0} \hat{g}^* = \arg\max_{\hat{g} \in \mathcal{B}} \mathrm{D_{KL}}(q(g' \mid \hat{g}) \parallel p_{ag}) + H[q(g' \mid \hat{g})] \tag{24}$$

$$= \arg\max_{\hat{g} \in \mathcal{B}} \mathrm{D_{KL}}(\mathbb{1}[g' = \hat{g}] \parallel p_{ag}) + H[\mathbb{1}[g' = \hat{g}]] \tag{25}$$

$$= \arg\max_{\hat{g} \in \mathcal{B}} \big(\sum_{g'} \mathbb{1}[g' = \hat{g}](\log \mathbb{1}[g' = \hat{g}] - \log p_{ag}(g')) \big) + 0 \tag{26}$$

$$= \arg\max_{\hat{g} \in \mathcal{B}} \log 1 - \log p_{ag}(\hat{g}) \tag{27}$$

$$= \arg\max_{\hat{g} \in \mathcal{B}} -\log p_{ag}(\hat{g}) \tag{28}$$

$$= \arg\min_{\hat{g} \in \mathcal{B}} p_{ag}(\hat{g}) \tag{29}$$

In particular, we eliminate the entropy term $H[\mathbb{1}[g' = \hat{g}]] = 0$, simplify the sum to only consider $g' = \hat{g}$ term, and note that maximizing the negative log probability is equivalent to finding the minimum probability. □

C. Implementation Details

Code Code to reproduce all experiments is available at `https://github.com/spitis/mrl`.

Base Implementation We use a single, standard DDPG agent (Lillicrap et al., 2015) that acts in multiple parallel environments (using Baseline's VecEnv wrapper (Dhariwal et al., 2017)). Our agent keeps a single replay buffer and copy of its parameters and training is parallelized using a GPU. We utilize many of the same tricks as Plappert et al. (2018), including clipping raw observations to [-200, 200], normalizing clipped observations, clipping normalized observations to [-5, 5], and clipping the Bellman targets to $[-\frac{1}{1-\gamma}, 0]$. Our agent uses independently parameterized, layer-normalized (Ba et al., 2016) actor and critic, each with 3 layers of 512 neurons with GeLU activations (Hendrycks & Gimpel, 2016). We apply gradient value clipping of 5, and apply action l2 regularization with coefficient 1e-1 to the unscaled output of the actor (i.e., so that each action dimension lies in [-1., 1.]). No weight decay is applied. We apply action noise of 0.1 to the actor at all exploration steps, and also apply epsilon random exploration of 0.1. Each time the goal is achieved during an exploratory episode, we increase the amount of epsilon random exploration by 0.1 (see Go Exploration below). We use a discount factor (γ) of 0.98 in `PointMaze` (50 steps / episode), `FetchPickAndPlace` (50 steps / episode), and `FetchStack2` (50 steps / episode), and a discount factor of 0.99 in the longer horizon `Antmaze` (500 steps / episode).

Optimization We use Adam Optimizer (Kingma & Ba, 2014) with a learning rate of 1e-3 for both actor and critic, and update the target networks every 40 training steps with a Polyak averaging coefficient of 0.05. We vary the frequency of training depending on the environment, which can stabilize training: we optimize every step in `PointMaze`, every two steps in `Antmaze`, every four steps in `FetchPickAndPlace`, and every ten steps in `FetchStack2`. Since optimization occurs every n environment steps, regardless of the number of environments (typically between 3 and 10), using a different number of parallel environments has neglible impact on performance, although we kept this number the same between different baselines. Optimization steps use a batch size of 2000, which is sampled uniformly from the buffer (no priorization). There is an initial "policy warm-up" period of 5000 steps, during which the agent acts randomly. Our replay buffer is of infinite length (this is obviously inefficient, and one should considering prioritized sampling (Schaul et al., 2015b) and diverse pruning of the buffer (Abels et al., 2018) for tasks with longer training horizons).

Goal Relabeling We generalize the `future` strategy by additionally relabeling transitions with goals randomly sampled (uniformly) from buffers of `actual` goals (i.e., a buffer of the past desired goals communicated to the agent at the start of each episode), past `achieved` goals, and behavioral goals (i.e., goals that agent pursues during training, whether intrinsic or extrinsic). We call this the `rfaab` strategy, which stands for Real (do not relabel), Future, Actual, Achieved, and Behavioral. Intuitively, relabeling transitions with goals outside the current trajectory allows the agent to generalize across trajectories. Relabeling with actual goals focuses optimization on goals from the desired goal distribution. Relabeling with achieved goals could potentially maintain performance with respect to past achieved goals. Relabeling with behavioral goals focuses optimization on past intrinsic goals, potentially allowing the agent to master them faster.

All relabeling is done online using an efficient, parallelized implementation. The `rfaab` strategy requires, as hyperparameters, relative ratios of each kind of experience, as it will appear in the minibatch. Thus, `rfaab_1_4_3_1_1` keeps 10% real experiences, and relabels approximately 40% with `future`, 30% with `actual`, 10% with `achieved`, and 10% with `behavioral`. The precise number of relabeled experiences in each minibatch is sampled from the appropriate multinomial distribution (i.e., with $k = 5$, $n =$ batch size, and the `rfaab` ratios). Note that `rfaab` is a strict generalization of `future`, and that the `future_4` strategy used by Andrychowicz et al. (2017) and Plappert et al. (2018) is the same as `rfaab_1_4_0_0_0`. Because of this, we do not do a thorough ablation of the benefit over future, simply noting that others have found `achieved` (e.g., Nair et al. (2018b)) and `actual` (e.g., Pitis et al. (2019)) sampling to be effective; rather, we simply conducted a random search over the `rfaab` hyperparameters that included pure `future` settings (see Hyperparameter Selection below). We found in preliminary experiments that using a pure `future` warmup (i.e., 100% of experiences relabeled using future achieved goals from the same trajectory) helped with early training, and relabeled goals using this strategy for the first 25,000 environment steps before switching to `rfaab`.

Density Modeling We considered three approaches to density modeling: a kernel density estimator (KDE) (Rosenblatt, 1956), a normalizing flow (Flow) (Papamakarios et al., 2019) based on RealNVP (Dinh et al., 2016), and a random network distillation (RND) approach (Burda et al., 2019). Based on the resulting performances and relatively complexity, we chose to use KDE throughout our experiments.

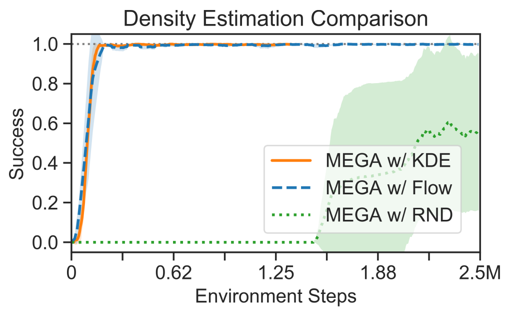

Figure 8. Density modeling on `Pointmaze`.

The KDE approach fits the KDE density model from Scikit-learn (Pedregosa et al., 2011) to normalized samples from the achieved goal distribution (and additionally, in case of OMEGA, the desired goal distribution). Since samples are normalized, the default bandwidth (0.1) and kernel (Gaussian) are good choices, although we ran a random search (see Hyperparameter Selection) to confirm. The model is computationally inexpensive, and so we refit the model on every optimization step using 10,000 normalized samples, sampled uniformly from the buffer.

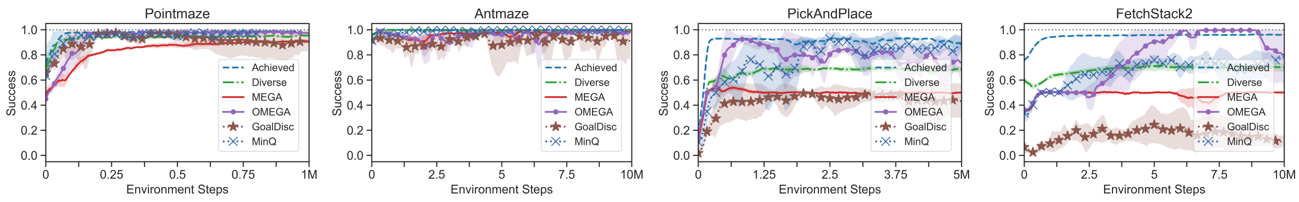

Figure 9. Intrinsic success in various environments, which is the proportion of *training* episodes that are successful on *first visit basis*—that is, training episodes in which the agent achieves its behavioral goal at least one time, on any timestep, using its exploratory policy.

The Flow approach optimizes a RealNVP model online, by training on a mini-batch of 1000 samples from the buffer every 2 optimization steps. It uses 3 pairs of alternating affine coupling layers, with the nonlinear functions modeled by 2 layer fully connected neural networks of 256 leaky ReLU neurons. It is optimized using Adam optimizer with a learning rate of 1e-3.

The RND approach does not train a true density estimator, and is only intended as an approximation to the minimum density (by evaluating relative density according to the error in predicting a randomly initialized network). Both our random network and learning network were modeled using 2 layer fully connected neural networks of 256 GeLU neurons. The learning network was optimized online, every step using a batch size of 256 and stochastic gradient descent, with learning rate of 0.1 and weight decay of 1e-5.

We tested each approach in `PointMaze` only —the results are shown in Figure 8. Both the KDE and Flow models obtain similar performance, whereas the RND model makes very slow progress. Between KDE and Flow, we opted to use KDE throughout our experiments as it is fast, easy to understand and implement, and equally effective in the chosen goal spaces (maximum 6 dimensions). It is possible that a Flow (or VAE-like model (Nair et al., 2018b)) would be necessary in a higher dimensional space, and we expect that RND will work better in high dimensional spaces as well.

Cutoff mechanism and Go Exploration As noted in the main text, the idea behind our minimum density heuristic is to quickly reachieve a past achieved, low density goal, and explore around (and hopefully beyond) it (Ecoffet et al., 2019). If the agent can do this, its conditional achieved goal distribution $q(g' \mid \hat{g})$ will optimize the MEGA objective well (see Propositions). This requires two things: first, that behavioral goals be achievable, and second, that the agent explores around them, and doesn't simply remain in place once they are achieved.

To ensure that goal are achievable, we use a simple cutoff mechanism that crudely prevents the agent from attempting unobtainable goals, as determined by its critic. During SELECT function (Algorithm 2, reproduced below), the agent "eliminates unachieved candidates" using this mechanism. This amounts to rejecting any goal candidates whose Q-values, according to the critic, are below the current cutoff. The current cutoff is initialized at -3., and decreased by 1 every time the agent has an "intrinsic success percentage" of more than 70% over the last 10 training episodes, but never below the minimum of the lowest Q-value in sampled goal candidates. It is increased by 1 every time the agent's intrinsic success percentage is less than 30% over the last 10 training episodes. An agent is intrinsically successful when it achieves its behavioral goal at least one time, on any time step, during training (using its exploratory policy). We can see the intrinsic success for MEGA, OMEGA, and the various baselines in Figure 9. Although we have found that this cutoff mechanism is not necessary and in most cases neither helps nor hurts performance, it helps considerably in certain circumstances (especially, e.g., if we add the environment's desired goal to the candidate goal set in the MEGA_SELECT function of Algorithm 2). See ablations in Appendix D.4. We applied this same cutoff mechanism to all baselines except the Goal Discriminator baseline, which has its own built-in approach to determining goal achievability. Note from Figure 9, however, that the cutoff was rarely utilized for non-MEGA algorithms, since they all quickly obtain high intrinsic success percentages (this is a direct consequence of their lower achieved goal entropy: since they explore less of the goal space in an equal amout of time, they have relatively more time to optimize their known goal spaces—e.g., Achieved sampling maintains close to 100% intrinsic success even in `Stack2`). Note that due to the cutoff mechanism, MEGA's intrinsic success hovers around 50% in the open-ended `Fetch` environments, where the block is often hit off the table and onto the floor and so achieved very difficult to re-achieve goals.

To encourage the agent to explore around the behavioral goal, we increase the agent's exploratory behaviors every time the behavioral goal is reachieved in any given episode. We refer to this as "Go Exploration" after Ecoffet et al. (2019), who used a similar approach to reset the environment to a frontier state, and explored around that state. We use a very simple exploration bonus, which increases the agent's epsilon exploration by a fixed percentage. We use 10% (see next paragraph), which means that an agent which accomplishes the achieved goal 10 times in an episode will be exploring purely at random. Intuitively, this corresponds to "filling in" sparse regions of the achieved goal space (see Figure 10). Likely there are more intelligent ways to do Go Exploration, but we leave this for future work. Note that all baselines benefited from this feature.

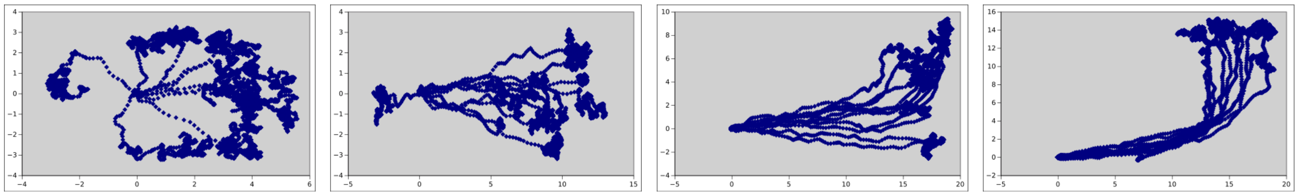

Figure 10. Plots of the last 10 trajectories leading up to 250K, 500K, 750K and 1M environment steps (from left to right) in `Antmaze`. Note that the scales on each plot are different. We see that the agent approximately travels directly to the achieved goal (with some randomness due to exploration noise), at which point it starts to explore rather randomly, forming a flowery pattern and "filling in" the low density region.

Maximum Entropy-based Prioritization (MEP) We implemented MEP (Zhao et al., 2019) in our code base to compare the effects of entropy-based prioritization during the OPTIMIZE step while inheriting the rest of techniques described earlier. We based our implementation of MEP on the official repository[3]. We represented the trajectory as a concatenation of the achieved goal vectors over the episode with T timesteps, $\tau = [g'_0; g'_1; \ldots; g'_T]$. To model the probability density of the trajectory of achieved goals, we used a Mixture of Gaussians model:

$$p(\tau \mid \phi) = \frac{1}{Z} \sum_{i=k}^{K} c_k \mathcal{N}(\tau | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{30}$$

where we denote each Gaussian as $\mathcal{N}(\tau|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, consisting of its mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. c_k refers to the mixing coefficients and Z to the partition function. ϕ denotes the parameters of the model, which includes all the means, covariances, and mixing coefficients for each Gaussian. We used $K = 3$ components in our experiments, as done in (Zhao et al., 2019).

To compute the probability that a trajectory is replayed after prioritization, we followed the implementation code which differed from the mathematical derivation in the main paper. Specifically, we first computed a shifted an normalized negative log-likelihood of the trajectories, $q(\tau_i)$:

$$q(\tau_i) = \frac{-\log p(\tau_i \mid \phi) - c}{\sum_{n=1}^{N} -\log p(\tau_n \mid \phi) - c}, \quad \text{where} \quad c = \min_j -\log p(\tau_j \mid \phi). \tag{31}$$

Then we assign the probability of sampling the trajectory, $\bar{q}(\tau_i))$, as the normalized ranking of $q(\tau_i)$:

$$\bar{q}(\tau_i) = \frac{\text{rank}(q(\tau_i))}{\sum_{n=1}^{N} \text{rank}(q(\tau_n))} \tag{32}$$

The intuition is that the higher probability trajectory $p(\tau)$ will have a smaller $q(\tau)$, and thus also smaller rank, leading to small probability $\bar{q}(\tau)$ of being sampled. Conversely, lower probability trajectory will be over sampled, leading to an increased entropy of the training distribution.

Hyperparameter Selection Many of the parameters above were initially based on what has worked in the past (e.g., many are based on Plappert et al. (2018) and Dhariwal et al. (2017)). We found that larger than usual neural networks, inspired by Nair et al. (2018a), accelerate learning slightly, and run efficiently due to our use of GPU.

To finetune the base agent hyperparameters, we ran two random searches on `PointMaze` in order to tune a MEGA agent. These hyperparameters were used for all agents. First, we ran a random search on the `rfaab` ratios. The search space consisted of 324 total configurations:

- real experience proportion in $\{0, \mathbf{1}, 2\}$

[3] `https://github.com/ruizhaogit/mep`

- future relabeling proportion in $\{3, \mathbf{4}, 5, 6\}$
- actual relabeling proportion in $\{1, 2, \mathbf{3}\}$
- achieved relabeling proportion in $\{\mathbf{1}, 2, 3\}$
- behavioral relabeling proportion in $\{0, \mathbf{1}, 2\}$

We randomly generated 32 unique configurations from the search space ($\sim 10\%$), ran a single seed of each on `PointMaze`, and chose the configuration that converged the fastest—the selected configuration, `rfaab_1_4_3_1_1`, is bolded above. We found, however, that using more `future` labels helped in `FetchPickAndPlace` and `FetchStack2`, for which we instead used `rfaab_1_5_2_1_1`.

Our second random search was done in a similar fashion, over various agent hyperparameters:

- epsilon random exploration in $\{0, \mathbf{0.1}, 0.2, 0.3\}$
- reinforcement learning algorithm in $\{$**DDPG**, TD3$\}$ (Fujimoto et al., 2018)
- batch size in $\{1000, \mathbf{2000}, 4000\}$
- optimize policy/critic networks every n steps in $\{\mathbf{1}, 2, 4, 8\}$
- go exploration percentage in $\{0., 0.02, 0.05, \mathbf{0.1}, 0.2\}$ (Ecoffet et al., 2019)
- action noise in $\{\mathbf{0.1}, 0.2\}$
- warmup period (steps of initial random exploration) in $\{2500, \mathbf{5000}, 10000\}$

In this case the search space consists of 960 total configurations, from which we generated 96 unique configurations (10%) and ran a single seed of each on `PointMaze`. Rather than choosing the configuration that converged the fastest (since many were very similar), we considered the top 10 best performers and chose the common hyper-parameter values between them. We found that performance is sensitive to the "optimize every" parameter, which impacts an agent's stability, and we needed to use less frequent optimization on the more challenging environments. Based on some informal experiments in other environments, we chose to optimize every two steps in `Antmaze`, ever four steps in `FetchPickAndPlace`, and every ten steps in `FetchStack2`.

We additionally ran a third, exhaustive search over the following parameters for the KDE density estimator:

- bandwidth in $\{0.05, \mathbf{0.1}, 0.2, 0.3\}$
- kernel in $\{$exponential, **Gaussian**$\}$

This confirmed our initial thought of using the default hyperparameters with normalized goals.

Finally, to tune our goal discriminator baseline, we ran one seed of each of the following settings for minibatch size and history length $\{(50, 200), (50, 500), (50, 50), \mathbf{(100, 200)}, (100, 500), (100, 1000), (20, 200), (20, 500), (20, 50)\}$ (details of these hyperparameters described below).

Goal Selection and Baselines We reproduce Algorithm 2 to the right, and note that all baselines are simple modifications of line $(*)$ of MEGA_SELECT, except the GoalDisc baseline, which does not eliminate unachievable candidates using the cutoff mechanism. All baselines inherit all of the Base Agent features and hyperparameters described above. Then implementation details of each baseline are described below. For MEGA and OMEGA implementations, please refer to Algorithm 2 and the paragraph on Density Modeling above.

Algorithm 2 O/MEGA SELECT functions

```
function OMEGA_SELECT (env goal $g_{\text{ext}}$, bias $b$, $*args$):
    $\alpha \leftarrow 1/\max(b + \mathrm{D}_{\mathrm{KL}}(p_{dg} \parallel p_{ag}), 1)$
    if $x \sim \mathcal{U}(0,1) < \alpha$ then return $g_{\text{ext}}$
    else return MEGA_SELECT($*args$)

function MEGA_SELECT (buffer $\mathcal{B}$, num_candidates $N$):
    Sample $N$ candidates $\{g_i\}_{i=1}^{N} \sim \mathcal{B}$
    Eliminate unachievable candidates (see text)
    return $\hat{g} = \arg\min_{g_i} \hat{p}_{ag}(g_i)$                    $(*)$
```

1. **Diverse**

   The Diverse baseline scores candidates using $\frac{1}{\hat{p}_{ag}}$, where $\hat{p}_{ag}$ is estimated by the density model (KDE, see above), and then samples randomly from the candidates in proportion to their scores. This is similar to using Skew-Fit with $\alpha = -1$ (Pong et al., 2019) or using DISCERN's diverse strategy (Warde-Farley et al., 2019).

2. **Achieved**

   The Achieved baseline samples a random candidate uniformly. This is similar to RIG (Nair et al., 2018b) and to DISCERN's naive strategy (Warde-Farley et al., 2019).

3. **GoalDisc**

   This GoalDisc baseline adapts Florensa et al. (2018)'s GoalGAN to our setting. To select goals, the Goal Discriminator baseline passes the goal candidates, along with starting states, as input to a trained goal discriminator, which predicts the likelihood that each candidate will be achieved from the starting state. The goals are ranked based on how close the output of the discriminator is to 0.5, choosing the goal with the minimum absolute value distance (the "most intermediate difficulty" goal). We do not use the cutoff mechanism based on Q-values in this strategy.

   To train the discriminator, we start by sampling a batch of 100 of the 200 most recent trajectories (focusing on the most relevant data at the frontier of exploration) (see paragraph on hyperparameters for other considered values). The start state and behavioural goal of each trajectory are the inputs, and the targets are binary values where 1 indicates that the behavioural goal was achieved at some point during the trajectory. The output is continuous valued between 0 and 1, and represents the difficulty of the given start state and behavioural goal combination, in terms of the probability of the agent achieving the goal from the start state. We use the same neural network architecture for the goal discriminator and critic, in terms of layers and neurons; for the discriminator, we apply a sigmoid activation for the final output. We train the discriminator every 250 steps using a binary cross entropy loss.

4. **MinQ**

   The MinQ strategy uses Q-values to identify the most difficult achievable goals. Candidate goals are ranked based on their Q-values, and the goal with the lowest Q-value is selected. This is similar to DDLUS (Hartikainen et al., 2020).

Compute resources Each individual seed was run on a node with 1 GPU and n CPUs, where n was between 3 and 12 (as noted above, we made efforts to maintain similar n for each experiment, although unlike the implementation of Plappert et al. (2017), CPU count does not materially impact performance in our implementation, as it only locally shuffles the order of environment and optimization steps), and the GPU was one of $\{$Nvidia Titan X, Nvidia 1080ti, Nvidia P100, Nvidia T4$\}$.

C.1. Environment Details

`PointMaze` The `PointMaze` environment is identical to that of Trott et al. (2019). The agent finds itself in the bottom left corner of a 10x10 maze with 2-dimensional continuous state and action spaces, and must achieve desired goals sampled from the top right corner within 50 time steps. The state space and goal space are (x, y) coordinates, and the agent moves according to its action, which is a 2-dimensional vector with elements constrained to $(-0.95, 0.95)$. The agent cannot move through walls, which the agent does not see except through experience (i.e., failed actions that press the agent against a wall). Because the agent does not see the walls directly, this is a very difficult exploration environment (indeed, it is the only environment where no seed of any baselines achieves any meaningful success).

`Antmaze` The `Antmaze` environment is identical to that of Trott et al. (2019), which is based on the ant maze environment of Nachum et al. (2018), which expanded the ant maze used in Florensa et al. (2018) by a factor of 4 in each direction (16x the total area). The agent is 3-dimensional ant in a 2-dimensional maze with limits [-4, 20] in both directions. The agent starts on one end of the U-shaped tunnel and must navigate to the desired goal distribution on the other end within 500 timesteps. As compared to `PointMaze` the horizon of this environment is significantly longer, but the exploratory behavior required to solve it is considerably simpler, since there are no deadends and only two 90 degree turns are required.

`FetchPickAndPlace` (Hard) The `FetchPickAndPlace` environment is based on `FetchPickAndPlace-v1` from OpenAI gym (Brockman et al., 2016), which was first introduced by Andrychowicz et al. (2017). In this environment the agent is a robotic arm that must lift a block to a desired 3-dimensional target position. Our only change to the environment is to change the desired goal distribution so that all desired goals are between 20cm and 45cm in the air. This is in contrast the easier `FetchPickAndPlace-v1`, which has 50% of all goals on the table, and the other 50% between 0cm and 45cm in the air. The "easy" goal distribution in `FetchPickAndPlace-v1` was specifically designed so that a plain HER agent could solve the environment (Andrychowicz et al., 2017; Plappert et al., 2018). To our knowledge, we are the first to solve the hard version without behavioral cloning (as done by Nair et al. (2018a)).

`FetchStack2` The `FetchStack2` environment is based on the `Fetch` simulation from OpenAI gym (Brockman et al., 2016), and is intended to replicate the `FetchStack2` tasks used by Duan et al. (2017) and Nair et al. (2018a). In fact, our `FetchStack2` design is somewhat more difficult than the version used by Nair et al. (2018a). Whereas desired goals in Nair et al. (2018a) always place the desired goal above one of the initial block positions (requiring the agent to move only 1 block), our implementation always requires the agent to move both blocks to a new stacked position. The agent has 50 timesteps to move the blocks into position, and success is computed on the last step of the episode (i.e., the blocks must stay stacked). Blocks are initialized in a square of width 6cm about locations (1.3, 0.6) and (1.3, 0.9), and the target position is sampled uniformly in a square of width 30cm in the center of the table. To our knowledge, we are the first to solve this difficult task without demonstrations (Duan et al., 2017) or a task curriculum (Colas et al., 2018).

D. Additional Experiment Details and Experiments

D.1. Number of seeds and plotting details

Figures display the average over a set of seeds, with shaded regions representing 1 standard deviation (our plotting script uses the same logic as that of Plappert et al. (2018) and Dhariwal et al. (2017)). Figures 3 and 5 display 5 seeds for each setting. Figures 2, 8 and 13 display 3 seeds. Visualizations of achieved goals are all from the same seed. Hyperparameter search was run with 1 seed for each setting, as described above.

D.2. FetchPickAndPlace - Increasing Horizon - Details

For this experiment, we modified the desired goal distribution of `FetchPickAndPlace` to be uniform over the range stated in the legend of Figure 2 and ran the HER and OMEGA agents (as described above) in the modified environment.

D.3. Minimum Density Approximation Versus Learned Conditional for Entropy Gain

In this section we investigate directly learning a conditional model $q(g'|\hat{g})$ of the achieved goal g' given the behavioural goal $\hat{g}$, in order to estimate the entropy gain, and compare it with our minimum density approximation.

For modeling the conditional $q(g'|\hat{g})$, we factorize the conditional using the joint and the behavioural goal models: $q(g'|\hat{g}) = p(g', \hat{g})/p_{bg}(\hat{g})$, both implemented with KDE. For the joint, we sample pairs of achieved and behavioural goals $(g', \hat{g})$ from the replay buffer, concatenate them in the input dimension (doubling the dimension), then fit the KDE. For the behavioural goal, we only sample the behaviour goals to train the marginal KDE. Similarly to the KDE for the achieved goal buffer, we also refit the model on every optimization step using 10,000 normalized samples, sampled uniformly from the respective buffers. We draw joint samples from the replay buffer then compute its conditional probability using the KDE model, instead of a generative approach where we sample potential achieved goals given behaviour goal from a conditional density model. This prevents potential hallucinating of unachievable goals, such as a coordinate outside the `PointMaze`.

During behavioural goal selection, we sample N candidates goals $\{g_i\}_{i=1}^{N} \sim \mathcal{B}$ from achieved goal replay buffer. For each candidate goals, we also sample $K = 10$ achieved goals, where one of the goal is the candidate goal itself. Given this set of $\{g'_i, \hat{g}_j\}_{i=1,\dots,K, j=1,\dots,N}$ candidate behaviour goal and achieved goal pairs, we compute the monte carlo estimate of the expected entropy gain for each candidate behaviour $\hat{g}_j$ using Equation 9.

We experiment on `PointMaze` to compare using minimum density versus learned conditional for estimating entropy gain for choosing behavioural goal. Figure 11 (left) illustrates that learning the conditional to estimate the entropy gain does lead to slightly faster learning progress (e.g. less environment steps to reach 95% success rate) than if we use the minimum density sampling. However, because we need to learn and perform inference on the conditional model as well, we find that the our current implementation of the entropy gain approach needs *triple* the wall clock time compared to the simpler minimum density variant of MEGA/OMEGA.

In theory, choosing the behavioural goals greedily based on the expected entropy gain estimation method should yield higher

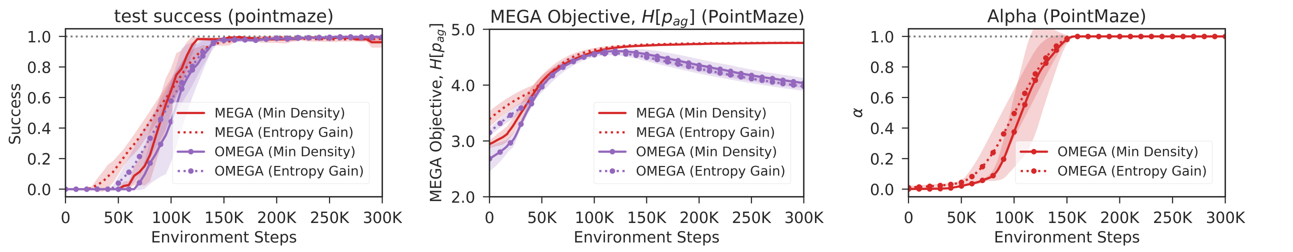

Figure 11. Comparing with MEGA and OMEGA on (**Left**) Test Success, (**Middle**) Achieved Goal buffer KDE estimated entropy, and (**Right**) α (OMEGA only) when using Minimum Density versus Estimated Entropy Gain on `PointMaze`.

entropy in the historical achieved goal buffer (MEGA objective), compared to the minimum density approximation. In practice, the approach depends on the the quality of the conditional model $q(g'|\hat{g})$ to give estimates of the entropy gain for each behavioural goal. Figure 11 (middle) shows that initially while the conditional model is being trained on limited data, the entropy of the historical achieved goals (MEGA objective) is actually less than if we simply use the minimum density sampling. However, as the training progresses, the conditional model is able to help estimate more optimal behavioural goals that lead to more entropy gain, overtaking the minimum density sampling approach. Finally, Figure 11 (right) compares the α parameter for OMEGA method, where the entropy gain approach starts to take off later than minimum density, but has a quicker raise to the top, correlating to the test success plot.

Due to the similar test success and MEGA objective performance while being much simpler and faster to run, we perform the rest of the experiments with the minimum density variant of our MEGA/OMEGA method.

D.4. Ablation of implementation features

As described above, our implementation uses three "tricks" that are only indirectly related to our MEGA objective, but that we found to be helpful in certain circumstances. They are: (1) `rfaab` goal sampling, (2) the use of a Q-value cutoff to eliminate unachievable goal candidates, and (3) an increase in action space exploration upon achieving intrinsic goals during exploration ("goexp"). Figure 12 ablates these features on `Pointmaze` and `FetchStack2`. We observe that `rfaab` sampling improves performance in both cases relative to HER's `future_4` strategy. While the cutoff slightly hurts performance on `Pointmaze`, it sometimes prevents the agent from diverging in more complex environment. Finally, we observe that go exploration substantially improves results on `Pointmaze`, and sometimes allows the agent to find a solution on `FetchStack2` when it otherwise wouldn't have. Note that the `FetchStack2` agent uses slightly revised hyperparameters here, which we found to be more stable than the ones used in our main experiments. The differences are: less action l2 regularization (1e-2), more frequent target network updates (every 10 steps), and no gradient value clipping.

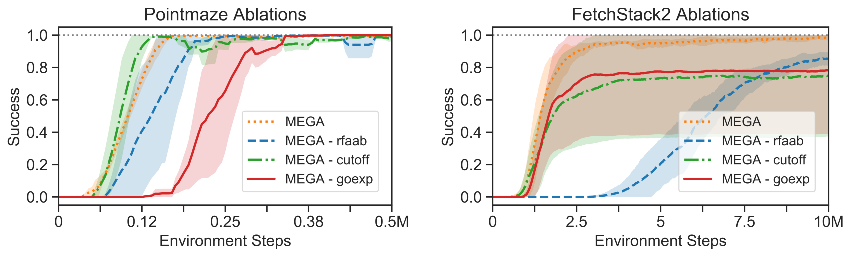

Figure 12. Ablation of certain implementation features in `Pointmaze` and `FetchStack2`.

D.5. MEGA Tested on Smaller AntMaze

We also ran HER and MEGA agents on an Antmaze of the same size as used by Florensa et al. (2018). We set the desired goal distribution to be the same as the grid used by Florensa et al. (2018) to compute their coverage objective, so that "success" in this desired goal space is effectively the same as the objective used by Florensa et al. (2018) (note that the HER agent samples directly from this goal distribution during training). The results are shown in Figure 13. We see that both agents are able to solve this smaller `Antmaze` in a fraction of the time required by the TRPO-based agents in Florensa et al. (2018) (more

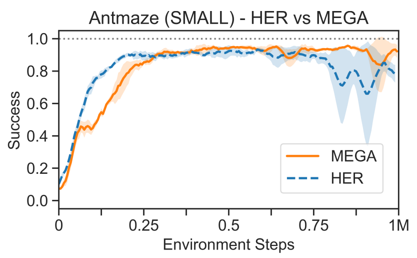

Figure 13. Small `Antmaze` results

than 1000 times faster!). This is more a testament to the power of off-policy learning and goal relabeling than it is to MEGA, since (1) this smaller version is not really a long horizon environment, and (2) our adaptation of Florensa et al. (2018)'s GOID (Goals of Intermediate Difficulty) criterion (GoalDisc) solves the larger `Antmaze` in approximately the same amount of time as MEGA.

D.6. Toy Example: Discrete Entropy Gain

To compare how different strategies optimize the MEGA objective in a controlled environment, we consider an MDP with goal space $g \in \{0, 1, ..., 2n\}$. The initial achieved goal buffer is $\mathcal{B} = \{n\}$. At each iteration, the agent chooses behaviour goal $\hat{g} \in \mathcal{B}$, samples $g' \sim q(g' \mid \hat{g})$ from the induced conditional, and adds g' to $\mathcal{B}$, where $q(g'|\hat{g})$ is defined as:

$$q(g'|\hat{g}) = \begin{cases} 0.4 & \text{if } g' = \hat{g} \\ 0.2 & \text{if } g' = \hat{g} \pm 1 \\ 0.1 & \text{if } g' = \hat{g} \pm 2, \\ 0 & \text{otherwise.} \end{cases}$$

At the boundaries, we truncate the $q(g' \mid \hat{g})$ and normalize.

We consider four policies for choosing $\hat{g}$ at each iteration:

1. $\pi_{\text{Achieved}}(g) = p_{ag}(g)$, which samples uniformly from the buffer, as used, approximately, by RIG (Nair et al., 2018b)
2. $\pi_{\text{Diverse}}(g) \propto p_{ag}(g) \cdot p_{ag}(g)^{-1}$, which is equivalent to sampling uniformly on the support set of the buffer, as approximated by DISCERN's (Warde-Farley et al., 2019) "diverse" strategy and Skew-Fit (Pong et al., 2019) (Skew-Fit parameterizes the exponent with $\alpha \in [-1, 0)$)
3. $\pi_{\text{MEGA}}(g) \propto \mathbb{1}[g = \arg\min p_{ag}(g)]$ (ours), which samples from the goal(s) with the minimum density or mass
4. $\pi_{\text{EG}}(g) = \mathbb{1}[g = \arg\min L(\hat{g})]$, which is an oracle that chooses $\hat{g}$ to maximize the next step entropy gain (9)

Figure 14 plots empirical entropy over iterations, where $n = 50$, averaged over 50 trials. Our minimum density heuristic is almost as fast as the oracle π_{EG}, and converges to the max entropy distribution much faster than π_{Achieved} and π_{Diverse}.

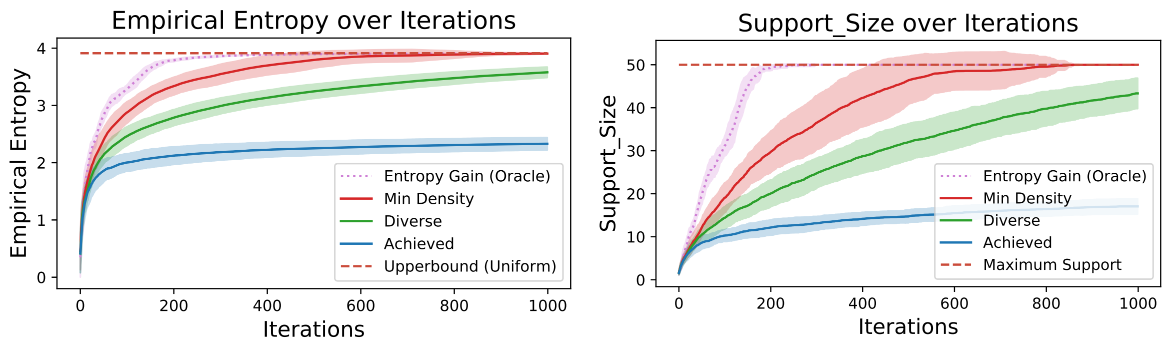

Figure 14. (**Left**) Empirical entropy of the buffer over time for different goal selection strategies in toy example of Section D.6. (**Right**) Support size of the buffer data over time steps with different behaviour goal selection strategies in the discrete goal toy example in Section D.6. The curves plots the mean averaged over 50 runs and shades one standard deviation in each direction.

We also visualize the empirical probability distribution of the achieved goal buffer over iterations in Figure 15, when using the oracle entropy gain sampling $\pi_{EG}(g)$ (top) which has access to ground truth conditional $q(g' \mid \hat{g})$, versus using the minimum density $\pi_{MEGA}(g)$ (bottom). With the oracle $\pi_{EG}(g)$, the support size (number of non-white rows at a given vertical slice) grows consistently at each iteration. The minimum density $\pi_{MEGA}(g)$ approach does spend time sampling goals in the "interior" section to "even out" the distribution (e.g. around iteration 55 to 100) without increasing the support size, before discovering new unseen goals that increase the support size.

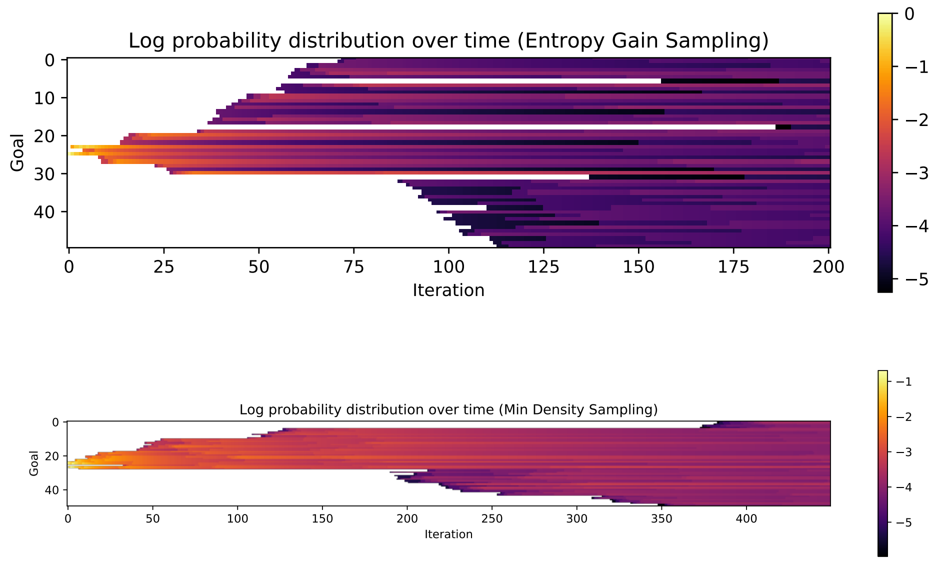

Figure 15. Log probability (denoted by the color) for each achieved goal ($n = 50$ possible goals represented in the y-axis) versus iterations for (top) entropy gain sampling, (bottom) minimum density sampling. Note the difference in the scale of the iteration axis between the two plots. Goals with zero probability (not in support) are shaded white. Best viewed in colour.