Exploration in Approximate Hyper-State Space for Meta Reinforcement Learning

Supplementary Material

A. Additional Background

A.1. Randomised Prior Functions

In reinforcement learning, we can use the fact that unseen states can be seen as out-of-distribution data of a model that is trained on all data the agent has seen so far. Getting uncertainty estimates on states can thus quantify our uncertainty about the value of a state and in turn whether we have explored these states sufficiently. We can think about why exploration purely in the state space $\mathcal{S}$ (which is shared across tasks) is not enough: if the agent has explored a state many times in one task and is certain of its value, it should not necessarily exploit this knowledge in a different task, because this same state could have a completely different value. We cannot view these as separate exploration problems however, since we also have to try out different deployed exploration strategies and combine the information to meta-learn Bayes-optimal behaviour.

Therefore, we want to incentivise the agent to explore in the hyper-state space $\mathcal{S}^+ = \mathcal{S} \times \mathcal{B}$. Only if an environment state together with a specific belief has been observed sufficiently often to determine its value should the agent trust its value estimate of that belief-state. This therefore amounts to exploration in a BAMDP state space, which essentially means trying out different exploration strategies in the environments of the training distribution. We use Random Network Distillation (RND) (Osband et al., 2018; Burda et al., 2019b; Ciosek et al., 2020) to obtain such uncertainty estimates and review them using the formulation of Ciosek et al. (2020) in the following.

Assume we are given a set of training data $\mathcal{D} = \{s_i\}_{i=1}^N$ of all states the agent has observed. To get uncertainty estimates, we first fit B predictor networks $g_j(s)$ ($j = 1, \dots, B$) to a random prior process $f_j(s)$ each (a network with randomly initialised weights, which is fixed and never updated). We then estimate the uncertainty for a state s_* as

$$\sigma^2(s_*) = \max(0,\ \sigma^2_\mu(s_*) + \beta v_\sigma(s_*) - \sigma^2_A), \qquad (7)$$

where $\sigma^2_\mu(s_*)$ is the sample mean of the squared errors between the B predictor networks and the prior processes; $v_\sigma(s_*)$ is the sample variance of the squared error. The first quantifies our uncertainty, whereas the second quantifies our uncertainty over what our uncertainty is. In practice, $B = 1$ is typically sufficient and the second term disappears (Ciosek et al., 2020). The term σ^2_A is the aleatoric noise inherent in the data which is an irreducible constant. In theory, this can be learned as well and depends on how much information can be extracted about the value of states and actions from the data. In practice, we set this term to 0.

Given a hyper-state $s^+_t = (s_t, b_t)$, an ensemble of B prior networks $\{f^i(s^+)\}_{i=1}^B$ and corresponding predictor networks $\{h^i(s^+)\}_{i=1}^B$, the reward bonus is defined as

$$r_c(s^+_t) = \max(0,\ \sigma_m u^2(s^+_t) + \beta v_\sigma(s^+_t) - \sigma^2_A) \qquad (8)$$

where $\sigma_m u^2(s^+_t)$ is the sample mean of the squared error between prior and predictor networks and $v_\sigma(s^+_t)$ is the sample variance of that error.

B. Additional Results

In this section we provide additional experimental results. The first two sections are additional environments – in particular sparse environments used in the literature before, but where we found that our baselines already performed very well. In addition, we provide more details and results for the experiments in the main paper.

Implementation details, including hyperparameters and environment specifications, are given in Appendix C. The (anonymised) source code is attached as additional supplementary material.

B.1. Meta-World

To test how our method scales up to more challenging problem settings, we evaluate it on the Meta-World benchmark (Yu et al., 2019), where a simulated robot arm has to perform tasks. We evaluate our method on the ML1 benchmark, of which three different versions exist: reach/push/pick-and-place (in increasing order of complexity). In each of these, task distributions are generated by varying the starting position of the agent and the goal/object positions.

Each environment has a dense reward function that was designed such that an agent trained on a single task (i.e., fixed starting/object/goal position) can learn to solve it. Evaluation is done in terms of success rate (rather than return), which is a task-specific binary signal indicating whether the task was accomplished (at any moment during the rollout). Yu et al. (2019) proposed a sparse version of this benchmark that uses this binary success indicator, rather than the dense reward, for training. This sparse version was used in (Zhang et al., 2020), on ML1-reach and ML1-push.

The agent is trained on a set of 50 environments and evaluated on unseen environments from the same task distribution. In all baselines, the agent has 10 episodes to adapt, and performance is measured in the last rollout. Since we consider the *online adaptation* setting where we want the agent has to perform well from the start, we trained VariBAD and HyperX to maximise online return during the first two episodes. This is more challenging since it includes exploratory actions.

		Dense Rewards			Sparse Rewards		
Method	Test Episode	Reach	Push	Pick-Place	Reach	Push	Pick-Place
MAML*	10	48	74	12	-	-	-
PEARL*	10	38	71	28	-	-	-
RL2*	10	45	87	24	-	-	-
E-RL2+	10	-	-	-	28	7	-
MetaCURE+	10	-	-	-	46	25	-
VariBAD	**1**	**100**	**100**	29 (6/20 seeds)	**100**	**100**	2 (1/20 seeds)
VariBAD	2	100	100	29	100	100	2
HyperX	**1**	**100**	**100**	**43** (9/20 seeds)	100	100	2 (1/20 seeds)
HyperX	2	100	100	43	100	100	2

Table 2. **Meta-test success rates on the ML1 Meta-World benchmark, for the dense and the sparse reward version.** *Results taken from Yu et al. (2019). +Results taken from Zhang et al. (2020). We ran VariBAD and HyperX for 5 random seeds for dense reach/push, and 20 seeds for dense pick-place. VariBAD and HyperX were trained to maximise expected online return within 2 episodes. The first (few) episodes often *includes exploratory actions*, yet have higher success rate than existing methods that maximise final episodic return. For the sparse Pick-Place environment, in brackets we report the number of seeds that learned something.

Table 2 shows the results for both the dense and sparse versions of ML1.

ML1-reach / ML1-push. VariBAD achieves 100% success rate on both the dense *and the sparse* version of ML1-reach and ML1-push *in the first rollout*. Compared to other existing methods – even MetaCURE (Zhang et al., 2018) which explicitly tries to deal with sparsity – this is a significant improvement. We confirm in our experiments that HyperX does not decrease performance and also reaches 100% success rate on these environments.

ML1-pick-place. The environment ML1-pick-place is more challenging, because the task consists of two steps: picking up an object and placing it somewhere (where both the object and goal location differ across tasks). Even on the dense version, existing methods struggle. HyperX achieves state of the art on this task with 44.5% success rate, suggesting HyperX can help meta-learning even when rewards are dense. For VariBAD and HyperX we found that our agents either learn the task near perfectly (and have close to 100% success rate in the first rollout), or not at all. VariBAD learned something for 6 out of 20 seeds, and HyperX learned something for 9 out of 20 seeds. For the sparse version of this environment, we only saw some success for 1/20 seeds for both VariBAD or HyperX.

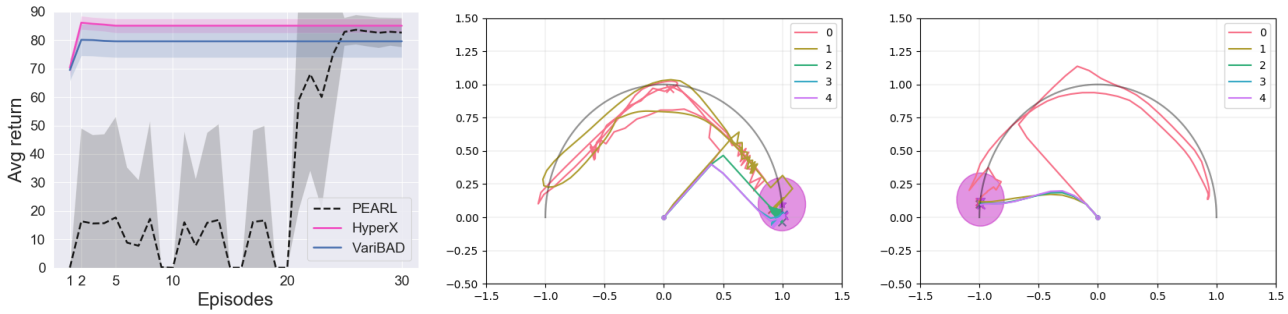
We suspect that the main challenge in ML1-Pick-Place is the short horizon (150), which does not give the agent enough time to explore during meta-training. This is why HyperX can give some improvement even in the dense version. In an upcoming version of Meta-World (Yu et al., 2019), the horizon will be increased to 200, opening up interesting opportunities for future research on sparse Pick-Place.

B.2. Sparse 2D Navigation

We evaluate on a Point Robot 2D navigation task used by Gupta et al. (2018); Rakelly et al. (2019); Humplik et al. (2019). The agent must navigate to an unknown goal sampled along the border of a semicircle of radius 1.0, and receives a reward relative to its proximity to the goal when it is within a goal radius of 0.2. Thus far, only Humplik et al. (2019) successfully meta-learn to solve this task by meta-training with sparse rewards, though they rely on privileged information during meta-training (the goal position). The other methods meta-train with dense rewards and evaluate using sparse rewards. We use a horizon of 100 here (instead of 20 as in the papers above) to give VariBAD and HyperX enough time to demonstrate interesting exploratory behaviour.

Figure 7A shows the performance of PEARL, VariBAD, and HyperX at test time, when rolling out for 30 episodes. Both VariBAD and HyperX adapt to the task quickly compared to PEARL, but HyperX reaches slightly lower final performance.

To shed light on these performance differences, Figures 7B and 7C visualise representative example rollouts for the meta-trained VariBAD and HyperX agents. We picked examples where the target goals are at the end of the semicircle, which we found are most difficult for the agents. VariBAD (7B) struggles to find the goal, taking several attempts to reach it. Once the goal is found, it does return to it but on a sub-optimal trajectory. By contrast, HyperX searches the space of goals more strategically, and returns to the goal faster in subsequent episodes.



(A) Meta-Test Performance (B) VariBad example rollout (C) HyperX example rollout

Figure 7. Meta-test performance on the Sparse 2D Navigation environment. *Left:* Performance averaged over the task distribution at the end of training. Because PEARL is not optimizing for optimal exploration, it requires many more episodes to find the goal. Both VariBad and HyperX optimise for optimal exploration and are able to quickly find the goal. However, VariBad's exploration is suboptimal, not covering all possible goal locations equally well (see middle plot), explaining the lower performance compared to HyperX.

B.3. Treasure Mountain

Ablations. Figure 8A shows the learning curves for the HyperX, in comparison to ablating different exploration bonuses. When using only the hyper-state novelty bonus r^{hyper}, HyperX learns the inferior strategy of walking in a circle: it has no incentive to go up the mountain early in training (because beliefs there are meaningless because the VAE has not learned yet to interpret the hint) and stars avoiding the mountain. When using only the VAE reconstruction error bonus r^{error}, the agent learns the superior strategy of walking up the mountain to see the goal location 70% of the time (7/10 seeds). In contrast, HyperX, which uses both exploration bonuses, learns the superior strategy for all 10 seeds. Lastly, we tested VariBAD with a simple state novelty exploration bonus: this again learns the inferior circle-walking strategy only, because it quickly learns to avoid the mountain top.

Baselines - Performance. Figure 8B shows the learning curves for HyperX and VariBAD (discussed in Sec 5.1), as well as additional baselines RL^2 (Duan et al., 2016; Wang et al., 2016) (which is a model-free method where the policy is a recurrent network that gets previous actions and rewards as inputs in addition to the environment state) and the Belief Learning method of Humplik et al. (2019) (which uses privileged information – the goal position – during meta-training). Both these baselines also only learn the inferior circle-walking strategy, because the correct incentives for meta-exploration are missing.

Baselines - Behaviour. Figures 8C and 8D show meta-test time behaviour of VariBAD and RL^2: both methods learn to walk in a circle until the goal is found. This was consistent across all (10) seeds.

B.4. Sparse CheetahDir

Figure 9 shows the learning curves for the Sparse CheetahDir experiments, with 95% confidence intervals (over 20 seeds). Fig 9A shows this for the Belief Oracle, with different exploration bonuses. Fig 9B shows this for HyperX, with different exploration bonuses.

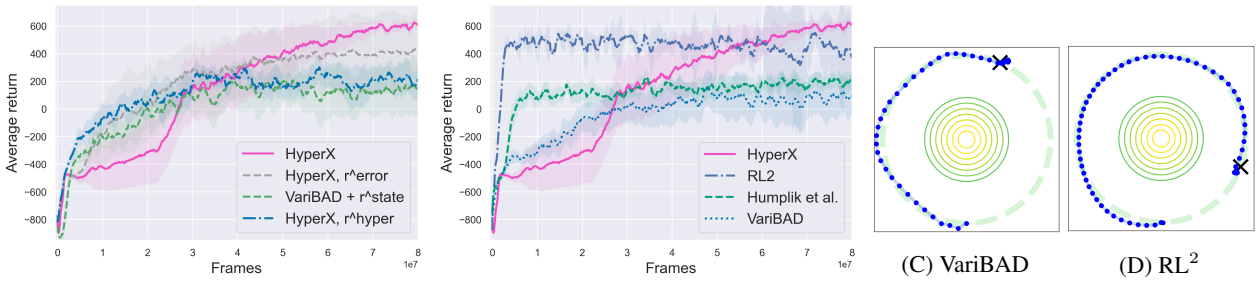
Figure 9C shows example behaviour of a suboptimal policy at test time. The agent returns back into the zero-reward zone after realising that the task was not "go left", but stays in there instead of behaving optimally, which is going further to the right and into the dense reward area beyond the sparse interval border.

B.5. Sparse MuJoCo AntGoal

In addition to the main results in the paper (Sec 5.4) we provide additional experimental results here.

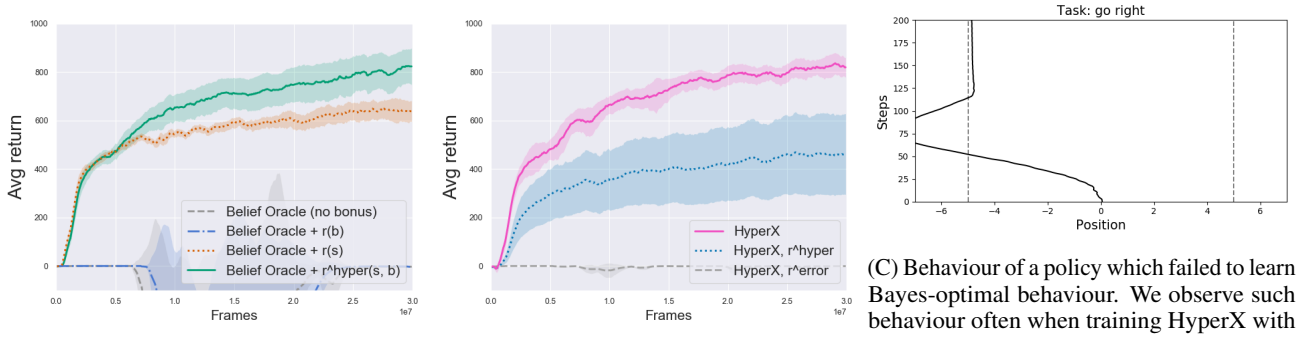
Figure 10A shows the returns achieved by the agents across different episodes. Figures 10B show the learning curves for the returns during the *first* episode, with 95% confidence intervals (shaded areas, 10 seeds). Figure 10C shows the combined learning curves, comprising of all 6 episodes, with 95% confidence intervals (shaded areas, 10 seeds). Figures 12 and 11 show example rollouts for VariBAD and HyperX.

Dense AntGoal. We also evaluated HyperX on the dense AntGoal environment. VariBAD and HyperX were trained to maximise performance within a single episode. PEARL was trained with the default hyperparameters provided by the open-sourced code of the authors. The results are:: VariBAD: -123 (Episode 1), HyperX: -127 (Episode 1), PEARL: -200 (Episode 6). This confirms that HyperX does not impact performance, but that there is also not much room for improvement.



(A) Learning curves for ablations. (B) Learning curves for baselines. (C) VariBAD (D) RL^2

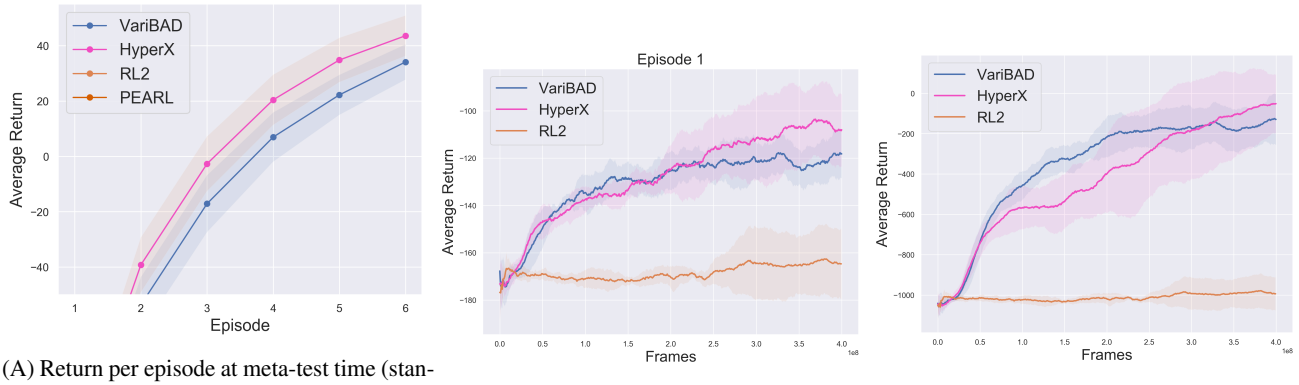
Figure 8. **Treasure Mountain - Additional Rollouts.** Shown are example rollouts for the final agents of VariBAD (Zintgraf et al., 2020) and RL^2 (Duan et al., 2016; Wang et al., 2016). They follow the inferior exploration strategy of walking around the circle until the treasure is found, instead of climbing the mountain to directly observe the treasure and get there faster.



(A) Belief Oracle. (B) HyperX

(C) Behaviour of a policy which failed to learn Bayes-optimal behaviour. We observe such behaviour often when training HyperX with the reward bonus on the hyper-states only, $r^{hyper}(b, s)$.

Figure 9. **HalfCheetahDir: Additional Results.** Learning curves for the Belief Oracle (A) and HyperX (B), with and without reward bonus, averaged over 20 seeds..



(A) Return per episode at meta-test time (standard error shaded). RL2 and PEARL do not learn to solve the task and achieve a reward of around -150 per episode.

(B) Learning curve (return in ep 1)

(C) Learning Curve (sum of returns in ep 1-6)

Figure 10. **Sparse AntGoal: Additional Plots.** (10 seeds).

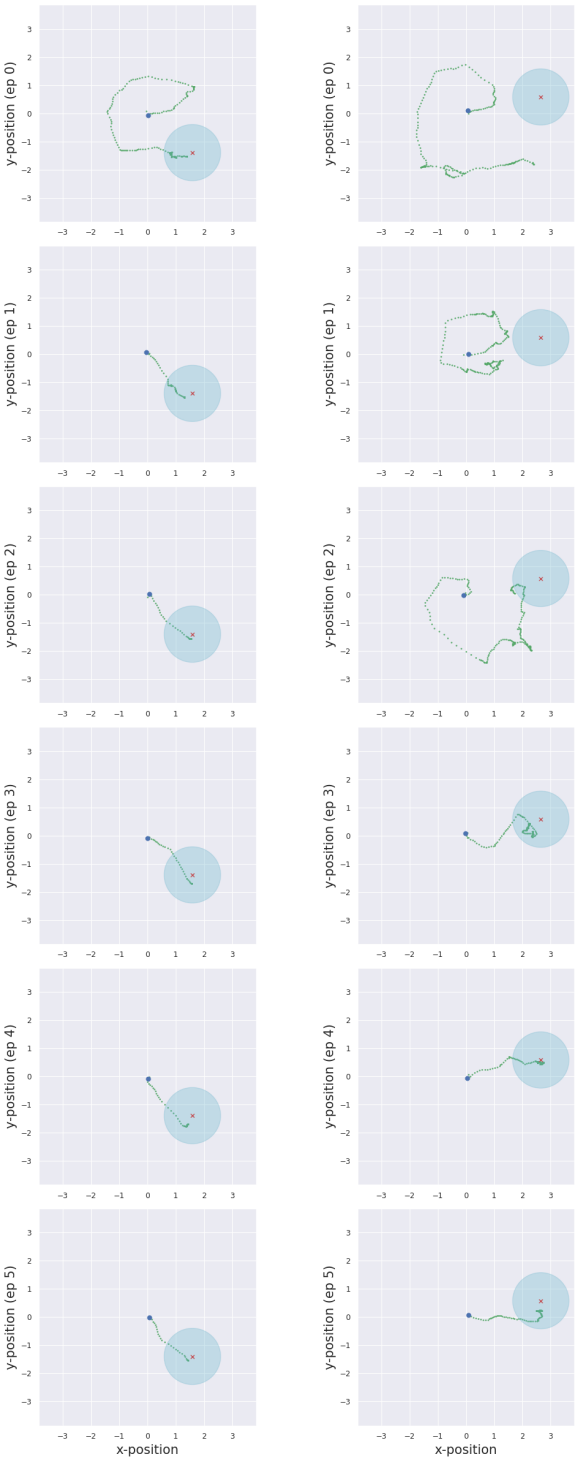


Figure 11. HyperX Example Rollouts

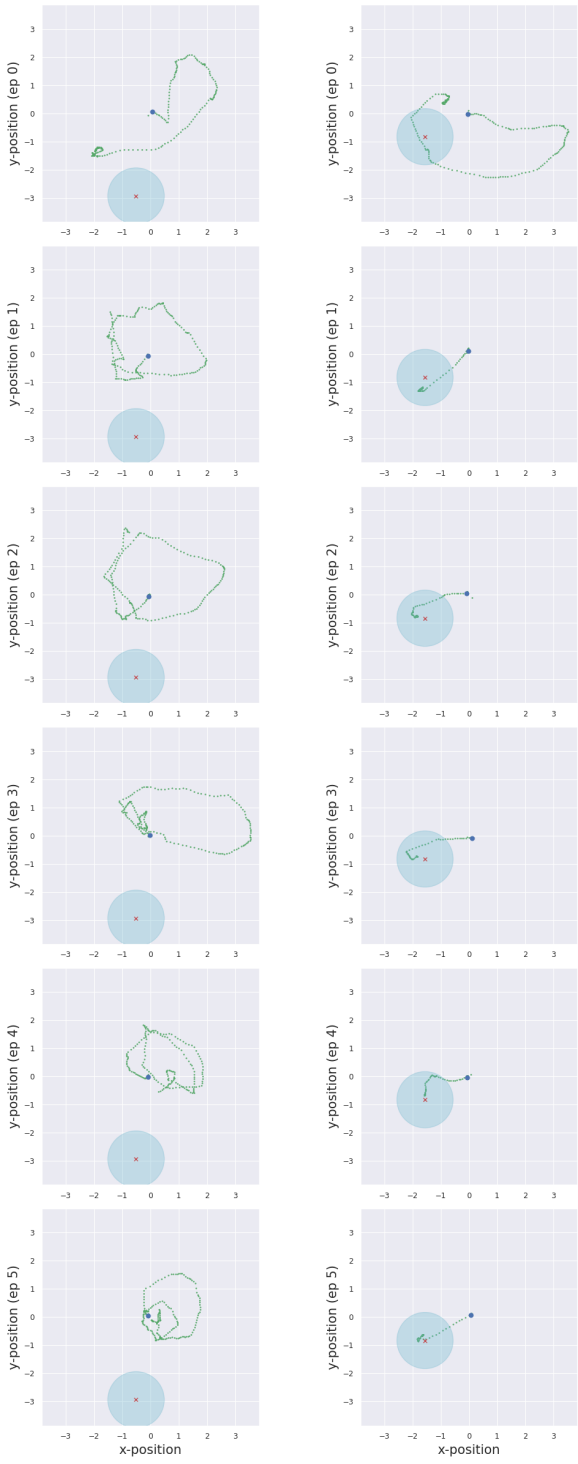


Figure 12. VariBAD Example Rollouts

C. Implementation Details

The source code is available as additional supplementary material. In this section, we provide the environment specifications (C.1), runtimes (C.5), and hyperparameters (C.6).

C.1. Environment Specifications

In this section we provide additional details on the environments that were used in the main paper. Implementation of these environments are in the provided source code.

C.1.1. Treasure Mountain

This environment is implemented as follows. The treasure can be anywhere along a circle of radius 1. Within that circle is a mountain – implemented as another circle with radius 0.5. The horizon is 100 and there are no resets. The agent always starts at he bottom of the circle. It receives a reward of 10 when it is within a Euclidean distance of 0.1 within the treasure (the treasure does not disappear, so it keeps receiving this reward if it stays there). It receives a penalty for climbing the mountain, given my $-5.5 + ||(x, y)||_2$ where (x, y) is the agent's position (the mountain center is $0, 0$, and the mountain radius 0.5). If not at the treasure or on the mountain, the agent gets a timestep penalty of at least -5, which increases as the agent walks further outside the outer circle (to discourage it from walking too far). The agent cannot walk outside $[-1.5, 1.5]$ in either direction.

The observations of the agent are 4D and continuous. The first two dimensions are the agent's (x, y)-position. The last two dimensions are zero if the agent is not on the mountain top, and are the (x, y)-coordinates of the treasure when the agent is on the mountain top (within a radius of 0.1). The agent's actions are the (continuous) stepsize it takes in (x, y)-direction, bounded in $[-0.1, 0.1]$.

C.1.2. Multi-Stage Gridworld

The layout of this environment is depicted in Fig 3. It consist of three rooms which are of size 3×3 grid, and corridors that connect the rooms of length 3. The environment state is the (x, y) position of the agent, unnormalised. There are five available actions: *no-op, up, right, down, left*.

Three (initially unknown) goals (G1-G3) are placed in corners of rooms: G1 in the middle room, G2 in the room that is on the side where G1 was placed, and G3 in the middle room (but not where G1 was placed). The agent always starts in the middle of the centre room and has $H = 50$ steps. The goals provide increasing rewards, i.e. $r_1 = 1$, $r_2 = 10$ and $r_3 = 100$, but are only sequentially unlocked; G2 (r_2) is only available after G1 has been reached; G3 (r_3) is only available after G2 has been reached. The environment is partially observable (Poupart & Vlassis, 2008; Cai et al., 2009) as the agent only observes its position in the environment and not which goals are unlocked. If the agent is not on an (available) goal it gets $r = -0.1$. When the agent stands on a goal, it keeps receiving the respective reward while standing there (the goal does *not* disappear). The best strategy is to search the first room for G1, then search the appropriate room for G2, and then return to the middle room to find G3.

C.2. Sparse HalfCheetahDir

We use the commonly used HalfCheetahDir meta-learning benchmark (based on code of Zintgraf et al. (2020)), and sparsify it as follows. If the agent's x-position is within $[-5, 5]$ it only gets the control penalty; otherwise it gets the standard dense reward comprised of the sum of the control penalty and the 1D velocity in the correct direction.

C.3. Sparse MuJoCo AntGoal

We use the commonly used AntGoal meta-learning benchmark (based on code of Rakelly et al. (2019)), and sparsify it as follows. We extend the environment's state space by including the x and y-position of the agent's torso. In the original AntGoal, the goal is sampled from within a circle of radius of 3 with a higher chance of the goal being sampled away from the centre of the circle. Unlike the dense version where the agent receives a dense goal-related reward at all times, our sparse AntGoal only receives goal-related rewards when within a radius of 1 of the goal.

The agent receives at all time a control penalty and contact forces penalty. When outside the goal circle, the agent receives an additional constant negative reward that is equivalent to the negative goal radius, i.e. -1. When within the goal circle, the agent receives a reward of 1 for being within the goal circle and a penalty equivalent to the negative distance to the goal, essentially encouraging the agent to walk towards the centre of the goal circle.

C.4. Meta-World

We use the official version of Meta-World as provided by Yu et al. (2019) at `https://github.com/rlworkgroup/metaworld`. As suggested by Yu et al. (2019) and as tested in Zhang et al. (2020), for the sparse version of this environment, we use the *success* criterion which the environment returns, and give the agent a reward of 0 if *success=False* and a reward of 1 if *success=True*. The success criterion depends on the environment; in 'Reach' for example it is *true* if the agent put its gripper close to the (initially unknown) goal position, and *false* otherwise. For evaluation, we report 'Success' if the agent was successful at any moment during an episode, following the evaluation protocol proposed by Yu et al. (2019).

C.5. Runtimes

Table 3 shows the runtimes for our experiments. Unless otherwise stated, we used a NVIDIA GeForce GTX 1080 GPU. These runtimes should serve as a rough estimate, and can vary depending on hardware and concurrent processes.

Environment	Frames	Runtime (ca.)
Treasure Mountain	$8e{+}7$	35h
Multi-Stage Gridworld	$1e{+}8$	65h (CPU)
Sparse HalfCheetahDir	$3e{+}7$	20h (CPU)
Sparse AntGoal	$4e{+}8$	65h
Meta-World	$5e{+}7$	45h
Sparse 2D Navigation	$5e{+}7$	12h

Table 3.

C.6. Hyperparameters

We train the policy using PPO, and we add the intrinsic bonus rewards to the extrinsic environment reward and use the sum when learning with PPO. We normalise the intrinsic and extrinsic rewards separately by dividing by a rolling estimate of the standard deviation.

On the next two pages we show the hyperparameters used for the policy, the VAE, and the exploration bonuses. Hyperparameters were selected using a simple (non-exhaustive) gridsearch.

For the MuJoCo environments, we only used the relevant state information for the RND hyper-state bonus (the x-axis for HalfCheetahDir, and the x-y-position for AntGoal).

RND Hyperparameter Sensitivity. To assess how sensitive HyperX to choices of hyperparameters that affect the hyperstate exploration bonus, we evaluated it on a range of different choices, shown in Table 4. There is little sensitivity to architecture depth and batchsize, as well as to the output dimension of the RND networks. Performance is stable for learning rates $10^{-3}{-}10^{-6}$ (possibly because we use the Adam optimiser), but we found that the best frequency ($freq$) at which the RND network is updated to be environment dependent. Performance is sensitive to the scaling factor (wsi in the table) for the initial prior network weights. We used a scaling factor of 10 in our experiments, and found that too small or too large scaling factors can hurt performance. An interesting direction for future work is to find more principled ways to guide the choice of the hyperparameters that are particularly sensitive to the exploration and across environments.

RND $dim_{out} = 32$ (default 128)	737
RND $dim_{out} = 256$ (default 128)	812
RND $depth = 1$ (default 2)	794
RND $depth = 3$ (default 2)	814
RND $batchsize = 32$ (default 128)	856
RND $batchsize = 256$ (default 128)	867
RND $lr = 1e-2$ (default $1e-4$)	108
RND $lr = 1e-3$ (default $1e-4$)	883
RND $lr = 1e-5$ (default $1e-4$)	845
RND $lr = 1e-6$ (default $1e-4$)	766
RND $wsi = 1$ (default 10)	597
RND $wsi = 5$ (default 10)	766
RND $wsi = 15$ (default 10)	533

Table 4. Additional Sparse CheetahDir Results, for different RND hyperparameter settings (averaged over three seeds). wsi stands for weight scale initialisation of the fixed random prior network.

	Treasure	GridWorld	CheetahDir	AntGoal	PointRobot	ML1-Reach	ML1-Push	Ml1-Pick-Place
max_rollouts_per_task	1	1	1	6	3	2	2	2
policy_state_embedding_dim	None	32	32	64	64	64	64	64
policy_latent_embedding_dim	None	32	32	64	64	64	64	64
norm_state_for_policy	False	True	True	True	True	True	True	True
norm_latent_for_policy	False	True	False	True	True	True	True	True
norm_rew_for_policy	True	True	True	False	True	True	True	True
norm_actions_pre_sampling	False	False	False	True	True	False	False	False
norm_actions_post_sampling	True	False	False	False	False	False	False	True
norm_rew_clip_param	100	NaN	NaN	100000	100000	100000	100000	100000
policy_layers	[128, 128]	[64]	[128, 128]	[128, 128, 128]	[128, 128, 128]	[128, 128]	[128, 128]	[128, 128]
policy_activation_function	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh
policy_initialisation	orthogonal	normc	normc	normc	normc	normc	normc	normc
policy_anneal_lr	False	False	False	False	False	False	False	False
policy	ppo	ppo	ppo	ppo	ppo	ppo	ppo	ppo
policy_optimiser	adam	adam	adam	adam	adam	adam	adam	adam
ppo_num_epochs	2	8	2	2	2	2	2	2
ppo_num_minibatch	8	4	4	8	8	8	8	8
ppo_clip_param	0.05	0.05	0.1	0.1	0.1	0.1	0.1	0.1
lr_policy	0.0007	0.0007	0.0007	0.0003	0.0007	0.0007	0.0007	0.0007
num_processes	16	16	16	16	16	8	8	8
policy_num_steps	150	50	200	1200	600	600	600	600
policy_eps	1e-08	1e-05	1e-08	1e-08	1e-08	1e-08	1e-08	1e-08
policy_value_loss_coef	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
policy_entropy_coef	0.001	0.1	0.0001	0.001	0.001	0.001	0.001	0.001
policy_gamma	0.97	0.98	0.97	0.99	0.99	0.97	0.97	0.97
policy_use_gae	True	True	True	True	True	True	True	True
policy_tau	0.9	0.95	0.9	0.9	0.9	0.9	0.9	0.9
use_proper_time_limits	True	False	True	True	True	True	True	True
vae_squash_targets	True	NaN	NaN	True	True	NaN	NaN	NaN
lr_vae	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
size_vae_buffer	10000	100000	10000	10000	10000	10000	10000	10000
precollect_len	100	5000	500	50000	5000	5000	5000	5000
vae_batch_num_trajs	15	25	10	10	10	10	10	10
tbptt_stepsize	None	None	None	50	50	None	None	None
vae_subsample_elbos	None	None	None	50	50	50	50	50
vae_subsample_decodes	None	None	None	None	50	50	50	50
vae_avg_elbo_terms	True	False	True	True	False	False	False	False
vae_avg_reconstruction_terms	False	False	False	False	False	False	False	False

	Treasure	GridWorld	CheetahDir	AntGoal	PointRobot	ML1-Reach	ML1-Push	Ml1-Pick-Place
num_vae_updates	1	1	1	10	3	1	1	3
pretrain_len	0	0	0	0	0	0	0	0
kl_weight	1.0	0.1	1.0	1.0	1.0	1.0	1.0	1.0
action_embedding_size	16	0	16	16	16	16	16	16
state_embedding_size	32	32	32	32	32	32	32	32
reward_embedding_size	16	8	16	16	16	16	16	16
encoder_layers_before_gru	[]	[]	[]	[]	[]	[]	[]	[]
encoder_gru_hidden_size	128	128	128	128	128	128	128	128
encoder_layers_after_gru	[]	[]	[]	[]	[]	[]	[]	[]
latent_dim	25	10	5	5	5	5	5	5
decode_reward	True	True	True	True	True	True	True	True
normalise_rew_targets	True	NaN	NaN	False	False	True	True	True
rew_loss_coeff	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
input_prev_state	True	False	True	True	True	True	True	True
input_action	True	False	True	True	True	True	True	True
reward_decoder_layers	[64, 32]	[64, 64]	[64, 32]	[64, 32]	[64, 32]	[64, 32]	[64, 32]	[128, 64, 32]
decode_state	True	False	False	False	False	False	False	True
state_loss_coeff	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
state_decoder_layers	[64, 32]	[32, 32]	[64, 32]	[64, 32]	[64, 32]	[128, 64, 32]	[64, 32]	[128, 64, 32]
rlloss_through_encoder	False	False	False	False	False	False	False	False
intrinsic_rew_normalise_rewards	True	True	True	True	True	True	True	True
intrinsic_rew_clip_rewards	None	10.0	None	10.0	None	10.0	10.0	10.0
rpf_weight_hyperstate	1.0	10.0	1.0	5.0	0.1	1.0	1.0	1.0
intrinsic_rew_anneal_weight	True	True	True	True	True	True	True	True
intrinsic_rew_for_vae_loss	True	True	True	True	True	True	True	True
intrinsic_weight_vae_loss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
lr_rpf	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
rpf_batch_size	128	128	128	128	128	128	128	128
rpf_update_frequency	1	1	1	3	1	1	1	50
size_rpf_buffer	10000	10000000	10000	10000000	10000	10000	10000	10000
rpf_output_dim	128	128	128	128	128	128	128	128
layers_rpf_prior	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]
layers_rpf_predictor	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]	[256, 256]
rpf_use_orthogonal_init	False	False	False	False	False	False	False	False
rpf_norm_inputs	False	NaN	False	False	False	False	False	False
rpf_init_weight_scale	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
state_expl_idx	None	None	[17]	[0, 1]	None	None	None	None