A. Supplementary Material

A.1. Exploration Environment Implementation Details

Here we describe in more detail the various implementation choices we used for our method as well as for the baselines.

Toy Chain Environment

The chain environment is implemented based on the NChain-v0 gym environment. We alter NChain-v0 to contain 40 states instead of 10 to reduce the possibility of solving the environment with random actions. We also modify the stochastic 'slipping' state behavior by fixing the behavior of the states respect to reversing an action. For both our method and *MAX*, we use ensembles of 5 deterministic neural networks with 4 layers, each is 256 units wide with tanh nonlinearities. As usual, our ensembles are sampled from the generator at each timestep, while *MAX* uses a static ensemble. We generate each layer in the target network with generators composed of two hidden layers, 64 units each with ReLU nonlinearities. Both models are trained by minimizing the regression loss on the observed data. We optimize using Adam with a learning rate of 10^{-4}, and weight decay of 10^{-6}. We use Monte Carlo Tree Search (MCTS) to find exploration policies for use in the environment. We build the tree with 25 iterations of 10 random trajectories, and UCB-1 as the selection criteria. Crucially, when building the tree, we query the dynamic models instead of the simulator, and we compute the corresponding intrinsic reward. For intrinsic rewards, *MAX* uses the Jensen Shannon divergence while our method uses the variance in the predictions within the ensemble. After building the tree we take an action in the real environment according to our selection criteria. There is a small discrepancy between the numbers reported in the MAX paper for the chain environment. This is due to using UCB-1 as the selection criteria instead of Thompson sampling as used in the *MAX*. We take actions in the environment based on the children with the highest value. The tree is then discarded after one step, after which, the dynamic models are fit for 10 additional epochs.

Continuous Control Environments

For each method where applicable, we use the method-specific hyperparameters given by the authors. Due to experimenting on potentially different environments, we search for a suitable learning rate which works the best for each method across all tasks. The common details of each exploration method are as follows. Each method uses (or samples) an ensemble of dynamic models to approximate environment dynamics. An ensemble consists of 32 networks with 4 hidden layers, 512 units wide with ReLU nonlinearities, except for *MAX* which uses swish[1]. *ICM*, *Disagreement*, and our method use ensembles of deterministic models, while *MAX* uses probabilistic networks which output a Gaussian distribution over next states. The approximate dynamic models (ensembles/generators) are optimized with Adam, using a minibatch size of 256, a learning rate of 1.0^{-4}, and weight decay of 1.0^{-5}.

For our dynamic model, each layer generator is composed of two hidden layers, 64 units wide and ReLU nonlinearity. The output dimensionality of each generator is equal to the product of the input and output dimensionality of the corresponding layer in the dynamic model. To sample one dynamic model, each generator takes as input an independent draw from $z \sim \mathcal{Z}$ where $\mathcal{Z} = \mathcal{N}(\mathbf{0}^{32}, \mathbf{1}^{32})$. We sample ensembles of a given size m by instead providing a batch $\{z\}_{i=1}^{m}$ as input. To train the generator such that we can sample accurate transition models, we update according to equation (4) in the main text; we compute the regression error on the data, as well as the repulsive term using an appropriate kernel. For all experiments we use a standard Gaussian kernel $K(f_{\theta_i}, f_{\theta_j}) = \exp\left(-d(f_{\theta_i}, f_{\theta_j})/h\right)$, where $d(f_{\theta_i}, f_{\theta_j}) = \frac{1}{n}\sum_{l=1}^{n} \|f_{\theta_i}(x_l) - f_{\theta_j}(x_l)\|_2^2$ for a training batch $\{x_l\}_{l=1}^{n}$. Where h is the median of the pairwise distances between sampled particles $\{f_\theta\}_{i=1}^{m}$. Because we sample functions $f_{\boldsymbol{\theta}}$ instead of data points, the pairwise distance is computed by using the likelihood of the data $\boldsymbol{x}$ under the model: $\log f_{\boldsymbol{\theta}}(\boldsymbol{x})$.

For *MAX*, we use the code provided from (Shyam et al., 2019)[2]. Each member in the ensemble of dynamic models is a probabilistic neural network that predicts a Gaussian distribution (with diagonal covariance) over the next state. The exploration policy is trained with SAC, given an experience buffer of rollouts $\bar{D} = \{s, a, s'\} \cup R\pi$ performed by the dynamic models, where R_π is the intrinsic reward: the Jensen-Renyi divergence between next state predictions of the dynamic models. The policy trained with SAC acts in the environment to maximize the intrinsic reward, and in doing so collects additional transitions that serve as training data for the dynamic models for the subsequent training phase.

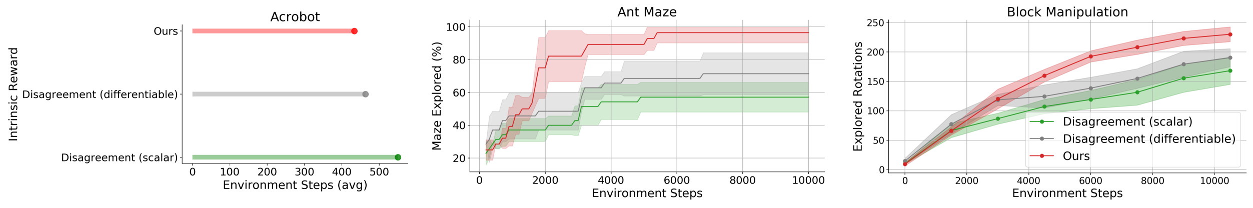
[1]Swish refers to the nonlinearity proposed by (Ramachandran et al., 2017) which is expressed as a scaled sigmoid function: $y = x + sigmoid(\beta x)$

[2]https://github.com/nnaisense/max

For *Disagreement* (Pathak et al., 2019), we implement this method under the MAX codebase, following the implementation given by the authors[3]. The intrinsic reward is formulated as the predictive variance of the dynamic models, where the models are represented by a bootstrap ensemble. In this work, we report results using two versions of this method. The proposed intrinsic reward specifically is formulated in a manner quite similar to our own, however, a fixed ensemble is used instead of a distribution for the approximate posterior. In section §4 of the main text we report results of *Disagreement* only using its intrinsic reward, instead of the full method, which makes use of a differentiable reward function and treats the reward as a supervised learning signal. We examine these methods separately because we are testing the effects of intrinsic rewards, as well as the form of the approximate dynamic model e.g. sampling vs fixed ensembles. The differentiable reward function is orthogonal to this effort. Nonetheless, in the next section §A.2 we report results using the full method of *Disagreement*, on each continuous control experiment.

A.2. Extended Disagreement Results

Here we report additional comparisons with *Disagreement* – including the original policy optimization method with a differentiable reward function (Pathak et al., 2019). We repeat our pure exploration experiments, comparing our method to both disagreement purely as an intrinsic reward, as well as the full method using the differentiable reward function for policy optimization. Figures 1(a), 1(b), and 1(c) show results on the Acrobot, Ant Maze, and Block Manipulation environments, respectively. In each figure, lines correspond to the mean of three seeds, and shaded regions denote $\pm$ one standard deviation. In each experiment, we can see that treating the intrinsic reward as a supervised loss (gray) improves on the baseline scalar-valued disagreement intrinsic reward (green). However, our method (red) remains the most sample efficient in these experiments.



(a) Acrobot (b) Ant Maze (c) HandManipulateBlock

Figure 1. Results for the full *Disagreement* method including the differentiable reward function on the Acrobot (a), Ant Maze (b), and HandManipulateBlock (c) environments.

A.3. Comparison to other exploration methods

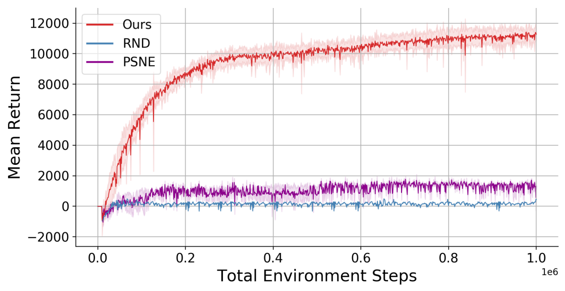
Here we compare our method to two other representative exploration methods. Parameter Space Noise for Exploration (PSNE) (Plappert et al., 2017) adds parametric noise to the weights of an agent, similar to (Fortunato et al., 2017). The noise parameters are learned by gradient descent, and the additional stochasticity in the induced policy is responsible for increased exploration ability. Random Network Distillation (RND) is another well-known method (Burda et al., 2018) that introduces a randomly initialized function $f : S \to \mathbb{R}^k$ which maps states s to a k-dimensional vector, similar to (Osband et al., 2018). A second function $\hat{f} : S \to \mathbb{R}^k$ is trained to match the predictions given by f. The prediction error $f(\hat{s}) - f(s)$ is used as an exploration bonus to the reward during training, similar to the psuedo-count based exploration bonus in (Bellemare et al., 2016). RND has been shown to be a strong baseline for both task-specific environments and pure exploration.

In Figure 2(a), we first compare our method with PSNE and RND on the HalfCheetah environment, as both can used to learn task-specific policies. For both methods, we use the author provided codes to run our experiments. Because RND is initially designed for discrete actions, we modify the policy to handle continuous action spaces. However, we were unable to recover the reported results from PSNE using the provided code[4]. In Figure 2(b), we further compare with RND in the pure exploration setting. We omit PSNE from this experiment, as PSNE does not have intrinsic reward, or another mechanism that can be directly used for pure exploration in the Ant Maze environment. For Ant Maze, each method runs

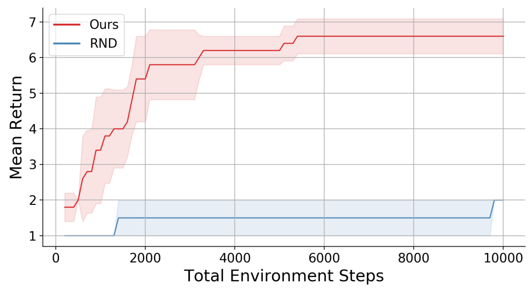
[3]https://github.com/pathak22/exploration-by-disagreement

[4]For PSNE we used the code at https://github.com/openai/baselines.

for 10k steps for pure exploration, without external reward.



(a) Baseline comparison on HalfCheetah



(b) Baseline comparison on Ant Maze

Figure 2. Comparison with RND and PSNE on HalfCheetah with external reward (a), and a pure exploration comparison with RND on Ant Maze (b).

These methods lack an explicit model of the environment dynamics. It has been shown many times that model-based methods have a considerable advantage in sample efficiency. RND in particular, takes hundreds of millions of environment steps to achieve its performance. We show in Figure 2(a) and 2(b) that our method enables superior downstream task performance, and better sample efficiency in exploration, respectively.

A.4. Policy Transfer Implementation Details

Here we describe in detail the specific settings and design choices used for the policy transfer methods and environments.

Policy Transfer with Warm-up

The exploration policies for our method, *MAX*, *ICM*, and *Disagreement* were trained exactly as in the pure exploration experiments. We trained each exploration policy for 10k steps. We then initialize a new SAC agent with the exploration policy. This agent is initialized within the HalfCheetah environment that includes the external reward. Given that the new agent has an empty replay buffer, we perform a warm-up stage to collect initial data before performing any parameter updates. We collect this initial data by rolling out the pure exploration policy for 10K steps, and storing the observed transitions in the fresh agent's replay buffer. Note that the policy is frozen during the warm-up. After this initial warm-up stage, we allow the fresh agent to train as normal for 1M steps (including the steps taken during warm-up and pure exploration), with respect to the external reward.

Policy Transfer Without Warm-up

The policy transfer experiments without the warm-up stage are similar in that the pure exploration polices are trained for 10K steps, agnostic of the downstream task, then frozen. However, instead of training a new SAC agent on transitions obtained via a warm-up stage, we only transfer the parameters of the pure exploration policy to the fresh SAC agent. Then the transition buffer is cleared, and the agent is trained in the standard setting with external reward for 1M steps (including the steps already taken during pure exploration).

Hyper-parameter Comparison

We show the hyper-parameters that we use for each pure exploration method, as well as the SAC baseline in table 1.

For the SAC baseline we use the hyper-parameters given by the authors (Haarnoja et al., 2018) for HalfCheetah. When training the pure exploration policies, we use the hyperparameters given by Shyam et al. (2019). To ensure that we are using the best set of hyper-parameters for each method, we have run baseline SAC with the hyper-parameters used in our method, but they did not perform better than the ones given by the authors. We show in table 2 that baseline SAC with and without a warm-up stage, performs best when using the hyper-parameters given by the authors, rather than those we selected for training pure exploration policies. In a similar vein, we can see that our method benefits from using the hyper-parameters given by Shyam et al. (2019). Note that when training on the task policy, we always use the original SAC v1 hyper-parameters.

	Ours	MAX	Disagreement	ICM	SAC Baseline
Learning Rate	1e-3	1e-3	1e-3	1e-3	3e-4
Batch Size	4096	4096	4096	4096	256
Alpha	0.02	0.02	0.02	0.02	1
Hidden Size	256	256	256	256	256
Gamma	0.99	0.99	0.99	0.99	0.99
Tau	5e-3	5e-3	5e-3	5e-3	5e-3
Reward Scale	1	1	1	1	5

Table 1. List of SAC Hyper-parameters used with each method for pure exploration and policy transfer experiments

	Ours	Ours (warm-up)	SAC Baseline	SAC Baseline (warm-up)
$\text{Hyperparameters}_{\text{SAC v1}}$	9701 ± 210	10999 ± 355	7273 ± 537	9363 ± 277
$\text{Hyperparameters}_{\text{exp}}$	9631 ± 559	11269 ± 395	6902 ± 696	9321 ± 677

Table 2. Comparison of hyper-parameter choices for both our method and baseline SAC, with and without a warm-up stage. We show the final performance of each method after 1M steps, over 3 trials. $\text{Hyper-parameters}_{\text{SAC v1}}$ refers to the hyper-parameters given in (Haarnoja et al., 2018) for HalfCheetah, and $\text{Hyperparameters}_{\text{exp}}$ refers to the hyper-parameters given in (Shyam et al., 2019).

References

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pp. 1471–1479, 2016.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

Ian Osband, John Aslanides, and Albin Cassirer. Randomized prior functions for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 8617–8629, 2018.

Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. Self-supervised exploration via disagreement. In *International Conference on Machine Learning*, pp. 5062–5071, 2019.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Pranav Shyam, Wojciech Jaśkowski, and Faustino Gomez. Model-based active exploration. In *International Conference on Machine Learning*, pp. 5779–5788, 2019.