Appendix

A. Details on DeepMind Control Suite Experiments

A.1. Environments

We evaluate the performance of RE3 on various tasks from DeepMind Control Suite (Tassa et al., 2020). For Hopper Hop, Quadruped Run, Cartpole Swingup Sparse, Pendulum Swingup, we use the publicly available environments without any modification. For environments which are not from the publicly available released implementation repository (https://github.com/deepmind/dm_control), we designed the tasks following Seyde et al. (2021) as below:

- **Walker/Cheetah Run Sparse**: The goal of Walker/Cheetah Run Sparse task is same as in Walker/Cheetah Run, moving forward as fast as possible, but reward is given sparsely until it reaches a certain threshold: $r = r_{\texttt{original}} \cdot \mathbb{1}_{r_{\texttt{original}} > 0.25}$, where $r_{\texttt{original}}$ is the reward in Walker/Cheetah Run from DeepMind Control Suite.

A.2. Implementation Details for Model-free RL

For all experimental results in this work, we report the results obtained by running experiments using the publicly available released implementations from the authors[2], i.e., RAD (https://github.com/MishaLaskin/rad) and DrQ (https://github.com/denisyarats/drq). We use random crop augmentation for RAD and random shift augmentation for DrQ. We provide a full list of hyperparameters in Table 1.

Implementation details for RE3. We highlight key implementation details for RE3:

- **Intrinsic reward.** We use $r^{\texttt{i}}(s_i) := ||y_i - y_i^{k\text{-NN}}||_2$ for the intrinsic reward. We got rid of $\log$ from intrinsic reward in (3) for simplicity in DeepMind Control Suite experiments, but results using $\log$ are also similar. To make the scale of intrinsic reward $r^{\texttt{i}}$ consistent across tasks, following Liu & Abbeel (2021), we normalize the intrinsic reward by dividing it by a running estimate of the standard deviation. As for newly introduced hyperparameters, we use $k = 3$, and perform hyperparameter search over $\beta_0 \in \{0.05, 0.25\}$ and $\rho \in \{0.00001, 0.000025\}$.

- **Architecture.** For all model-free RL methods, we use the same encoder architecture as in Yarats et al. (2021). Specifically, this encoder consists of 4 convolutional layers followed by ReLU activations. We employ kernels of size 3 $\times$ 3 with 32 channels for all layers, and 1 stride except of the first layer which has stride 2. The output of convolutional layers is fed into a single fully-connected layer normalized by LayerNorm (Ba et al., 2016). Finally, tanh nonlinearity is added to the 50-dimensional output of the fully-connected layer.

- **Unsupervised pre-training.** For unsupervised pre-training, we first train a policy to maximize intrinsic rewards in (3) without extrinsic rewards for 500K environment steps. For fine-tuning a policy in downstream tasks, we initialize the parameters using the pre-trained parameters and then learn a policy to maximize RE3 objective in (4) for 500K environment steps. To stabilize the initial fine-tuning phase by making the scale of intrinsic reward consistent across pre-training and fine-tuning, we load the running estimate of the standard deviation from pre-training phase.

Implementation details for representation learning baselines . We highlight key implementation details for state entropy maximization schemes that involve representation learning, i.e., contrastive learning and inverse dynamics prediction, and that employs a pre-trained ImageNet Encoder:

- **Contrastive learning.** For state entropy maximization with contrastive learning, we introduce a separate convolutional encoder with randomly initialized parameters learned to minimize contrastive loss (Srinivas et al., 2020). Note that we use the separate encoder to investigate the independent effect of representation learning. Then we compute intrinsic reward $r^{\texttt{i}}$ using representations obtained by processing observations through this separate encoder.

- **Inverse dynamics prediction.** For state entropy maximization with inverse dynamics prediction, we introduce a separate convolutional encoder with randomly initialized parameters learned to predict actions from two consecutive observations. Specifically, we introduce 2 fully-connected layers with ReLU activations to predict actions on top of encoder representations. We remark that this separate encoder is separate from RL. Then we compute intrinsic reward $r^{\texttt{i}}$ using representations obtained by processing observations through this separate encoder.

[2]We found that there are some differences from the reported results in the original papers which are due to different random seeds. We provide full source code and scripts for reproducing the main results to foster reproducibility.

- **Pre-trained ImageNet encoder.** For state entropy maximization with pre-trained ImageNet encoder, we utilize a ResNet-50 (He et al., 2016) encoder from publicly available torchvision models (https://pytorch.org/vision/0.8/models.html). To compute intrinsic reward $r^{\texttt{i}}$, we utilize representations obtained by processing observations through this pre-trained encoder.

- **Pre-trained ATC encoder.** For state entropy maximization with pre-trained ATC (Stooke et al., 2021) encoder, we utilize pre-trained encoders from the authors. Specifically, these encoders are pre-trained by contrastive learning on pre-training datasets that contain samples encountered while training a RAD agent on DeepMind Control Suite environments (see Stooke et al. (2021) for more details). We use pre-trained parameters from Walker Run, Hopper Stand, and Cheetah Run for RAD + SE w/ ATC on Walker Run Sparse, Hopper Hop, and Cheetah Run Sparse, respectively. To compute intrinsic reward $r^{\texttt{i}}$, we utilize representations obtained by processing observations through this pre-trained encoder.

Implementation details for exploration baselines. We highlight key implementation details for exploration baselines, i.e., RND (Burda et al., 2019) and ICM (Pathak et al., 2017):

- **RND.** For RND, we introduce a random encoder f_θ whose architecture is same as in Yarats et al. (2021), and introduce a predictor network g_ϕ consisting of a convolutional encoder with the same architecture and 2-layer fully connected network with 1024 units each. Then, parameters ϕ of the predictor network are trained to predict representations from a random encoder given the same observations, i.e., minimize $\epsilon = ||f_\theta(s_i) - g_\phi(s_i)||_2$. We use prediction error ϵ as an intrinsic reward and learn a policy that maximizes $r^{\texttt{total}} = r^{\texttt{e}} + \beta \cdot r^{\texttt{i}}$. We perform hyperparameter search over the weight $\beta \in \{0.05, 0.1, 1.0, 10.0\}$ and report the best result on each environment.

- **ICM.** For ICM, we introduce a convolutional encoder g_ϕ whose architecture is same as in Yarats et al. (2021), and introduce a inverse dynamics predictor h_ψ with 2 fully-connected layers with 1024 units. These networks are learned to predict actions between two consecutive observations, i.e., minimize $\mathcal{L}_{\texttt{inv}} = ||a_t - h_\psi(g_\phi(s_t), g_\phi(s_{t+1}))||_2$. We also introduce a forward dynamics predictor f_Ψ that is learned to predict the representation of next time step, i.e., minimize $\mathcal{L}_{\texttt{forward}} = \frac{1}{2}||g_\phi(s_{t+1}) - f_\Psi(g_\phi(s_t), a_t)||_2^2$. Then, we use prediction error as an intrinsic reward and learn a policy that maximizes $r^{\texttt{total}} = r^{\texttt{e}} + \beta \cdot r^{\texttt{i}}$. For joint training of the forward and inverse dynamics, following Pathak et al. (2017), we minimize $0.2 \cdot \mathcal{L}_{\texttt{forward}} + 0.8 \cdot \mathcal{L}_{\texttt{inv}}$. We perform hyperparameter search over the weight $\beta \in \{0.05, 0.1, 1.0\}$ and report the best result on each environment.

A.3. Implementation Details for Model-based RL.

For Dreamer, we use the publicly available released implementation repository (https://github.com/danijar/dreamer) from the authors[3]. We highlight key implementation details for the combination of Dreamer with RE3:

- **Intrinsic reward.** We use $r^{\texttt{i}}(s_i) := ||y_i - y_i^{k\text{-NN}}||_2$ for the intrinsic reward. We got rid of $\log$ from intrinsic reward in (3) for simplicity in DeepMind Control Suite experiments, but results using $\log$ are also similar. To make the scale of intrinsic reward $r^{\texttt{i}}$ consistent across tasks, following Liu & Abbeel (2021), we normalize the intrinsic reward by dividing it by a running estimate of the standard deviation. Since Dreamer utilizes trajectory segments for training batch, we use large value of $k = 50$ to avoid find k-NN only within a trajectory segment. We also use $\beta_0 = 0.1$, and $\rho = 0.0$, i.e., no decay schedule. We also remark that we find k-NN only within a minibatch instead of the entire buffer as in model-free RL, since the large batch size of Dreamer is enough for stable entropy estimation.

- **Architecture.** We use the same convolutional architecture as in Dreamer. Specifically, this encoder consists of 4 convolutional layers followed by ReLU activations. We employ kernels of size 4×4 with $\{32, 64, 128, 256\}$ channels, and 2 stride for all layers. To obtain low-dimensional representations, we additionally introduce a fully-connected layer normalized by LayerNorm (Ba et al., 2016). Finally, tanh nonlinearity is added to the 50-dimensional output of the fully-connected layer.

[3]We use the newer implementation of Dreamer following the suggestion of the authors, but we found that there are some difference from the reported results in the original paper which are might be due to the difference between newer implementation and the original implementation, or different random seeds. We provide full source code and scripts for reproducing the main results to foster reproducibility.

Table 1. Hyperparameters of RAD + RE3 used for DeepMind Control Suite experiments.

Hyperparameter	**Value**
Augmentation	Crop
Observation rendering	$(100, 100)$
Observation downsampling	$(84, 84)$
Replay buffer size	100000
Initial steps	10000 quadruped, run; 1000 otherwise
Stacked frames	3
Action repeat	4 quadruped, run; 2 otherwise
Learning rate (actor, critic)	0.0002
Learning rate (α)	0.001
Batch size	512
Q function EMA τ	0.01
Encoder EMA τ	0.05
Critic target update freq	2
Convolutional layers	4
Number of filters	32
Latent dimension	50
Initial temperature	0.1 RAD + RE3; 0.01 RAD + RE3 + PT
k	3
Initial intrinsic reward scale β_0	0.25 pendulum, swingup; 0.05 otherwise
Intrinsic reward decay ρ	0.000025 walker, run sparse; 0.00001 otherwise
Intrinsic reward scale (RND)	10.0 cartpole, swingup sparse 0.1 pendulum, swingup; cheetah, run sparse 0.05 otherwise
Intrinsic reward scale (ICM)	1.0 cheetah, run sparse 0.1 otherwise

Table 2. Hyperparameters of Dreamer + RE3 used for DeepMind Control Suite experiments. We only specify hyperparameters different from original paper of Hafner et al. (2020).

Hyperparameter	**Value**
Initial episodes	5
Precision[4]	32
Latent dimension of a random encoder	50
k	53
Initial reward scale β_0	0.1
Intrinsic reward decay ρ	0.0

[4]We found that precision of 32 is necessary for avoiding NaN in intrinsic reward normalization, as running estimate of standard deviation is very small.

B. Additional Experimental Results on DeepMind Control Suite

We provide additional experimental results on various tasks from DeepMind Control Suite (Tassa et al., 2020). We observe that RE3 improves sample-efficiency in several tasks, e.g., Reacher Hard and Hopper Stand, while not degrading the performance in dense-reward tasks like Cartpole Balance. This demonstrates the simple and wide applicability of RE3 to various tasks.

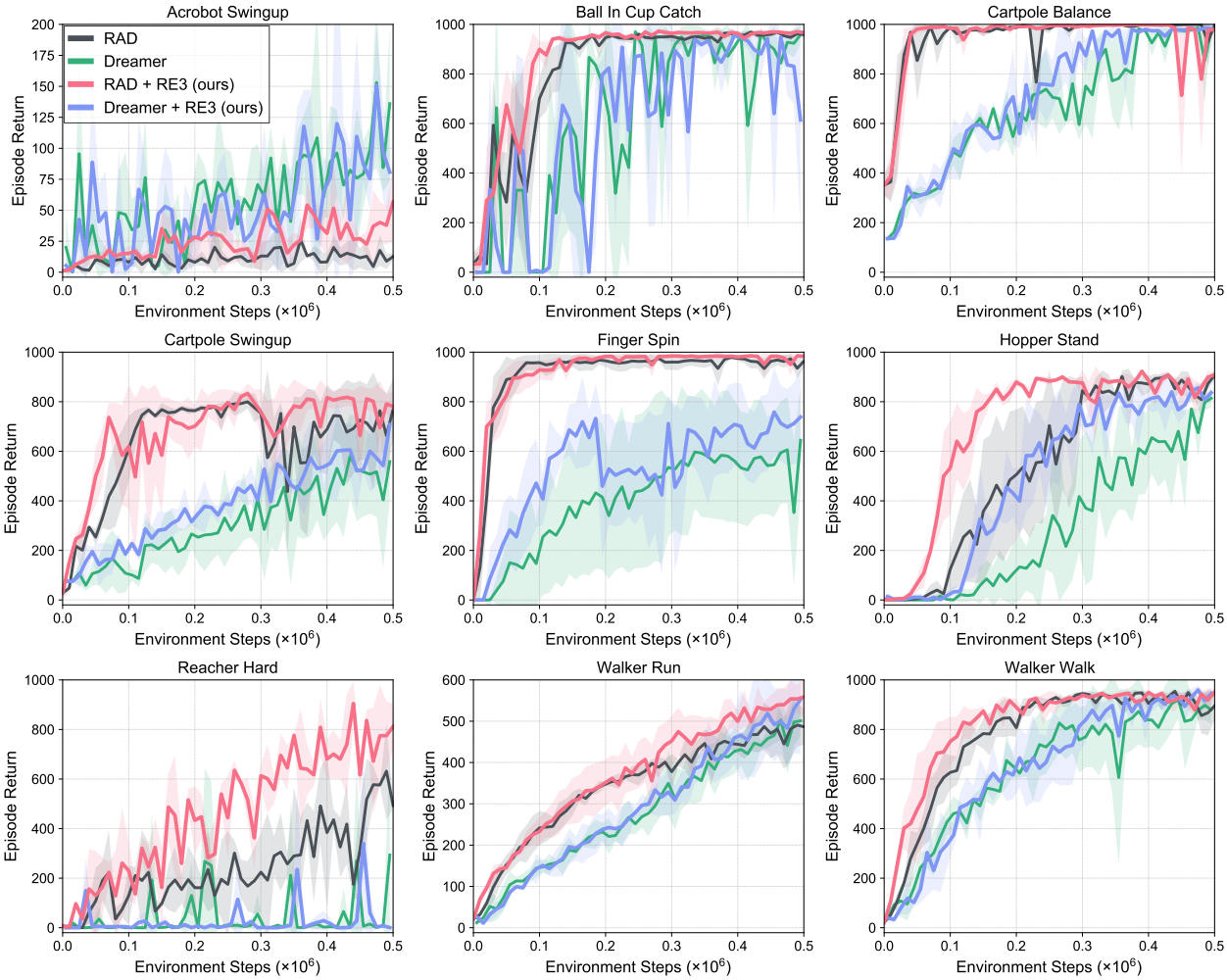

Figure 14. Performance on locomotion tasks from DeepMind Control Suite. The solid line and shaded regions represent the mean and standard deviation, respectively, across five runs.

C. Details on MiniGrid Experiments

For our A2C implementation in MiniGrid, we use the publicly available released implementation repository (https://github.com/lcswillems/rl-starter-files) and use their default hyperparameters. We provide a full list of hyperparameters that are introduced by our method or emphasized for clarity in Table 3.

We highlight some key implementation details:

- We use $r^{\texttt{i}}(s_i) := \log(||y_i - y_i^{k\text{-NN}}||_2 + 1)$ for the intrinsic reward. The additional 1 is for numerical stability.
- We use the average distance between y_i and its k nearest neighbors (i.e., $y_i^{2\text{-NN}}, \cdots, y_i^{k\text{-NN}}$) for the intrinsic reward, instead of the single k nearest neighbor. This provides a less noisy state entropy estimate and empirically improves performance in MiniGrid environments.
- We perform hyperparameter search over $\beta \in \{0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1\}$ for RE3, ICM, and RND and report the best result.
- We do not change the network architecture from the above publicly available implementation. We use the same encoder architecture as the RL encoder for state entropy maximization. This encoder architecture consists of 3 convolutional layers with kernel 2, stride 1, and padding 0, each followed by a ReLU layer. The convolutional layers have 16, 32, and 64 filters respectively. The first ReLU is followed by a two-dimensional max pooling layer with kernel 2. The actor and critic MLPs both contain two fully-connected layers with hidden dimension 64, with a tanh operation in between.

Table 3. Hyperparameters used for MiniGrid experiments. Most hyperparameter values are unchanged across environments with the exception of evaluation frequency and intrinsic reward weight β.

Hyperparameter	**Value**
Input Size	$(7, 7, 3)$
Replay buffer size (for RE3 intrinsic reward)	10000
Stacked frames	1
Action repeat	1
Evaluation episodes	100
Optimizer	RMSprop
k	3
Evaluation frequency	6400 Empty-16x16; 12800 DoorKey-6x6 64000 DoorKey-8x8
Intrinsic reward weight β in Empty-16x16 experiments	0.1 A2C + RE3 + PT 0.1 A2C + RE3 0.00001 A2C + ICM 0.00005 A2C + RND
Intrinsic reward weight β in DoorKey-6x6 experiments	0.05 A2C + RE3 + PT 0.005 A2C + RE3 0.0001 A2C + ICM 0.0001 A2C + RND
Intrinsic reward weight β in DoorKey-8x8 experiments	0.05 A2C + RE3 + PT 0.01 A2C + RE3 0.001 A2C + ICM 0.00005 A2C + RND
Intrinsic reward decay ρ	0
Number of processes	16
Frames per process	5
Discount γ	0.99
GAE λ	0.95
Entropy coefficient	0.01
Value loss term coefficient	0.5
Maximum norm of gradient	0.5
RMSprop ϵ	0.01
Clipping ϵ	0.2
Recurrence	None

D. Details on Atari Experiments

For our Rainbow implementation in Atari games, we use the publicly available implementation repository (https://github.com/Kaixhin/Rainbow) and use their default hyperparameters.

We highlight some key implementation details:

- We use $r^{\texttt{i}}(s_i) := \log(||y_i - y_i^{k\text{-NN}}||_2 + 1)$ for the intrinsic reward. The additional 1 is for numerical stability.
- We perform hyperparameter search over $\beta \in \{0.0001, 0.001, 0.01\}$ for Rainbow + SE w/ RE3 and report the best result for each environment. We used 0.01 for Asterix and BeamRider, 0.001 for Seaquest, and 0.0001 for other games.
- We perform hyperparameter search over $\beta \in \{0.0001, 0.001\}$ for Rainbow + SE w/ Contrastive and report the best result for each environment. Specifically, we used 0.001 for Seaquest, 0.0001 for other games.
- We do not change the network architecture from the above publicly available implementation. We use the same encoder architecture as the RL encoder for state entropy maximization, with additional linear layer to reduce the dimension of latent representations. This encoder consists of 3 convolutional layers with kernels $\{8, 4, 3\}$, strides $\{4, 4, 3\}$, and padding 0, each followed by a ReLU layer. The convolutional layers have 32, 32, 64 filters, respectively. Then the outputs from a convolutional encoder is flattened and followed by a linear layer of size $\{50, 150\}$. We find that large dimension of 150 is effective for Montezuma's Revenge, but 50 is enough for all other environments.

E. Additional Experimental Results on Atari Games

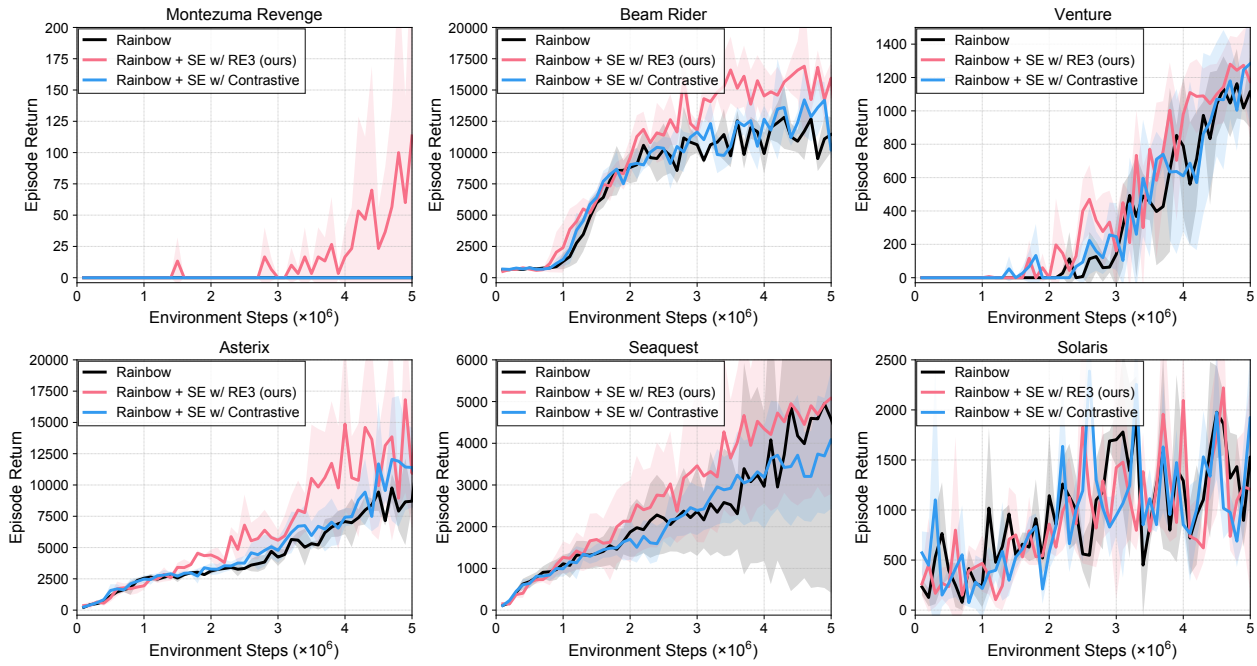

Figure 15. Performance on Atari games from Arcade Learning Environment. The solid line and shaded regions represent the mean and standard deviation, respectively, across three runs.

F. Calculation of Floating Point Operations

We explain our FLOP counting procedure for comparing the compute-efficiency of RAD, RAD + SE w/ Random (ours), RAD + SE w/ Contrastive, and RAD + SE w/ Inverse dynamics in Figure 7. We consider each backward pass to require twice as many FLOPs as a forward pass, as done in https://openai.com/blog/ai-and-compute/. Each weight requires one multiply-add operation in the forward pass. In the backward pass, it requires two multiply-add operations: at layer i, the gradient of the loss with respect to the weight at layer i and with respect to the output of layer $(i-1)$ need to be computed. The latter computation is necessary for subsequent gradient calculations for weights at layer $(i-1)$.

We use functions from Huang et al. (2018) and Jeong & Shin (2019) to obtain the number of operations per forward pass for all layers in the encoder (denoted E) and number of operations per forward pass for all MLP layers (denoted M).

We assume that (1) the number of updates per iteration is 1, (2) the architecture of the encoder used for state entropy estimation is the same as the RL encoder used in RAD, and (3) the FLOPs required for computations (e.g., finding the k-NNs in representation space) that are not forward and backward passes through neural network layers is negligible.[5]

We denote the number of forward passes per training update F, the number of backward passes per training update B, and the batch size b (in our experiments $b = 512$). Then, the number of FLOPs per iteration of RAD is:

$$bF(E+M) + 2bB(E+M) + (E+M),$$

where the last term is for the single forward pass required to compute the policy action.

Specifically, RAD + SE w/ Random only requires E extra FLOPs per iteration to store the fixed representation from the random encoder in the replay buffer for future k-NN calculations. In comparison, RAD + SE w/ Contrastive requires $4bE$ extra FLOPs per iteration, and RAD + SE w/ Inverse dynamics requires more than $6bE$ extra FLOPs.

[5] Letting the dimension of y be d, batch size m, computing distances between y_i and all entries $y \in \mathcal{B}$ over 250000 training steps requires $\sum_{n=1000}^{250000}(d(2m+2\cdot\min(n,|\mathcal{B}|)+3m\cdot\min(n,|\mathcal{B}|))+2m\cdot\min(n,|\mathcal{B}|))$, which is 1.569e+15 FLOPs in our case of $m = 512$, $|\mathcal{B}| = 100000$, and $d = 50$.

G. Comparison to State Entropy Maximization with Contrastive Encoder for MiniGrid Pre-training

We show a comparison to state entropy maximization with contrastive learning (i.e., A2C + SE w/ Contrastive + PT) in Figure 16. In the results shown, we do not use state entropy intrinsic reward during the fine-tuning phase for A2C + SE w/ Contrastive + PT, following the setup of Liu & Abbeel (2021), but found that using intrinsic reward during fine-tuning results in very similar performance. As discussed in Section 4.2, we observe that the contrastive encoder does not work well for state entropy estimation, as contrastive learning depends on data augmentation specific to images (e.g., random shift, random crop, and color jitter), which are not compatible with the compact embeddings used as inputs for MiniGrid. We re-mark that RE3 eliminates the need for carefully chosen data augmentations by employing a random encoder.

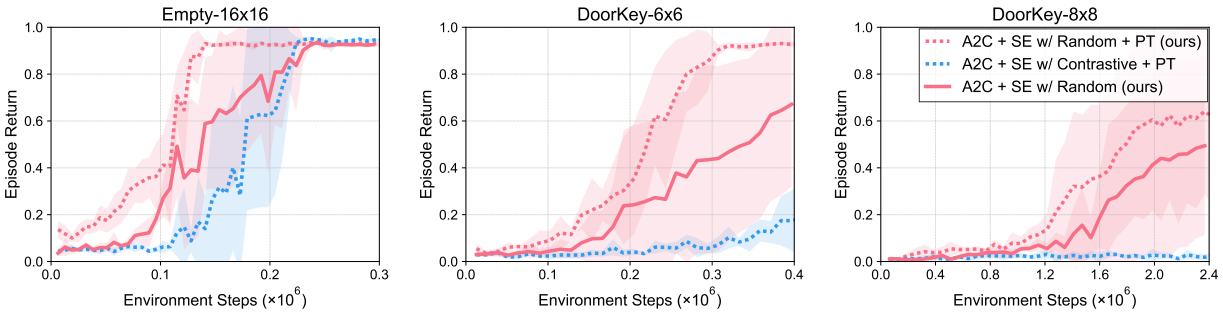

Figure 16. Performance on navigation tasks from MiniGrid. We find that pre-training using state entropy (SE) with a random encoder (ours) outperforms pre-training using state entropy with a contrastive encoder (Liu & Abbeel, 2021), using random shift as the data augmentation. The solid (or dotted) line and shaded regions represent the mean and standard deviation, respectively, across five runs.

H. RE3 with On-policy Reinforcement Learning

We provide the full procedure for RE3 with on-policy RL in Algorithm 2.

Algorithm 2 RE3: On-policy RL version

Initialize parameters of random encoder θ, policy ϕ
Initialize replay buffer $\mathcal{B} \leftarrow \emptyset$, step counter $t \leftarrow 0$
repeat
  // COLLECT TRANSITIONS
  $t_{\texttt{start}} \leftarrow t$
  repeat
    Collect a transition $\tau_t = (s_t, a_t, s_{t+1}, r_t^{\texttt{e}})$ from the interaction with the environment using policy π_ϕ
    Get a fixed representation $y_t = f_\theta(s_t)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\tau_t, y_t)\}$
    $t \leftarrow t + 1$
  until terminal s_t or $t - t_{\texttt{start}} = t_{\texttt{max}}$
  // COMPUTE INTRINSIC REWARD
  for $j = t - 1$ **to** $t_{\texttt{start}}$ **do**
    Compute the distance $||y_j - y||_2$ for all representations $y \in \mathcal{B}$ and find the k-nearest neighbor $y_j^{k\text{-NN}}$
    Compute $r_j^{\texttt{i}} \leftarrow \log(||y_j - y_j^{k\text{-NN}}||_2 + 1)$
    Let $r_j^{\texttt{total}} \leftarrow r_j^{\texttt{e}} + \beta \cdot r_j^{\texttt{i}}$
  end for
  // UPDATE POLICY
  Update ϕ with transitions $\{(s_j, a_j, s_{j+1}, r_j^{\texttt{total}})\}_{j=t_{\texttt{start}}}^{t-1}$
until $t > t_{\texttt{max}}$