Appendix

A Source Code

We provide source code in the supplementary materials.

B Algorithm

We detail the specifics of modifying off-policy RL methods with SEER in Algorithm 1. For concreteness, we describe SEER combined with deep Q-learning methods.

Algorithm 1 Stored Embeddings for Efficient Reinforcement Learning (DQN Base Agent)

Initialize replay buffer $\mathcal{B}$ with capacity C
Initialize action-value network Q with parameters θ and encoder f with parameters ψ
for each timestep t **do**
    Select action: $a_t \leftarrow \text{argmax}_a \, Q_\theta(f_\psi(o_t), a)$
    Collect observation o_{t+1} and reward r_t from the environment by taking action a_t
    if $t \leq T_f$ **then**
        Store transition (o_t, a_t, o_{t+1}, r_t) in replay buffer $\mathcal{B}$
    else
        Compute latent states $z_t, z_{t+1} \leftarrow f_\psi(o_t), f_\psi(o_{t+1})$
        Store transition (z_t, a_t, z_{t+1}, r_t) in replay buffer $\mathcal{B}$
    end if
    // REPLACE PIXEL-BASED TRANSITIONS WITH LATENT TRAJECTORIES
    if $t = T_f$ **then**
        Compute latent states $\{(z_t, z_{t+1})\}_{t=1}^{\min(T_f,c)} \leftarrow \{(f_\psi(o_t), f_\psi(o_{t+1}))\}_{t=1}^{\min(T_f,c)}$
        Replace $\{(o_t, a_t, o_{t+1}, r_t)\}_{t=1}^{\min(T_f,c)}$ with latent transitions $\{(z_t, a_t, z_{t+1}, r_t)\}_{t=1}^{\min(T_f,c)}$
        Increase the capacity of $\mathcal{B}$ to $\widehat{C}$
    end if
    // UPDATE PARAMETERS OF Q-NETWORK WITH SAMPLED IMAGES OR LATENTS
    for each gradient step **do**
        if $t < T_f$ **then**
            Sample random minibatch $\{(o_j, a_j, o_{j+1}, r_j)\}_{j=1}^{b} \sim \mathcal{B}$
            Calculate target $y_j = r_j + \gamma \max_{a'} Q_{\bar{\theta}}(f_{\bar{\psi}}(o_{j+1}), a')$
            Perform a gradient step on $\mathcal{L}^{\texttt{DQN}}(\theta, \psi)$
        else
            Sample random minibatch $\{(z_j, a_j, z_{j+1}, r_j)\}_{j=1}^{b} \sim \mathcal{B}$
            Calculate target $y_j = r_j + \gamma \max_{a'} Q_{\bar{\theta}}(z_{j+1}, a')$
            Perform a gradient step on $\mathcal{L}^{\texttt{DQN}}(\theta)$
        end if
    end for
end for

C Calculation of Floating Point Operations

We consider each backward pass to require twice as many FLOPs as a forward pass. [3] 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 $(i-1)$.

[3] This method for FLOP calculation is used in https://openai.com/blog/ai-and-compute/.

We use functions from Huang et al. [17] and Jeong & Shin [20] 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 denote the number of forward passes per training update F, the number of backward passes per training update B, and the batch size b. We assume the number of updates per timestep is 1. Then, the number of FLOPs per iteration before freezing at time $t = T_f$ 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. For the baseline, FLOPs are computed using this formula throughout training.

SEER reduces computational overhead by eliminating most of the encoder forward and backward passes. The number of FLOPs per iteration after freezing is:

$$bFM + 2bBM + (E + M) + EKN,$$

where K is the number of data augmentations and N is the number of networks as described in Section 4.2. The forward and backward passes of the encoder for training updates are removed, with the exception of the forward pass for computing the policy action and the EKN term at the end that arises from calculating latent vectors for the current observation.

At freezing time $t = T_f$, we need to compute latent vectors for each transition in the replay buffer. This introduces a one-time cost of $(EKN \min(T_f, C))$ FLOPs, since the number of transitions in the replay buffer is $\min(T_f, C)$, where C is the initial replay capacity.

D Discussions on Constrained-Memory Experiments

We acknowledge that the memory efficiency advantage of SEER is conditioned on the assumption that a larger replay buffer capacity would improve performance. While the replay buffer capacity used in DM Contorl and Atari benchmarks is typically large enough to achieve strong performance, there are many real-world scenarios where memory may be limited, such as training on small devices (e.g., on the scale of mobile phones, drones, Raspberry Pi's). Our constrained-memory experiments aim to show the potential of SEER to improve performance in scenarios such as these. As a side note, another potential benefit of reduced memory requirements is the ability to store the replay buffer in GPU and reduce expensive CPU to GPU transfers, allowing for fast data reads, which would be interesting future work.

E Wall-Clock Time

Given our computational constraints, it is difficult to accurately measure wall-clock time and we did not run all agents on the same machine without other jobs running. To give a rough idea of wall-clock time, Figure 10 shows learning curves for Amidar where the x-axis shows wall-clock time. Since wall-clock time takes into account computational costs besides neural network training (e.g., interacting with the environment in the simulator), the gains are less noticeable, but Rainbow + SEER is still more compute-efficient than Rainbow. We remark that this is a very imperfect estimate of wall-clock time, due to our computational constraints.

F Transfer Setting Analysis

In Figure 7a we show the computational efficiency of SEER on Walker-walk with Walker-stand pretrained for 60K steps, with four convolutional layers frozen. We provide analysis for the number of layers frozen and number of environment interactions before freezing T_f in Figure 11. We find that freezing more layers allows for more computational gain, since we can avoid computing gradients for the frozen layers without sacrificing performance. Longer pretraining in the source task improves compute-efficiency in the target task; however, early convergence of encoder parameters enables the agent to learn a good policy even with only 20K interactions before transfer.

We remark that Yosinski et al. [55] examine the generality of features learned by neural networks and the feasibility of transferring parameters between similar image classification tasks. Yarats et al.

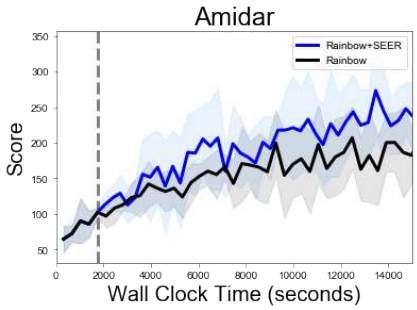
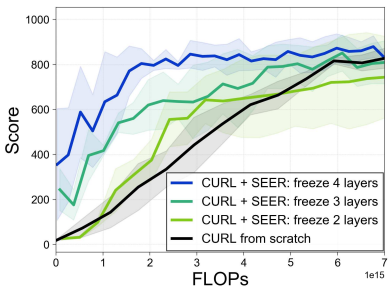
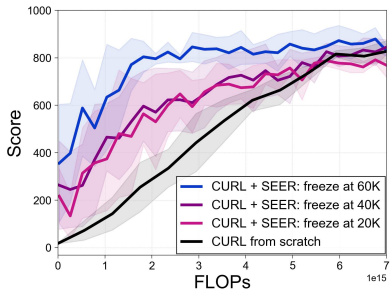


Figure 10: Learning curves for Rainbow with and without SEER in Amidar, where the x-axis shows wall-clock time. The dotted gray line denotes the encoder freezing time $t = T_f$. The solid line and shaded regions represent the mean and standard deviation, respectively, across five runs.



(a) Number of frozen layers.



(b) Freezing time hyperparameter T_f.

Figure 11: (a) Analysis on the number of frozen convolutional layers in Walker-walk training from Walker-stand pretrained for 60K steps. (b) Analysis on the number of environment steps Walker-stand agent is pretrained prior to Walker-walk transfer, where the first four convolutional layers are frozen.

[54] show that transferring encoder parameters pretrained from Walker-walk to Walker-stand and Walker-run can improve the performance and sample-efficiency of a SAC agent. For the first time, we show that encoder parameters trained on simple tasks can be useful for compute-efficient training in complex tasks and new domains.

G Compute-Efficiency in Constrained-Memory Settings

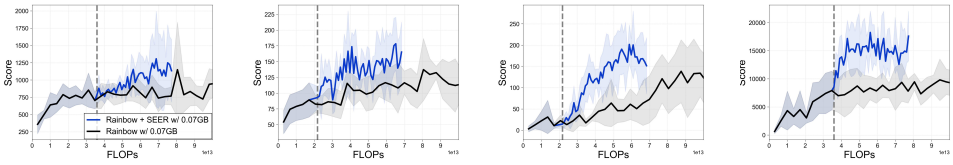
In our main experiments, we isolate the two major contributions of our method, reduced computational overhead and improved sample-efficiency in constrained-memory settings. In Figure 12 we show that these benefits can also be combined for significant computational gain in constrained-memory settings.

H Sample-Efficiency Plots

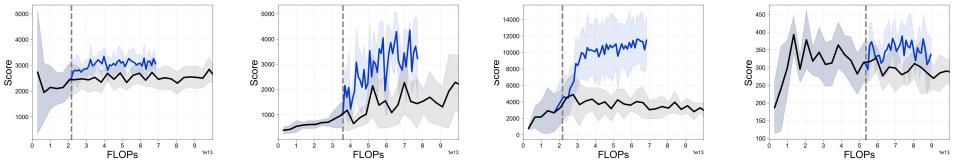
In section 5.2 we show the compute-efficiency of our method in DMControl and Atari environments. We show in Figure 13 that our sample-efficiency is very close to that of baseline CURL [41], with only slight degradation in Cartpole-swingup and Walker-walk. In Atari games (Figure 14), we match the sample-efficiency of baseline Rainbow [15] very closely, with no degradation.

I General Implementation Details

SEER can be applied to any convolutional encoder which compresses the input observation into a latent vector with smaller dimension than the observation. We generally freeze all the convolutional

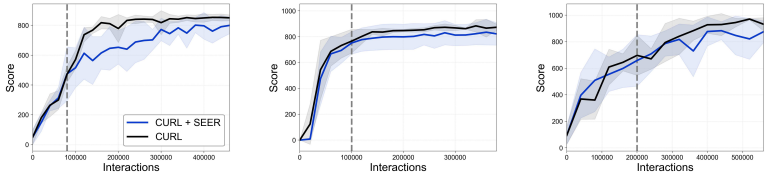


(a) Alien (b) Amidar (c) BankHeist (d) CrazyClimber

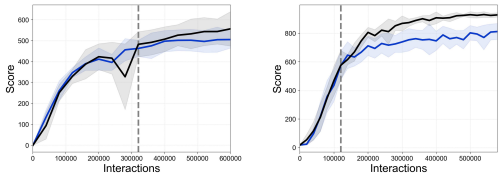


(e) Krull (f) Qbert (g) RoadRunner (h) Seaquest

Figure 12: Comparison of Rainbow in constrained-memory settings with and without SEER, where the x-axis shows estimated cumulative FLOPs, corresponding to Figure 4. The dotted gray line denotes the encoder freezing time $t = T_f$. The solid line and shaded regions represent the mean and standard deviation, respectively, across five runs.



(a) Cartpole-swingup (b) Finger-spin (c) Reacher-easy



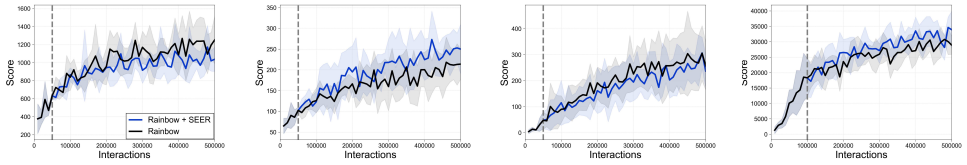
(d) Cheetah-run (e) Walker-walk

Figure 13: Comparison of the sample-efficiency of CURL with and without SEER, corresponding to Figure 2. The dotted gray line denotes the encoder freezing time $t = T_f$. The solid line and shaded regions represent the mean and standard deviation, respectively, across five runs.

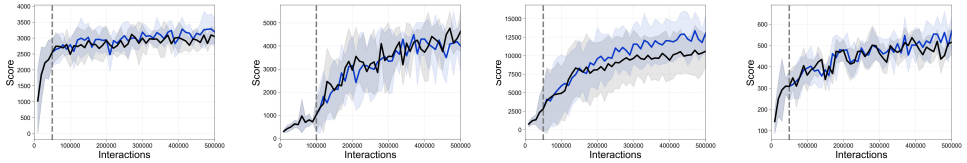
layers and possibly the first fully-connected layer. In our main experiments, we chose to freeze the first fully-connected layer for DM Control experiments and the last convolutional layer for Atari experiments. We made this choice in order to simultaneously save computation and memory; for those architectures, if we freeze an earlier layer, we save less computation, and the latent vectors (convolutional features) are too large for our method to save memory. In DM Control experiments, the latent dimension of the first fully-connected layer is 50, which allows a roughly 12X memory gain. In Atari experiments, the latent dimension of the last convolutional layer is 576, which allows a roughly 3X memory gain.

J Freezing Time Ablation

The general trend for the freezing time hyperparameter T_f is that freezing time around $T_f = 100000$ usually works well in Atari, and in our experiments, $T_f \in \{50000, 100000, 150000\}$ produce similar

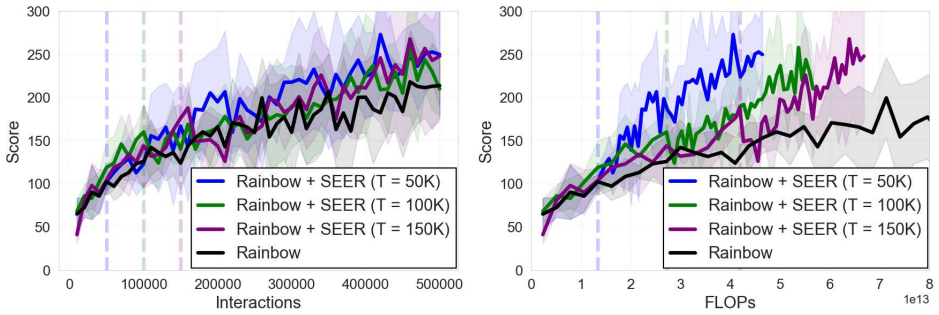


(a) Alien (b) Amidar (c) BankHeist (d) CrazyClimber



(e) Krull (f) Qbert (g) RoadRunner (h) Seaquest

Figure 14: Comparison of the sample-efficiency of Rainbow with and without SEER, corresponding to Figure 3. The dotted gray line denotes the encoder freezing time $t = T_f$. The solid line and shaded regions represent the mean and standard deviation, respectively, across five runs.



(a) Sample-efficiency in Amidar (b) Compute-efficiency in Amidar

Figure 15: (a) Rainbow + SEER with freezing at $T_f \in \{50000, 100000, 150000\}$, and no freezing (Rainbow) all result in similar sample-efficiency. This demonstrates that SEER is not extremely sensitive to freezing time. (b) Looking at compute-efficiency (with x-axis showing FLOPs), freezing earlier generally produces more compute-efficiency gains, but freezing at $T_f = 150000$ still results in better compute-efficiency than baseline Rainbow.

results so it is not particularly sensitive to freezing time (see Figure 15 for learning curves for Amidar with $T_f \in \{50000, 100000, 150000\}$). In DM Control you need to do per-environment hyperparameter tuning since the tasks are more varied.

K DMControl Implementation details

We use the network architecture in https://github.com/MishaLaskin/curl for our CURL [41] implementation. We show a full list of hyperparameters in Table 2.

L Atari Implementation details

We use the network architecture in https://github.com/Kaixhin/Rainbow for our Rainbow [15] implementation and the data-efficient Rainbow [50] encoder architecture and hyperparameters. We show a full list of hyperparameters in Table 3.

Table 2: Hyperparameters used for DMControl experiments. Most hyperparameter values are unchanged across environments with the exception of initial replay buffer size, action repeat, and learning rate.

Hyperparameter	**Value**
Augmentation	Crop
Observation rendering	$(100, 100)$
Observation down/upsampling	$(84, 84)$
Replay buffer size in Figure 2	Number of training steps
Initial replay buffer size in Figure 5	1000 cartpole, swingup; cheetah, run; finger, spin
	2000 reacher, easy; walker, walk
Number of updates per training step	1
Initial steps	1000
Stacked frames	3
Action repeat	2 finger, spin; walker, walk
	4 cheetah, run; reacher, easy
	8 cartpole, swingup
Hidden units (MLP)	1024
Evaluation episodes	10
Evaluation frequency	2500 cartpole, swingup
	10000 cheetah, run; finger, spin; reacher, easy; walker, walk
Optimizer	Adam
$(\beta_1, \beta_2) \rightarrow (f_\psi, \pi_\phi, Q_\theta)$	$(.9, .999)$
$(\beta_1, \beta_2) \rightarrow (\alpha)$	$(.5, .999)$
Learning rate $(f_\psi, \pi_\phi, Q_\theta)$	$2e-4$ cheetah, run
	$1e-3$ cartpole, swingup; finger, spin; reacher, easy; walker, walk
Learning rate (α)	$1e-4$
Batch Size	512 cheetah, run
	128 cartpole, swingup; finger, spin; reacher, easy; walker, walk
Q function EMA τ	0.01
Critic target update freq	2
Convolutional layers	4
Number of filters	32
Non-linearity	ReLU
Encoder EMA τ	0.05
Latent dimension	50
Discount γ	.99
Initial temperature	0.1
Freezing time T_f in Figure 2	10000 cartpole, swingup
	50000 finger, spin; reacher, easy
	60000 walker, walk
	80000 cheetah, run
Freezing time T_f in Figure 5	10000 cartpole, swingup
	50000 finger, spin
	30000 reacher, easy
	80000 cheetah, run; walker, walk

Table 3: Hyperparameters used for Atari experiments. All hyperparameter values are unchanged across environments with the exception of encoder freezing time.

Hyperparameter	**Value**
Augmentation	None
Observation rendering	$(84, 84)$
Replay buffer size in Figure 3	Number of training steps
Initial replay buffer size in Figure 4	10000
Number of updates per training step	1
Initial steps	1600
Stacked frames	4
Action repeat	1
Hidden units (MLP)	256
Evaluation episodes	10
Evaluation frequency	10000
Optimizer	Adam
$(\beta_1, \beta_2) \rightarrow (f_\psi, Q_\theta)$	$(.9, .999)$
Learning rate (f_ψ, Q_θ)	$1e-3$
Learning rate (α)	0.0001
Batch Size	32
Multi-step returns length	20
Critic target update freq	2000
Convolutional layers	2
Number of filters	$32, 64$
Non-linearity	ReLU
Discount γ	.99
Freezing time T_f in Figure 3	50000 Alien; Amidar; BankHeist; Krull; RoadRunner; Seaquest 100000 CrazyClimber; Qbert
Freezing time T_f in Figure 4	50000 Amidar; BankHeist; Krull; RoadRunner 100000 Alien; CrazyClimber; Qbert 150000 Seaquest