APPENDIX

A EXTENDED BACKGROUND

Reinforcement Learning from Images We formulate image-based control as an infinite-horizon partially observable Markov decision process (POMDP) (Bellman, 1957; Kaelbling et al., 1998). An POMDP can be described as the tuple $(\mathcal{O}, \mathcal{A}, p, r, \gamma)$, where $\mathcal{O}$ is the high-dimensional observation space (image pixels), $\mathcal{A}$ is the action space, the transition dynamics $p = Pr(o'_t|o_{\leq t}, a_t)$ capture the probability distribution over the next observation o'_t given the history of previous observations $o_{\leq t}$ and current action a_t, $r : \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ is the reward function that maps the current observation and action to a reward $r_t = r(o_{\leq t}, a_t)$, and $\gamma \in [0, 1)$ is a discount factor. Per common practice (Mnih et al., 2013), throughout the paper the POMDP is converted into an MDP (Bellman, 1957) by stacking several consecutive image observations into a state $s_t = \{o_t, o_{t-1}, o_{t-2}, \ldots\}$. For simplicity we redefine the transition dynamics $p = Pr(s'_t|s_t, a_t)$ and the reward function $r_t = r(s_t, a_t)$. We then aim to find a policy $\pi(a_t|s_t)$ that maximizes the cumulative discounted return $\mathbb{E}_\pi[\sum_{t=1}^{\infty} \gamma^t r_t | a_t \sim \pi(\cdot|s_t), s'_t \sim p(\cdot|s_t, a_t), s_1 \sim p(\cdot)]$.

Soft Actor-Critic The Soft Actor-Critic (SAC) (Haarnoja et al., 2018) learns a state-action value function Q_θ, a stochastic policy π_θ and a temperature α to find an optimal policy for an MDP $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ by optimizing a γ-discounted maximum-entropy objective (Ziebart et al., 2008). θ is used generically to denote the parameters updated through training in each part of the model. The actor policy $\pi_\theta(a_t|s_t)$ is a parametric $\tanh$-Gaussian that given s_t samples $a_t = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t)\epsilon)$, where $\epsilon \sim \mathcal{N}(0, 1)$ and μ_θ and σ_θ are parametric mean and standard deviation.

The policy evaluation step learns the critic $Q_\theta(s_t, a_t)$ network by optimizing a single-step of the soft Bellman residual

$$J_Q(\mathcal{D}) = \mathbb{E}_{\substack{(s_t, a_t, s'_t) \sim \mathcal{D} \\ a'_t \sim \pi(\cdot|s'_t)}}[(Q_\theta(s_t, a_t) - y_t)^2]$$
$$y_t = r(s_t, a_t) + \gamma[Q_{\theta'}(s'_t, a'_t) - \alpha \log \pi_\theta(a'_t|s'_t)],$$

where $\mathcal{D}$ is a replay buffer of transitions, θ' is an exponential moving average of the weights as done in (Lillicrap et al., 2015). SAC uses clipped double-Q learning (van Hasselt et al., 2015; Fujimoto et al., 2018), which we omit from our notation for simplicity but employ in practice.

The policy improvement step then fits the actor policy $\pi_\theta(a_t|s_t)$ network by optimizing the objective

$$J_\pi(\mathcal{D}) = \mathbb{E}_{s_t \sim \mathcal{D}}[D_{\mathrm{KL}}(\pi_\theta(\cdot|s_t) || \exp\{\frac{1}{\alpha} Q_\theta(s_t, \cdot)\})].$$

Finally, the temperature α is learned with the loss

$$J_\alpha(\mathcal{D}) = \mathbb{E}_{\substack{s_t \sim \mathcal{D} \\ a_t \sim \pi_\theta(\cdot|s_t)}}[-\alpha \log \pi_\theta(a_t|s_t) - \alpha \bar{\mathcal{H}}],$$

where $\bar{\mathcal{H}} \in \mathbb{R}$ is the target entropy hyper-parameter that the policy tries to match, which in practice is usually set to $\bar{\mathcal{H}} = -|\mathcal{A}|$.

Deep Q-learning DQN (Mnih et al., 2013) also learns a convolutional neural net to approximate Q-function over states and actions. The main difference is that DQN operates on discrete actions spaces, thus the policy can be directly inferred from Q-values. The parameters of DQN are updated by optimizing the squared residual error

$$J_Q(\mathcal{D}) = \mathbb{E}_{(s_t, a_t, s'_t) \sim \mathcal{D}}[(Q_\theta(s_t, a_t) - y_t)^2]$$
$$y_t = r(s_t, a_t) + \gamma \max_{a'} Q_{\theta'}(s'_t, a').$$

In practice, the standard version of DQN is frequently combined with a set of tricks that improve performance and training stability, wildly known as Rainbow (van Hasselt et al., 2015).

B THE DEEPMIND CONTROL SUITE EXPERIMENTS SETUP

Our PyTorch SAC (Haarnoja et al., 2018) implementation is based off of Yarats & Kostrikov (2020).

B.1 ACTOR AND CRITIC NETWORKS

We employ clipped double Q-learning (van Hasselt et al., 2015; Fujimoto et al., 2018) for the critic, where each Q-function is parametrized as a 3-layer MLP with `ReLU` activations after each layer except of the last. The actor is also a 3-layer MLP with `ReLU`s that outputs mean and covariance for the diagonal Gaussian that represents the policy. The hidden dimension is set to 1024 for both the critic and actor.

B.2 ENCODER NETWORK

We employ an encoder architecture from Yarats et al. (2019). This encoder consists of four convolutional layers with 3×3 kernels and 32 channels. The `ReLU` activation is applied after each conv layer. We use stride to 1 everywhere, except of the first conv layer, which has stride 2. The output of the convnet is feed into a single fully-connected layer normalized by `LayerNorm` (Ba et al., 2016). Finally, we apply `tanh` nonlinearity to the 50 dimensional output of the fully-connected layer. We initialize the weight matrix of fully-connected and convolutional layers with the orthogonal initialization (Saxe et al., 2013) and set the bias to be zero.

The actor and critic networks both have separate encoders, although we share the weights of the conv layers between them. Furthermore, only the critic optimizer is allowed to update these weights (e.g. we stop the gradients from the actor before they propagate to the shared conv layers).

B.3 TRAINING AND EVALUATION SETUP

Our agent first collects 1000 seed observations using a random policy. The further training observations are collected by sampling actions from the current policy. We perform one training update every time we receive a new observation. In cases where we use action repeat, the number of training observations is only a fraction of the environment steps (e.g. a 1000 steps episode at action repeat 4 will only results into 250 training observations). We evaluate our agent every 10000 true environment steps by computing the average episode return over 10 evaluation episodes. During evaluation we take the mean policy action instead of sampling.

B.4 PLANET AND DREAMER BENCHMARKS

We consider two evaluation setups that were introduced in PlaNet (Hafner et al., 2018) and Dreamer (Hafner et al., 2019), both using tasks from the DeepMind control suite (Tassa et al., 2018). The PlaNet benchmark consists of six tasks of various traits. Importantly, the benchmark proposed to use a different action repeat hyper-parameter for each task, which we summarize in Table 2.

The Dreamer benchmark considers an extended set of tasks, which makes it more difficult that the PlaNet setup. Additionally, this benchmark requires to use the same set hyper-parameters for each task, including action repeat (set to 2), which further increases the difficulty.

Table 2: The action repeat hyper-parameter used for each task in the PlaNet benchmark.

Task name	Action repeat
Cartpole Swingup	8
Reacher Easy	4
Cheetah Run	4
Finger Spin	2
Ball In Cup Catch	4
Walker Walk	2

B.5 PIXELS PREPROCESSING

We construct an observational input as an 3-stack of consecutive frames (Mnih et al., 2013), where each frame is a RGB rendering of size 84×84 from the 0th camera. We then divide each pixel by 255 to scale it down to $[0, 1]$ range.

B.6 OTHER HYPER PARAMETERS

Due to computational constraints for all the continuous control ablation experiments in the main paper and appendix we use a minibatch size of 128, while for the main results we use minibatch of size 512. In Table 3 we provide a comprehensive overview of all the other hyper-parameters.

Table 3: An overview of used hyper-parameters in the DeepMind control suite experiments.

Parameter	Setting
Replay buffer capacity	100000
Seed steps	1000
Ablations minibatch size	128
Main results minibatch size	512
Discount γ	0.99
Optimizer	Adam
Learning rate	10^{-3}
Critic target update frequency	2
Critic Q-function soft-update rate τ	0.01
Actor update frequency	2
Actor log stddev bounds	$[-10, 2]$
Init temperature	0.1

C THE ATARI 100K EXPERIMENTS SETUP

For ease of reproducibility in Table 4 we report the hyper-parameter settings used in the Atari 100k experiments. We largely reuse the hyper-parameters from OTRainbow (Kielak, 2020), but adapt them for DQN (Mnih et al., 2013). Per common practise, we average performance of our agent over 5 random seeds. The evaluation is done for 125k environment steps at the end of training for 100k environment steps.

Table 4: A complete overview of hyper parameters used in the Atari 100k experiments.

Parameter	Setting
Data augmentation	Random shifts and Intensity
Grey-scaling	True
Observation down-sampling	84×84
Frames stacked	4
Action repetitions	4
Reward clipping	$[-1, 1]$
Terminal on loss of life	True
Max frames per episode	108k
Update	Double Q
Dueling	True
Target network: update period	1
Discount factor	0.99
Minibatch size	32
Optimizer	Adam
Optimizer: learning rate	0.0001
Optimizer: β_1	0.9
Optimizer: β_2	0.999
Optimizer: ϵ	0.00015
Max gradient norm	10
Training steps	100k
Evaluation steps	125k
Min replay size for sampling	1600
Memory size	Unbounded
Replay period every	1 step
Multi-step return length	10
Q network: channels	$32, 64, 64$
Q network: filter size	$8 \times 8, 4 \times 4, 3 \times 3$
Q network: stride	$4, 2, 1$
Q network: hidden units	512
Non-linearity	`ReLU`
Exploration	ϵ-greedy
ϵ-decay	5000

D FULL ATARI 100K RESULTS

Besides reporting in Figure 5 median human-normalized episode returns over the 26 Atari games used in (Kaiser et al., 2019), we also provide the mean episode return for each individual game in Table 5.

Table 5: Mean episode returns on each of 26 Atari games from the setup in Kaiser et al. (2019). The results are recorded at the end of training and averaged across 5 random seeds (the CURL's results are averaged over 3 seeds as reported in Srinivas et al. (2020)). On each game we mark as bold the highest score. Our method demonstrates better overall performance (as reported in Figure 5).

Game	Rainbow	SimPLe	OTRainbow	Eff. Rainbow	OT/Eff. Rainbow +CURL	Eff. DQN	Eff. DQN +DrQ (Ours)
Alien	318.7	616.9	824.7	739.9	**1148.2**	558.1	702.5
Amidar	32.5	88.0	82.8	188.6	**232.3**	63.7	100.2
Assault	231.0	527.2	351.9	431.2	543.7	**589.5**	490.3
Asterix	243.6	**1128.3**	628.5	470.8	524.3	341.9	577.9
BankHeist	15.6	34.2	182.1	51.0	193.7	74.0	**205.3**
BattleZone	2360.0	5184.4	4060.6	10124.6	**11208.0**	4760.8	6240.0
Boxing	-24.8	**9.1**	2.5	0.2	4.8	-1.8	5.1
Breakout	1.2	16.4	9.8	1.9	**18.2**	7.3	14.3
ChopperCommand	120.0	**1246.9**	1033.3	861.8	1198.0	624.4	870.1
CrazyClimber	2254.5	**62583.6**	21327.8	16185.3	27805.6	5430.6	20072.2
DemonAttack	163.6	208.1	711.8	508.0	834.0	403.5	**1086.0**
Freeway	0.0	20.3	25.0	**27.9**	**27.9**	3.7	20.0
Frostbite	60.2	254.7	231.6	866.8	**924.0**	202.9	889.9
Gopher	431.2	771.0	778.0	349.5	**801.4**	320.8	678.0
Hero	487.0	2656.6	6458.8	**6857.0**	6235.1	2200.1	4083.7
Jamesbond	47.4	125.3	112.3	301.6	**400.1**	133.2	330.3
Kangaroo	0.0	323.1	605.4	779.3	345.3	448.6	**1282.6**
Krull	1468.0	**4539.9**	3277.9	2851.5	3833.6	2999.0	4163.0
KungFuMaster	0.0	**17257.2**	5722.2	14346.1	14280.0	2020.9	7649.0
MsPacman	67.0	1480.0	941.9	1204.1	**1492.8**	872.0	1015.9
Pong	-20.6	**12.8**	1.3	-19.3	2.1	-19.4	-17.1
PrivateEye	0.0	58.3	100.0	97.8	105.2	**351.3**	-50.4
Qbert	123.5	**1288.8**	509.3	1152.9	1225.6	627.5	769.1
RoadRunner	1588.5	5640.6	2696.7	**9600.0**	6786.7	1491.9	8296.3
Seaquest	131.7	**683.3**	286.9	354.1	408.0	240.1	299.4
UpNDown	504.6	**3350.3**	2847.6	2877.4	2735.2	2901.7	3134.8
Median human-normalised episode returns	0.020	0.135	0.208	0.147	0.240	0.094	**0.270**

E IMAGE AUGMENTATIONS ABLATION

Following (Chen et al., 2020), we evaluate popular image augmentation techniques, namely random shifts, cutouts, vertical and horizontal flips, random rotations and imagewise intensity jittering. Below, we provide a comprehensive overview of each augmentation. Furthermore, we examine effectiveness of these techniques in Figure 6.

Random Shift We bring our attention to random shifts that are commonly used to regularize neural networks trained on small images (Becker & Hinton, 1992; Simard et al., 2003; LeCun et al., 1989; Ciresan et al., 2011; Ciregan et al., 2012). In our implementation of this method images of size 84×84 are padded each side by 4 pixels (by repeating boundary pixels) and then randomly cropped back to the original 84×84 size.

Cutout Cutouts introduced in DeVries & Taylor (2017) represent a generalization of Dropout (Hinton et al., 2012). Instead of masking individual pixels cutouts mask square regions. Since image pixels can be highly correlated, this technique is proven to improve training of neural networks.

Horizontal/Vertical Flip This technique simply flips an image either horizontally or vertically with probability 0.1.

Rotate Here, an image is rotated by r degrees, where r is uniformly sampled from $[-5, -5]$.

Intensity Each $N \times C \times 84 \times 84$ image tensor is multiplied by a single scalar s, which is computed as $s = \mu + \sigma \cdot \text{clip}(r, -2, 2)$, where $r \sim \mathcal{N}(0, 1)$. For our experiments we use $\mu = 1.0$ and $\sigma = 0.1$.

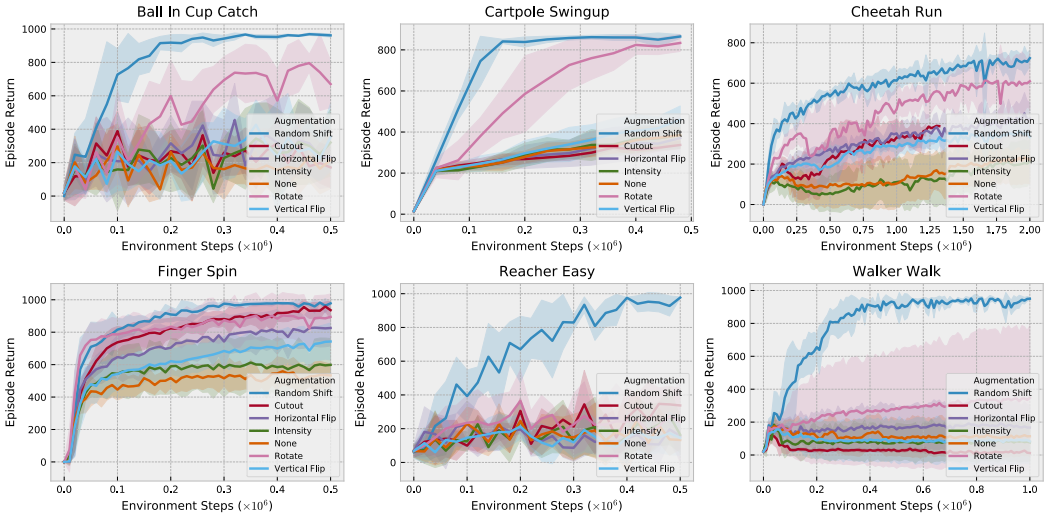


Figure 6: Various image augmentations have different effect on the agent's performance. Overall, we conclude that using image augmentations helps to fight overfitting. Moreover, we notice that random shifts proven to be the most effective technique for tasks from the DeepMind control suite.

Implementation Finally, we provide Python-like implementation for the aforementioned augmentations powered by Kornia (Riba et al., 2020).

```
import torch
import torch.nn as nn
import kornia.augmentation as aug

random_shift = nn.Sequential(nn.ReplicationPad2d(4),aug.RandomCrop((84, 84)))

cutout = aug.RandomErasing(p=0.5)

h_flip = aug.RandomHorizontalFlip(p=0.1)

v_flip = aug.RandomVerticalFlip(p=0.1)

rotate = aug.RandomRotation(degrees=5.0)

intensity = Intensity(scale=0.1)

class Intensity(nn.Module):
  def __init__(self, scale):
    super().__init__()
    self.scale = scale

  def forward(self, x):
    r = torch.randn((x.size(0), 1, 1, 1), device=x.device)
    noise = 1.0 + (self.scale * r.clamp(-2.0, 2.0))
    return x * noise
```

F K AND M HYPER-PARAMETERS ABLATION

We further ablate the K,M hyper-parameters from Algorithm 1 to understand their effect on performance. In Figure 7 we observe that increase values of K,M improves the agent's performance. We choose to use the [K=2,M=2] parametrization as it strikes a good balance between performance and computational demands.

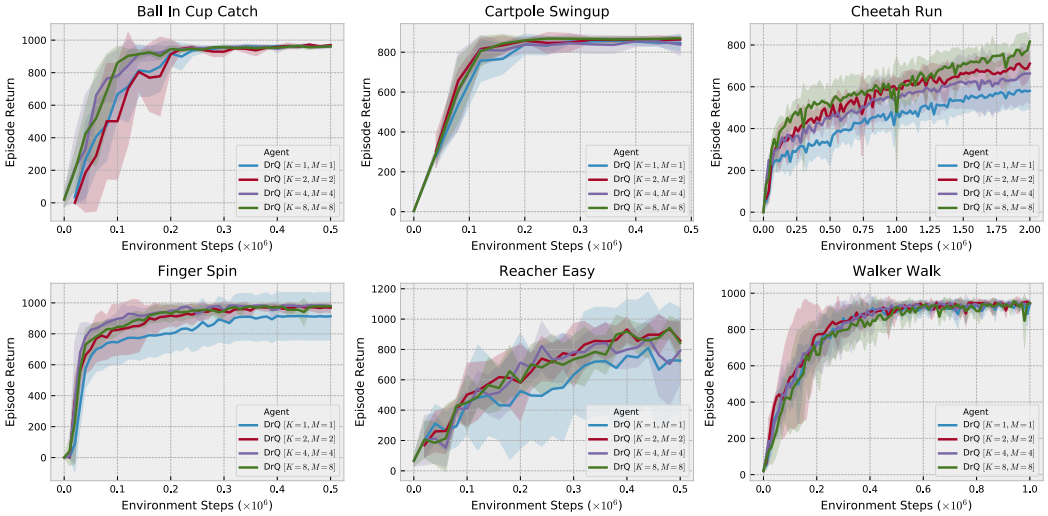
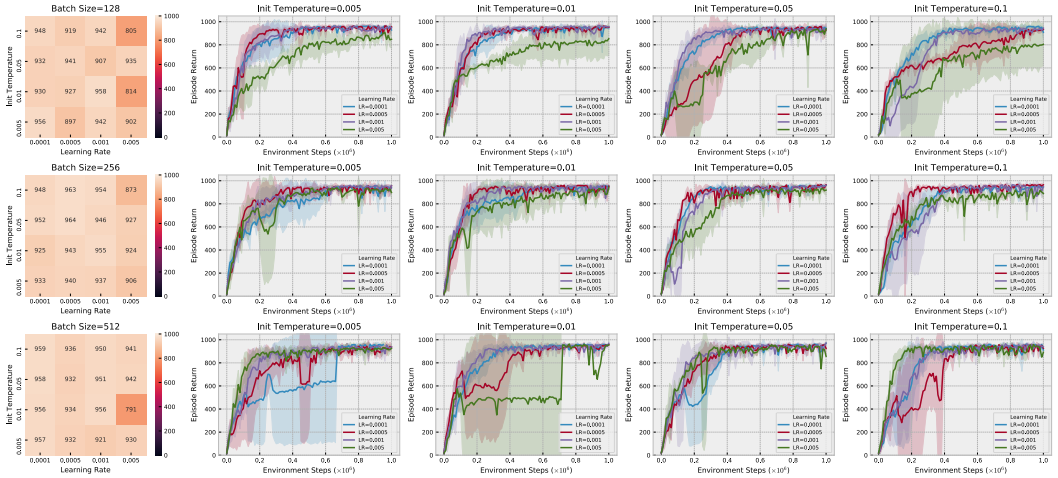


Figure 7: Increasing values of K,M hyper-parameters generally correlates positively with the agent's performance, especially on the harder tasks, such as Cheetah Run.

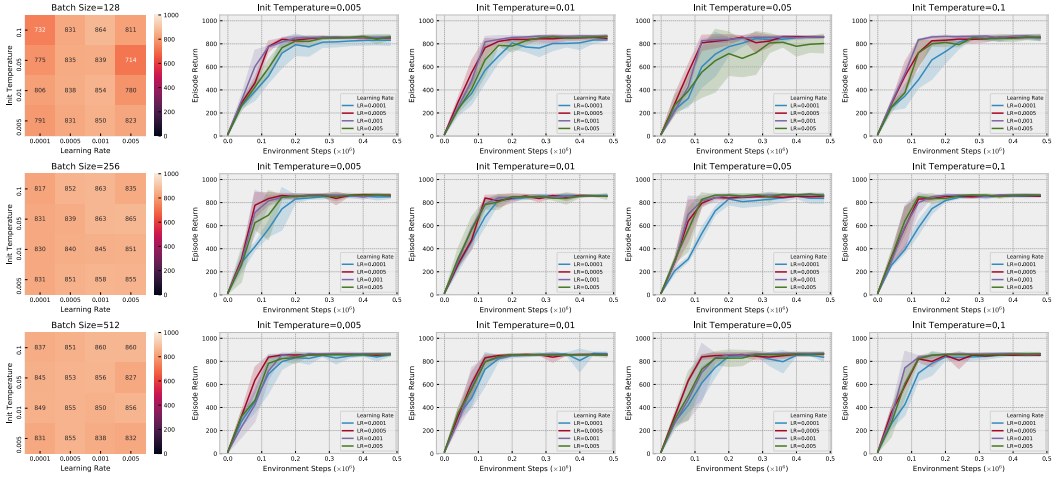
G ROBUSTNESS INVESTIGATION

To demonstrate the robustness of our approach (Henderson et al., 2018), we perform a comprehensive study on the effect different hyper-parameter choices have on performance. A review of prior work (Hafner et al., 2018; 2019; Lee et al., 2019; Srinivas et al., 2020) shows consistent values for discount $\gamma = 0.99$ and target update rate $\tau = 0.01$ parameters, but variability on network architectures, mini-batch sizes, learning rates. Since our method is based on SAC (Haarnoja et al., 2018), we also check whether the initial value of the temperature is important, as it plays a crucial role in the initial phase of exploration. We omit search over network architectures since Figure 1b shows our method to be robust to the exact choice. We thus focus on three hyper-parameters: mini-batch size, learning rate, and initial temperature.

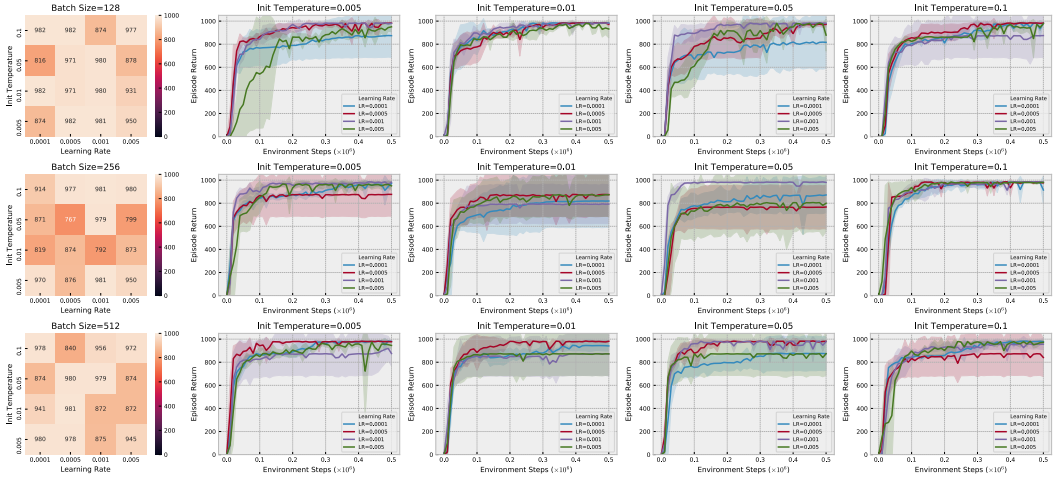
Due to computational demands, experiments are restricted to a subset of tasks from Tassa et al. (2018): Walker Walk, Cartpole Swingup, and Finger Spin. These were selected to be diverse, requiring different behaviors including locomotion and goal reaching. A grid search is performed over mini-batch sizes $\{128, 256, 512\}$, learning rates $\{0.0001, 0.0005, 0.001, 0.005\}$, and initial temperatures $\{0.005, 0.01, 0.05, 0.1\}$. We follow the experimental setup from Appendix B, except that only 3 seeds are used due to the computation limitations, but since variance is low the results are representative.



(a) Walker Walk.



(b) Cartpole Swingup.



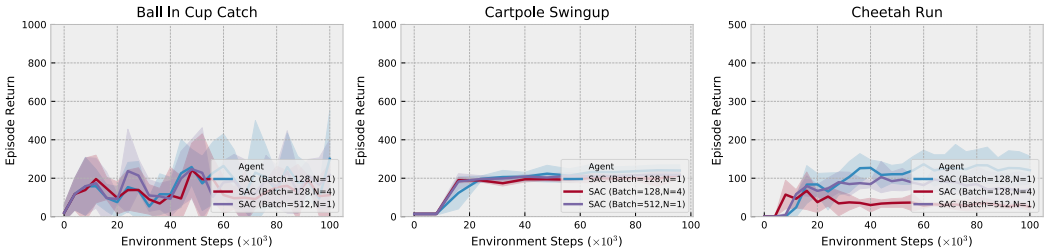
(c) Finger Spin.

Figure 8: A robustness study of our algorithm (**DrQ**) to changes in mini-batch size, learning rate, and initial temperature hyper-parameters on three different tasks from (Tassa et al., 2018). Each row corresponds to a different mini-batch size. The low variance of the curves and heat-maps shows **DrQ** to be generally robust to exact hyper-parameter settings.

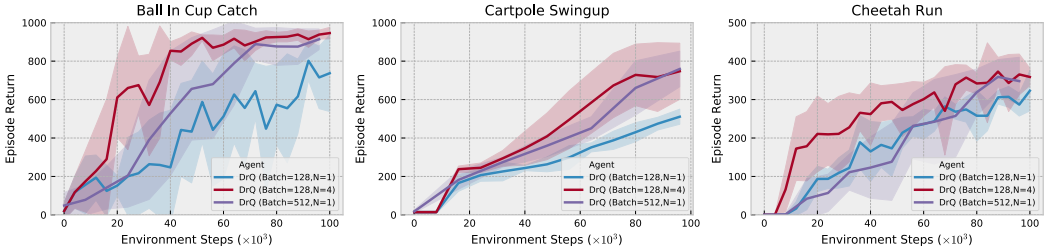
Figure 8 shows performance curves for each configuration as well as a heat map over the mean performance of the final evaluation episodes, similar to Mnih et al. (2016). Our method demonstrates good stability and is largely invariant to the studied hyper-parameters. We emphasize that for simplicity the experiments in Section 5 use the default learning rate of Adam (Kingma & Ba, 2014) (0.001), even though it is not always optimal.

H IMPROVED DATA-EFFICIENT REINFORCEMENT LEARNING FROM PIXELS

Our method allows to generate many various transformations from a training observation due to the data augmentation strategy. Thus, we further investigate whether performing more training updates per an environment step can lead to even better sample-efficiency. Following van Hasselt et al. (2019b) we compare a single update with a mini-batch of 512 transitions with 4 updates with 4 different mini-batches of size 128 samples each. Performing more updates per an environment step leads to even worse over-fitting on some tasks without data augmentation (see Figure 9a), while our method **DrQ**, that takes advantage of data augmentation, demonstrates improved sample-efficiency (see Figure 9b).



(a) Unmodified SAC.



(b) Our method **DrQ**.

Figure 9: In the data-efficient regime, where we measure performance at 100k environment steps, **DrQ** is able to enhance its efficiency by performing more training iterations per an environment step. This is because **DrQ** allows to generate various transformations for a training observation.