A. Routine TD-Loss Pseudocode

As mentioned in Section 4.3, we can learn the routine Q-function by performing efficient TD-updates for routines of all lengths, making use of matrix multiplication. Particularly, starting with a sampled action sequence from the replay buffer $a_{1:L}$, we calculate the TD-loss for all routines $n^{a_1}, n^{a_1,a_2}, ..., n^{a_{1:L}}$ as described by the following pseudocode based on the *TensorFlow* syntax:

```
# D, E: Routine decoder and encoder network
# Q, Q_tar: Q and target Q networks
# Pi: Policy
# gamma: discount factor
# L: maximum routine length

# dim: L x L
#[[1,     1,         1, ...,           1,             1],
# [0, gamma,     gamma, ...,       gamma,         gamma],
# [0,     0, gamma**2, ...,     gamma**2,      gamma**2],
# [                   ...                             ],
# [0,     0,         0, ...,gamma**(L-2), gamma**(L-2)],
# [0,     0,         0, ...,           0, gamma**(L-1)]]
r_discounts = (band_part(ones((L, L)), 0, -1)
    *constant([[gamma**i] for i in range(L)])

# dim: L
#[gamma, gamma**2, ..., gamma**L]
next_q_discounts = constant(
    [gamma**i for i in range(1, L+1)])

# input dim: N x |s|, N x |a|, N x L x |s|,
#            N x L,   N x L
def routine_TD_loss(s, a, next_s, r, t):
    # on-policy routine from next states
    # dim: N x L x |a|
    next_n = Pi(next_s)

    # autoencoded on-policy routine
    # dim: N x L x |a|
    ae_next_n = D.get_routine(
        E.sample_actions(next_n))

    # targets
    # dim: N x L
    y = (matmul(r, r_discounts) +
         t*next_q_discounts*Q_tar(next_s, ae_next_n))

    # routines from actions subsequences in a
    # dim N x |n|
    n = D.get_routine_subsequences(a)

    # tiled Q predictions
    # dim N x L
    q = Q(tile(reshape(s, [N, |s|, 1]), [1, 1, L]), n)

    # return average TD-loss and encoded routines
    return mean(square(q - y)), n
```

For further details, please refer to our shared implementation.

B. Integration Details

In Algorithm 1 we show the common optimization structure of our routine framework. Below, we further provide more details regarding the integration of our framework with *TD3* (Fujimoto et al., 2018) and *SAC* (Haarnoja et al., 2018a), including few conceptual dissimilarities with the original algorithms.

Algorithm 1 *Routine off-policy framework*

Initialize $\pi_\theta, Q^{\pi_\theta}_{\phi_1}, Q^{\pi_\theta}_{\phi_2}, D_{\omega_1}, E_{\omega_2}$.
Initialize $B_\pi \leftarrow \emptyset$, $count \leftarrow 0$
for $i = 1, 2, ...,$ **do**
  Observe s from the environment
  Query policy to obtain $n \sim \pi_\theta(s)$
  Sample $a_1, a_2, ..., a_l \sim D_{\omega_1}(n)$
  for $a = a_1, a_2, ..., a_l$ **do**
    Execute a in the environment, collect (s', r, t)
    Store $B = B \cup (s, a, s', r, t)$
    Sample $b = \{(s, a_{1:L}, s'_{1:L}, r_{1:L}, t_{1:L})_{|b|}\} \in B$
    for $m = 1, 2$ **do**
      Approximate J_Q and J_{lc} with b
      Update $Q^{\pi_\theta}_{\phi_m}$ with $\nabla_{\phi_m} J_Q$
      Update D_{ω_1}, E_{ω_2} with $\nabla_\omega (J_Q + J_{lc})$
    end for
    Update $count \leftarrow count + 1$
    if $count \mod delay = 0$ **then**
      Approximate J_π and J_{mto} with b
      Update π_θ with $\nabla_\theta J_\pi$
      Update D_{ω_1} with $\nabla_{\omega_1}(J_\pi - J_{mto})$
    end if
  end for
end for

The *Routine TD3* algorithm explores the environment by making use of independent Gaussian noise injected both at the routine and action levels. Moreover, when calculating the TD-loss for learning the routine Q-function, we chose to utilize the actual policy to obtain the next state target routine and avoid parameterizing a target policy as in the original algorithm. This choice did not appear to influence particularly the performance and was done with the purpose of simplification.

The *Routine SAC* algorithm bases its implementation on the automatic temperature adjustment version of *SAC* (Haarnoja et al., 2018b). Particularly, we keep the same original heuristic for the environment-specific action-selection entropy target of $-|a|$. However, when updating the temperature parameter α, we still utilize this target against the decoder's 'per-action entropy', rather than the overall entropy of chosen routines. Practically, the decoder's 'per-action entropy' is simply calculated dividing the overall entropy from the decoded action sequence distribution by its recovered length. Additionally, we make use of delayed policy training, as in *TD3*, updating our policy and target networks less frequently than the routine Q-function.

C. Algorithms Parameters

In this section, we describe the hyper-parameters choices made for all the evaluated algorithms in Section 6.

For the *TD3* and *SAC* algorithms we utilized the parameters provided in the original implementations. We share most of the *TD3* and *SAC* parameters with our routine-based versions of these algorithms with only minor differences. For example, in *TD3* we utilize a smaller target routine smoothing value of 0.1 to regularize the auto-encoded version of the predicted next state routine. Additionally, in both routine versions of *TD3* and *SAC* we use 2-layer fully-connected networks with 256 hidden units for both policy and Q function models.

We utilized simple rules to select the dimensionality of the routine representations and keep the structure of the additional routine decoder and encoder models light and efficient, comprising only a few hundred additional parameters. Particularly, using the notation from Figure 1, we let the routine space representation dimensionality be based on the original environment's action space dimensionality: $|n| = L \times |a|$. Additionally, we respectively set the first layer embeddings dimensionality to $|h| = 2^{\lceil log_2(|a|) \rceil}$ and the aggregated representation dimensionalities to $|g| = L \times |h|$. As we wanted to evaluate the general applicability of our framework, we did not substantially tune the hyper-parameters of these models. Thus, we did not explore using any information bottleneck between the action sequences space S^A and the routine space, but hypothesize this could yield even further efficiency improvements.

We list all the hyper-parameter choices in Table 2.

Table 2. Hyper-parameters used for the experimental evaluation.

Shared parameters	
buffer size $\|B\|$	100000
batch size $\|b\|$	256
minimum data to train	1000
optimizer	Adam
learning rate	0.001
optimizer β_1	0.9
policy delay	2
discount γ	0.99
polyak coefficient ρ	0.995
policy/Q network hidden layers	2
policy/Q network hidden dimensionality	256
routine space dimensionality $\|n\|$	$L \times \|a\|$
decoder/encoder hidden dimensionality $\|h\|$	$2^{\lceil log_2(\|a\|) \rceil}$
encoder aggregated dimensionality $\|g\|$	$L \times \|h\|$
J_{mto} coefficient	1
J_{lc} coefficient	1
Routine *TD3* parameters	
routine exploration noise	0.2
action exploration noise	0.1
target routine smoothing noise	0.1
Routine *SAC* parameters	
starting entropy temperature α	0.1
entropy temperature learning rate	0.0001
entropy temperature optimizer β_1	0.5

D. Implementation of Prior Algorithms

To compare the routine framework with prior methods reasoning with action repetitions, we implemented the off-policy version of the *FiGAR* algorithm by Sharma et al. (2017), named *FiGAR DDPG*. This algorithm works by parameterizing a policy outputting both an action and a probability distribution over a set of possible action repetitions. However, strictly following the implementations details and hyper-parameters described in the original paper yielded an algorithm which failed to learn meaningful behavior for the DeepMind Control Suite tasks. Thus, we implemented *FiGAR TD3*, a new algorithm that extends *FiGAR DDPG* by incorporating advances from *TD3*, together with several additional practices to stabilize its optimization procedures.

Particularly, *FiGAR TD3* makes use of double Q-learning, target policy smoothing, and delayed policy updates, as outlined in the paper by Fujimoto et al. (2018). Additionally, we found two additional changes that played an even more significant role on performance. These consist in greatly reducing the range of possible action repetitions and augmenting the original experience collection procedure. Specifically, *FiGAR DDPG* only records transitions in the replay buffer corresponding to the executed actions and repetitions. Instead, we augment the experience collection procedure by recording transitions after each environment step, relabeling the intermediate steps with the appropriate action repetitions. The original paper by Sharma et al. (2017) is also ambiguous on how the repetition values are logged in the replay buffer. We tried utilizing both the actor's normalized outputted logits and a one-hot representation, with the latter approach yielding substantially better results. We further modified most of the original hyper-parameters for performance, as shown Table 3, for further details please refer to our shared implementation.

E. Full Results

In this section, we provide the per-environment experimental results for the proposed routine framework.

In Figure 5 we show the performance curves representing the average cumulative returns obtained and the average number of policy queries as a function of the epoch in each of the fourteen tested environments for the *Performance Analysis* (from Section 6.1). We see the greatest perfor-

Table 3. Hyper-parameters used for *FiGAR TD3*

FiGAR-TD3 parameters	
buffer size $\|B\|$	100000
batch size $\|b\|$	256
minimum data to train	1000
optimizer	Adam
learning rate	0.001
optimizer β_1	0.9
policy delay	2
discount γ	0.99
polyak coefficient ρ	0.995
policy/Q network hidden layers	2
policy/Q network hidden dimensionality	256
action exploration noise	0.1
repetition exploration ϵ	0.2
exploration ϵ annealing steps	50000
target action smoothing noise	0.1

mance gains of the routine framework occur for the *TD3* algorithms in the harder exploration tasks. Overall, for the great majority of tasks, both routine versions of the examined algorithms provide improvements over their baselines. The average number of policy queries required to complete an episode appears to vary across the different environments. This can be seen as additional evidence that within the routine framework, agents do adaptively select routines of different lengths based on the granularity required to effectively solve a task. *Routine TD3* also outperforms both *FiGAR TD3* and *DynE TD3*, which appear to particularly struggle in some of the more complex locomotion tasks in the *Cheetah* and *Walker* environments.

In Figure 6 we provide the performance curves detailing the cumulative returns obtained by varying the maximum routine length L to 2, 4, 8, and 16 for our *Expressivity Analysis* (from Section 6.2). Overall, in terms of performance and stability, the best results are obtained by using a maximum routine length of 4 for our integration with *TD3* and a maximum routine length of 8 for our integration with *SAC*. These values appear to most optimally tradeoff the increased optimization complexity with the exploration, reward propagation and abstraction advantages provided by the routine framework.

F. Routine Analysis

In this section, we provide a further analysis of the routine framework and its main components through additional ablations and visualizations. For the experiments in this section, we show the average performance of different algorithms and configurations obtained on the subset of four task introduced in Section 6.2.

F.1. Exploration and Learning Benefits of Routines

As explained in Section 4, we hypothesize that the performance benefits observed from applying the routine framework to off-policy reinforcement learning algorithms come from both structured exploration and faster reward propagation.

To reinforce the hypothesis that routines facilitate structured exploration, we compare the states encountered through action-based and routine-based exploration. Particularly, we consider the *Cheetah run* task and collect different states from uniformly sampling either actions or routines. We categorize the states based on an internal Mujoco property named 'speed', representing the velocity of the agent. We use this property as an indicative way of separating states corresponding to behavior with different effects on the underlying task. We show the results in Figure 7, illustrating that routine-based exploration reaches states covering a significantly wider range of 'speeds', validating our hypothesis.

We also perform additional ablation experiments aimed at decoupling the benefits of structured exploration from faster reward propagation. Particularly, we implement new versions of our routine algorithms which are forced to re-plan while acting, by selecting a new routine at every environment step and only executing its first action. Hence, these agents should still benefit from faster reward propagation during learning, but lose the hypothesized structured exploration benefits during experience collection.

We summarize the performance of the *Routine re-plan* algorithms in Figure 8. For comparison, we use the performance obtained by the original routine algorithms both through standard execution and also matching the evaluation procedure of their re-planning counterparts, while still using full routines for experience collection. We average the performance of applying each considered setting of the routine framework to both *SAC* and *TD3* algorithms. The results show that the *Routine re-plan* agents initially learn slower, yet, eventually clearly outperform the *SAC* and *TD3* baselines. Their performance also lags consistently behind the standard routine algorithms (under both evaluation schemes), reinforcing our hypothesis that our framework provides complementary benefits both regarding structured exploration and faster reward propagation.

F.2. Effects of Routine Space Noise

We analyze the effects of removing either the routine space or action space exploration noise from the *Routine TD3* algorithm. We summarize the results in Figure 9. Both types of noise appear to have a positive impact on both final performance and learning speed. Action space noise

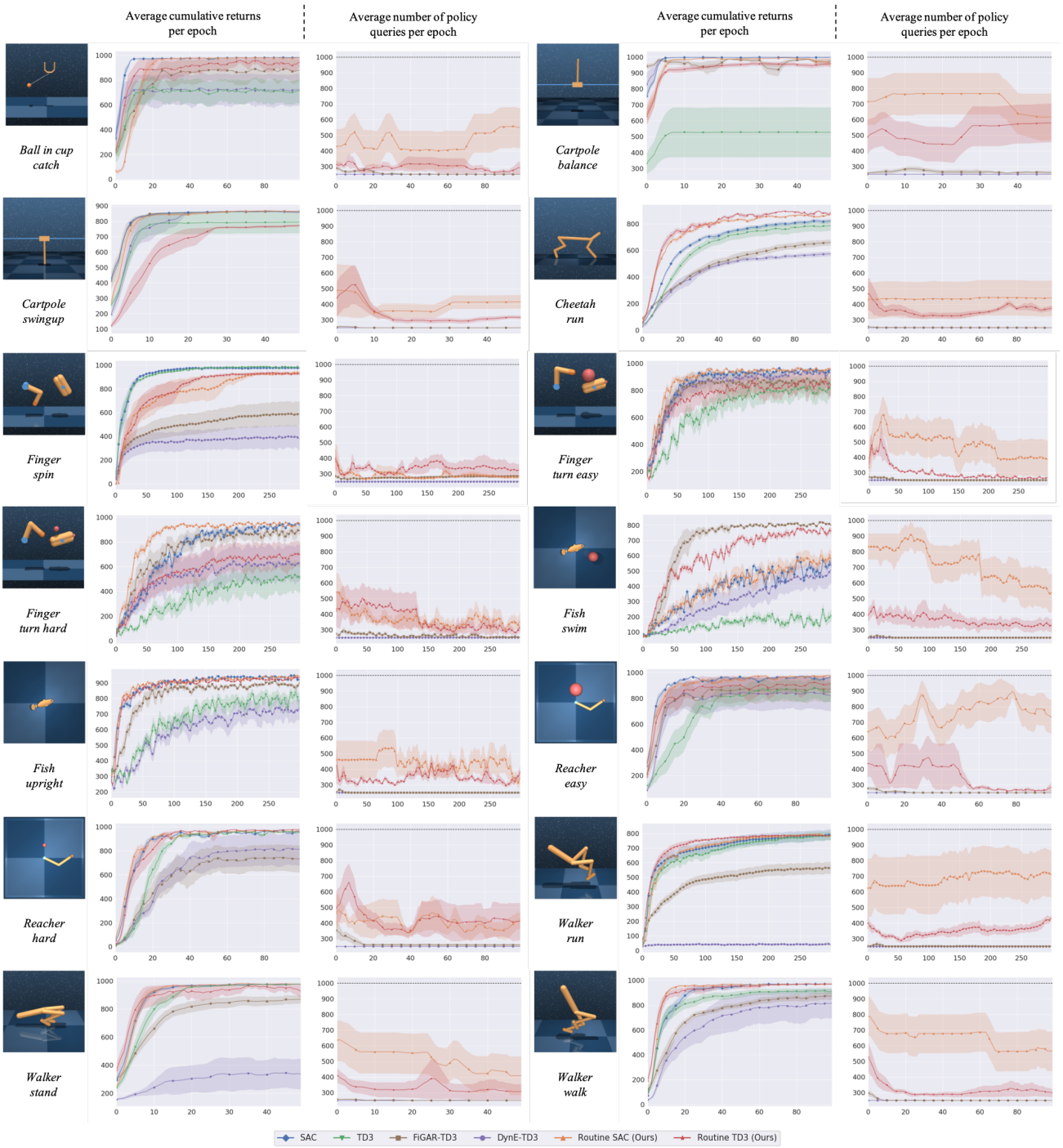

Figure 5. Average cumulative returns and number of policy queries in each of the fourteen different tested environments for the *Performance Analysis* from Section 6.1. Particularly, we report the mean and the standard deviation of these quantities across ten different runs as a function of the epoch number.

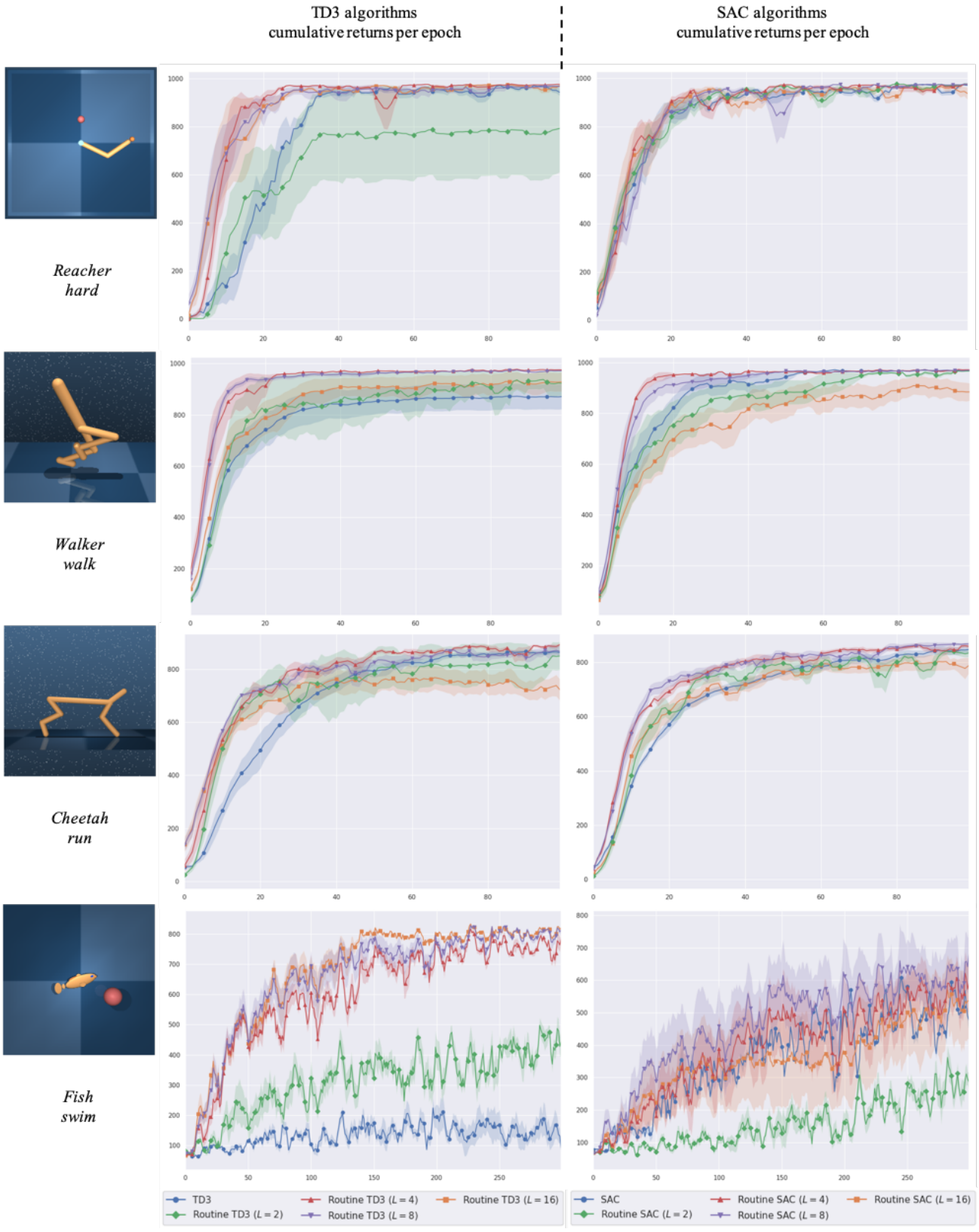

Figure 6. Cumulative returns in each of the four environments considered for the *Expressivity Analysis* in Section 6.2. We show the performance curves for the original algorithms and the integration of the routine framework for *TD3* (Left) and *SAC* (Right).

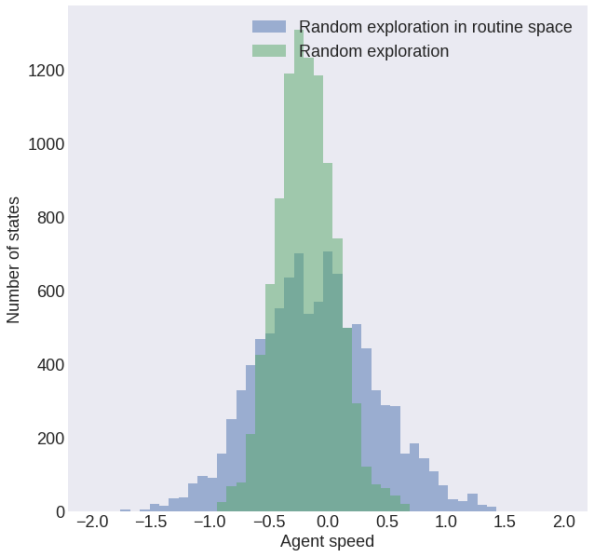

Figure 7. Number of states visited in ten episodes of experience collected in the *Cheetah run* task, as classified by the internal Mujoco 'speed' property (corresponding to the agent's velocity). We compare random uniform exploration by sampling from either the action space or the routine space.

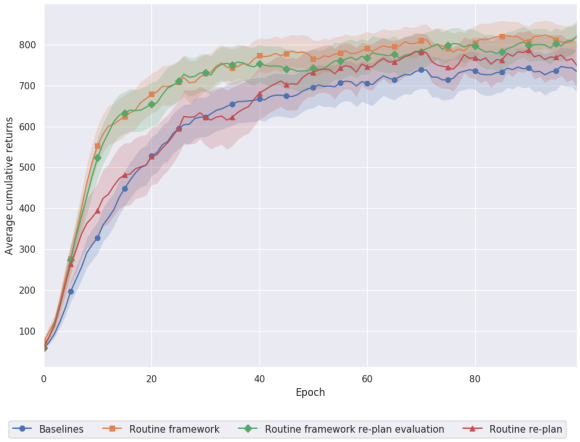

Figure 8. Average cumulative returns across the analyzed subset of tasks from decomposing the advantages of structured exploration from faster reward propagation. We average the performance of both *SAC*- and *TD3*-based algorithms for each of the different considered configurations of the routine framework. We repeat each experiment five times.

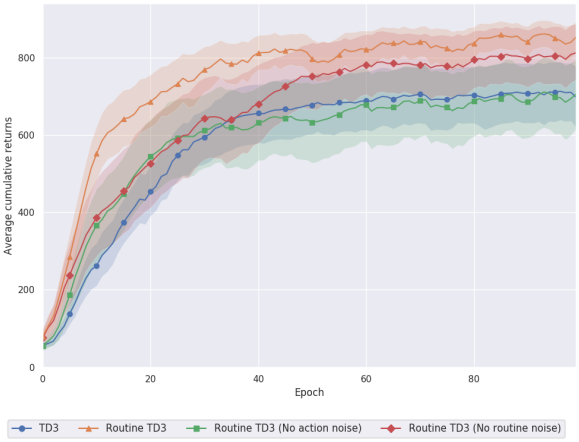

Figure 9. Average cumulative returns across the analyzed subset of tasks from removing either action space or routine space noise from *Routine TD3*. We repeat each experiment five times.

appears to be a crucial component in exploration throughout learning and disabling it makes *Routine TD3* converge to significantly worse policies. Routine space noise appears to have a greater effect on exploration early on, affecting more prominently the algorithm's learning speed.

F.3. Routines Visualization

To understand what kinds of behavior are encoded in routines, we collect visualizations by rendering the considered environments after performing each of the actions sampled by the routine decoder. We show these renderings in Figure 10, assigning them corresponding semantic labels. Specifically, different routines appear to perform simple behaviors that can be reused effectively in multiple situations, allowing the policy to reason with higher-level abstractions. For example, in the *Fish swim* task, different routines correspond to moving the agent's tail in different directions and to different extents, allowing the agent to maneuver towards any target.

Figure 10. Visualizations of random routines by rendering the environments after executing each action sampled by the routine decoder. We assign each a semantic label based on their observed effects.