Towards Real Robot Learning in the Wild: A Case Study in Bipedal Locomotion - Supplementary Material

1 Video and Visualisations

See website for supplementary video and further visualisations: https://sites.google.com/view/op3-vision-wild

2 Detailed Framework Overview

We employ a distributed training framework, which, in principle, admits an arbitrary number of robots in different geographic locations with different environment properties, and does not require additional hardware or human intervention (except for battery exchange and robot repair as required). All robots share the same control policy. Policy inference for action selection is performed asynchronously on each robot's onboard computer relying only on onboard sensors. Collected data is sent to a central data storage ("replay buffer"). A learner (running on a separate compute server "in the cloud") draws data from the data storage and computes updates to the policy and value function. The robots update their policy parameters at the beginning of each episode. Rewards are computed using onboard sensors only. When a robot falls it is reset to a standing pose via a built-in reset controller. In the experiments presented in this paper we prevent the robots from leaving their workspace by enclosing the workspace with a foam wall (akin to a child's play pen).

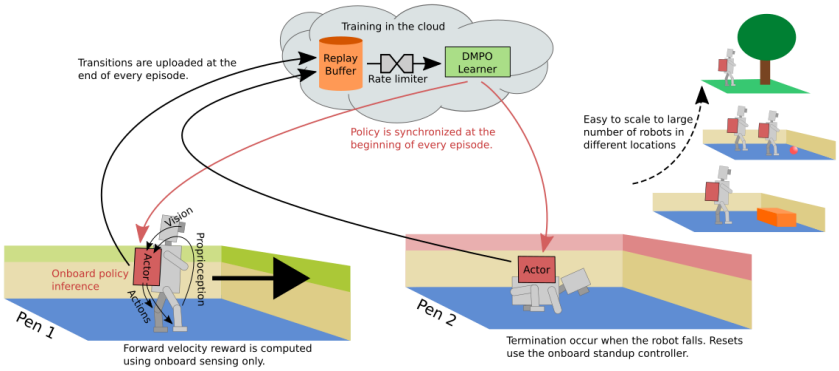

Figure 1: A distributed learning framework is employed. The policies are directly run on the different robots and synchronized at the beginning of every episode. Only proprioception and vision are required without additional instrumentation, allowing easy deployment in different environments. After every episode the collected transitions are uploaded to the replay buffer and sampled by the learner. A rate limiter imposes an upper limit on the number of learner steps as a function of environment steps.

5th Conference on Robot Learning (CoRL 2021), London, UK.

3 Practical Difficulties and Solutions

To further clarify our instrumentation free approach (see also Figure 1), Table 1 compares it to a hypothetical fully instrumented approach which is more representative of today's reinforcement learning setups in the real world.

Difficulty	Instrumented Approach	Our Approach
Reward computation	Use motion capture to estimate velocity and compute reward.	Use proprioception to estimate velocity and compute reward.
Limited workspace	Use motion capture based reward or termination to keep robot away from workspace boundaries.	Let the robot interact with the workspace boundary and learn to remain in the workspace.
Robot reset	Use a gantry to reset the robot to the middle of the workspace.	Use a standup controller and start from wherever the robot terminated the last episode.
Environment observability	Keep environment fix or provide map and localisation information (e.g. via motion capture) or provide processed images.	Provide access to camera images.

Table 1: List of practical difficulties and solutions.

4 Proprioceptive Reward Computation

For the following we employ three different coordinate frames, an arbitrary gravity-aligned inertial frame I, a robot body coordinate frame B, and a gravity-aligned robot coordinate frame A. We denote the coordinate frame a vector is expressed in using a prefix, e.g. ${}_I g = (0, 0, -9.81)$ is the gravity vector g expressed in the inertial coordinate frame I. A rotation matrix mapping from coordinates B to A is written as C_{AB}.

The employed reward, $R = w_f R_f + w_u R_u + w_n R_n$, has three components, a forward velocity component R_f with weight $w_f = 1.0$, an upright component R_u with weight $w_u = 0.1$, and a not turning component R_n with weight $w_n = 0.22$. All components are derived from the IMU's rotational rate measurement ${}_B\omega$ or orientation estimate C_{IB} and the encoders' angular measurements a (and its time differential $\dot{a}$ via numeric differentiation).

The location ${}_B l_i$ of the foot i w.r.t. the robot main body can be computed using forward kinematics, ${}_B l_i = {}_B l_i(a)$. Taking the differential gives ${}_B\dot{l}_i = J_i(a)\dot{a}$ using the Jacobian $J_i(a)$. The position of the foot ${}_I f_i$ in world coordinates can be written as ${}_I f_i = {}_I r + C_{IB}{}_B l_i$, where ${}_I r$ is the position of the robot in world coordinates. Taking the total derivative w.r.t. time gives ${}_I v_f^i = {}_I v_r + C_{IB}({}_B\omega \times {}_B l_i + {}_B\dot{l}_i)$, where ${}_I v_f^i$ is the velocity of foot i and ${}_I v_r$ the velocity of the body. This can also be expressed in the body coordinate frame, avoiding the dependency on the robot orientation: ${}_B v_f^i = {}_B v_r + {}_B\omega \times {}_B l_i + {}_B\dot{l}_i$, with which one can compute the velocity of a particular foot i in the body coordinate frame. But this identity can also be used to compute the velocity of the robot given that a particular foot s is stationary: ${}_B v_r = -{}_B\omega \times {}_B l_s - {}_B\dot{l}_s$. We can combine the two last identities to obtain the following foot velocity estimator (subscript s is a stationary foot):

$$ {}_B v_f^i = {}_B\omega \times ({}_B l_i - {}_B l_s) + {}_B\dot{l}_i - {}_B\dot{l}_s \tag{1} $$

Once we have estimated foot velocities, we simply compute the forward velocity reward R_f by summing the forward velocities of all feet in a gravity aligned coordinate frame

$$ R_f = \sum_i (1, 0, 0) \cdot C_{ABB} v_f^i \tag{2} $$

where C_{AB} maps to gravity aligned coordinates and is a function of the gravity direction ${}_B g = C_{IB}^T {}_I g$.

The upright reward is based on the norm of the x-y component of the normalized gravity when expressed in the body frame $R_u = \text{clip}(1 - (\|{}_B g_{xy}\|/\|{}_B g\| - o)/s, 0, 1)$, with offset $o = 0.2$ and scale $s = 0.2$.

The not turning reward is computed by taking the dot product between the estimate of the current forward direction and the one K steps ago: $R_n = C^k_{IB}(1,0,0) \cdot C^{k-K}_{IB}(1,0,0)$. Note that this relies on the IMU orientation estimates which are subject to drift in the heading component. But given that $K = 10$ (corresponding to 0.5 s), this remain within a time frame where the drift can be neglected.

5 DMPO

DMPO is an actor critic algorithm which iterates between policy improvement and policy evaluation. The essence of the policy improvement step is to solve (for each state s)

$$\max_{\pi} \mathop{\mathbb{E}}_{a\sim\pi^k(\cdot|s)} \left[\frac{e^{Q^k(a,s)/T}}{Z^k(T,s)} \log \pi(a|s) \right] - \alpha \, KL\left[\pi^k(\cdot|s)||\pi(\cdot|s)\right], \tag{3}$$

where $\pi^k(a|s)$ is the policy after the kth iteration, $Q^k(a,s)$ is an estimate of its Q-function, $Z^k(T,s)$ normalizes the exponential factors to a probability distribution, and T and α are parameters which are automatically adapted during learning in a way which ensures that policy changes remain within a desired trust region (specified by a set of hyperparameters typically denoted ϵ). The full loss employs several tricks which are discussed in the original references [1] and [2] as well as the open-source reference implementation https://github.com/deepmind/acme/tree/master/acme/agents/tf/dmpo.

Policy evaluation uses distributional reinforcement learning [3] which models the distribution of future returns $R^k(a,s) \sim p^k(q|a,s)$ rather than the Q-function which represents the average future returns, i.e. $Q^k(a,s) = \mathbb{E}[R^k(a,s)]$. TD(0) policy evaluation aims to find the random variable $R(a,s) \sim p(q|a,s)$ which minimizes its distance from the random variable $r(a,s,s') + \gamma \, \mathbb{E}_{a'\sim\pi^k(\cdot|s')}\left[R^k(a',s')\right]$ over all transitions (s,a,s') encountered by the agent. In our case we model $p(\cdot|a,s)$ as a histogram, and the distance between the random variables is the KL distance between their distributions. Again, we refer the reader to the implementation at https://github.com/deepmind/acme/tree/master/acme/agents/tf/dmpo for all details regarding the policy evaluation loss (such as TD(n) instead of TD(0), binning hyperparameters etc.).

The only differences between our and the reference implementation are that:

1. We use JAX instead of TensorFlow.
2. We run in a distributed setting with separate actor and learner processes.
3. We calculate losses and gradient updates over batches of multi-step trajectories instead of batches of single-step transitions.
4. We estimate n-step returns directly inside the critic loss from the trajectories stored in the replay buffer.

All hyperparameters are listed in Table 2.

6 Action Filter

An exponential filter is applied to the actions before sending them to the actuators. Let the agent action be denoted by $u_t = \pi(o_t) \in R^{20}$. The final command that is sent to the actuators is given by $\bar{p}_t \in R^{20}$, $\bar{p}_t = \bar{p}_{t-1} * c + u_t * (1 - c)$, where c is the filter coefficient (we set $c = 0.9$ for all experiments).

7 Detailed Simulation Results

A more detailed analysis of the simulation results is provided in Figure 2. In addition to the total episode reward, we also plot the average episode length as well as the average per-step reward. The

policy learning rate	0.0001
critic learning rate	0.0001
dual learning rate	0.01
trajectory length	48
batch size	32
updates per step	0.25
num samples	20
discount	0.99
init log temperature	10
init log alpha mean	10
init log alpha stddev	1000
epsilon	0.1
epsilon mean	0.0025
epsilon stddev	$1e-6$
epsilon penalty	0.001
per dim constraining	True
action penalization	True
n step	5
target actor update period	25
target critic update period	100
clipping	True
vmin	-150
vmax	150

Table 2: List of DMPO hyperparameters.

average episode length is indicative of how long the robot stands before falling. At the beginning, it increases sharply as the agents learn to maintain balance. It then decreases as the agents attempt to move forwards, before increasing again when the agents' behavior mature and become more reliable. If we add a wall to the setup, the *pose* agent's episode length decrease strongly as it runs into the wall and is unable to prevent falling. *Blind* and *vision* agents are both able to maintain much longer episodes.

The per-step reward is mostly indicative of the speed the robot achieves and its dependence on the wall configuration is weak. Both the *pose* and vision agents have higher per-step reward than the *blind* agent. This is likely due to the type of gait that can be employed when not having to recover from collisions.

Figure 3 visualises the behavior of the learned policies. The *vision* agent achieves a good performance and is able to avoid the additional wall. The *pose* agent shows the best performance in the open court but is unable to deal with the additional wall, running into it and falling. The *blind* agent is slower (as can be seen from the closer distance between the robot snapshots) but is able to recover from wall collisions.

8 More Details for Real Experiments

We provide two additional figures for the real world experiments. Figure 4 compares the sample complexity of the single and the two-robot experiments. Figure 5 depicts the reset to runtime ratio for the two-robot experiment.

9 Image Processing Network Ablations

In order to test the effect of network architectures on data efficiency we ran different ablations in simulation. Here we report some of the more important findings. While our simulation does not try to mimic the visuals encountered in the real world, we randomize textures in simulation to roughly match the diversity of possible visual inputs. This ensures that our conclusions regarding

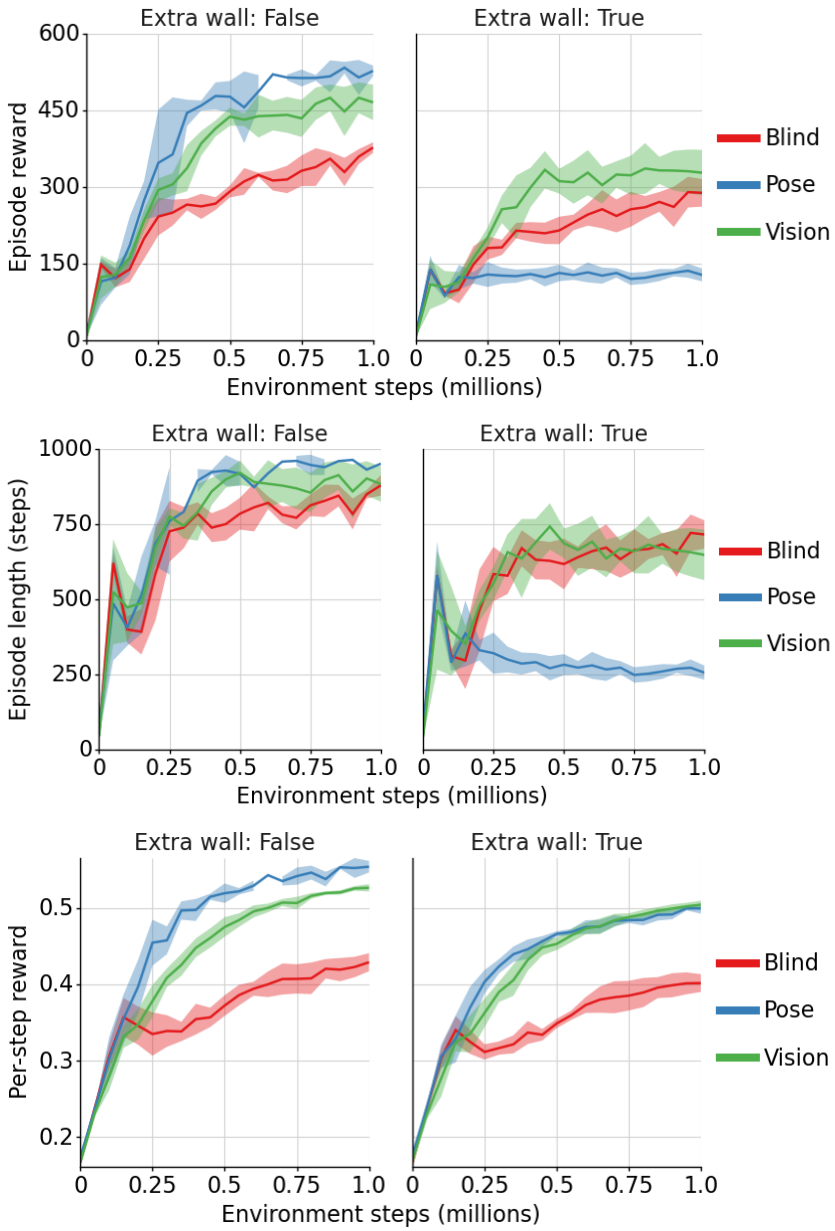

Figure 2: Simulation results comparing agents trained with vision and proprioceptive sensing (green), with groundtruth pose and proprioceptive sensing (blue), and with proprioceptive sensing only (red). The right plots show the performance of the same agents in a court with an additional wall which was not present during training. The shaded area corresponds to 95% confidence interval across seeds. **Top**: total episode reward. **Middle**: episode length. **Bottom**: per-step reward.

viable network architectures are likely to also apply in the real world. See Figure 6 for an example demonstration of the visual gap between simulation and the real robot.

We used the employed 3-block Resnet as the base network for our studies, and ablated different settings such as the depth and width of the resnet channels, usage of a shared or separate resnet torso between the actor and critic as well as the number of past stacked frames provided as input to our vision pipeline (see Figure 7 and Figure 8). With regards to channel depth and width we found little improvement in performance with larger networks; in practice, a fairly small network worked quite well for our setting. We did find that sharing the resnet torso between the actor and critic led to faster

Figure 3: Trajectory visualisation for the learned agents. **Top**: *vision* agent. **Middle**: *pose* agent. **Bottom**: *blind* agent. The two left examples are tested with additional wall to observe the reaction of the robot. The white square highlights the start position. Spacing between robot snapshots is 1 s.

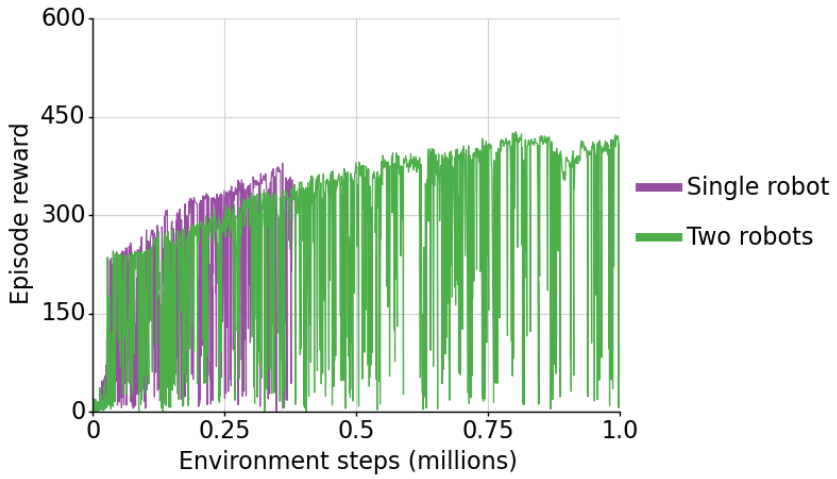

Figure 4: A comparison of learning curves from two independent real robot experiments. The green curve corresponds to a two robot experiment distributed across two different locations while the purple curve corresponds to a single robot experiment which took place in one of the two locations. We see that the sample complexity is similar (within variability observed in analogous simulation experiments where it is easy to run multiple seeds). Note that this is despite the fact that the agent has to deal with more diverse inputs in the two-robot experiment.

learning (Figure 7, left vs right). Lastly, we did not see a change in performance when increasing the history of frames we stack as inputs to our network (Figure 8).

References

[1] A. Abdolmaleki, J. T. Springenberg, J. Degrave, S. Bohez, Y. Tassa, D. Belov, N. Heess, and M. Riedmiller. Relative entropy regularized policy iteration, 2018.

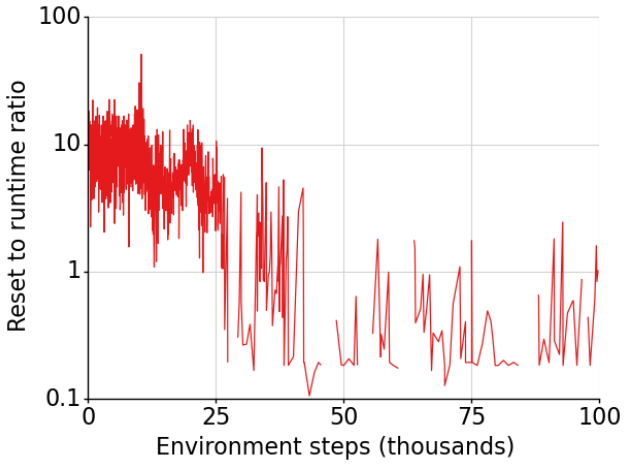

Figure 5: The rate at which we can collect data is not fixed throughout an experiment. The robot falls very often at the beginning of an experiment, which means that we have to execute our stand up reset controller much more frequently. The plot shows the ratio of the time we spent on resets to that spent on collecting data as the learning progresses in the two-robot experiment. We see that the experiment time during the first 25k steps is dominated by the resetting behavior.

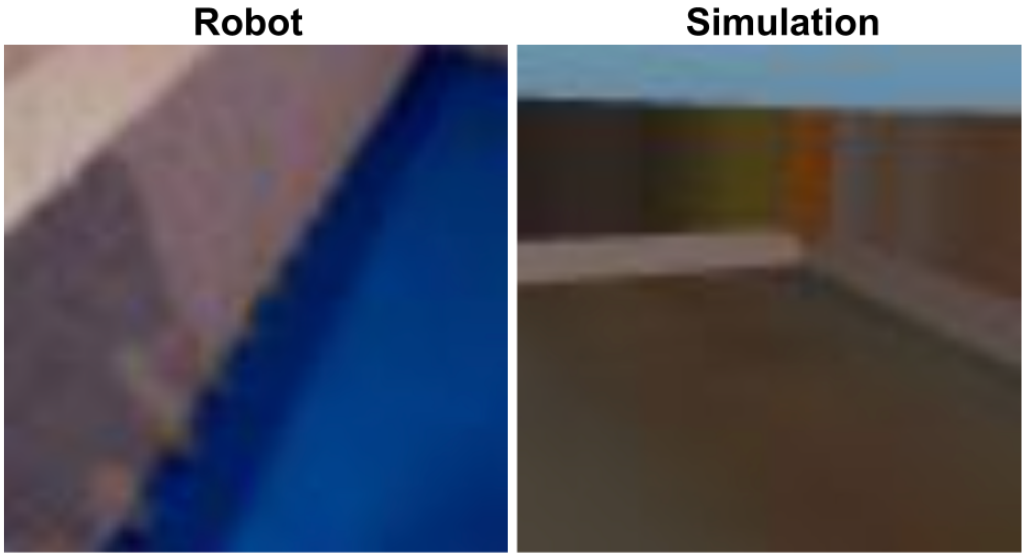

Figure 6: An example comparison of the camera observations on the robot and in simulation. We make no attempt to match the robot's visual inputs in simulation since that would require realistic modeling of shadows, varying lighting conditions, and artifacts such as motion blur. We only randomize the simulated textures to roughly match the expected diversity which allows us to more accurately tune the vision architecture.

[2] A. Abdolmaleki, J. T. Springenberg, Y. Tassa, R. Munos, N. Heess, and M. Riedmiller. Maximum a posteriori policy optimisation, 2018.

[3] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

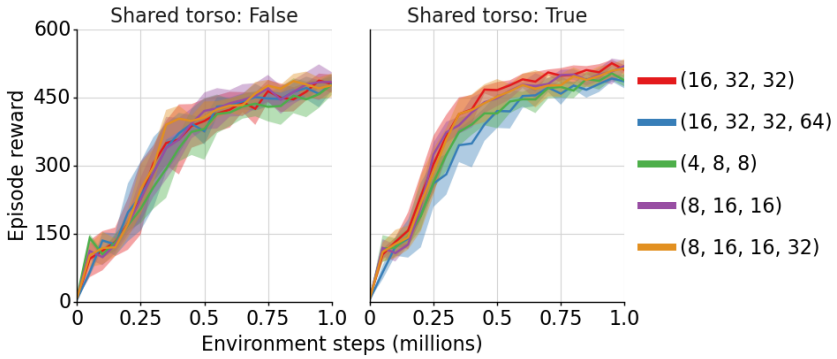

Figure 7: Ablation of ResNet channels. **Left**: without shared torso. **Right**: with shared torso. A small network with a shared torso between the actor and critic tends to be best in terms of data efficiency.

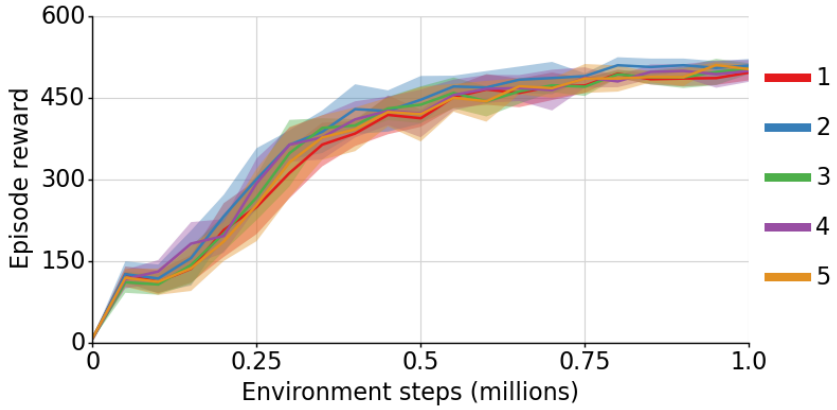

Figure 8: Ablation for different image stacking lengths. Providing more past images to the agent does not seem to improve performance.