Learning to Locomote: Understanding How Environment Design Matters for Deep Reinforcement Learning

Daniele Reda∗
dreda@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Tianxin Tao∗
taotianx@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Michiel van de Panne
van@cs.ubc.ca
University of British Columbia
Vancouver, Canada

ABSTRACT

Learning to locomote is one of the most common tasks in physics-based animation and deep reinforcement learning (RL). A learned policy is the product of the problem to be solved, as embodied by the RL environment, and the RL algorithm. While enormous attention has been devoted to RL algorithms, much less is known about the impact of design choices for the RL environment. In this paper, we show that environment design matters in significant ways and document how it can contribute to the brittle nature of many RL results. Specifically, we examine choices related to state representations, initial state distributions, reward structure, control frequency, episode termination procedures, curriculum usage, the action space, and the torque limits. We aim to stimulate discussion around such choices, which in practice strongly impact the success of RL when applied to continuous-action control problems of interest to animation, such as learning to locomote.

CCS CONCEPTS

• Computing methodologies → Animation; Reinforcement learning.

KEYWORDS

reinforcement learning, simulation environments, locomotion

1 INTRODUCTION

The skilled control of movement, via learned control policies, is an important problem of shared interest across physics-based character animation, robotics, and machine learning. In particular, the past few years have seen an explosion of shared interest in learning to locomote using deep reinforcement learning (RL). From the machine learning perspective, this interest stems in part from the multiple locomotion tasks that found in RL benchmark suites, such as the OpenAI Gym. This allows for a focus on the development of new RL algorithms, which can then be benchmarked against problems that exist as predefined RL environments. In their basic form, these environments are innocuous encapsulations of the simulated world and the task rewards: at every time step they accept an action as input, and provide a next-state observation and reward as output. Therefore, this standardized encapsulation is general in nature and is well suited for benchmark-based comparisons.

However, there exist a number of issues and decisions that arise when attempting to translate a given design intent into the canonical form of an RL environment. These translational issues naturally arise in the context of animation problems, where design intent is the starting point, rather than a predefined RL environment such as found in common RL benchmarks. In this paper, we examine and discuss the roles of the following issues related to the design of RL environments for continuous-action problems: (a) initial-state distribution and its impact on performance and learning; (b) choices of state representation; (c) control frequency or “action repeat”; (d) episode termination and the “infinite bootstrap” trick; (e) curriculum learning; (f) choice of action space; (g) survival bonus rewards; and (h) torque limits. In the absence of further understanding, these RL-environment issues contribute to the reputation of RL as yielding brittle and unpredictable results. Some of these issues are mentioned in an incidental fashion in work in animation and RL, which serves to motivate the more comprehensive synthesis and experiments presented in this paper. We document these use cases in the sections dedicated to individual issues.

Our work is complementary to recent RL work that examines the effect of algorithm hyperparameters, biases, and implementation details, e.g., [Andrychowicz et al. 2020; Henderson et al. 2018; Hessel et al. 2017; Zhang et al. 2018]. Henderson et al. [2018] discusses the problem of reproducibility in RL, due to extrinsic factors, such as hyperparameter selections and different code bases, and intrinsic factors, such as random seeds and implicit characteristics of environments. Hessel et al. [2019] analyze the importance of multiple inductive biases and their impact on the policy performance, including discount, reward scaling and action repetitions. Engstrom et al. [2020] points to certain code-level optimizations that sometimes contribute the bulk of performance improvements. Some of these configuration decisions are more impactful than others. The performance of RL policies can be sensitive to the change of hyperparameters which indicates the brittleness of RL algorithms. Andrychowicz et al. [2020], concurrent to our work, performs a large scale ablation of hyperparameters and implementation choices for on-policy RL, including a mix of algorithmic and environment choices.

2 EXPERIMENTAL SETTING

We consider the traditional RL setting, in which a Markov Decision Process (MDP) is defined by a set of states $S$, a set of actions $A$, a transition probability function $p: S \times A \to \mathcal{P}(S)$, which assigns to every pair $(s, a) \in S \times A$ a probability distribution $p(\cdot|s, a)$, representing the probability of entering a state from state $s$ using action $a$, a reward function $R: S \times S \times A \to \mathbb{R}$, which describes the reward $R(s_{t+1}, s_t, a_t)$, associated with entering state $s_{t+1}$ from state $s_t$, using action $a_t$, and a future discount factor $\gamma \in [0, 1]$ representing the importance of future rewards. The solution of an MDP is a policy $\pi^*_A: S \to A$, parameterized by parameters $\phi$, that for every $s_0 \in S$ maximises: $J(\phi) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, s_t, a_t) \right]$, where
where the expectation is taken over states $s_{t+1}$ sampled according to $p(s_{t+1}|s_t, a_t)$ and $a_t$ is the action sampled from $\pi_a(s_t)$.

Our algorithm of choice for the following experiments is TD3 [Fujimoto et al. 2018], a state of the art, off-policy, model free, actor-critic, RL algorithm. In TD3, an actor represents the policy $\pi(\phi)$ where $\phi$ are the weights of a neural network optimized by taking the gradient of the expected return $\nabla J(\phi)$, through the deterministic policy gradient algorithm [Silver et al. 2014]: $\nabla J(\phi) = \mathbb{E} [\nabla_a Q^\pi(s, a) \nabla_\phi \pi_a(s)]$, and a critic is used to approximate the expected return $Q^\pi(s, a)$ which corresponds to taking an action $a$ in state $s$ and following policy $\pi$ thereafter. The critic is also a neural network parameterized by $\theta$ and optimized using temporal difference learning using target networks [Mnih et al. 2015] to maintain a fixed objective $\nabla J(\theta) = \mathbb{E} [Q^\pi(s, a) - y]^2$ where $y$ is the target and is defined as $y = r_t + \gamma \max_{a} Q^\pi(s_{t+1}, a_{t+1})$.

We explore, evaluate, and discuss the impact of each defining component of the environment, each potentially affecting the final performance and learning efficiency of policies in locomotion environments. For each of these experiments, unless otherwise stated, we use an implementation of TD3 based on the original code from the authors and adapted for more experimental flexibility. Hyperparameters and network architectures are given in appendix A. All of our experiments are based on the Bullet physics simulator [Coumans and Bai 2019] and its default Roboschool locomotion environments available through Gym [Brockman et al. 2016]. Each experiment is performed a total of 10 times, and averaged across seeds. The set of seeds is always the same. In a few cases, we rely on existing published results, using them to provide relevant insights in our context.

We now present each RL environment design issue in turn. For clarity, we combine the explanation, related work, and results for each.

## 3 INITIAL-STATE DISTRIBUTION

The distribution from which the initial states of simulation episodes are sampled from plays an important role in learning and in exploration in RL. Many reinforcement learning locomotion benchmarks [Coumans and Bai 2019; Todorov et al. 2012], the default initial state is sampled from a very narrow distribution, nearly deterministic, at the beginning of a new episode. Recent work [Ghosh et al. 2018] addresses the problem of solving harder tasks by partitioning the initial state space into slices and train an ensemble of policies, one on each slice. Packer et al. [2018] looks at a similar problem, but instead of modifying the initial-state distribution, changes the environment dynamics, by changing mass, forces and lengths of links at every trajectory. In our experiments, changing the initial-state distribution does not affect the underlying environment dynamics, which remain the same. It instead affects the way the state-space is presented to the RL algorithm.

Default PyBullet locomotion environments create a new initial state by sampling the joint angles of each link from a uniform distribution $U(-0.1, 0.1)$. Instead, we sample each joint angle from a new distribution $U(\kappa \theta_{\text{min}}, \kappa \theta_{\text{max}})$, where $\theta_{\text{min}}$ and $\theta_{\text{max}}$ are the specified lower and upper limit of the joint, predefined in the robot description files (i.e. usually URDF or XML files). The parameter $\kappa$ quantifies the width of the initial-state distribution to investigate how its variance affects the policy.

We run experiments with the following environments: Ant, HalfCheetah, Hopper and Walker2D, and illustrate the test episode return in Figure 1. For Ant and HalfCheetah environments, results for the original narrow initial-state distribution and the broad initial-state distribution with $\kappa = 1$ are shown. For the Hopper and Walker environment, we show additional training curves with different values of $\kappa$ to investigate the influence of $\kappa$ at finer scales. A broader initial-state distribution leads to worse sampling efficiency and a lower episode return in the end for the majority of environments. We believe that for Ant and HalfCheetah, the largely invariant nature of the learning to the initial state distributions stems from the large degree of natural static stability for these systems after falling to the ground. This stability leads to a rapid convergence to similar states from a wide range of initial states.

The difference in performance caused by broadening the initial-state distribution reflects its impact, which is often neglected, in policy training. A narrow initial-state distribution increases learning efficiency compared with tasks with same underlying mechanisms $P(s_{t+1}|s_t, a_t)$ but a broader initial-state distribution $p(s_0)$. Intuitively, having a restricted range of initial states allows the agent to focus on those initial states, from which it learns how to act. If the agent is always dropped in the environment in a very different state, it is more difficult for the policy to initially learn to act, given that the experiences will represent very disjoint regions of the state space. However, learning on a broader initial-state distribution results in a more robust and general final policy. Table 1 shows results for policies trained on the default environments, with narrow initial-state distribution, when tested with different initial-state distributions. The policies trained on narrow initial-state distributions

| Test environment | $\kappa$ | Matching ISD | Narrow ISD |
|------------------|----------|--------------|------------|
| AntBulletEnv     | 0        | 3422.26      | N/A        |
| AntBulletEnv     | 1        | 3139.58      | 1647.18    |
| HalfCheetahBulletEnv | 0        | 2808.26      | N/A        |
| HalfCheetahBulletEnv | 1        | 2001.52      | 2117.88    |
| HopperBulletEnv  | 0        | 2300.24      | N/A        |
| HopperBulletEnv  | 0.2      | 1622.55      | 1286.58    |
| HopperBulletEnv  | 0.3      | 1357.53      | 761.47     |
| HopperBulletEnv  | 0.4      | 995.27       | 396.90     |
| HopperBulletEnv  | 0.5      | 528.76       | 325.70     |
| HopperBulletEnv  | 1        | 528.76       | 325.70     |
| Walker2DBulletEnv | 0        | 2459.30      | N/A        |
| Walker2DBulletEnv | 0.3      | 2358.83      | 644.02     |
| Walker2DBulletEnv | 0.5      | 1792.50      | 199.11     |
| Walker2DBulletEnv | 0.6      | 1497.13      | 131.47     |
| Walker2DBulletEnv | 0.7      | 1255.69      | 60.60      |
| Walker2DBulletEnv | 1        | 584.91       | 9.78       |

Table 1: Initial state distributions (ISD) generalization experiments. Here, Matching ISD gives the test reward when the training and test environments are matched in terms of their initial state distribution. Narrow ISD gives the test rewards when trained on a narrow distribution ($\kappa=0$), and tested on the various broader initial state distributions.
Learning to Locomote: Understanding How Environment Design Matters for Deep Reinforcement Learning

Figure 1: Learning curves as a function of the initial-state distribution parameterized by $\kappa$. For the Hopper and Walker environments, multiple values between 0 and 1 are sampled to better illustrate the relationship between the initial-state distribution and training performance. deliver significant worse run-time performances as $\kappa$ increases, showing a failure to generalize with respect to much of the state space. Thus, training with a broader initial-state distribution lead to a more robust policy that covers a wider range of the state and action space.

Summary: RL results can be strongly impacted by the choice of initial state distribution. A policy trained with the narrow initial distribution in commonly used environments often fails to generalize to regions of the state space that likely remain unseen during training. The broader the initial state distribution $p_0(s)$, the more difficult the RL problem can become. As a result, there is a drop in both sample-efficiency and reward with a wider initial-state distribution. However, the learned policies are more generalizable and more robust.

4 STATE REPRESENTATION

The state, at any given time, captures the information needed by the policy in order to understand the current condition of the environment and decide what action to take next. Despite the success of end-to-end learning, the choice or availability in the state representation can still affect the problem difficulty. In this section, we investigate how adding, removing and modifying information in the state affects the RL benchmarks. We discuss the following modifications: (1) adding a phase variable for cyclic motions, (2) augmenting the state with the Cartesian joint position in Cartesian coordinate, (3) removing contact boolean variables, (4) using the initial layers of a pre-trained policy as state representation for a new policy.

4.1 Phase Variable

Previous work on motion-imitation based control, e.g., [Adholahossein et al. 2019; Peng et al. 2018] uses a phase variable $\phi \in [0, 2\pi]$ as part of the state representation to implicitly index the desired reference pose to imitate at a given time. We can use prior knowledge about the desired period of the motion, $T$, to define a phase variable, even when there is no specific motion to imitate:

$$\phi = \left(\frac{2\pi}{T}\right)t, \quad 0 \leq t < T$$

The phase variable, $\phi$, can assist mastering periodic motion because it provides a compact abstraction of the state variables for nearly cyclic motions. Ideally, given a precise phase value for periodic motion for a specific character, the RL algorithm should perform better since the phase variable acts as a state machine to guide the learning process. We estimate $\phi$ from a previously learned controller and include this in the state representation.

Phase has been shown to serve as an effective internal representation for kinematic motion synthesis in Phase-Functioned Neural Networks (PFNN) [Holden et al. 2017]. They study the impact of phase by comparing three cases: using phase as an input to a gating network; using phase as an input to a fully connected network; and a baseline case, e.g., fully connected network without phase. The gating network with phase input provides the best motion quality, with crisp motion and ground contacts. Without a phase input, the simulated characters exhibit strong foot-skate artifacts, while a fully connected network with phase input often ignores the phase input, similarly resulting in lower-quality motions.

To prevent the phase variable being ignored, and inspired by the solution proposed by [Hawke et al. 2020] for the control command, we input the phase variable at every layer. We expect this modification to encourage the network to use these additional input features while both actor and critic networks still remain functional. Instead
of feeding the phase variable $\phi$ to the actor and critic networks, we input $\sin(\phi)$ and $\cos(\phi)$.

The averaged episode return over 10 runs are plotted in Figure 2. In the Hopper environment, the episode return for the state with phase has a more efficient learning curve, but eventually it converges to a similar result at the end. This indicates the fact that adding phase information to training can accelerate the learning process and result in better sampling efficiency. However, phase information does not lead to a higher return and this is explained by the fact that phase only provides an abstraction of the joint angles and velocities, adding no extra information that is not already present in the rest of the state. This can be further seen in the HalfCheetah environment, where there is no improvement when adding the phase information.

**Summary:** Adding phase information has limited impact on learning locomotion in the benchmark environments, which do not use reference motions but that could in principle benefit from a basic form of periodicity, such as that provided by a phase variable.

### 4.2 Joint Position

The inclusion of Cartesian joint position in the state description is another major difference between the work from the computer graphics community and RL community. In previous studies on motion synthesis and motion imitation [Holden et al. 2017; Park et al. 2018; Zhang et al. 2018b], both joint angles and Cartesian joint position are included in the state description. Although the joint positions provide redundant information, they are still found to be helpful in learning kinematic motion models and physics-based controllers. We expect it to be more important for complex 3D characters because of the utility of knowing where the feet are with respect to the ground and the potential complexity of the forward kinematics computation needed to compute this. Here we investigate the use of Cartesian joint position for learning physics-driven locomotion. The Cartesian joint positions are represented in a Cartesian coordinate system centered at the root of the character.

We train with state representations that are augmented with Cartesian joint position, for the 3D Humanoid and Walker2D environments, and compare them against the results trained with the original state representations. As shown in Figure 3, the Humanoid learns faster with the augmented state representation, and yields the same final performance. The augmented state representation has negligible benefits for the Walker2D environment.

**Summary:** Adding Cartesian joint position to the state improves learning speed.

### 4.3 Contact Information

The PyBullet default state includes the binary valued contact state information about link contacts with the ground. The other popular physics simulator for locomotion environment, Mujoco [Todorov et al. 2012], does not include contact link variables. We ask the question: Is contact information really useful? The experimental results for HalfCheetah, Walker2D, Hopper and Ant environments are shown in Figure 4. For Ant, removing contact information from the state affects the final reward, making it worse, as this environment has a relatively high number of points of contact with the ground (six). For Hopper, Walker2D and HalfCheetah, one, two and four points of contact respectively, the final return is not affected. Other ways of representing contact information, e.g., contact forces, could produce different outcomes.

**Summary:** Adding binary contact information to state variable can be helpful in some cases, but may have negligible benefits in other cases.

### 4.4 Pre-trained Representations

When the observation given by the environment is high-dimensional, pixel-based for example, it is typical to learn a more compact representation that is then used for control, whether end-to-end, through pre-training or in a 2 step procedure. When a compact state is provided, in principle a neural network transforms multiple time the state to build, layer after layer, a increasingly more suitable representation for control. In this scenario, the representation immediately
Learning to Locomote: Understanding How Environment Design Matters for Deep Reinforcement Learning

Figure 5: Learning with a pre-trained state representation.

before the output layer is the most suited. Inspired by the usage of pre-trained networks, we applied this to the original compact state of Ant, HalfCheetah, Hopper and Walker2D environments.

After training a full end-to-end policy with TD3, we take the policy network and remove the output layer. We obtain a network that transforms the compact state representation into a representation of the size of the last hidden layer, and we use this as input representation of our new experiment. Interestingly, shown in Figure 5, we discover that this setup provides a sub-optimal representation, which learns as efficiently as the original state initially, but plateaus earlier. This is surprising, given that the similar approach of using a pre-trained representation is widely used with pixel-based states, but explainable by the fact that the final trained policy used as representation has steered towards the relevant part of the state space and so is unable to represent properly more exploratory states.

Summary: Unlike the success of pre-trained network in computer vision tasks, the pre-trained policy without the last layer does not provide a state representation easier to learn from than the raw state variable.

5 CONTROL FREQUENCY

The choice of action repeat (AR), also called frame skip, has been fundamental to early work in DQN [Mnih et al. 2015] and it has been overlooked as a simple hyperparameter in many other algorithms. This relevant domain knowledge consists of repeating each action selected by the agent a fixed number of times. This effectively lowers the frequency at which the agent operates, compared to the environment frequency. The impact of the action repeat parameter has been studied by [Hessel et al. 2019], in which they train an adaptive agent that tunes the action repetition. In [Lakshminarayanan et al. 2017], the authors propose a simple-but-limited approach that improves DQN. The policy outputs both an action and an action-repeat value, which is selected from one of two possible values. Tuning these (hyperparameter) values is highly beneficial to the learning. More recently, Metelli et al. [2020] also explore the choice of control frequency and action persistency for low-D continuous environments, such as CartPole and MountainCar.

We systematically examine the effect of action repeat in common PyBullet locomotion environments. The environment frequency is fixed and given by the simulator. With $AR = 1$, the control frequency and the environment frequency are equal, and a gradient step is computed after each control step. By choosing a different action repeat $AR$ value, we are scaling down the control frequency and taking a gradient step after $AR$ environment steps. The learning curves are shown in Figure 6. As expected, the choice of action repeat has a significant impact on the learning. For the Walker and Hopper, $AR=1$ produces the best performance, with larger values producing worse performance. This also leaves open the possibility that a higher frequency control rate could improve learning. Humanoid learns best with $AR=3$ or $AR=4$, and Ant with $AR=2$. An exception is HalfCheetah, which is not particularly sensitive to the choice of action repeat.

Summary: A good choice of control frequency, usually implemented as action repeat parameter for RL, is essential for good learning performance. Controlling the actions at frequencies that are too high or too low is usually harmful.

Figure 6: Impact of Action Repeat.

6 EPISODE TERMINATION

Locomotion environments such as HalfCheetah are constrained with a time limit that defines the maximum number of steps per episode. The characters continue their exploration until this time limit is reached. In both PyBullet [Coumans and Bai 2019] and MuJoCo [Todorov et al. 2012] versions of these environments, the default time limit is 1000. In some environments, such as Ant, Hopper, Walker2D, and Humanoid, an additional natural terminal condition
which corresponds to considering the time limit termination as a non-terminal.

Here $l_{\text{term}} \in \{0, 1\}$ is an indicator variable, set to 0 for any terminal transition, and is otherwise 1, i.e., for non-terminal transitions. With infinite bootstrapping, we use $l_{\text{term}} = 1$ for time limit terminations, which corresponds to considering the time limit termination as being non-terminal.

Our experiments with TD3 are shown in Figure 7. These show significant improvements for this infinite bootstrapping trick, in particular for Ant and HalfCheetah, where the time limit termination is reached more readily. This finding is consistent with Pardo et al. [2018] experiments with PPO [Schulman et al. 2017]. Given that it plays an important role in the final policy and the return value, and that it is often overlooked or left undocumented, we list it as one of the factors that matters for learning a locomotion policy. Like the choice of discount factor, small changes can play an important role in the many environments that do not have a natural limited duration.

Summary: Infinite bootstrap is critical for accurately optimizing the action-value function, which usually corresponds to high-quality motion.

7 CURRICULUM LEARNING

Curriculum-based learning can enable faster learning by progressively increasing the task difficulty throughout the learning process. These strategies have been applied to supervised learning [Bengio et al. 2009] and are becoming increasingly popular in the RL setting, e.g., [Florensa et al. 2017; Narvekar and Stone 2019; Xie et al. 2020; Yu et al. 2018] and many of which are summarized in a recent survey [Narvekar et al. 2020]. The curriculum therefore serves as a critical source of inductive bias, particularly as the task complexity grows. The important problem of curriculum generation is being tackled from many directions, including the use of RL to learn curriculum-generating policies. These still face the problem that learning a full curriculum policy can take significantly more experience data than learning the target policy from scratch [Narvekar and Stone 2019].

Summary: By training on task environments with progressively increasing difficulty, curriculum learning enables learning locomotion skills that would otherwise be very difficult to learn.

8 CHOICE OF ACTION SPACE

In most locomotion benchmarks, torque is the dominant actuation model used to drive the articulated character. In contrast, low-level stable Proportional-Derivative (PD) controller are used in a variety of recent results from animation and robotics, e.g., [Peng et al. 2017; Tan et al. 2011; Xie et al. 2019]. The PD controller takes in a target joint angle $\theta$ as input and outputs torque $\tau$ according to:

$$\tau = -k_p (\theta - \bar{\theta}) - k_d \dot{\theta}$$

where $\theta$ and $\dot{\theta}$ represent the current joint angles and velocities, and $k_p$ and $k_d$ are manually defined. With a low-level PD controller, the policy produces joint angles rather than torques. In previous motion imitation work, e.g., [Chentanez et al. 2018; Peng et al. 2018; Xie et al. 2019], the reference trajectory provides default values for the PD-target angles, and the policy then learns a residual offset. However, this PD-residual policy is infeasible for non-imitation tasks since a reference trajectory is not available. Here we study the impact of using (non-residual) PD-action spaces for purely objective-driven (vs imitation-driven) locomotion tasks. For implementation and training details on the PD controller, we direct the reader to appendix B.

Training with low-level PD controllers is more prone to local minimum than the same environment with torque-based RL policy. This is because it is relatively easy to learn a constant set of PD-targets which maintain a static standing posture, which is rewarded via the survival bonus. To address this local-minimum issue, we first train with a broader initial state distribution, and then, after mastering a basic gait, the agent is trained with the original environment with a nearly-deterministic initial-state. Among the PyBullet locomotion benchmarks, Walker2D suffers more severely from the problem of a local minimum.

As shown in Figure 8, using PD target angle as action space often leads to a faster early learning rate. However, it may converge to lower-reward solutions than the torque action space, and it may hinder quick adaption to rapid changes in the state as the character moves faster [Chentanez et al. 2018].
Learning to Locomote: Understanding How Environment Design Matters for Deep Reinforcement Learning

Previous work [Peng and van de Panne 2017] shows that PD-control-targets outperform torques as a choice of action space in terms of both final reward and sampling efficiency. The PD-control action space is far better at motion imitation tasks than learning locomotion from scratch. We believe this arises for two reasons. First, in motion imitation, the reference trajectory already serves as a good default, and the policy only needs to fine-tune the PD target, which is simpler than learning the PD target from the scratch. Second, current locomotion benchmarks are designed and optimized for direct torque control rather than low-level PD control, including the state initialization and the reward function.

Summary: For non-imitation locomotion tasks, PD-control-targets actions appears to have a limited advantage over a torque-based control strategy. However, PD-control-targets often learn a basic gait more efficiently during the early stages of training.

9 SURVIVAL BONUS

The reward function plays an obvious and important role in learning natural and fluid motion. In PyBullet locomotion benchmarks, the character is usually rewarded for moving forward with a positive velocity, and penalized for control costs, unnecessary collisions, and approaching the joint limit. Additionally, the reward function also contains a survival bonus term, which is a positive constant when the character has not fallen; otherwise it is negative or zero. In Henderson et al. [2018] and Mania et al. [2018], the existence of the survival bonus terms can lead to getting stuck in local minima for certain types of algorithms. Here we investigate the impact of the value of the survival bonus in the reward function.

We experiment with 3 different values of survival bonus: 0, 1 and 5, and train the agent with TD3 algorithm. To allow for a fair comparison among policies trained with different survival bonus value, the policies are always tested on the environments without the survival bonus reward. The default PyBullet environment set the value of survival bonus to 1, for which the episodic test reward significantly outperforms the other two tested values for the survival bonus, as shown in Figure 9. Including the survival bonus term in the reward function is necessary; setting it to 0 makes the discovery of a basic walking gait too difficult. If the survival bonus term is too large, however, the algorithm exploits the survival bonus reward while neglecting other reward terms. This results in a character that balances but never steps forward.

Summary: The survival bonus value provides a critical form of reward shaping when learning to locomote. Values that are too small or too large leads to local minima corresponding to falling-forward and standing still, respectively.

Figure 9: Impact of survival bonus (SB) on Walker2DBulletEnv, evaluated for SB = {0,1,5}

10 TORQUE LIMIT

The capabilities of characters are defined in part by their torque limits. Both high and low torque limits can be harmful to learning locomotion in simulation or actual robots. A high torque limit is prone to unnatural behavior. On the other hand, a low torque limit is also problematic since it can make it much more difficult to discover the solution modes that yield efficient and natural locomotion. In Abdolhosseini [2019], the authors experiment with the impact of torque limits by multiplying the default limits by a scalar multiplier \( \lambda \in [0.6, 1.6] \). Policies are trained using PPO [Schulman et al. 2017]. The observed result is that higher torque limits obtain higher episodic rewards. As \( \lambda \) decreases, the agent is more likely to get stuck at local minimum, and the agent fails to walk completely when the multiplier is lower than a threshold [Abdolhosseini 2019]. Such results motivate the idea of apply a torque limit curriculum during training, as a form of continuation method. This allows for large torques early in the learning process, to allow for the discovery of good solution modes, followed by progressively decreasing torque limits, which then allows the optimization to find more natural, low torque motions within the good solution modes.
Summary: Large torque limits benefit the exploration needed to find good locomotion modes, while low torque limits benefit natural behavior. A torque limit curriculum can allow for both.

11 CONCLUSIONS

Reinforcement learning has enormous potential as means of developing physics-based movement skills for animation, robotics, and beyond. Successful application of RL requires not only efficient RL algorithms, but also the design of suitable RL environments. In this paper we have investigated a number of issues and design choices related to RL environments, and we evaluate these on locomotion tasks. As a caveat, we note that these results have been applied to fairly simple benchmark systems, and thus more realistic and complex environments may yield different results. Furthermore, we evaluate these environment design choices using a specific RL algorithm (TD3). Other RL algorithms may be impacted differently, and we leave that as future work. However, our work provides a better understanding of the often brittle-and-unpredictable nature of RL solutions, as bad choices made with regard to defining RL environments quickly become problematic. Efficient learning of locomotion skills in humans and animals can arguably be attributed in large part to their “RL environment”. For example, Underlying reflex movements and central pattern generators help constrain the state distribution, provide a suitable action space, and are tuned to control at a particular time scale.

Many environment design issues can be viewed as a form of inductive bias, e.g., the survival bonus for staying upright, or a given choice of action space. We anticipate that much of the progress needed for efficient learning of motion skills will require leveraging the many aspects of RL that are currently precluded by the canonical environment-and-task structure that is reflected in common RL benchmarks. As such, state-of-the-art RL-based approaches for locomotion should be inspired by common RL algorithms and RL benchmarks, but should not be constrained by them.

REFERENCES

Farzad Abdolhosseini. 2019. Learning locomotion: symmetry and torque limit considerations. https://doi.org/10.14288/1.0383251
Farzad Abdolhosseini, Hung Yu Ling, Zhaoming Xie, Xue Bin Peng, and Michiel van de Panne. 2019. On Learning Symmetric Locomotion. In Proc. ACM SIGGRAPH Motion, Interaction, and Games (MIG 2019).
Marcin Andrychowicz, Austin Rabinchuk, Piotr Sankowski, Manu Srivastava, Sertan Girgin, and Jeffery Hawke. 2016. PyBullet, a Python module for physics simulations. In Eurographics Symposium on Computer Animation (MIG '18).
Mimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills. ACM Trans. Graph. 37, 4 (July 2018), 14 pages. https://doi.org/10.1145/3197571.3203111
Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. 2018. DeepMimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills. In Proc. ACM SIGGRAPH 2018. https://doi.org/10.1145/3197571.3203111
Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curricula learning. In Proceedings of the 26th annual international conference on machine learning 41–48.
Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540
Nuttapong Chentanez, Matthias Müller, Miles Macklin, Viktor Machovikychuk, and Stefan Jeschke. 2018. Physics-Based Motion Capture Imitation with Deep Reinforcement Learning. In Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games (Limassol, Cyprus) (MIG ’18). Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3274247.3274506
Erwin Coumans and Yunfei Bai. 2016–2019. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.
Logan Engstrom, Andrew Ilyas, Shihab Sanjuro, Dimitris Tsipras, Firdaus Janoo, Larry Rudolph, and Aleksander Madry. 2020. Implementation Matters in Deep RL. A Case Study on FPO and TRPO. In International Conference on Learning Representations. https://openreview.net/forum?id=rIetN1rtPB
Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. 2017. Reverse Curriculum Generation for Reinforcement Learning. In Conference on Robot Learning, 482–495.
Scott Fujimoto, Herke Van Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. arXiv preprint arXiv:1802.09477 (2018).
Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. 2018. Divide-and-Conquer Reinforcement Learning. In International Conference on Learning Representations. https://openreview.net/forum?id=J1WelmBR
Jeffrey Hawke, Richard Shen, Corina Gurau, Siddharth Sharma, Daniele Reda, Nikolay Nikolov, Prenziaylav Mazur, Sean Middelheuwe, Nicolas Griffiths, Amir Shah, et al. 2020. Urban Driving with Conditional Imitation Learning. International Conference on Robotics and Automation (2020).
Peter Henderson, Riahsat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In Thirty-Second AAAI Conference on Artificial Intelligence.
Matteo Hessel, Hado van Hasselt, Joseph Modayil, and David Silver. 2019. On inductive biases in deep reinforcement learning. arXiv preprint arXiv:1907.02980 (2019).
Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned neural networks for character control. ACM Transactions on Graphics (TOG) 36, 4 (2017), 1–13.
Aravind S Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. 2017. Dynamic action repetition for deep reinforcement learning. In Thirty-First AAAI Conference on Artificial Intelligence.
Horia Mania, Aurelia Gy, and Benjamin Recht. 2018. Simple random search provides a competitive approach to reinforcement learning. arXiv preprint arXiv:1803.07055 (2018).
Alberto Maria Metelli, Flavio Mazzolino, Lorenzo Bisi, Luca Sabbioni, and Marcello Restelli. 2020. Control Frequency Adaptation via Action Persistence in Batch Reinforcement Learning. arXiv:2002.06836 [cs.LG]
Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc C Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al 2015. Human-level control through deep reinforcement learning. Nature 518, 7540 (2015), 529–533.
Sammit Narkekar, Bei Peng, Matteo Leonetti, Jirko Sinapov, Matthew E. Taylor, and Peter Stone. 2020. Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. arXiv preprint arXiv:2003.04960 (2020).
Sammit Narkekar and Peter Stone. 2019. Learning curriculum policies for reinforcement learning. In Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. International Foundation for Autonomous Agents and Multiagent Systems, 25–33.
Charles Packer, Katelyn Gao, Jernei Kos, Philipp Krabenbühl, Vladlen Koltun, and Dawn Song. 2018. Assessing generalization in deep reinforcement learning. arXiv preprint arXiv:1810.12282 (2018).
Fabio Pardo, Arash Tavakoli, Vitaly Levdi, and Petar Kormushev. 2018. Time Limits in Reinforcement Learning. In Proc. 35th International Conference on Machine Learning (ICML 2018).
Soochwan Park, Hoseok Ryu, Seyoung Lee, Summin Lee, and Jehee Lee. 2019. Learning Predict-and-Simulate Policies From Unorganized Human Motion Data. ACM Trans. Graph. 38, 6, Article 205 (2019).
Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. 2018. DeepMimic Example-guided Deep Reinforcement Learning of Physics-based Character Skills. ACM Trans. Graph. 37, 4, Article 143 (July 2018), 14 pages. https://doi.org/10.1145/3197571.3203111
Xue Bin Peng, Glen Berseth, Kangtu Yin, and Michiel van de Panne. 2017. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. ACM Transactions on Graphics (SIGGRAPH 2017) 36, 4 (2017).
Xue Bin Peng and Michiel van de Panne. 2017. Learning Locomotion Skills Using DeepRL: Does the Choice of Action Space Matter?. In Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation.
Marc Ponsen, Matthew E Taylor, and Karl Tuyt. 2009. Abstraction and generalization in reinforcement learning: A summary and framework. In International Workshop on Adaptive and Learning Agents, Springer, 1–32.
Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. 2017. Towards generalization and simplicity in continuous control. In Advances in Neural Information Processing Systems. 6550–6561.
John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]
Scott Fujimoto, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. Jie Tan, Karen Liu, and Greg Turk. 2011. Stable proportional–divergence controllers. IEEE Computer Graphics and Applications 31, 4 (2011), 34–44.
Emanuel Todorov, Tom Erz, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 5026–5033.
Zhaoxing Xie, Patrick Clary, Jeremy Dao, Pedro Morais, Jonathan Hurst, and Michiel van de Panne. 2019. Learning Locomotion Skills for Cassie: Iterative Design and Sim-to-Real. In Proc. Conference on Robot Learning (CORL 2019).
Zhaoxing Xie, Hung Yu Ling, and Michiel van de Panne. 2020. ALL-STEPS: Curriculum-driven Learning of Stepping Stone Skills. ArXiv preprint arXiv:2005.04323 (2020). arXiv:2005.04323 [cs.GR]
A HYPERPARAMETERS

| Hyperparameter          | Value       |
|-------------------------|-------------|
| Critic Learning Rate    | $10^{-3}$   |
| Critic Regularization   | None        |
| Actor Learning Rate     | $10^{-3}$   |
| Actor Regularization    | None        |
| Optimizer               | Adam        |
| Target Update Rate ($\tau$) | $5 \cdot 10^{-3}$ |
| Batch Size              | 100         |
| Iterations per time step| 1           |
| Discount Factor         | 0.99        |
| Reward Scaling          | 1.0         |
| Normalized Observations | False       |
| Gradient Clipping       | False       |
| Exploration Policy      | $N(0, 0.1)$ |

Table 2: Hyperparameters used for training the TD3 algorithm.

A.1 TD3 Actor Architecture

```python
nn.Linear(state_dim, 256)
nn.ReLU
nn.Linear(256, 256)
nn.ReLU
nn.Linear(256, action_dim)
nn.tanh
```

A.2 TD3 Critic Architecture

```python
nn.Linear(state_dim + action_dim, 256)
nn.ReLU
nn.Linear(256, 256)
nn.ReLU
nn.Linear(256, 1)
```

B IMPLEMENTATION DETAILS OF PD CONTROLLER

The PD controller is commonly implemented as an action space wrapper over the original torque based control. To produce stable simulation results, the PyBullet simulator runs at 1200 Hz. Every four substeps, the simulator will receive a torque command to update the dynamics of the character. For each PD target command, the PD controller will convert it to torque iteratively using Equation 1 for 5 times such that the low level PD controller runs at 300Hz. The control policy producing PD target angles runs at the same control frequency as the torque policy, 60Hz. The actuator outputs the PD target angles as a vector ranged from $[0, 1]$ with the same dimension as the actuator, and then each value from 0 to 1 is mapped to the range bounded by torque limits of each joint. To obtain a fair comparison with the original environment, the energy cost term and the cost for approaching joint limits term are integrated over the 5 substeps.

To better avoid local minimum, we set the exploration noise as a Gaussian distribution $N(0, 0.2)$, except for Walker2D environment where the agent is trained on environment with broader initial-state distribution ($\kappa=0.3$) for 180K interactions before trained on the default environment.

C STATE DEFINITION

Here we describe the full state definition for the PyBullet environments used in experiments.

A target position $(1000, 0, 0)$ is defined 1000 units away from the root $(0, 0, 0)$. The default state description in PyBullet environments is composed by the concatenation of the following variables:

- change in $z$ coordinate in the world frame, computed as $z_t - z_0$ where $z_t$ is the current position on the $z$ axis and $z_0$ is the initial position (1 real number);
- $\sin$ and $\cos$ of the angle between the character’s position and the target position (2 real numbers);
- linear velocity of the character with respect to the world frame (3 real numbers);
- angle and angular velocity of each joint ($2 \cdot n_{\text{joints}}$ real numbers, where $n_{\text{joints}}$ is the number of joints, specified in table);
- contact information for each foot ($n_{\text{feet}}$ binary values, where $n_{\text{feet}}$ is described for each environment in the table).

| Environment Name          | $n_{\text{joints}}$ | $n_{\text{feet}}$ |
|---------------------------|---------------------|-------------------|
| HalfCheetahBulletEnv      | 6                   | 6                 |
| AntBulletEnv              | 8                   | 4                 |
| HopperBulletEnv           | 3                   | 1                 |
| Walker2DBulletEnv         | 6                   | 2                 |
| HumanoidBulletEnv         | 17                  | 2                 |

Table 3: Dimension of the joints and limbs for different characters.