Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning

Nikita Rudin
ETH Zurich and NVIDIA
rudinn@ethz.ch

David Hoeller
ETH Zurich and NVIDIA
dhoeller@ethz.ch

Philipp Reist
NVIDIA
preist@nvidia.com

Marco Hutter
ETH Zurich
mahutter@ethz.com

Abstract: In this work, we present and study a training set-up that achieves fast policy generation for real-world robotic tasks by using massive parallelism on a single workstation GPU. We analyze and discuss the impact of different training algorithm components in the massively parallel regime on the final policy performance and training times. In addition, we present a novel game-inspired curriculum that is well suited for training with thousands of simulated robots in parallel. We evaluate the approach by training the quadrupedal robot ANYmal to walk on challenging terrain. The parallel approach allows training policies for flat terrain in under four minutes, and in twenty minutes for uneven terrain. This represents a speedup of multiple orders of magnitude compared to previous work. Finally, we transfer the policies to the real robot to validate the approach. We open-source our training code to help accelerate further research in the field of learned legged locomotion: https://leggedrobotics.github.io/legged_gym/.

Keywords: Reinforcement Learning, Legged Robots, Sim-to-real

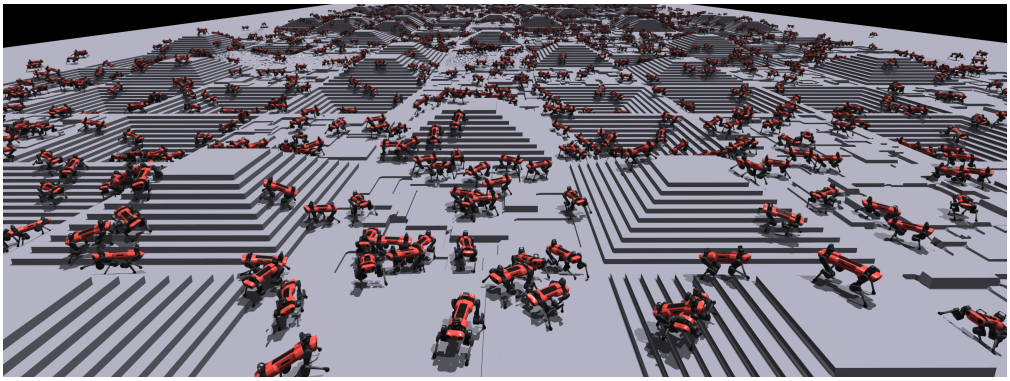


Figure 1: Thousands of robots learning to walk in simulation.

1 Introduction

Deep reinforcement learning (DRL) is proving to be a powerful tool for robotics. Tasks such as legged locomotion [1], manipulation [2], and navigation [3], have been solved using these new tools, and research continues to keep adding more and more challenging tasks to the list. The amount of data required to train a policy increases with the task complexity. For this reason, most work focuses on training in simulation before transferring to real robots. We have reached a point where multiple days or even weeks are needed to fully train an agent with current simulators. For example, OpenAI's block reorientation task was trained for up to 14 days and their Rubik's cube solving policy took several months to train [4]. The problem is exacerbated by the fact that deep

reinforcement learning requires hyper-parameter tuning to obtain a suitable solution which requires sequentially rerunning time-consuming training. Reducing training times using massively parallel approaches such as presented here can therefore help improve the quality and time-to-deployment of DRL policies, as a training setup can be iterated on more often in the same time frame.

In this paper, we examine the effects of massive parallelism for on-policy DRL algorithms and present considerations in how the standard RL formulation and the most commonly used hyper-parameters should be adapted to learn efficiently in the highly parallel regime. Additionally, we present a novel game-inspired curriculum which automatically adapts the task difficulty to the performance of the policy. The proposed curriculum architecture is straightforward to implement, does not require tuning, and is well suited for the massively parallel regime. Common robotic simulators such as Mujoco [5], Bullet [6], or Raisim [7] feature efficient multi-body dynamics implementations. However, they have been developed to run on CPUs with only a reduced amount of parallelism. In this work, we use NVIDIA's Isaac Gym simulation environment [8], which runs both the simulation and training on the GPU and is capable of simulating thousands of robots in parallel.

The massively parallel training regime has been explored before [4, 9] in the context of distributed systems with a network of thousands of CPUs each running a separate instance of the simulation. The parallelization was achieved by averaging the gradients between the different workers without reducing the number of samples provided by each agent. This results in large batch sizes of millions of samples for each policy update which improves the learning dynamics, but does not optimize the overall training time. In parallel, recent works have aimed to increase the simulation throughput and reduce training times of standard DRL benchmark tasks. A framework combining parallel simulation with multi-GPU training [10] was proposed to achieve fast training using hundreds of parallel agents. In the context of visual navigation, large batch simulation has been used to increase the training throughput [11]. Furthermore, GPU accelerated physics simulation has been shown to significantly improve the training time of the Humanoid running task [12]. A differentiable simulator running on Google's TPUs has also been shown to greatly accelerate the training of multiple tasks [13]. We build upon [10, 12] by pushing the parallelization further, optimizing the training algorithm, and applying the approach to a challenging real-world robotics task.

Perceptive and dynamic locomotion for legged robots in unstructured environments is a demanding task that, until recently, had only been partially demonstrated with complex model-based approaches [14, 15]. Learning-based approaches are emerging as a promising alternative. For quadrupeds, DRL has been used to train blind policies robust to highly uneven ground [16] (12 hours of training). Perceptive locomotion over challenging terrain has been achieved by combining learning with optimal control techniques [17, 18] (82 and 88 hours of training) and recently, a fully learned approach has shown great robustness in this setting [19] (120 hours of training). Similarly, bipedal robots have also been trained to walk blindly on stairs [20] (training time not reported). With our approach we can train a perceptive policy in under 20 minutes on a single GPU, with the complexity of sim-to-real transfer to the hardware, which increases the performance and robustness requirements and provides clear validation of the overall approach. Training such behaviors in minutes opens up new exciting possibilities ranging from automatic tuning to customized training using scans of particular environments.

2 Massively Parallel Reinforcement Learning

Current (on-policy) reinforcement learning algorithms are divided into two parts: data collection and policy update. The policy update, which corresponds to back-propagation for neural networks, is easily performed in parallel on the GPU. Parallelizing data collection is not as straightforward. Each step consists of policy inference, simulation, reward, and observation calculation. Current popular pipelines have the simulation and reward/observation calculation computed on the CPU, making the GPU unsuitable for policy inference because of communication bottle-necks. Data transfer over PCIe is known to be the weakest link of GPU acceleration, and can be as much as 50 times slower than the GPU processing time alone [21]. Furthermore, with CPU data collection, a large amount of data must be sent to the GPU for each policy update, slowing down the overall process. Limited parallelization can be achieved by using multiple CPU cores and spawning many processes, each running the simulation for one agent. However, the number of agents is quickly limited by the number of cores and other issues such as memory usage. We explore the potential of massive parallelism

with Isaac Gym's end-to-end data collection and policy updates on the GPU, significantly reducing data copying and improving simulation throughput.

2.1 Simulation Throughput

The main factor affecting the total simulation throughput is the number of robots simulated in parallel. Modern GPUs can handle tens of thousands of parallel instructions. Similarly, IsaacGym's PhysX engine can process thousands of robots in a single simulation and all other computations of our pipeline are vectorized to scale favorably with the number of robots. Using a single simulation with thousands of robots presents some new challenges. For example, a single common terrain mesh must be used, and it cannot be easily changed at each reset. We circumvent this problem by creating the whole mesh with all terrain types and levels tiled side by side. We change the terrain level of the robots by physically moving them on the mesh. In supplementary material, we show the computational time of different parts of the pipeline, examine how these times scale with the number of robots, and provide other techniques to optimize the simulation throughput.

2.2 DRL Algorithm

We build upon a custom implementation of the Proximal Policy Optimization (PPO) algorithm [22]. Our implementation is designed to perform every operation and store all the data on the GPU. In order to efficiently learn from thousands of robots in parallel, we perform some essential modifications to the algorithm and change some of the commonly used hyper-parameter values.

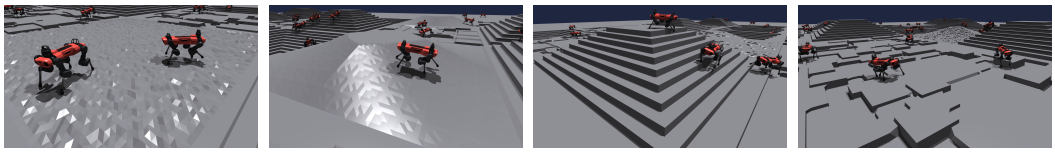
2.2.1 Hyper-Parameters Modification

In an on-policy algorithm such as PPO, a fixed policy collects a selected amount of data before doing the next policy update. This *batch size, B*, is a crucial hyper-parameter for successful learning. With too little data, the gradients will be too noisy, and the algorithm will not learn effectively. With too much data, the samples become repetitive, and the algorithm cannot extract more information from them. These samples represent wasted simulation time and slow down the overall training. We have $B = n_{robots} n_{steps}$, where n_{steps} is the number of steps each robot takes per policy update and n_{robots} the number of robots simulated in parallel. Since we increase n_{robots} by a few orders of magnitude, we must choose a small n_{steps} to keep B reasonable and hence optimize training times, which is a setting that has not been extensively explored for on-policy reinforcement learning algorithms. It turns out that we can not choose n_{steps} to be arbitrarily low. The algorithm requires trajectories with coherent temporal information to learn effectively. Even though, in theory, information of single steps could be used, we find that the algorithm fails to converge to the optimal solution below a certain threshold. This can be explained by the fact that we use Generalized Advantage Estimation (GAE) [23], which requires rewards from multiple time steps to be effective. For our task, we find that the algorithm struggles when we provide fewer than 25 consecutive steps, corresponding to $0.5\,\mathrm{s}$ of simulated time. It is important to distinguish n_{steps} from the maximum episode length leading to a time-out and a reset, which we define as $20\,\mathrm{s}$. The environments are reset when they reach this maximum length and not after each iteration, meaning that a single episode can cover many policy updates. This limits the total number of robots training in parallel, and consequently, prohibits us from using the full computational capabilities of the GPU.

The *mini-batch size* represents the size of the chunks in which the *batch size* is split to perform back-propagation. We find that having mini-batch sizes much larger than what is usually considered best practice is beneficial for our massively parallel use case. We use mini-batches of tens of thousands of samples and observe that it stabilizes the learning process without increasing the total training time.

2.2.2 Reset Handling

During training, the robots must be reset whenever they fall, and also after some time to keep them exploring new trajectories and terrains. The PPO algorithm includes a *critic* predicting an infinite horizon sum of future discounted rewards. Resets break this infinite horizon assumption and can lead to inferior critic performance if not handled carefully. Resets based on failure or reaching a goal are not a problem because the critic can predict them. However, a reset based on a time out can not be predicted (we do not provide episode time in the observations). The solution is to distinguish



(a) (b) (c) (d)

Figure 2: Terrain types used for training and testing in simulation. (a) Randomly rough terrain with variations of $0.1\,\mathrm{m}$. (b) Sloped terrain with an inclination of $25\,\mathrm{deg}$. (c) Stairs with a width of $0.3\,\mathrm{m}$ and height of $0.2\,\mathrm{m}$. (d) Randomized, discrete obstacles with heights of up to $\pm 0.2\,\mathrm{m}$.

the two termination modes and augment the reward with the expected infinite sum of discounted future rewards in a time-out case. In other words, we bootstrap the target of the critic with its own prediction. This solution has been discussed in [24], but interestingly, this distinction is not part of the widely used *Gym* environment interface [25] and is ignored by popular implementations such as *Stable-Baselines* [26][1]. After investigating multiple implementations, we conclude that this important detail is often avoided by assuming that the environments either never time out or only on the very last step of a batch collection. In our case, with few robot steps per batch, we can not make such an assumption since a meaningful episode length covers the collection of many batches. We modify the standard *Gym* interface to detect time-outs and implement the bootstrapping solution. In supplementary material, we show the effect of this solution on the total reward as well as the critic loss.

3 Task Description

A quadruped robot must learn to walk across challenging terrain, including uneven surfaces, slopes, stairs, and obstacles, while following base-heading and linear-velocity commands. We conduct most of the simulation and real-world deployment experiments on the ANYbotics ANYmal C robot. However, in simulation, we demonstrate the broader applicability of the approach by additionally training policies for ANYmal B, ANYmal C with an attached arm, and the Unitree A1 robots.

3.1 Game-Inspired Curriculum

The terrains are selected to be representative of real-world environments. We create five types of procedurally generated terrains presented in Fig. 2: flat, sloped, randomly rough, discrete obstacles, and stairs. The terrains are tiled squares with 8m sides. The robots start at the center of the terrain and are given randomized heading and velocity commands (kept constant for the duration of an episode) pushing them to walk across the terrain. Slopes and stairs are organized in pyramids to allow traversability in all directions.

Previous works have shown the benefits of using an automated curriculum of task difficulty to learn complex locomotion policies [28, 29, 16]. Similarly, we find that it is essential to first train the policy on less challenging terrain before progressively increasing the complexity. We adopt a solution inspired by [16], but replace the particle filter approach with a new game-inspired automatic curriculum. All robots are assigned a terrain type and a level that represents the difficulty of that terrain. For stairs and randomized obstacles, we gradually increase the step height from $5\,\mathrm{cm}$ to $20\,\mathrm{cm}$. Sloped terrain inclination is increased from $0\,\mathrm{deg}$ to $25\,\mathrm{deg}$. If a robot manages to walk past the borders of its terrain, its level is increased, and at the next reset, it will start on more difficult terrain. However, if at the end of an episode it moved by less than half of the distance required by its target velocity, its level is reduced again. Robots solving the highest level are looped back to a randomly selected level to increase the diversity and avoid catastrophic forgetting. This approach has the advantage of training the robots at a level of difficulty tailored to their performance without requiring any external tuning. It adapts the difficulty level for each terrain type individually and provides us with visual and quantitative feedback on the progress of the training. When the robots have reached the final level and are evenly spread across all terrains due to looping back, we can conclude they have fully learned to solve the task.

[1] The *Spinning-up* [27] implementation of PPO uses the same bootstrapping solution by keeping track of episode lengths within the algorithm, thus circumventing the limitation of the *Gym* interface.

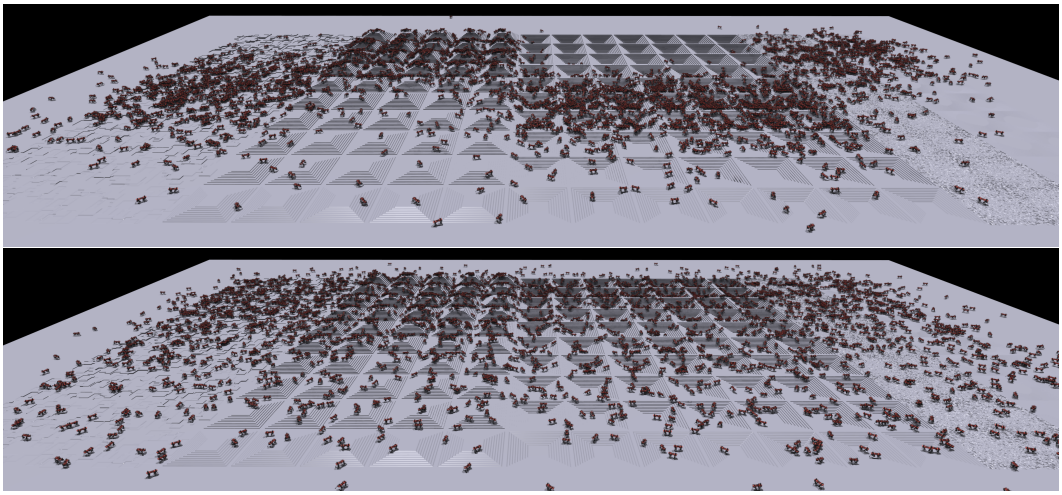


Figure 3: 4000 robots progressing through the terrains with automatic curriculum, after 500 (top) and 1000 (bottom) policy updates. The robots start the training session on the first row (closest to the camera) and progressively reach harder terrains.

The proposed curriculum structure is well suited for the massively parallel regime. With thousands of robots we can directly use their current progress in the curriculum as the distribution of the policy's performance, and do not need learn it with a generator network [30]. Furthermore, our method doesn't require tuning and is straightforward to implement in a parallel manner with near-zero processing cost. We remove the computational overhead of re-sampling and re-generating new terrains needed for the particle filter approach.

Fig. 3 shows robots progressing through the terrains at two different stages of the training process. On complex terrain types, the robots require more training iterations to reach the highest levels. The distribution of robots after 500 iterations shows that while the policy is able to cross sloped terrains and to go down stairs, climbing stairs and traversing obstacles requires more training iterations. However, after 1000 iterations, the robots have reached the most challenging level for all terrain types and are spread across the map. We train for a total for 1500 iterations to let the policy converge to its highest performance.

3.2 Observations, Actions, and Rewards

The policy receives proprioceptive measurements of the robot as well as terrain information around the robot's base. The observations are composed of: base linear and angular velocities, measurement of the gravity vector, joint positions and velocities, the previous actions selected by the policy, and finally, 108 measurements of the terrain sampled from a grid around the robot's base. Each measurement is the distance from the terrain surface to the robot's base height.

The total reward is a weighted sum of nine terms, detailed in supplementary material. The main terms encourage the robot to follow the commanded velocities while avoiding undesired base velocities along other axes. In order to create a smoother, more natural motion, we also penalize joint torques, joint accelerations, joint target changes, and collisions. Contacts with the knees, shanks or between the feet and a vertical surface are considered collisions, while contacts with the base are considered crashes and lead to resets. Finally, we add an additional reward term encouraging the robot to take longer steps, which results in a more visually appealing behavior. We train a single policy with the same rewards for all terrains.

The actions are interpreted as desired joint positions sent to the motors. There, a PD controller produces motor torques. In contrast to other works [16, 20], neither the reward function nor the action space has any gait-dependent elements.

3.3 Sim-to-Real Additions

In order to make the trained policies amenable for sim-to-real transfer, we randomize the friction of the ground, add noise to the observations and randomly push the robots during the episode to

teach them a more stable stance. Each robot has a friction coefficient sampled uniformly in [0.5, 1.25]. The pushes happen every $10\,\mathrm{s}$. The robots' base is accelerated up to $\pm 1\,\mathrm{m/s}$ in both x and y directions. The amount of noise is based on real data measured on the robot and is detailed in supplementary material.

The ANYmal robot uses series elastic actuators with fairly complex dynamics, which are hard to model in simulation. For this reason and following the methodology of previous work [1], we use a neural network to compute torques from joint position commands. However, we simplify the inputs of the model. Instead of concatenating past measurements at fixed time steps and sending all of that information to a standard feed-forward network, we only provide the current measurements to an LSTM network. A potential drawback of this set-up is that the policy does not have the temporal information of the actuators as in previous work. We have experimented with various ways of providing that information through memory mechanisms for the policy but found that it does not improve the final performance.

4 Results

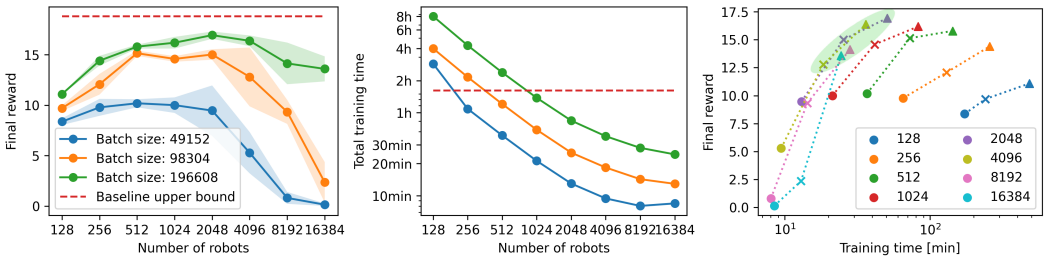
4.1 Effects of Massive Parallelism

In this section, we study the effects of the number of parallel robots on the final performance of the policy. In order to use the total reward as a single representative metric, we have to remove the curriculum, otherwise a more performant policy sees its task difficulty increase and consequently a decrease in the total reward. As such, we simplify the task by reducing the maximum step size of stairs and obstacles and directly train robots on the full range of difficulties.

We begin by setting a baseline with $n_{robots} = 20000$ and $n_{steps} = 50$, resulting in a batch size of 1M samples. Using this very large batch size results in the best policy but at the cost of a relatively long training time.

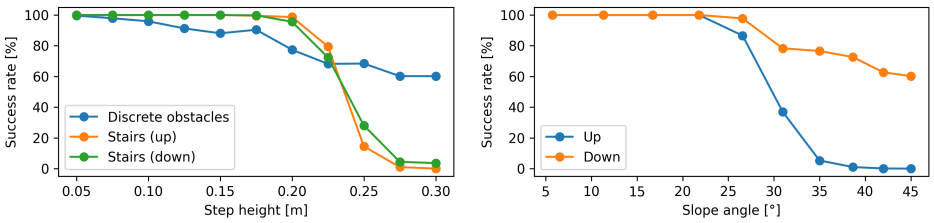
We then conduct experiments in which we increase the number of robots while keeping the batch size constant. As a result, the number of steps each robot takes per policy update decreases. In this case, the training time decreases with a higher number of robots, but the policy performance drops if that number is too high. We start from 128 robots corresponding to the level of parallelization of previous CPU implementations and increase that number up to 16384, which is close to the maximum amount of robots we could simulate on rough terrain with Isaac Gym running on a single workstation GPU.

In Fig. 4, we compare these results with the baseline, which allows us to select the most favorable trade-off between policy performance and training time. We see two interesting effects at play. First, when the number of robots is too high, the performance drops sharply, which can be explained by the time horizon of each robot becoming too small. As expected, with larger batch sizes, the overall reward is higher, and the time horizon effect is shifted, meaning that we can use more robots before seeing the drop. On the other hand, below a certain threshold, we see a slow decrease in



(a) (b) (c)

Figure 4: (a) Average and standard deviation (over 5 runs) of the total reward of an episode after 1500 policy updates for different number of robots and 3 different batch sizes. The ideal case of a batch size of 1M samples with 20000 robots is shown in red. (b) Total training time for the same experiments. (c) Reward dependency on total training time. Colors represent the number of robots, while shapes show the batch size (circles: 49152, crosses: 98304, triangles: 196608). Points in the upper left part of the graph (highlighted in green) represent the most desirable configuration.



(a) (b)

Figure 5: Success rate of the tested policy on increasing terrain complexities. Robots start in the center of the terrain and are given a forward velocity command of $0.75\,\mathrm{m/s}$, and a side velocity command randomized within $[-0.1, 0.1]\,\mathrm{m/s}$. (a) Success rate for climbing stairs, descending stairs and traversing discrete obstacles. (b) Success rate for climbing and descending sloped terrains.

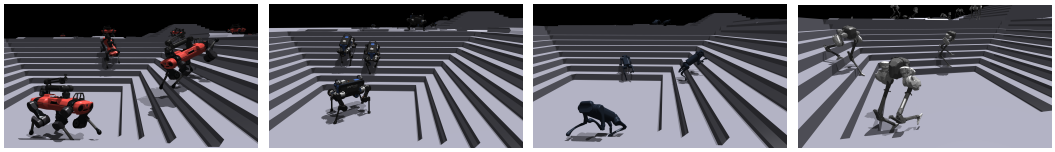


Figure 6: ANYmal C with a fixed arm, ANYmal B, A1 and Cassie in simulation.

performance with fewer robots. We believe this is explained by the fact that the samples are very similar with many steps per robot because of the relatively small time steps between them. This means that for the same amount of samples, there is less diversity in the data. In other words, with a low number of robots, we are further from the standard assumption that the samples are independent and identically distributed, which seems to have a noticeable effect on the training process. In terms of training time, we see a nearly linear scaling up to 4000 robots, after which simulation throughput gains slow down. As such, we can conclude that increasing the number of robots is beneficial for both final performance and training time, but there is an upper limit on this number after which an on-policy algorithm cannot learn effectively. Increasing the batch size to values much larger than what is typically used in similar works seems highly beneficial. Unfortunately, it also scales the training time so it is a trade-off that must be balanced. From the third plot we can conclude that using 2048 to 4096 robots with a batch size of $\approx 100k$ or $\approx 200k$ provides the best trade-off for this specific task.

4.2 Simulation

For our simulation and deployment experiments, we use a policy trained with 4096 robots and a batch size of 98304, which we train for 1500 policy updates in under 20 minutes[2]. We begin by measuring the performance of our trained policy in simulation. To that end, we perform robustness and traversability tests. For each terrain type, we command the robots to traverse the representative difficulty of the terrain at high forward velocity and measure the success rate. A success is defined as managing to cross the terrain while avoiding any contacts on the robot's base. Fig. 5 shows the results for the different terrains. For stairs, we see a nearly $100\,\%$ success rate for steps up to $0.2\,\mathrm{m}$, which is the hardest stair difficulty we train on and close to the kinematic limits of our robot. Randomized obstacles seem to be more demanding, with the success rate decreasing steadily. We must note that in this case, the largest step is double the reported height since neighboring obstacles can have positive and negative heights. In the case of slopes, we can observe that after $25\,\mathrm{deg}$ the robots are not able to climb anymore but still learn to slide down with a moderate success rate.

Given our relatively simple rewards and action space, the policy is free to adopt any gait and behavior. Interestingly, it always converges to a trotting gait, but there are often artifacts in the behavior, such as a dragging leg or unreasonably high or low base heights. After tuning of the reward weights, we can obtain a policy that respects all our constraints and can be transferred to the physical robot.

To verify the generalizability of the approach, we train policies for multiple robots with the same set-up. We use the ANYmal C robot with a fixed robotic arm, which adds about $20\,\%$ of additional

[2]Trained on: i9-11900k CPU, NVIDIA RTX A6000 GPU. VRAM requirements are in the supplementary material.

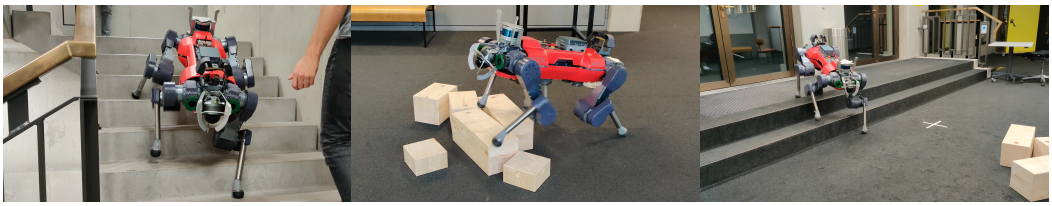


Figure 7: Locomotion policy, trained in under 20min, deployed on the physical robot.

weight, and the ANYmal B robot, which has comparable dimensions but modified kinematic and dynamic properties. In these two cases, we can retrain a policy without any modifications to the rewards or algorithm hyper-parameters and obtain a very similar performance. Next, we use the Unitree A1 robot, which has smaller dimensions, four times lower weight, and a different leg configuration. In this case, we remove the actuator model of the ANYdrive motors, reduce PD gains and the torque penalties, and change the default joint configurations. We can train a dynamic policy that learns to solve the same terrains even with the reduced size of the robot. Finally, we apply our approach to Agility Robotics' bipedal robot Cassie. We find that an additional reward encouraging standing on a single foot is necessary to achieve a walking gait. With this addition, we are able to train the robot on the same terrains as its quadrupedal counterparts. Fig. 6 shows the different robots.

4.3 Sim-to-real Transfer

On the physical robot, our policy is fixed. We compute the observations from the robot's sensors, feed them to the policy, and directly send the produced actions as target joint positions to the motors. We do not apply any additional filtering or constraint satisfaction checks. The terrain height measurements are queried from an elevation map that the robot is building from Lidar scans.

Unfortunately, this height map is far from perfect, which results in a decrease in robustness between simulation and reality. We observe that these issues mainly occur at high velocities and therefore reduce the maximum linear velocity commands to $0.6\,\mathrm{m/s}$ for policies deployed on the hardware. The robot can walk up and down stairs and handles obstacles in a dynamic manner. We show samples of these experiments in Fig. 7 and in the supplementary video. To overcome issues with imperfect terrain mapping or state estimation drift, the authors of [19] implemented a teacher-student set-up, which provided outstanding robustness even in adverse conditions. As part of future work, we plan to merge the two approaches.

5 Conclusion

In this work, we demonstrated that a complex real-world robotics task can be trained in minutes with an on-policy deep reinforcement learning algorithm. Using an end-to-end GPU pipeline with thousands of robots simulated in parallel, combined with our proposed curriculum structure, we showed that the training time can be reduced by multiple orders of magnitude compared to previous work. We discussed multiple modifications to the learning algorithm and the standard hyper-parameters required to use the massively parallel regime effectively. Using our fast training pipeline, we performed many training runs, simplified the set-up, and kept only essential components. We showed that the task can be solved using simple observation and action spaces as well as relatively straightforward rewards without encouraging particular gaits or providing motion primitives.

The purpose of this work is not to obtain the absolute best-performing policy with the highest robustness. For that use case, many other techniques can be incorporated into the pipeline. We aim to show that a policy can be trained in record time with our set-up while still being usable on the real hardware. We wish to shift other researchers' perspective on the required training time for a real-world application, and hope that our work can serve as a reference for future research. We expect many other tasks to benefit from the massively parallel regime. By reducing the training time of these future robotic tasks, we can greatly accelerate the developments in this field.

Acknowledgments

We would like to thank Mayank Mittal, Joonho Lee, Takahiro Miki, and Peter Werner for their valuable suggestions and help with hardware experiments as well as the Isaac Gym and PhysX teams for their continuous support.

References

[1] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[2] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2017.

[3] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine. Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[4] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang. Solving rubik's cube with a robot hand, 2019.

[5] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[6] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

[7] J. Hwangbo, J. Lee, and M. Hutter. Per-contact iteration method for solving contact dynamics. *IEEE Robotics and Automation Letters*, 3(2), 2018. URL www.raisim.com.

[8] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

[9] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.

[10] A. Stooke and P. Abbeel. Accelerated methods for deep reinforcement learning. *CoRR*, abs/1803.02811, 2018.

[11] B. Shacklett, E. Wijmans, A. Petrenko, M. Savva, D. Batra, V. Koltun, and K. Fatahalian. Large batch simulation for deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2021.

[12] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2018.

[13] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax - a differentiable physics engine for large scale rigid body simulation. In *35th Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

[14] A. Bouman, M. F. Ginting, N. Alatur, M. Palieri, D. D. Fan, T. Touma, T. Pailevanian, S.-K. Kim, K. Otsu, J. Burdick, and A.-a. Agha-Mohammadi. Autonomous spot: Long-range autonomous exploration of extreme environments with legged locomotion. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[15] C. Gehring, P. Fankhauser, L. Isler, R. Diethelm, S. Bachmann, M. Potz, L. Gerstenberg, and M. Hutter. Anymal in the field: Solving industrial inspection of an offshore hvdc platform with a quadrupedal robot. In *Field and Service Robotics*, 2021.

[16] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47), 2020.

[17] V. Tsounis, M. Alge, J. Lee, F. Farshidian, and M. Hutter. Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning. *IEEE Robotics and Automation Letters*, PP, 03 2020.

[18] S. Gangapurwala, M. Geisert, R. Orsolino, M. Fallon, and I. Havoutis. Real-time trajectory adaptation for quadrupedal locomotion using deep reinforcement learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[19] T. Miki, J. Lee, L. Wellhausen, V. Koltun, and M. Hutter. Wild anymal: Robust zero-shot perceptive locomotion. *Submitted to Science Robotics*, 2021.

[20] J. Siekmann, K. Green, J. Warila, A. Fern, and J. W. Hurst. Blind bipedal stair traversal via sim-to-real reinforcement learning. *CoRR*, abs/2105.08328, 2021.

[21] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[23] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[24] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev. Time limits in reinforcement learning. *CoRR*, abs/1712.00378, 2017.

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[26] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

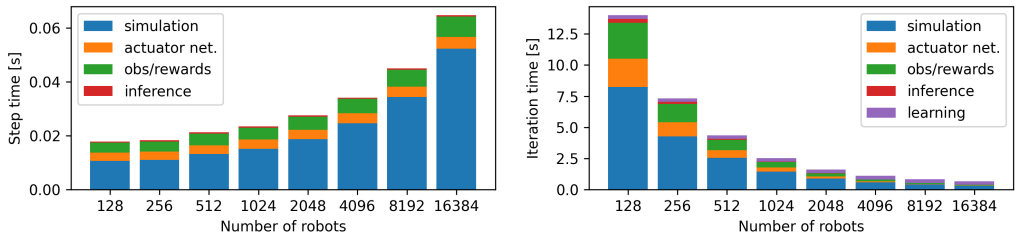
[27] J. Achiam. Spinning up in deep reinforcement learning, 2018. URL https://spinningup.openai.com/en/latest/.

[28] R. Wang, J. Lehman, J. Clune, and K. O. Stanley. Paired open-ended trailblazer (POET): endlessly generating increasingly complex and diverse learning environments and their solutions. *CoRR*, abs/1901.01753, 2019.

[29] Z. Xie, H. Y. Ling, N. H. Kim, and M. van de Panne. Allsteps: Curriculum-driven learning of stepping stone skills. *Proceedings of ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2020.

[30] C. Florensa, D. Held, X. Geng, and P. Abbeel. Automatic goal generation for reinforcement learning agents. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, 2018.

A Appendix



(a) (b)

Figure 8: (a) Computational time of an environment step. (b) Total time for a learning iteration with a batch size of $B = 98304$ samples.

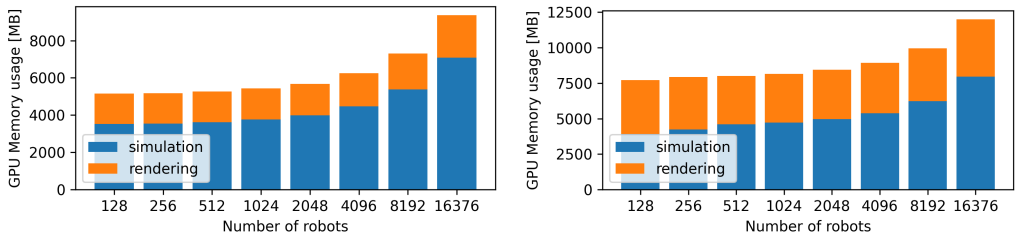
A.1 Simulation Throughput Analysis

In Fig. 8, we show the computational time of each part of an environment step, as well as the total times required for a learning iteration for different numbers of robots. In Fig. 8 (a) we observe that the simulation is the most time consuming step and its time slowly increases with the number of robots. The time spent on the computation of observations and rewards is the second slowest step and also slowly increases with the number of robots, while inference of the policy and the actuator network are computed in nearly constant time. Fig. 8 (b) shows the total time required to collect a fixed number of samples and perform the policy update. Increasing the number of parallel robots decreases the total time of all sub parts except the learning step which is independent on the number of robots. In Fig. 9 we examine the GPU VRAM required to train different number of robots both with and without graphical rendering. We see that $9\,\mathrm{Gb}$ are required to run 4096 robots with rendering enabled. Without a graphical output, $6\,\mathrm{Gb}$ are sufficient. On flat terrain these numbers are reduced to $7\,\mathrm{Gb}$ and $5\,\mathrm{Gb}$ respectively.

In Sec. A.1.1 and Sec. A.1.2 we describe additional techniques that we use to optimize the simulation throughput.

A.1.1 Time Step

The simulation time step needs to be maximized to get the maximum throughput. For each policy step, which we run at $50\,\mathrm{Hz}$, we need to perform multiple actuator and simulator steps to obtain a stable simulation. Since these added steps directly scale the amount of computation, we aim to reduce them as much as possible. We find that we can not use a time step smaller than $0.005\,\mathrm{s}$ which corresponds to four simulation steps per policy step. This limit is imposed by the actuator network (approximating a discrete-time PD-controller) becoming unstable and not by the simulation itself.



(a) Flat Terrain (b) Rough Terrain

Figure 9: GPU VRAM usage for different number of robots during training with a batch size of $B = 98304$ samples on flat and rough terrains.

A.1.2 Contact Handling

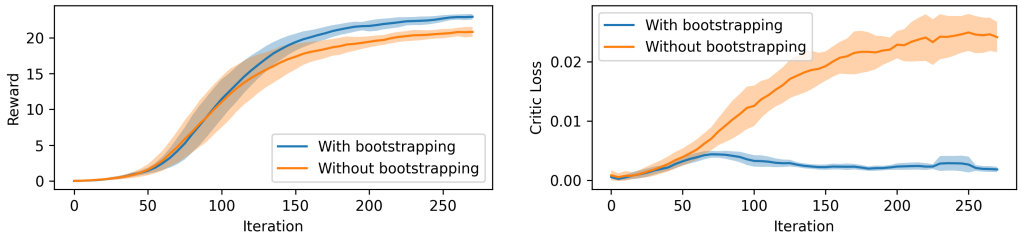
A large part of the simulator's computational time is spent on contact detection and handling. Reducing the number of potential contact pairs increases simulation throughput. We optimize the model of the robot to keep only the necessary collision bodies: feet, shanks, knees, and the base.

The resolution of the terrain plays an important role in contact optimization. A height field is a very common type of terrain representation, in which heights are defined on a uniform grid. However one of its unfortunate properties is that it is impossible to get vertical surfaces. In order to get a steep slope approximating a vertical step, we need high resolution, which degrades the simulation performance. Instead, we convert a low-resolution height field to a triangle mesh and correct the vertical surfaces.

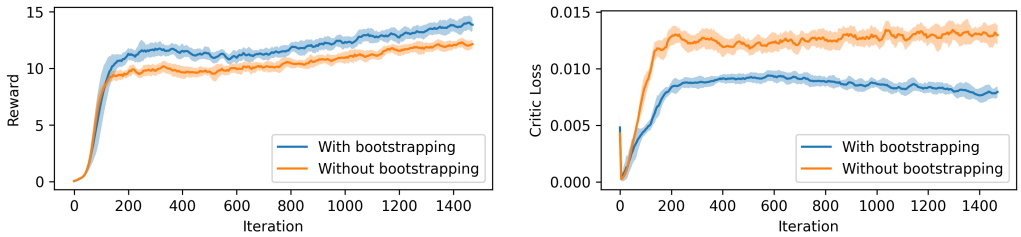
Finally, in PhysX (Isaac Gym's physics engine), even though contacts between the different robots are ignored, they are still detected. As such, the placement of robots in the terrain influences the computational load. Spreading the robots further apart from each other is highly beneficial. In the curriculum presented here, we need to place many robots close together at the beginning of training, but they quickly disperse as the training progresses. Additionally, once the robots learn to avoid crashes, there are fewer contacts with the base and knees and fewer costly resets. We see a factor of two difference in simulation time between the first minutes and the end of the training.

A.2 Effect of Time-out Bootstrapping

We analyze the effects of bootstrapping the reward at timeouts as described in Sec. 2.2.2 by comparing the total reward and the critic loss with and without the particular handling of timeouts. Fig. 10 shows the results for both flat and rough terrain tasks. We see that the critic loss is higher without bootstrapping, and correspondingly, the total reward is lower. Even though learning can be successful without this addition, it greatly reduces the critic loss and improves the total reward by approximately $10\,\%$ to $20\,\%$ for both tasks.



(a) Flat Terrain



(b) Rough Terrain

Figure 10: Comparison of total reward and critic loss, when training with and without reward bootstrapping on time-outs.

A.3 Reward Terms

Joint positions	$\mathbf{q}_j$
Joint velocities	$\dot{\mathbf{q}}_j$
Joint accelerations	$\ddot{\mathbf{q}}_j$
Target joint positions	$\mathbf{q}_j^*$
Joint torques	$\boldsymbol{\tau}_j$
Base linear velocity	$\mathbf{v}_b$
Base angular velocity	$\boldsymbol{\omega}_b$
Commanded base linear velocity	$\mathbf{v}_b^*$
Commanded base angular velocity	$\boldsymbol{\omega}_b^*$
Number of collisions	n_c
Feet air time	$\mathbf{t}_{air}$
Environment time step	dt

Table 1: Definition of symbols.

	definition	weight
Linear velocity tracking	$\phi(\mathbf{v}_{b,xy}^* - \mathbf{v}_{b,xy})$	$1dt$
Angular velocity tracking	$\phi(\boldsymbol{\omega}_{b,z}^* - \boldsymbol{\omega}_{b,z})$	$0.5dt$
Linear velocity penalty	$-\mathbf{v}_{b,z}^2$	$4dt$
Angular velocity penalty	$-\|\|\boldsymbol{\omega}_{b,xy}\|\|^2$	$0.05dt$
Joint motion	$-\|\|\ddot{\mathbf{q}}_j\|\|^2 - \|\|\dot{\mathbf{q}}_j\|\|^2$	$0.001dt$
Joint torques	$-\|\|\boldsymbol{\tau}_j\|\|^2$	$0.00002dt$
Action rate	$-\|\|\mathbf{q}_j^*\|\|^2$	$0.25dt$
Collisions	$-n_{collision}$	$0.001dt$
Feet air time	$\sum_{f=0}^{4}(\mathbf{t}_{air,f} - 0.5)$	$2dt$

Table 2: Definition of reward terms, with $\phi(x) := \exp(-\frac{\|\|x\|\|^2}{0.25})$. The z axis is aligned with gravity.

A.4 PPO Hyper-Parameters

Batch size	98304 (4096x24)
Mini-bach size	24576 (4096x6)
Number of epochs	5
Clip range	0.2
Entropy coefficient	0.01
Discount factor	0.99
GAE discount factor	0.95
Desired KL-divergence kl^*	0.01
Learning rate α	adaptive*

Table 3: PPO hyper-parameters used for the training of the tested policy. (*) Similarly to [9], we use an adaptive learning rate based on the KL-divergence, the corresponding algorithm is described in Alg. 1

Algorithm 1 Adaptive learning rate computation.

$kl \leftarrow KL(\pi_{new}, \pi_{old})$
if $kl > 2kl^*$ **then**
  $\alpha \leftarrow \max(10^{-5}, \alpha/1.5)$
else
  if $kl < 0.5kl^*$ **then**
    $\alpha \leftarrow \min(10^{-2}, 1.5\alpha)$
  end if
end if

A.5 Noise Level in Observations

Joint positions	$\pm 0.01\,\mathrm{rad}$
Joint velocities	$\pm 1.5\,\mathrm{rad/s}$
Base linear velocity	$\pm 0.01\,\mathrm{m/s}$
Base angular velocity	$\pm 0.2\,\mathrm{rad/s}$
Projected gravity	$\pm 0.05\,\mathrm{rad/s^2}$
Commanded base linear velocity	$0\,\mathrm{m/s}$
Commanded base angular velocity	$0\,\mathrm{rad/s}$
Measured terrain heights	$\pm 0.1\,\mathrm{m}$

Table 4: Noise scale for the different components of the observations. For each element, the noise value is sampled from a uniform distribution with the given scale and added to the observations.