Hierarchical Reinforcement Learning for Quadruped Locomotion

Deepali Jain, Atil Iscen, and Ken Caluwaerts

***Abstract*— Legged locomotion is a challenging task for learning algorithms, especially when the task requires a diverse set of primitive behaviors. To solve these problems, we introduce a hierarchical framework to automatically decompose complex locomotion tasks. A high-level policy issues commands in a latent space and also selects for how long the low-level policy will execute the latent command. Concurrently, the low-level policy uses the latent command and only the robot's on-board sensors to control the robot's actuators. Our approach allows the high-level policy to run at a lower frequency than the low-level one. We test our framework on a path-following task for a dynamic quadruped robot and we show that steering behaviors automatically emerge in the latent command space as low-level skills are needed for this task. We then show efficient adaptation of the trained policy to a different task by transfer of the trained low-level policy. Finally, we validate the policies on a real quadruped robot. To the best of our knowledge, this is the first application of end-to-end hierarchical learning to a real robotic locomotion task.**

I. INTRODUCTION

Locomotion for legged robots is a challenging control problem that requires high-speed control of actuators as well as precise coordination between multiple legs based on various types of sensor data. In addition to basic locomotion, different terrains, tasks or environmental conditions might require specific primitive behaviors.

Recent research shows promising results on learning based systems for locomotion tasks in simulation and real hardware [1], [2], [3]. Various techniques can be used to discover policies for such tasks. In this work, we focus on Reinforcement Learning (RL) to obtain robust policies.

Robot locomotion is an excellent match for hierarchical control architectures. Indeed, the separation of low-level control of the legs and high-level decision making based on the environment and task at hand provides multiple advantages such as reuse of the learned low-level skills across tasks, and interpretability of the high-level decisions.

Given a complex task, manually defining a suitable hierarchy is typically a tedious task that requires engineering of the state and action spaces as well as reward functions for each primitive. To overcome this, we introduce a hierarchical framework to automatically decompose complex locomotion tasks. A high-level policy issues commands to a low-level policy and decides for how long to execute the low-level policy at a time. The low-level policy acts according to commands from the high-level policy and on-board sensors. Our approach allows separation of the state variables that are used for low-level control, from state variables only required for higher-level control. Our architecture naturally allows the high-level to operate at a slower timescale than the low-level.

Robotics at Google, 10011 New York, USA.
{jaindeepali, atil, kencaluwaerts}@google.com

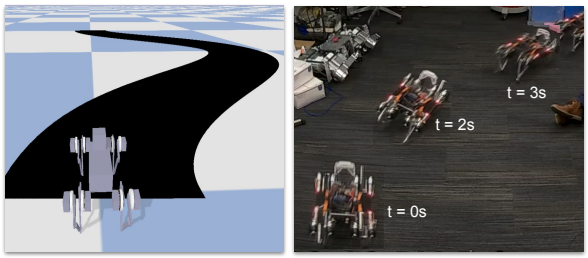


Fig. 1: Simulated task on the left and the robot performing a hierarchical policy learned in simulation. During execution the high-level policy executes intermittently to update the latent command for the low-level policy.

We test our framework on a path following task for a dynamic quadruped robot. The task requires walking into different directions to complete the track while keeping balance. Using our architecture, we train both levels of the hierarchical policy end-to-end. We show that steering behavior automatically emerges in the latent command space between the high-level and low-level policies, which allows reuse of the learned low-level behaviors. We show transfer of the low-level policy to a different track to achieve fast adaptation to a new task. Lastly, we deploy our policies to hardware to validate the learned behaviors on a real robot.

II. RELATED WORK

Hierarchical Reinforcement Learning (HRL) methods focus on decomposing complex tasks into simpler sub-tasks. Not only does this help simplify a single difficult problem, it can also help in adapting the solution faster to a new problem if sub-tasks are general enough. The framework based on predefined options [4], or temporally extended actions, is one of the first popular methods in this direction. More recently, considerable research attention is given to the problem of automatically discovering options through experience.

In methods like HRL with hindsight [5] and data-efficient HRL [6], hierarchy is introduced using universal value functions (value functions that are parameterized by 'goal'). Actions of a higher-level policy, running at a fixed slower timescale, act as goals for a lower-level. A goal is explicitly defined as a point in observation space and the low-level is rewarded for reaching that point. This allows both levels to be trained through their respective reward signals. However, this goal specification is not suitable in all situations. If the observation space is high dimensional, then the high-level task of selecting a goal becomes very difficult. Also,

determining when the goal is achieved requires task-specific domain knowledge.

Latent space policies for HRL [7] use a different approach to parameterize the low-level. The high-level outputs a set of latent variables as goal for the lower level that are learned through maximum entropy reinforcement learning. Both levels are then trained to maximize the main task reward. This, however, prevents the low-level from being reused for any other task.

Along similar lines, Osa et. al. [8] recently proposed a method based on information maximization to learn latent variables of a hierarchical policy.

In their paper on meta learning shared hierarchies [9], Kevin et al. propose a HRL framework that is learned on multiple related tasks. The low-level skills are reused across tasks while the meta-controller is task-specific. Instead of parameterizing a single low-level policy, the meta-controller selects a different low level policy from a set for each sub-task. In order for general low-level policies to emerge, the framework needs to be trained on a number of related tasks.

In our method, we use a latent goal representation to remove the need to hand design low-level rewards or deciding on the number of low-level policies. We also use different state representations for both levels to ensure that reusable low-level skills are learned even when trained on a single task. Moreover, in our method, the high-level policy runs at a variable timescale, easing processing requirements for higher-level state information.

The task of robot navigation lends itself to a hierarchical solution with path-planning at the high-level and point-to-point locomotion at the low-level. In this context, many methods [10], [11], [12] have been tried to solve these two tasks separately. Nicolas et al. [11], propose a hierarchical framework for locomotion based on modulated locomotor controllers. A low-level *spinal* network learns primitive locomotion by training on simple tasks. A high-level *cortical* network, drives behavior by modulating the inputs to the pre-trained spinal network. HRL with pre-trained primitives is also applied to the task of robot locomotion on rough terrains [13], [14]. In the DeepLoco [13] paper, low-level controllers achieve robust walking gaits that satisfy a stepping-target. High-level controllers then invoke desired step targets for the low-level controller.

We apply our hierarchical learning method to the robot locomotion task of following a path in 2D. Our method does not need specification of timescales for the two levels nor a low-level reward signal. Our end-to-end hierarchical learning framework automatically discovers steering behaviors at the low-level which can transfer to a real quadruped robot.

III. METHOD

A. Hierarchical Policy Structure and Execution

Our hierarchical policy is structured as shown in Fig. 2. The high-level policy (HL) receives higher-level observations from the environment and issues commands in a latent space to a low-level policy. The high-level also decides the duration for which the low-level is executed before the next high-level evaluation. The low-level (LL) receives observations from on-board sensors (low-level) and the current latent command from the high-level. It outputs actions to execute on the hardware. At the end of the duration set by the high-level, the high-level is invoked again and the process repeats (Fig. 3). Both high-level and low-level policies in this architecture are neural networks. Algorithm 1 shows how an episode is executed using a hierarchical policy in which the high-level and low-level have weights ϕ_h and ϕ_l respectively.

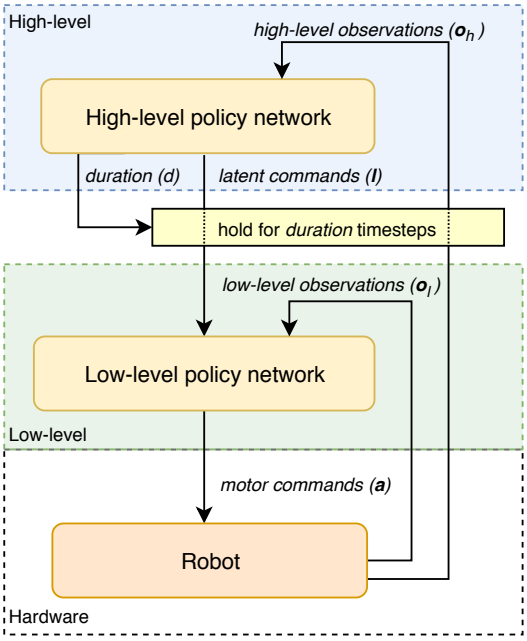


Fig. 2: Hierarchical policy. The high-level policy with parameters ϕ_h receives high-level observations $\boldsymbol{o}_h$ and outputs a *latent command vector* $\boldsymbol{l}$ and a duration d. The low-level policy (parameters ϕ_l) computes motor commands $\boldsymbol{a}$ based on $\boldsymbol{l}$ and low-level observations $\boldsymbol{o}_l$. The high-level policy is only evaluated every d steps. The architecture is trained end-to-end.

B. Learning Parameters of a Hierarchical Policy

To jointly learn the parameters of the high-level and low-level neural networks, we optimize a standard reinforcement learning objective. Consider a state space $\mathcal{S}$ and action space $\mathcal{A}$. A sequential decision making or control problem can be modeled as a Markov Decision Process (MDP). An MDP is defined by a transition function $P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ and a reward function, $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$. A policy $\pi_\theta(\boldsymbol{s})$, parameterized by a weight vector $\boldsymbol{\theta}$, maps states $\boldsymbol{s}$ to actions $\boldsymbol{a}$. For a hierarchical policy, $\boldsymbol{\theta}$ is the collection of parameters from all levels ($\boldsymbol{\theta} = \{\phi_h, \phi_l\}$) and the subset of state variables observable by the high-level and low-level are denoted as $\boldsymbol{o}_h$ and $\boldsymbol{o}_l$ respectively. The policy interacts with the MDP for an episode of T timesteps at a time. The reinforcement learning objective is to maximize the expected total reward

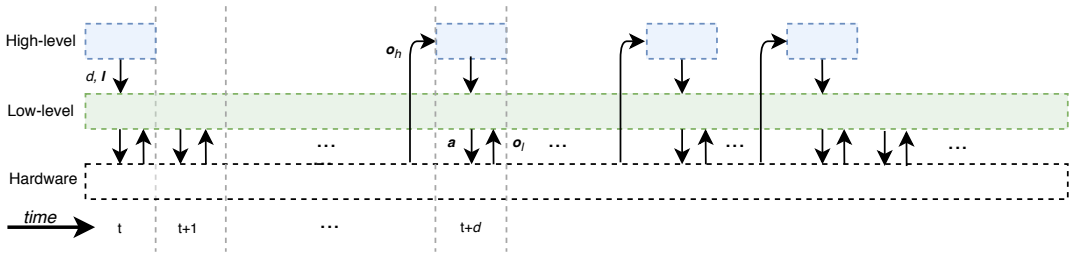


Fig. 3: Hierarchical policy evaluation timeline. The high-level policy computes a latent command for the low-level policy and a duration for which to execute the low-level policy. The low-level policy interacts with the hardware at a constant frequency. At the end of the high-level period, the high-level receives updated high-level observations and computes a new latent command and duration.

Algorithm 1 Executing a Hierarchical Policy

procedure RUNHRL($\boldsymbol{\theta}$) ▷ HRL policy weights
  $\{\boldsymbol{\phi}_h, \boldsymbol{\phi}_l\} = \boldsymbol{\theta}$
  $\boldsymbol{o}_h \leftarrow$ initial HL observation
  $R \leftarrow 0$ ▷ Episode reward
  $d \leftarrow 0$ ▷ LL duration
  while not end of episode **do**
    if $d = 0$ **then**
      $\boldsymbol{o}_h \leftarrow$ HL observation
      $\{d, \boldsymbol{l}\} \leftarrow f_{\boldsymbol{\phi}_h}(\boldsymbol{o}_h)$ ▷ Duration, latent command
    $\boldsymbol{a} = f_{\boldsymbol{\phi}_l}(\boldsymbol{o}_l, \boldsymbol{l})$ ▷ LL action (motor commands)
    $\boldsymbol{o}_l, r \leftarrow$ StepInEnvironment($\boldsymbol{a}$)
    $d \leftarrow d - 1$
    $R \leftarrow R + r$
  return R ▷ Total reward for the episode

at the end of episode:

$$\arg\max_{\theta} \quad \mathbb{E}\left[\sum_{t=1}^{T} r\left(\boldsymbol{s}_t, \pi_\theta\left(\boldsymbol{s}_t\right)\right)\right]. \tag{1}$$

We use a simple derivative-free optimization algorithm called Augmented Random Search (ARS) [15] to maximize R. The algorithm proceeds by choosing a number of directions uniformly at random on a sphere in policy parameter space, then evaluates the policy along these directions and finally updates the parameters along the top performing directions.

C. Transferring Low-Level Policies

An interesting aspect of our hierarchical method is that after learning a policy on one task, the low-level policy can be transferred to a new task from a similar domain. This allows sharing of primitive skills across related problems and is faster than learning from scratch on each task. The low-level policy can be transferred by keeping $\boldsymbol{\phi}_l$ fixed after learning on the original task and re-initializing $\boldsymbol{\phi}_h$. Then, during training only $\boldsymbol{\phi}_h$ is updated by ARS.

IV. EXPERIMENTS

A. Task Details

We apply our method to a path-following task for a quadruped robot. For this, we use the Minitaur quadruped robot from Ghost Robotics[1]. The Minitaur robot has 8 degrees of freedom (2 per leg). The swing and extension of each the legs is controlled using a PD position controller provided with the robot. We train our policies in simulation using pyBullet [16], [17].

For the locomotion task, we tackle the problem of following a curved path in 2D while staying within the allowed region. The robot is rewarded for moving towards the end of the path. The task requires the robot to steer left and right at different angles. The optimal trajectory for the center of mass for the robot is not defined and depends on the robot's anatomy and learned low-level behaviors. Steering poses additional challenges because the legs of the robot can only move in the sagittal plane. The reward function is given by:

$$r(t) = \mathrm{d}\left(\boldsymbol{x}(t-1), \boldsymbol{x}^{\mathrm{goal}}\right) - \mathrm{d}\left(\boldsymbol{x}(t), \boldsymbol{x}^{\mathrm{goal}}\right) \tag{2}$$

$$R = \sum_{t \geq 1} r(t), \tag{3}$$

where $\mathrm{d}(.,.)$ is the Euclidean distance, $\boldsymbol{x}$ is the position of the robot, and $\boldsymbol{x}^{\mathrm{goal}}$ is the final position of the path. We terminate an episode as soon as the robot moves out of the path.

To learn locomotion, we use the recent *Policies Modulating Trajectory Generators* (PMTG) architecture, which has shown success at learning forward locomotion on quadruped robots [2]. The PMTG architecture takes advantage of the cyclic characteristic of locomotion and of leg movement primitives by using trajectory generators. Trajectory generators serve as parameterized functions that provide circular leg positions. The policy is responsible to modulate the generator and adjust leg trajectories with a residual as needed. A more detailed explanation of the architecture can be found in the paper [2]. Our hierarchical policy is responsible for controlling the PMTG architecture which issues motor position commands.

[1] ghostrobotics.io

B. Hierarchical Architecture

As demonstrated in previous work [2], a well-trained linear neural network policy in combination with the PMTG can produce locomotion. Therefore we use linear neural networks for the high-level and the low-level policies. However, we clip the latent command space to $[-1, 1]^{\dim(\boldsymbol{l})}$, which allows us to more easily study the latent space. The number of dimensions of the latent command $\dim(\boldsymbol{l})$ is a hyper-parameter. Note that while the policy networks are linear, PMTG introduces recurrency and non-linearities [2].

We separate the state information into two. We only feed the robot's position $\boldsymbol{x}$ and the robot's orientation (yaw direction) into the high-level policy (4-dimensional). The high-level policy outputs the latent command $\boldsymbol{l}$ and a duration d.

The low-level policy network observes the 8-dimensional PMTG state (we use 4 trajectory generators, one per leg), 4-dimensional IMU sensor data (roll, pitch, roll rate, pitch rate), and the latent command $\boldsymbol{l}$ from the high-level policy. The output of the low-level network are 8 motor positions and 8 PMTG parameters.

We update the low-level's output every 6ms. The high-level is executed every d low-level steps (where d was calculated during the previous high-level cycle). In practice d is rescaled to $[100, 700]$ from the $[-1, 1]$ clipped value. Since the low-level timestep is 6ms, the time between high-level evaluations is between 0.6s and 4.2s. This highly simplifies the process of estimating the position and direction of the robot.

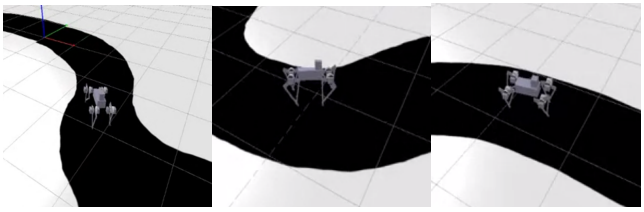
C. Transfer of Low-Level Policies to New Tasks

We show that our architecture can adapt to 2 different paths shown in Figure 4. We first train the architecture for path on the left side of Figure 4. The low-level policy only has access to proprioceptive sensor data and this forces it to learn generic steering primitives that can be reused across different paths. We test this property of our hierarchical architecture by reusing the trained low-level policies from path 1 when training on path 2.

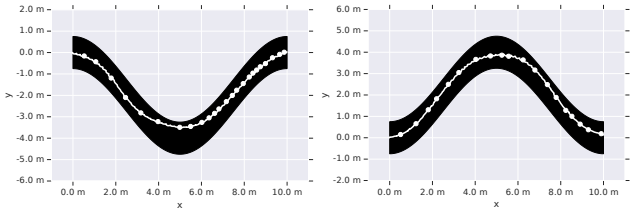
D. Baselines

For comparison, we train flat policies on these tasks. The input to the flat policies is the same as the high-level's observations concatenated with the low-level's in the hierarchical setup (except, trivially, for the latent commands) and the output is the same as the low-level actions. The flat policy also uses the same PMTG architecture for a fair comparison.

Secondly, we implement an expert hierarchical policy for additional comparison. We pre-train the low-level policy for this baseline using a carefully designed and tuned reward function to follow a target steering angle. The high-level policy computes the running duration d for the pre-trained low-level policy and also outputs a steering angle (a scalar in the range -1 (far left) to 1 (far right), instead of the latent command $\boldsymbol{l}$). The input for the expert policy's high-level and low-level is exactly the same as in the HRL case.



(a) Robot path tracking in simulation. If the robot's center of mass exits the black area, the episode is terminated.



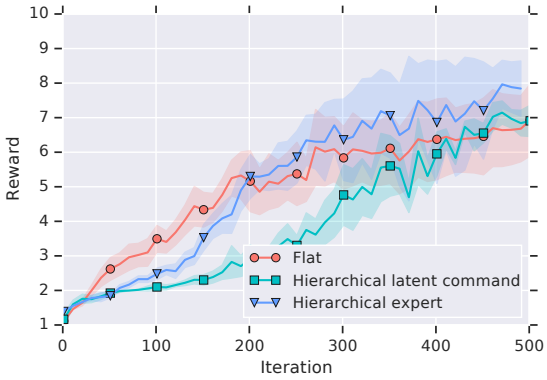
(b) Trajectory on 2 paths with a shared low-level policy (trained on the path on the left). Dots indicate when the high-level policy takes a new decision.

Fig. 4: Sample rollouts in simulation of the path-tracking task with a 4D latent command space.

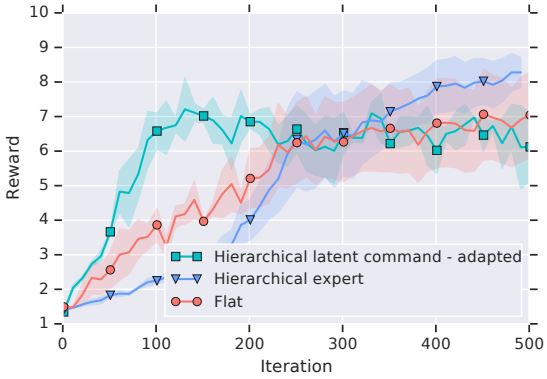
As in the HRL case, the baseline policies are trained by directly optimizing R using Augmented Random Search (ARS) [15]. We perform evaluation across different search directions in parallel. We train each method with a set of hyper-parameters (number of directions to search in ARS, number of top direction for updating parameters and number of latent command dimensions in case of our hierarchical method). Finally, we pick the best hyper-parameter for each and compare the average performance of 5 random training runs with those hyper-parameter settings.

In Fig. 5 we show learning curves for 3 policies, a flat policy, hierarchical policy with expert-designed, pre-trained low-level, and a hierarchical policy with latent command space (our method). The policies are trained on 2 different paths. All three methods succeed in solving the task of following the first path (Fig. 5a). For the second path, our method is able to solve the task significantly faster than other policies (Fig. 5b). On the second path, the flat policy has to learn the parameters from scratch. The expert policy's high-level learns to use the same low-level policy used in the first path. This low-level policy was pre-trained (see Appendix). Therefore, the expert policy needs extra training time to learn both levels separately. On the other hand, both levels of our latent command based hierarchical policy are trained from scratch on the first path. The best performing policy uses a 4 dimensional latent space. We can see that this policy can still reuse the same low-level and 4D latent commands to adapt quickly to a new task.

Fig. 4 shows how the robot trained with a hierarchical policy behaves in simulation. It successfully follows the path using steering behaviors. Complete trajectories can be seen in Fig. 4b. Markers along the trajectory show points at which the high-level becomes active and computes the next latent command and duration. The low-level policy was only



(a) Learning curves for path 1. All policies are trained from scratch.



(b) Learning curves for path 2. Our method (hierarchical latent) reuses the low-level policy learned for path 1.

Fig. 5: Learning curves of a flat policy, a hierarchical policy with latent commands and an expert hierarchical policy. We plot the average of 5 statistical runs with shaded area representing the standard error.

trained on the first path and is reused for the second path.

To simplify the analysis, we study a 2 dimensional latent command space learned by our method in Fig. 7. We evaluated the low-level for different points in the latent space. In Fig. 7a we show the movement direction of the robot when giving different points in latent space as commands to the low-level and executing the low-level for a fixed number of steps (1000). The length of the arrow is proportional to the distance covered. Corresponding color-coded robot trajectories are shown in Fig. 7b. We can observe that for the path following task, robot steering behaviors of varying velocities emerge automatically as low-level behaviors. The high-level uses these steering behaviors to navigate different parts of the path as show in Fig. 7b. Moreover, the high-level also decides a variable duration for each latent command (see Fig. 7b). We can observe that for straighter parts of the path, the high-level selects a longer duration to go forward, while for curved parts, it switches latent commands more frequently.

E. Hardware Validation

Finally, we validate our results by transferring an HRL policy to a real robot and recording the resulting trajectories.

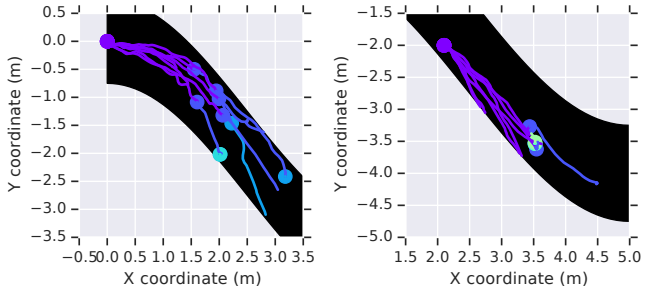


Fig. 6: The trajectories of the real robot measured with motion capture while using a trained HRL policy at different segments of the path.

We use a motion capture system (PhaseSpace Impulse X2E) to estimate the robot's current position and heading, which is then fed into the high-level policy. Since our architecture allows execution of the different levels at different frequencies, it is sufficient to transmit motion capture data to the high-level policy at a much lower rate compared to low-level sensor data such as IMU readings.

Because of the limited capture volume in our lab setting, we were only able to track the robot's trajectory along part of the task (see Fig. 1 and 6). To overcome this limitation, we recorded shorter robot trajectories starting at the origin. We then virtually moved the robot down the path by adding an offset to the motion capture's position estimate and recorded another set of trajectories. Note the significant variance for the real trajectories at the start of the path due to slippage of the legs during dynamic turning gaits.

V. CONCLUSION

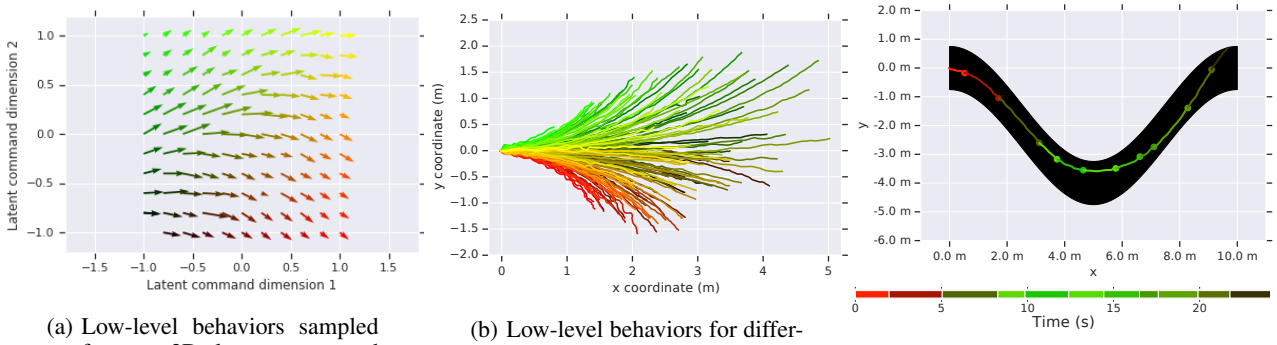
We presented a hierarchical control approach particularly suited for legged robots. By separating the architecture into two parts, a high-level and a low-level policy network, and jointly training them, we obtained a number of advantages over previous algorithms.

First, the architecture is agnostic to the task: we do not need to manually pick or pretrain the behaviors (primitives) of the low-level policy. As a consequence we also remove the need to design individual reward functions for each behavior. In fact, our algorithm outperforms a similar setup in which the low-level behaviors are predefined.

Secondly, our method can be used to bootstrap when training on a new task by transferring the trained low-level policy.

Finally, the high-level and low-level policies operate at different timescales and can use different state representations. This is of particular practical importance, since motor commands should be able to be calculated in mere milliseconds by a low-level policy for safety and stability reasons. High-level signals such as rewards or position estimates are often updated at much lower frequencies and might have to be transmitted via a wireless connection. Our approach provides a natural way to decouple these timescales.

The task at hand allowed us to study the results in detail in both simulation and hardware to validate our approach



(a) Low-level behaviors sampled from a 2D latent command space. Vector directions correspond to the movement direction of the robot. Vector length is proportional to the distance covered.

(b) Low-level behaviors for different latent commands (colors correspond to Fig. 7a). Notice that while diverse, the low-level behaviors are biased towards left turns because of the task at-hand.

(c) Sample trajectory of the HRL policy with a 2D latent command space. Dots indicate new high-level commands. The timeline shows the high-level activations.

Fig. 7: Analysis of latent command space $\boldsymbol{l}$ and low-level duration d.

and implementation. We show that given the path following task, the steering behaviors automatically emerge in a latent space, and the robot can easily adapt to a new path with low-level transfer. We also deployed these policies to hardware to validate the learned hierarchical policy.

In future work, we plan to apply this algorithm on tasks requiring a high level of agility in more complex environments. As an example, if the robot has to jump over an obstacle or climb stairs, manually defining a set of low-level behaviors will become even more cumbersome. We believe that the latent command space will allow us to tackle these challenges through automatic discovery of the complex primitives required to solve the task. In addition, we are planning to incorporate more complex sensors such as camera images, which naturally operate at different timescales and require significant computational power. In this case our approach would allow for distributed processing, without compromising performance.

APPENDIX

As part of the baselines, a low-level expert steering policy is trained separately. This policy is controlled by a scalar input from the high-level l, which determines the target direction. We train the policy using the ARS algorithm by rewarding the magnitude of the average steering angle over the past 50 timesteps. The reward is capped by the input l. Then another component (weighted by α) is added to the reward for moving forward, which is capped by a fixed value, $r^{\text{fw}}_{\text{cap}}$:

$$r^{\text{steer}}(t) = \min\left(l, \theta^{\text{steer}}_t\right) \tag{4}$$

$$r^{\text{fw}}(t) = \min\left(r^{\text{fw}}_{\text{cap}}, \boldsymbol{x}(t) - \boldsymbol{x}(t-1)\right) \tag{5}$$

$$r(t) = r^{\text{steer}}(t) + \alpha r^{\text{fw}}(t) \tag{6}$$

$$R = \sum_{t \geq 1} r(t). \tag{7}$$

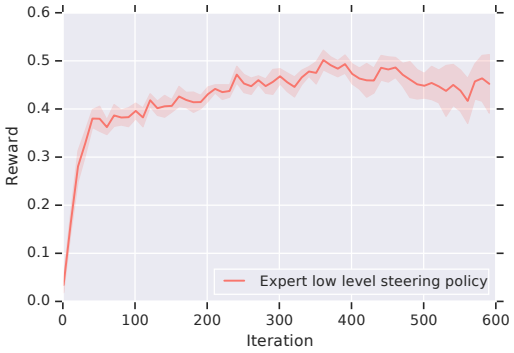


Fig. 8: Learning curve for the pre-training phase of the expert low-level policy.

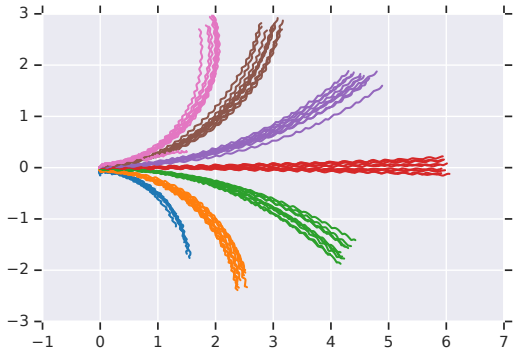


Fig. 9: Expert low-level policy with different inputs (axes in m).

For training, we randomly sample an input l from a uniform distribution for each episode. The learning curve for training this policy is shown in Fig. 8. Sample trajectories after training are shown in Fig. 9.

ACKNOWLEDGMENT

We would like to thank Jie Tan, Tingnan Zhang, Erwin Coumans, Sehoon Ha (Robotics at Google), Honglak Lee,

Ofir Nachum (Google Brain), and Arun Ahuja (DeepMind) for insightful discussions.

REFERENCES

[1] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[2] Atil Iscen, Ken Caluwaerts, Jie Tan, Tingnan Zhang, Erwin Coumans, Vikas Sindhwani, and Vincent Vanhoucke. Policies modulating trajectory generators. In *Conference on Robot Learning*, pages 916–926, 2018.

[3] Wenhao Yu, C Karen Liu, and Greg Turk. Policy transfer with strategy optimization. *arXiv preprint arXiv:1810.05751*, 2018.

[4] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[5] Andrew Levy, Robert Platt, and Kate Saenko. Hierarchical reinforcement learning with hindsight. *arXiv preprint arXiv:1805.08180*, 2018.

[6] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3307–3317, 2018.

[7] Tuomas Haarnoja, Kristian Hartikainen, Pieter Abbeel, and Sergey Levine. Latent space policies for hierarchical reinforcement learning. *arXiv preprint arXiv:1804.02808*, 2018.

[8] Takayuki Osa, Voot Tangkaratt, and Masashi Sugiyama. Hierarchical reinforcement learning via advantage-weighted information maximization. *arXiv preprint arXiv:1901.01365*, 2019.

[9] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

[10] Bastian Bischoff, Duy Nguyen-Tuong, IH Lee, Felix Streichert, Alois Knoll, et al. Hierarchical reinforcement learning for robot navigation. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2013)*, 2013.

[11] Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

[12] Aleksandra Faust, Kenneth Oslund, Oscar Ramirez, Anthony Francis, Lydia Tapia, Marek Fiser, and James Davidson. PRM-RL: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5113–5120. IEEE, 2018.

[13] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):41, 2017.

[14] Xue Bin Peng, Glen Berseth, and Michiel Van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 35(4):81, 2016.

[15] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[16] Erwin Coumans. Bullet Physics SDK.

[17] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.