Learning a Contact-Adaptive Controller for Robust, Efficient Legged Locomotion

Xingye Da[*], **Zhaoming Xie**[*†], **David Hoeller**[*], **Byron Boots**[*‡],
Animashree Anandkumar[*§], **Yuke Zhu**[*♯], **Buck Babich**[*], **Animesh Garg**[⋆*]

Abstract: We present a hierarchical framework that combines model-based control and reinforcement learning (RL) to synthesize robust controllers for a quadruped (the Unitree Laikago). The system consists of a high-level controller that learns to choose from a set of primitives in response to changes in the environment and a low-level controller that utilizes an established control method to robustly execute the primitives. Our framework learns a controller that can adapt to challenging environmental changes on the fly, including novel scenarios not seen during training. The learned controller is up to 85 percent more energy efficient and is more robust compared to baseline methods. We also deploy the controller on a physical robot without any randomization or adaptation scheme.

Keywords: Legged Locomotion, Hierarchical Control, Reinforcement Learning

1 Introduction

Quadruped locomotion is often characterized in terms of *gaits* (walking, trotting, galloping, bounding, etc.) that have been well-studied in animals [1] and reproduced on robots [2, 3]. A gait is a periodic contact sequence that defines a specific contact timing for each foot. Controllers designed for these gaits have demonstrated robust behaviors on flat ground and rough terrain locomotion. However, it is rarer to find controllers that can change gaits or contact sequences to adapt to environmental changes. An adaptive gait can reduce energy usage by removing unnecessary movement, as suggested in horse studies [1]. It is also required for completing more challenging scenarios such as riding a skateboard or recovery from leg slipping, as shown in Figure 1 (a, b).

In most model-based and learning-based control designs, the contact sequence is fixed or predefined [2, 3, 4, 5, 6, 7, 8]. Dynamic adaptation of the contact sequence is challenging because of the hybrid nature of legged locomotion dynamics as well as the inherent instability of such systems. While it is possible to generate adaptive contact schemes via trajectory optimization [9, 10, 11], such approaches are generally too compute-intensive for real-time use.

Here we present a hierarchical control framework for quadrupedal locomotion that learns to adaptively change contact sequences in real-time. A high-level controller is trained with reinforcement learning (RL) to specify the contact configuration of the feet, which is then taken as input by a low-level controller to generate ground reaction forces via quadratic programming (QP). At inference time, the high-level controller needs only evaluate a small multi-layer neural network, avoiding the use of an expensive model predictive control (MPC) strategy that might otherwise be required to optimize for long-term performance. The low-level controller provides high-bandwidth feedback to track base and foot positions and also helps ensure that learning is sample-efficient. The framework produces a controller that is up to 85 percent more energy efficient and also more robust than baseline approaches.

We train our controller with a simulated Unitree Laikago [12] on a split-belt treadmill, as shown in Figure 1 (c). The two belts can adjust speed independently, and we change the robot orientation to increase variation. In addition to comparing energy use and robustness to the baselines, we also demonstrate zero-shot transferability by testing the controller in novel situations such as one where a foot encounters a slippery surface (e.g., with zero friction), which we call the "banana peel" test.

---

[*]NVIDIA, [†]Univ. of British Columbia, [‡]Univ. of Washington, [§]Caltech, [♯]UT Austin, [⋆]Univ. of Toronto, Vector Institute. Work done at NVIDIA. Correspondence: `xda@nvidia.com, zxie47@cs.ubc.ca`
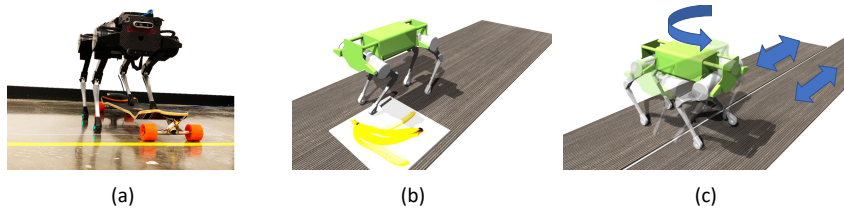
Figure 1: (a) Riding a skateboard requires a contact sequence that only moves the feet on the ground while keeping the feet on the board still. (b) "banana peel" test: we put a frictionless mat under a foot to test robustness. (c) We train and test the robot on a split-belt treadmill where the speeds of the two belts are changed separately with the robot facing different directions.

Furthermore, we show that the controller learns to generate novel contact sequences that have not been previously shown in either the model-based or learning-based approaches. Finally, we deploy the controller on a physical robot to demonstrate sim-to-real transfer,[2] which succeeds without any randomization or adaptation scheme due to the robustness of the low-level controller.

Summary of Contributions:

1. We introduce a hierarchical control structure that combines model-based control design and model-free reinforcement learning for legged locomotion.
2. We demonstrate that our framework allows sample-efficient learning, zero-shot adaptation to novel scenarios, and direct sim-to-real transfer without randomization or adaptation schemes.
3. Our framework learns adaptive contact sequences that are not present in either model-based or learning-based methods in real-time control. This is evidenced in the natural-looking behaviors that minimize unnecessary movement and energy usage in our split-belt treadmill scenarios.

2 Related Work

Model-based Legged Locomotion Control   Model-based control designs [2, 4, 5, 6] use trajectory optimization and model predictive control methods that optimize the performance for a finite horizon, where the input includes a predefined contact sequence. Although one can change the control sequence externally to demonstrate various gaits, it cannot adapt to changes in the environment. Contact-implicit optimization [9, 10, 11] is used to solve non-convex, stiff problems that are not amenable to real-time use. The work in [13] introduced the Feasible Impulse Set which allows online gait adaptation in planar models, but no 3D work has been presented.

Learning Legged Locomotion   Recently there has been significant work investigating the use of reinforcement learning to obtain locomotion policies for legged robots [3, 7, 8, 14]. However, the resulting policies are often either less robust compared to controllers obtained from model-based methods (e.g., [3, 7]) or require the design of complicated reward functions and a large number of training samples, e.g [8]. A learned policy is often brittle and can fail under mild environmental changes. Many approaches like meta learning [15] or Bayesian Optimization in behavior space [16] have been proposed to adaptively update the policies. This usually requires the policy to interact with the target environment to collect additional data. In contrast to these approaches, our method can adapt to a changing environment without any online data collection.

Hierarchical Control   Hierarchical framework can greatly improve learning efficiency, as shown in many prior works, e.g., [17]. In robotics tasks, it is beneficial to decompose a controller into modules and obtain controllers in a hierarchical manner. A low-level controller can be model-based [18, 19, 20] or learned [21, 22] such that it can achieve subgoals specified by a learned high-level controller. Specifically for locomotion tasks, it is natural to decompose a controller into direction-following and navigation modules [21, 22, 23, 24, 25]. We also adopt a hierarchical structure, however, our work is distinguished from previous work in that the goal of the high-level controller is to choose a low-level primitive in order to adapt to environmental changes instead of specifying

---

[2]Due to the COVID-19 pandemic, access to the physical robot has been limited. We thus focus our quantitative analysis on simulation while demonstrating only qualitative results on the physical robot.
Video https://youtu.be/JJOmFZKpYTo
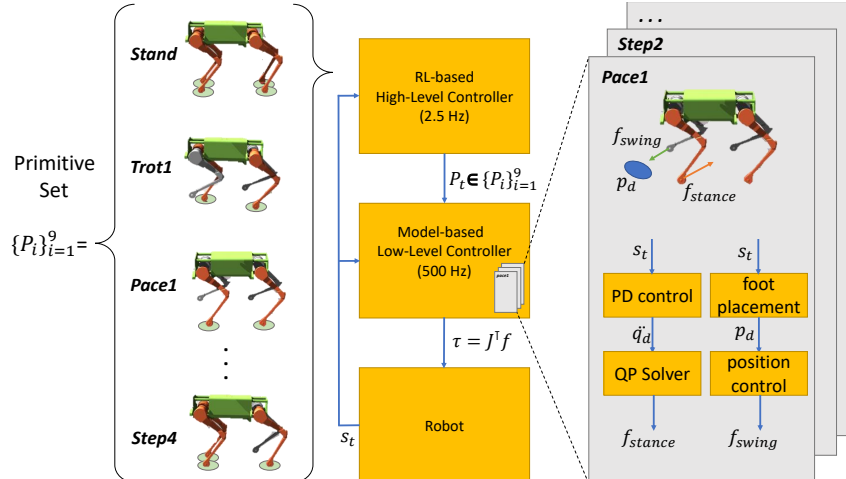Website https://sites.google.com/view/learn-contact-controller/home

Figure 2: Overview of our system. **Left**: Primitives P_i are distinguished by the contact configuration. The stance legs in each primitive are colored orange. **Center**: Hierarchical structure of the controller. The high-level controller chooses from a set of primitives based on the robot state s_t, and the low-level controller computes the motor torques τ based on the robot state and the primitive chosen. **Right**: The low-level controller uses stance foot forces to control the base pose and moves the swing feet to their target positions.

subgoals. This does not preclude other high-level policies or behaviors; e.g.. one could easily add a navigation module in our framework.

3   Method

We propose a hierarchical framework to perform locomotion that combines model-based control with reinforcement learning. The system is visualized in Figure 2.

The state of the robot $s = (q, \dot{q}, p_{\text{foot}})$ consists of the base pose $q \in \mathbb{R}^3 \times S^3$, containing the position p_{body} and orientation Θ of the robot's body, the velocity vector $\dot{q} \in \mathbb{R}^6$, and the four Cartesian foot positions $p_{\text{foot}} = (p_1, p_2, p_3, p_4) \in \mathbb{R}^{12}$ relative to the base. A primitive $P = \{0 : \text{Stance}, 1 : \text{Swing}\}^4 \in \mathbb{Z}^4$ is a Boolean array that specifies the stance/swing state of the four feet. The high-level controller chooses the appropriate primitive and the low-level controller uses the stance feet to generate ground reaction forces by solving a QP for base pose control and moves the swing feet based on a foot-placement algorithm. The low-level controller runs at $500\,\text{Hz}$ for high-speed feedback and the high-level controller runs at $2.5\,\text{Hz}$ to match the primitive execution time.

The high-level controller can be designed manually. If we set all foot states to *Stance*, the result is a standing gait. If we instead synchronize diagonally opposed feet and switch the state at every time step, the result is the trotting gait. As we demonstrate, however, a more adaptive high-level controller is needed to reduce energy consumption, reject disturbance, or react to friction changes.

In this section, we first define the primitives that are used in the controller. We then give details on the low-level controller that implements these primitives and on the high-level controller that learns to select from these primitives to complete multiple tasks.

3.1   Primitives

A primitive P represents a contact configuration for the four feet. Each foot is in either a *Stance* or *Swing* state, and there are thus $2^4 = 16$ possible primitives in total. We only use 9 primitives that are commonly used in quadruped locomotion, as summarized in the following table:

Primitive	*Stand*	*Trot1*	*Trot2*	*Pace1*	*Pace2*
Feet State	[0 0 0 0]	[1 0 0 1]	[0 1 1 0]	[0 1 0 1]	[1 0 1 0]

Primitive	*Step1*	*Step2*	*Step3*	*Step4*	
Feet State	[1 0 0 0]	[0 1 0 0]	[0 0 1 0]	[0 0 0 1]	

3

where 0 indicates that the corresponding foot is a stance foot and 1 indicates otherwise. The order of the foot states is {Left Front, Right Front, Left Rear, Right Rear} or {LF, RF, LR, RR} in short.

3.2 Low-Level Controller

We implement each primitive with a low-level torque controller. We find that a simple model-based method is sufficient to complete most of our tasks and is straightforward to transfer to the real robot.

Base Pose Control The low-level controller receives a primitive P_t from the high-level controller. It also receives the target base pose q_d and velocity $\dot{q}_d$ from user command. The controller computes foot forces by solving a QP, so the base pose can track the target pose and respect contact constraints.

Similar to [2], we approximates the quadruped dynamics as a linearized centroidal dynamics,

$$\ddot{q} = \mathbf{M}f - \tilde{g}, \tag{1}$$

where $\mathbf{M} \in \mathbb{R}^{6 \times 12}$ is the inverse inertia matrix, $f = (f_1, f_2, f_3, f_4) \in \mathbb{R}^{12}$ is the column vector of Cartesian forces for each foot, and $\tilde{g} = (g, 0_3) \in \mathbb{R}^6$ is the augmented gravity vector. The detailed derivation is given in Appendix A.

Given a target base pose q_d and velocity $\dot{q}_d$, we use PD control to compute the target acceleration

$$\ddot{q}_d = k_p(q_d - q) + k_d(\dot{q}_d - \dot{q}). \tag{2}$$

This is then used to construct a QP to find foot forces that minimize the acceleration error while respecting the contact configuration and friction constraints

$$\min_f \quad ||\mathbf{M}f - \tilde{g} - \ddot{q}_d||_{\mathbf{Q}} + ||f||_{\mathbf{R}}$$
$$\text{subject to} \quad f_{z,i} \geq f_{z,\min} \quad \text{if } P_{t,i} \text{ is } \mathbf{Stance}$$
$$f_{z,i} = 0 \quad \text{if } P_{t,i} \text{ is } \mathbf{Swing} \tag{3}$$
$$-\mu \, f_x \leq f_z \leq \mu \, f_x$$
$$-\mu \, f_y \leq f_z \leq \mu \, f_y,$$

where $\mathbf{Q}$ and $\mathbf{R}$ are diagonal matrices that adjust weights in the cost function.

Swing Foot Control The desired foot position $p_{d,i}$ for foot i is computed by a linear foot-placement heuristic

$$p_{d,i} = p_{0,i} + k(\dot{p}_{\text{body}} - \dot{p}_{d,\text{body}}), \tag{4}$$

that adjusts position from the default state $p_{0,i}$.

A position controller is then used to compute the swing foot force by

$$f_i = k_{p,i}(p_{d,i} - p_i) - k_{d,i}\dot{p}_i. \tag{5}$$

Torque Control The foot forces computed via pose control and swing foot control are converted to motor torques by $\tau = J^T f$, where $J \in \mathbb{R}^{12 \times 12}$ is the feet positions Jacobian matrix with respect to motor states. This is updated at $500\,\text{Hz}$.

3.3 High-level Controller

Our high-level controller selects primitives based on the current robot state and is queried at $2.5\,\text{Hz}$. Here we describe how we use RL to learn the high-level controller in detail.

State Space and Action Space We model the environment as a partially observable Markov decision process (POMDP). Specifically, the high-level controller takes the body pose $\mathbf{q}$, excluding the x, y linear positions, and the relative foot positions $\mathbf{p}_{foot}$ as input. To endow the controller with the capability to learn common gaits such as pacing and trotting that alternate between primitives, we also include the previously-used primitive as an input. The output of the controller is a 9-dimensional one-hot vector that indicates which primitive will be selected for the low-level controller. Assuming the environment is deterministic, the input provided is enough to determine the next robot state. However, environments are often parameterized by some unobserved random variables, causing the transition dynamics to be stochastic with high variance. The goal of the high-level controller is then to choose primitives that can adapt to this high variance environment while optimizing for simple objectives such as energy efficiency and stability.
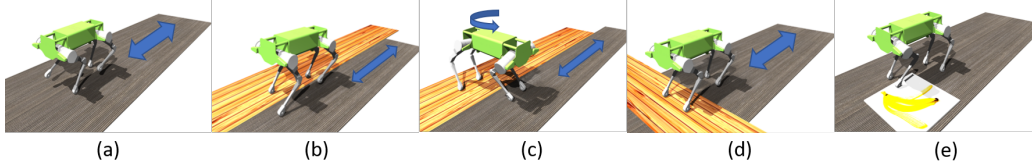
Figure 3: Training and Testing scenarios. Scenarios (a)-(c) scenarios are used during training where we vary the treadmill speeds, the number of moving belts, and the orientation of the robot. Scenarios (d)-(e) are introduced only during testing. Scenario (d) introduces a fixed plywood bridge on top of the treadmill, and scenario (e) inserts a frictionless mat under the feet of the robot to test stability.

Policy Representation   Since the action space is discrete, instead of learning a policy directly, we choose to learn a Q-function that takes the state and action as input, and output the sum of discounted future rewards. At test time, the action that yields the maximum Q value is selected.

Reward Design   We use a simple reward function of the form

$$r = 1 - 0.0025\frac{1}{T}\sum \|\tau\|^2 - \frac{1}{T}\sum \|\dot{p}_{d,\text{body}} - \dot{p}_{\text{body}}\|^2 \,, \tag{6}$$

where the constant 1 ensures that the reward is positive, $\tau, p_{d,\text{body}}$ and $\dot{p}_{\text{body}}$ are control torques, body linear velocity, and desired body linear velocity respectively. T is the number of simulation timesteps within a primitive cycle.

Training   We adopt a DQN-like [26] training procedure, and implement a double Q-function [27] and delayed target network update [28]. In addition, instead of using an epsilon greedy strategy during training, the probability of applying an action is based on Q value estimates normalized by the softmax operator. More specifically, let $\{Q_1, Q_2, \ldots, Q_9\}$ be the Q value estimates of the different actions at a particular state, and let the maximum Q value be $Q_{\max} = \max(Q_1, Q_2, \ldots, Q_9)$, the probability of an action i being sampled will be proportional to $\exp(-\nu\frac{Q_i}{Q_{\max}})$, where ν is a hyperparameter controlling how sensitive the sampling probability distribution is to the Q value. The pseudocode of the algorithm is described in Appendix B.

The Q-function for the high-level policies is implemented as a two-layer feedforward neural network with ReLU activation functions, each layer has 64 neurons. We set the temperature $\nu = 5$ and update the Q-function every 100 samples, with 50 stochastic gradient descents and mini-batch size of 512. The learned policy already performs well with 10^5 samples, and we collect a maximum of 5×10^5 samples. Note that this is orders of magnitude fewer samples than used in previous work that employed RL with similar-scale quadrupeds like the ANYmal or Laikago [3, 8, 29].

4   Results

We use a GPU-accelerated simulator [30] Isaac Gym to train the Laikago robot and to compare different controllers. This simulator has been used for various robotics manipulation sim-to-real tasks [31, 32] and is validated to simulate rigid body physics with reasonable accuracy. In this section, we describe the experiments we use to validate our framework and show improvement over baseline methods.

4.1   Baseline Controllers

As a baseline, we have created five manually-designed high-level controllers. These are comparable to those used in typical model-based control approaches to quadrupedal locomotion [2, 4]. Here we briefly describe how they work.

Standing   Only the *Stand* primitive is used in this controller. It is the most energy efficient controller in the absence of perturbations but also the least robust one if perturbations are present.

Trotting   The trotting controller alternates between *Trot1* and *Trot2* and displays a trotting gait. It is commonly used for quadruped locomotion due to its stability.

(a) Energy comparison over different speeds.　　　(b) Energy comparison over different yaws.
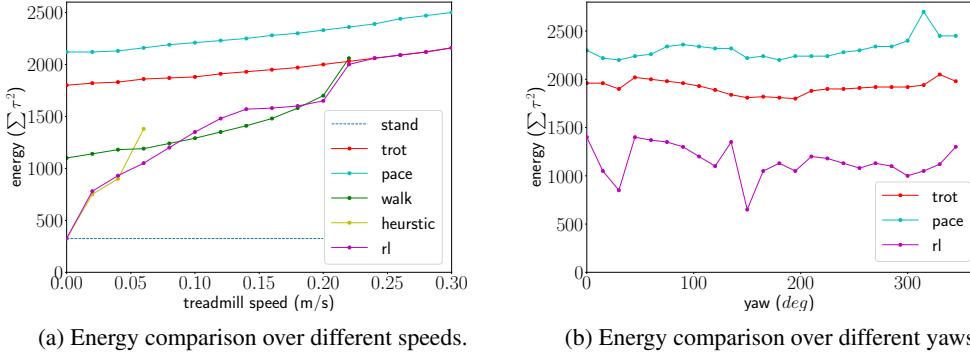
Figure 4: Comparison of the average energy used. (a) The standing, walking and heuristic controllers fails at high speed, while trotting and pacing controllers remain on high-energy level. The learned controller (rl) can handle all speed variation and more energy efficient. (b) The only baseline controllers that can handle split-treadmill are trotting and pacing. The learned controller is 50 percent more energy efficient on average. Energy for the learned controller drops significantly at $yaw = 150$ deg because only one foot moves while two feet move in nearby orientations.

Pacing　　The pacing controller alternates between ***Pace1*** and ***Pace2***. It is another commonly used gait in quadrupedal locomotion but usually less stable and less energy efficient than trotting.

Walking　　The walking controller lifts one foot at a time by switching between four stepping primitives in the order: ***Step1*** → ***Step4*** → ***Step2*** → ***Step3*** → ***Step1*** → . . .

Heuristic-Based Controller　　In this controller, we create a heuristic Q-function $\hat{Q}_i = \hat{Q}(P_i)$ for each primitive and the controller executes the primitive with the largest $\hat{Q}$. We define

$$\hat{Q}_i = J_{QP,i} + k_q \sum_{j=1}^{4} ||p_{d,j} - p_j||_2, \tag{7}$$

where $J_{QP,i}$ is the QP cost in Equation 3 for primitive P_i. This represents a trade-off between the pose control and the swing feet control. When more feet are in *Stance* state, $J_{QP,i}$ will be smaller because of the contact constraints. When more feet are in *Swing* state, the foot position error will be smaller since only the swing feet can move to the target position. The heuristic-based controller is able to adapt contact behaviors in a few scenarios but is less robust overall.

End-to-end Learned Controller　　We can also directly learn a control policy end-to-end instead of using the hierarchical framework proposed here. However, a purely learned policy on similar scale quadrupeds like the ANYmal and Laikago requires a training sample count on the order of 10^8 to 10^9, with careful reward shaping [3, 8, 29], while with the hierarchical framework, we use simple reward specification and get good performance at around 10^5 samples. We train a controller similar to [3], where the robot is tracking a reference trotting motion. The resulting controller is not as robust as the manually designed trotting controller and fails in most of our testing scenarios. This is consistent with [3], where it is also found that a purely learned controller is not robust, and a sophisticated adaptation scheme is needed to deal with environmental changes. Given these drawbacks, we do not compare with the end-to-end learned controller.

4.2　Training Scenarios

We use a split-belt treadmill to train the policy so that the policy learns to choose different primitives to adapt to changing dynamics. During a new rollout, the speed of the treadmill is sampled from $[-0.3, 0.3]$ m/s, see Figure 3 (a). We also randomly pause one side of the treadmill and command the robot to face different orientations. this is shown in Figure 3 (b) and (c), where the plywood represents the side of the treadmill that is not moving. This provides a rich set of changing dynamics that the policy must learn to adapt to. Note the policy have no knowledge of the underlying treadmill
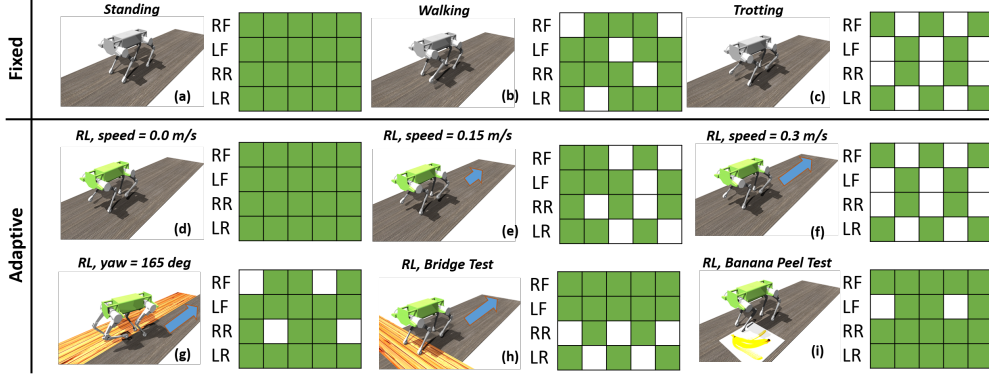
Figure 5: Contact sequence of different high-level controllers under different scenarios. A filled green block indicates that the corresponding foot is in contact with the ground. The three baseline controllers (standing, walking, and trotting) each use a fixed contact sequence for all scenarios, while the learned controller adapts the contact sequence to the scenario.

parameters. The low-level controller is commanded to stay at the origin with a target velocity of $0\ m/s$.

4.3 Comparison

Energy We compare the energy use across different high-level controllers, shown in Figure 4. The energy is computed as an average of sum square motor torques over ten seconds.

First, we compare the energy consumption in a scenario shown in Figure 3 (a) where the treadmill is moving parallel to the robot in speed range $[0, 0.3]\ m/s$. The standing gait is the most energy efficient gait which uses 76.7%, 83.3%, and 85.7% less energy than the walking, trotting and pacing gaits respectively. The downside is that it can be only used at zero speed. The heuristic and learned (rl) controllers can start with the same lowest energy level and gradually increase the energy level as speed increases. As the treadmill speed reaches $0.2\ m/s$, the heuristic controller quickly fails while the learned controller's energy usage is comparable to the walking controller. The walking controller fails when the treadmill speed exceeds $0.2\ m/s$ while the learned controller adjusts the primitive such that the energy is the same as the trotting controller. The trotting and pacing controllers are able to cover the full speed range but consume more energy due to unnecessary leg movements.

We then compare the energy consumption in scenarios similar to Figure 3 (b) and (c) where only one side of the treadmill is moving at $0.3\ m/s$ and the robot is commanded to face different directions. Most of the baseline controllers fail except trotting and pacing. The learned controller consumes on average 40.1% and 50.4% less energy than the trotting and pacing controllers respectively.

Contact Sequence The energy efficiency of the learned controller is mostly due to adaptive contact planning. Figure 5 shows the contact pattern of baseline and learned controllers. (a), (b), and (c) show the fixed pattern of standing, walking and trotting. Note that they use the same contact sequence for all scenarios. Contrarily, the learned controller adapts contact sequence in different scenarios. When speed increases, the learned controller transients from standing gait to trotting gait, shown in (d), (e), and (f). We highlight the contact pattern in (e), where the learned controller uses a combination of *Stand*, *Trot* and *Step* primitives at speed $0.15\ m/s$. In (g), only one belt is moving and we replace the other non-moving belt with plywood for clarity, the learned controller only moves the two right feet, thus more energy efficient compared to trotting or pacing. (h) and (i) are two scenarios not seen in training and the learned controller demonstrates novel contact sequences.

4.4 Zero-Shot Adaptation to Novel Scenarios

We test the learned controller in novel scenarios that are not present during training to show that the policy can generalize. A purely learned controller usually overfits the training dynamics and requires the collection of additional data in the target environment to make the adaptation [3]. With our hierarchical framework, the learned controller is able to adapt to these scenarios directly.

Figure 6: **Real robot tests**.
Left: Timelapse of forward walking.
Right: The left rear leg is perturbed.

Bridge Test   We test the learned policy on a scenario where the treadmill is placed parallel to the robot while the front legs of the robot are placed on a fixed bridge, see Figure 3 (d). This scenario is not present during training while the learned policy is able to adapt and choose not to move the front legs while adjusting the rear legs based on the movement of the treadmill.

Banana peel stability test   We test the robot in another scenario where the treadmill is not moving while a frictionless mat is placed under a foot, represented by a banana peel in Figure 3 (e). The only baseline controller that can recover is the heuristic controller, where the slipping foot is adjusted. The learned controller performs similarly to the heuristic controller even though it never sees this situation during training. One can also pass this test with a freeze-joint standing controller, but we emphasize that the high-level contact adjustment can improve the robustness without changing the low-level controller.

4.5   Sim-to-Real Test

We validate the learned controller on the physical robot; snapshots of the experiment are shown in Figure 6. Due to the robustness of the low-level controller, we observe that the sim-to-real gap is small compared to other approaches [3], and the controller is able to perform well without tuning.

Walking Forward   To emulate the treadmill, we send the low-level controller command speeds so that the body will move forward and the high-level controller will need to choose primitives to stay balanced. At low speed, the high-level controller first adopts the *Stand* primitive with the body leaning forward; as the robot is close to falling over, other primitives are used to move the leg forward to regain balance. At high speed, the robot mostly uses the *Trot* primitives.

Leg Perturbation   We perturb the legs of the robot by manually pulling them in different directions. The learned controller is able to adopt the corresponding *Step* primitive to move the perturbed leg back to the nominal position while keeping the unperturbed legs still.

5   Conclusion

We have presented a hierarchical framework that combines model-based control and reinforcement learning. By leveraging the advantages of both paradigms, we obtain a contact-adaptive controller that is more robust and energy efficient than those employing a fixed contact sequence. The learned controller generates novel contact sequences that are generally not produced by either approach alone, at least in the context of real-time control. We demonstrate our framework using a Laikago quadruped in various challenging scenarios such as walking on a split-belt treadmill with only one side moving or stepping onto a "banana peel." We also validate the controller on the physical robot, finding that sim-to-real transfer is relatively straightforward. We believe this is a promising step toward combining the best features of model-based control and reinforcement learning.

References

[1] D. F. Hoyt and C. R. Taylor. Gait and the energetics of locomotion in horses. *Nature*, 292 (5820):239–240, 1981.

[2] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim. Dynamic locomotion in the mit cheetah 3 through convex model-predictive control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[3] X. B. Peng, E. Coumans, T. Zhang, T.-W. Lee, J. Tan, and S. Levine. Learning agile robotic locomotion skills by imitating animals, 2020.

[4] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli. Real-time motion planning of legged robots: A model predictive control approach. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 577–584. IEEE, 2017.

[5] A. W. Winkler, D. C. Bellicoso, M. Hutter, and J. Buchli. Gait and trajectory optimization for legged systems through phase-based end-effector parameterization. *IEEE Robotics and Automation Letters (RA-L)*, 3:1560–1567, July 2018. doi:10.1109/LRA.2018.2798285.

[6] C. Mastalli, R. Budhiraja, W. Merkt, G. Saurel, B. Hammoud, M. Naveau, J. Carpentier, S. Vijayakumar, and N. Mansard. Crocoddyl: An efficient and versatile framework for multi-contact optimal control. *arXiv preprint arXiv:1909.04947*, 2019.

[7] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[8] 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.

[9] M. Posa, C. Cantu, and R. Tedrake. A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research*, 33(1):69–81, 2014.

[10] Z. Manchester and S. Kuindersma. Variational contact-implicit trajectory optimization. In *Robotics Research*, pages 985–1000. Springer, 2020.

[11] I. Mordatch, E. Todorov, and Z. Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 31(4):1–8, 2012.

[12] Laikago website. URL http://www.unitree.cc/e/action/ShowInfo.php?classid=6&id=1#.

[13] C. Boussema, M. J. Powell, G. Bledt, A. J. Ijspeert, P. M. Wensing, and S. Kim. Online gait transitions and disturbance recovery for legged robots via the feasible impulse set. *IEEE Robotics and Automation Letters*, 4(2):1611–1618, 2019.

[14] Z. Xie, P. Clary, J. Dao, P. Morais, J. Hurst, and M. van de Panne. Learning locomotion skills for cassie: Iterative design and sim-to-real. In *Proc. Conference on Robot Learning (CORL 2019)*, 2019.

[15] W. Yu, J. Tan, Y. Bai, E. Coumans, and S. Ha. Learning fast adaptation with meta strategy optimization. *IEEE Robotics and Automation Letters*, 5(2):2950–2957, 2020.

[16] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[17] D. Precup. Temporal abstraction in reinforcement learning. 2001.

[18] T. Li, K. Srinivasan, M. Q.-H. Meng, W. Yuan, and J. Bohg. Learning hierarchical control for robust in-hand manipulation. *arXiv preprint arXiv:1910.10985*, 2019.

[19] Z. Su, O. Kroemer, G. E. Loeb, G. S. Sukhatme, and S. Schaal. Learning to switch between sensorimotor primitives using multimodal haptic signals. In *International Conference on Simulation of Adaptive Behavior*, pages 170–182. Springer, 2016.

[20] X. B. Peng, G. Berseth, and M. van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Trans. Graph.*, 35(4):81:1–81:12, July 2016. ISSN 0730-0301. doi:10.1145/2897824.2925881. URL http://doi.acm.org/10.1145/2897824.2925881.

[21] O. Nachum, M. Ahn, H. Ponte, S. Gu, and V. Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real. *arXiv preprint arXiv:1908.05224*, 2019.

[22] X. B. Peng, G. Berseth, K. Yin, and M. van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017.

[23] O. Nachum, S. S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3303–3313, 2018.

[24] D. Jain, A. Iscen, and K. Caluwaerts. Hierarchical reinforcement learning for quadruped locomotion. *arXiv preprint arXiv:1905.08926*, 2019.

[25] 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*, 5(2):3699–3706, 2020.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[27] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[28] S. Fujimoto, H. Van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[29] S. Gangapurwala, A. Mitchell, and I. Havoutis. Guided constrained policy optimization for dynamic quadrupedal robot locomotion. *IEEE Robotics and Automation Letters*, 5(2):3642–3649, 2020.

[30] NVIDIA. *Isaac Gym - Preview Release*, 2020. URL https://developer.nvidia.com/isaac-gym.

[31] J. Liang, A. Handa, K. Van Wyk, V. Makoviychuk, O. Kroemer, and D. Fox. In-hand object pose tracking via contact feedback and gpu-accelerated robotic simulation. *arXiv preprint arXiv:2002.12160*, 2020.

[32] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

A   Linearized Centroidal Dynamics

The dynamics is similar to [2] with a few modifications. The general centroidal dynamics is

$$\ddot{p}_{body} = \frac{\sum_{i=1}^{4} f_i}{m} - g, \tag{8}$$

$$\mathbf{I}\dot{\omega} = \sum_{i=1}^{4} p_i \times f_i, \tag{9}$$

where $\ddot{p}_{body}$ is the base linear acceleration, f_i is the ground reaction force on each foot, and m, g are the mass and gravity vector respectively. The $\mathbf{I}$ is the mass inertia, $\dot{\omega}$ is the derivative of angular velocity. The p_i is the foot position respect to the base. All variables are represented in the world frame. We ignore the Coriolis force $\omega \times (\mathbf{I}\omega)$ since it does not contribute significantly to the dynamics of the robot.

We use the small angular assumption to linearize the dynamics. The robot's orientation is expressed as a vector of Z-Y-X Euler angles $\Theta = [\phi \ \theta \ \psi]^\top$, where ϕ is roll, θ is pitch, and ψ is yaw. For small values of roll and pitch (ϕ, θ), the angular velocity is approximated by

$$\omega \approx \mathbf{R}_z(\psi)\dot{\Theta}, \tag{10}$$

where

$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

is the rotation matrix of yaw. The inertia matrix in the world frame can be approximated by

$$\mathbf{I} \approx \mathbf{R}_z(\psi) \ _\mathcal{B}\mathbf{I} \ \mathbf{R}_z^\top(\psi), \tag{11}$$

where $_\mathcal{B}\mathbf{I}$ is the inertia matrix in body frame.

The linearized dynamic is

$$\ddot{q} = \mathbf{M}f - \begin{bmatrix} g \\ 0_3 \end{bmatrix}, \tag{12}$$

where

$$\mathbf{M} = \begin{bmatrix} 1_3/m & \cdots & 1_3/m \\ \mathbf{R}_z^\top \ _\mathcal{B}\mathbf{I}^{-1} [p_1]_\times & \cdots & \mathbf{R}_z^\top \ _\mathcal{B}\mathbf{I}^{-1} [p_4]_\times \end{bmatrix}. \tag{13}$$

B Q-Learning Algorithm

We use DQN like algorithm to train our high-level policy. Details are shown in Algorithm 1.

Algorithm 1: Q Learning

---

initialization Q-function parameters $\theta_1.\theta_2$ for $Q_{\theta_1}, Q_{\theta_2}$, empty replay buffer D ;

set target network parameters $\theta_{targ,1}, \theta_{targ,2} \leftarrow \theta_1, \theta_2$ for $Q_{\theta_{targ,1}}, Q_{\theta_{targ,2}}$;

while *not done* **do**

    observe current state s ;

    sample action a based on Q-function;

    observe next state s', reward r and done signal d;

    store (s, a, r, d, s') in replay buffer D;

    if *d is True or time limit reached* **then**

        reset environment;

    end

    if *time to update* **then**

        for $j = 1, 2, \ldots$ *number of update* **do**

            sample batch of transition data $B = \{s, a, r, d, s'\}$;

            compute $a' = \arg\max_a Q_\theta(s', a)$;

            compute target $q_{targ} = r + (1 - d)\gamma \min_{i=1,2}(Q_{\theta_{targ,i}}(s', a'))$;

            update θ_1, θ_2 by taking gradient descent w.r.t the objective function
$\frac{1}{|B|} \sum_{(s,a,r,d,s') \in B}((Q_{\theta_1}(s, a) - q_{targ})^2 + (Q_{\theta_2}(s, a) - q_{targ})^2)$;

            if $j \mod 2 = 1$ **then**

                $\theta_{targ,1} \leftarrow \rho\theta_{targ,1} + (1 - \rho)\theta_1$;

                $\theta_{targ,2} \leftarrow \rho\theta_{targ,2} + (1 - \rho)\theta_2$;

            end

        end

    end

end
