Data Efficient Reinforcement Learning for Legged Robots

Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, Vikas Sindhwani
Robotics at Google
United States
{yxyang, kencaluwaerts, atil, tingnan, jietan, sindhwani}@google.com

Abstract: We present a model-based reinforcement learning framework for robot locomotion that achieves walking based on only 4.5 minutes of data collected on a quadruped robot. To accurately model the robot's dynamics over a long horizon, we introduce a loss function that tracks the model's prediction over multiple timesteps. We adapt model predictive control to account for planning latency, which allows the learned model to be used for real time control. Additionally, to ensure safe exploration during model learning, we embed prior knowledge of leg trajectories into the action space. The resulting system achieves fast and robust locomotion. Unlike model-free methods, which optimize for a particular task, our planner can use the same learned dynamics for various tasks, simply by changing the reward function.[1] To the best of our knowledge, our approach is more than an order of magnitude more sample efficient than current model-free methods.

Keywords: Legged Locomotion, Model-based Reinforcement Learning, Model Predictive Control

1 Introduction

Robust and agile locomotion of legged robots based on classical control stacks typically requires accurate dynamics models, human expertise, and tedious manual tuning [1, 2, 3, 4]. As a potential alternative, model-free reinforcement learning (RL) algorithms optimize the target policy directly and do not assume prior knowledge of environmental dynamics. Recently, they have enabled automation of the design process for locomotion controllers [2, 5, 6, 7, 8]. Yet, all too often, progress with model-free methods is only demonstrated in simulated environments [9, 10], due to the amount of data required to learn meaningful gaits. Attempting to take these methods to physical legged robots presents major challenges: Namely, how to mitigate the laborious and time-consuming data collection process [11], and how to minimize hardware wear and tear during exploration? Additionally, what the robot learns is often a task-specific policy. As a result, adapting to new tasks typically involves finetuning based on new rounds of robot experiments [12].

In this paper, we propose a model-based learning [13, 14, 15] framework that significantly improves sample efficiency and task generalization compared to model-free methods. The key idea is to learn a dynamics model from data and consequently plan for action sequences according to the learned model. While model-based learning is commonly considered as a more sample-efficient alternative to model-free methods, its successful application to legged locomotion has been limited [16]. Our main challenges are threefold. First, the learned model needs to be sufficiently accurate for long-horizon planning, as an inaccurate model can dramatically degrade the performance of the final controller. This is particularly evident for locomotion due to frequent and abrupt contact events. The predicted and real trajectories can quickly diverge after a contact event, even if the single-step model error is small. The second challenge is real-time action planning at a high control frequency. To maintain balance, locomotion controllers often run at a frequency of hundreds or even thousands of times per second. Therefore, even a short latency in action planning can significantly affect the performance of controller. Finally, safe data collection for model learning is nontrivial. To ensure

[1] A video showing the learning process and results can be found at: https://youtu.be/oB9IXKmdGhc

3rd Conference on Robot Learning (CoRL 2019), Osaka, Japan.

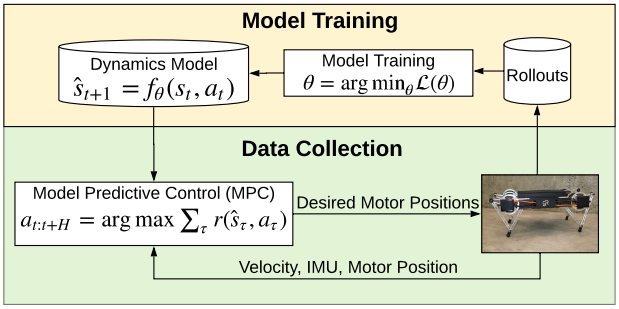


Figure 1: Overview of our learning system on the robot. The system alternates between collecting trajectories and learning a dynamics model.

sufficient exploration, RL algorithms typically drive the actuators using random noise. However, such random actuation patterns can impose a lot of stress on the actuators and cause mechanical failures, especially during the initial stages of training.

Our proposed algorithm addresses the above challenges. During model learning, we use multi-step loss to prevent accumulation of errors in long-horizon prediction. To achieve real-time planning, we parallelize a sampling-based planning algorithm on a GPU. Additionally, we plan actions based on a predicted future state using the learned dynamics model to compensate for planning latency. We develop safe exploration strategies using a trajectory generator [17], which ensures that the planned actions are smooth and do not damage the actuators. Combining these three improvements with model-based learning, stable locomotion gaits can be learned efficiently on a real robot.

The main contribution of our paper is a highly efficient learning framework for legged locomotion. With our framework, a Minitaur robot can successfully learn to walk from scratch after 36 rollouts, which corresponds to 4.5 minutes of data (45,000 control steps) or approximately 10 minutes of robot experimentation time (accounting for the overhead of robot setup). To the best of our knowledge, this is at least an order of magnitude more sample efficient than the state-of-the-art on-robot learning method using the same hardware platform [11]. More importantly, we show that the learned model can generalize to new tasks without additional data collection or fine tuning.

2 Model-learning and Model-Predictive Control Loop

We formulate the locomotion problem as a Markov Decision Process (MDP) defined by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a state transition distribution $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$, an initial state distribution $p(\boldsymbol{s}_0)$ and a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We apply model-based RL to solve this MDP, which learns a deterministic dynamics model f_θ to approximate $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ by fitting to collected trajectories. The learned model estimates the next state given the current state-action pair, and is used by an action planner to optimize the cumulative reward. To account for model inaccuracies, we use a model predictive control (MPC) framework that periodically replans based on the latest robot observation.

Using learned models for action planning raises extra challenges for model accuracy. Although a learned dynamics model can generally remain accurate around trajectories in the training data, its performance for unseen state-actions is not guaranteed. As a result, the planner might exploit such imperfections in the model and optimize for actions that are actually suboptimal on the robot. To compensate for this distribution mismatch between training and testing data, we keep track of all collected trajectories in a replay buffer and periodically retrain the model using all trajectories [18]. The updated model is then used to collect more trajectories from the robot, which are added to the replay buffer for future model training (Fig. 1). By interleaving model training and data collection, we improve the model's accuracy on parts of the state space where the planner is more likely to visit, which in turn increases the quality of the plan.

3 Model-based Learning for Locomotion

3.1 Accurate Dynamics Modeling with Multi-step Loss

We model the difference between consecutive states as a function $f_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{s}_{t+1} - \boldsymbol{s}_t$, where f_θ is a feed-forward neural network with weights θ. Given a set of state transitions $\mathcal{D} = \{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})\}$, a standard way to train the model is to directly minimize the prediction error:

$$\mathcal{L}_{\text{single-step}}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) \in \mathcal{D}} \left\| (\boldsymbol{s}_{t+1} - \boldsymbol{s}_t) - f_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t) \right\|_2^2. \tag{1}$$

Although Eq. 1 ensures the model's accuracy for one time step, it does not prevent the accumulation of errors over longer planning horizons. In previous works, ensembles of models have been exploited to reduce uncertainty and improve the model's long-term accuracy [19, 20]. However, prediction using ensembles of models can significantly increase the planning time. Instead, we use a multi-step loss function [21, 22] to combat the accumulation of model errors. From the current state $\boldsymbol{s}_t$, we use the learned dynamic model f_θ to predict the next n steps $\{\hat{\boldsymbol{s}}_{t+1}, \hat{\boldsymbol{s}}_{t+2}, ..., \hat{\boldsymbol{s}}_{t+n}\}$ where $\hat{\boldsymbol{s}}_{t+\tau+1} = \hat{\boldsymbol{s}}_{t+\tau} + f_\theta(\hat{\boldsymbol{s}}_{t+\tau})$. Note that the prediction of $\hat{\boldsymbol{s}}_{t+\tau+1}$ is based on the predicted state of $\hat{\boldsymbol{s}}_{t+\tau}$, not on the true state $\boldsymbol{s}_{t+\tau}$, except when $\tau = 0$. This is important because this recursively predicted sequence of states allow us to define a multi-step loss function that measures the accumulation of errors over time.

$$\mathcal{L}_{\text{multi-step}}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{s}_{t:t+n}, \boldsymbol{a}_{t:t+n-1}) \in \mathcal{D}} \frac{1}{n} \sum_{\tau=1}^{n} \left\| (\boldsymbol{s}_{t+\tau} - \boldsymbol{s}_{t+\tau-1}) - f_\theta(\hat{\boldsymbol{s}}_{t+\tau-1}, \boldsymbol{a}_{t+\tau-1}) \right\|_2^2, \tag{2}$$

Note that when $n = 1$, Eq. 2 reduces to the single-step loss. As n increases, the loss focuses on the accuracy of the model over multiple steps, making the learned model suitable for long-horizon planning. We empirically validate the effect of multi-step loss in Section 5.4.1.

3.2 Efficient Planning of Smooth Actions

We use a model predictive control (MPC) framework to plan for optimal actions. Instead of optimizing for the entire episode offline, MPC replans periodically using the most recent robot state, so that the controller is less sensitive to model inaccuracies. Since replanning happens simultaneously with robot execution, the speed of the planning algorithm is critical to the performance of MPC.

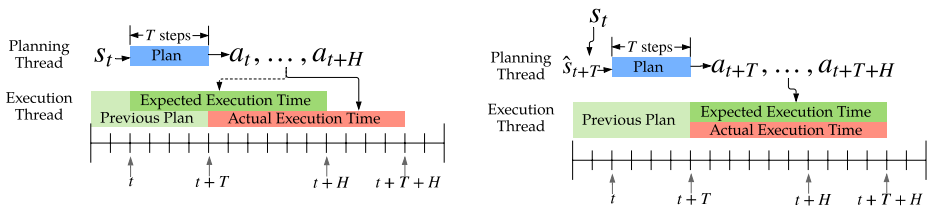
With handcrafted models, a number of efficient planning algorithms have been tested for robot locomotion [23, 24, 25]. However, they either assume a linear dynamics model, or compute model gradients for linear approximations, which is costly to evaluate for neural networks. Instead, we use the Cross Entropy Method (CEM) to plan for optimal actions [26]. CEM is an efficient, derivative-free optimization method that is easily parallelizable and less prone to local minima. It has demonstrated good performance in optimizing neural network functions [19, 27] and can handle non-smooth reward functions. To compute an action plan, CEM samples a population of action sequences at each iteration, fits a normal distribution to the best samples, and samples the next population from the updated distribution in the next iteration.

Sampling each action independently in an H-step action sequence is unlikely to generate high quality plans. While good plans often consist of actions that are smooth and periodic, time-independent samples are more likely to produce jerky motions, making it difficult for CEM to select smooth actions. Instead, we apply a filter to smooth out the noises added to the mean action. Given a filter coefficient $\gamma \in [0, 1]$, we first generate H time-correlated samples $\boldsymbol{n}_1, \ldots, \boldsymbol{n}_H$ from H i.i.d. samples $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_H \sim \mathcal{N}(0, 1)^{\dim(\mathcal{A})}$:

$$\boldsymbol{n}_1 = \boldsymbol{u}_1 \tag{3}$$

$$\boldsymbol{n}_t = \gamma \boldsymbol{n}_{t-1} + \sqrt{1 - \gamma^2} \boldsymbol{u}_t. \tag{4}$$

Given the mean and standard deviation of the action sequences $\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t, t \in \{1, \ldots, H\}$, the actions are computed as $\boldsymbol{a}_t = \boldsymbol{\mu}_t + \boldsymbol{\sigma}_t \circ \boldsymbol{n}_t$. Although each $\boldsymbol{a}_t$ still follows the desired normal distribution $\mathcal{N}(\boldsymbol{\mu}_t, \text{diag}(\boldsymbol{\sigma}_t))$, the exploration noise in consecutive samples is now time-correlated, which makes it less likely to damage the actuators.



(a) Without asynchronous control. (b) With asynchronous control.

Figure 2: Timing diagram of our asynchronous controller. (a) The planner outputs $\boldsymbol{a}_t, ..., \boldsymbol{a}_{t+H}$ given the current state $\boldsymbol{s}_t$. However, due to the planning latency T, the action $\boldsymbol{a}_t$ is actually executed at time $t+T$, on the unplanned state $\boldsymbol{s}_{t+T}$, which leads to a suboptimal action. (b) Our system predicts the state $\hat{\boldsymbol{s}}_{t+T}$ when the planning completes, and uses it as the input to the planner. As a result, the planned action $\boldsymbol{a}_{t+T}$ is executed on the right state $\boldsymbol{s}_{t+T}$.

3.3 Online Replanning in the Presence of Latency

In classic MPC, replanning happens at every timestep, and only the first action of the planned action sequence is executed. As a result, the planning frequency dictates the control frequency. Since planning usually takes much longer than execution, the reduced control frequency will severely limit the capabilities of the controller. We decouple the planning and the control frequencies by parallelizing these two loops. While the planning thread is optimizing for actions in the background, the execution thread simultaneously applies the actions computed in the previous planning step at a higher control frequency.

Another problem of the long planning time, or the *planning latency*, is that the current state $\boldsymbol{s}_t$ used by the planner is out-dated when the planning step finishes at $t+T$ (Fig. 2a), where T is the computation time needed for planning. In the presence of this latency T, we actually apply $\boldsymbol{a}_t$ at $\boldsymbol{s}_{t+T}$, which is not optimal. Thus instead of using $\boldsymbol{s}_t$, we should use $\boldsymbol{s}_{t+T}$ as the input to the planner. Although $\boldsymbol{s}_{t+T}$ is a future state that we cannot observe at time t, we can predict it using the learned model f_θ and the actions from the previous planning step. We feed this predicted future state, $\hat{\boldsymbol{s}}_{t+T}$ to the planner, and the output actions $\boldsymbol{a}_{t+T}, \ldots, \boldsymbol{a}_{t+T+H}$, are then executed when the planning step completes at $t+T$ (Fig. 2b). This technique, which we call *asynchronous control*, precisely aligns the state that the planner uses and the state when the planned actions will be executed. This provides the planner with a more accurate estimation of the state during execution, and significantly increases the plan quality in the presence of latency.

4 Safe Exploration with Trajectory Generators

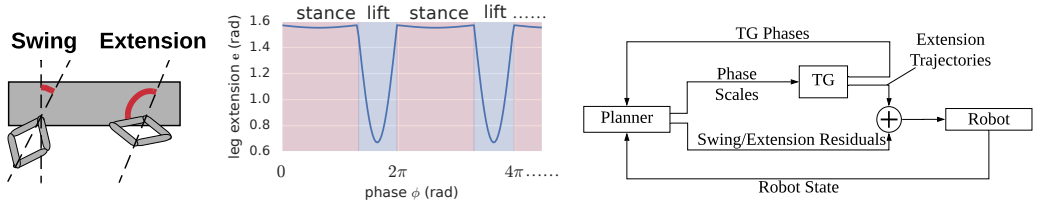
Whereas formulating the action space using desired motor angles is easier to learn [28], controlling the motors in position control mode can result in abrupt changes in desired motor angles, which may cause large torque output that could potentially damage the robot and its surroundings. Instead, we use trajectory generators (TGs) to encourage smooth trajectories. Similar to [17], TGs output periodic trajectories in the extension space of each leg (Fig. 3a), and can be modulated by the planner for more complex behaviors.

We use four independent trajectory generators to control all four legs of the robot. Each TG maintains an internal phase $\phi \in [0, 2\pi)$ and controls the leg extension e following a periodic function:

$$e = c_e + a' \cdot \sin(\phi'), \text{ where } a', \phi' = \begin{cases} a_{\text{stance}}, \frac{\phi}{\phi_{\text{stance}}}\pi & \phi < \phi_{\text{stance}} \\ a_{\text{lift}}, \left(1 + \frac{\phi - \phi_{\text{stance}}}{2\pi - \phi_{\text{stance}}}\right)\pi & \phi \geq \phi_{\text{stance}} \end{cases} \tag{5}$$

Here $c_e, a_{\text{stance}}, a_{\text{lift}}, \phi_{\text{stance}}$ are parameters for the TG. As the phase ϕ evolves, the TG alternates between the stance mode ($\phi < \phi_{\text{stance}}$) and lift mode ($\phi \geq \phi_{\text{stance}}$). We choose a different amplitude $a_{\text{stance}}, a_{\text{lift}}$ for each mode of the TG, and rescale the original phase to ϕ' so that the resulting leg extension is a continuous function. Note that the TGs do not control the leg swing angles. As a result, our planner starts with an open-loop TG that generates an in-place stepping gait (Fig. 3b).

We augment the state and action space of the environment so that the planner can interact with TGs (Fig. 3c). Our new action space is 12 dimensional. The first 8 dimensions correspond to the



(a) Action space. (b) Leg extension of open-loop TG. (c) Interaction between planner and TG.

Figure 3: Illustration of TGs and their interaction with the planner.

swing and extension residual of each leg, which is added to the TG outputs before the command is sent to the robot. The residuals allow the planner to complement the TG outputs for more complex behaviors. The remaining 4 dimensions specify the phase scales $\omega_{1\ldots4}(t)$ for each TG at time t, so that the phase of each TG can be propagated independently $\phi_i(t+1) = \phi_i(t) + \omega_i(t)\Delta t$. This gives the controller additional freedom to synchronize arbitrary pairs of legs and coordinate for varied gait patterns. Finally, we augment the state space with the phase of each TG to make the state of TGs fully observable.

5 Experiments

5.1 Experimental Setup

We use the Minitaur robot from Ghost Robotics [29] as the hardware platform for our experiments. We run our controller with a timestep of 6ms. Similar to [5], the controller outputs desired swing and extension of each leg, which is converted to desired motor positions and tracked by a Proportional Derivative (PD) controller.

We include base linear velocity, IMU readings (roll, pitch, and yaw), and motor positions in the state space of the robot, where the readings come from motion capture (PhaseSpace Inc. Impulse X2E) and on-board sensors. The state space is 18-dimensional (TG state and sensors). Similar to [11], we concatenate a history of the past four observations as the input to our dynamics model to account for hardware latency and partial observability of the system. The dynamics are modeled as a feed-forward neural network with one hidden layer of 256 units and tanh activation. We choose $n = 20$ as the number of steps to propagate the model and compute the loss.

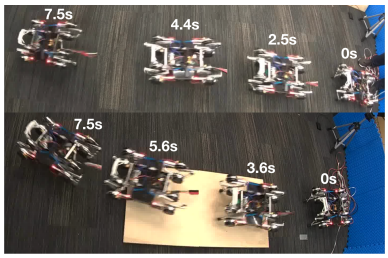
For MPC, we run CEM for 5 iterations with 400 samples per iteration and a planning horizon of 75 control steps (450 ms). We implement our algorithm in JAX [30] for compiled execution and run the algorithm on a Nvidia GTX 1080 GPU. With software and hardware acceleration, our CEM implementation executes in less than 60ms. We replan every 72ms to handle model inaccuracies.

In all experiments where we collect data to train the model, the robot's task is to walk forward following a desired speed profile over an episode of 7.5 seconds. The desired speed starts at zero and increases linearly to a top speed of 0.66 m/s within the first 3 seconds, and remains constant for the rest of the episode. The reward fuction is $r = -|v - \tilde{v}| - 0.001|y| - 0.01(r^2 + p^2)$, where v and $\tilde{v}$ are the current and desired walking speed, and (r, p, y) are the roll, pitch, and yaw of the base. The second term encourages walking forward, and the last term stabilizes the base during walking.

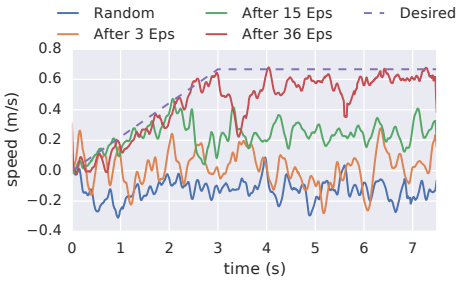
5.2 Learning on Hardware

Our method successfully learns a dynamics model based on data from a real robot and optimizes a forward walking gait in only 36 episodes (45,000 control steps), which corresponds to approximately 10 minutes of robot time, including rollouts, data collection, and experiment resets (Fig. 4a top). We update the dynamics model every 3 episodes. The robot tracks a desired speed of 0.66 m/s (Fig. 4b), or 1.6 body lengths per second, which is twice the fastest speed achieved by [11]. The entire learning process, including data collection and offline model training, takes less than one hour to complete. The attached video illustrated the learning process.

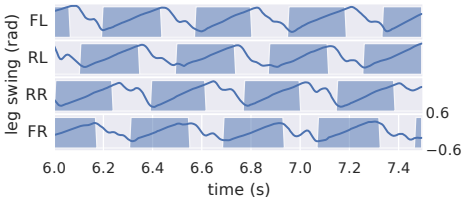
It is important to interleave data collection and model training, and update the dynamics model using new data (Fig. 4b). Initially, when trained only on random trajectories, the model cannot predict the robot dynamics accurately, and MPC only achieves a slow forward velocity. As more



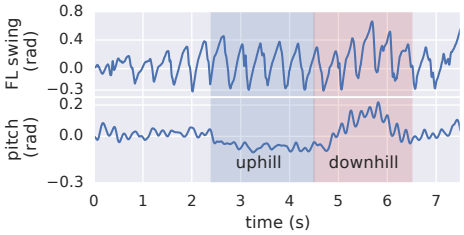
(a) Walk on flat terrain (top) and slope (bottom).



(b) Tracking of desired speed.



(c) Final gait (dark blue indicates stance phase).



(d) Robot states while walking on a slope.

Figure 4: Learning on real robot. (4a) The robot walks on different terrains. (4b) The robot gradually tracks the desired speed profile. (4c) Swing angles and gait pattern of all four legs. (4d) Robot trajectory when the robot walks up and down a slope.

data is collected, the model becomes more accurate in the part of the state space which the planner is likely to utilize, leading to better planning performance.

Periodic and distinctive gait patterns develop as the training proceeds (Fig. 4c). With TGs providing the underlying trajectory, MPC swings the legs forward in the lift phase and backward in the stance phase, leading to a periodic forward-walking behavior. Note that TGs affect leg extensions only, and the leg swing angles are controlled exclusively by MPC. Additionally, the ability for MPC to control the phase of each TG allows individual legs to be coordinated. In the learned gait, MPC swings the four legs in succession, resulting in a walking pattern.

We also test the robustness of MPC on an unseen terrain. We place a slope in the robot's path (Fig. 4a bottom). Although the robot has not trained on the slope, it still can maintain a periodic gait using MPC (Fig. 4d). The robot's pitch angle shows slight perturbations while walking uphill and downhill, but the robot remains upright most of the time.

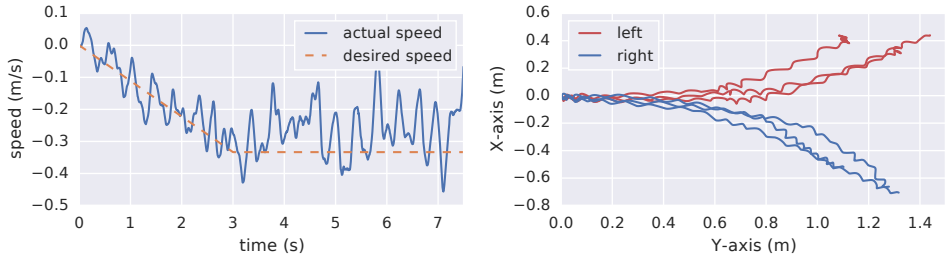
5.3 Generalization to Unseen Tasks

We test the ability of our learned dynamics model to generalize to unseen tasks. We take the dynamics model learned in Section 5.2, which is trained for walking forward, and perform MPC on new tasks with unseen reward functions. For example, to make the robot turn left, we change the reward function to $r = -|v - \tilde{v}| - 0.001\left|\dot{y} - \widetilde{\dot{y}}\right| - 0.01(r^2 + p^2)$ for a desired turning rate $\widetilde{\dot{y}}$, where $\dot{y}$ is approximated by finite difference.

Even though we only train our dynamics model on the task of walking forward, the model is sufficiently accurate to allow MPC to plan for new tasks, including walking backwards and turning (Fig. 5a, 5b). By learning the dynamics instead of the policy, our algorithm achieves zero-shot generalization to related tasks.

5.4 Ablation Study

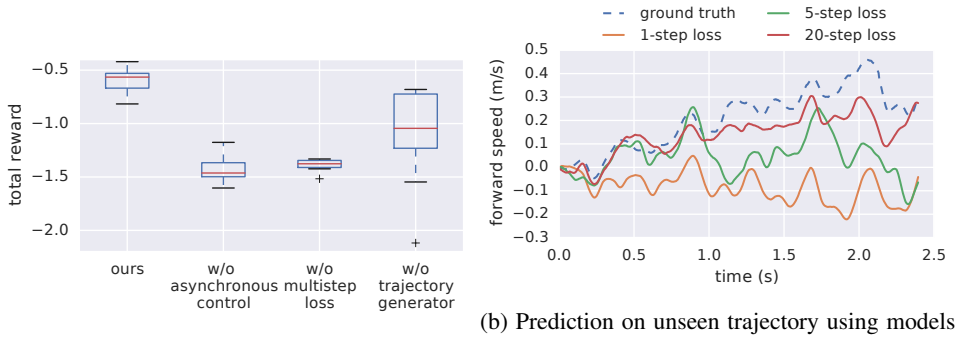
To evaluate the importance of the key components of our algorithm, including multi-step loss, asynchronous control and trajectory generators, we perform an ablation study in a highly accurate, open-source simulation of Minitaur [5]. Simulations help us collect a larger amount of data and reduce the variance of analysis due to algorithmic and environmental stochasticity.



(a) Tracking of a desired backward speed.

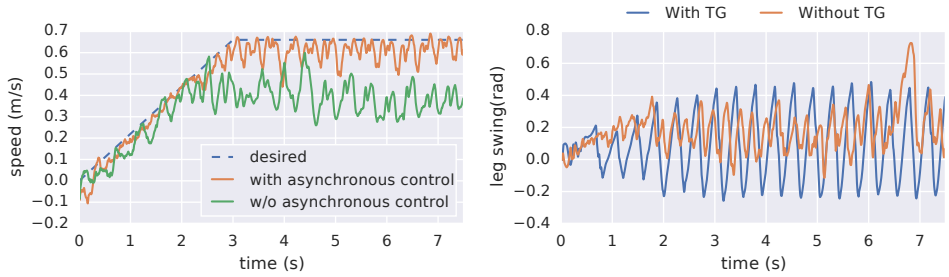
(b) Trajectory of robot turning in x-y plane.

Figure 5: Generalization of MPC to unseen reward functions using the existing dynamics model. In both cases, the dynamics model is trained only on the task of forward-walking. In 5a, the new cost function is to track a desired backward speed. In 5b, the new cost function is to keep the same forward speed while turning left or right at a rate of 15 degrees per second.



(a) Final return using various methods.

(b) Prediction on unseen trajectory using models trained with different loss functions.



(c) Tracking desired speed with or without asynchronous control.

(d) Swing angle of front left leg during MPC, where the model is trained with or without TG.

Figure 6: Ablation study of multistep loss, asynchronous control, and trajectory generators.

5.4.1 Dynamics Modeling with Multi-step Loss

We find that the number of steps (n in Eq. 2) to compute the model loss $\mathcal{L}_{\text{multi-step}}$ is an important hyperparameter that affects the model accuracy. Without multi-step loss, the model cannot accurately track the robot dynamics over a long horizon and the MPC controller does not achieve a high reward (Fig. 6a). We further validate this by training models using multiple values of n and testing their performances on an unseen trajectory (Fig. 6b). With more timesteps propagated in computing the model loss, the trained model tracks the ground truth trajectory increasingly better. Note that the plotted state is the velocity of the robot, for which the planner directly optimizes. Inaccurate estimation of the robot velocity is likely to result in suboptimal planning. We choose $n = 20$ as a tradeoff between model accuracy and training time.

5.4.2 Asynchronous CEM Controller

Planning with asynchronous control is important for fast locomotion (Fig. 6c). Without asynchronous control, the MPC controller could only track the desired speed up to approximately 0.4m/s.

As the robot moves faster, the robot states can change rapidly even within a few timesteps. Therefore, it is important to perform planning with respect to an accurate state. This is also illustrated in Fig. 6a, where the system struggles to achieve a good final reward without asynchronous control.

We identify additional important hyperparameters for CEM in Table. 1: CEM requires at least 5 iterations for optimal performance. While smoothing out sampled actions can significantly improve the plan quality, excessive smoothing can make the legs overly compliant for dynamic behaviors. It is important to plan over a sufficiently long horizon to optimize for long-term return. On the other hand, planning for too long makes the planner susceptible to imperfections of the model.

(a) Number of CEM iterations.

# Iterations	1	3	**5**	10
Return	-1.83	-0.95	**-0.44**	-0.43

(b) Smoothing parameter (γ in Eq. 3).

Smoothing	0	0.3	**0.5**	0.9
Return	-1.38	-0.80	**-0.44**	-1.65

(c) CEM planning horizon (ms).

Horizon	150	300	**450**	600
Return	-2.44	-0.40	**-0.44**	-0.68

Table 1: Ablation study on various parameters of CEM. All rollouts share the same dynamics model. Results show average return over 5 episodes. In bold: selected values.

5.4.3 Role of Trajectory Generators

The TGs play an important role in regulating planned actions and ensuring periodicity of leg motion. In Fig. 6d, we compare the final behavior of MPC using models trained with and without TG. While both rollouts achieve similar total reward, planning with TG smooths the motor actions and makes the leg behavior periodic. The learning process is also less stable without TG (Fig. 6a). We attempted to learn a model without TG on the real robot. The motors overheated quickly and the jerky motions damaged the motor mounts, forcing us to stop the experiment early.

5.4.4 Comparison with Model-free Algorithms

We compare the sample efficiency of our model-based learning method with model-free ones (Fig. 7). We obtain the implementations of model-free algorithms from TF-Agents [31]. As a state-of-the-art on-policy algorithm, Proximal Policy Optimization (PPO) [32] achieves a similar reward but requires nearly 1000 times more samples, making it difficult to run on the real robot. While the off-policy method, Soft Actor Critic (SAC) [33], significantly improved sample efficiency and has been demonstrated to learn walking on a Minitaur robot [11], it still requires an order of magnitude more samples compared to our method, with a less stable learning curve.

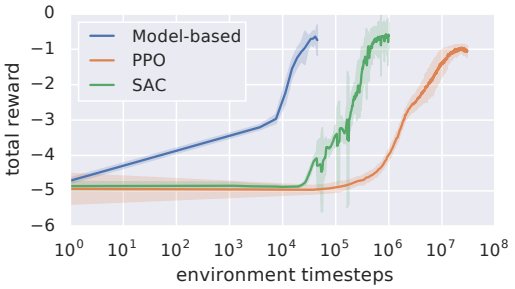


Figure 7: Learning curve of our model-based algorithm compared to model-free ones. Note tha x-axis is on log-scale.

6 Perspectives and Future Work

The combination of accurate long-horizon dynamics learning with multi-step loss functions, careful handling of real-time requirements by compensating for planning latency, and embedding periodicity priors into MPC walking policies, yields an approach that requires only 4.5 minutes of real-world data collection to induce robust and fast gaits on a quadruped robot. Such learning efficiency is more than an order of magnitude superior to model-free methods. The learnt dynamics model can then be reused to induce new locomotion behaviors.

Yet, many questions remain to be answered in future work: How can rigid-body dynamics be best combined with function approximators for even greater sample efficiency, how should predictive controllers be made aware of model misspecification, and how should predictive uncertainty be best captured and exploited for improved exploration and real-time online adaptation to enable more agile and complex behaviors? Interfacing vision, contact sensing and other perceptual modules with an end-to-end model learning and real-time planning stack is also critical for greater autonomy.

References

[1] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 40(3):429–455, 2016.

[2] M. Hutter, C. Gehring, A. Lauber, F. Gunther, C. D. Bellicoso, V. Tsounis, P. Fankhauser, R. Diethelm, S. Bachmann, M. Blösch, et al. Anymal-toward legged robots for harsh environments. *Advanced Robotics*, 31(17):918–931, 2017.

[3] G. Bledt, M. J. Powell, B. Katz, J. Di Carlo, P. M. Wensing, and S. Kim. MIT cheetah 3: Design and control of a robust, dynamic quadruped robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2245–2252, Oct 2018.

[4] S. Kim, P. M. Wensing, et al. Design of dynamic legged robots. *Foundations and Trends® in Robotics*, 5(2):117–190, 2017.

[5] 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. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[6] Z. Xie, G. Berseth, P. Clary, J. Hurst, and M. van de Panne. Feedback control for cassie with deep reinforcement learning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1241–1246. IEEE, 2018.

[7] 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):eaau5872, 2019.

[8] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2004.

[9] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami, M. Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[10] 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 (TOG)*, 36(4): 41, 2017.

[11] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. In *Proceedings of Robotics: Science and Systems*, FreiburgimBreisgau, Germany, June 2019.

[12] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[13] A. Marjaninejad, D. Urbina-Meléndez, B. A. Cohn, and F. J. Valero-Cuevas. Autonomous functional movements in a tendon-driven limb via limited experience. *Nature machine intelligence*, 1(3):144, 2019.

[14] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[15] M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. *arXiv preprint arXiv:1906.08253*, 2019.

[16] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[17] A. Iscen, K. Caluwaerts, J. Tan, T. Zhang, E. Coumans, V. Sindhwani, and V. Vanhoucke. Policies modulating trajectory generators. In *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 916–926. PMLR, 29–31 Oct 2018.

[18] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[19] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[20] I. Clavera, J. Rothfuss, J. Schulman, Y. Fujita, T. Asfour, and P. Abbeel. Model-based reinforcement learning via meta-policy optimization. In *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 617–629. PMLR, 29–31 Oct 2018.

[21] E. Talvitie. Self-correcting models for model-based reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[22] E. Talvitie. Model regularization for stable sample rollouts. In *UAI*, pages 780–789, 2014.

[23] M. Neunert, M. Stäuble, M. Giftthaler, C. D. Bellicoso, J. Carius, C. Gehring, M. Hutter, and J. Buchli. Whole-body nonlinear model predictive control through contacts for quadrupeds. *IEEE Robotics and Automation Letters*, 3(3):1458–1465, 2018.

[24] 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.

[25] T. Apgar, P. Clary, K. Green, A. Fern, and J. W. Hurst. Fast online trajectory optimization for the bipedal robot cassie. In *Robotics: Science and Systems*, 2018.

[26] R. Y. Rubinstein and D. P. Kroese. The cross-entropy method: A unified approach to monte carlo simulation, randomized optimization and machine learning. *Information Science & Statistics, Springer Verlag, NY*, 2004.

[27] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 651–673. PMLR, 29–31 Oct 2018.

[28] X. B. Peng and M. van de Panne. Learning locomotion skills using deeprl: Does the choice of action space matter? In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, page 12. ACM, 2017.

[29] G. Kenneally, A. De, and D. E. Koditschek. Design principles for a family of direct-drive legged robots. *IEEE Robotics and Automation Letters*, 1(2):900–907, 2016.

[30] R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[31] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Chris Harris, Vincent Vanhoucke, Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. URL `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019].

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.