Policies Modulating Trajectory Generators

Atil Iscen[1] **Ken Caluwaerts**[1] **Jie Tan**[2] **Tingnan Zhang**[2]

Erwin Coumans[2] **Vikas Sindhwani**[1] **Vincent Vanhoucke**[2]

[1]Google Brain, New York, United States
[2]Google Brain, Mountain View, United States
{atil, tensegrity, jietan, tingnan, erwincoumans, sindhwani, vanhoucke}
@google.com

Abstract: We propose an architecture for learning complex controllable behaviors by having simple Policies Modulate Trajectory Generators (PMTG), a powerful combination that can provide both memory and prior knowledge to the controller. The result is a flexible architecture that is applicable to a class of problems with periodic motion for which one has an insight into the class of trajectories that might lead to a desired behavior. We illustrate the basics of our architecture using a synthetic control problem, then go on to learn speed-controlled locomotion for a quadrupedal robot by using Deep Reinforcement Learning and Evolutionary Strategies. We demonstrate that a simple linear policy, when paired with a parametric Trajectory Generator for quadrupedal gaits, can induce walking behaviors with controllable speed from 4-dimensional IMU observations alone, and can be learned in under 1000 rollouts. We also transfer these policies to a real robot and show locomotion with controllable forward velocity.

Keywords: Reinforcement Learning, Control, Locomotion

1 Introduction

The recent success of Deep Learning (DL) on simulated robotic tasks has opened an exciting research direction. Nevertheless, many robotic tasks such as locomotion still remain an open problem for learning-based methods due to their complexity or dynamics. From a Deep Learning (DL) perspective, one way to tackle these complex problems is by using more and more complex policies (such as recurrent networks). Unfortunately, more complex policies are harder to train and require even more training data which is often problematic for robotics.

Robotics is naturally a great playground for combining strong prior knowledge with DL. The robotics literature contains many forms of prior knowledge about locomotion tasks and nature provides impressive examples of similar architectures. Note that this knowledge does not need to be in the form of perfect examples, it can also be in form of intuition about the specific robotic problem. As an example, for locomotion it can be defined as leg movements patterns based on certain gait and external parameters.

We incorporate this intuitive type of prior knowledge into learning in the form of a parameterized Trajectory Generator (TG). We keep the TG separate from the learned policy and define it as a stateful module that outputs actions $\boldsymbol{u}_{tg}$ (e.g. target motor positions) which depend on its internal state and external parameters. We introduce a new architecture in which the policy has control over the TG by modulating its parameters as well as directly correcting its output (Fig. 1). In exchange, the policy receives the TG's state as part of its observation. As the TG is stateful, these connections yield a controller that is implicitly recurrent while using a feed-forward Neural Network (NN) as the learned policy. The advantage of using a feed-forward NN is that learning is often significantly less demanding than with recurrent NNs. Moreover, this separation of the feed-forward policy and the stateful TG makes the architecture compatible with any reward based learning method.

2nd Conference on Robot Learning (CoRL 2018), Zürich, Switzerland.

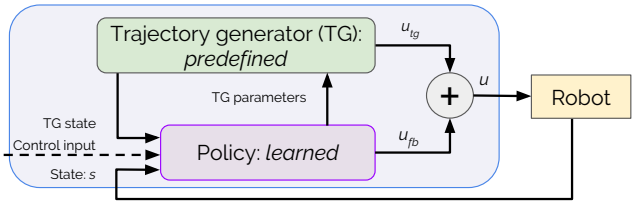


Figure 1: Overview of PMTG: The output (actions) u_{tg} of of a predefined Trajectory Generator (TG) is combined with that of a learned policy network (u_{fb}). The learned policy also modulates the parameters of the TG at each time step and observes its state.

We call our architecture *Policies Modulating Trajectory Generators* (PMTG) to stress the interaction between the learned policy and the predefined TG. In essence, we replace the task of learning a locomotion controller by that of learning to modulate a TG in parallel with learning to control a robot.

In this manuscript, we first illustrate the architecture of PMTG using a synthetic control problem. Next, we tackle quadruped locomotion using PMTG. We use desired speed as the control input and different TGs that generate leg trajectories based on parameters such as stride length, frequency and walking height. We train our policies in simulation using Reinforcement Learning (RL) or Evolutionary Strategies (ES) with policies as simple as one linear layer using only a four-dimensional proprioceptive observation space (IMU). Finally we transfer the learned policies to a real robot and demonstrate learned locomotion with controllable speed.

2 Related Work

Optimization is an effective tool to automatically design locomotion controllers. Popular techniques include Black-box Optimization [1], Bayesian Optimization [2], Evolutionary Algorithms [3, 4] and Reinforcement Learning [5, 6, 7]. In recent works, neural networks are commonly used parameterizations of the control policy. While the architecture of the neural network plays an important role in learning, simple fully-connected feed-forward networks are often used due to the challenges to design network architecture. Although several prior works [8, 9, 10] can automatically search for the optimal architecture, they require tremendous amounts of time and computation. In computer graphics, special architectures, such as Phase-Functioned Neural Networks [11] and Mode-Adaptive Neural Networks [12], have been proposed to synthesize locomotion controllers from motion capture data. While these methods generate vivid animations, they do not consider physics and balance control, and thus, are not suitable for robotics.

Locomotion is periodic and structured motion. Classical locomotion controllers often use a cyclic open loop trajectory, such as sine curves [7, 13] or Central Pattern Generators (CPG) [14] to parameterize the movement of each actuated degrees of freedom. To control locomotion, Gay et al. [15] learned a neural network that modulated a CPG. Sharma and Kitani [16] designed phase-parametric policies by exploiting the cyclic nature of locomotion. Tan et al. [7] learnt a feedback balance controller on top of a user-specified pattern generator. Although our method is inspired by [7], there are key differences. In Tan et al. [7], the pattern generator is fixed and independent to the feedback control. In this way, the feedback control can only modify the gait in the vicinity of the signal defined by pattern generator. In contrast, in our architecture, the feedback control modulates the TG including its frequency. This is crucial since we are interested in dynamically changing the high-level behavior of the locomotion, which requires changing the underlying trajectory and its frequency.

In this paper, our focus is to learn controllable locomotion, in which the robot can change its behavior given external control signals (e.g. changing running speed with a remote). One way to achieve it is to train separate controllers for corresponding behaviors and switch them online according to the user-specified signals [17]. However, abruptly switching controllers can cause jerky motion and loss of balance. An alternative is to learn a generic controller and add the external control signals to the observation space [18]. As only one controller is learned and there is no need for transitions, this formulation significantly decreases the difficulty of the task. We choose the second approach and show that with PMTG, we can learn controllable locomotion efficiently.

3 Architecture

Our basic architecture is shown in Fig. 1 and consists of three main blocks: an existing/predefined controller, a learned policy, and the system to control (a robot). In this manuscript, we refer to the existing controller as the Trajectory Generator (TG), because we limit our experimental section to periodic motions. However, PMTG can be extended to various types of predefined controllers (e.g. a kinematic controller) in a straightforward manner.

Just like the robotic system to control, the TG is a black box from the policy's point of view. It receives a number of parameters as inputs from the policy and it outputs its state and actions at every time step. Hence, learning a policy in PMTG is equivalent to learning to control the original dynamical system (robot) extended by the TG. One simply concatenates the action space of the original problem and the controllable parameters of the TG. Similarly, the observation space is extended with the state variables of the TG. Note, that the state of the TG does not affect the reward.

The policy can optionally accept control inputs to allow external control of the robot. These control inputs are also appended to the robot's observations/state and fed into the policy. This simple formulation allows PMTG to be trained using a large selection of policy optimization methods.

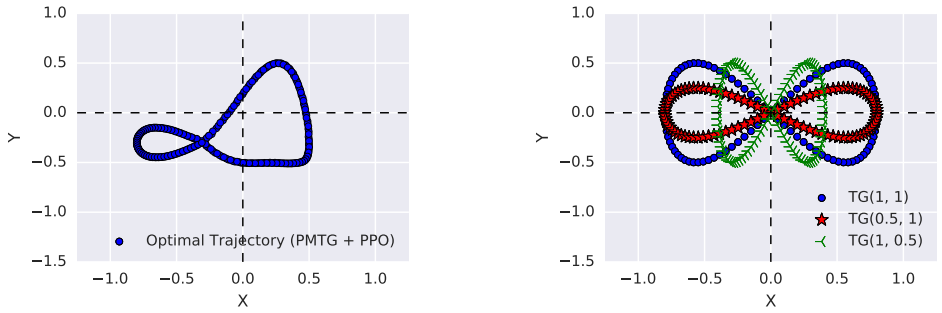
The outputs of the controller are the actions $\boldsymbol{u}$ that control the robot's actuators. These actions are computed as the sum of the output of the TG and the policy[1]:

$$\boldsymbol{u} = \boldsymbol{u}_{tg} + \boldsymbol{u}_{fb}.$$

One interpretation of this equation is that the TG generates possibly sub-optimal actions based on parameters learned by the policy. To improve upon these sub-optimal actions, the policy learns correction terms that are added to the output or learns to suppress u_{tg} if needed.

A different, yet important, interpretation is that the policy optimization algorithm can use the TG as a memory unit. Because we do not place restrictions on the type of TG, it makes sense to think about very simple choices of TGs that still provide the policy with useful memory. For example, imagine choosing a leaky integrate-and-fire neuron [19] with a constant input as the TG and letting the policy control the integration leak rate. In this case, the policy could use the TG as a controllable clock signal. Because of this last interpretation, we only consider feed-forward neural networks for the policy in this work. All the memory is to be provided by the TG. As we will demonstrate using both the synthetic control problem and robot locomotion, these benefits of the TG allow us to efficiently learn architecturally simple policies (e.g. linear) that still generate complex and robust behavior.

We now introduce a synthetic control problem to illustrate how PMTG works. We consider a 2D environment of $2\,\mathrm{m}$ by $2\,\mathrm{m}$ in which a point is to be moved along a desired cyclic trajectory to maximize the returned reward (Fig. 2a. The input to the environment (action space) is the desired next position $\boldsymbol{u} = [u_x\, u_y]^T$.



(a) Optimal (learned) behavior.

(b) TG with 3 different sets of parameters.

Figure 2: 2D Synthetic Control problem with a desired pattern and the TG that generates figure-eights.

[1]We use the subscript $_{fb}$ to refer to the policy because it computes feedback signals.

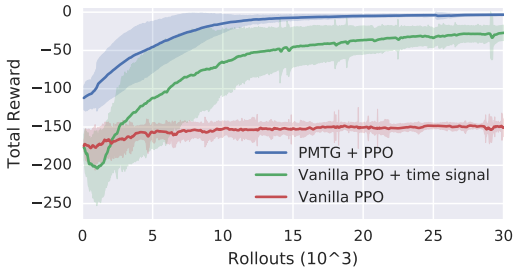


Figure 3: Learning curves for the 2D Synthetic Control Problem using PMTG + PPO and Vanilla PPO. For Vanilla PPO, we simply remove the TG from the setup ($\boldsymbol{u} = \boldsymbol{u}_{fb}$, $\boldsymbol{s} = [x, y]^T$.)

As prior knowledge, under the assumption that we know that the trajectory will be a highly deformed and displaced version of figure-eight, we pick an eight curve as the trajectory generator and allow the policy to change the amplitudes along the x and y axes[2] (Fig. 2b):

$$\boldsymbol{u}_{tg}(a_x, a_y) = \begin{bmatrix} a_x \sin(2\pi t) \\ \frac{a_y}{2} \sin(2\pi t) \cos(2\pi t) \end{bmatrix},$$

where t represents the current timestep and is stored by TG as its internal state.

For the policy, the observations are the current position along x and y coordinates and the state of the TG (the current time step). The actions are the desired position $\boldsymbol{u}_{fb}$ and the parameters of the TG a_x, a_y (amplitudes used for the figure-eight). The reward is the negative Euclidian distance to a deformed figure-eight. We used the Proximal Policy Optimization (PPO) algorithm for learning with a fully connected neural networks with two layers and ReLU non-linearities [6] [3].

Using the architecture, PMTG + PPO reaches almost optimal behavior with a reward close to zero (Fig. 3). For comparison, training a pure reactive controller using Vanilla PPO fails to produce any good result. The failure to learn by the reactive controller can be explained by the nature of the task, partially observable state space, and lack of memory to distinguish different phases of the target figure. Since the reactive controller lacks time-awareness (or external memory), we also tested Vanilla PPO with a time signal as an additional observation. This combination performed better than Vanilla PPO, but still worse than PMTG. In this example problem we showed a basic TG, its combination with a feed-forward policy, and how PMTG allows a feed-forward policy to learn a problem that is challenging for a pure reactive controller.

4 Quadruped Locomotion

4.1 Controller Design

Robot locomotion is a challenging problem for learning. Partial observations, noisy sensors combined with latency, and rich contacts all increase the difficulty of the task. Despite the challenges, the nature of locomotion makes it a good fit for PMTG. TG design can be based on the idea that legs follow a periodic motion with specific characteristics. A clear definition of the legs' trajectories is not needed. Instead, we can roughly define the family of trajectories using parameters such as stride length, leg clearance, walking height, and frequency. Fig. 4 shows a sample trajectory of the leg and parameters based on this idea. The detailed definition of a TG for locomotion can be found in Appendix.

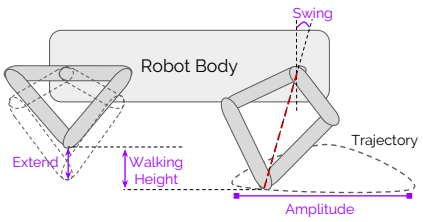


Figure 4: Illustration of robot leg trajectories generated by the TG.

[2] To limit the complexity of this example, we do not use an external control signal nor allow the policy to control the offset or frequency of the trajectory generator.

[3] With hyperparameter search for up to 200 neurons per layer.

The detailed architecture adapted to quadruped locomotion is shown in Fig. 5. At every timestep, the policy receives observations ($\boldsymbol{s}$), desired velocity (v, control input) and the phase (ϕ) of the trajectory generator. It computes 3 parameters for the TG (frequency f, amplitude a and walking height h) and 8 actions for the legs of the robot ($\boldsymbol{u}_{fb}$) that will directly be added to the TG's calculated leg positions ($\boldsymbol{u}_{tg}$). The sum of these actions is used as desired motor positions, which are tracked by Proportional-Derivative controllers. Since the policy dictates the frequency at each time step, it dictates the step-size that will be added to TG's phase. This eventually allows the policy to warp time and use the TG in a time-independent fashion.

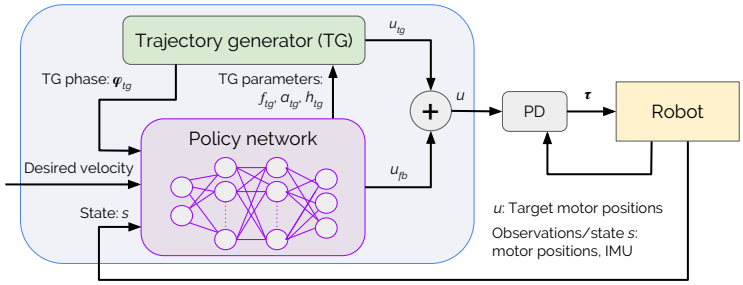


Figure 5: Adaptation of PMTG to the quadruped locomotion problem.

For locomotion, the design of TG can be as simple as Fig. 4 or can be composed of more complex open loop mechanisms. We use a stateful module that generates trajectories in an open loop fashion by using 3 parameters (walking height, amplitude, frequency). It is possible to use a TG that is pre-optimized for the given task, or hand-tuned to roughly generate a desired gait. For walking and running gaits, we used a TG that uses a gait shown in Fig. 6a and pre-optimized as a standalone open-loop controller. Despite the pre-optimization, the TG itself cannot provide stable forward locomotion since it lacks the feedback from the robot. In addition, for the bounding gait, we tested PMTG with a simpler and a hand-tuned TG that is not optimized. The only behavior provided by the TG is swinging front and back legs in half period phase difference (Fig. 6b).

4.2 Training

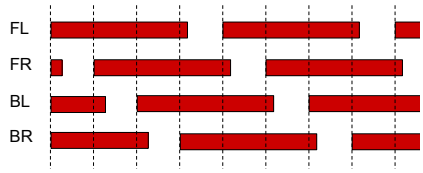
We train the locomotion policy using PyBullet [20] to simulate the Minitaur robot from Ghost Robotics. As the training algorithm we use both Evolutionary Strategies (specifically ARS [21]) and Reinforcement Learning (specifically PPO [6]). During training, we vary the desired forward velocity during each rollout. We start with $0\,\mathrm{m\,s^{-1}}$, gradually increase the desired speed to $0.4\,\mathrm{m\,s^{-1}}$, and keep it there for a while, then decrease it back to $0\,\mathrm{m\,s^{-1}}$ by the end of the rollout. The exact speed profile is shown in Fig. 8a. During each rollout, We add random directional perturbation forces (up to 60N vertical, 10N horizontal) to the robot multiple times to favor more stable behaviors. Each rollout ends either at $25\,\mathrm{s}$ or when the robot falls. The reward function is calculated based on the difference between the desired speed and robot's speed as:

$$R = v_{\max} e^{-\frac{(v_R - v_T)^2}{2{v_{\max}}^2}},$$

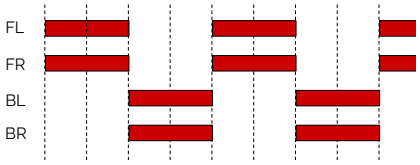
where $v_{\max}$ is the maximum desired velocity for the task, $v_R - v_T$ are the robot's actual velocity and the target velocity at the current timestep. We selected this reward function because it provides the maximum reward when the robot is within the range (20% of the top speed) of the desired speed and decreases to $0\,\mathrm{m\,s^{-1}}$ if the difference is higher.

The observation includes the robot's roll and pitch and angular velocities along these two axes received (IMU sensor reading, 4 dimensions total). Overall the policy uses 7 input dimensions: 4 observation dimensions, the desired velocity as the control input, and the phase of the TG represented by $sin(\phi), cos(\phi)$. The action space for the policy is 11 dimensional: 8 dimensions for the legs (swing and extension for each leg), and 3 parameters consumed by the TG (frequency, amplitude, walking height).

For policy complexity, we evaluated both a two-layer fully connected neural network (up to 200 neurons per layer) as well as a simple linear policy (77 parameters). We trained the policies using 3 separate tasks: slow walking (up to $0.4\,\mathrm{m\,s^{-1}}$), fast walking (up to $0.8\,\mathrm{m\,s^{-1}}$) and bounding (up to

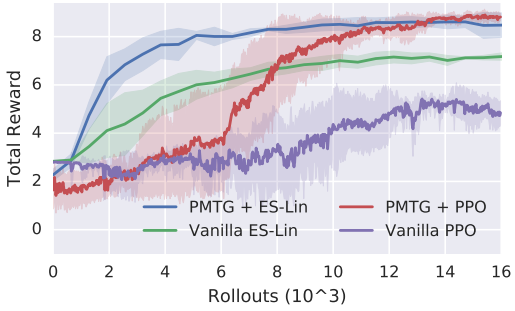


(a) Leg phases of TG for walking and running.

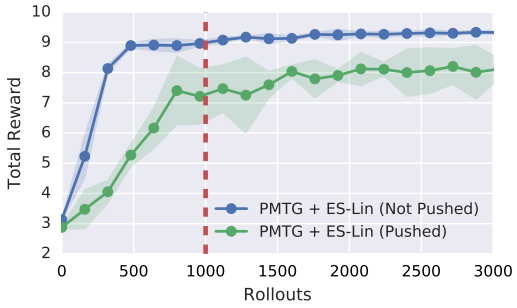


(b) Leg phases of TG for bounding.

Figure 6: Leg phases for 2 different gaits used for TG. For each leg, the red bar indicates the duration of the stance (compared to swing represented by empty areas). FL: Front Left, FR: Front Right, BL: Back Left, BR: Back Right.



(a) Learning with and without PMTG.



(b) Fastest learning configurations (rollouts < 1000).

Figure 7: Training curves for quadruped walking with speed tracking using PMTG. Left: Learning curves of PMTG vs. PPO and ES. Right: Examples of experiments that learned the desired gait in fewer than 1000 rollouts (ES with Linear Policy using 8 directions per iteration).

$0.4\,\mathrm{m\,s^{-1}}$). For the bounding gait, we use a different TG with phases shown in Fig. 6b. For walking gaits, the TG alone does not provide forward motion, but the robot does not immediately fall. The open loop (TG only) bounding gait fails immediately.

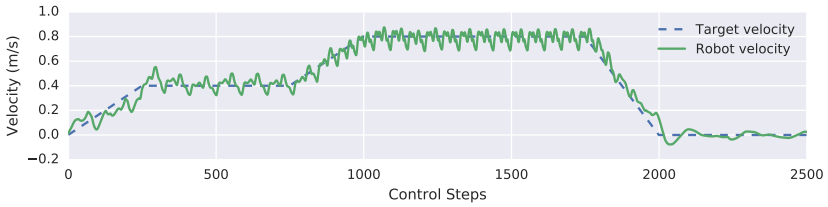
4.3 Results

Our architecture makes learning the complex locomotion task easier. When we use PMTG, both algorithms successfully learn controllable locomotion (Fig. 7a). Both the linear controller and the two-layer feed-forward neural network achieve the desired behavior. The curves for Vanilla ES-Lin and Vanilla PPO show the results for a reactive controller instead of PMTG (we simply remove the TG, $\boldsymbol{u} = \boldsymbol{u}_{fb}$). Without PMTG both algorithms fail to achieve the optimal reward levels. Lower rewards show that the controller learns the walking behavior but cannot fully keep up with the changing target speed [4].

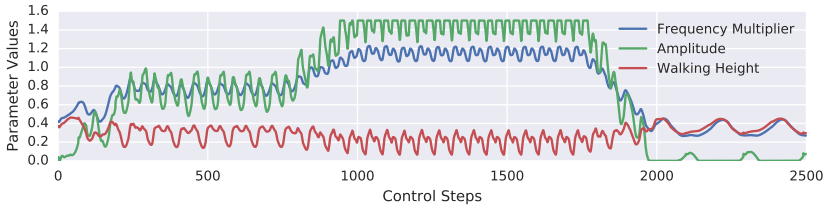
By combining PMTG with ARS and a linear policy, we achieved high data efficiency for learning locomotion. We also note that the linear policy has relatively few parameters (77). As an added benefit, we observed that learning with PMTG combined with a linear policy required fewer rollouts for the given locomotion task. Fig. 7b shows learning curves for the hyperparameters with the fastest learning speed (ES with 8 directions per iteration). We observe that it is possible to learn good policies with ES in fewer than 1000 rollouts. This is possible because we were able to embed prior knowledge into the TG and because the architecture reduces the complexity of the policy learning problem. The number of rollouts is low relative to the complexity of the locomotion task. This opens a research direction to using PMTG for on-robot learning, which we are planning as a future work.

Next, we look at the characteristics of a sample converged controller using PMTG and ES with a linear policy. We focus on running instead of walking because it shows considerable changes in

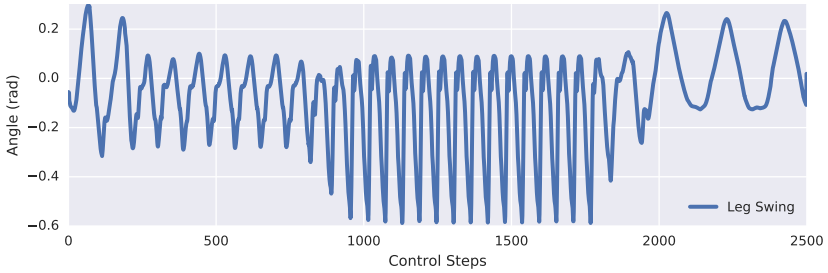
[4] The literature contains successful learning of reactive controllers on locomotion tasks with less complexity, richer state space and different reward functions [7].



(a) Speed profile.



(b) TG parameters selected by the policy.



(c) Swing angle of one of the legs.

Figure 8: Characteristics of a converged policy during running. The robot follows the change in desired speed while changing the parameters during the course. The plot of leg swing angle shows major changes in the emerged pattern at different speeds.

TG parameters and gait during a single rollout. Fig. 8a shows a single run after training: The robot does not have any problems tracking the desired speed. Fig. 8b shows that the policy significantly modulates both the amplitude (which commands stride length) and the frequency of the gait depending on the desired speed. These parameters affect the output coming from the TG, but they do not necessarily show the eventual leg movement since the policy can add corrections. Fig. 8c shows the swing angle of one the legs during the same rollout. The motion of the leg is periodic, but the shape of the signal changes significantly depending on the speed.

4.4 Robot Experiments

The reality gap between simulation and real environments is a major challenge for learning in robotics. In many scenarios, learning in simulation can easily converge to unrealistic behaviors or exploit the corner cases of a simulator. In PMTG, we provide a class of initial trajectories (TG) that the policy can build upon. The converged policies usually follow the characteristics of the TG, avoiding unrealistic behaviors. Additionally, we use randomization by applying random directional virtual forces to the robot during training to avoid overfitting to the simulation model.

We deployed a number of the learned controllers to the robot to see how our results transfer to the real world. We define success if the robot successfully moves forward at various speeds and does not fall during the rollout. A short summary of these results is shown in the supplementary video. For slower walking, all the policies successfully worked on the robot. The emerged behaviors are similar to the simulation. The robot slowly increases its speed and walks at different desired speeds and slows down to stop without any observable problems.

The policies trained for walking at faster speeds (up to $0.8\,\mathrm{m\,s^{-1}}$) mostly completed the rollouts successfully, but the legs were occasionally slipping at higher speeds. Although slippage affected the overall behavior for certain policies (i.e. distance run, direction) the robot recovered from falling and continued walking in most trials.

When we used the TG with a bounding gait, we observed different emerged gaits for different learning algorithms and hyperparameters. The policies trained with PPO were the most stable, jumping forward by modulating the parameters for walking height. The policy significantly overrode the gait using its correction ability. The resulting behavior shows forward movement of the robot using jumps at different speeds. We also include these different behaviors in the supplementary video.

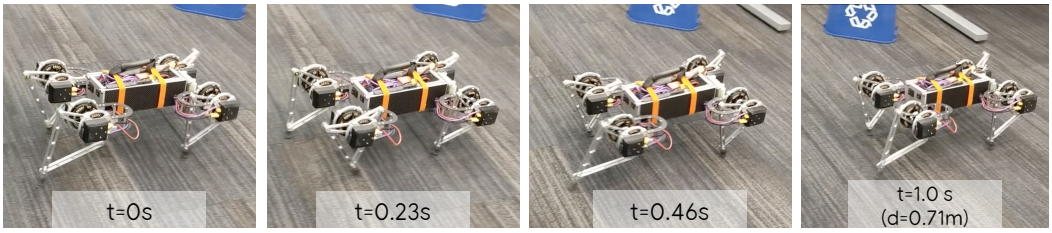


Figure 9: Minitaur robot walking using the learned controller.

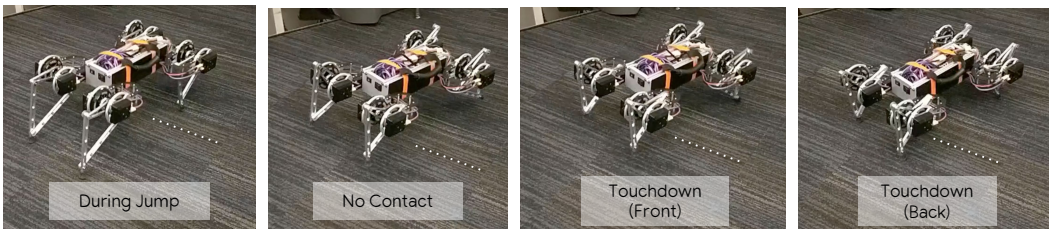


Figure 10: Minitaur robot trained with the TG for bounding.

5 Conclusion

We introduced a new control architecture (PMTG) that represents prior knowledge as a parameterized Trajectory Generator (TG) and combines it with a learned policy. Unique to our approach, the policy modulates the trajectory generator at every time step based on observations coming from the environment and TGs internal state. This allows the policy to generate many behaviors based on this prior knowledge, while also using TG as a memory unit. This combination enables simple reactive policies (e.g. linear) to learn complex behaviors.

To validate our technique, we used PMTG to tackle quadruped robot locomotion. We show the generality of PMTG by training the architecture using both ES and RL algorithms on two different gaits. We used relatively simple policies considering the complexity of the locomotion task, and had success with a linear policy. The policy uses only IMU readings – a low dimensional set of observations for a robot locomotion task – and takes the desired velocity as an external control input. We showed successful transfer of these policies from simulation to the Minitaur robot.

We plan to use our current approach as a starting point for this class of architectures with simple learned policies. In this work we relied on ad hoc trajectory generators that were chosen based on our intuition about a given problem. In future work, we are interested in getting a deeper understanding of which types of trajectory generators work best for a specific domain, possibly extracting trajectory generators from demonstrations. Finally, we are interested in theoretical foundations for PMTG and how it relates to existing models, specifically recurrent ones.

Acknowledgments

The authors would like to thank the members of the Google Brain Robotics group for their support and help.

References

[1] K. Choromanski, A. Iscen, V. Sindhwani, J. Tan, and E. Coumans. Optimizing simulations with noise-tolerant structured exploration. In *Robotics and Automation (ICRA), 2018 IEEE International Conference on*. IEEE, 2018.

[2] R. Calandra, A. Seyfarth, J. Peters, and M. P. Deisenroth. Bayesian optimization for learning gaits under uncertainty. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):5–23, 2016.

[3] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[4] H. Mania, A. Guy, and B. Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

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

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

[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. In *Robotics: Science and Systems*, 2018.

[8] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[9] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, 2017.

[10] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2016.

[11] D. Holden, T. Komura, and J. Saito. Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG)*, 36(4):42, 2017.

[12] H. Zhang, W. Starke, T. Komura, and J. Saito. Mode-adaptive neural networks for quadruped motion control. *ACM Transactions on Graphics*, 37(4), 3 2018. ISSN 0730-0301.

[13] A. Iscen, A. Agogino, V. SunSpiral, and K. Tumer. Controlling tensegrity robots through evolution. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1293–1300. ACM, 2013.

[14] A. J. Ijspeert. Central pattern generators for locomotion control in animals and robots: a review. *Neural networks*, 21(4):642–653, 2008.

[15] S. Gay, J. Santos-Victor, and A. Ijspeert. Learning robot gait stability using neural networks as sensory feedback function for central pattern generators. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 194–201. Ieee, 2013.

[16] A. Sharma and K. M. Kitani. Phase-parametric policies for reinforcement learning in cyclic environments. In *AAAI Conference on Artificial Intelligence*, Pittsburgh, PA, 2018.

[17] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *arXiv preprint arXiv:1804.02717*, 2018.

[18] J. Tan, Y. Gu, C. K. Liu, and G. Turk. Learning bicycle stunts. *ACM Transactions on Graphics (TOG)*, 33(4):50, 2014.

[19] W. Gerstner and W. M. Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge University Press, 2002.

[20] E. Coumans and Y. Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2018.

[21] H. Mania, A. Guy, and B. Recht. Simple random search provides a competitive approach to reinforcement learning. *CoRR*, abs/1803.07055, 2018. URL http://arxiv.org/abs/1803.07055.

Appendix

The phase of the trajectory generator (between 0 and 2π) is defined as:

$$\phi_t = \phi_{t-1} + 2\pi f_{tg} \Delta t \bmod 2\pi, \tag{1}$$

where f_{tg} defines the frequency of the trajectory generator. In PMTG architecture, f_{tg} is selected by the policy at each time step as an action.

In this work, we use the following trajectory generator for the legs:

$$\boldsymbol{u}_{tg} = \begin{bmatrix} S(t) \\ E(t) \end{bmatrix} = \begin{bmatrix} C_s + \alpha_{tg} cos(t') \\ h_{tg} + A_e sin(t') + \theta cos(t') \end{bmatrix}. \tag{2}$$

- $S(t)$, and $E(t)$ are respectively the swing and extension of the leg as shown in Fig. 4.
- C_s defines the center for the swing DOF and extension DOF (in radians).
- h_{tg} defines the center for the extension DOF. Extension is represented in terms of rotation of the two motors in the opposite direction, hence the unit is also radians. Since all legs share the same h_{tg}, it corresponds to the walking height of the robot.
- α_{tg} defines the amplitude of the swing signal (in radians). This corresponds to the size of a stride during locomotion.
- A_e defines the amplitude of the extension during swing phase. This corresponds to the ground clearance of the feet during the swing phase.
- θ defines the extension difference between when the leg is at the end of the swing and when the leg is at end of the stance. This is mostly useful for climbing up or down.

We compute t' based on the swing and stance phases:

$$t' = \begin{cases} \frac{\phi_{\text{leg}}}{2(1-\beta)} & \text{if } 0 < \phi_{\text{leg}} < 2\pi\beta; \\ 2\pi - \frac{(2\pi - \phi_{\text{leg}})}{2\beta} & \text{otherwise,} \end{cases} \tag{3}$$

where β defines the proportion of the duration of the swing phase to the stance phase.

For each leg, the phase is calculated separately as

$$\phi_{\text{leg}} = \phi_t + \Delta\phi_{\text{leg}} \bmod 2\pi, \tag{4}$$

where $\Delta\phi_{\text{leg}}$ represents the phase difference of this leg compared to the first (left front) leg. This is defined by the selected gait (i.e. walking vs bounding).