Developing Locomotion Skills with Deep Reinforcement Learning

by

Xue Bin Peng

B.Sc., The University of British Columbia, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2017

Abstract

While physics-based models for passive phenomena such as cloth and fluids have been widely adopted in computer animation, physics-based character simulation remains a challenging problem. One of the major hurdles for character simulation is that of control, the modeling of a character's behaviour in response to its goals and environment. This challenge is further compounded by the high-dimensional and complex dynamics that often arise from these systems. A popular approach to mitigating these challenges is to build reduced models that capture important properties for a particular task. These models often leverage significant human insight, and may nonetheless overlook important information. In this thesis, we explore the application of deep reinforcement learning (DeepRL) to develop control policies that operate directly using high-dimensional low-level representations, thereby reducing the need for manual feature engineering and enabling characters to perform more challenging tasks in complex environments.

We start by presenting a DeepRL framework for developing policies that allow character to agilely traverse across irregular terrain. The policies are represented using a mixture of experts model, which selects from a small collection of parameterized controllers. Our method is demonstrated on planar characters of varying morphologies and different classes of terrain. Through the learning process, the networks develop the appropriate strategies for traveling across various irregular environments without requiring extensive feature engineering. Next, we explore the effects of different action parameterizations on the performance of RL policies. We compare policies trained using low-level actions, such as torques, target velocities, target angles, and muscle activations. Performance is evaluated using a motion imitation benchmark. For our particular task, the choice of higher-level actions that incorporate local feedback, such as target angles, leads to significant improvements in performance and learning speed. Finally, we describe a hierarchical reinforcement learning framework for controlling the motion of a simulated 3D biped. By training each level of the hierarchy to operate at different spatial and temporal scales, the character is able to perform a variety of locomotion tasks that require a balance between short-term and long-term planning. Some of the tasks include soccer dribbling, path following, and navigation across dynamic obstacles.

Lay Abstract

Humans and other animals are able to agilely move through and interact with their environment using a rich repertoire of motor skills. Modeling these skills has been a long standing challenge with far-reaching implications for fields ranging from computer animation, robotics, and many more. Reinforcement learning has emerged as a promising paradigm for developing these skills, where an agent learns through trial-and-error in order to discover the appropriate behaviours for accomplishing a task. The goal of this work is to leverage deep reinforcement learning techniques to develop locomotion skills that enable simulated agents to move agilely through their surroundings in a task-driven manner.

Preface

Chapter 4 was published as Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2016)*, 35(4), 2016. Chapter 5 is currently under review, and Chapter 6 will be published as Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017. My supervisor, Michiel van de Panne was instrumental in the paper-writing, and in providing the vision and direction for the projects. My co-authors Glen and KangKang also made significant contributions to the paper-writing. This work would not have been possible if not for their efforts and our countless hours of discussions. I was responsible for designing and implementing the learning frameworks, tuning the algorithms, performing experiments, and writing the papers. Glen also contributed to the coding and collection of results.

Table of Contents

List of Tables

List of Figures

Glossary

S state space

s state

s' next state

A action space

a action

r reward

γ discount factor

π policy

V state-value function

Q action-value function

$\mathbb{A}$ advantage function

Acknowledgments

I would like to thank my supervisor, Dr. Michiel van de Panne, for his mentoring and support throughout my time at UBC. It has been a pleasure working with you, and I cannot thank you enough for all that you have taught me over these past years. I will always recall fondly of our many conversations and the atmosphere that you have fostered in the group. Your vision and poise have been, and will continue to be, a great source of inspiration for my aspirations both in work and life.

I would like to extend my thanks to Dr. Dinesh Pai for being part of my examining committee and for his feedback on this thesis. Also, a great debt of gratitude to my co-authors Glen Berseth and Dr. KangKang Yin. All of this work would not be what it is had it not been for your efforts. Furthermore, I want to thank my friends and colleagues, it has been pleasure to meet and learn alongside all of you.

Finally, and most importantly, I want to thank my family for their support throughout my life, and for the opportunities that they have provided me, at times at the cost of their own. Though it can often be taken for granted and left overlooked, your perseverance and tireless efforts have been an enduring source of inspiration. At the risk of being banal, none of this would have been possible without you.

Chapter 1

Introduction

Be it the leisurely stride of a walk or the acrobatic stunts of parkour, humans and other animals are able to agilely move through and interact with their environments using a rich repertoire of motor skills. Modeling these skills have been a long standing challenge with far-reaching applications for fields ranging from biomechanics, robotics, computer animation, and many more. While significant efforts have been devoted to building models of motions, such as walking and running, the generalization of such models to a more diverse array of skills can be limited. The manual engineering of models is further complicated as the domain shifts towards more dynamic motions where human insight becomes increasingly scarce.

In this work, we adopt a learning-based approach that aims to develop skillful agents while reducing the dependency on human domain knowledge. At the core of our work is the reinforcement learning (RL) paradigm, whereby an agent develops skillful behaviours through trial-and-error. While our work is presented in the context of physics-based character simulation for computer animation, we believe that the underlying insights gained may be of interest to other domains where motion control is of prominent interest.

1.1 Motivation

In computer animation, developing physics-based models of character motion offers a wealth of potential applications. The most immediate may be in alleviating the tedious manual efforts currently required by artists to author motions for characters. By simulating the physics and low-level details of a motion, artistic effort can be better directed towards the high-level and narrative-relevant content of a production. In the space of interactive media, such as games, character simulation offers the possibility of behaviour synthesis, in which appropriate reactions are generated for characters in response to different scenarios, where exhaustive manual scripting becomes infeasible. With the advent of virtual reality, the interactions a user can have with virtual agents are richer than would be possible through

more tradition interfaces, such as keyboards and game controllers. As the possible forms of interactions grow, the capacity of developers to generate or collect natural responsive behaviours for the various forms of interaction becomes increasingly strained. Therefore the role of simulation will likely become more prominent as new technologies emerge that enable ever more immersive virtual experiences. Reinforcement learning may also offer the potential for adaptive agents that can develop cooperative or adversarial strategies in accordance to a user's behaviour.

For robotics, learning models for legged-locomotion skills will prove valuable as machines become more prevalent in urban environments, commonly designed for human traversal. As agent are presented with unstructured environments that can be difficult to parameterize a priori, building systems that can process rich high-dimensional sensory information, while also adapting its behaviours to unfamiliar situations, will be vital. Better models of human and animal motions are also of interest for biomechanics and physiotherapy, with applications such as injury prevention and rehabilitation. Such model can also assist in the design and personalization of prosthetics that can help patients better recover their natural range of motion. Models of human motion will also benefit the development of exoskeletons, where systems need to anticipate and assist users in performing their intended tasks.

1.2 Thesis Overview

The work presented in this thesis explores the use of reinforcement learning (RL) to develop control policies for motion control tasks, with an emphasis on locomotion. We begin with an overview of related work in character animation and reinforcement learning (Chapter 2). Discussions of relevant work are also available within each chapter. Chapter 3 provides a review of fundamental concepts and algorithms in reinforcement learning, which form the foundations for the methods explored in the following chapters. The framework presented for Terrain-Adaptive Locomotion (Chapter 4) develops control policies that enable simulated 2D character to traverse across irregular obstacle-filled terrains. The use of a deep convolutional neural network to represent the policy provides a means of processing high-dimensional low-level descriptions of the environment, giving rise to policies that can adapt to a variety of obstacles without requiring additional manual feature engineering. Next, the work on Action Parameterizations (Chapter 5) explores the effects of different choices of action parameterizations on the performance of RL policies. We show that simple low-level action abstractions can offer significant improvements to performance and learning speed. Finally, in Hierarchical Locomotion Skills (Chapter 6), we present a hierarchical RL framework that enables a 3D simulated biped to perform a variety of locomotion tasks, such as soccer dribbling, path following, and obstacle avoidance. The hierarchical control policies eliminate much of the hand-engineered control structures (e.g. finite-state machines and feedback strategies) often leveraged by previous systems, and replaces them with learned neural network controllers that operate at different spatial and temporal abstractions. This hierarchical decompositions allows the policies to fulfill high-level task objectives while also satisfying low-level goals. We conclude with a discussion on the limitations of our methods and suggest possible directions for future work.

1.3 Terrain-Adaptive Locomotion

In this work, we consider the task of traversing across irregular terrain with randomly generated sequences of obstacles (Chapter 4). The goal of a policy is to select the appropriate actions for a character such that it is able to continue moving through the environment while maintaining a desired forward velocity. The environment consists of randomly varying irregular slopes interrupted by step, gap, and wall obstacles. One of the challenges of this task lies in representing the shape of the terrain. Due to the irregularity of the environment, it can be challenging to manually design a compact set of features that can sufficiently parameterize the terrain variations encountered by the characters. With deep reinforcement learning, we are able to directly provide a neural network policy with high-dimensional height-field representations of the terrain, and the network in turn learns to extract relevant features for traversing across the obstacles.

Our method is characterized by high-dimensional low-level state representations, parameterized finite-state machines (FSM) that provide the policies with high-level action abstractions, and a mixture of actor-critic experts (MACE) policy model. The MACE model is composed of a collection of subpolicies, referred to as actors, and their corresponding set of value functions, referred to as critics. At the start of a cycle of the FSM, each actor proposes a different action in response to the state of the character and the up-comping terrain. The critics then predict the expected performance of each actor, and the action from the actor with the highest predicted performance is selected as the action for the cycle. In comparison to the more common choice of a unimodal Gaussian distribution, the mixture model allows for richer multi-modal action distributions.

Our framework is applied to train policies for planar dog, raptor, and goat characters. The policies enable the characters to agilely traverse across different classes of terrain without requiring terrain-specific or character-specific feature engineering. We evaluate the impact of different design decisions and show that the mixture model leads to significant improvements in performance compared to more conventional RL models.

1.4 Action Parameterization

While the previous work leveraged hand-crafted FSMs to provide high-level action abstractions, in this work we look to further reduce the domain knowledge involved in crafting such action parameterizations (Chapter 5). Instead of high-level action abstractions, we explore a number of low-level action representations, such as torques and target angles, and evaluate the impact of different choices of action spaces on policy performance. In alignment with recent trends in machine learning of moving towards high-dimensional low-level feature representations, DeepRL for motion control have often adopted torques as the action of choice. These policies are trained to directly map state observations to torques for each of the character's joints. However, the motion quality from these systems often fall short of what has been previously achieved with hand-crafted action abstractions, such as FSMs. Directly using torques

as the action parameterization also overlooks the synergy between control and biomechanics exhibited in nature. Passive-dynamics, from muscles and tendons, play an important role in providing low-level feedback that shapes the motions produced by humans and other animals.

In this work, we compare the use of torques, target velocities, target angles, and muscle activations as action parameterizations for a policy. Our benchmark consists of a motion-imitation task for various simulated 2D characters. Policies are trained to control each character to imitate a given reference motion, specified by a sequence of kinematic keyframes. Policies using different action parameterizations are then evaluated according to a number of metrics, such as final performance and learning speed.

Our results suggest that the choice of action parameterization can have significant impact on performance. For our task, high-level action abstractions that incorporate low-level feedback, such as target angles, can lead to faster learning and better overall performance. The differences between the various actions grow as the morphologies of the character becomes more complex. With the performance of low-level actions, such as torques, deteriorating noticeably as character complexity increases. Additionally, we propose an actuator optimization method that interleaves policy learning and actuator optimization. This method enables neural networks to directly control complex muscle models through muscle activations, without requiring additional control abstractions such as Jacobian transpose control or other inverse-dynamics methods.

1.5 Hierarchical Locomotion Skills

Finally, we present a use of hierarchical deep reinforcement learning to develop locomotion skills for a 3D biped (Chapter 6). The method developed for Terrain-Adaptive Locomotion (Chapter 4) trains policies that leverage hand-crafted FSMs to provide high-level action abstractions, enabling the policies to operate at the timescale of running steps (e.g. 2 Hz). The work in Action Parameterization (Chapter 5) developed control policies that utilize low-level action parameterizations, such as torques and target angles specified for every joint, resulting in policies that operate over finer timescales (e.g. 60 Hz). In this chapter, we look to develop hierarchical policies, where each level of the hierarchy operates at different spatial and temporal scales, thereby allowing the policies to simultaneously address high-level task objectives while also satisfying low-level goals.

Our control hierarchy consists of a high-level controller and a low-level controller, operating at 2 Hz and 30 Hz respectively. At the start of each walking step, the high-level controller specifies footstep goals for the low-level controller for the upcoming step. The low-level controller then specifies target angles for PD-controllers positioned at each of the character's joints in order to realize the footstep goals. We demonstrate the capabilities of this hierarchical control structure on different locomotion tasks such as soccer dribbling, path following, and navigation through an environment consisting of static or dynamic obstacles.

We show that the hierarchical decomposition allows the character to learn tasks that would otherwise

be challenging for a single-timescale policy. The resulting policies are also robust to significant external perturbations, rivaling controllers build using hand-crafted feedback strategies. Furthermore, we demonstrate transfer learning between different combinations of high-level and low-level controllers for the various tasks, yielding significant reductions in training time compared to retraining from random initialization.

Chapter 2

Related Work

Modeling movement skills, locomotion in particular, has a long history in computer animation, robotics, and biomechanics. It has also recently seen significant interest from the machine learning community as an interesting and challenging domain for reinforcement learning. In this chapter we focus on the most closely related work in computer animation and reinforcement learning.

2.1 Physics-based Character Animation

Significant progress has been made in recent years in developing methods to create motion from first principles, i.e., control and physics, as a means of character animation. A recent survey on physics-based character animation and control techniques [Geijtenbeek and Pronost, 2012] provides a comprehensive overview of work in this area. These methods can be coarsely categorized as model-free and model-based. Model-free methods assume no access to the equations of motion and rely on domain knowledge to develop simplified models that can be used in the design of controllers. An early and enduring approach to controller design has been to structure control policies around finite state machines (FSMs) and feedback rules that use a simplified abstract model or feedback law. These general ideas have been applied to human athletics, running, and a rich variety of walking styles [Hodgins et al., 1995, Laszlo et al., 1996, Yin et al., 2007, Sok et al., 2007, Coros et al., 2010, Lee et al., 2010a]. Model-based methods that assumes access to the equations of motion also provide highly effective solutions. A dynamics model is often utilized by inverse-dynamics based methods, with the short and long-term goals being encoded into the objectives of a quadratic program that is solved at every time-step, e.g., [da Silva et al., 2008, Muico et al., 2009, Mordatch et al., 2010, Ye and Liu, 2010]. Many model-free and model-based methods use some form of model-free policy search, wherein a controller is first designed and then has a number of its free parameters optimized using episodic evaluations. Policy search methods, e.g., stochastic local search or CMA [Hansen, 2006], can be used to optimize the parameters of the given control structures to achieve a richer variety of motions e.g., [Yin et al., 2008, Wang et al., 2009, Coros et al., 2011b, Liu et al., 2012, Tan et al., 2014b], and efficient muscle-driven locomotion [Wang

et al., 2009, Lee et al., 2014]. Policy search has also been successfully applied directly to time-indexed splines and neural networks in order to learn a variety of bicycle stunts [Tan et al., 2014b].

An alternative class of approach is given by trajectory optimization methods, which can compute solutions offline, e.g., [Al Borno et al., 2013], and then later adapted for online model-predictive control [Tassa et al., 2012, Hämäläinen et al., 2015]. Alternative, optimized actions can be computed for the current time-step using quadratic programming, e.g., [Macchietto et al., 2009, de Lasa et al., 2010]. To further improve motion quality and enrich the motion repertoire, data-driven models incorporate motion capture examples in constructing controllers, most often using a learned or model-based trajectory tracking method [Sok et al., 2007, da Silva et al., 2008, Muico et al., 2009, Liu et al., 2012, 2016b].

2.2 Reinforcement Learning for Motion Control

Reinforcement learning as guided by state-value or action-value functions have been used for synthesis of kinematic motions. In particular, this type of RL has been highly successful for making decisions about which motion clip to play next in order to achieve a given objective [Lee and Lee, 2006, Treuille et al., 2007, Lee et al., 2009, 2010b, Levine et al., 2012]. Work that applies RL with nonparametric function approximators to the difficult problem of controlling the movement of physics-based characters has been more limited [Coros et al., 2009, Peng et al., 2015]. Due to the curse of dimensionality, nonparametric methods have often relied on the manual selection of a compact set of important features that succintly describes the state of the character and its environment. The use of parametric function approximators, most notably neural networks, have demonstrated impressive capabilities for processing high-dimensional low-level input features. This advantage has spurred significant efforts towards tackling reinforcement learning problems for physics-based characters with high-dimensional continuous state and action spaces. In the following section, we review some of the approaches used to develop deep neural network control policies.

2.3 Deep Reinforcement Learning

The recent successes of deep learning have also seen a resurgence of its use for learning control policies. The use of deep neural networks as function approximators for RL is generally referred to as deep reinforcement learning (DeepRL). These methods can be broadly characterized into three categories, direct policy approximation, deep Q-learning, and policy gradient methods, which we now elaborate on.

2.3.1 Direct Policy Approximation

Deep neural networks (DNNs) can be used to directly approximate a control policy from example data points generated by an oracle, represented by some other control process. For example, trajectory op-

timization can be used to compute families of optimal control solutions from various initial states, and each trajectory then yields a large number of data points suitable for supervised learning. A naive application of these ideas will often fail because when the approximated policy is deployed, the system state will nevertheless easily drift into regions of state space for which no data has been collected. This problem is rooted in the fact that a control policy and the state-distribution that it encounters are tightly coupled. Recent methods have made progress on this issue by proposing iterative techniques. Optimal trajectories are generated in close proximity to the trajectories resulting from the current policy; the current policy is then updated with the new data, and the iteration repeats. These *guided policy search* methods have been applied to produce motions for robust bipedal walking and swimming gaits for planar models [Levine and Koltun, 2014, Levine and Abbeel, 2014] and a growing number of challenging robotics applications. Relatedly, methods that leverage contact-invariant trajectory optimization have also demonstrated many capabilities, including planar swimmers, planar walkers, and 3D arm reaching [Mordatch and Todorov, 2014], and, more recently, simulated 3D swimming and flying, as well as 3D bipeds and quadrupeds capable of skilled interactive stepping behaviors [Mordatch et al., 2015b]. In this most recent work, the simulation is carried out as a state optimization, which allows for large timesteps, albeit with the minor caveat that the final motion comes from an optimization step rather than directly from a forward dynamics simulation

2.3.2 Deep Q-Learning

For decision problems with discrete action spaces, a DNN can be used to approximate a set of value functions, one for each action, thereby learning a complete state-action value (Q) function. One of the distinguishing characteristics of Q-learnings is that a policy can be implicitly represented by the Q-function. When a policy is queried for a particular state, an action can be chosen by first enumerating over all possible actions using the Q-function, and then selecting the action with the maximum predicted value. A notable achievement of this approach has been the ability to learn to play a large suite of Atari games at a human-level of skill, using only raw screen images and the score as input [Mnih et al., 2015]. Many additional improvements have since been proposed, including prioritized experience replay [Schaul et al., 2015], double Q-learning [Van Hasselt et al., 2015], better exploration strategies [Stadie et al., 2015], and accelerated learning using distributed computation [Nair et al., 2015]. However, it is not obvious how to directly extend these methods to control problems with continuous action spaces. Since action selection involves optimizing the Q-function over all possible actions, this process quickly becomes intractable for continuous action spaces. Methods have been proposed that impose particular structures onto the Q-function, which allows the optimization over continuous action spaces to be solved efficiently [Gu et al., 2016b].

2.3.3 Policy Gradient Methods

In the case of continuous actions learned in the absence of an oracle, DNNs can be used to explicitly model a policy $\pi(s)$. Policy gradient methods is a popular class of techniques for training a parameterized policy [Sutton et al., 2001]. These methods perform gradient ascent on the overall task objective using Monte-Carlo estimates of the objective gradient with respect to the policy parameters. The REINFORCE algorithm is a classic example of a policy gradient method for stochastic policies [Williams, 1992]. More recently, deterministic policy gradient methods have been proposed where a Q-function, $Q(s,a)$, is learned alongside the policy, $\pi(s)$. This leads to a single composite network, $Q(s,\pi(s))$, that allows for the back-propagation of value-gradients back through to the control policy, and therefore provides a mechanism for policy improvement. Since the original method for using deterministic policy gradients [Silver et al., 2014b], several variations have been proposed with a growing portfolio of demonstrated capabilities. This includes a method for stochastic policies that can span a range of model-free and model-based methods [Heess et al., 2015], as demonstrated on examples that include a monoped, a planar biped walker, and an 8-link planar cheetah model. Recently, further improvements have been proposed to allow end-to-end learning of image-to-control-torque policies for a wide selection of physical systems [Lillicrap et al., 2015b]. Another recent work proposes the use of policy-gradient DNN-RL with parameterized controllers for simulated robot soccer [Hausknecht and Stone, 2015a]. Recent work using generalized advantage estimation together with TRPO [Schulman et al., 2016] demonstrates robust 3D locomotion for a biped with ball feet, although with (subjectively speaking) awkward movements. While the aforementioned methods are promising, the resulting capabilities and motion quality as applied to locomoting articulated figures still fall well short of what is needed for animation applications.

Chapter 3

Background

Our tasks will be structured as standard reinforcement problems where an agent interacts with its environment according to a policy in order to maximize a reward signal. Let $\pi(s) : S \to A$ represent a deterministic policy, which maps a state $s \in S$ to an action $a \in A$, while a stochastic policy $\pi(s,a) : S \times A \to \mathbb{R}$ represents the conditional probability distribution of a given s, $\pi(s,a) = p(a|s)$. At each control step t, the agent observes a state s_t and samples an action a_t from π. The environment in turn responds with a scalar reward r_t, and a new state $s'_t = s_{t+1}$ sampled from its dynamics $p(s'|s,a)$. The reward function $r_t = r(s_t, a_t)$ provides the agent with a feedback signal on the desirability of performing an action at a given state. In reinforcement learning, the objective is often to learn an optimal policy π^*, which maximizes the expected long term cumulative reward $J(\pi)$, expressed as the discounted sum of immediate rewards r_t,

$$\begin{aligned} J(\pi) &= \mathbb{E}_{r_0, r_1, \ldots r_T}\left[r_0 + \gamma r_1 + \ldots + \gamma^T r_T \middle| \pi\right] \\ &= \mathbb{E}_{\{r_t\}}\left[\sum_{t=0}^{T} \gamma^t r_t \middle| \pi\right] \end{aligned}$$

with $\gamma \in [0,1]$ as the discount factor, and T as the horizon, which may be infinite. The discount factor ensures that the cumulative reward is finite, and captures the intuition that events occurring in the distant future are likely to be of less consequence than those occurring in the more immediate future. The optimal policy is then defined as

$$\pi^* = \arg\max_{\pi} J(\pi)$$

3.1 Value Functions

Value functions provide estimates of a policy's expected performance as measured by the cumulative reward. This information is leveraged by value function based methods to improve a policy's performance [Sutton and Barto, 1998]. Given a policy, two common classes of value functions are state-value

functions $V(s)$ and action-value functions $Q(s,a)$. The state-value function $V(s)$ can be interpreted as the desirability of the agent being in a given state. This is formulate as the expected cumulative reward of following a policy starting at a given state,

$$V(s) = \mathbb{E}_{\{r_t\}}\left[\sum_{t=0}^{T} \gamma^t r_t \middle| s_0 = s, \pi\right]$$

Similarly, the action-value function $Q(s,a)$ can be interpreted as the desirability of performing a particular action at a given state. This is represented as the expected cumulative reward of performing action a at state s and following the policy starting at the next state s',

$$Q(s,a) = \mathbb{E}_{\{r_t\}}\left[\sum_{t=0}^{T} \gamma^t r_t \middle| s_0 = s, a_0 = a, \pi\right]$$

Whenever convenient and without ambiguity, the dependency on π will be implicitly assumed and dropped from the notation, $V(s)$ will be referred simply as the value function, and $Q(s,a)$ as the Q-function. The value functions can be defined recursively according to

$$V(s) = \mathbb{E}_{r,s'}\left[r + \gamma V(s') \middle| s_0 = s\right]$$

$$Q(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma \int_{a'} \pi(s',a') Q(s',a') da' \middle| s_0 = s, a_0 = a\right]$$

The relationship between the two value functions can be seen as follows

$$V(s) = \mathbb{E}_a\left[Q(s,a)\right]$$

$$Q(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma V(s') \middle| s_0 = s, a_0 = a\right]$$

These recursive definitions give rise to the value iteration algorithm for learning the value function through repeated rollouts of a policy. Starting from an initial guess of the value function $V^0(s)$, for each state s encountered by the agent, the policy π is queried to sample an action a, which then results in a new state s' and reward r. Each state transition can be summarized by an experience tuple $\tau = (s,a,r,s')$, and used to update the value function,

$$V^{k+1}(s) \leftarrow V^k(s) + \alpha\left[r + \gamma V^k(s') - V^k(s)\right]$$

Where $\alpha < 1$ is a stepsize, and $V^k(s)$ is the value function at the kth iteration. The update step $r + \gamma V(s') - V(s)$ is referred to as the temporal difference (TD), and can be interpreted as the difference between the predicted value at a given state $V(s)$ and an updated approximation of the actual value

observed by the agent $r + \gamma V(s')$. Algorithm 1 illustrates a method for learning a value function.

Algorithm 1 Policy Evaluation Using Value Iteration

1: V^0 initialize value function
2: **while** not done **do**
3: $s \leftarrow$ start state
4: $a \sim \pi(s,a)$
5: Apply a for one step
6: $s' \leftarrow$ end state
7: $r \leftarrow$ reward
8: $V^{k+1}(s) \leftarrow V^k(s) + \alpha\left[r + \gamma V^k(s') - V^k(s)\right]$
9: **end while**

The Q-function can be learned in a similar manner, but as it is a function of both state and action, in addition to recording the next state s', the next action a' selected by the policy at s' is also required. Each experience tuple therefore consists of $\tau = (s,a,r,s',a')$, and the update proceeds according to

$$Q^{k+1}(s) \leftarrow Q^k(s,a) + \alpha\left[r + \gamma Q^k(s',a') - Q^k(s,a)\right]$$

This update provides the foundation for the SARSA algorithm [Sutton and Barto, 1998].

3.2 Q-Learning

An advantage of learning a Q-function over a state-value function is that the Q-function provides an implicit representation of a policy. Suppose that π^* is the optimal policy and Q^* the optimal Q-function. An optimal deterministic policy can be defined as a policy that selects the action that maximizes the Q-value at a given state

$$\pi^*(s) = \arg\max_a \; Q^*(s,a)$$

where the optimal Q-function satisfies the property

$$Q^*(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma \max_{a'} \; Q^*(s',a')\right] \tag{3.1}$$

$$= \mathbb{E}_{r,s'}\left[r + \gamma \, Q^*(s',\pi^*(s'))\right]$$

More generally, given a Q-function that may or may not be optimal, a policy can be constructed by selecting the action that maximizes the Q-function at a given state. Of course, the resulting policy may no longer be optimal.

Q-learning takes advantage of the recursive definition of the optimal Q-function to learn an approximation of Q^* starting from an initial guess Q^0. The agent proceeds by collecting experiences

$\tau = (s, a, r, s')$ using the current Q-function to define a policy $\pi(s) = \arg\max_a Q(s,a)$. The experiences are then used to improve the Q-function by interpreting Equation 3.1 as an update step.

$$Q^{k+1}(s) \leftarrow Q^k(s,a) + \alpha \left[r + \gamma \max_{a'} Q^k(s',a') - Q^k(s,a) \right]$$

However, the method described so far is insufficient for convergence to the optimal Q-function. In particular, the definition of a determinisitc policy as always selecting the action which maximizes the Q-function does not allow the agent to explore new action that may yield higher rewards. Since the initial Q-function is likely a poor approximation of Q^*, the recommended action may not be optimal. This leads to the *exploration-exploitation tradeoff*, where an agent must balance between exploiting the current policy (i.e. selecting the actions proposed by a policy), and exploring new actions that may lead to improved performance. A simple heuristic to control this tradeoff is ε-greed exploration, using an exploration rate $\varepsilon \in [0,1]$. With ε-greed, the agent selects an action according to a policy with probability $(1-\varepsilon)$ and selects a random action with probaiblity ε. ε can therefore be interpreted as an agent's skepticism of its current policy, where $\varepsilon = 0$ results in a deterministic policy that always selects the proposed actions, and $\varepsilon = 1$ results in a stochastic policy that ignores the proposed actions. As such, a common heuristic for selecting ε is to start with a high value (e.g. $\varepsilon \approx 1$) and slowly decrease it as training progresses, and the policy improves (e.g. $\varepsilon \approx 0.2$). Algorithm 2 summarizes the overall learning process. With sufficient exploration and under suitable assumptions Q^k can be shown to converge to Q^*, yielding the optimal policy π^* [Sutton and Barto, 1998].

Algorithm 2 Q-Learning With ε-Greedy Exploration

Q^0 initialize value function
while not done **do**
  $s \leftarrow$ start state
  with probability ε **do**
    $a \leftarrow$ random action
  else
    $a \leftarrow \arg\max_a Q^k(s,a)$
  end with
  Apply a for one step
  $s' \leftarrow$ end state
  $r \leftarrow$ reward

  $Q^{k+1}(s,a) \leftarrow Q^k(s,a) + \alpha \left[r + \gamma \max_{a'} Q^k(s',a') - Q^k(s,a) \right]$
end while

3.3 Policy Gradient Methods

Q-learning has led to groundbreaking advances for a rich repertoire of control problems, with a notable recent example being Deep Q-Networks [Mnih et al., 2015], which combines deep neural networks with

Q-learning to achieve human-level performance on a suite of Atari games. However, many of the effective application of Q-learning has been for tasks with discrete action spaces, where the set of possible actions are reasonably small. This limitation partly stems from the optimization over the action space needed during policy evaluation and Q-function updates. This optimization quickly becomes intractable for large sets of possible actions, or if the action space becomes continuous. For motion control problems, the actions are often naturally parameterized by continuous action spaces, with parameters such as forces or target positions. This limitation of Q-learning motivates the use of a different class of methods that can better cope with continuous action spaces. Nonetheless, value functions continue to play an important role in the methods explored in the following sections.

One of the challenges of applying Q-learning to continuous action spaces is that the policy is implicitly represented by the Q-function. As such, during policy evaluation, when the agent queries the policy for an action, it requires optimizing the Q-function over the action space in order to find the action with the maximum predicted value. This can lead to intractable runtime costs when deploying a policy. An alternative approach is to learn an explicit representation of the policy that can directly map a query state to an action. A popular class of techniques for learning an explicit policy representation is policy gradient methods [Sutton et al., 2001]. Consider a policy modeled with as a parametric function $\pi(s,a|\theta)$ with parameters θ, then the expected cumulative reward can be re-expressed as $J(\theta)$, and the goal of learning π^* can be formulated as finding the optimal set of parameters θ^*

$$\theta^* = \arg\max_{\theta} J(\theta)$$

Policy gradient methods learn a policy by performing gradient ascent on the objective using empirical estimates of the policy gradient $\nabla_\theta J(\theta)$, i.e. the gradient of $J(\theta)$ with respect to the policy parameters θ. The policy can be determined according to the policy gradient theorem [Sutton et al., 2001], which provides a direction of improvement to adjust the policy parameters θ.

$$\nabla_\theta J(\theta) = \int_S d(s|\theta) \int_A \nabla_\theta \log(\pi(s,a|\theta)) \mathbb{A}(s,a) da\, ds$$

where $d(s|\theta) = \int_S \sum_{t=0}^{T} \gamma^t p_0(s_0) p(s_0 \to s|t,\theta) ds_0$ is the unnormalized discounted state distribution, with $p_0(s)$ representing the initial state distribution, and $p(s_0 \to s|t,\theta)$ modeling the likelihood of reaching state s by starting at s_0 and following the policy $\pi(s,a|\theta)$ for T steps [Silver et al., 2014a]. $\mathbb{A}(s,a)$ represents a generalized advantage function. The choice of advantage function gives rise to a family of policy gradient algorithms, but in this work, we will focus on the one-step temporal difference (TD) advantage function [Schulman et al., 2015]

$$\mathbb{A}(s,a) = r + \gamma V(s') - V(s)$$

where $V(s)$ is the state-value function of the policy $\pi(s,a|\theta)$. An action with a positive advantage implies better than average performance for a given state, and a negative advantage implies worse than

average performance. The policy gradient can therefore be interpreted as increasing the likelihood of actions that result in higher than average performance, while decreasing the likelihood of actions that result in lower than average performance. A parameterized policy $\pi(s,a|\theta_\pi)$ and parameterized value function $V(s|\theta_V)$, with parameters θ_π and θ_V, can be learned in tandem using an actor-critic framework [Konda and Tsitsiklis, 2000]. Algorithm 3 provides an example of an Actor-Critic algorithm using policy gradients. In an actor-critic algorithm, the policy is commonly referred as the actor, and the value function is referred to as the critic. α_π and α_V denote the actor and critic stepsizes.

Algorithm 3 Actor-Critic Algorithm Using Policy Gradients

θ_π initialize actor parameters
θ_V initialize critic parameters
while not done **do**
  $s \leftarrow$ start state
  $a \sim \pi(s,a|\theta_\pi)$
  Apply a for one step
  $s' \leftarrow$ end state
  $r \leftarrow$ reward

  $y \leftarrow r + \gamma V(s'|\theta_V)$
  $\theta_V \leftarrow \theta_V + \alpha_V \nabla_{\theta_V} V(s|\theta_V)(y - V(s|\theta_V))$
  $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \nabla_{\theta_\pi} \log(\pi(s,a|\theta_\pi))\mathbb{A}(s,a)$
end while

Next, we consider some design decisions for modeling the distribution from stochastic policies. Given a state s, a stochastic policy $\pi(s,a|\theta) = p(a|s,\theta)$ models a distribution over the action space. With discrete action spaces, a policy can often be represented as a discrete probability distribution over the possible actions, but predicting a score for each action. However, with continuous action spaces, this is no longer feasible and additional modeling decisions are often needed to represent the action distribution. A common choice is to model the action distribution as a unimodal Gaussian distribution with a parameterized mean $\mu(s|\theta)$ and fixed covariance matrix Σ.

$$\pi(s,a|\theta) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(a-\mu(s|\theta))^T \Sigma^{-1}(a-\mu(s|\theta))\right)$$

where $n = |A|$ is the dimension of the action space. Actions can be sampled from this distribution by applying Gaussian noise to the mean action

$$a = \mu(s|\theta) + N(0,\Sigma)$$

The corresponding policy gradient for a Gaussian policy assumes the form

$$\nabla_\theta J(\theta) = \int_S d(s|\theta) \int_A \nabla_\theta \mu(s|\theta) \Sigma^{-1} (a - \mu(s|\theta)) \mathbb{A}(s,a) da\, ds$$

which can be interpreted as shifting the mean of the action distribution towards actions that lead to higher

than expected rewards, while moving away from actions that lead to lower than expected rewards.

3.4 CACLA

The Continuous Actor-Critic Learning Automaton (CACLA) is a variant of policy gradient methods that attempts to address some of the challenges stemming from the choice of Gaussian policies [Van Hasselt, 2012]. To motivate CACLA, consider the use of the TD-advantage function with a Gaussian policy. A positive-TD update has the effect of shifting the mean of the action distribution towards an action that was observed to perform better than average, while a negative-TD update shifts the mean away from an action observed to perform worse than average. In the case of negative-TD updates, shifting the mean away from an observed action is equivalent to shifting it towards an unknown action, which may perform worse than the current mean action [Van Hasselt, 2012]. Though in expectation, the empirical estimates of the policy gradient converge to the true policy gradient, in practice, the gradients are estimated using minibatches of experience tuples, which only provide noisy estimates of the true gradient. Therefore, negative-TD updates can result in the policy adopting less desirable actions, which in turn influence the behaviour of the agent as it collects subsequent experiences. In practice, this can result in instabilities during training, where policy performance fluctuates drastically between iterations.

To mitigate some of these challenges, CACLA proposes the use of an alternative advantage function based on positive temporal differences

$$\mathbb{A}(s,a) = I[\delta > 0] = \begin{cases} 1, & \delta > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\delta = r + \gamma V(s') - V(s)$$

δ represents the temporal difference. When using a Gaussian policy, the CACLA update shifts the mean towards an action, only if it was observed to perform better than average, otherwise the policy remains unchanged. However, the resulting policy gradient is no longer a valid gradient of the original objective $J(\theta)$. Instead, CACLA can be interpreted as learning a policy that maximizes the likelihood of performing better than average. The choice of a step function has the additional advantage of being invariant to the scale of the reward function. As with most gradient descent algorithms, selecting the appropriate stepsize can significantly impact performance. With the standard TD-advantage function, scaling the reward function will often entail retuning the stepsize in order to compensate for the scaling. However with CACLA, the algorithm is invariant to uniform scaling of the reward function, which in practice reduces the tuning required for the stepsize. CACLA's positive temporal difference update is at the heart of the learning frameworks detailed in the following chapters.

Chapter 4

Terrain-Adaptive Locomotion

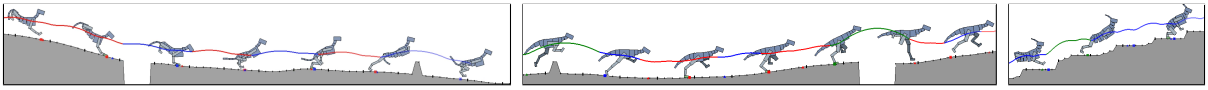


Figure 4.1: Terrain traversal using a learned actor-critic ensemble. The color-coding of the center-of-mass trajectory indicates the choice of actor used for each leap.

Reinforcement learning offers a promising methodology for developing skills for simulated characters, but typically requires working with sparse hand-crafted features. Building on recent progress in deep reinforcement learning, we introduce a mixture of actor-critic experts (MACE) approach that learns terrain-adaptive dynamic locomotion skills using high-dimensional state and terrain descriptions as input, and parameterized leaps or steps as output actions. MACE learns more quickly than a single actor-critic approach and results in actor-critic experts that exhibit specialization. Additional elements of our solution that contribute towards efficient learning include Boltzmann exploration and the introduction of initial actor biases to encourage specialization. Results are demonstrated for multiple planar characters and terrain classes.

4.1 Introduction

In practice, a number of challenges need to be overcome when applying the RL framework to problems with continuous and high-dimensional states and actions, as required by movement skills. A control policy needs to select the best actions for the distribution of states that will be encountered, but this distribution is often not known in advance. Similarly, the distribution of actions that will prove to be useful for these states is also seldom known in advance. Furthermore, the state-and-action distributions are not static in nature; as changes are made to the control policy, new states may be visited, and, conversely, the best possible policy may change as new actions are introduced. It is furthermore not obvious how to best represent the state of a character and its environment. Using large descriptors allows for very general and complete descriptions, but such high-dimensional descriptors define large

state spaces that pose a challenge for many RL methods. Using sparse descriptors makes the learning more manageable, but requires domain knowledge to design an informative-and-compact feature set that may nevertheless be missing important information.

In this chapter we use deep neural networks in combination with reinforcement learning to address the above challenges. This allows for the design of control policies that operate directly on high-dimensional character state descriptions (83D) and an environment state that consists of a height-field image of the upcoming terrain (200D). We provide a parameterized action space (29D) that allows the control policy to operate at the level of bounds, leaps, and steps. We introduce a novel *mixture of actor-critic experts (MACE) architecture* to enable accelerated learning. MACE develops *n* individual control policies and their associated value functions, which each then specialize in particular regimes of the overall motion. During final policy execution, the policy associated with the highest value function is executed, in a fashion analogous to Q-learning with discrete actions. We show the benefits of *Boltzmann exploration* and various algorithmic features for our problem domain. We demonstrate improvements in motion quality and terrain abilities over previous work.

4.2 Related Work

While signnificant progress has been made in developing physics-based controllers for a rich repertoire of locomotion skills, many methods focus mainly on controlling locomotion over flat terrain. A combination of manually-crafted and learned feedback strategies have been incoporated into controllers to improve robustness to unexpected perturbations and irregularities in the environment [Yin et al., 2007, Coros et al., 2010, 2011a, Geijtenbeek et al., 2013, Ding et al., 2015]. Nonetheless, these controllers are still purely reactive and cannot actively anticipate changes in the environment. Planning algorithms can be incorporated to guide characters through irregular terrains. But due to the complex dynamics and high-dimensional state space of articulated systems, these methods often perform planning using reduced state representations [Coros et al., 2008, Mordatch et al., 2010]. Reinforcement learning using nonparametric function approximators have been previously explored for terrain traversal [Peng et al., 2015]. But again relied on hand-crafted reduced state representations to describe the character and environment. As a result, the policies are limited to classes of terrain where compact descriptions are available.

In this work, we present a mixture model policy representation where subpolicies are trained to specialize in handling different situations. The idea of modular selection and identification for control has been proposed in many variations. Well-known work in sensorimotor control proposes the use of a responsibility predictor that divides experiences among several contexts [Haruno et al., 2001]. Similar concepts can be found in the use of skill libraries indexed based on sensory information [Pastor et al., 2012], Gaussian mixture models for multi-optima policy search for episodic tasks [Calinon et al., 2013], and the use of random forests for model-based control [Hester and Stone, 2013]. Ensemble methods for RL problems with discrete actions have been investigated in some detail [Wiering and Van Hasselt,

2008]. Adaptive mixtures of local experts [Jacobs et al., 1991] allow for specialization by allocating learning examples to a particular expert among an available set of experts according to a local gating function, which is also learned so as to maximize performance. This has also been shown to work well in the context of reinforcement learning [Doya et al., 2002, Uchibe and Doya, 2004]. More recently, there has been strong interest in developing deep RL architectures for multi-task learning, where the tasks are known in advance [Parisotto et al., 2015, Rusu et al., 2015], with a goal of achieving policy compression. In the context of physics-based character animation, a number of papers propose to use selections or combinations of controllers as the basis for developing more complex locomotion skills [Faloutsos et al., 2001, Coros et al., 2008, da Silva et al., 2009, Muico et al., 2011].

Our work: We propose a deep reinforcement learning method based on learning Q-functions and a policy, $\pi(s)$, for continuous action spaces as modeled on the CACLA RL algorithm [Van Hasselt and Wiering, 2007, Van Hasselt, 2012]. In particular, we show the effectiveness of using a *mixture of actor-critic experts (MACE)*, as constructed from multiple actor-critic pairs that each specialize in particular aspects of the motion. Unlike prior work on dynamic terrain traversal using reinforcement learning [Peng et al., 2015], our method can work directly with high-dimensional character and terrain state descriptions without requiring the feature engineering often needed by nonparametric methods. Our results also improve on the motion quality and expand upon the types of terrains that can be navigated with agility.

4.3 Overview

An overview of the system is shown in Figure 4.2, which illustrates three nested loops that each correspond to a different timescale. In the following description, we review its operation for our dog control policies, with other control policies being similar.

The inner-most loop models the low-level control and physics-based simulation process. At each time-step $\triangle t$, individual joint torques are computed by low-level control structures, such as PD-controllers and Jacobian transpose forces (see §4.4). These low-level control structures are organized into a small number of motion phases using a finite state machine. The motion of the character during the time step is then simulated by a physics engine.

The middle loop operates at the time scale of locomotion cycles, i.e., leaps for the dog. Touch-down of the hind-leg marks the beginning of a motion cycle, and at this moment the control policy, $a = \pi(s)$, chooses the action that will define the subsequent cycle. The state, s, is defined by C, a set of 83 numbers describing the character state, and T, a set of 200 numbers that provides a one-dimensional heightfield "image" of the upcoming terrain. The output action, a, assigns specific values to a set of 29 parameters of the FSM controller, which then governs the evolution of the motion during the next leap.

The control policy is defined by a small set of actor-critic pairs, whose outputs taken together represent the outputs of the learned deep network (see Figure 4.7). Each actor represents an individual control

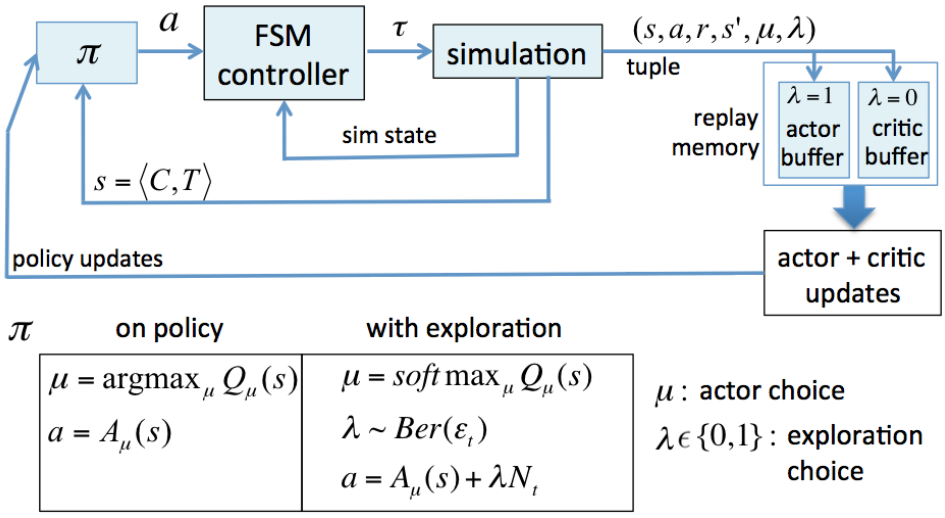


Figure 4.2: System Overview

policy; they each model their own actions, $A_\mu(s)$, as a function of the current state, s. The critics, $Q_\mu(s)$, each estimate the quality of the action of their corresponding actor in the given situation, as given by the Q-value that they produce. This is a scalar that defines the objective function, i.e., the expected value of the cumulative sum of (discounted) future rewards. The functions $A_\mu(s)$ and $Q_\mu(s)$ are modeled using a single deep neural network that has multiple corresponding outputs, with most network layers being shared. At run-time, the critics are queried at the start of each locomotion cycle in order to select the actor that is best suited for the current state, according to the highest estimated Q-value. The output action of the corresponding actor is then used to drive the current locomotion cycle.

Learning requires exploration of "off-policy" behaviors. This is implemented in two parts. First, an actor can be selected probabilistically, instead of deterministically choosing the max-Q actor. This is done using a *softmax*-based selection, which probabilistically selects an actor, with higher probabilities being assigned to actor-critic pairs with larger Q-values. Second, Gaussian noise can be added to the output of an actor with a probability ε_t, as enabled by an exploration choice of $\lambda = 1$.

For learning purposes, each locomotion cycle is summarized in terms of an experience tuple $\tau = (s,a,r,s',\mu,\lambda)$, where the parameters specify the starting state, action, reward, next state, index of the active actor, and a flag indicating the application of exploration noise. The tuples are captured in a replay memory that stores the most recent 50k tuples and is divided into a critic buffer and an actor buffer. Experiences are collected in batches of 32 tuples, with the motion being restarted as needed, i.e., if the character falls. Tuples that result from added exploration noise are stored in the actor buffer, while the remaining tuples are stored in the critic buffer, which are later used to update the actors and critics respectively. Our use of actor buffers and critic buffers in a MACE-style architecture is new, to the best of our knowledge, although the actor buffer is inspired by recent work on prioritized experience replay [Schaul et al., 2015].

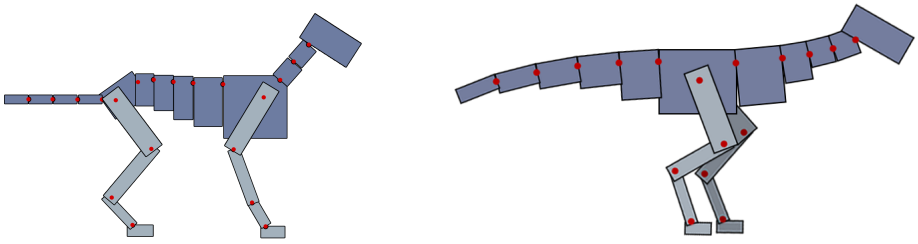


Figure 4.3: Left: 21-link planar dog. **Right:** 19-link raptor.

The outer loop defines the learning process. After the collection of a new minibatch of tuples, a learning iteration is invoked. This involves sampling minibatches of tuples from the replay memory, which are used to improve the actor-critic experts. The actors are updated according to a positive-temporal difference strategy, modeled after CACLA [Van Hasselt, 2012], while the critics are updated using the standard temporal difference updates, regardless of the sign of their temporal differences. For more complex terrains, learning requires on the order of 300k iterations.

4.4 Characters and Terrains

Our planar dog model is a reconstruction of that used in previous work [Peng et al., 2015], although it is smaller, standing approximately 0.5 m tall at the shoulders, as opposed to 0.75 m. It is composed of 21 links and has a mass of 33.7 kg. The pelvis is designated as the root link and each link is connected to its parent link with a revolute joint, yielding a total of 20 internal degrees of freedom and a further 3 degrees of freedom defined by the position and orientation of the root in the world. The raptor is composed of 19 links, with a total mass of 33 kg, and a head-to-tail body length of 1.5 m. The motion of the characters are driven by the application of internal joint torques and is simulated using the Bullet physics engine [Bullet, 2015] at 600 Hz, with friction set to 0.81.

4.4.1 Controllers

Similar to much prior work in physics-based character animation, the motion is driven using joint torques and is guided by a finite state machine. Figure 4.4 shows the four phase-structure of the controller. In each motion phase, the applied torques can be decomposed into three components,

$$\tau = \tau_{\text{spd}} + \tau_g + \tau_{\text{vf}}$$

where τ_{spd} are torques computed from joint-specific stable (semi-implicit) proportional-derivative (SPD) controllers [Tan et al., 2011], τ_g provides gravity compensation for all links, as referred back to the root link, and τ_{vf} implements virtual leg forces for the front and hind legs when they are in contact with the ground, as described in detail in [Peng et al., 2015]. The stable PD controllers are integrated into

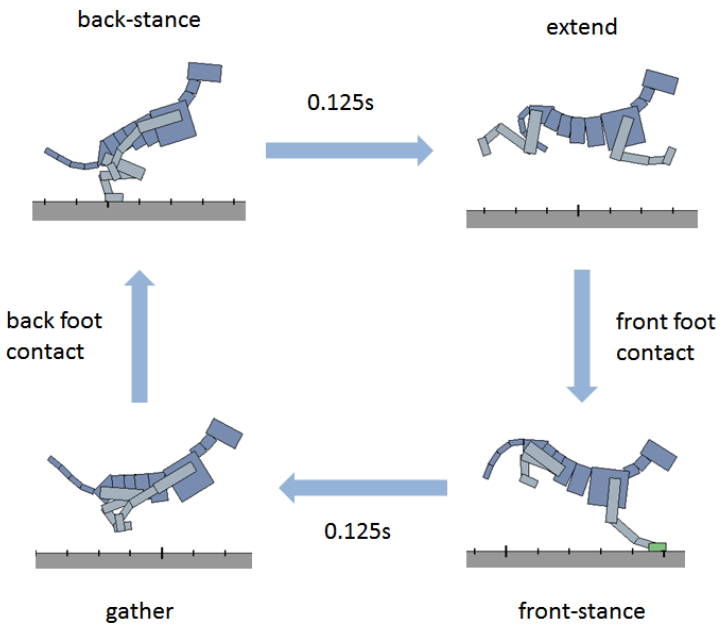


Figure 4.4: Dog controller motion phases.

Bullet with the help of a Featherstone dynamics formulation [Featherstone, 2014]. For our system, conventional PD-controllers with explicit Euler integration require a time-step $\triangle t = 0.0005s$ to remain stable, while SPD remains stable for a time-step of $\triangle t = 0.0017s$, yielding a two-fold speedup once the setup computations are taken into account.

The character's propulsion is principally achieved by exerting forces on the ground with its end effectors, represented by the front and back feet. Virtual force controllers are used to compute the joint torques τ_e needed to exert a desired force f_e on a particular end effector e.

$$\tau_e = \delta_e J_e^T f_e$$

where δ_e is the contact indicator variable for the end effector, and J_e is the end effector Jacobian. The final control forces for the virtual force controllers are the sum of the control forces for the front and back feet.

$$\tau_{vf} = \tau_f + \tau_b$$

4.4.2 Terrain classes

We evaluate the learning method on multiple classes of terrain obstacles that include gaps, steps, walls, and slopes. It is possible to make the obstacles arbitrarily difficult and thus we use environments that are challenging while remaining viable. All of the terrains are represented by 1D height-fields, and generated randomly by drawing uniformly from predefined ranges of values for the parameters that

characterize each type of obstacle. In the flat terrains, gaps are spaced between 4 to 7m apart, with widths ranging from 0.5 to 2m, and a fixed gap depth of 2m. Steps are spaced 5 to 7m apart, with changes in height ranging from 0.1 to 0.4m. Walls are spaced 6 to 8m apart, with heights ranging between 0.25 to 0.5m, and a fixed width of 0.2m. Slopes are generated by varying the change in slope of the terrain at each vertex following a momentum model. The height y_i of vertex i is computed according to

$$y_i = y_{i-1} + \triangle x s_i$$

$$s_i = s_{i-1} + \triangle s_i$$

$$\triangle s_i = \text{sign}(U(-1,1) - \frac{s_{i-1}}{s_{max}}) \times U(0, \triangle s_{max})$$

where $s_{max} = 0.5$ and $\triangle s_{max} = 0.05$, $\triangle x = 0.1$m, and vertices are ordered such that $x_{i-1} < x_i$. When slopes are combined with the various obstacles, the obstacles are adjusted to be smaller than those in the flat terrains.

4.5 Policy Representation

A policy is a mapping between a state space S and an action space A, i.e., $\pi(s) : S \mapsto A$. For our framework, S is a continuous space that describes the state of the character as well as the configuration of the upcoming terrain. The action space A is represented by a 29D continuous space where each action specifies a set of parameters to the FSM. The following sections provide further details about the policy representation.

4.5.1 State

A state s consists of features describing the configuration of the character and the upcoming terrain. The state of the character is represented by its pose q and velocity $\dot{q}$, where q records the positions of the center of mass of each link with respect to the root and $\dot{q}$ records the center of mass velocity of each link. The terrain features, T, consist of a 1D array of samples from the terrain height-field, beginning at the position of the root and spanning 10 m ahead. All heights are expressed relative to the height of the terrain immediately below the root of the character. The samples are spaced 5 cm apart, for a total of 200 height samples. Combined, the final state representation is 283-dimensional. Figure **??** illustrate the character and terrain features.

4.5.2 Actions

A total of 29 controller parameters serve to define the available policy actions. These include specifications of the target spine curvature as well as the target joint angles for the shoulder, elbow, hip, knee, hock, and hind-foot, for each of the four motion phases defined by the controller FSM. Additionally,

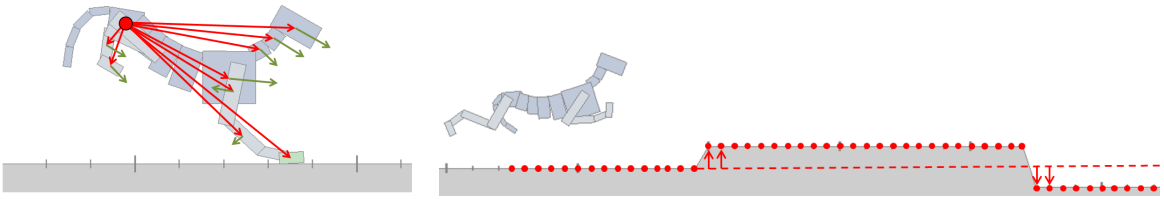


Figure 4.5: Left: The character features consist of the displacements of the centers of mass of all links relative to the root (red) and their linear velocities (green). **Right:** Terrain features consist of height samples of the terrain in front of the character, evenly spaced 5cm apart. All heights are expressed relative to the height of the ground immediately under the root of the character

the x and y components of the hind-leg and front-leg virtual forces, as applied in phases 1 and 3 of the controller, are also part of the parameter set. Phases 2 and 4 apply the same forces as phases 1 and 3, respectively, if the relevant leg is still in contact with the ground. Lastly, the velocity feedback gain for the swing hip (and shoulder) provides one last action parameter.

Prior to learning the policy, a small set of initial actions are created which are used to seed the learning process. The set of actions consists of 4 runs and 4 leaps. All actions are synthesized using a derivative-free optimization process, CMA [Hansen, 2006]. Two runs are produced that travel at approximately 4 m/s and 2 m/s, respectively. These two runs are then interpolated to produce 4 runs of varying speeds. Given a sequence of successive fast-run cycles, a single cycle of that fast-run is then optimized for distance traveled, yielding a 2.5 m leap that can then be executed from the fast run. The leap action is then interpolated with the fast run to generate 4 parametrized leaps that travel different distances.

4.5.3 Reward

In reinforcement learning the reward function, $r(s,a)$, is used as a training signal to encourage or discourage behaviors in the context of a desired task. The reward provides a scalar value reflecting the desirability of a particular state transition that is observed by performing action a starting in the initial state s and resulting in a successor state s'. Figure 4.6 is an example of a sequence of state transitions for terrain traversal. For the terrain traversal task, the reward is provided by

$$r(s,a) = \begin{cases} 0, & \text{character falls during the cycle} \\ e^{-\omega(v^*-v)^2}, & \text{otherwise} \end{cases}$$

where a fall is defined as any link of the character's trunk making contact with the ground for an extended period of time, v is the average horizontal velocity of the center of mass during a cycle, $v^* = 4m/s$ is the desired velocity, and $\omega = 0.5$ is the weight for the velocity error. This simple reward is therefore designed to encourage the character to travel forward at a consistent speed without falling. If the char-

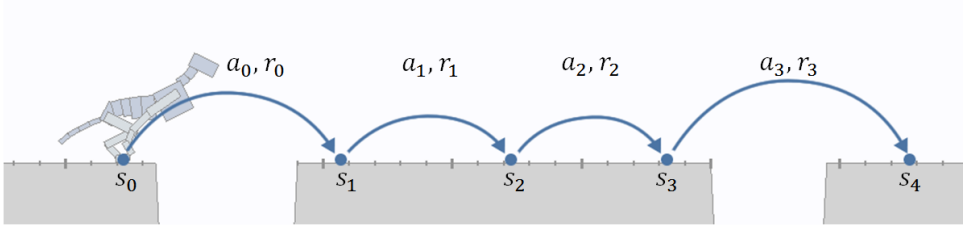


Figure 4.6: Each state transition can be recorded as a tuple $\tau_t = (s_t, a_t, r_t, s'_t)$. s_t is the initial state, a_t is the action taken, s'_t is the resulting state, and r_t is the reward received during cycle t.

acter falls during a cycle, it is reset to a default state and the terrain is regenerated randomly. The goal of learning is to find a control policy that maximizes the expected value of the long term cumulative reward.

4.5.4 Policy Representation

To represent the policy, we use a convolutional neural network, with weights θ, following the structure illustrated in Figure 4.7. The network is queried once at the start of each locomotion cycle. The overall structure of the convolutional network is inspired by the recent work of Minh et al. [Mnih et al., 2015]. For a query state $s = (q, \dot{q}, T)$, the network first processes the terrain features T by passing it through 16 8×1 convolution filters. The resulting feature maps are then convolved with 32 4×1 filters, followed by another layer of 32 4×1 filters. A stride of 1 is used for all convolutional layers. The output of the final convolutional layer is processed by 64 fully-connected units, and the resulting features are then concatenated with the character features q and $\dot{q}$, as inspired by [Levine et al., 2015]. The combined features are processed by a fully-connected layer composed of 256 units. The network then branches into critic and actor subnetworks. The critic sub-network predicts the Q-values for each actor, while each actor subnetwork proposes an action for the given state. All subnetworks follow a similar structure with a fully connected layer of 128 units followed by a linear output layer. The size of the output layers vary depending on the subnetwork, ranging from 3 output units for the critics to 29 units for each actor. The combined network has approximately 570k parameters. Rectified linear units are used for all layers, except for the output layers.

During final runtime use, i.e., when learning has completed, the actor associated with the highest predicted Q-value is selected and its proposed action is applied for the given state.

$$\mu^* = \arg\max_{\mu} Q_\mu(s)$$

$$\pi(s) = A_{\mu^*}(s)$$

The inputs are standardized before being used by the network. The mean and standard deviation for this standardization are determined using data collected from an initial random-action policy. The outputs

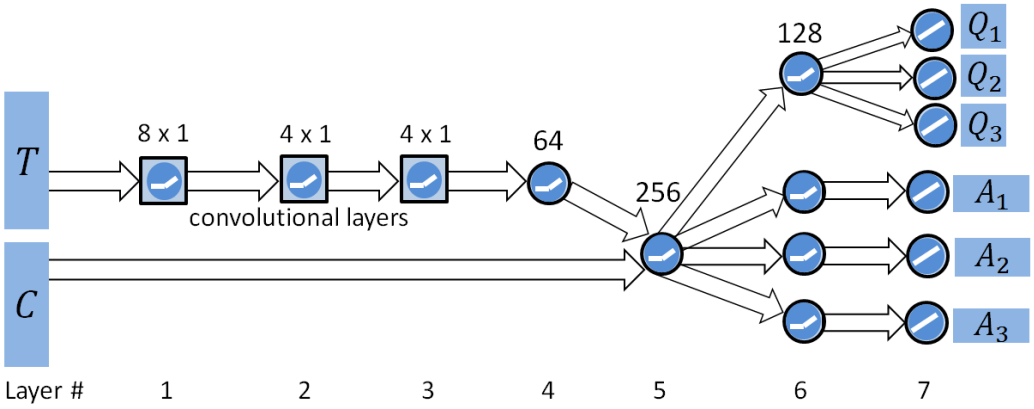


Figure 4.7: Schematic illustration of the MACE convolutional neural network. T and C are the input terrain and character features. Each A_μ represents the proposed action of actor μ, and Q_μ is the critic's predicted reward when activating the corresponding actor.

are also followed by the inverse of a similar transformation in order to allow the network to learn standardized outputs. We apply the following transformation:

$$A_\mu(s) = \Sigma \bar{A}_\mu(s) + \beta_\mu$$

where $\bar{A}_\mu$ is the output of each actor subnetwork, $\Sigma = diag(\sigma_0, \sigma_1, ...)$, $\{\sigma_i\}$ are pre-specified scales for each action parameter, and β_μ is an actor bias. $\{\sigma_i\}$ are selected such that the range of values for each output of $\bar{A}_\mu$ remains approximately within [-1, 1]. This transformation helps to prevent excessively small or large gradients during back-propagation, improving stability during learning. The choice of different biases for each actor helps to encourage specialization of the actors for different situations, a point which we revisit later. For the dog policy, we select a fast run, slow run, and large jump as biases for the three actors.

4.6 Learning

Algorithm 4 illustrates the overall learning process. θ represents the weights of the composite actor-critic network, and $Q_\mu(s|\theta)$ is the Q-value predicted by the critic for the result of activating actor A_μ in state s, where $A_\mu(s|\theta)$ is the action proposed by the actor for s. Since an action is decided by first selecting an actor followed by querying the chosen actor for its proposed action, exploration in MACE can be decomposed into **critic exploration** and **actor exploration**. Critic exploration allows for the selection of an actor other than the "best" actor as predicted by the Q-values. For this we use Boltzmann exploration, which assigns a selection probability p_μ to each actor based on its predicted Q-value:

$$p_\mu(s) = \frac{e^{Q_\mu(s|\theta)/T_t}}{\sum_j e^{Q_{\mu_j}(s|\theta)/T_t}},$$

where T_t is a temperature parameter. Actors with higher predicted values are more likely to be selected, and the bias in favor of actors with higher Q-values can be adjusted via T_t. Actor exploration results in changes to the output of the selected actor, in the form of Gaussian noise that is added to the proposed action. This generates a new action from the continuous action space according to:

$$a = A_\mu(s) + N(0, \Sigma),$$

where Σ are pre-specified scales for each action parameter. Actor exploration is enabled via a Bernoulli selector variable, $\lambda \sim \text{Ber}(\varepsilon_t)$: $\text{Ber}(\varepsilon_t) = 1$ with probability ε_t, and $\text{Ber}(\varepsilon_t) = 0$ otherwise.

Once recorded, experience tuples are stored into separate replay buffers, for use during updates. Tuples collected during actor exploration are stored in an actor buffer D_a, while all other tuples are stored in a critic buffer D_c. This separation allows the actors to be updated using only the off-policy tuples in D_a, and the critics to be updated using only tuples without exploration noise in D_c During a critic update, a minibatch of $n = 32$ tuples $\{\tau_i\}$ are sampled from D_c and used to perform a Bellman backup,

$$y_i = r_i + \gamma \max_\mu Q_\mu(s_i'|\theta)$$

$$\theta \leftarrow \theta + \alpha \left(\frac{1}{n} \sum_i \nabla_\theta Q_{\mu_i}(s_i|\theta)(y_i - Q_{\mu_i}(s_i|\theta)) \right)$$

During an actor update, a minibatch of tuples $\{\tau_j\}$ are sampled from D_a and a CACLA-style positive-temporal difference update is applied to each tuple's respective actor,

$$\delta_j = y_j - \max_\mu Q_\mu(s_j|\theta)$$

$$\text{if } \delta_j > 0: \qquad \theta \leftarrow \theta + \alpha \left(\frac{1}{n} \nabla_\theta A_{\mu_j}(s_j|\theta)(a_j - A_{\mu_j}(s_j|\theta)) \right)$$

Target network: Similarly to [Mnih et al., 2015], we used a separate target network when computing the target values y_i during updates. The target network is fixed for 500 iterations, after which it is updated by copying the most up-to-date weights θ, and then held fixed again for another 500 iterations.

Hyperparameter settings: $m = 32$ steps are simulated before each update. Updates are performed using stochastic gradient descent with momentum, with a learning rate, $\alpha = 0.001$, a weight decay of 0.0005 for regularization, and momentum set to 0.9. ε_t is initialized to 0.9 and linearly annealed to 0.2 after 50k iterations. Similarly, the temperature T_t used in Boltzmann exploration is initialized to 20 and linearly annealed to 0.025 over 50k iterations.

Initialization: The actor and critic buffers are initialized with 50k tuples from a random policy that selects an action uniformly from the initial action set for each cycle. Each of the initial actions are manually associated with a subset of the available actors using the actor bias. When recording an initial experience tuple, μ will be randomly assigned to be one of the actors in its respective set.

Algorithm 4 MACE

$\theta \leftarrow$ random weights
Initialize D_c and D_a with tuples from a random policy

while not done **do**
  for $step = 1, ..., m$ **do**
    $s \leftarrow$ character and terrain initial state
    $\mu \leftarrow$ select each actor with probability $p_{\mu_i} = \frac{\exp(Q_{\mu_i}(s|\theta)/T_i)}{\sum_j(\exp(Q_{\mu_j}(s|\theta)/T_i))}$
    $\lambda \leftarrow Ber(\varepsilon_t)$
    $a \leftarrow A_\mu(s|\theta) + \lambda N_t$
    Apply a and simulate forward 1 cycle
    $s' \leftarrow$ character and terrain terminal state
    $r \leftarrow$ reward
    $\tau \leftarrow (s, a, r, s', \mu, \lambda)$
    if $\lambda = 1$ **then**
      Store τ in D_a
    else
      Store τ in D_c
    end if
  end for

  Update critic:
  Sample minibatch of n tuples $\{\tau_i = (s_i, a_i, r_i, s'_i, \mu_i, \lambda_i)\}$ from D_c
  $y_i \leftarrow r_i + \gamma \max_\mu Q_\mu(s'_i|\theta)$ for each τ_i
  $\theta \leftarrow \theta + \alpha\left(\frac{1}{n}\sum_i \nabla_\theta Q_{\mu_i}(s_i|\theta)(y_i - Q_{\mu_i}(s_i|\theta))\right)$

  Update actors:
  Sample minibatch of n tuples $\{\tau_j = (s_j, a_j, r_j, s'_j, \mu_j, \lambda_j)\}$ from D_a
  for each τ_j **do**
    $y_j \leftarrow \max_\mu Q_\mu(s_j|\theta)$
    $y'_j \leftarrow r_j + \gamma \max_\mu Q_\mu(s'_j|\theta)$
    if $y'_j > y_j$ **then**
      $\theta \leftarrow \theta + \alpha\left(\frac{1}{n}\nabla_\theta A_{\mu_j}(s_j|\theta)(a_j - A_{\mu_j}(s_j|\theta))\right)$
    end if
  end for
end while

4.7 Results

The motions resulting from the learned policies are best seen in the supplemental video. The majority of the results we present are on policies for the dog, as learned for the 7 different classes of terrain shown in Figure 4.13. By default, each policy uses three actor-critic pairs. The final policies are the result of 300k iterations of training, collecting about 10 million tuples, and requiring approximately 20h of compute time on a 16-core cluster, using a multithreaded C++ implementation. All networks are built and trained

using Caffe [Jia et al., 2014b]. Source code is available at https://github.com/xbpeng/DeepTerrainRL. The learning time remains dominated by the cost of simulating the motion of the character rather than the neural network updates. Because of this, we did not pursue the use of GPU-accelerated training. Once the control policies have been learned, all results run faster than real time. Exploration is turned off during the evaluation of the control policies. Separate policies are learned for each class of terrain. The development of a single policy that would be capable of all these terrain classes is left as future work.

In order to evaluate attributes of the learning algorithm, we use the mean distance before a fall as our performance metric, as measured across 250 evaluations. We do not use the Q-function estimates computed by the policies for evaluation as these can over time provide inflated estimates of the cumulative rewards that constitute the true objective.

We compare the final performance of using a mixture of three actor-critic experts, i.e., MACE(3), to two alternatives. First, we compare to Double Q-learning using a set of DNN approximators, $Q_i(s)$, for each of the 8 initial actions [Van Hasselt et al., 2015]. This is motivated by the impressive success of DNNs in learning control policies for discrete action spaces [Mnih et al., 2015]. We use a similar DNN architecture to the network shown in Figure 4.7, except for the exclusion of the actor subnetworks. Second, we compare against the CACLA algorithm [Van Hasselt, 2012]. In our case, CACLA is identical to MACE(1), except without the use of a critic buffer, which means that CACLA uses all experience tuples to update the critic, while MACE(1) only uses the tuples generated without applied exploration noise. In practice, we often find the performance of CACLA and MACE(1) to be similar.

Table 5.4 gives the final performance numbers for all the control policies, including the MACE/Q/-CACLA comparisons. The final mean performance may not always tell the full story, and thus we also compare the learning curves, as we shall discuss shortly. MACE(3) significantly outperforms Q-learning and CACLA for all three terrain classes used for comparison. In two out of three terrain classes, CACLA outperforms Q-learning with discrete actions. This may reflect the importance of being able to learn new actions. As measured by their final performance, all three approaches rank the terrains in the same order in terms of difficulty: `narrow-gaps` (hardest), `slopes-mixed`, `mixed` (easiest). The `tight-gaps` terrain proved to be the most difficult for MACE(3); it consists of the same types of gaps as `narrow-gaps`, but with half the recovery distance between sequences of gaps.

In addition to the dog, we apply MACE(3) to learn control policies for other characters and terrain types, as shown in Figure 4.12. The raptor model uses a 4-states-per-step finite state machine controller, with fixed 0.0825 s state transitions and a final state transition occurring when the swing foot strikes the ground. The control policy is invoked at the start of each step to make a decision with regard to the FSM parameters to apply in the next step. It uses a set of 28 control parameters, similar to those of the dog. We also explored goat locomotion by training the dog to climb never-ending sequences of steep steps that have variable widths and heights. A set of 5 initial actions are provided, corresponding to jumps of heights varying from 0.2 m to 0.75 m, and 3 of which are used as the initial actor biases. Though the

Scenario	Performance (m)
dog + `mixed`: MACE(3)	2094
dog + `mixed`: Q	194
dog + `mixed`: CACLA	1095
dog + `slopes-mixed`: MACE(3)	1364
dog + `slopes-mixed`: Q	110
dog + `slopes-mixed`: CACLA	739
dog + `narrow-gaps`: MACE(3)	176
dog + `narrow-gaps`: Q	74
dog + `narrow-gaps`: CACLA	38
dog + `tight-gaps`: MACE(3)	44
dog + `slopes-gaps`: MACE(3)	1916
dog + `slopes-steps`: MACE(3)	3782
dog + `slopes-walls`: MACE(3)	4312
goat + `variable-steps`: MACE(3)	1004
raptor + `mixed`: MACE(3)	1111
raptor + `slopes-mixed`: MACE(3)	562
raptor + `narrow-gaps`: MACE(3)	145

Table 4.1: Performance of the final policies.

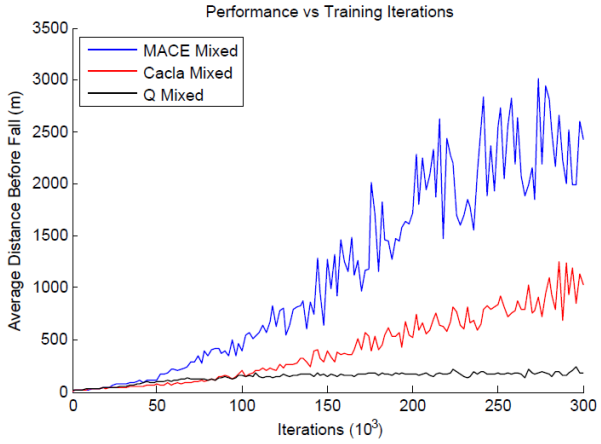
character uses the same underlying model as the dog, it is rendered as a goat in the figures and video as homage to the inspiration for this scenario.

In Figure 4.8, we provide learning curve comparisons for different architectures and algorithm features. As before, we evaluate performance by measuring the mean distance before a fall across 100 randomly generated terrains. Figure 4.8(a) compares MACE(3) with discrete-action-set Q-learning and CACLA, as measured for the dog on mixed terrain. The Q-learning plateaus after 50k iterations, while CACLA continues to improve with further learning iterations, but at a slower rate than MACE(3).

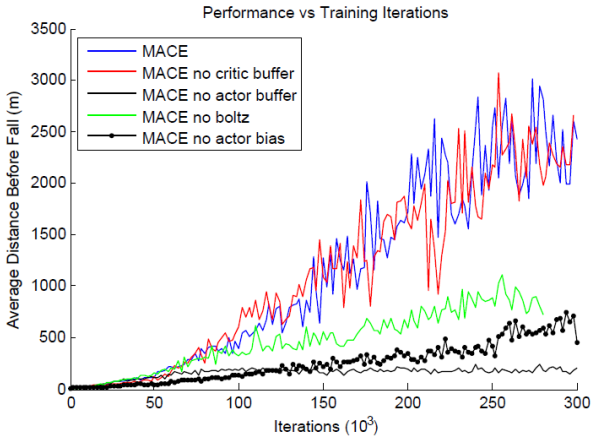
Figure 4.8(b) shows the effects of disabling features of MACE that are enabled by default. Boltzmann exploration has a significant impact on learning. This is likely because it helps to encourage specialization by selecting actors with high predicted Q-values more frequently, while also enabling exploration of multiple actors in cases where they share similar predicted values, thus helping to better disambiguate between the utilities of the different actors. The actor buffer has a large impact, as it allows learning to focus on exploratory actions that yielded an improvement. The initial use of actor bias also proves to be significant, which we attribute to the breaking of initial symmetry between the actor-critic pairs. Lastly, the critic buffer, which enables the critics to learn exclusively from the actions of the deterministic actor policies without exploration noise, does not show a significant benefit for this particular example. However, we found the critic buffer to yield improved learning in earlier experiments, and we further keep this feature because of the more principled learning that it enables.

Figure 4.8(c) shows the impact of the number of actor-critic pairs, comparing MACE(1), MACE(2), MACE(3), MACE(6) for the dog on `mixed` terrain. MACE(2) and MACE(3) yield the best learning

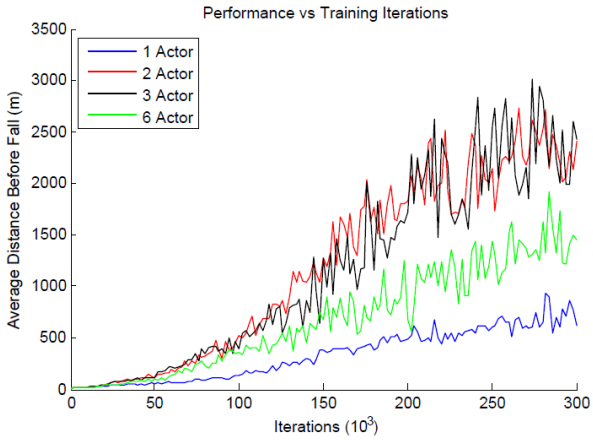
performance, while MACE(6) results in a drop in learning performance, and MACE(1) is the worst. A larger number of actor-critic pairs allows for increased specialization, but results in fewer learning tuples for each actor-critic pair, given that our current MACE learning method only allows for a single actor-critic pair to learn from a particular tuple.



(a) MACE(3) vs CACLA and Q-learning on mixed terrain



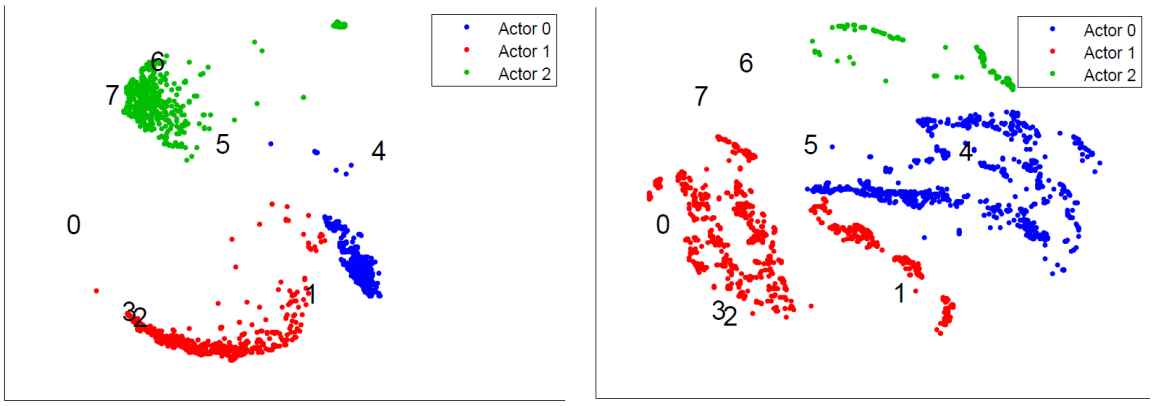
(b) MACE(3) with features disabled.



(c) MACE(n) for various number of actors.

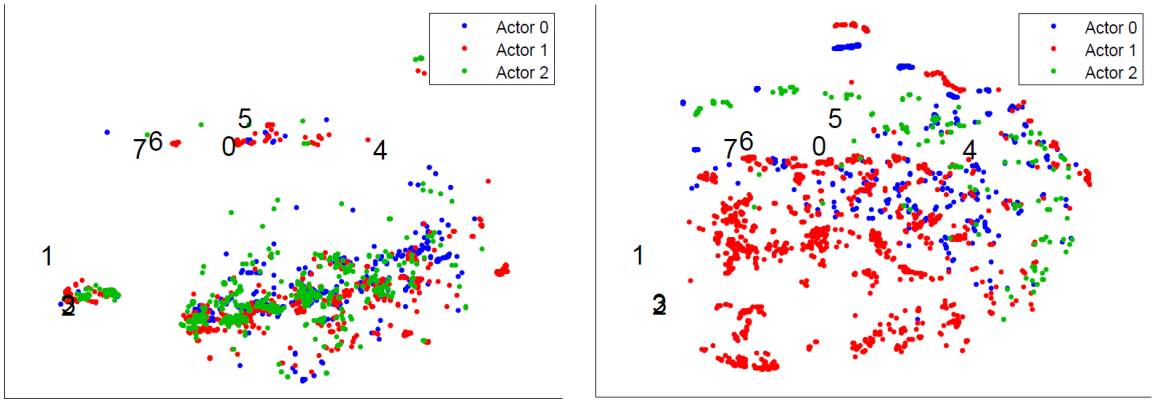
Figure 4.8: Comparisons of learning performance.

Learning good control policies requires learning new actions that yield improved performance. The MACE architecture supports actor-critic specialization by having multiple experts that can specialize in different motions, as well as taking advantage of unique biases for each actor to encourage diversity in their specializations. Figure 4.9 illustrates the space of policy actions early and late in the learning process. This visualization is created using t-SNE (t-distributed Stochastic Neighbor Embedding) [van der Maaten and Hinton, 2008], where a single embedding is constructed using all the action samples collected at various iterations in the training process. Samples from each iteration are then rendered separately. The numbers embedded in the plots correspond to the set of 8 initial actions. The actions begin nearby the initial actions, then evolve over time as demanded by the task, while remaining specialized. The evolution of the actions during learning is best seen in the supplementary videos.



(a) 10k iterations (b) 300k iterations

Figure 4.9: Action space evolution for using MACE(3) with initial actor bias.



(a) 10k iterations (b) 300k iterations

Figure 4.10: Action space evolution for using MACE(3) without initial actor bias.

To encourage actor specialization, the actor-specific initialization bias helps to break symmetry early on in the learning process. Without this initial bias, the benefit of multiple actor-critic experts is diminished. Figure 4.10 illustrates that actor specialization is much less evident when all actors receive the same initial bias, i.e., the fast run for the dog.

We test the generalization capability of the MACE(3) policy for the dog in the `slopes-mixed` terrain, which can be parameterized according to a scale parameter, ψ, that acts a multiplier for the size of all the gaps, steps, and walls. Here, $\psi > 1$ implies more difficult terrains, while $\psi < 1$ implies easier terrains. Figure 4.11 shows that the performance degrades gracefully as the terrain difficulty is increased. To test the performance of the policies when applied to unfamiliar terrain, i.e., terrains not encountered during training, we apply the policy trained in `mixed` to `slopes-mixed`, `slopes-gaps` to `slopes-mixed`, and `slopes-mixed` to `slopes-gaps`. Table 5.4 summarizes the results of each scenario. The policies perform poorly when encountering unfamiliar obstacles, such as slopes for the `mixed` policy and walls for the `slopes-gaps` policy. The `slopes-mixed` policy performs well

Scenario	Perf. (m)
`mixed` policy in `slopes-mixed`	80
`slopes-gaps` policy in `slopes-mixed`	35
`slopes-mixed` policy in `slopes-gaps`	1545

Table 4.2: Performance of applying policies to unfamiliar terrains.

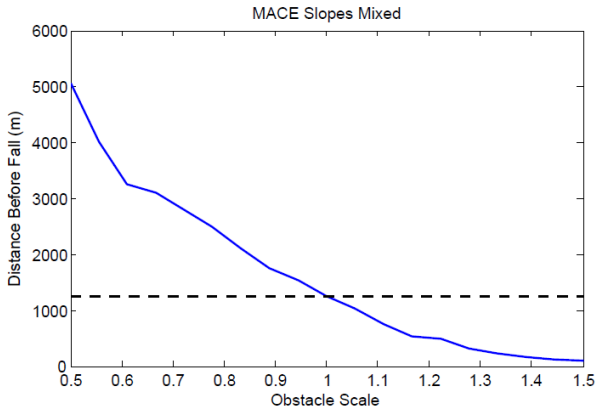


Figure 4.11: Policy generalization to easier and more-difficult terrains.

in `slopes-gaps`, since it has been previously exposed to gaps in `slopes-mixed`, but nonetheless does not reach a similar level of performance as the policy trained specifically for `slopes-gaps`.

4.8 Discussion

The use of a predefined action parameterization helps to provide a significant degree of action abstraction as compared to other recent deep-RL methods that attempt to learn control policies that are directly based on joint torques, e.g., [Levine and Abbeel, 2014, Levine and Koltun, 2014, Mordatch and Todorov, 2014, Mordatch et al., 2015b, Heess et al., 2015, Lillicrap et al., 2015b]. Instead of focusing on the considerable complexities and challenges of *tabula rasa* learning we show that deep-RL enables new capabilities for physics-based character animation, as demonstrated by agile dynamic locomotion across a multitude of terrain types. In future work, it will be interesting to explore the best ways of learning action abstractions and of determining the benefits, if any, of working with actuation that allows for control of stiffness, as allowed by PD-controllers or muscles.

Our overall learning approach is quite different from methods that interleave trajectory optimization (to generate reference data) and neural network regression for supervised learning of the control policy [Levine and Abbeel, 2014, Levine and Koltun, 2014, Mordatch and Todorov, 2014, Mordatch et al., 2015b]. The key role of trajectory optimization makes these methods strongly model-based, with the caveat that the models themselves can possibly be learned. In contrast, our approach does not need an oracle that can provide reference solutions. Another important difference is that much of our network is devoted to processing the high-dimensional terrain description that comprises a majority of our high-D

state description.

Recent methods have also demonstrated the use of more direct policy-gradient methods with deep neural network architectures. This involves chaining a state-action value function approximator in sequence with a control policy approximator, which then allows for backpropagation of the value function gradients back to the control policy parameters, e.g., [Silver et al., 2014b, Lillicrap et al., 2015b]. Recent work applies this approach with predefined parameterized action abstractions [Hausknecht and Stone, 2015a]. We leave comparisons to these methods as important future work. To our knowledge, these methods have not yet been shown to be capable of highly dynamic terrain-adaptive motions. Our work shares many of the same challenges of these methods, such as making stationary-distribution assumptions which are then violated in practice. We do not yet demonstrated the ability to work directly with input images to represent character state features, as shown by others [Lillicrap et al., 2015b].

We have encountered difficult terrains for which learning does not succeed, such as those that have small scale roughness, i.e., bumps 20-40 cm in width that are added to the other terrain features. With extensive training of 500k iterations, the dog is capable of performing robust navigation across the `tight-gaps` terrain, thereby in some sense "aiming for" the best intermediate landing location. However, we have not yet seen that a generalized version of this capability can be achieved, i.e., one that can find a suitable sequence of foot-holds if one exists. Challenging terrains may need to learn new actions and therefore demand a carefully staged learning process. A common failure mode is that of introducing overly-challenging terrains which therefore always cause early failures and a commensurate lack of learning progress.

There remain many architecture hyperparameters that we set based on limited experimentation, including the number of layers, the number of units per layer, regularization parameters, and learning batch size. The space of possible network architectures is large and we have explored this in only a minimal way. The magnitude of exploration for each action parameter is currently manually specified; we wish to develop automated solutions for this. Also of note is that all the actors and critics for any given controller currently share most of their network structure, branching only near the last layers of the network. We would like to explore the benefit, if any, of allowing individual critic and actor networks, thereby allowing additional freedom to utilize more representational power for the relevant aspects of their specialization. We also note that a given actor in the mixture may become irrelevant if its expected performance, as modeled by its critic, is never better than that of its peers for all encountered states. This occurs for MACE(3) when applied to the goat on `variable-steps`, where the distribution of actor usages is (57, 0, 43), with all figures expressed as percentages. Other example usage patterns are (43,48,9) for the dog on `mixed`, (23,66,11) for dog on `slopes-mixed`, and (17,73,10) for dog on `narrow-gaps`. One possible remedy to explore in future work would be to reinitialize an obsolete actor-critic pair with a copy of one of its more successful peers.

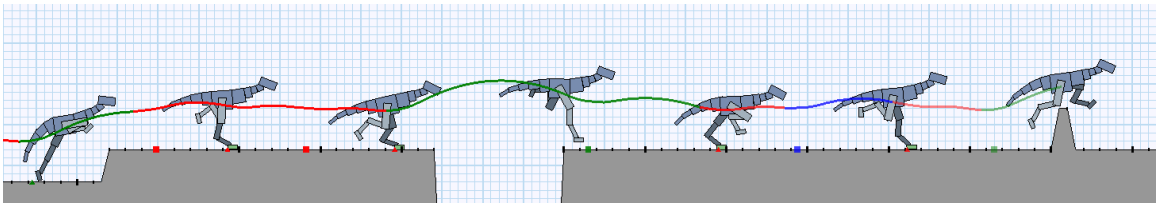
We wish to develop principled methods for integrating multiple controllers that are each trained for a specific class of terrain. This would allow for successful divide-and-conquer development of control

strategies. Recent work has tackled this problem in domains involving discrete actions [Parisotto et al., 2015]. One obvious approach is to use another mixture model, wherein each policy is queried for its expected Q-value, which is in turn modeled as the best Q-value of its individual actors. In effect, each policy would perform its own analysis of the terrain for the suitability of its actions. However, this ignores the fact that the Q-values also depend on the expected distributions of the upcoming character states and terrain, which remain specific to the given class of terrain. Another problem is the lack of a model for the uncertainty of Q-value estimates. Nevertheless, this approach may yield reasonable results in many situations that do not demand extensive terrain-specific anticipation. The addition of models to predict the state would allow for more explicit prediction and planning based on the available set of controllers. We believe that the tradeoff between implicit planning, as embodied by the actor-critic network, and explicit planning, as becomes possible with learned forward models, will be a fruitful area for further reseach.

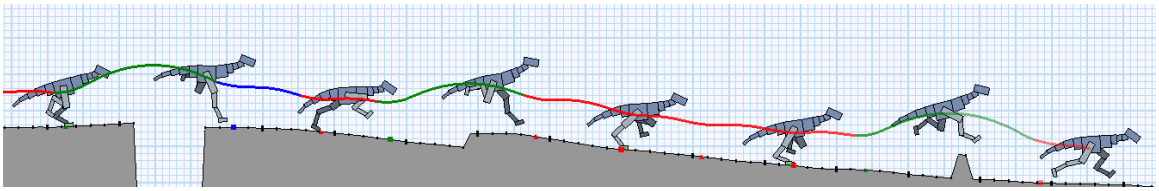
We have not yet demonstrated 3D terrain adaptive locomotion. We expect that the major challenge for 3D will arise in the case of terrain structure that requires identifying specific feasible foothold sequences. The capacity limits of a MACE(n) policy remain unknown.

Control policies for difficult terrains may need to be learned in a progressive fashion via some form of curriculum learning, especially for scenarios where the initial random policy performs so poorly that no meaningful directions for improvement can be found. Self-paced learning is also a promising direction, where the terrain difficulty is increased once a desired level of competence is achieved with the current terrain difficulty. It may be possible to design the terrain generator to work in concert with the learning process by synthesizing terrains with a bias towards situations that are known to be problematic. This would allow for "purposeful practice" and for learning responses to rare events. Other paths towards more data-efficient learning include the ability to transfer aspects of learned solutions between classes of terrain, developing an explicit reduced-dimensionality action space, and learning models of the dynamics, e.g., [Assael et al., 2015]. It would also be interesting to explore the coevolution of the character and its control policies.

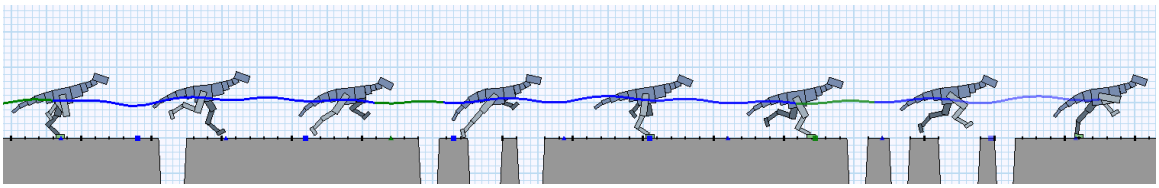
It is not yet clear how to best enable control of the motion style. The parameterization of the action space, the initial bootstrap actions, and the reward function all provide some influence over the final motion styles of the control policy. Available reference motions could be used to help develop the initial actions or used to help design style rewards.



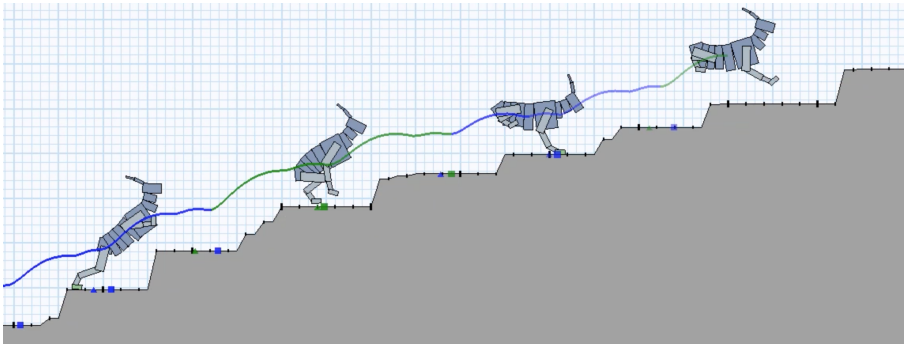
(a) Raptor `mixed` terrain



(b) Raptor `slopes-mixed` terrain

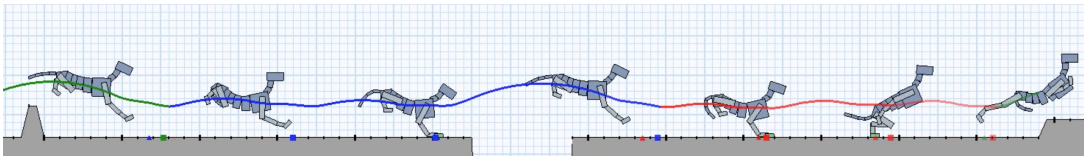


(c) Raptor `narrow-gaps` terrain

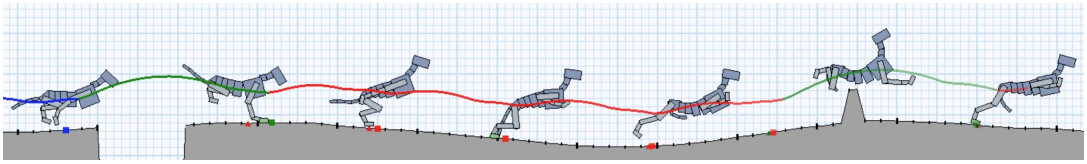


(d) Goat on `variable-steps` terrain

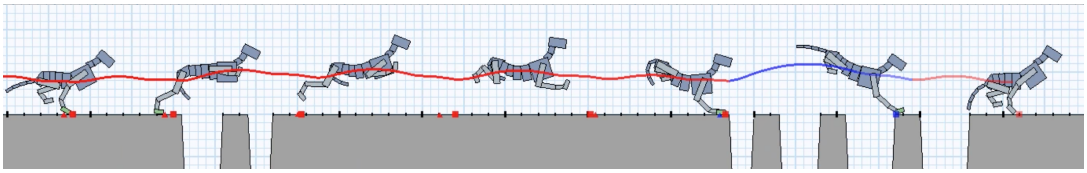
Figure 4.12: Raptor and goat control policies.



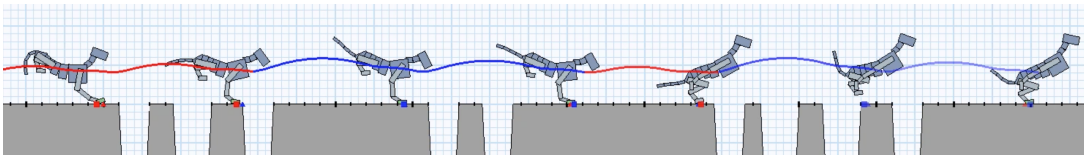
(a) `mixed` terrain



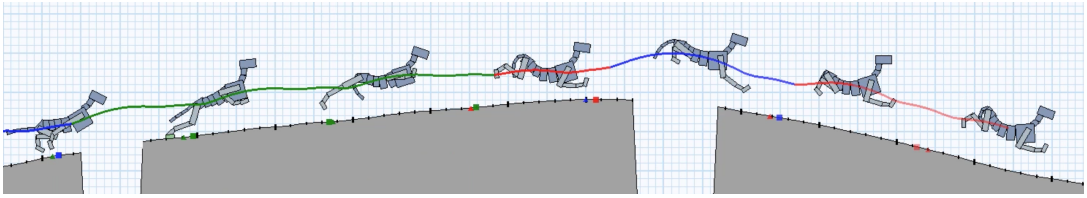
(b) `slopes-mixed` terrain



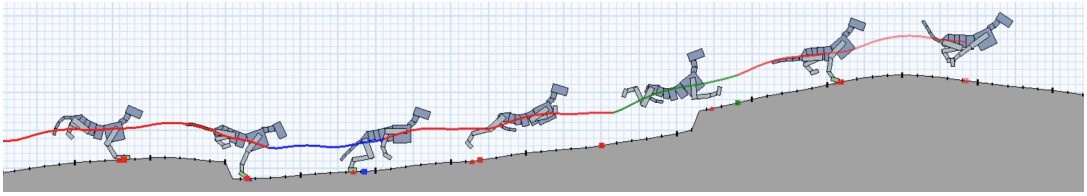
(c) `narrow-gaps` terrain



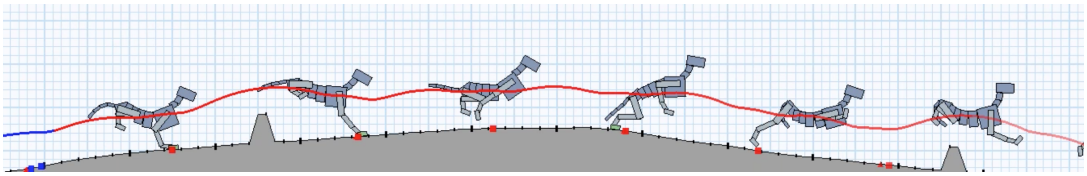
(d) `tight-gaps` terrain



(e) `slopes-gaps` terrain



(f) `slopes-steps` terrain



(g) `slopes-walls` terrain

Figure 4.13: Dog control policies.

Chapter 5

Action Parameterizations

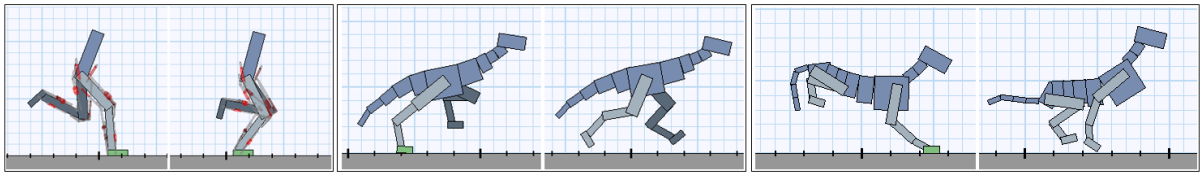


Figure 5.1: Neural network control policies trained for various simulated planar characters.

The use of deep reinforcement learning allows for high-dimensional state descriptors, but little is known about how the choice of action representation impacts the learning difficulty and the resulting performance. The work presented in Chapter 4 leveraged a hand-crafted FMS to provide the policy with a high-level action parameterization. In this chapter, we reduce the manual effort needed to craft high-level action representations by training policies that directly use low-level action parameterizations. We compare the impact of four different low-level action parameterizations (torques, muscle-activations, target joint angles, and target joint-angle velocities) in terms of learning time, policy robustness, motion quality, and policy query rates. Our results are evaluated on a gait-cycle imitation task for multiple planar articulated figures and multiple gaits. We demonstrate that the local feedback provided by higher-level action parameterizations can significantly impact the learning, robustness, and motion quality of the resulting policies.

5.1 Introduction

The introduction of deep learning models to reinforcement learning (RL) has enabled policies to operate directly on high-dimensional, low-level state features. As a result, deep reinforcement learning has demonstrated impressive capabilities, such as developing control policies that can map from input image pixels to output joint torques [Lillicrap et al., 2015a]. However, the motion quality and robustness often falls short of what has been achieved with hand-crafted action abstractions, e.g., Coros et al. [2011a],

Geijtenbeek et al. [2013]. While much is known about the learning of state representations, the *choice of action parameterization* is a design decision whose impact is not yet well understood.

Joint torques can be thought of as the most basic and generic representation for driving the movement of articulated figures, given that muscles and other actuation models eventually result in joint torques. However this ignores the intrinsic embodied nature of biological systems, particularly the synergy between control and biomechanics. Passive-dynamics, such as elasticity and damping from muscles and tendons, play an integral role in shaping motions: they provide mechanisms for energy storage, and mechanical impedance which generates instantaneous feedback without requiring any explicit computation. Loeb coins the term *preflexes* [Loeb, 1995] to describe these effects, and their impact on motion control has been described as providing *intelligence by mechanics* [Blickhan et al., 2007]. This can also be thought of as a kind of partitioning of the computations between the control and physical system.

In this paper we explore the impact of four different actuation models on learning to control dynamic articulated figure locomotion: (1) torques (Tor); (2) activations for musculotendon units (MTU); (3) target joint angles for proportional-derivative controllers (PD); and (4) target joint velocities (Vel). Because DeepRL methods are capable of learning control policies for all these models, it now becomes possible to directly assess how the choice of actuation model affects the learning difficulty. We also assess the learned policies with respect to robustness, motion quality, and policy query rates. We show that action spaces which incorporate local feedback can significantly improve learning speed and performance, while still preserving the generality afforded by torque-level control. Such parameterizations also allow for more complex body structures and subjective improvements in motion quality.

Our specific contributions are: (1) We introduce a DeepRL framework for motion imitation tasks; (2) We evaluate the impact of four different actuation models on the learned control policies according to four criteria; and (3) We propose an optimization approach that combines policy learning and actuator optimization, allowing neural networks to effective control complex muscle models.

5.2 Related Work

DeepRL has driven impressive recent advances in learning motion control, i.e., solving for continuous-action control problems using reinforcement learning. All four of the actions types that we explore have seen previous use in the machine learning literature. WawrzyńSki and Tanwani [2013] use an actor-critic approach with experience replay to learn skills for an octopus arm (actuated by a simple muscle model) and a planar half cheetah (actuated by joint-based PD-controllers). Recent work on deterministic policy gradients [Lillicrap et al., 2015a] and on RL benchmarks, e.g., OpenAI Gym, generally use joint torques as the action space, as do the test suites in recent work [Schulman et al., 2015] on using generalized advantage estimation. Other recent work uses: the PR2 effort control interface as a proxy for torque control [Levine et al., 2015]; joint velocities [Gu et al., 2016a]; velocities under an implicit

control policy [Mordatch et al., 2015a]; or provide abstract actions [Hausknecht and Stone, 2015b]. Our learning procedures are based on prior work using actor-critic approaches with positive temporal difference updates [Van Hasselt, 2012].

Work in biomechanics has long recognized the embodied nature of the control problem and the view that musculotendon systems provide *"preflexes"* [Loeb, 1995] that effectively provide a form intelligence by mechanics [Blickhan et al., 2007], as well as allowing for energy storage. The control strategies for physics-based character simulations in computer animation also use all the forms of actuation that we evaluate in this paper. Representative examples include quadratic programs that solve for joint torques [de Lasa et al., 2010], joint velocities for skilled bicycle stunts [Tan et al., 2014a], muscle models for locomotion [Wang et al., 2012, Geijtenbeek et al., 2013], mixed use of feed-forward torques and joint target angles [Coros et al., 2011a], and joint target angles computed by learned linear (time-indexed) feedback strategies [Liu et al., 2016a]. Lastly, control methods in robotics use a mix of actuation types, including direct-drive torques (or their virtualized equivalents), series elastic actuators, PD control, and velocity control. These methods often rely heavily on model-based solutions and thus we do not describe these in further detail here.

5.3 Task Representation

5.3.1 Reference Motion

In our task, the goal of a policy is to imitate a given reference motion $\{q_t^*\}$ which consists of a sequence of kinematic poses q_t^* in reduced coordinates. The reference velocity $\dot{q_t^*}$ at a given time t is approximated by finite-difference $\dot{q_t^*} \approx \frac{q_{t+\triangle t}^* - q_t^*}{\triangle t}$. Reference motions are generated via either using a recorded simulation result from a preexisting controller ("Sim"), or via manually-authored keyframes. Since hand-crafted reference motions may not be physically realizable, the goal is to closely reproduce a motion while satisfying physical constraints.

5.3.2 States

To define the state of the agent, a feature transformation $\Phi(q, \dot{q})$ is used to extract a set of features from the reduced-coordinate pose q and velocity $\dot{q}$. The features consist of the height of the root (pelvis) from the ground, the position of each link with respect to the root, and the center of mass velocity of each link. When training a policy to imitate a cyclic reference motion $\{q_t^*\}$, knowledge of the motion phase can help simplify learning. Therefore, we augment the state features with a set of target features $\Phi(q_t^*, \dot{q_t}^*)$, resulting in a combined state represented by $s_t = (\Phi(q_t, \dot{q}_t), \Phi(q_t^*, \dot{q_t}^*))$.

5.3.3 Actions

We train separate policies for each of the four actuation models, as described below. Each actuation model also has related actuation parameters, such as feedback gains for PD-controllers and musculotendon properties for MTUs. These parameters can be manually specified, as we do for the PD and Vel models, or they can be optimized for the task at hand, as for the MTU models. Table 5.1 provides a list of actuator parameters for each actuation model.

Target Joint Angles (PD): Each action represents a set of target angles $\hat{q}$, where $\hat{q}^i$ specifies the target angles for joint i. $\hat{q}$ is applied to PD-controllers which compute torques according to

$$\tau^i = k_p^i(\hat{q}^i - q^i) + k_d^i(\hat{\dot{q}}^i - \dot{q}^i)$$

where $\hat{\dot{q}}^i = 0$, and k_p^i and k_d^i are manually-specified gains.

Target Joint Velocities (Vel): Each action specifies a set of target velocities $\hat{\dot{q}}$ which are used to compute torques according to

$$\tau^i = k_d^i(\hat{\dot{q}}^i - \dot{q}^i)$$

where the gains k_d^i are specified to be the same as those used by the PD-controllers for target angles.

Torques (Tor): Each action directly specifies torques for every joint, and constant torques are applied for the duration of a control step. Due to torque limits, actions are bounded by manually specified limits for each joint. Unlike the other actuation models, the torque model does not require additional actuator parameters, and can thus be regarded as requiring the least amount of domain knowledge. Torque limits are excluded from the actuator parameter set as they are common for all parameterizations.

Muscle Activations (MTU): Each action specifies activations for a set of musculotendon units (MTU). Detailed modeling and implementation information are available in Wang et al. [2012]. Each MTU is modeled as a contractile element (CE) attached to a serial elastic element (SE) and parallel elastic element (PE). The force exerted by the MTU can be calculated according to

$$F_{MTU} = F_{SE} = F_{CE} + F_{PE}$$

Both F_{SE} and F_{PE} are modeled as passive springs, while F_{CE} is actively controlled according to

$$F_{CE} = a_{MTU} F_0 f_l(l_{CE}) f_v(v_{CE})$$

with a_{MTU} being the muscle activation, F_0 the maximum isometric force, l_{CE} and v_{CE} being the length and velocity of the contractile element. The functions $f_l(l_{CE})$ and $f_v(v_{CE})$ represent the force-length and force-velocity relationships, modeling the variations in the maximum force that can be exerted by a muscle as a function of its length and contraction velocity. Analytic forms are available in Geyer et al. [2003]. Activations are bounded between [0, 1]. The length of each contractile element l_{CE} are included

as state features. To simplify control and reduce the number of internal state parameters per MTU, the policies directly control muscle activations instead of indirectly through excitations [Wang et al., 2012].

Actuation Model	Actuator Parameters
Target Joint Angles (PD)	proportional gains k_p, derivative gains k_d
Target Joint Velocities (Vel)	derivative gains k_d
Torques (Tor)	none
Muscle Activations (MTU)	optimal contractile element length, serial elastic element rest length, maximum isometric force, pennation, moment arm, maximum moment arm joint orientation, rest joint orientation.

Table 5.1: Actuation models and their respective actuator parameters.

5.3.4 Reward

The reward function consists of a weighted sum of terms that encourage the policy to track a reference motion.

$$r = w_{pose} r_{pose} + w_{vel} r_{vel} + w_{end} r_{end} + w_{root} r_{root} + w_{com} r_{com}$$

$$r_{pose} = \exp\left(-||q^* - q||_W^2\right), \qquad r_{vel} = \exp\left(-||\dot{q}^* - \dot{q}||_W^2\right)$$

$$r_{end} = \exp\left(-40 \sum_e ||x_e^* - x_e||^2\right)$$

$$r_{root} = \exp\left(-10(h_{root}^* - h_{root})^2\right), \qquad r_{com} = \exp\left(-10||\dot{x}_{com}^* - \dot{x}_{com}||^2\right)$$

$$w_{pose} = 0.5,\ w_{vel} = 0.05,\ w_{end} = 0.15,\ w_{root} = 0.1,\ w_{com} = 0.2$$

r_{pose} penalizes deviation of the character pose from the reference pose, and r_{vel} penalizes deviation of the joint velocities. r_{end} and r_{root} accounts for the position error of the end-effectors and root. r_{com} penalizes deviations in the center of mass velocity from that of the reference motion. q and q^* denotes the character pose and reference pose represented in reduced-coordinates, while $\dot{q}$ and $\dot{q}^*$ are the respective joints velocities. W is a manually-specified per joint diagonal weighting matrix. h_{root} is the height of the root from the ground, and $\dot{x}_{com}$ is the center of mass velocity.

5.3.5 Initial State Distribution

We design the initial state distribution, $p_0(s)$, to sample states uniformly along the reference trajectory (Figure 5.2). At the start of each episode, q^* and $\dot{q}^*$ are sampled from the reference trajectory, and used to initialize the pose and velocity of the agent. This helps guide the agent to explore states near the target trajectory. Figure 5.2 illustrates a comparison between fixed and sampled initial state distributions.

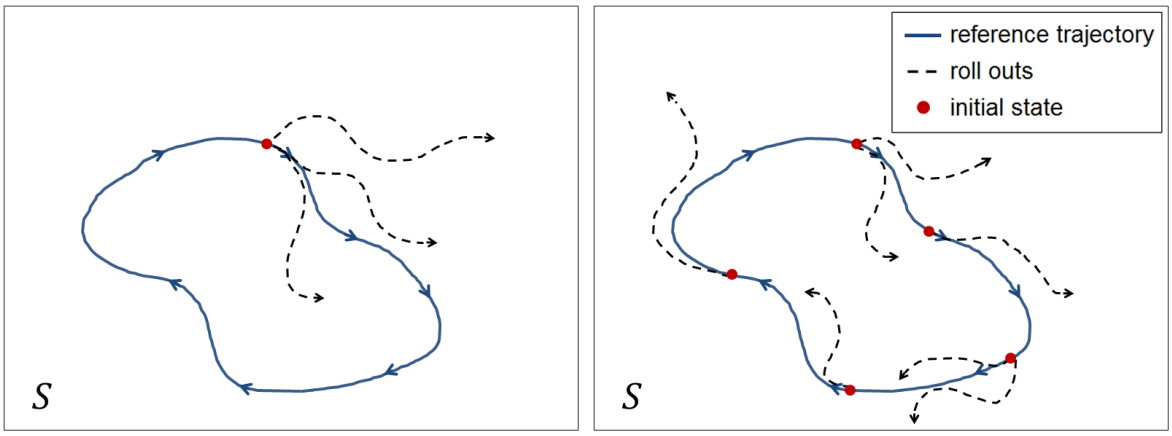


Figure 5.2: Left: fixed initial state biases agent to regions of the state space near the initial state, particular during early iterations of training. **Right:** initial states sampled from reference trajectory allows agent to explore the state space more uniformly around the reference trajectory.

5.4 Learning Framework

For our learning algorithm, we adapt a positive temporal difference (PTD) update as proposed by Van Hasselt [2012]. Stochastic policies are used during training for exploration, while deterministic policies are used for evaluation at runtime. The choice between a stochastic and deterministic policy can be specified by the addition of a binary indicator variable $\lambda \in [0,1]$

$$a_t = \mu(s_t|\theta_\pi) + \lambda N(0,\Sigma)$$

where $\lambda = 1$ corresponds to a stochastic policy with exploration noise, and $\lambda = 0$ corresponds to a deterministic policy that always selects the mean of the distribution. Noise from a stochastic policy will result in a state distribution that differs from that of the deterministic policy at runtime. To mitigate this discrepancy, we incorporate ε-greedy exploration to the original Gaussian exploration strategy. During training, λ is determined by a Bernoulli random variable $\lambda \sim \text{Ber}(\varepsilon)$, where $\lambda = 1$ with probability $\varepsilon \in [0,1]$. The exploration rate ε is annealed linearly from 1 to 0.2 over 500k iterations, which slowly adjusts the state distribution encountered during training to better resemble the distribution at runtime. Since the policy gradient is defined for stochastic policies, only tuples recorded with exploration noise (i.e. $\lambda = 1$) can be used to update the actor, while the critic can be updated using all tuples.

Training proceeds episodically, where the initial state of each episode is sampled from $p_0(s)$, and the episode duration is drawn from an exponential distribution with a mean of 2s. To discourage falling, an episode will also terminate if any part of the character's trunk makes contact with the ground for an extended period of time, leaving the agent with zero reward for all subsequent steps. Algorithm 7 summarizes the complete learning process. A summary of the hyperparameter settings is available in Table 5.2.

Parameter	Value	Description
γ	0.9	cumulative reward discount factor
α_π	0.001	actor learning rate
α_V	0.01	critic learning rate
momentum	0.9	stochastic gradient descent momentum
ϕ weight decay	0	L2 regularizer for critic parameters
θ_π weight decay	0.0005	L2 regularizer for actor parameters
minibatch size	32	tuples per stochastic gradient descent step
replay memory size	500000	number of the most recent tuples stored for future updates

Table 5.2: Training hyperparameters.

Bounded Action Space: Properties such as torque and neural activation limits result in bounds on the range of values that can be assumed by actions for a particular parameterization. Improper enforcement of these bounds can lead to unstable learning as the gradient information outside the bounds may not be reliable [Hausknecht and Stone, 2015b]. To ensure that all actions respect their bounds, we adopt a method similar to the inverting gradients approach proposed by Hausknecht and Stone [2015b]. Let $\nabla a = (a - \mu(s))A(s,a)$ be the empirical action gradient from the policy gradient estimate of a Gaussian policy. Given the lower and upper bounds $[l^i, u^i]$ of the ith action parameter, the bounded gradient of the ith action parameter $\nabla \tilde{a}^i$ is determined according to

$$\nabla \tilde{a}^i = \begin{cases} l^i - \mu^i(s), & \mu^i(s) < l^i \text{ and } \nabla a^i < 0 \\ u^i - \mu^i(s), & \mu^i(s) > u^i \text{ and } \nabla a^i > 0 \\ \nabla a^i, & \text{otherwise} \end{cases}$$

Unlike the inverting gradients approach, which scales all gradients depending on proximity to the bounds, this method preserves the empirical gradients when bounds are respected, and alters the gradients only when bounds are violated.

MTU Actuator Optimization: Actuation models such as MTUs are defined by further parameters whose values impact performance [Geijtenbeek et al., 2013]. Geyer et al. [2003] uses existing anatomical estimates for humans to determine MTU parameters, but such data is not be available for more arbitrary creatures. Alternatively, Geijtenbeek et al. [2013] uses covariance matrix adaptation (CMA), a derivative-free evolutionary search strategy, to simultaneously optimize MTU and policy parameters. This approach is limited to policies with reasonably low dimensional parameter spaces, and is thus ill-suited for neural network models with hundreds of thousands of parameters. To avoid manual-tuning of actuator parameters, we propose a heuristic approach that alternates between policy learning and actuator optimization. The actuator parameters ψ can be interpreted as a parameterization of the dynamics of the system $p(s'|s,a,\psi)$. The expected cumulative reward $J(\theta_\pi)$ can then be re-parameterized according to

$$J(\theta_\pi, \psi) = \int_S d(s|\theta_\pi, \psi) \int_A \pi_\theta(s, a|\theta_\pi) \mathbb{A}(s, a) da\, ds$$

where $d(s|\theta_\pi, \psi) = \int_S \sum_{t=0}^{T} \gamma^t p_0(s_0) p(s_0 \rightarrow s|t, \pi_\theta, \psi) ds_0$ is the discounted state distribution [Silver et al., 2014a]. θ_π and ψ are then learned in an alternating fashion as per Algorithm 5. This alternating method optimizes both the control and dynamics in order to maximize the expected value of the agent, as analogous to the role of evolution in biomechanics. During each pass, the policy parameters θ_π are trained to improve the agent's expected value for a fixed set of actuator parameters ψ. Next, ψ is optimized using CMA to improve performance while keeping θ_π fixed. The expected value of each CMA sample of ψ is estimated using the average (undiscounted) cumulative reward over multiple rollouts.

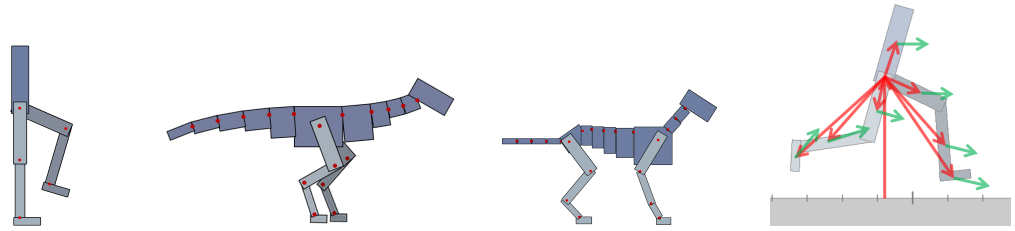


Figure 5.3: Simulated articulated figures and their state representation. Revolute joints connect all links. **left-to-right:** 7-link biped; 19-link raptor; 21-link dog; State features: root height, relative position (red) of each link with respect to the root and their respective linear velocity (green).

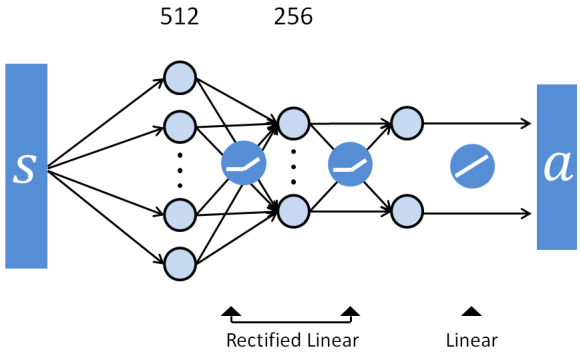


Figure 5.4: Neural Network Architecture. Each policy is represented by a three layered network, with 512 and 256 fully-connected hidden units, followed by a linear output layer.

Algorithm 5 Alternating Actuator Optimization

1: $\theta_\pi \leftarrow \theta_\pi^0$
2: $\psi \leftarrow \psi^0$
3: **while** not done **do**
4: $\quad \theta_\pi \leftarrow \text{argmax}_{\theta'_\pi} J(\theta'_\pi, \psi)$ with Algorithm 7
5: $\quad \psi \leftarrow \text{argmax}_{\psi'} J(\theta_\pi, \psi')$ with CMA
6: **end while**

Algorithm 6 Actor-critic Learning Using Positive Temporal Differences

$\theta_\pi \leftarrow \theta_\pi^0$
$\theta_V \leftarrow \theta_V^0$

while not done **do**
  for $step = 1, ..., m$ **do**
    $s \leftarrow$ start state
    $\lambda \leftarrow \text{Ber}(\varepsilon_t)$
    $a \leftarrow \mu(s|\theta_\pi) + \lambda N(0, \Sigma)$
    Apply a and simulate forward 1 step
    $s' \leftarrow$ end state
    $r \leftarrow$ reward
    $\tau \leftarrow (s, a, r, s', \lambda)$
    store τ in replay memory

    if episode terminated **then**
      Sample s_0 from $p_0(s)$
      Reinitialize state s to s_0
    end if
  end for

  Update critic:
  Sample minibatch of n tuples $\{\tau_i = (s_i, a_i, r_i, \lambda_i, s_i')\}$ from replay memory
  for each τ_i **do**
    $\delta_i \leftarrow r_i + \gamma V(s_i'|\theta_V) - V(s_i|\theta_V)$
    $\theta_V \leftarrow \theta_V + \alpha_V \frac{1}{n} \nabla_{\theta_V} V(s_i|\theta_V) \delta_i$
  end for

  Update actor:
  Sample minibatch of n tuples $\{\tau_j = (s_j, a_j, r_j, \lambda_j, s_j')\}$ from replay memory where $\lambda_j = 1$
  for each τ_j **do**
    $\delta_j \leftarrow r_j + \gamma V(s_j'|\theta_V) - V(s_j|\theta_V)$
    if $\delta_j > 0$ **then**
      $\nabla a_j \leftarrow a_j - \mu(s_j|\theta_\pi)$
      $\nabla \tilde{a}_j \leftarrow \text{BoundActionGradient}(\nabla a_j, \mu(s_j|\theta_\pi))$
      $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \frac{1}{n} \nabla_{\theta_\pi} \mu(s_j|\theta_\pi) \Sigma^{-1} \nabla \tilde{a}_j$
    end if
  end for
end while

5.5 Results

The motions are best seen in the supplemental video https://youtu.be/L3vDo3nLI98. We evaluate the action parameterizations by training policies for a simulated 2D biped, dog, and raptor as shown in Figure 5.3. Depending on the agent and the actuation model, our systems have 58–214 state dimensions, 6–44 action dimensions, and 0–282 actuator parameters, as summarized in Table 5.3. The MTU models

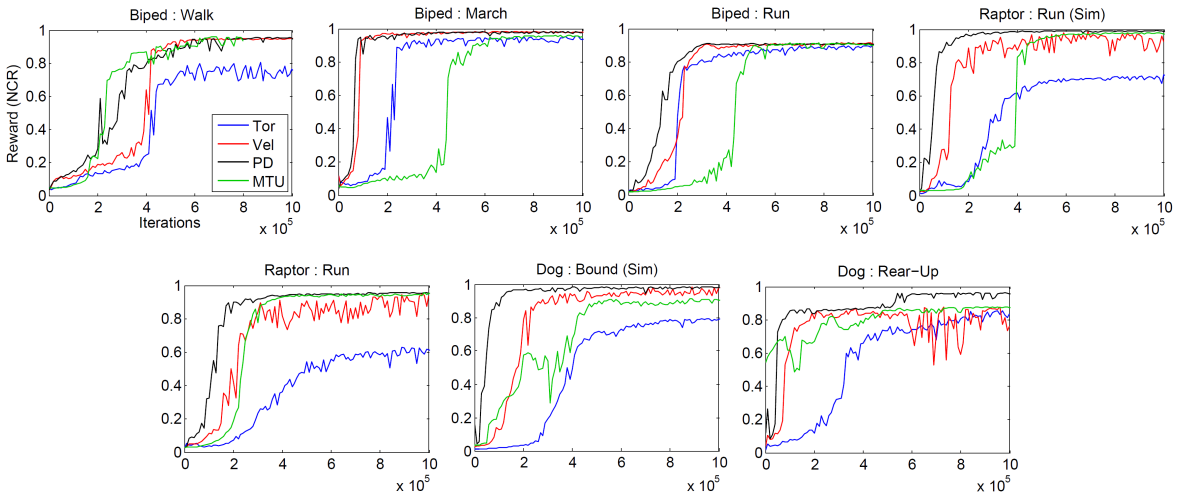


Figure 5.5: Learning curves for each policy during 1 million iterations.

have at least double the number of action parameters because they come in antagonistic pairs. As well, additional MTUs are used for the legs to more accurately reflect bipedal biomechanics. This includes MTUs that span multiple joints.

Character + Actuation Model	State Parameters	Action Parameters	Actuator Parameters
Biped + Tor	58	6	0
Biped + Vel	58	6	6
Biped + PD	58	6	12
Biped + MTU	74	16	114
Raptor + Tor	154	18	0
Raptor + Vel	154	18	18
Raptor + PD	154	18	36
Raptor + MTU	194	40	258
Dog + Tor	170	20	0
Dog + Vel	170	20	20
Dog + PD	170	20	40
Dog + MTU	214	44	282

Table 5.3: The number of state, action, and actuation model parameters for different characters and actuation models.

Each policy is represented by a three layer neural network, as illustrated in Figure 5.4 with 512 and 256 fully-connected units, followed by a linear output layer where the number of output units vary according to the number of action parameters for each character and actuation model. ReLU activation functions are used for both hidden layers. Each network has approximately 200k parameters. The value function is represented by a similar network, except having a single linear output unit. The policies are queried at 60Hz for a control step of about 0.0167s. Each network is randomly initialized and trained for about 1 million iterations, requiring 32 million tuples, the equivalent of approximately 6 days of simulated time. Each policy requires about 10 hours for the biped, and 20 hours for the raptor and dog

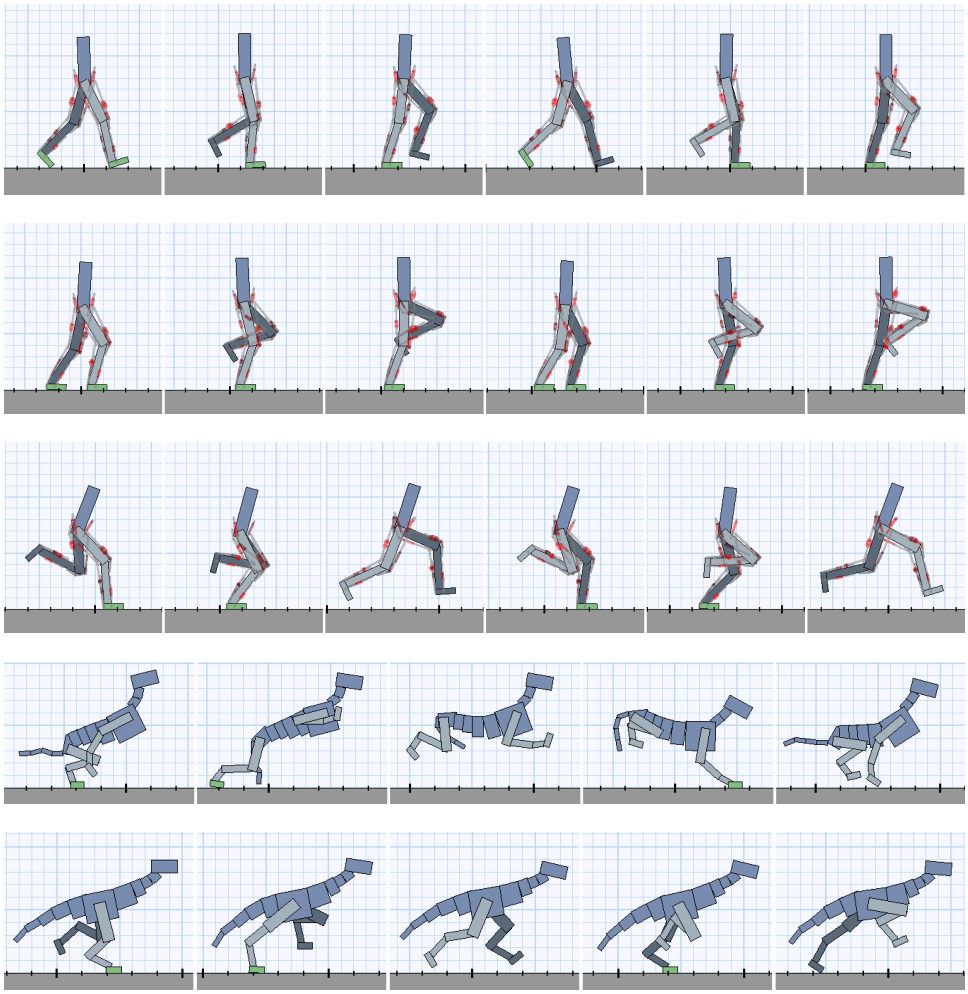


Figure 5.6: Simulated motions. The biped uses an MTU action space while the dog and raptor are driven by a PD action space.

on an 8-core Intel Xeon E5-2687W.

Only the actuator parameters for MTUs are optimized with Algorithm 5, since the parameters for the other actuation models are few and reasonably intuitive to determine. The initial actuator parameters ψ^0 are manually specified, while the initial policy parameters θ_π^0 are randomly initialized. Each pass optimizes ψ using CMA for 250 generations with 16 samples per generation, and θ_π is trained for 250k iterations. Parameters are initialized with values from the previous pass. The expected value of each CMA sample of ψ is estimated using the average cumulative reward over 16 rollouts with a duration of 10s each. Separate MTU parameters are optimized for each character and motion. Each set of parameters is optimized for 6 passes following Algorithm 5, requiring approximately 50 hours. Figure 5.7 illustrates the performance improvement per pass. Figure 5.8 compares the performance of MTUs before and after optimization. For most examples, the optimized actuator parameters significantly improve learning speed and final performance. For the sake of comparison, after a set of actuator parameters has been optimized, a new policy is retrained with the new actuator parameters and its

performance compared to the other actuation models.

Policy Performance and Learning Speed: Figure 5.5 shows learning curves for the policies and the performance of the final policies are summarized in Table 5.4. Performance is evaluated using the normalized cumulative reward (NCR), calculated from the average cumulative reward over 32 episodes with lengths of 10s, and normalized by the maximum and minimum cumulative reward possible for each episode. No discounting is applied when calculating the NCR. The initial state of each episode is sampled from the reference motion according to $p(s_0)$. To compare learning speeds, we use the normalized area under each learning curve (AUC) as a proxy for the learning speed of a particular actuation model, where 0 represents the worst possible performance and no progress during training, and 1 represents the best possible performance without requiring training. Figure 5.7 illustrates the improvement in performance during the optimization process, as applied to motions for three different agents. Figure 5.8 compares the learning curves for the initial and final MTU parameters, for the same three motions.

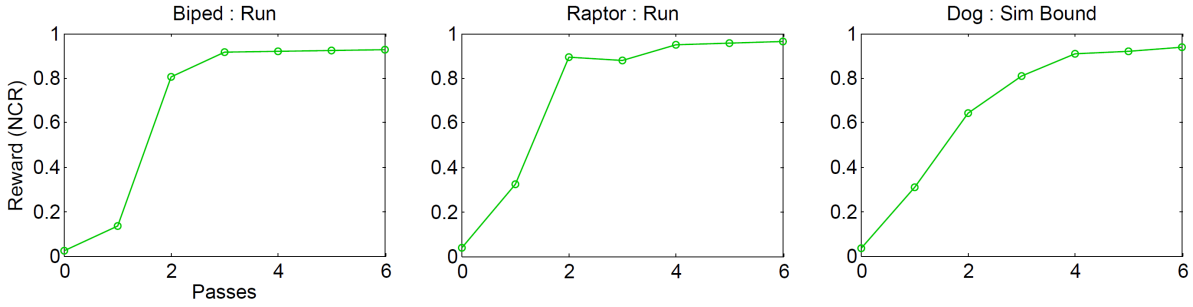


Figure 5.7: Performance of intermediate MTU policies and actuator parameters per pass of actuator optimization following Algorithm 5.

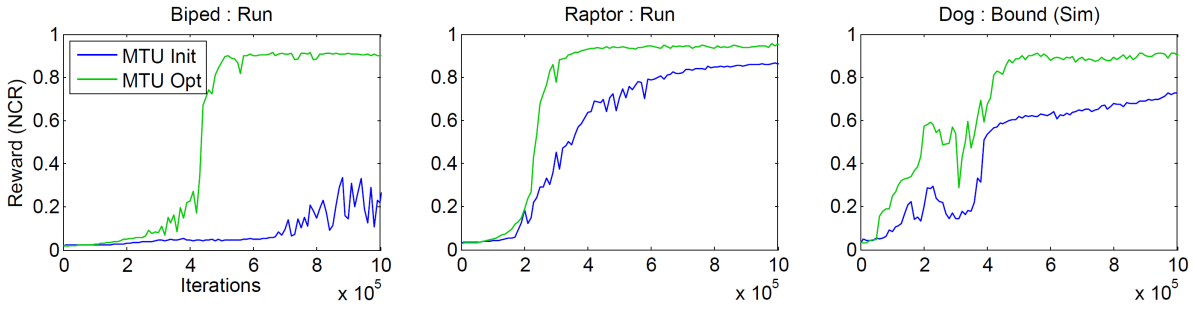


Figure 5.8: Learning curves comparing initial and optimized MTU parameters.

PD performs well across all examples, achieving comparable-to-the-best performance for all motions. PD also learns faster than the other parameterizations for 5 of the 7 motions. The final performance of Tor is among the poorest for all the motions. Differences in performance appear more pronounced as characters become more complex. For the simple 7-link biped, most parameterizations achieve similar performance. However, for the more complex dog and raptor, the performance of Tor policies deteriorate with respect to other policies such as PD and Vel. MTU policies often exhibited

Character + Actuation	Motion	Performance (NCR)	Learning Speed (AUC)
Biped + Tor	Walk	0.7662 ± 0.3117	0.4788
Biped + Vel	Walk	0.9520 ± 0.0034	0.6308
Biped + PD	Walk	0.9524 ± 0.0034	0.6997
Biped + MTU	Walk	$\mathbf{0.9584 \pm 0.0065}$	**0.7165**
Biped + Tor	March	0.9353 ± 0.0072	0.7478
Biped + Vel	March	$\mathbf{0.9784 \pm 0.0018}$	0.9035
Biped + PD	March	0.9767 ± 0.0068	**0.9136**
Biped + MTU	March	0.9484 ± 0.0021	0.5587
Biped + Tor	Run	0.9032 ± 0.0102	0.6938
Biped + Vel	Run	$\mathbf{0.9070 \pm 0.0106}$	0.7301
Biped + PD	Run	0.9057 ± 0.0056	**0.7880**
Biped + MTU	Run	0.8988 ± 0.0094	0.5360
Raptor + Tor	Run (Sim)	0.7265 ± 0.0037	0.5061
Raptor + Vel	Run (Sim)	0.9612 ± 0.0055	0.8118
Raptor + PD	Run (Sim)	$\mathbf{0.9863 \pm 0.0017}$	**0.9282**
Raptor + MTU	Run (Sim)	0.9708 ± 0.0023	0.6330
Raptor + Tor	Run	0.6141 ± 0.0091	0.3814
Raptor + Vel	Run	0.8732 ± 0.0037	0.7008
Raptor + PD	Run	$\mathbf{0.9548 \pm 0.0010}$	**0.8372**
Raptor + MTU	Run	0.9533 ± 0.0015	0.7258
Dog + Tor	Bound (Sim)	0.7888 ± 0.0046	0.4895
Dog + Vel	Bound (Sim)	0.9788 ± 0.0044	0.7862
Dog + PD	Bound (Sim)	$\mathbf{0.9797 \pm 0.0012}$	**0.9280**
Dog + MTU	Bound (Sim)	0.9033 ± 0.0029	0.6825
Dog + Tor	Rear-Up	0.8151 ± 0.0113	0.5550
Dog + Vel	Rear-Up	0.7364 ± 0.2707	0.7454
Dog + PD	Rear-Up	$\mathbf{0.9565 \pm 0.0058}$	**0.8701**
Dog + MTU	Rear-Up	0.8744 ± 0.2566	0.7932

Table 5.4: Performance of policies trained for the various characters and actuation models. Performance is measured using the normalized cumulative reward (NCR) and learning speed is represented by the normalized area under each learning curve (AUC). The best performing parameterizations for each character and motion are in bold.

the slowest learning speed, which may be a consequence of the higher dimensional action spaces, i.e., requiring antagonistic muscle pairs, and complex muscle dynamics. Nonetheless, once optimized, the MTU policies produce more natural motions and responsive behaviors as compared to other parameterizations. We note that the naturalness of motions is not well captured by the reward, since it primarily gauges similarity to the reference motion, which may not be representative of natural responses when perturbed from the nominal trajectory. Action and torque trajectories over one cycle of a desired motion are available in Figure 5.9.

Sensitivity Analysis: Due to the non-convex nature of the optimization problem, policies trained from different random initializations may converge to different results. To analyze the sensitivity of the results, we compare policies trained with different initializations and design decisions. Figure 5.10

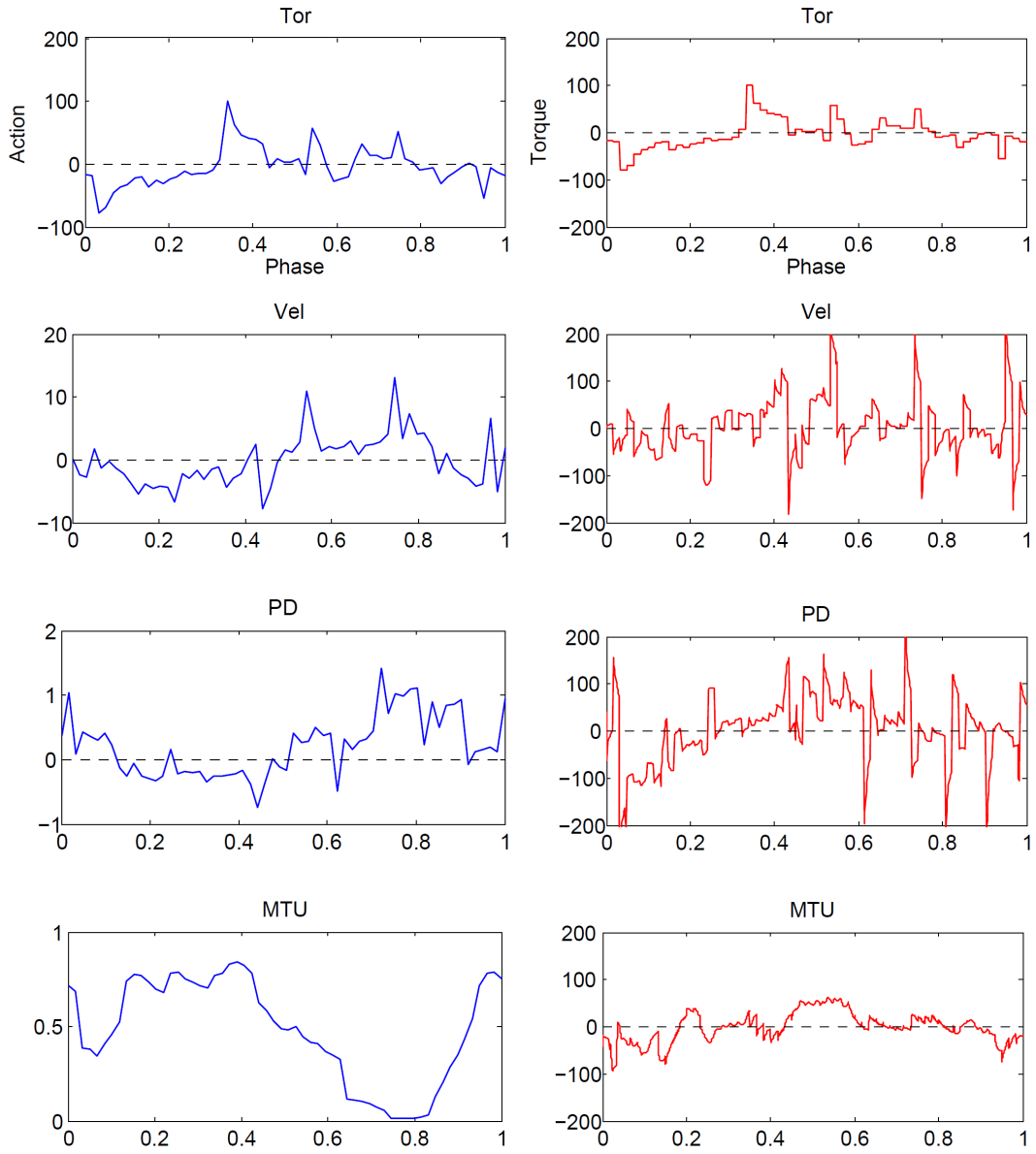


Figure 5.9: Policy actions over time and the resulting torques for the four action types. Data is from one biped walk cycle (1s). Left: Actions (60 Hz), for the right hip for PD, Vel, and Tor, and the right gluteal muscle for MTU. Right: Torques applied to the right hip joint, sampled at 600 Hz.

compares the learning curves from multiple policies trained using different random initializations of the networks. Four policies are trained for each actuation model. The results for a particular actuation model are similar across different runs, and the trends between the various actuation models also appear to be consistent. To evaluate the sensitivity to the amount of exploration noise applied during training, we trained policies where the standard deviation of the action distribution is twice and half of the default values. Figure 5.11 illustrates the learning curves for each policy. Overall, the performance of the policies do not appear to change significantly for the particular range of values. Finally, Figure 5.12

compares the results using different network architectures. The network variations include doubling the number of units in both hidden layers, halving the number of hidden units, and inserting an additional layer with 512 units between the two existing hidden layers. The choice of network structure does not appear to have a noticeable impact on the results, and the differences between the actuation models appear to be consistent across the different networks.

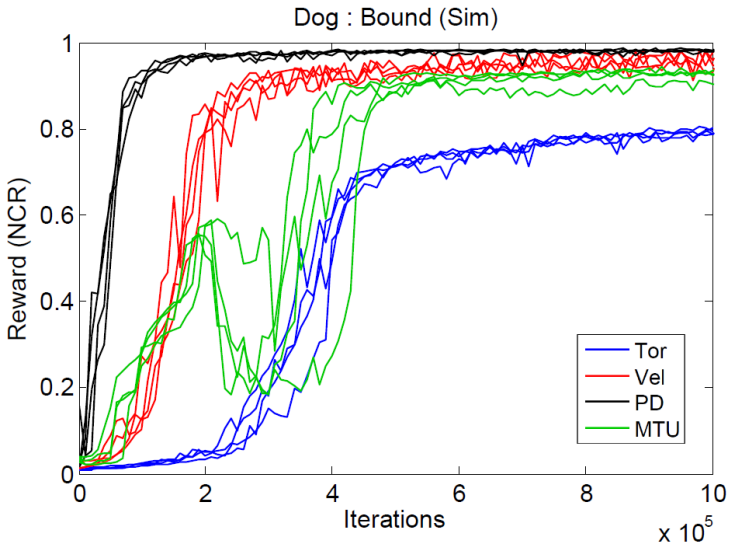


Figure 5.10: Learning curves from different random network initializations. Four policies are trained for each actuation model.

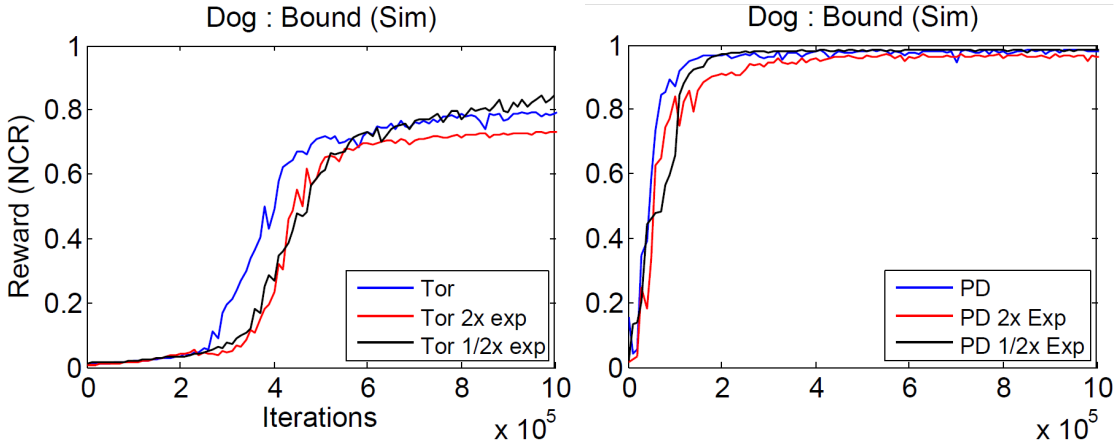


Figure 5.11: Learning curves comparing the effects of scaling the standard deviation of the action distribution by 1x, 2x, and 1/2x.

Policy Robustness: To evaluate robustness, we recorded the NCR achieved by each policy when subjected to external perturbations. The perturbations assume the form of random forces applied to the trunk of the characters. Figure 5.13 illustrates the performance of the policies when subjected to perturbations of different magnitudes. The magnitude of the forces are constant, but direction varies randomly. Each force is applied for 0.1 to 0.4s, with 1 to 4s between each perturbation. Performance

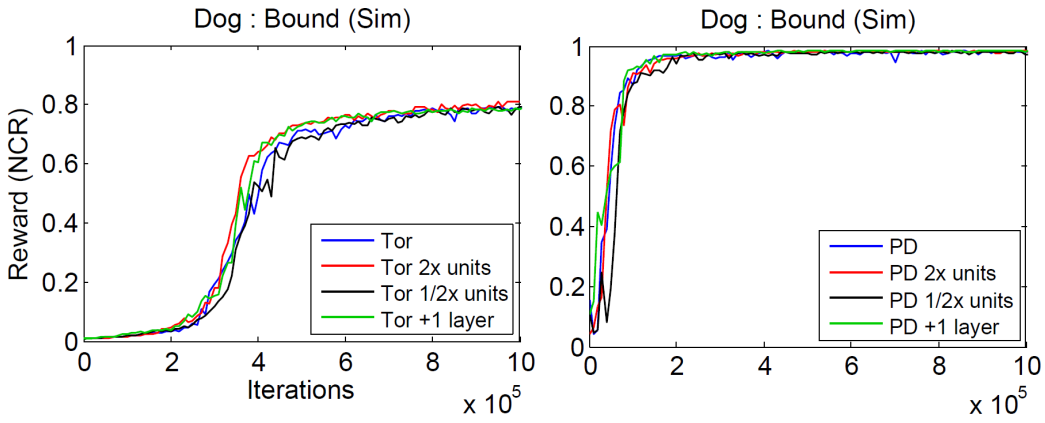


Figure 5.12: Learning curves for different network architectures. The network structures include, doubling the number of units in each hidden layer, halving the number of units, and inserting an additional hidden layer with 512 units between the two existing hidden layers.

is estimated using the average over 128 episodes of length 20s each. For the biped walk, the Tor policy is significantly less robust than those for the other types of actions, while the MTU policy is the least robust for the raptor run. Overall, the PD policies are among the most robust for all the motions. In addition to external forces, we also evaluate the robustness when locomoting over randomly generated terrain consisting of bumps with varying heights and slopes with varying steepness (Figure 5.14). There are a few consistent patterns for this test. The Vel and MTU policies are significantly worse than the Tor and PD policies for the dog bound on the bumpy terrain. The unnatural jittery behavior of the dog Tor policy proves to be surprisingly robust for this scenario. We suspect that the behavior prevents the trunk from contacting the ground for extended periods for time, and thereby escaping our system's fall detection.

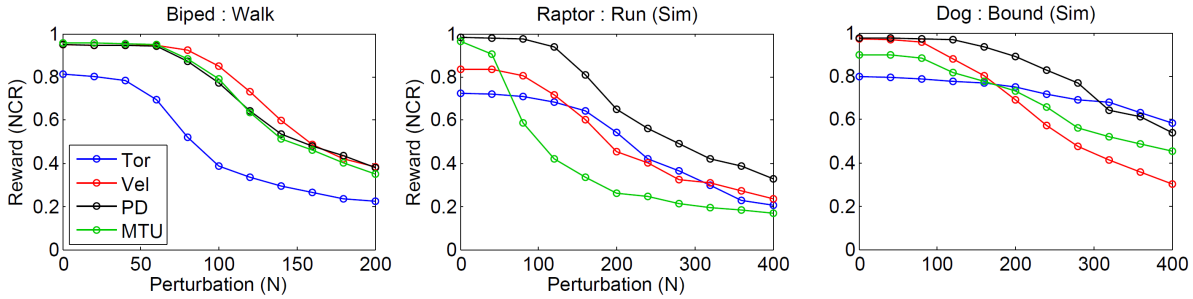


Figure 5.13: Performance when subjected to random perturbation forces of different magnitudes.

Query Rate: Figure 5.15 compares the performance of different parameterizations for different policy query rates. Separate policies are trained with queries of 15Hz, 30Hz, 60Hz, and 120Hz. Actuation models that incorporate low-level feedback such as PD and Vel, appear to cope more effectively to lower query rates, while the Tor degrades more rapidly at lower query rates. It is not yet obvious to us why MTU policies appear to perform better at lower query rates and worse at higher rates. Lastly, Figure 5.9

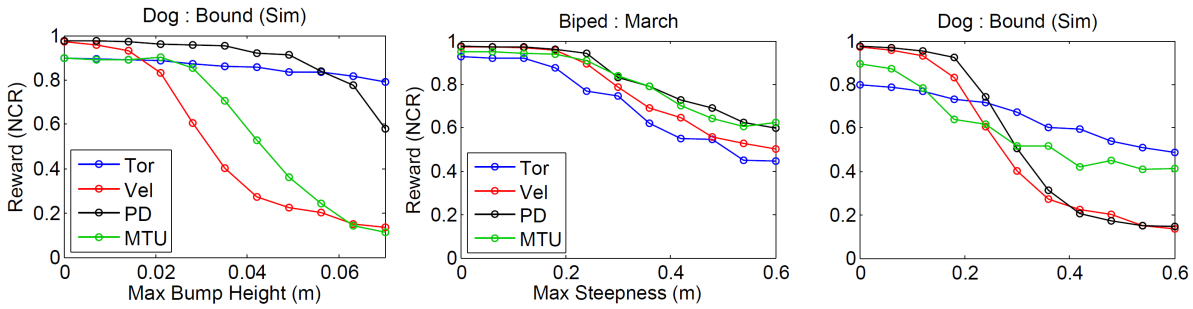


Figure 5.14: Performance of different action parameterizations when traveling across randomly generated irregular terrain. **Left:** Dog running across bumpy terrain, where the height of each bump varies uniformly between 0 and a specified maximum height. **Middle:** and **Right:** biped and dog traveling across randomly generated slopes with bounded maximum steepness.

shows the policy outputs as a function of time for the four actuation models, for a particular joint, as well as showing the resulting joint torque. Interestingly, the MTU action is visibly smoother than the other actions and results in joint torques profiles that are smoother than those seen for PD and Vel.

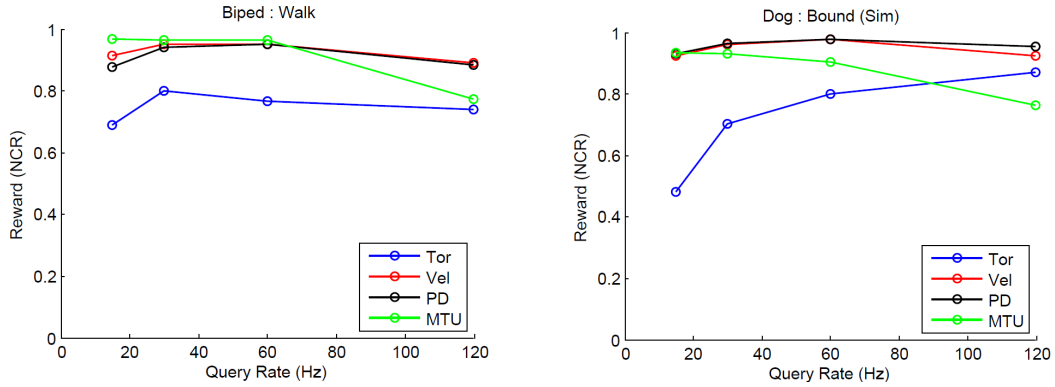


Figure 5.15: Left: Performance of policies with different query rates for the biped. **Right:** Performance for the dog. Separate policies are trained for each query rate.

5.6 Discussion

Our experiments suggest that action parameterizations that include basic local feedback, such as PD target angles, MTU activations, or target velocities, can improve policy performance and learning speed across different motions and character morphologies. Such models more accurately reflect the embodied nature of control in biomechanical systems, and the role of mechanical components in shaping the overall dynamics of motions and their control. The difference between low-level and high-level action parameterizations grow with the complexity of the characters, with high-level parameterizations scaling more gracefully to complex characters. As a caveat, there may well be tasks, such as impedance control, where lower-level action parameterizations such as Tor may prove advantageous. We believe that no

single action parameterization will be the best for all problems. However, since objectives for motion control problems are often naturally expressed in terms of kinematic properties, higher-level actions such as target joint angles and velocities may be effective for a wide variety of motion control problems. We hope that our work will help open discussions around the choice of action parameterizations.

Our results have only been demonstrated on planar articulated figure simulations; the extension to 3D currently remains as future work. Furthermore, our current torque limits are still large as compared to what might be physically realizable. Tuning actuator parameters for complex actuation models such as MTUs remains challenging. Though our actuator optimization technique is able to improve performance as compared to manual tuning, the resulting parameters may still not be optimal for the desired task. Therefore, our comparisons of MTUs to other action parameterizations may not be reflective of the full potential of MTUs with more optimal actuator parameters. Furthermore, our actuator optimization currently tunes parameters for a specific motion, rather than a larger suite of motions, as might be expected in nature.

Since the reward terms are mainly expressed in terms of positions and velocities, it may seem that it is inherently biased in favour of PD and Vel. However, the real challenges for the control policies lie elsewhere, such as learning to compensate for gravity and ground-reaction forces, and learning foot-placement strategies that are needed to maintain balance for the locomotion gaits. The reference pose terms provide little information on how to achieve these hidden aspects of motion control that will ultimately determine the success of the locomotion policy. While we have yet to provide a concrete answer for the generalization of our results to different reward functions, we believe that the choice of action parameterization is a design decision that deserves greater attention regardless of the choice of reward function.

Finally, it is reasonable to expect that evolutionary processes would result in the effective co-design of actuation mechanics and control capabilities. Developing optimization and learning algorithms to allow for this kind of co-design is a fascinating possibility for future work.

Chapter 6

Hierarchical Locomotion Skills

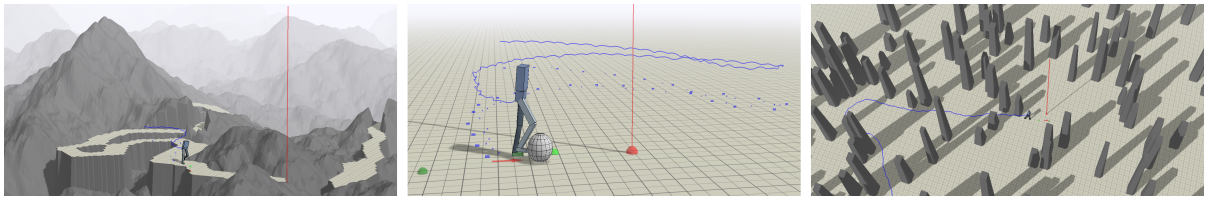


Figure 6.1: Locomotion skills learned using hierarchical reinforcement learning. (a) Following a varying-width winding path. (b) Dribbling a soccer ball. (c) Navigating through obstacles.

In this chapter we aim to learn a variety of environment-aware locomotion skills with a limited amount of prior knowledge. We adopt a two-level hierarchical control framework. First, low-level controllers are learned that operate at a fine timescale and which achieve robust walking gaits that satisfy stepping-target and style objectives. Second, high-level controllers are then learned which plan at the timescale of steps by invoking desired step targets for the low-level controller. The high-level controller makes decisions directly based on high-dimensional inputs, including terrain maps or other suitable representations of the surroundings. Both levels of the control policy are trained using deep reinforcement learning. Results are demonstrated on a simulated 3D biped. Low-level controllers are learned for a variety of motion styles and demonstrate robustness with respect to force-based disturbances, terrain variations, and style interpolation. High-level controllers are demonstrated that are capable of following trails through terrains, dribbling a soccer ball towards a target location, and navigating through static or dynamic obstacles.

6.1 Introduction

Physics-based simulations of human skills and human movement have long been a promising avenue for character animation, but it has been difficult to develop the needed control strategies. While the learning of robust balanced locomotion is a challenge by itself, further complexities are added when the

locomotion needs to be used in support of tasks such as dribbling a soccer ball or navigating among moving obstacles. Hierarchical control is a natural approach towards solving such problems. A low-level controller (LLC) is desired at a fine timescale, where the goal is predominately about balance and limb control. At a larger timescale, a high-level controller (HLC) is more suitable for guiding the movement to achieve longer-term goals, such as anticipating the best path through obstacles. In this paper, we leverage the capabilities of deep reinforcement learning (RL) to learn control policies at both timescales. The use of deep RL allows skills to be defined via objective functions, while enabling for control policies based on high-dimensional inputs, such as local terrain maps or other abundant sensory information. The use of a hierarchy enables a given low-level controller to be reused in support of multiple high-level tasks. It also enables high-level controllers to be reused with different low-level controllers.

Our principal contribution is to demonstrate that environment-aware 3D bipedal locomotion skills can be learned with a limited amount of prior structure being imposed on the control policy. In support of this, we introduce the use of a two-level hierarchy for deep reinforcement learning of locomotion skills, with both levels of the hierarchy using an identical style of actor-critic algorithm. To the best of our knowledge, we demonstrate some of the most capable dynamic 3D walking skills for model-free learning-based methods, i.e., methods that have no direct knowledge of the equations of motion, character kinematics, or even basic abstract features such as the center of mass, and no *a priori* control-specific feedback structure. Our method comes with its own limitations, which we also discuss.

6.2 Related Work

Learning of high-level controllers for physics-based characters has been successfully demonstrated for several locomotion and obstacle-avoidance tasks [Coros et al., 2009, Peng et al., 2015, 2016]. Alternatively, planning using a learned high-level dynamics model has also been proposed for locomotion tasks [Coros et al., 2008]. However, the low-level controllers for these learned policies are still designed with significant human insight, and the work presented in Chapter 4 are demonstrated only for planar motions.

Motion planning is a well-studied problem, which typically investigates how characters or robots should move in constrained environments. For wheeled robots, such problem can usually be reduced to finding a path for a point robot [Kavraki et al., 1996]. Motion planning for legged robots is significantly more challenging due to the increased degrees of freedom and tight coupling with the underlying locomotion dynamics. When quadrupeds are equipped with robust mobility control, a classic A^* path planner can be used to compute steering and forward speed commands to the locomotion controller to navigate in real-world environment with high success [Wooden et al., 2010]. However, skilled balanced motions are more difficult to achieve for bipeds and thus they are harder to plan and control [Kuffner et al., 2005]. Much of the work in robotics emphasizes footstep planning, e.g., [Chestnutt et al., 2005], with some work on full-body motion generation, e.g., [Grey et al., 2016]. Possibility graphs are pro-

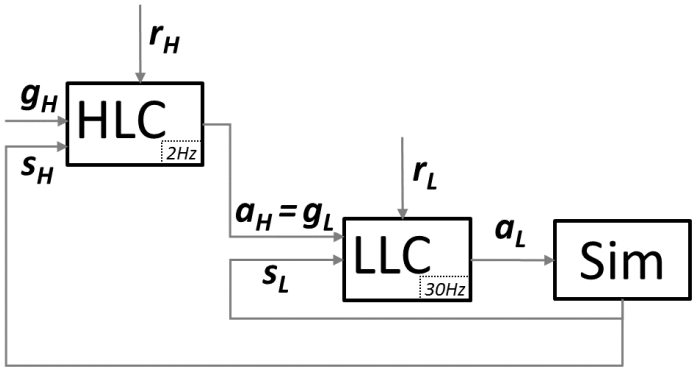


Figure 6.2: System overview

posed [Grey et al., 2016] to use high-level approximations of constraint manifolds to rapidly explore the possibility of actions, thereby allowing lower-level motion planners to be utilized more efficiently. Our hierarchical planning framework and the step targets produced by the HLC are partly inspired by this previous work from humanoid robotics.

Motion planning in support of character animation has been studied for manipulation tasks [Yamane et al., 2004, Bai et al., 2012] as well as full-body behaviours. The full-body behaviour planners often work with kinematic motion examples [Pettr et al., 2003, Lee and Lee, 2004, Lau and Kuffner, 2005]. Planning for physics-based characters is often achieved with the help of abstract dynamic models in low-dimensional spaces [Mordatch et al., 2010, Ye and Liu, 2010]. A hybrid approach is adopted in [Liu et al., 2012] where a high-level kinematic planner directs the low-level dynamic control of specific motion skills.

6.3 Overview

An overview of the DeepLoco system is shown in Figure 6.2. The system is partitioned into two components that operate at different timescales. The high-level controller (HLC) operates at a coarse timescale of 2 Hz, the timescale of walking steps, while the low-level controller (LLC) operates at 30 Hz, the timescale of low-level control actions such as PD target angles. Finally, the physics simulation is performed at 3 kHz. Together, the HLC and LLC form a two-level control hierarchy where the HLC processes the high-level task goals g_H and provides the LLC with low-level intermediate goals g_L that direct the character towards fulfilling the overall task objectives. When provided with an intermediate goal from the HLC, the LLC coordinates the motion of the character's various joints in order to fulfill the intermediate goals. This hierarchical partitioning of control allows the controllers to explore behaviours spanning different spatial and temporal abstractions, thereby enabling more efficient exploration of task-relevant strategies.

The inputs to the HLC consist of the state, s_H, and the high-level goal, g_H, as specified by the task. It outputs an action, a_H, which then serves as the current goal g_L for the LLC. s_H provides both

proprioceptive information of the character's configuration as well as exteroceptive information about its environment. In our framework, the high level action, a_H, consists of a footstep plan for the LLC.

The LLC receives the state, s_L, and an intermediate goal, g_L, as specified by the HLC, and outputs an action a_L. Unlike the high-level state s_H, s_L consists mainly of proprioceptive information describing the state of the character. The low-level action a_L specifies target angles for PD controllers positioned at each joint, which in turn compute torques that drive the motion of the character.

The actions from the LLC are applied to the simulation, which in turn produces updated states s_H and s_L by extracting the relevant features for the HLC and LLC respectively. The environment then also provides separate reward signals r_H and r_L to the HLC and LLC, reflecting progress towards their respective goals g_H and g_L. Both controllers are trained with a common actor-critic learning algorithm. The policy (actor) is trained using a positive-temporal difference update scheme modeled after CACLA [Van Hasselt, 2012], and the value function (critic) is trained using Bellman backups.

6.4 Policy Representation and Learning

Let $\pi(s,g) : S \times G \rightarrow A$ represent a deterministic policy, which maps a state $s \in S$ and goal $g \in G$ to an action $a \in A$, while a stochastic policy $\pi(s,g,a) : S \times G \times A \rightarrow \mathbb{R}$ represents the conditional probability distribution of a given s and g, $\pi(s,g,a) = p(a|s,g)$. For a particular s and g, the action distribution is modeled by a Gaussian $\pi(s,g,a) = G(\mu(s,g),\Sigma)$, with a parameterized mean $\mu(s,g)$ and fixed covariance matrix Σ. Each policy query in turn samples an action from the distribution according to

$$a = \mu(s,g) + N, \quad N \sim G(0,\Sigma) \tag{6.1}$$

generated by applying Gaussian noise to the mean action $\mu(s,g)$. While the covariance $\Sigma = diag(\{\sigma_i\})$ is represented by manually-specified values $\{\sigma_i\}$ for each action parameter, the mean is represented by a neural network $\mu(s,g|\theta)$ with parameters θ.

During training, a stochastic policy enables the character to explore new actions that may prove promising, but the addition of exploration noise can impact performance at runtime. Therefore, at runtime, a deterministic policy, which always selects the mean action $\pi(s,g) = \mu(s,g)$, is used instead. The choice between a stochastic and deterministic policy can be denoted by the addition of a binary indicator variable $\lambda \in \{0,1\}$

$$a = \mu(s,g) + \lambda N \tag{6.2}$$

where 1 indicates a stochastic policy with added exploration noise, and 0 a deterministic policy that always selects the mean action. During training, ε-greedy exploration can be incorporated by randomly enabling and disabling exploration noise according to a Bernoulli distribution $\lambda \sim \text{Ber}(\varepsilon)$, where ε represents the probability of action exploration by applying noise to the mean action.

The reward function $r_t = r(s_t, g_t, a_t)$ provides the agent with feedback regarding the desirability of

performing action a_t at state s_t given goal g_t. The reward function is therefore an interface through which users can shape the behaviour of the agent by assigning higher rewards to desirable behaviours, and lower rewards to less desirable ones. The policies are trained using an actor-critic framework to maximize the expected cumulative reward. Algorithm 7 illustrates the common learning algorithm for both the LLC and HLC. For the purpose of learning, the character's experiences are summarized by tuples $\tau_i = (s_i, g_i, a_i, r_i, s'_i, \lambda_i)$, recording the start state, goal, action, reward, next state, and application of exploration noise for each action performed by the character. The tuples are stored in an experience replay memory D and used to update the policy. Each policy is trained using an actor-critic framework, where a policy $\pi(s, g, a|\theta_\pi)$ and value function $V(s, g|\theta_V)$ are learned in tandem. The value function is trained to predict the expected cumulative reward of following the policy starting at a given state s and goal g. To update the value function, a minibatch of n tuples $\{\tau_i\}$ are sampled from D and used to perform a Bellman backup

$$y_i \leftarrow r_i + \gamma V(s'_i, g_i|\theta_V) \tag{6.3}$$

$$\theta_V \leftarrow \theta_V + \alpha_v \left(\frac{1}{n} \sum_i \nabla_{\theta_V} V(s_i, g_i|\theta_V)(y_i - V(s_i, g_i|\theta_V)) \right) \tag{6.4}$$

The learned value function is then used to update the policy. Policy improvement is performed using a CACLA-style positive temporal difference update [Van Hasselt, 2012]. Since the policy gradient as defined above is for stochastic policies, policy updates are performed using only tuples with added exploration noise (i.e. $\lambda_i = 1$).

$$\delta_i \leftarrow r_i + \gamma V(s'_i, g_i|\theta_V) - V(s_i, g_i|\theta_V) \tag{6.5}$$

$$\begin{aligned} &\text{if } \delta_i > 0 : \\ &\theta_\pi \leftarrow \theta_\pi + \alpha_\mu \left(\frac{1}{n} \nabla_{\theta_\pi} \mu(s_i, g_i|\theta_\pi) \Sigma^{-1} (a_i - \mu(s_i, g_i|\theta_\pi)) \right) \end{aligned} \tag{6.6}$$

Equation 6.6 can be interpreted as a stochastic gradient ascent step along an estimate of the policy gradient for a Gaussian policy.

6.5 Low-Level Controller

The low-level controller LLC is responsible for coordinating joint torques to mimic the overall style of a reference motion while satisfying footstep goals and maintaining balance. The reference motion is represented by keyframes that specify target poses at each timestep t. The LLC is queried at 30 Hz , where each query provides as input the state s_L, representing the character state, and goal g_L, representing a footstep plan. The LLC then produces an action a_L specifying PD target angles for every joint, relative to their parent link.

Algorithm 7 Actor-Critic Algorithm Using Positive Temporal Difference Updates

$\theta_\pi \leftarrow$ random weights
$\theta_V \leftarrow$ random weights
while not done **do**
  for $step = 1, ..., m$ **do**
    $s \leftarrow$ start state
    $g \leftarrow$ goal
    $\lambda \leftarrow \text{Ber}(\varepsilon_t)$
    $a \leftarrow \mu(s, g|\theta_\pi) + \lambda N, \quad N \sim G(0, \Sigma)$
    Apply a and simulate forward one step
    $s' \leftarrow$ end state
    $r \leftarrow$ reward
    $\tau \leftarrow (s, g, a, r, s', \lambda)$
    store τ in D
  end for

  Update value function:
  Sample minibatch of n tuples $\{\tau_i = (s_i, g_i, a_i, r_i, s'_i, \lambda_i)\}$ from D
  for each τ_i **do**
    $y_i \leftarrow r_i + \gamma V(s'_i, g_i|\theta_V) - V(s_i, g_i|\theta_V)$
  end for
  $\theta_V \leftarrow \theta_V + \alpha_v \left(\frac{1}{n} \sum_i \nabla_{\theta_V} V(s_i, g_i|\theta_V)(y_i - V(s_i, g_i|\theta_V))\right)$

  Update policy:
  Sample minibatch of n tuples $\{\tau_j = (s_j, g_j, a_j, r_j, s'_j, \lambda_j)\}$ from D where $\lambda_j = 1$
  for each τ_j **do**
    $\delta_j \leftarrow r_j + \gamma V(s'_j, g_j|\theta_V) - V(s_j, g_j|\theta_V)$
    if $\delta_j > 0$ **then**
      $\triangle a_j \leftarrow a_j - \mu(s_j, g_j|\theta_\pi)$
      $\theta_\pi \leftarrow \theta_\pi + \alpha_\mu \left(\frac{1}{n} \nabla_{\theta_\pi} \mu(s_j, g_j|\theta_\pi) \Sigma^{-1} \triangle a_j\right)$
    end if
  end for
end while

LLC State: The LLC input state s_L, shown in Figure 6.3 (left), consists mainly of features describing the character's configuration. These features include the center of mass positions of each link relative to the character's root, designated as the pelvis, their relative rotations with respect to the root expressed as quaternions, and their linear and angular velocities. Two binary indicator features are included, corresponding to the character's feet. The features are assigned 1 when their respective foot is in contact with the ground and 0 otherwise. A phase variable $\phi \in [0, 1]$ is also included as an input, which indicates the phase along a walk cycle. Each walk cycle has a fixed period of 1 s, corresponding to 0.5 s per step. The phase variable advances at a fixed rate and helps keep the LLC in sync with the reference motion. Combined, these features create a 110D state space.

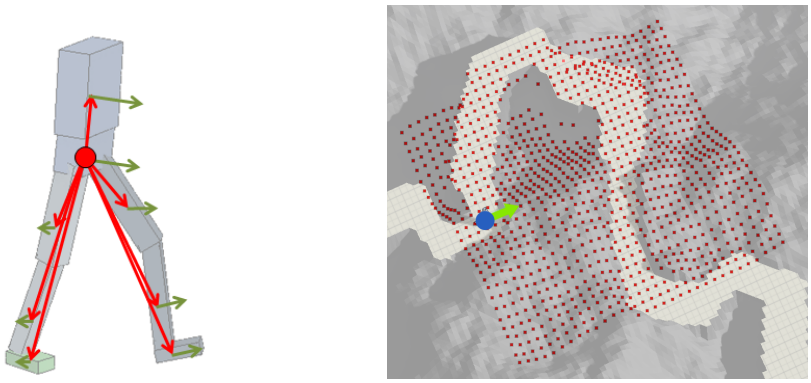


Figure 6.3: left: The character state features consist of the positions of each link relative to the root **(red arrows)**, their rotations, linear velocities **(green arrows)**, and angular velocities. **right:** The terrain features consist of a 2D heightmap of the terrain sampled on a regular grid. All heights are expressed relative to height of the ground immediately under the root of the character. The heightmap has a resolution of 32x32 and occupies an area of approximately 11x11m.

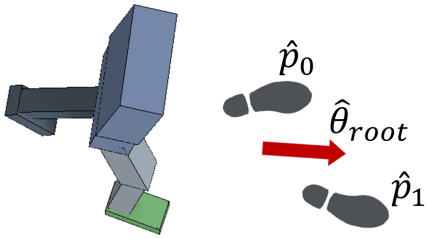


Figure 6.4: The goal g_L for the LLC is represented as a footstep plan, specifying the target positions $\hat{p}_0$ and $\hat{p}_1$ for the next two steps, and the target heading for the root $\hat{\theta}_{root}$.

LLC Goal: Each footstep plan $g_L = (\hat{p}_0, \hat{p}_1, \hat{\theta}_{root})$, as shown in Figure 6.4, specifies the 2D target position $\hat{p}_0$ relative to the character on the horizontal plane for the swing foot at the end of the next step, as well as the target location for the following step $\hat{p}_1$. This is motivated by work showing that "two steps is enough" [Zaytsev et al., 2015]. In addition to target step positions, the footstep plan also provides a desired heading $\hat{\theta}_{root}$ for the root of the character for the immediate next step.

LLC Action: The action a_L produced by the LLC specifies target positions for PD controllers positioned at each joint. The target joint positions are represented in 4 dimensional axis-angle form, with axis normalization occurring when applying the actions, i.e., the output action from the network need not be normalized. This yields a 22D action space.

6.5.1 Reference Motion

A reference motion (or set of motions) serves to help specify the desired walking style, while also helping to guide the learning. The reference motion can be a single manually keyframed motion cycle, or one or more motion capture clips. The goal for the LLC is to mimic the overall style of the reference motion rather than precisely tracking it. The reference motion will generally not satisfy the desired footstep goals, and is often not physically realizable in any case because of the approximate nature of a hand-animated walk cycle, or model mismatches in the case of a motion capture clip. At each timestep t a reference motion provides a reference pose $\hat{q}(t)$ and reference velocity $\hat{\dot{q}}(t)$, computed via finite-differences $\hat{\dot{q}}(t) \approx \frac{\hat{q}(t+\triangle t)-\hat{q}(t)}{\triangle t}$. The use of multiple reference motion clips can help produce better turning behaviors, as best seen in the supplemental video. To make use of multiple reference motions, we construct a kinematic controller $\hat{q}^*(\cdot) \leftarrow K(s, g_L)$, when given the simulated character state s and a footstep plan g_L, selects the appropriate motion from a small set of motion clips that best realizes the footstep plan g_L. To construct the set of reference motions for the kinematic controller, we segmented 7 s of motion capture data of walking and turning motions into individual clips $\hat{q}^j(\cdot)$, each corresponding to a single step. A step begins on the stance foot heel-strike and ends on the swing foot heel-strike. Each clip is preprocessed to be in right stance with a step duration of 0.5 s. During training, the reference motions are mirrored as necessary to be consistent with the simulated character's stance leg. A vector of features $\Lambda(\hat{q}^j(\cdot)) = (p_{stance}, p_{swing}, \theta_{root})$ are then extracted for each clip and later used to select the appropriate clip for a given query. The features include the stance foot position p_{stance} at the start of a clip, the swing foot position p_{swing} at the end of the clip, and the root orientation θ_{root} on the horizontal plane at the end of the clip.

During training, $K(s, g_L)$ is queried at the beginning of each step to select the reference clip for the upcoming step. To select among the motion clips, a similar set of features $\Lambda(s, g_L)$ are extracted from s and g_L, where p_{stance} is specified by the stance foot position from the simulated character state s, p_{swing} and θ_{root} are specified by the target footstep position $\hat{p}_0$ and root orientation $\hat{\theta}_{root}$ from g_L. The most suitable clip is then selected according to:

$$K(s, g_L) = \arg\min_{\hat{q}^j(\cdot)} ||\Lambda(s, g_L) - \Lambda(\hat{q}^j(\cdot))|| \tag{6.7}$$

The selected clip then acts as the reference motion to shape the reward function for the LLC over the course of the upcoming step.

6.5.2 LLC Reward

Given the reference pose $\hat{q}(t)$ and velocity $\hat{\dot{q}}(t)$ the LLC reward r_L is defined as a weighted sum of objectives that encourage the character to imitate the style of the reference motion while following the

footstep plan,

$$r_L = w_{pose} r_{pose} + w_{vel} r_{vel} + w_{root} r_{root} + w_{com} r_{com} + w_{end} r_{end} + w_{heading} r_{heading} \tag{6.8}$$

using $(w_{pose}, w_{vel}, w_{root}, w_{com}, w_{end}, w_{heading}) = (0.5, 0.05, 0.1, 0.1, 0.2, 0.1)$. r_{pose}, r_{vel}, r_{root}, and r_{com} encourages the policy to reproduce the given reference motion, while r_{end} and $r_{heading}$ encourages it to follow the footstep plan.

$$r_{pose} = \exp\left(-\sum_i w_i d(\hat{q}_i(t), q_i)^2\right)$$

$$r_{vel} = \exp\left(-\sum_i w_i ||\hat{\dot{q}}_i(t) - \dot{q}_i||^2\right)$$

$$r_{root} = \exp\left(-10(\hat{h}_{root} - h_{root})^2\right)$$

$$r_{com} = \exp\left(-||\hat{v}_{com} - v_{com}||^2\right)$$

$$r_{end} = \exp\left(-||\hat{p}_{swing} - p_{swing}||^2 - ||\hat{p}_{stance} - p_{stance}||^2\right)$$

$$r_{heading} = 0.5\cos\left(\hat{\theta}_{root} - \theta_{root}\right) + 0.5$$

where q_i represents the quaternion rotation of joint i and $d(\cdot,\cdot)$ computes the distance between two quaternions. w_i are manually specified weights for each joint. h_{root} represents the height of the root from the ground, v_{com} is the center of mass velocity, p_{swing} and p_{stance} are the positions of the swing and stance foot. The target position for the swing foot $\hat{p}_{swing} = \hat{p}_0$ is provided by the footstep plan, while the target position for the stance foot $\hat{p}_{stance}$ is provided by the reference motion. θ_{root} represents the heading of the root on the horizontal plane, and $\hat{\theta}_{root}$ is the desired heading provided by the footstep plan. We choose to keep rewards constrained to $r \in [0, 1]$.

6.5.3 Bilinear Phase Transform

While the phase variable ϕ helps to keep the LLC in sync with the reference motion, in our experiments this did not appear to be sufficient for the network to clearly distinguish the different phases of a walk, often resulting in foot-dragging artifacts. To help the network better distinguish between different phases

of a motion, we take inspiration from bilinear pooling models for vision tasks [Fukui et al., 2016]. From the scalar phase variable ϕ we construct a tile-coding $\Phi = (\Phi_0, \Phi_1, \Phi_2, \Phi_3)^T$, where $\Phi_i \in \{0, 1\}$ is 1 if ϕ lies within its phase interval and 0 otherwise. For example, $\Phi_0 = 1$ iff $0 \leq \phi < 0.25$, and $\Phi_1 = 1$ iff $0.25 \leq \phi < 0.5$, etc. Given the original input vector (s_L, g_L), the bilinear phase transform computes the outer product

$$\begin{pmatrix} s_L \\ g_L \end{pmatrix} \Phi^T = \left[\Phi_0 \begin{pmatrix} s_L \\ g_L \end{pmatrix}, \Phi_1 \begin{pmatrix} s_L \\ g_L \end{pmatrix}, \Phi_2 \begin{pmatrix} s_L \\ g_L \end{pmatrix}, \Phi_2 \begin{pmatrix} s_L \\ g_L \end{pmatrix} \right] \tag{6.9}$$

which is then processed by successive layers of the network. This representation results in a feature set where only a sparse subset of the features, corresponding to the current phase interval, are active at a given time. This effectively encodes a prior into the network that different behaviours are expected at different phases of the motion. Note that the scalar phase variable ϕ is still included in s_L to allow the LLC to track its progress within each phase interval.

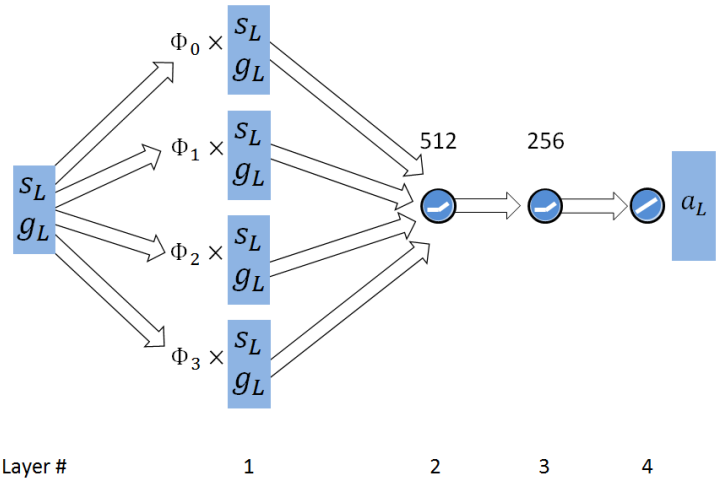


Figure 6.5: Schematic illustration of the LLC network. The input consists of the state s_L and goal g_L. The first layer applies the bilinear phase transform and the resulting features are processed by a series of fully-connected layers. The output layer produces the action a_L, which specifies PD targets for each joint.

6.5.4 LLC Network

A schematic diagram of the LLC network is shown in Figure 6.5. The LLC is represented by a 4-layered neural network that receives as input s_L and g_L, and outputs the mean $\mu(s_L, g_L)$ of the action distribution. The first layer applies the bilinear phase transform to the inputs, and the resulting bilinear features are then processed by two fully-connected layers with 512 and 256 units each. ReLU activation functions are applied to both hidden layers [Nair and Hinton, 2010]. Finally a linear output layer computes the mean action. The LLC value function $V_L(s_L, g_L)$ is modeled by a similar network, but with a single linear unit in the output layer. Each LLC network has approximately 500k parameters.

6.5.5 LLC Training

LLC training proceeds episodically where the character is initialized to a default pose at the beginning of each episode. An episode is simulated for a maximum of 200 s but is terminated early if the character falls, leaving the character with 0 reward for the remainder of the episode. A fall is detected when the torso of the character makes contact with the ground. At the beginning of each walking step, a new footstep plan $g_L^k = (\hat{p}_0^k, \hat{p}_1^k, \hat{\theta}_{root}^k)$ is generated by randomly adjusting the previous plan g_L^{k-1} according to

$$\begin{aligned} \hat{p}_0^k &= \hat{p}_1^{k-1} \\ \hat{\theta}_{root}^k &= \hat{\theta}_{root}^{k-1} + N, \quad N \sim G(0, 0.25^2) \\ \hat{p}_1^k &= \hat{p}_0^k + \triangle p(\hat{\theta}_{root}^k) \end{aligned} \tag{6.10}$$

where $\triangle p(\hat{\theta}_{root}^k)$ advances the step position along the heading direction $\hat{\theta}_{root}^k$ by a fixed step length of 0.4 m to obtain a new target step position.

After a footstep plan has been determined for the new step, the kinematic controller $K(s_L, g_L)$ is queried for a new reference motion. The reference motion $\hat{q}(\cdot)$ is then used by the reward function for the duration of the step, which guides the LLC towards a stepping motion that approximately achieves the desired footstep goal g_L.

6.5.6 Style Modification

In addition to imitating a reference motion, the LLC can also be stylized by simple modifications to the reward function. In the following examples, we consider the addition of a style term c_{style} to the pose reward r_{pose}.

$$r_{pose} = \exp\left(-\sum_i w_i d(\hat{q}_i(t), q_i)^2 - w_{style} c_{style} \right) \tag{6.11}$$

c_{style} provides an interface through which the user can shape the motion of the LLC. w_{style} is a user-specified weight that trades off between conforming to the reference motion and satisfying the desired style.

Forward/Sideways Lean: By using c_{style} to specify a desired waist orientation, the LLC can be steered towards learning a robust walk while leaning forward or sideways.

$$c_{style} = d(\hat{q}(t)_{waist}, q_{waist})^2 \tag{6.12}$$

where $\hat{q}(\cdot)_{waist}$ is a quaternion specifying the desired waist orientation.

Straight Leg(s): Similarly, c_{style} can be used to penalize bending of the knees, resulting in a locked-knee walk.

$$c_{style} = d(q_I, q_{knee})^2 \tag{6.13}$$

with q_I being the identity quaternion. Using this style term, we trained two LLC's, one with the right leg encouraged to be straight, and one with both legs straight.

High-Knees: A high-knees walk can be created by using c_{style} to encourage the character to lift its knees higher during each step,

$$c_{style} = (\hat{h}_{knee} - h_{knee})^2 \tag{6.14}$$

where $\hat{h}_{knee} = 0.8m$ is the target height for the swing knee with respect to the ground.

In-place Walk: By replacing the reference motion $\hat{q}(\cdot)$ with a single hand-authored clip of an in-place walk, the LLC can be trained to step in-place.

Separate networks are trained for each stylized LLC by bootstrapping from the nominal walk LLC. The weights of each network are initialized from those of the nominal walk, then fine-tuned using the stylized reward functions. Furthermore, we show that it is possible to interpolate different stylized LLC's while also remaining robust. Let $\pi_L^a(s_L, g_L)$ and $\pi_L^b(s_L, g_L)$ represent LLC's trained for style a and b. A new LLC $\pi_L^c(s, g)$ can be defined by linearly interpolating the outputs of the two LLC's

$$\pi_L^c(s_L, g_L) = (1-u)\pi_L^a(s_L, g_L) + u\pi_L^b(s_L, g_L) \tag{6.15}$$

$u \in [0, 1]$, allowing the character to seamlessly transition between the different styles. As shown in the results, we can also allow for moderate extrapolation.

6.6 High-level Controller

While the LLC is primarily responsible for low-level coordination of the character's limbs for locomotion, the HLC is responsible for high-level task-specific objectives such as navigation. The HLC is queried at 2 Hz, corresponding to the beginning of each step. Every query provides as input a state s_H and a task-specific goal g_H. The HLC output action a_H specifies a footstep plan g_L for the LLC. The role of the HLC is therefore to provide intermediate goals for the LLC in order to achieve the overall task objectives.

HLC State: Unlike s_L, which provides mainly proprioceptive information describing the configuration of the character, s_H includes both proprioceptive and exteroceptive information describing the character and its environment. Each state $s_H = (C, T)$, consists of a set of character features C and terrain features

T, shown in Figure 6.3 (right). C shares many of the same features as the LLC state s_L, but excludes the phase and contact features. T is represented by a 32×32 heightmap of the terrain around the character. The heightmap is sampled on a regular grid with an area of approximately 11×11 m. The samples extend 10 m in front of the character and 1 m behind. Example terrain maps are shown in Figure 6.6. The combined features result in a 1129D state space.

6.6.1 HLC Training

As with the LLC training, the character is initialized to a default pose at the start of each episode. Each episode terminates after 200 s or when the character falls. At the start of each step, the HLC is queried to sample an action a_H from the policy, which is then applied to the LLC as a footstep goal g_L. The LLC is executed for 0.5 s, the duration of one step, and an experience tuple τ is recorded for the step. Note that the weights for the LLC are frozen and only the HLC is being trained. Therefore, once trained, the same LLC can be applied to a variety of tasks by training task-specific HLC's that specify the appropriate intermediate footstep goals.

6.6.2 HLC Network

A schematic diagram of the HLC network is available in Figure 6.7. The HLC is modeled by a deep convolutional neural network that receives as input the state $s_H = (C, T)$ and task-specific goal g_H, and the output action $a_H = g_L$ specifies a footstep plan for the LLC for a single step. The terrain map T is first processed by a series of three convolutional layers, with 16 5×5 filters, 32 4×4 filters, and 32 3×3 filters respectively. The features maps from the final convolutional layer are processed by 128 fully-connected units. The resulting feature vector is concatenated with C and g_H, and processed by two additional fully-connected layers with 512 and 256 units. ReLUs are used for all hidden layers. The linear output layer produces the final action. Each HLC network has approximately 2.5 million parameters.

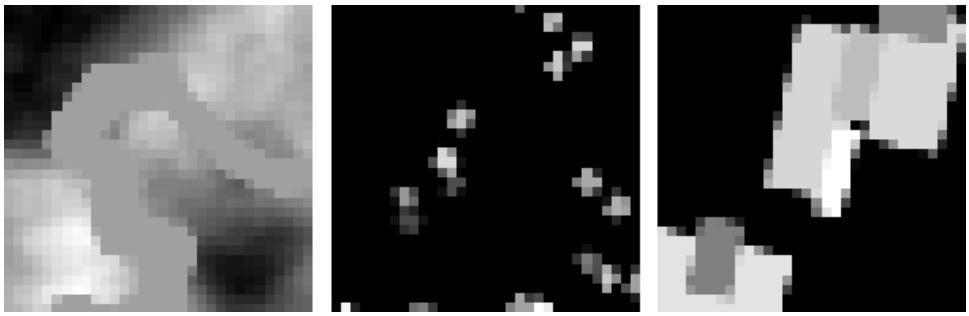


Figure 6.6: 32×32 height maps are included as input features to the HLC. Each map covers a 11×11 m area. Values in the images are normalized by the minimum and maximum height within each map. **left:** path; **middle:** pillar obstacles; **right:** block obstacles.

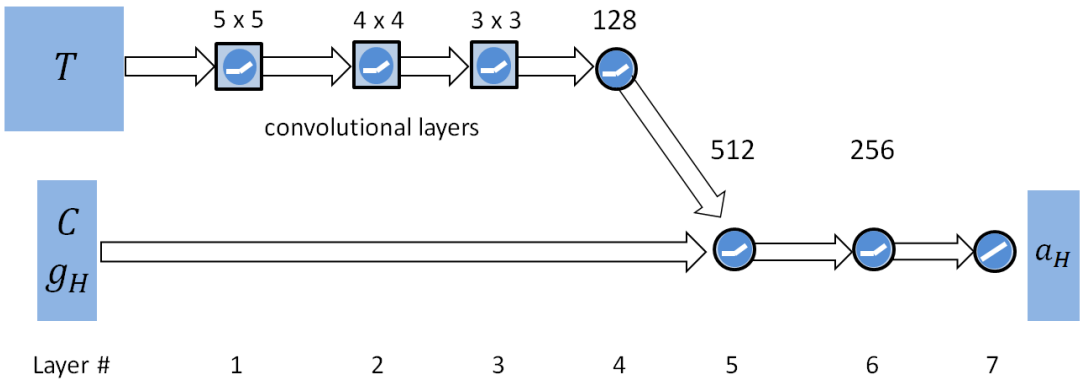


Figure 6.7: Schematic illustration of the HLC network. The input consists of a terrain map T, character features C, and goal g_H. The output action a_H specifies a footstep plan g_L for the LLC.

6.6.3 HLC Tasks

Path Following: In this task an HLC is trained to navigate narrow trails carved into rocky terrain. A target location is placed randomly along the trail, and the target advances along the trail as the character moves sufficiently close to the target. The HLC goal $g_H = (\theta_{tar}, d_{tar})$ is represented by the direction to the target θ_{tar} relative to the character's facing direction, and the distance d_{tar} to the target on the horizontal plane. The path and terrain are randomly generated, with the path width varying between 0.5 m and 2 m. Since the policy is not provided with an explicit parameterization of the path as input, it must learn to recognize the path from the terrain map T and plan its footsteps accordingly.

The reward for this task is designed to encourage the character to move towards the target at a desired speed.

$$r_H = \exp\left(-\left(\min(0, u_{tar}^T v_{com} - \hat{v}_{com})\right)^2\right) \tag{6.16}$$

where v_{com} is the agent's centre of mass velocity on the horizontal plane, and u_{tar} is a unit vector on the horizontal plane pointing towards the target. $\hat{v}_{com} = 1\ m/s$ specifies the desired speed at which the character should move towards the target.

Soccer Dribbling: Dribbling is a challenging task requiring both high-level and low-level planning. The objective is to move a ball to a target location, where the initial ball and target locations are randomly set at the beginning of each episode. The ball has a radius of 0.2 m and a mass of 0.1 kg. Having to learn a proper sequence of sub-tasks in the correct order makes this task particularly challenging. The agent must first move to the ball, and once it has possession of the ball, dribble the ball towards the target. When the ball has arrived at the target, the agent must then learn to stop moving the ball to avoid kicking the ball past the target. Since the policy does not have direct control over the ball, it must rely on complex contact dynamics in order to manipulate the ball. Furthermore, considering the LLC was not trained with

motion data comparable to dribbling, the HLC has to learn to provide the appropriate footstep plans in order to elicit the necessary LLC behaviour. The goal $g_H = (\theta_{tar}, d_{tar}, \theta_{ball}, d_{ball}, h_{ball}, v_{ball}, \omega_{ball})$ consists of the target direction relative to the ball θ_{tar}, distance between the target and ball d_{tar}, ball direction relative to the agent's root θ_{ball}, distance between the ball and the agent d_{ball}, height of the ball's center of mass from the ground h_{ball}, the ball's linear velocity v_{ball}, and angular velocity ω_{ball}. Because the dribbling task occurs on a flat plane, the terrain map T is excluded from the HLC inputs, and the convolutional layers are removed from the network.

The reward for the soccer task consists of a weighted sum of terms which encourages the agent to move towards the ball r_{cv}, stay close to the ball r_{cp}, move the ball towards the target r_{bv}, and keep the ball close to the target r_{bp}.

$$r_H = w_{cv} r_{cv} + w_{cp} r_{cp} + w_{bv} r_{bv} + w_{bp} r_{bp} \tag{6.17}$$

$$r_{cv} = \exp\left(-\left(\min(0, u_{ball}^T v_{com} - \hat{v}_{com})\right)^2\right)$$

$$r_{cp} = \exp\left(-d_{ball}^2\right)$$

$$r_{bv} = \exp\left(-\left(\min(0, u_{tar}^T v_{ball} - \hat{v}_{ball})\right)^2\right)$$

$$r_{bp} = \exp\left(-d_{tar}^2\right)$$

with weights $(w_{cv}, w_{cp}, w_{bv}, w_{bp}) = (0.17, 0.17, 0.33, 0.33)$. u_{ball} is a unit vector pointing in the direction from the character to the ball, v_{com} the character's center of mass velocity, and $\hat{v}_{com} = 1m/s$ the desired speed with which the character should move towards the ball. Similarly, u_{tar} represents the unit vector pointing from the ball to the target position, v_{ball} the velocity of the ball, and $\hat{v}_{ball} = 1m/s$ the desired speed for the ball with which to move towards the target. Once the ball is within 0.5 m of the target and the character is within 2 m of the ball, then the goal is considered fulfilled and the character receives a constant reward of 1 from all terms, corresponding to the maximum possible reward.

Pillar Obstacles: A more common task is to traverse a reasonably dense area of static obstacles. Similar to the path following task, the objective is to reach a randomly placed target location. However, unlike the path following task, there exists many possible paths to reach the target. The HLC is therefore responsible for planning and steering the agent along a particular path. When the agent reaches the target, the target location is randomly changed. The base of each obstacle measures 0.75×0.75 m, with height varying between 2 m and 8 m. Each environment instance is generated by randomly placing

obstacles at the start of each episode. The goal g_H and reward function are the same as those used for the path following task.

Block Obstacles: This environment is a variant of the pillar obstacles environment, where the obstacles consist of large blocks with side lengths varying between 0.5 m and 7 m. The policy therefore must learn to navigate around large obstacles to find paths leading to the target location.

Dynamic Obstacles: In this task, the objective is to navigate across a dynamically changing environment in order to reach a target location. The environment is populated with obstacles moving at fixed velocities back and forth along randomly oriented linear paths. The velocities vary from 0.2 m/s to 1.3 m/s, with the agent's maximum velocity being approximately 1 m/s. Given the presence of dynamic obstacles, rather than using a heightfield as input, the policy is provided with a velocity-map. The environment is sampled for moving obstacles where each sample records the 2D velocity along the horizontal plane if a sample overlaps with an obstacle. If a sample point does not contain an obstacle, then the velocity is recorded as 0. The goal features and reward function are identical to those used in the path following task. This example should not be confused with a multiagent simulation because the moving obstacles themselves are not reactive.

6.7 Results

The motions from the policies are best seen in the supplemental videos. We learn locomotion skills for a 3D biped, as modeled by eight links: three links for each leg and two links for the torso. The biped is 1.6 m tall and has a mass of 42 kg. The knee joints have one degree of freedom (DOF) and all other joints are spherical, i.e., three DOFs. We use a ground friction of $\mu = 0.9$. The character's motion is driven by internal joint torques from stable PD controllers [Tan et al., 2011] and simulated using the Bullet physics engine [Bullet, 2015] at 3000 Hz. The k_p gains for the PD controllers are (1000, 300, 300, 100) Nm/rad for the (waist, hip, knee, ankle), respectively. Derivative gains are specified as $k_d = 0.1k_p$. Torque limits are (200, 200, 150, 90) Nm, respectively. Joint limits are also in effect for all joints. All neural networks are built and trained with Caffe [Jia et al., 2014a]. The values of the input states and output actions of the networks are normalized to range approximately between [-1, 1] using manually-specified offsets and scales. The output of the value network is normalized to be between [0, 1] by multiplying the cumulative reward by $(1-\gamma)$. This normalization helps to ensure reasonable gradient magnitudes during backpropagation. Once trained, all results run faster than real-time.

LLC reference motions: We train controllers using a single planar keyframed motion cycle as a motion style to imitate, as well as a set of ten motion capture steps that correspond to approximately 7 s of data from a single human subject. The clips consist of walking motions with different turning rates. The

character was designed to have similar measurements to those of the human subject. By default, we use the results based on the motion capture styles, as they allow for sharper turns and produce a moderate improvement in motion quality. Please see the supplementary video for a direct comparison.

Hyperparameter settings: Both LLC and HLC training share similar hyperparameter settings. Batches of $m = 32$ are collected before every update. The experience replay memory D records the 50k most recent tuples. Updates are performed by sampling minibatches of $n = 32$ tuples from D and applying stochastic gradient descent with momentum, with value function stepsize $\alpha_V = 0.01$, policy stepsize $\alpha_\mu = 0.001$, and momentum 0.9. L2 weight decay of 0.0005 is applied to the policy, but none is applied to the value function. Both the LLC and HLC use a discount factor of $\gamma = 0.95$. For the LLC, the ε-greedy exploration rate ε_t is initialized to 1 and linearly annealed to 0.2 over 1 million iterations. For the HLC, ε_t is initialized to 1 and annealed to 0.5 over 200k iterations. The LLC is trained for approximately 6 million iterations, requiring about 2 days of compute time on a 16-core cluster using a multithreaded C++ implementation. Each HLC is trained for approximately 1 million iterations, requiring about 7 days. All computations are performed on the CPU and no GPU-acceleration was leveraged.

6.7.1 LLC Performance

Figure 6.8 illustrates an LLC learning curve. Performance of intermediate policies are evaluated every 40k iterations by applying the policies for 32 episodes with a length of 20 s each. Performance is measured using the normalized cumulative reward (NCR), which is calculated as the sum of immediate rewards over an episode normalized by the minimum and maximum possible cumulative reward. No discounting is applied when calculating the NCR. A comparison between an LLC trained using 10 mocap clips and another trained using a single hand-authored forward walking motion is also available in Figure 6.8. Since the two LLC's use different reference motions, the NCR is measured using only the footstep terms $r = w_{end} r_{end} + w_{heading} r_{heading}$. A richer repertoire of reference motions leads to noticeable improvements in learning speed and final performance. Learning curves for the stylized LLC's are available in Figure 6.9. Each network is initialized using the LLC trained for the nominal walk. Performance is measured using only the style term c_{style}, measuring the LLC's conformity to the style objectives.

LLC Robustness: LLC's trained for different styles are evaluated for robustness by measuring the maximum perturbation force that each LLC can tolerate before falling. The character is first directed to walk forward, then a push is applied to the torso mid-point. Forward and sideways pushes were tested separately where each perturbation is applied for 0.25 s. The magnitude of the forces are increased in increments of 10 N until the character falls. We also evaluated the LLC's robustness to terrain variation by measuring the steepest incline and decline that each LLC can successfully travel across without falling for 20 s. The maximum force that each LLC is able to recover from, and the steepest incline

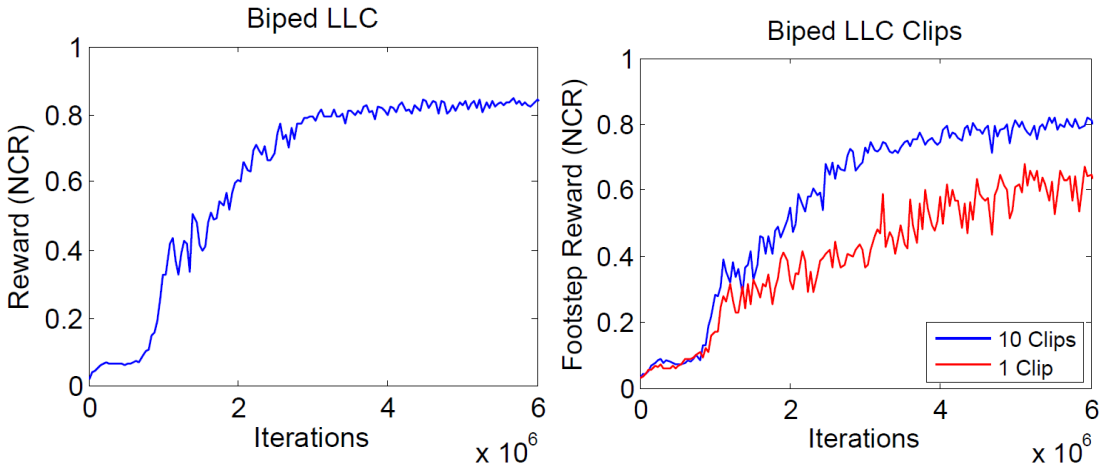


Figure 6.8: left: Learning curve for the LLC. The network is randomly initialized and trained to mimic a nominal walk while following randomly generated footstep plans. **right:** learning curves for policies trained with 10 mocap clips and 1 hand-authored clip.

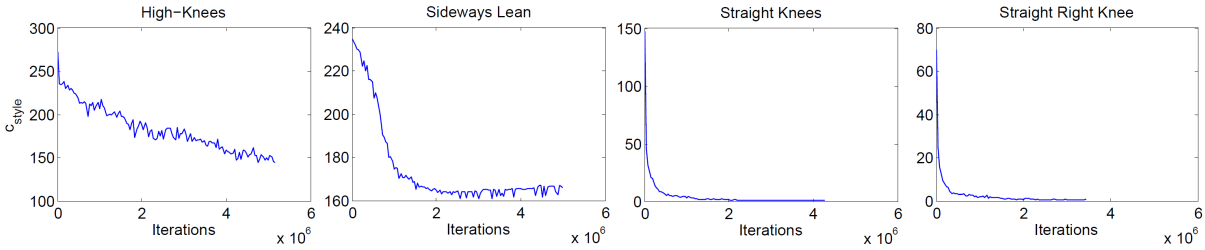


Figure 6.9: Learning curves for each stylized LLC.

and decline are summarized in Table 6.1. The nominal walk proves fairly robust to the different perturbations, while the straight leg walks are generally less robust than the other styles. Though the LLC's were trained exclusive on flat terrain, the nominal LLC is able to walk up 16% inclines without falling. After normalizing for character weight and size differences, the robustness of the nominal walk LLC is comparable to figures reported for SIMBICON, which leverages manually-crafted balance strategies [Yin et al., 2007]. The LLC's robustness likely stems from the application of exploration noise during training. The added noise perturbs the character away from its nominal trajectory, requiring it to learn recovery strategies for unexpected perturbations. We believe that robustness could be further improved by presenting the character with examples of different pushes and terrain variations during training, and by letting it anticipate pushes and upcoming terrain. We also test for robustness with respect to changes in the gait period, i.e., forcing the controller to walk with shorter or longer duration steps. The gaits are generally robust to changes in gait period of $\pm 20\%$.

To better understand the feedback strategies developed by the networks we analyze the action outputs from the nominal walk LLC for different character state configurations. Figure 6.10 illustrates the swing and stance hip target angles as a function of character's state. The state variations we consider include the waist leaning forward and backward at different angles, and pushing the root at different

LLC	Forward	Side	Incline	Decline
Nominal Walk	200N	210N	16%(9.1°)	11%(6.3°)
High-Knees	140N	190N	9%(5.1°)	5%(2.9°)
Straight Leg	150N	180N	12%(6.8°)	6%(3.4°)
Straight Legs	90N	130N	9%(5.1°)	5%(2.9°)
Forward Lean	180N	290N	10%(5.7°)	16%(9.1°)
Sideways Lean	160N	220N	7%(4.0°)	16%(9.1°)

Table 6.1: Maximum forwards and sideways push, and steepest incline and decline each LLC can tolerate before falling. Each push is applied for 0.25 *s*.

velocities. The LLC exhibits intuitive feedback strategies reminiscent of SIMBICON [Yin et al., 2007]. When the character is leaning too far forward or its forward velocity is too high, then the swing hip is raised higher to help position the swing foot further in front to regain balance in the following step, and vice-versa. but unlike SIMBICON, whose linear balance strategies are manually-crafted, the LLC develops nonlinear strategies without explicit user intervention.

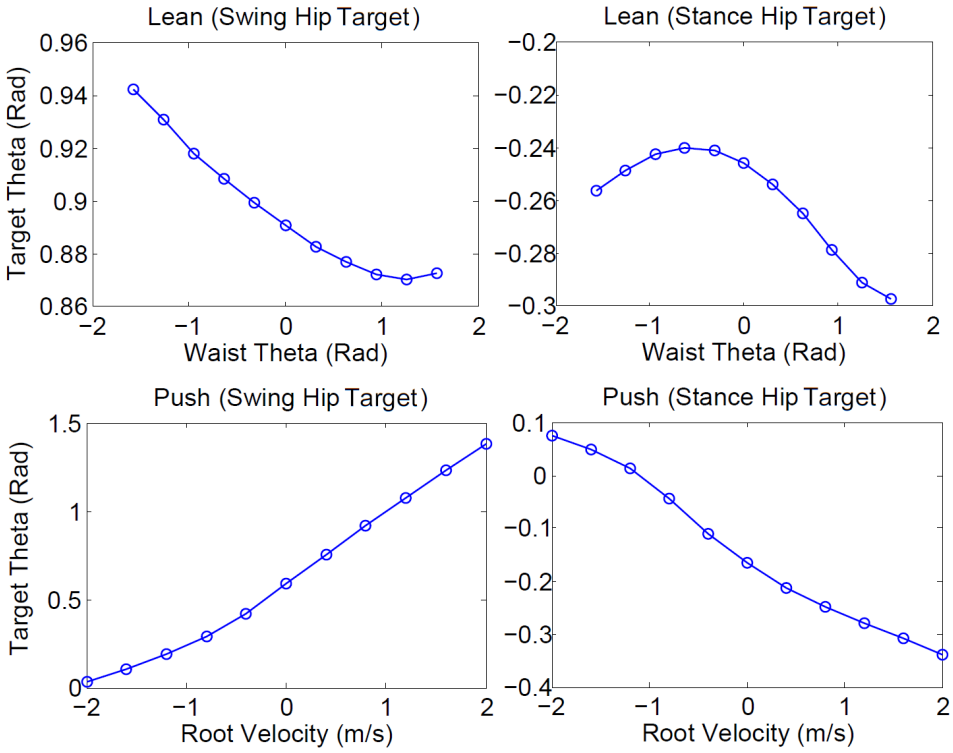


Figure 6.10: PD target angles for the swing and stance hip as a function of character state. **top:** character's waist is leaning forward at various angles, with positive theta indicates a backward lean. **bottom:** the root is given a push at different velocities.

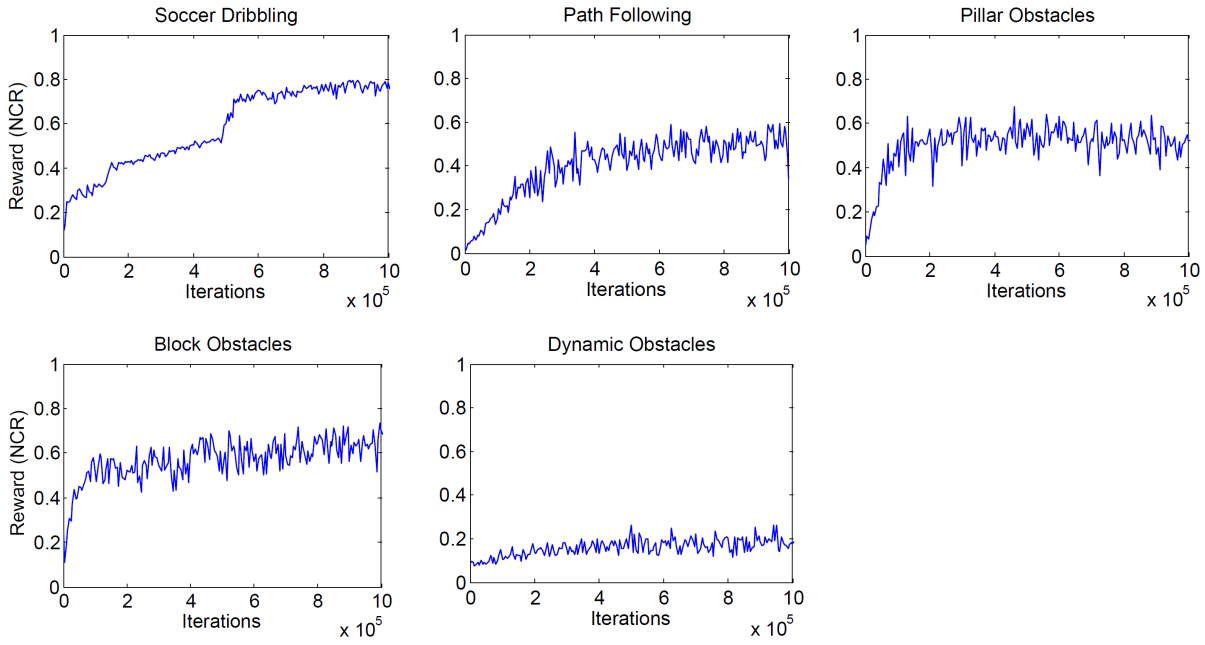


Figure 6.11: HLC learning curves.

Task	Reward (NCR)
Path Following	0.55
Soccer Dribbling	0.77
Pillar Obstacles	0.56
Block Obstacles	0.70
Dynamic Obstacles	0.18

Table 6.2: Performance summary of HLC's trained for each task. The NCR is calculated using the average of 256 episodes.

6.7.2 HLC Performance

Learning curves for HLC's trained for different tasks are available in Figure 6.11. Intermediate policy performance is evaluated every 5k iterations using 32 episodes with a length of 200 s each. Note that the maximum normalized cumulative reward, NCR $= 1$, may not always be attainable. For soccer dribbling, the maximum NCR would require instantly moving the ball to the target location. For the navigation tasks, the maximum NCR would require a straight and unobstructed path between the character and target location.

For soccer dribbling, the HLC learns to correctly sequence the required sub-tasks. The HLC first directs the character towards the ball. It then dribbles the ball towards the target. Once the ball is sufficiently close to the target, the HLC developed a strategy of circling around the ball, while maintaining some distance, to avoid perturbing the ball away from the target or tripping over the ball. Alternatively, the ball can be replaced with a box, and the HLC is able to generalize to the different dynamics without additional training. The HLC's for the path following, pillar obstacles, and block obstacles tasks

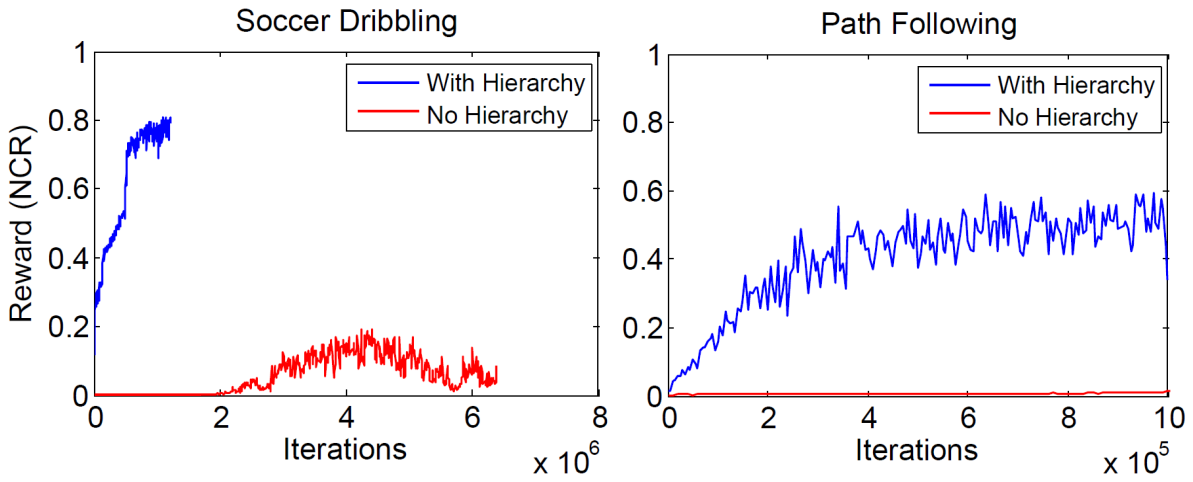
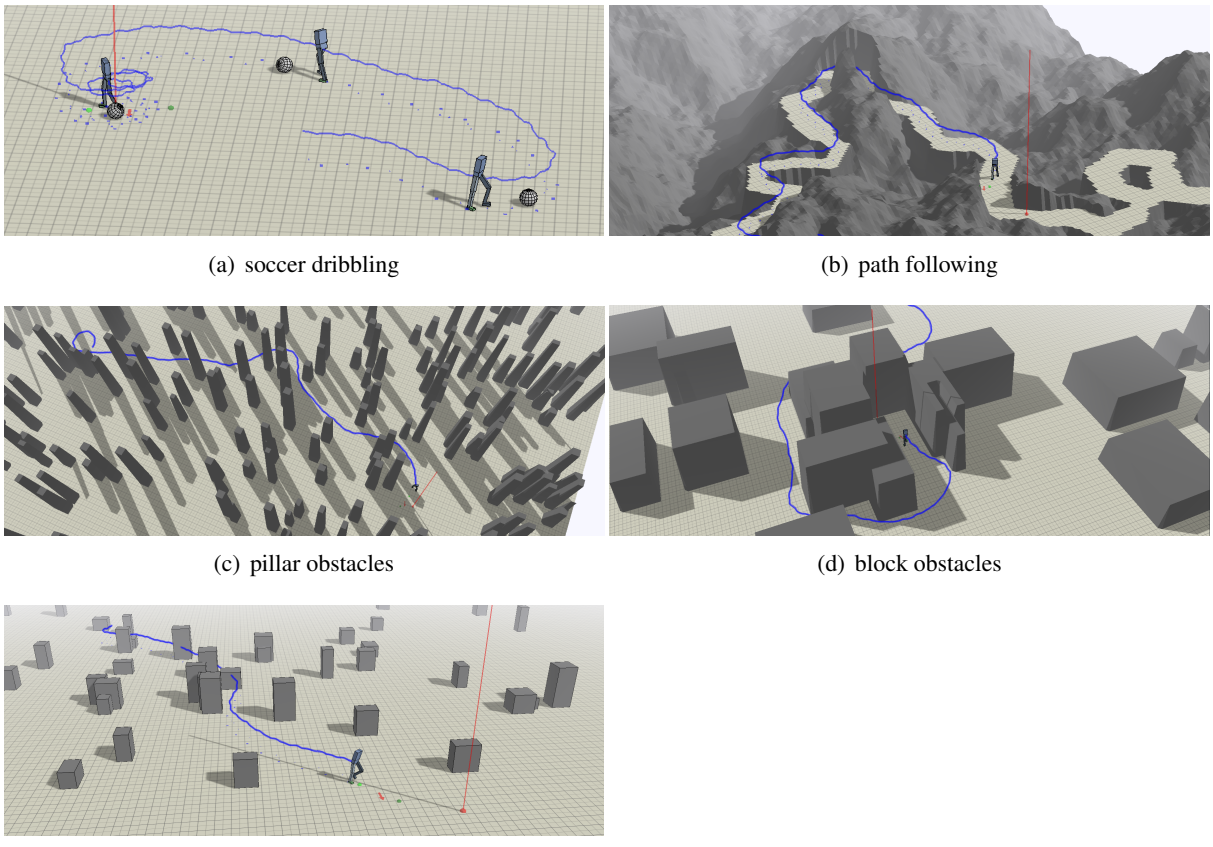


Figure 6.12: Learning curves with and without control hierarchy.



(a) soccer dribbling

(b) path following

(c) pillar obstacles

(d) block obstacles

(e) dynamic obstacles

Figure 6.13: Snapshots of HLC tasks. The red marker represents the target location and the blue line traces the trajectory of the character's center of mass.

all learned to identify and avoid obstacles using heightmaps and navigate across different environments seeking randomly placed targets. The more difficult dynamic obstacles environment, proved challenging for the HLC, reaching a competent level of performance, but still prone to occasional missteps, particu-

larly when navigating around faster moving obstacles. We note that the default LLC training consists of constant speed forward walks and turns but no stopping, which limits the options available to the HLC when avoiding obstacles.

Figure 6.12 compares the learning curves with and without the control hierarchy for soccer dribbling and path following. To train the policies without the control hierarchy, the LLC's inputs are augmented with g_H and for the path following task, the terrain map T is also included as part of the input. Convolutional layers are added to the path following LLC. The augmented LLC's are then trained to imitate the reference motions and perform the high-level tasks. Without the hierarchical decomposition, both LLC's failed to perform their respective tasks.

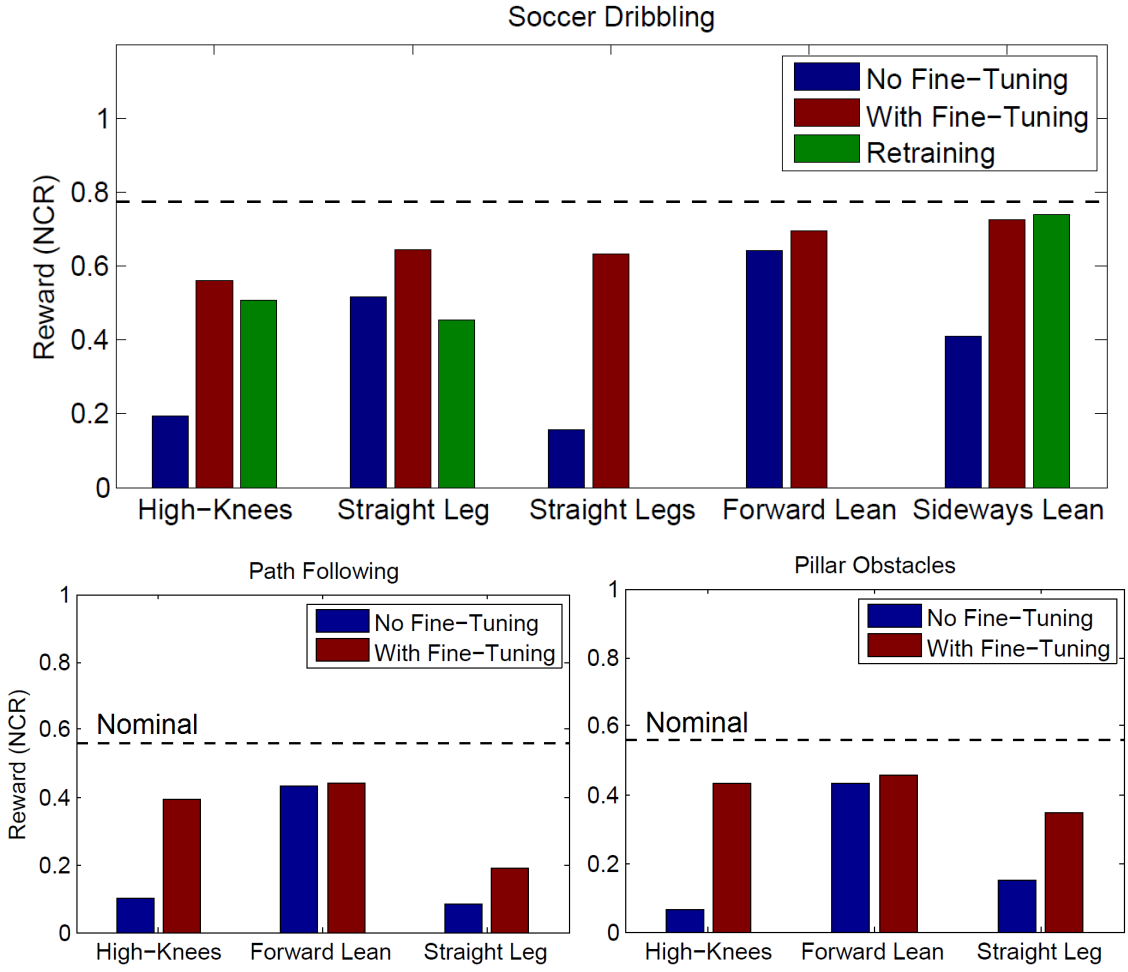


Figure 6.14: Performance using different LLC's for various tasks with and without HLC fine-tuning, and retraining. HLC are originally trained for the nominal LLC.

6.7.3 Transfer Learning

Another advantage of a hierarchical structure is that it enables a degree of interchangeability between the different components. While a common LLC can be used by the various task-specific HLC's, a

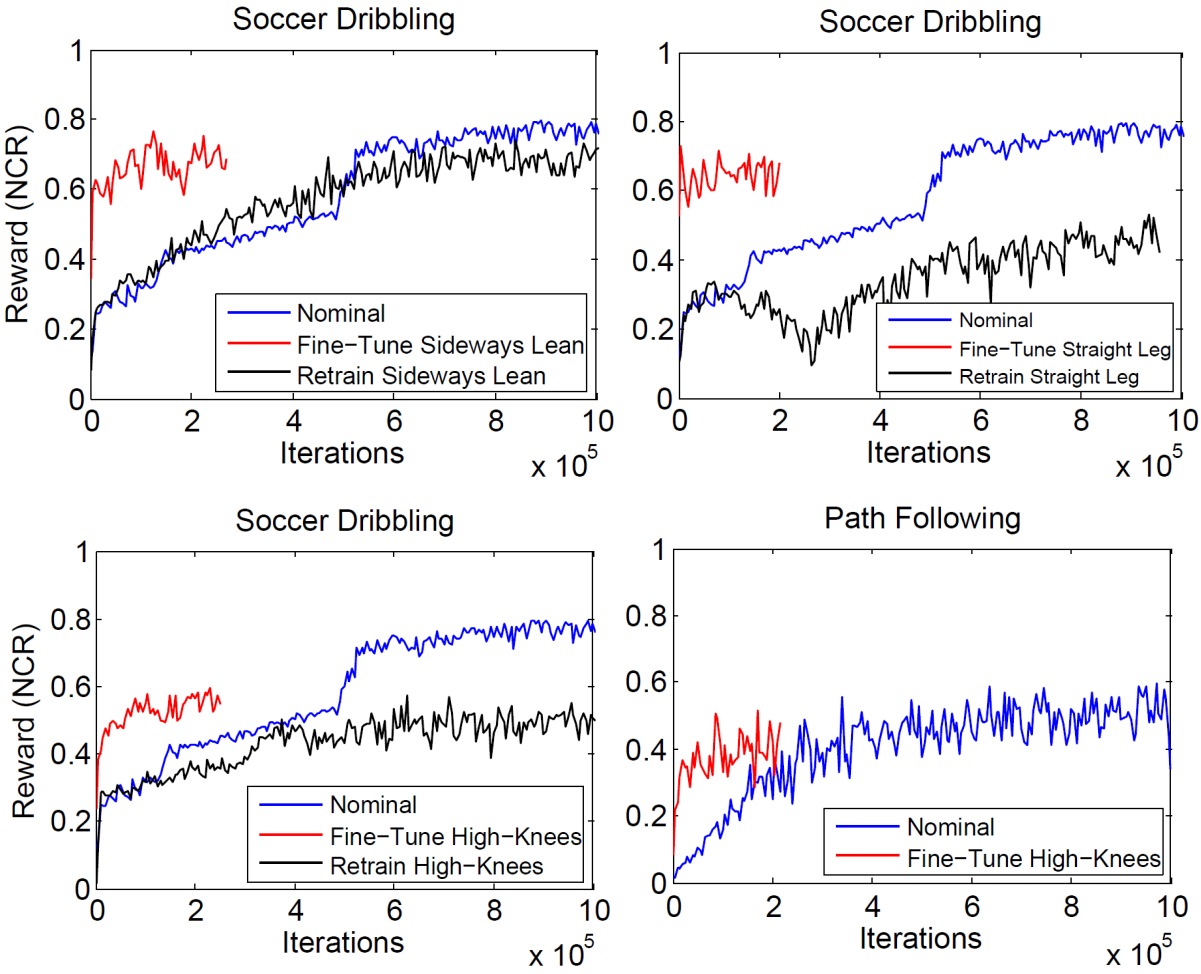


Figure 6.15: Learning curves for fine-tuning HLC's trained for the nominal LLC to different LLC's, and retraining HLC's from scratch.

common HLC can also be applied to multiple LLC's without additional training. This form of zero-shot transfer allows the character to swap between different LLC's while retaining a reasonable level of aptitude for a task. Furthermore, the HLC can then be fine-tuned to improve performance with a new LLC, greatly decreasing the training time required when compared to retraining from scratch. In Figure 6.14 the performance when using different LLC's is shown for soccer dribbling before and after HLC fine-tuning, and retraining from scratch. Fine-tuning is applied for 200k iterations using the HLC trained for the nominal LLC for initialization. Retraining is performed for 1 million iterations from random initialization. For soccer dribbling, the ability to substitute different LLC's is style dependent, with the forward lean exhibiting the least degradation and high-knees exhibiting the most. Table 6.3 summarizes the results of transfer learning between different HLC and LLC combinations.

Task + LLC	No Fine-Tuning	With Fine-Tuning	Retraining
Soccer + High-Knees	0.19	0.56	0.51
Soccer + Straight Leg	0.51	0.64	0.45
Soccer + Straight Legs	0.16	0.63	-
Soccer + Forward Lean	0.64	0.69	-
Soccer + Sideways Lean	0.41	0.72	0.74
Path + High-Knees	0.10	0.39	-
Path + Forward Lean	0.43	0.44	-
Path + Straight Leg	0.08	0.19	-
Pillars + High-Knees	0.06	0.43	-
Pillars + Forward Lean	0.43	0.45	-
Pillars + Straight Leg	0.15	0.35	-

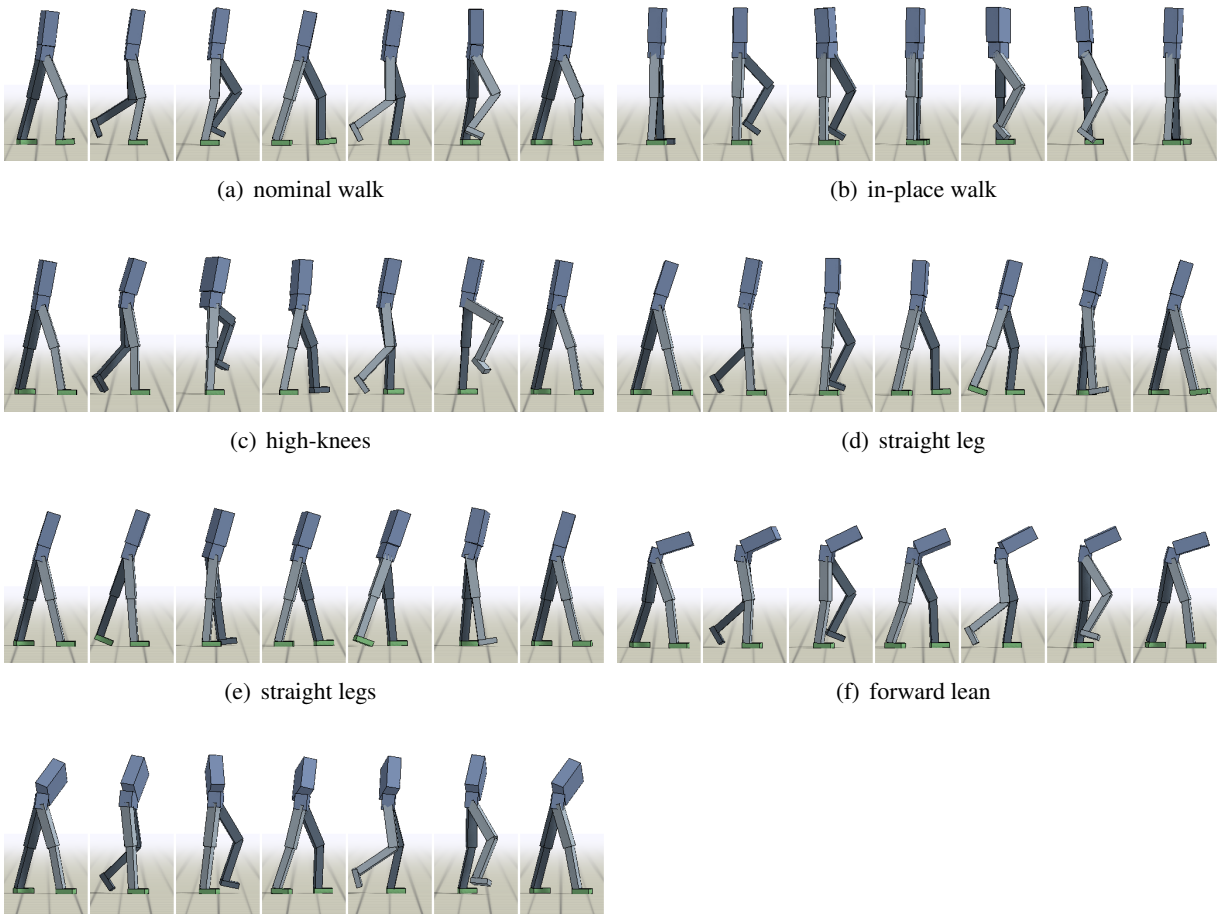
Table 6.3: Performance (NCR) of different combinations of LLC's and HLC's. **No Fine-Tuning:** directly using the HLC's trained for the nominal LLC. **With Fine-Tuning:** HLC's fine-tuned using the nominal HLC's as initialization. **Retraining:** HLC's are retrained from random initialization for each task and LLC.

6.8 Discussion

The method described in this paper allows for skills to be designed while making few assumptions about the controller structure or explicit knowledge of the underlying dynamics. Skill development is guided by the use of objective functions for low-level and high-level policies. Taken together, the hierarchical controller allows for combined planning and physics-based movement based on high-dimensional inputs. Overall, the method further opens the door to learning-based approaches that allow for rapid and flexible development of movement skills, at multiple levels of abstraction. The same deep RL method is used at both timescales, albeit with different states, actions, and rewards. Taken as a whole, the method allows for learning skills that directly exploit a variety of information, such as the terrain maps for navigation-based tasks, as well as skills that require finer-scale local interaction with the environment, such as soccer dribbling.

Imitation objective: The LLC learns in part on a motion imitation objective, utilizing a reference motion that provides a sketch of the expected motion. This can be as simple as a single keyframed planar walk cycle that helps guide the control policy towards a reasonable movement pattern, as opposed to learning it completely from scratch. Importantly, it further provides a means of directing the desired motion style. Once a basic control policy is in place, the policy can be further adapted using new goals or objective functions, as demonstrated in our multiple LLC style variations.

Phase information: Our LLC's currently still use phase information as part of the character state, which can be seen as a basic memory element, i.e., "where am I in the gait cycle." We still do not fully understand why a bilinear phase representation works better for LLC learning, in terms of achieving a given motion quality, than the alternative of using continuously-valued phase representation, i.e., $\cos(\phi), \sin(\phi)$. In future work, we expect that the phase could be stored and updated using an internal



(a) nominal walk (b) in-place walk

(c) high-knees (d) straight leg

(e) straight legs (f) forward lean

(g) sideways lean

Figure 6.16: LLC walk cycles.

memory element in a recurrent network or LSTM. This would also allow for phase adaptations, such as stopping or reversing the phase advancement when receiving a strong backwards push.

HLC-LLC interface and integration: Currently, the representation used as the interface between the HLC and LLC, $a_H \equiv g_L$, is manually specified, in our case corresponding to the next two footstep locations and the body orientation at the first footstep. The HLC treats these as abstract handles for guiding the LLC, and thus may exploit regions of this domain for which the LLC has never been trained. This is evident in the HLC behaviour visualizations, which show that unattainable footsteps are regularly demanded of the LLC by the HLC. This is not problematic in practice because the HLC will learn to avoid regions of the action space that lead to problematic behaviors from the LLC. Learning the best representation for the interface between the HLC and LLC, as demonstrated in part by [Heess et al., 2016], is an exciting avenue for future work. It may be possible to find representations which then allow for LLC substitutions with less performance degradation. An advantage of the current explicitly-defined LLC goals, g_L, is that it can serve to define the reward to be used for LLC training. However,

it does result in the LLC's and HLC's being trained using different reward functions, whereas a more conceptually pure approach might simply use a single objective function.

Motion planning: Some tasks, such as path navigation, could also be accomplished using existing motion planning techniques based on geometric constraints and geometric objectives. However, developing efficient planning algorithms for tasks involving dynamic worlds, such as the dynamic obstacles task or the soccer dribbling task, is much less obvious. In the future, we also wish to develop skills that are capable of automatically selecting efficient and feasible locomotion paths through challenging 3D terrains.

Transfer and parameterization: Locomotion can be seen as encompassing a parameterized family of related movements and skills. Knowing one style of low-level motion should help in learning another style, and, similarly, knowing the high-level control for one task, e.g., avoiding static obstacles, should help in learning another related task, e.g., avoiding dynamic obstacles. This paper has demonstrated several aspects of transfer and parameterization. The ability to interpolate (and to do moderate extrapolation) between different LLC motion styles provides a rich and conveniently parameterized space of motions. The LLC motions are robust to moderate terrain variations, external forces, and changes in gait period, by virtue of the exploration noise they experience during the learning process. As demonstrated, the HLC-LLC hierarchy also allows for substitution of HLC's and LLC's. However, for HLC/ LLC pairs that have never been trained together, the performance will be degraded for tasks that are sensitive to the dynamics, such as soccer dribbling. However, the HLC's can be efficiently readapted to improve performance with additional fine-tuning.

Learning efficiency: The sample efficiency of the training process can likely be greatly improved. Interleaving improvements to a learned dynamics model with policy improvements is one possible approach. While we currently use a positive-temporal difference advantage function in our actor-critic framework, we intend to more fully investigate other alternatives in future work.

Chapter 7

Conclusion

7.1 Discussion

In this thesis we have presented a number of frameworks for developing locomotion skills for simulated characters using deep reinforcement learning. Though the systems share similar underlying learning algorithms, the choice of timescales and action abstractions plays a critical role in determining the skills that can be achieved by the different policies. In Chapter 4, the use of parameterized FSMs enabled policies to operate and explore actions at the timescale of running and jumping steps. The MACE model can be viewed as a control hierarchy where the critics make high-level decisions in selecting the class of actions to perform, while the individual actors make low-level decisions regarding the execution of their respective class of actions. An important distinction to the hierarchy introduced in Chapter 6 is that both levels of the hierarchy in MACE operate at the same timescale but at different levels of action abstraction. Chapter 5 then explores the development of low-level policies that operate at fine timescales (e.g. 60Hz). Low-level policies often rely on less domain knowledge in crafting task-specific action abstractions, and instead work directly on simple representations such as torques and target angles. This flexibility partly stems from the policies' fine timescale, which provides a tight feedback loop that can quickly respond and adjust the actions in accordance to changes in the state. However, since reward discounting is performed in a per-step fashion, small timesteps limit a policy's ability to plan over longer time horizons, thereby producing more myopic behaviours. In contrast, policies that operate over coarse timescales can be more effective at planning over longer time horizons, enabling them to perform more complex tasks that require longer term planning. But larger timesteps often require low-level feedback structures, such as the SIMBICON balance strategy used by the FSMs, to handle unexpected perturbations over the duration of a control step.

The hierarchical policy outlined in Chapter 6 attempts to combine the advantages of different timescales by training each level of the hierarchy to operate with a different timestep. The LLC replaces the hand-crafted FSM utilized in Chapter 4 with learned low-level control strategies that remain robust in spite

of significant perturbations. The LLC then provides the HLC with a higher-level action abstraction, in the form of footstep goals, that allows the HLC to explore actions over larger timescales. By acting over larger timesteps, the HLC is able to better accommodate the long term planning required for more complex tasks such as navigation and soccer dribbling.

7.2 Conclusion

One of the defining characteristics of physics-based character animation is the high-dimensional continuous state and action spaces that often arise from motion control problems. While handcrafted reduced models have been effective in reproducing a rich repertoire of motions, individual models are often limited in their capacity to generalize to new skills, leading to a catalog of task-specific controllers. Deep reinforcement learning offers the potential for a more ubiquitous framework for motion modeling. By leveraging expressive but general models, control policies can be developed for diverse motions and character morphologies while reducing the need for task-specific control structures. Neural networks' capacity to process high-dimensional inputs also opens the opportunity to develop sensorimotor policies that utilize rich low-level descriptions of the environment.

While DeepRL has enabled the use of high-dimensional low-level state descriptions, our experience suggest that the choice of action abstraction remains a crucial design decision. The choice of action representation influences the timescale at which a policy can operate and shapes the exploration behaviour of the agent during training. Hierarchical policies provide a means of leveraging the advantages of different action abstractions, and can be beneficial for complex tasks that require a balance between long and short term objectives. In the future, we hope to explore techniques that can more effectively learn useful task-specific action abstractions directly from low-level action representations.

7.3 Future Work

Deep reinforcement learning opens many exciting avenues for computer animation and motion control. The methods we have presented in this thesis have been predominantly model-free, as the policies do not leverage any models of the dynamics of their environments. Model-based RL, which utilizes learned dynamics models, may allow the agents to more efficiently explore potentially beneficial behaviours as compared to the current use of random exploration noise. A learned dynamics model can also provide agents with valuable mechanisms for planning, which may improve performance for more challenging tasks that require strategic long-term behaviours.

While most of our work has been focused on single agents interacting with a predominantly passive environment, multi-agent systems is a promising future application, particularly for films and games, where agents are seldom situated in isolation. Training policies for both cooperative and adversarial tasks opens a wealth of potential applications. Cooperative policies that can observe and anticipate the

behaviours of other agents will be a vital component in integrating robots into assistive roles alongside their human counterparts.

With deep reinforcement learning, our policies are able to utilize rich sensory information from high-dimensional low-level state representations, such as heightmaps of upcoming terrain. However, our applications have mainly utilized visual and proprioceptive information. Exploring the use of additional sensory data, such as tactile information, may help in developing more dexterous skills for manipulation tasks. Furthermore, incorporating memory units, such as long-short term memory (LSTM), into the network can be beneficial for tasks that place greater emphasis on exploration and memorization, such as maze navigation.

By providing the policies with reference motions during training, we are able to significantly improve the motion quality as compared to previous model-free RL methods [Lillicrap et al., 2015a, Schulman et al., 2016]. But the quality of our resulting motions still falls short of what has been previously achieved with carefully engineered models [Coros et al., 2011b, Geijtenbeek et al., 2013]. Due to the prevalent use of simplified character models, designing appropriate motion priors is crucial to reproduce biologically plausible motions. Motion manifolds such as those proposed by Holden et al. [2016] may be a promising tool for providing a policy with feedback on the naturalness of its motions. Alternatively, motion priors can be incorporated into the actuation model by leveraging more biologically-plausible actuators [Geijtenbeek et al., 2013]. More realistic muscle models may lead to the emergence of natural behaviours without the need for extensive reward shaping, while also offering more predictive power for applications such as injury prevention and rehabilitation. We believe deep learning will be a valuable tool in developing policies for controlling these complex muscle models.

Bibliography

M. Al Borno, M. de Lasa, and A. Hertzmann. Trajectory optimization for full-body movements with complex contacts. *TVCG*, 19(8):1405–1414, 2013. → pages 7

J.-A. M. Assael, N. Wahlström, T. B. Schön, and M. P. Deisenroth. Data-efficient learning of feedback policies from image pixels using deep dynamical models. *arXiv preprint arXiv:1510.02173*, 2015. → pages 35

Y. Bai, K. Siu, and C. K. Liu. Synthesis of concurrent object manipulation tasks. *ACM Transactions on Graphics (TOG)*, 31(6):156, 2012. → pages 58

R. Blickhan, A. Seyfarth, H. Geyer, S. Grimmer, H. Wagner, and M. Günther. Intelligence by mechanics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 365(1850):199–220, 2007. → pages 39, 40

Bullet. Bullet physics library, Dec. 2015. http://bulletphysics.org. → pages 21, 71

S. Calinon, P. Kormushev, and D. G. Caldwell. Compliant skills acquisition and multi-optima policy search with em-based reinforcement learning. *Robotics and Autonomous Systems*, 61(4):369–379, 2013. → pages 18

J. Chestnutt, M. Lau, G. Cheung, J. Kuffner, J. Hodgins, and T. Kanade. Footstep planning for the honda ASIMO humanoid. In *ICRA05*, pages 629–634, 2005. → pages 57

S. Coros, P. Beaudoin, K. K. Yin, and M. van de Pann. Synthesis of constrained walking skills. *ACM Trans. Graph.*, 27(5):Article 113, 2008. → pages 18, 19, 57

S. Coros, P. Beaudoin, and M. van de Panne. Robust task-based control policies for physics-based characters. *ACM Transctions on Graphics*, 28(5):Article 170, 2009. → pages 7, 57

S. Coros, P. Beaudoin, and M. van de Panne. Generalized biped walking control. *ACM Transctions on Graphics*, 29(4):Article 130, 2010. → pages 6, 18

S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. van de Panne. Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics*, 30(4):Article TBD, 2011a. → pages 18, 38, 40

S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. van de Panne. Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics*, 30(4):Article 59, 2011b. → pages 6, 84

M. da Silva, Y. Abe, and J. Popović. Interactive simulation of stylized human locomotion. *ACM Trans. Graph.*, 27(3):Article 82, 2008. → pages 6, 7

M. da Silva, F. Durand, and J. Popović. Linear bellman combination for control of character animation. *ACM Trans. Graph.*, 28(3):Article 82, 2009. → pages 19

M. de Lasa, I. Mordatch, and A. Hertzmann. Feature-based locomotion controllers. In *ACM Transactions on Graphics (TOG)*, volume 29, page 131. ACM, 2010. → pages 7, 40

K. Ding, L. Liu, M. van de Panne, and K. Yin. Learning reduced-order feedback policies for motion skills. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2015. → pages 18

K. Doya, K. Samejima, K.-i. Katagiri, and M. Kawato. Multiple model-based reinforcement learning. *Neural computation*, 14(6):1347–1369, 2002. → pages 19

P. Faloutsos, M. van de Panne, and D. Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of SIGGRAPH 2001*, pages 251–260, 2001. → pages 19

R. Featherstone. *Rigid body dynamics algorithms*. Springer, 2014. → pages 22

A. Fukui, D. H. Park, D. Yang, A. Rohrbach, T. Darrell, and M. Rohrbach. Multimodal compact bilinear pooling for visual question answering and visual grounding. *CoRR*, abs/1606.01847, 2016. URL http://arxiv.org/abs/1606.01847. → pages 65

T. Geijtenbeek and N. Pronost. Interactive character animation using simulated physics: A state-of-the-art review. In *Computer Graphics Forum*, volume 31, pages 2492–2515. Wiley Online Library, 2012. → pages 6

T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013. → pages 18, 39, 40, 44, 84

H. Geyer, A. Seyfarth, and R. Blickhan. Positive force feedback in bouncing gaits? *Proc. Royal Society of London B: Biological Sciences*, 270(1529):2173–2183, 2003. → pages 41, 44

M. X. Grey, A. D. Ames, and C. K. Liu. Footstep and motion planning in semi-unstructured environments using possibility graphs. *CoRR*, abs/1610.00700, 2016. URL http://arxiv.org/abs/1610.00700. → pages 57, 58

S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation. *arXiv preprint arXiv:1610.00633*, 2016a. → pages 39

S. Gu, T. P. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. *CoRR*, abs/1603.00748, 2016b. URL http://arxiv.org/abs/1603.00748. → pages 8

P. Hämäläinen, J. Rajamäki, and C. K. Liu. Online control of simulated humanoids using particle belief propagation. *ACM Transactions on Graphics (TOG)*, 34(4):81, 2015. → pages 7

N. Hansen. The cma evolution strategy: A comparing review. In *Towards a New Evolutionary Computation*, pages 75–102, 2006. → pages 6, 24

M. Haruno, D. H. Wolpert, and M. Kawato. Mosaic model for sensorimotor learning and control. *Neural computation*, 13(10):2201–2220, 2001. → pages 18

M. Hausknecht and P. Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015a. → pages 9, 34

M. J. Hausknecht and P. Stone. Deep reinforcement learning in parameterized action space. *CoRR*, abs/1511.04143, 2015b. → pages 40, 44

N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2926–2934, 2015. → pages 9, 33

N. Heess, G. Wayne, Y. Tassa, T. P. Lillicrap, M. A. Riedmiller, and D. Silver. Learning and transfer of modulated locomotor controllers. *CoRR*, abs/1610.05182, 2016. URL http://arxiv.org/abs/1610.05182. → pages 80

T. Hester and P. Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *Machine Learning*, 90(3):385–429, 2013. → pages 18

J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien. Animating human athletics. In *Proceedings of SIGGRAPH 1995*, pages 71–78, 1995. → pages 6

D. Holden, J. Saito, and T. Komura. A deep learning framework for character motion synthesis and editing. *ACM Trans. Graph.*, 35(4):Article 138, 2016. → pages 84

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991. → pages 19

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, MM '14, pages 675–678. ACM, 2014a. ISBN 978-1-4503-3063-3. doi:10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889. → pages 71

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-3063-3. doi:10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889. → pages 29

L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics & Automation*, 12(4):566–580, 1996. → pages 57

V. Konda and J. Tsitsiklis. Actor-critic algorithms. In *SIAM Journal on Control and Optimization*, pages 1008–1014. MIT Press, 2000. → pages 15

J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. *Motion Planning for Humanoid Robots*, pages 365–374. Springer Berlin Heidelberg, 2005. → pages 57

J. Laszlo, M. van de Panne, and E. Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 155–162. ACM, 1996. → pages 6

M. Lau and J. Kuffner. Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280, 2005. → pages 58

J. Lee and K. H. Lee. Precomputing avatar behavior from human motion data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, pages 79–87, 2004. → pages 58

J. Lee and K. H. Lee. Precomputing avatar behavior from human motion data. *Graphical Models*, 68 (2):158–174, 2006. → pages 7

Y. Lee, S. J. Lee, and Z. Popović. Compact character controllers. *ACM Transctions on Graphics*, 28 (5):Article 169, 2009. → pages 7

Y. Lee, S. Kim, and J. Lee. Data-driven biped control. *ACM Transctions on Graphics*, 29(4):Article 129, 2010a. → pages 6

Y. Lee, K. Wampler, G. Bernstein, J. Popović, and Z. Popović. Motion fields for interactive character locomotion. *ACM Transctions on Graphics*, 29(6):Article 138, 2010b. → pages 7

Y. Lee, M. S. Park, T. Kwon, and J. Lee. Locomotion control for many-muscle humanoids. *ACM Trans. Graph.*, 33(6):218:1–218:11, Nov. 2014. ISSN 0730-0301. doi:10.1145/2661229.2661233. URL http://doi.acm.org/10.1145/2661229.2661233. → pages 7

S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1071–1079. Curran Associates, Inc., 2014. → pages 8, 33

S. Levine and V. Koltun. Learning complex neural network policies with trajectory optimization. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 829–837, 2014. → pages 8, 33

S. Levine, J. M. Wang, A. Haraux, Z. Popović, and V. Koltun. Continuous character control with low-dimensional embeddings. *ACM Transactions on Graphics (TOG)*, 31(4):28, 2012. → pages 7

S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015. → pages 25, 39

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015a. → pages 38, 39, 84

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015b. → pages 9, 33, 34

L. Liu, K. Yin, M. van de Panne, and B. Guo. Terrain runner: control, parameterization, composition, and planning for highly dynamic motions. *ACM Trans. Graph.*, 31(6):154, 2012. → pages 6, 7, 58

L. Liu, M. van de Panne, and K. Yin. Guided learning of control graphs for physics-based characters. *ACM Transactions on Graphics*, 35(3), 2016a. → pages 40

L. Liu, M. van de Panne, and K. Yin. Guided learning of control graphs for physics-based characters. *ACM Trans. Graph.*, 35(3):Article 29, 2016b. doi:10.1145/2893476. → pages 7

G. Loeb. Control implications of musculoskeletal mechanics. In *Engineering in Medicine and Biology*

Society, 1995., IEEE 17th Annual Conference, volume 2, pages 1393–1394. IEEE, 1995. → pages 39, 40

A. Macchietto, V. Zordan, and C. R. Shelton. Momentum control for balance. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 80:1–80:8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-726-4. doi:10.1145/1576246.1531386. URL http://doi.acm.org/10.1145/1576246.1531386. → pages 7

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. → pages 8, 13, 25, 27, 29

I. Mordatch and E. Todorov. Combining the benefits of function approximation and trajectory optimization. In *Robotics: Science and Systems (RSS)*, 2014. → pages 8, 33

I. Mordatch, M. de Lasa, and A. Hertzmann. Robust physics-based locomotion using low-dimensional planning. *ACM Trans. Graph.*, 29(4):Article 71, 2010. → pages 6, 18, 58

I. Mordatch, K. Lowrey, G. Andrew, Z. Popovic, and E. Todorov. Interactive control of diverse complex characters with neural networks. In *Advances in Neural Information Processing Systems 28*, pages 3132–3140, 2015a. → pages 40

I. Mordatch, K. Lowrey, G. Andrew, Z. Popovic, and E. V. Todorov. Interactive control of diverse complex characters with neural networks. In *Advances in Neural Information Processing Systems*, pages 3114–3122, 2015b. → pages 8, 33

U. Muico, Y. Lee, J. Popović, and Z. Popović. Contact-aware nonlinear control of dynamic characters. *ACM Trans. Graph.*, 28(3):Article 81, 2009. → pages 6, 7

U. Muico, J. Popović, and Z. Popović. Composite control of physically simulated characters. *ACM Trans. Graph.*, 30(3):Article 16, 2011. → pages 19

A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015. → pages 8

V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In J. Frnkranz and T. Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010. URL http://www.icml2010.org/papers/432.pdf. → pages 65

E. Parisotto, J. L. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015. → pages 19, 35

P. Pastor, M. Kalakrishnan, L. Righetti, and S. Schaal. Towards associative skill memories. In *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, pages 309–315. IEEE, 2012. → pages 18

X. B. Peng, G. Berseth, and M. van de Panne. Dynamic terrain traversal skills using reinforcement learning. *ACM Transactions on Graphics*, 34(4), 2015. → pages 7, 18, 19, 21, 57

X. B. Peng, G. Berseth, and M. van de Panne. Terrain-adaptive locomotion skills using deep

reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2016)*, 35(5), 2016. → pages 57

J. Pettr, J.-P. Laumond, and T. Simon. 2-stages locomotion planner for digital actors. In *SCA '03: Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 258–264, 2003. → pages 58

A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015. → pages 19

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. → pages 8, 20

J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015. → pages 14, 39

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR 2016)*, 2016. → pages 9, 84

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proc. International Conference on Machine Learning*, pages 387–395, 2014a. → pages 14, 45

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014b. → pages 9, 34

K. W. Sok, M. Kim, and J. Lee. Simulating biped behaviors from human motion data. *ACM Trans. Graph.*, 26(3):Article 107, 2007. → pages 6, 7

B. C. Stadie, S. Levine, and P. Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015. → pages 8

R. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation, 2001. → pages 9, 14

R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981. → pages 10, 12, 13

J. Tan, K. Liu, and G. Turk. Stable proportional-derivative controllers. *Computer Graphics and Applications, IEEE*, 31(4):34–44, 2011. → pages 21, 71

J. Tan, Y. Gu, C. K. Liu, and G. Turk. Learning bicycle stunts. *ACM Trans. Graph.*, 33(4):50:1–50:12, 2014a. ISSN 0730-0301. → pages 40

J. Tan, Y. Gu, C. K. Liu, and G. Turk. Learning bicycle stunts. *ACM Transactions on Graphics (TOG)*, 33(4):50, 2014b. → pages 6, 7

Y. Tassa, T. Erez, and E. Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913. IEEE, 2012. → pages 7

A. Treuille, Y. Lee, and Z. Popović. Near-optimal character animation with continuous control. *ACM Transactions on Graphics (TOG)*, 26(3):Article 7, 2007. → pages 7

E. Uchibe and K. Doya. Competitive-cooperative-concurrent reinforcement learning with importance sampling. In *Proc. of International Conference on Simulation of Adaptive Behavior: From Animals and Animats*, pages 287–296, 2004. → pages 19

L. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008. → pages 31

H. Van Hasselt. Reinforcement learning in continuous state and action spaces. In *Reinforcement Learning*, pages 207–251. Springer, 2012. → pages 16, 19, 21, 29, 40, 43, 59, 60

H. Van Hasselt and M. A. Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 272–279. IEEE, 2007. → pages 19

H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015. → pages 8, 29

J. M. Wang, D. J. Fleet, and A. Hertzmann. Optimizing walking controllers. *ACM Transctions on Graphics*, 28(5):Article 168, 2009. → pages 6

J. M. Wang, S. R. Hamner, S. L. Delp, V. Koltun, and M. Specifically. Optimizing locomotion controllers using biologically-based actuators and objectives. *ACM Trans. Graph*, 2012. → pages 40, 41, 42

P. WawrzyńSki and A. K. Tanwani. Autonomous reinforcement learning with experience replay. *Neural Networks*, 41:156–167, 2013. → pages 39

M. Wiering and H. Van Hasselt. Ensemble algorithms in reinforcement learning. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 38(4):930–936, 2008. → pages 18

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992. ISSN 0885-6125. doi:10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696. → pages 9

D. Wooden, M. Malchano, K. Blankespoor, A. Howardy, A. A. Rizzi, and M. Raibert. Autonomous navigation for bigdog. In *ICRA10*, pages 4736–4741, 2010. → pages 57

K. Yamane, J. J. Kuffner, and J. K. Hodgins. Synthesizing animations of human manipulation tasks. *ACM Trans. Graph.*, 23(3):532–539, 2004. → pages 58

Y. Ye and C. K. Liu. Optimal feedback control for character animation using an abstract model. *ACM Trans. Graph.*, 29(4):Article 74, 2010. → pages 6, 58

K. Yin, K. Loken, and M. van de Panne. Simbicon: Simple biped locomotion control. *ACM Transctions on Graphics*, 26(3):Article 105, 2007. → pages 6, 18, 73, 74

K. Yin, S. Coros, P. Beaudoin, and M. van de Panne. Continuation methods for adapting simulated skills. *ACM Transctions on Graphics*, 27(3):Article 81, 2008. → pages 6

P. Zaytsev, S. J. Hasaneini, and A. Ruina. Two steps is enough: no need to plan far ahead for walking

balance. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6295–6300. IEEE, 2015. → pages 62