Learning Locomotion: Symmetry and Torque Limit Considerations

by

Farzad Abdolhosseini

B.Sc., Sharif University of Technology, 2017

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)

September 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Learning Locomotion: Symmetry and Torque Limit Considerations

submitted by **Farzad Abdolhosseini** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Michiel van de Panne, Computer Science
Supervisor

Leonid Sigal, Computer Science
Additional Examiner

Abstract

Deep reinforcement learning offers a flexible approach to learning physics-based locomotion. However, these methods are sample-inefficient and the result usually has poor motion quality when learned without the help of motion capture data. This work investigates two approaches that can make motions more realistic while having equal or higher learning efficiency.

First, we propose a way of enforcing torque limits on the simulated character without degrading the performance. Torque limits indicate how strong a character is and therefore has implications on how realistic the resulting motion looks. We show that using realistic limits from the beginning can hinder training performance. Our method uses a curriculum learning approach in which the agent is gradually faced with more difficult tasks. This way the resulting motion becomes more realistic without sacrificing performance.

Second, we explore methods that can incorporate left-right symmetry into the learning process which highly increases the motion quality. Gait symmetry is an indicator of health and asymmetric motion is easily noticeable by human observers. We compare two novel approaches as well as two existing methods of incorporating symmetry in the reinforcement learning framework. We also introduce a new metric for evaluating gait symmetry and confirm that the resulting motion has higher motion quality.

Lay Summary

Reinforcement learning offers a flexible approach to learning locomotion skills in simulation. This work investigates two approaches that can make the learned motions more realistic. We incorporate gait symmetry into the learning process. We also propose to begin the learning process with exceptionally strong characters, which enables them to rapidly discover good solution modes, and then progressively revert to a weaker character in order to obtain a more realistic motion.

Preface

Chapter 4 is unpublished work done by myself with additional inputs from my supervisor, Michiel van de Panne.

Chapter 5 has been accepted as a long paper at Motion, Interaction and Games (MIG) 2019 as Farzad Abdolhosseini, Hung Yu Ling, Zhaoming Xie, Xue Bin Peng, Michiel van de Panne. *On Learning Symmetric Locomotion*. The DUP, PHASE, and NET methods were respectively invented by Hung Yu (Ben) Ling, Xue Bin (Jason) Peng, and myself. The majority of coding, writing the article, and conducting the experiments were done by me and Hung Yu (Ben) Ling. Zhaoming Xie and Xue Bin (Jason) Peng also contributed by conducting the experiments on Cassie and DeepMimic, respectively. Michiel van de Panne helped in writing the paper.

Table of Contents

List of Tables

List of Figures

Glossary

DRL deep reinforcement learning

FSM finite-state machine

MDP Markov decision process

PPO proximal policy optimization

TL torque limit

TLM torque limit multiplier

TRPO trust-region policy optimization

Acknowledgments

I would like to express my gratitude for those that have helped me throughout this journey. First, my supervisor, Michiel van de Panne, who has guided me at every step of the way. Your cheerful attitude is always motivating and the way you earnestly care for everyone around you is inspiring. I admire you, not just for your knowledge, but as a great human being.

Second, I would like to thank my lovely partner, Ainaz Hajimoradlu, for her unwavering love and support. Next in line are my colleagues who have helped me improve day by day: Glen Berseth, Hung Yu (Ben) Ling, Zhaoming Xie, Xue Bin (Jason) Peng, and Prashant Sachdeva.

Even more important, I would like to thank my family who have supported and encouraged me all my life. Particularly my mom and my aunt, Eshrat, without whose sacrifices I would not be here right now. Your love knows no bounds and I am forever grateful for all that you have given me.

Chapter 1

Introduction

Humans can gracefully move around without consciously thinking about how to coordinate our muscles. Our level of mastery over this basic skill makes it seem so mundane that most of us never think much of it. Yet, we are still unable to create controllers that replicate what a newborn horse can do after two hours. This is not, however, due to a lack of effort, as the problem of locomotion has been studied in many fields.

Computer graphics is interested in locomotion in order to bring virtual characters to life. We can divide motion generation techniques into two categories: *kinematic* and *dynamic* (physics-based) methods. The kinematic approach produces the desired motion by directly manipulating the object positions and joint angles as well as their respective velocities. However, modifying these values directly can lead to unrealistic motions. One major reason for this is that the resulting motions can break the laws of physics. As humans, we are adept at picking up such inconsistencies. An intuitive approach to fixing this problem is to use the laws of physics as constraints. This method is known as physics-based animation.

Physics-based animation first requires us to create accurate models of objects and articulated bodies by taking into account their parameters such as masses, dimensions, joint types, torque limits, and so on. We can then apply well studied Newtonian laws to simulate the interactions between these objects. Finally, the question then becomes that of, how we can provide the control to generate movements that we desire.

This question has led many researchers to tackle this challenging problem. Since 1985 when physics was first used for animation [1], many different approaches for motion generation have been proposed. These range from huge optimization problems constrained by the laws of physics that generated motions for a Luxo character [42], to the highly structured FSM based approaches [44] and the beloved CMA algorithm [12]. Recently, reinforcement learning (RL) has emerged as a promising approach to learning locomotion skills and it holds significant potential because of its flexibility. However, there remains a compelling need to improve learning efficiency and motion quality for RL to become a widely-adopted animation tool. One way of going about this problem is by incorporating expert knowledge, such as left-right symmetry, into the model.

RL for locomotion is generally formulated as a control task where the agent can manipulate

joint motors individually at each time step to achieve certain goals that are modelled by a reward signal. The generality of the RL formulation makes it applicable to a wide variety of settings. Recent success of RL in animation and machine learning shows the ability to produce robust learned locomotion for simulated humans, animals, and imaginary legged creatures.

Classical control theory studies almost the same problem as RL but from a different perspective. Control theory usually assumes complete knowledge about the system that is under the investigation, including the environment. Disturbances and uncertainties are modelled explicitly and extra assumptions about their nature are made in this framework. A control algorithm, such as LQR, can solve the task at hand very efficiently. However, this approach can fail to deliver any solution for non-linear systems.

RL has been moving in the opposite direction with model-free methods that attempt to make little to no assumption about the task at hand. This is at odds with the classical control theory perspective and it reflects why we still talk about RL and control theory as different disciplines. This separation from concrete models allows RL to solve problems at a higher abstraction level than control theory does. This can lead to a universal algorithm that can solve many sorts of problems and is highly flexible. However, this generality comes at a price. These algorithms are generally known to be inefficient as they require countless interactions with the system that is under investigation. To make this more concrete, it is common for a task to be considered solved after ten million time-steps which translates to more than one hundred days of non-stop interaction with the character or the robot.

Furthermore, encoding the properties of a desirable motion through the reward signal can prove highly challenging and can fail in unintuitive ways. Common reward functions for walking and running are primarily based on forward progress or a fixed root velocity. Surprisingly, the agent can produce a walk like behaviour, given little to no extra knowledge. However, the motions are usually far from appealing. Because progress is the primary reward signal, the agent tends to learn peculiar motions with the hands flailing around in the air and the head fixed in unnatural positions. Even characters without an upper body commonly find irregular gaits, such as a fencing gait that keeps one foot in front of the other.

Further engineering of the reward can alleviate some of the problems, but it can also produce other issues that are difficult to debug. Specifically, a common argument is that even though different styles of walking exist, humans and other animals tend to choose the most energy-efficient one. Therefore a common remedy to the problem raised above is to use an energy expenditure cost in the reward function. However, it has been argued that RL practitioners tend to set the weight of this term tends to be low enough that it can be ignored since this term can make learning difficult [45].

This motivates us to look at other ways to augment the traditional RL formulation with prior knowledge from the experts in order to gain more natural-looking motions. In this work, we explore two methods that can help achieve higher quality motion as well as faster learning. In Chapter 4, we will look at the effects of modifying the torque limits on the learning process.

This can be seen as an alternative to the problem caused by the energy expenditure cost as discussed above. We will show that using realistic torque limits from the start can hinder the training process to the point that the agent never learns to move in the allotted time. We can overcome this by using a simple curriculum schedule.

In Chapter 5, we will look at a big contributor to the poor quality in the motions that are generated via the RL paradigm, namely, asymmetric walking patterns. The left and the right sides of the human body are approximately symmetric. Consequently, walking patterns of healthy humans are generally quite symmetric as well. Symmetric motions are also perceived to be more attractive, e.g., for dance [5], and gait symmetry is seen as a desirable outcome for physiological manipulation [30]. However, RL agents commonly find asymmetric gait patterns such as fencing, which leads with one primary foot and tends not to switch the leading foot. We explore four methods for incorporating symmetry into the RL paradigm and discuss their advantages and drawbacks. We then compare their motion quality as well as the degree to which they achieve gait symmetry in practice.

Chapter 2

Related Work

Locomotion in humans and other animals is a long-standing problem. Different aspects of this problem have been the subject of study in numerous fields for decades, such as computer graphics, robotics, biomechanics, control, and more recently machine learning. In this work, however, we will be focusing on the results from computer graphics and to a more limited extent, the machine learning community.

2.1 Motion Symmetry

Motion symmetry has been a topic of interest for many years in the study of human motion and movement biomechanics. Symmetric motions are perceived to be more attractive, e.g., for dance [5], and gait symmetry is seen as a desirable outcome for physiological manipulation [30]. While symmetry is a common assumption in the study of gait and posture, individual gaits often do exhibit asymmetries due to various possible functional causes [34]. We refer the reader to a past review article [31] for insights into the degree of symmetry of lower limbs movement during able-bodied gait and the potential influence of limb dominance on the motion symmetry of the lower extremities and human gaits [29]. It is also not obvious how to best quantify the asymmetry of human gaits, and thus specific symmetry metrics have been proposed [18, 39].

2.2 Kinematic Locomotion

Key-framing is a common approach to making characters move. In this technique the animator has to decide the character position and joint orientations for each frame, often using computer-assisted software. However, this approach is time-consuming and requires a highly skilled user. The use of motion capture systems is an alternative approach, but in the past, this has required expensive equipment and is constrained by the confines of the studio. Much work has been put into reusing captured motions using techniques such as Motion Graphs [21]. Recent approaches, such as Phase-Functioned Neural Networks [17] and Mode-Adaptive Neural Networks [46], use neural networks to learn kinematic models. They can act based on a directional command from the user and can respond to terrain height variations.

Key-framing relies heavily on the mental biases and knowledge of the animator. It is therefore perhaps the simplest way of incorporating all forms of expert knowledge into the resulting motion, including energy efficiency and symmetry. It is also common in kinematic-based approaches to mirror all the available motion data in order to double the effective size of the dataset and to reflect the often-symmetric nature of human locomotion, e.g., [6, 16].

2.3 Physics-Based Locomotion

The robust control of physics-based character locomotion is a long-standing challenge for character animation. We refer the reader to a survey paper for a detailed history [11]. 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 [15], and a rich variety of walking styles [7, 22, 44]. Many controllers developed for physics-based animation further use optimization methods to improve controllers developed around an FSM-structure, or use an FSM to define phase-dependent objectives for an inverse dynamics optimization to be solved at each time step. Policy search methods, e.g., stochastic local search or CMA [12], can be used to optimize the parameters of the given control structures to achieve a richer variety of motions, e.g., [8, 44], and efficient muscle-driven locomotion [40]. Many of the FSM controllers use hard-coded symmetries, which assign the roles of stance-leg or swing-leg to the left and right legs, as a function of the FSM state. The trajectory optimization-based methods also commonly assume motion symmetry when convenient, e.g., [25].

More recently, locomotion synthesis has attracted significant attention from the reinforcement learning (RL) community, where the OpenAI Gym tasks have become a popular RL benchmark [4]. In this context, symmetry constraints are commonly not imposed and the torque limits are unrealistic. As a consequence, the resulting motions are often idiosyncratic and have noticeable asymmetries. Further work extends these efforts in a variety of ways, including traversing challenging terrains [13]. More realistic and dynamic motions can be achieved with the help of motion-capture clips [27, 28] and these use what in Chapter 5 is referred to as the PHASE symmetry method, with the goal of more efficient learning. [24] uses a variation of PHASE in which individual strides (half steps) are mirrored and concatenated to generate symmetric reference motions. However, there exist no robust documented experiments to verify efficiency gains. The efficient learning of controllers that are capable of producing high-quality motion for realistic-strength characters remains a challenging problem in the absence of motion capture data. Recent work makes progress on this problem using RL with a combination of energy optimization, learning curriculum, and an auxiliary motion symmetry loss [45], which we shall refer to as the LOSS method.

Chapter 3

Reinforcement Learning

In this chapter, we provide a brief review of reinforcement learning (RL). RL emerges from the idea that humans and other animals tend to learn about the world through interaction. We can observe the world around us and act in certain ways to achieve our goals, and at the same time learn more about the world that we live in. RL is a computational approach to learning from interactions with the final goal of maximizing a numerical reward signal [36].

RL is applicable to a wide variety of applications, but this generality can also be problematic. The learner is not told which actions are better or worse and it needs to figure everything out itself using a possibly weak reward signal. To make matters worse, the consequences of an action may not be immediately visible to the learner as they might be revealed only after a long duration of time. This article is focused on introducing some structure into this formulation while keeping its flexibility.

3.1 Markov Decision Process

The problem formulation in RL is based on the concept of a Markov decision process (MDP). The MDP is defined by a tuple $\{\mathcal{S}, \mathcal{A}, P, r, \gamma\}$, where $S \in \mathbb{R}^n$ and $A \in \mathbb{R}^m$ are the state space and action space of the problem, respectively. The transition function $P : S \times A \times S \to [0, \infty)$ is a probability density function, with $P(s_{t+1} \mid s_t, a_t)$ being the probability density of visiting s_{t+1} given that at state s_t, the system takes action a_t. The reward function $r : S \times A \to \mathbb{R}$ gives a scalar reward for each transition of the system. $\gamma \in (0, 1]$ is the discount factor. A deterministic MDP is a special case where the transition function and the initial state distribution are deterministic.

Tasks can be categorized into two categories: *episodic* and *continuing* [36]. Episodic tasks can naturally be divided into subsequences known as *episodes*, such as a single match in sports or one play of a game. Each episode ends after a certain period of time, known as the *time horizon*, has passed or a pre-specified *terminal state* has been reached. In continuing tasks, the task goes on without limit and there is no natural notion of an episode present. Here, we will consider the episodic case with a fixed time horizon T. For a more in depth discussion

please refer to [36]. The goal of reinforcement learning is to find a parameterized policy π_θ, where $\pi_\theta : S \times A \to [0, \infty)$ is the probability density of a_t given s_t, that solves the following optimization problem:

$$\max_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right]. \tag{3.1}$$

Here, $\tau = (s_0, a_0, r_0, ..., s_T)$ is known as a *trajectory* and the probability of encountering a trajectory is computed according to $p_\theta(\tau) = P(s_0)\Pi_{t=0}^{T-1}\pi_\theta(a_t|s_t)P(s_{t+1}|s_t, a_t)$. The discount factor ($\gamma$) controls how much we care about future rewards rather than immediate rewards. The (possibly discounted) sum of the future rewards is also known as the *return* of a trajectory:

$$G(\tau) \doteq \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t).$$

Another important categorization of tasks is based on whether state and action spaces are discrete or continuous. Discrete spaces are generally known to have more well-defined solutions, namely tabular algorithms, than the continuous spaces. In this thesis, we focus on tasks where both state and action spaces are continuous.

3.2 Policy Gradient Methods

Many algorithms have been proposed for solving RL tasks and each is useful in certain scenarios. This section will explain the proximal policy optimization (PPO) algorithm which is used in the following chapters. For a discussion of other existing methods please refer to [2].

PPO belongs to the policy gradient methods class which have been shown to work well on continuous tasks. The idea behind the Policy Gradients (PG) algorithm is straight-forward, namely, to optimize the average return by computing an approximate gradient with respect to the underlying policy parameters, and then taking a gradient ascent step to increase it.

To optimize the objective, PG directly optimizes the policy π. One of the underlying assumptions of PG is that the policy should be stochastic rather than deterministic for this algorithm to work, although this assumption can be relaxed [23]. Furthermore, we assume that this stochastic policy is parametrized by parameters θ and therefore the algorithm's job is to find the optimal parameters θ^* that maximize Equation (3.1). To maximize the objective, we need to know the gradient $\nabla_\theta J(\theta)$. Since computing this gradient requires knowledge about the dynamics of the MDP, PG approximates it by using the REINFORCE trick [41]:

$$\nabla_\theta J(\theta) = \nabla_\theta \int G(\tau) p_\theta(\tau) \tag{3.2}$$

$$= \int G(\tau) \nabla_\theta p_\theta(\tau) \tag{3.3}$$

$$= \int G(\tau) p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \tag{3.4}$$

$$= \mathbb{E}_{\tau \sim p_\theta} [G(\tau) \nabla_\theta \log p_\theta(\tau)] \tag{3.5}$$

We can switch the integral and the gradient operator if the policy is differentiable everywhere. The equality is based on the following identity:

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$$

Next, we can expand $\nabla_\theta \log p_\theta$:

$$\nabla_\theta \log p_\theta = \nabla_\theta \left[\log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right] \tag{3.6}$$

$$= \nabla_\theta \left[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \right] \tag{3.7}$$

$$= \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{3.8}$$

The extra terms are independent of θ and therefore they do not contribute to the gradient. Substituting this back into Equation (3.5) we arrive at the following expression. For simplicity the discount factor, γ, has been set to one:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta} \left[G(\tau) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{3.9}$$

$$= \mathbb{E}_{\tau \sim p_\theta} \left[\left(\sum_{t=1}^{T} r_t \right) \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \right] \tag{3.10}$$

Using this formulation we can use Monte Carlo sampling [35] to approximately compute the gradient in order to iteratively improve the policy:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=1}^{T} r_{i,t} \right) \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \right] \tag{3.11}$$

Where, $\tau_i = (s_{i,1}, a_{i,1}, r_{i,1}, ..., s_{i,T})$ are sample trajectories from p_θ. With this we arrive at the REINFORCE algorithm [41].

Unfortunately, this is not a good estimator in practice as its variance can be quite high. Multiple tricks exists that try to alleviate this problem. The simplest is to increase the number of sampled trajectories N, however, this also makes the algorithm less efficient. One observation is that the reward time step t only causally depends on actions that were made until time t and are independent of decisions that are made afterwards. In other words, action a_t can only be responsible for the cost to go from time t forward. With some abuse of notation we can write:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p_\theta} \left[\sum_{t=1}^{T} \left(\nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_t \right) \right] \tag{3.12}$$

$$= \mathbb{E}_{s_t \sim p_\theta} \left[T \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_t \right], \tag{3.13}$$

where s_t is any state randomly sampled from a randomly sampled trajectory.

Another trick is to use a learned value function $\hat{V}(s)$ also known as a *critic*. It can be shown that subtracting out a fixed value from the cumulative return in the above formula does not change the value of the expectation. Therefore, if $\hat{V}(s_t)$ is a good estimate of $\sum_{t'=t}^{T} r_t$ then the following approximator would have lower variance:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{s_t \sim p_\theta} \left[T \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} r_t - \hat{V}(s_t) \right) \right]. \tag{3.14}$$

3.3 Proximal Policy Optimization (PPO)

PG is difficult to get working in practice. This problem is partly attributed to destructive updates, where a bad update during training can make the performance drop rapidly. The trust-region policy optimization (TRPO) algorithm [32] was introduced to solve this problem. The authors explain that destructive updates happen because the algorithm makes large updates based on an optimistic guess. Therefore TRPO only allows the policy to be updated within a trusted region. In other words, it is a step-size control mechanism.

Later on, PPO was introduced as a faster alternative to TRPO. Instead of explicitly enforcing a trust-region, PPO slightly changes the PG update rule as follows:

$$L_{CLIP} = \mathbb{E}_{s_t \sim p_\theta} \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right], \tag{3.15}$$

where the probability ratio is defined as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and the advantage is just a shorthand for $\hat{A}_t = \sum_{t'=t}^{T} r_t - \hat{V}(s_t)$. The hyper-parameter ϵ controls the step-size and it commonly takes on a value in the range $[0.1, 0.2]$. For more information please refer to [33].

Intuitively, by clipping the objective, the flow of gradient is blocked if it tries to push the

current policy outside the interval $[1-\epsilon, 1+\epsilon]$ around the old policy. Therefore, PPO has been advertised as enforcing a trust-region on the updates. Although the validity of this claim has recently been challenged [19], the fact remains that with the right hyper-parameter setting PPO is one of the best model-free algorithms in practice today.

Chapter 4

Torque Limit Considerations

Much effort in reinforcement learning has been spent on finding better or faster algorithms that can solve any RL problem in a model-free way, i.e. only through interactions with the environment. However, most success stories, such as TD-Gammon [37] and AlphaGo [10], still required careful engineering. Therefore, we believe that the design of the character and the tasks can be in some cases far more important to achieving superior results than the algorithm itself. In other words, making the problem itself simpler and more compatible with the algorithm used can be more productive than finding new algorithms that claim to solve more difficult problems. In this chapter, we look at the effects of different torque limit (TL) settings and we show that a simple torque limit curriculum can help achieve higher rewards and more reliable results.

4.1 Introduction

To measure the strength of a person we can measure the strength of his/her muscles. Similarly, we can measure the strength of a robot by measuring the strength of its motors. To do this, we need to consider each joint separately. A natural measure of strength is the amount of torque that the joint can produce. Furthermore, as ideal robots are consistent and never get tired like humans do, we only need to measure the maximum amount of torque output. Hence, a standard way of representing robot strength is through a TL vector. Naturally, a simulated character modelled after a robot needs to have a pre-specified TL. The correctness of TL is of high importance as it determines the capabilities of the robot. Having a higher or lower TL setting, therefore, has serious implications on what the agent can or cannot do and how quickly or more reliably the task can be solved.

Experimenting in a simulator, however, allows us to use torque limits that are much higher or lower than in real life and to investigate their effects. In most applications, lower limits tend to be more desirable. In robotics, using excessive forces can cause damage to the robot as well as its surroundings including any human or animals that may be in its vicinity. Using a low torque limit can be seen as a simple but effective safety mechanism that can alleviate many problems. The consequences of using unrealistic limits in computer graphics may not be

as dire, but humans are good at perceiving inconsistent movements performed by human-like characters. An impossibly high jump, pushing a heavy object with one hand, and recovering from a fall by using excessive strength are all immediately recognizable by a human observer. Low limits, on the other hand, are usually not noticeable unless the character or robot fails to accomplish the task at hand, as people rarely ever use maximum strength for day to day tasks.

Consequently, researchers in robotics and computer graphics tend to care about using realistic limits on the characters. On the other hand, the machine learning community is usually less concerned with such details and tends to use characters with excessive TL. This has been a source of (informal) complaints from the aforementioned communities.

One common way of compelling the agent to use less force is via an energy consumption cost. This approach is widely used in practice. However, it has two disadvantages. First, the agent can still use excessive force if it so desires. More importantly, the energy consumption can be harmful for training. Therefore, the weight of the energy consumption cost is set low enough that it may be ignored by the agent [45].

We aim to solve both of these problems by using TL as hard constraints. As a result, this chapter aims to answer the following questions: how much does the torque limit affect learning and can we make use of this knowledge to find better solutions?

Our results indicate that TL settings strongly affect the final solution. Below a certain threshold, the learning algorithm generally fails to find any solution, in a reasonable amount of time, that can solve the task. We demonstrate that this phenomenon is not a consequence of the problem setup but rather a limitation of the optimization procedure by showing the existence of higher-performing solutions with the same setting. Finally, we offer a simple solution to fix this problem by using a curriculum during learning. This helps the agent to learn in a simpler environment and then transfer the learned solution to the more difficult setting.

4.2 Environments

We experiment with a set of existing locomotion environments from Roboschool [20]. All characters are simulated using PyBullet [9] which is a Python interface for the Bullet3 physics engine. In all of the environments, the task is for the character to walk as far as possible in the forward direction in the allotted time. The reward function also includes terms to encourage the agent to use less energy and stay "alive" longer, i.e. to not fall. The observation space in all of these environments consists of root information (root z-coordinate, x and y heading vector, root velocity, roll, and pitch), joint angles, joint angular velocities, and binary foot contact information. The torques are all normalized between -1 and 1.

Walker2D is a simplified bipedal character whose movements are constrained to a 2D plane. An action is a 6D vector corresponding to torques at the hip, knee, and ankle on both left and right legs. The observation space is 22D and it weighs about 24kg.

Hopper is a one-legged character constrained to a 2D plane. It is similar to the Walker2D with one leg missing. The action space is 3D where each dimension controls the torque at the hip, knee, and ankle. The observation space is 15D and it weights about 16jg.

HalfCheetah is a 2D model that closely resembles a quadruped with only a fore and a hind left (no sides). An action is a 6D vector, similar to Walker2D, corresponding to the normalized torques at the thigh, shin, and the foot for each of the fore and hind legs. The observation space is 26D and consists of the same information as Walker2D with the addition of more fine-grained contact information. The character weighs about 38kg.

Ant is a 3D character that resembles an insect with four legs. It consists of a torso as well as four legs that are each divided into two segments. The action space is 8D and the observation space is 28D containing the same information as Walker2D. Despite being three dimensional, this character is highly stable due to having four legs. The character weighs about 182kg.

4.3 Methods

To experiment with different torque limits we need a way to dynamically modify the limits in each environment. Therefore we slightly modified the aforementioned environments to include a parameter denoted by the torque limit multiplier (TLM). TLM is a single scalar variable and the torque limits are altogether multiplied by this shared scalar which lets us shrink or expand the limits all at once. It is possible to have a specific multiplier for each joint, however, a single variable is sufficient for the purposes of our work.

We use our implementation of PPO [33] for all the experiments[1]. The hyper-parameters used can be found in Section A.1. Each experiment collected 6 million environment time steps in total and all the experiments were replicated five times each.

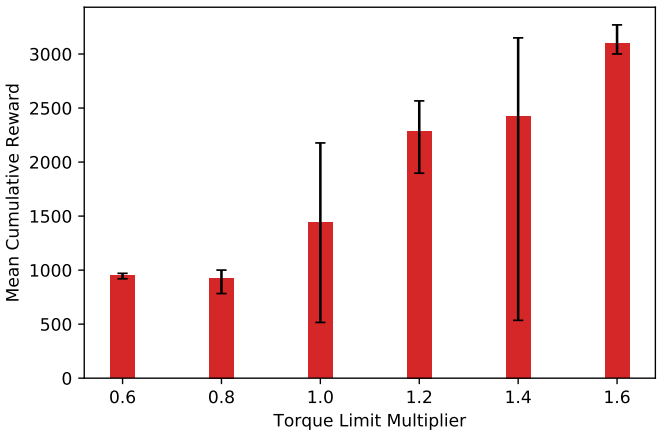
4.4 Results

In this section, we begin by demonstrating the effect of torque limit on training and then provide a more appropriate way of enforcing this limit. All plots in this section report the average performance as well as the minimum and maximum across five runs.

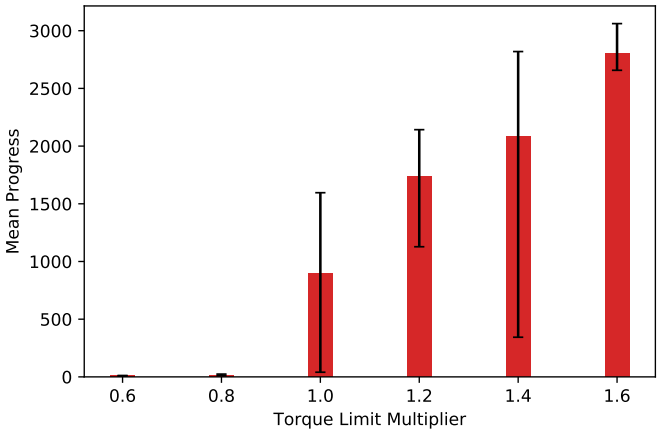
4.4.1 Torque Limit Baseline Experiments

To determine how the torque limit can affect the final solution, we run a set of experiments on the Walker2D environment with different values of TLM specified. To rule out the effect of random noise which is an important contributing factor in deep reinforcement learning (DRL) experiments [14] we run each experiment five times with different random seeds and the mean of the data is reported as well as the best and the worst results.

[1]The source code is available at https://github.com/farzadab/walking-benchmark



(a) Cumulative rewards



(b) Amount of progress in each iteration

Figure 4.1: Final performance of the agent with different TLM values.
The error bars reflect the best and the worst performance achieved by the same algorithm over five runs with different random seeds.

The final cumulative return, as well as the amount of progress made at test time, are shown in Figure 4.1. Not surprisingly the agents that had a higher torque limit constraint achieved higher returns. Interestingly, below a certain point (all runs with $TLM \leq 0.8$ as well as some runs with $TLM = 1$) the agent failed to make any forward progress. Note that the cumulative return in these cases is still as high as one thousand. This is the result of the agent learning to stand still and avoiding early termination instead of learning to walk.

The results for the default torque limit ($TLM = 1$ in the same figure) show us something interesting. First, the variance in the results is high. This is a known problem of reinforcement learning algorithms [14]. More importantly, this hints at the existence of a local optimum where the agent does not learn to move and just avoids early termination by standing still. Even though the results of training with higher torque limits show variations as well, none of them seem to be stuck in this local optima. Lastly, the results seem to indicate that it is not

possible to walk with the lower torque limits of 0.6 and 0.8.

4.4.2 Torque Limit Curriculum

If our hypothesis is correct that the agents are getting stuck in a local optimum with lower torque limits, it may be possible to get a better controller simply by using a better initialization. Agents trained with higher torque limits can intuitively provide a good starting point. This leads us to *curriculum learning* [3].

Curriculum learning is motivated by how humans and animals learn and is based on the idea of learning gradually from simple concepts to more difficult ones. In this approach, instead of training on the most difficult version of the problem, the training is divided into multiple stages where the first stage is a simplified version of the problem and task becomes more difficult at each stage until the last stage in which the agent is faced with the original version of the problem.

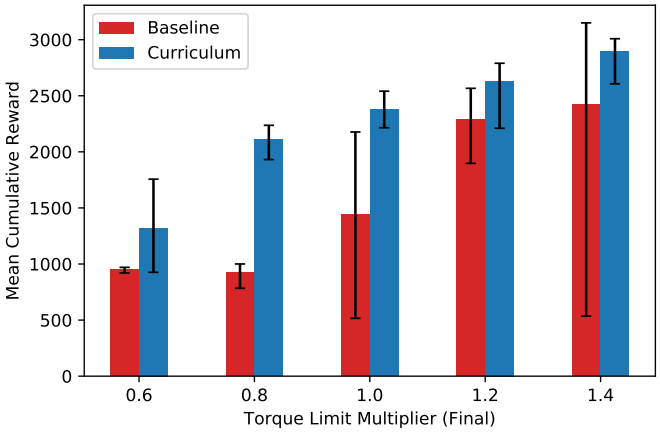
According to Figure 4.1, tasks with higher torque limits seem to be easier to solve. Therefore, we can define a curriculum where the agent first sees high torque limit environments but gradually the limit is lowered linearly until it matches our final target. To make the training more stationary the training is divided into several levels during which TLM stays fixed. The number of these levels is a hyper-parameter denoted by $NLevels$. For most experiments we used $NLevels = 10$.

The results of applying the torque limit curriculum can be seen in Figure 4.2. This method not only achieves higher performance, it also manages to sidestep the local optima as evident in Figure 4.2b. As a bonus, this approach seems to be more reliable in most cases as its variance seems to be lower than the baseline approach, except at $TLM = 0.6$. In the latter case, the baseline always converges to the local optima which has relatively low variance, however, this is not the desired behaviour.

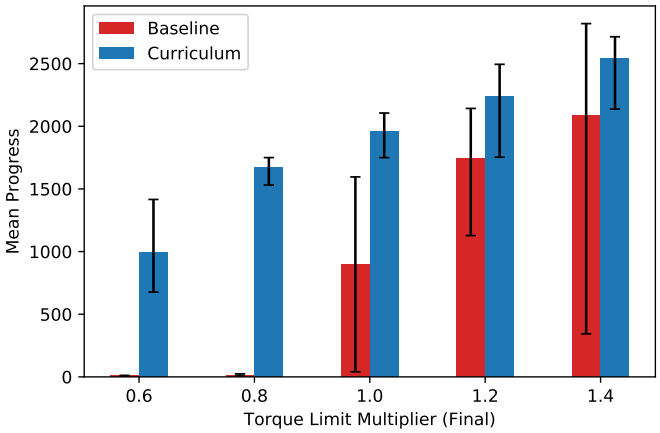
4.4.3 Ablation and More Environments

To further validate our approach, we test it on the Half Cheetah, Ant, and Hopper environments as well. As a baseline, we keep TLM fixed at the target value. Also, we compare our results with another curriculum approach that is very similar to our own, namely, an exploration rate curriculum, where the TLM is kept fixed the same as in our baseline. In this approach, the amount of exploration noise is varied at training time by starting at a high value to explore the solution space well and annealing it to a lower value in order to increase the final motion quality.

The results are shown in Figure 4.3. All experiments used the same hyper-parameters, namely with $NLevels = 10$ and the TLM going from 1.2 to 0.6. The exploration curriculum seems to be helpful to some extent but the TLM curriculum works best for all environments, specifically for the Ant.



(a) Cumulative rewards



(b) Amount of progress in each iteration

Figure 4.2: The effect of curriculum learning for Walker2D.
Blue plots show the final performance of the agents which started out with $TLM = 1.6$ and the TLM was decreased to the target value in ten steps. The red bars are the same as in Figure 4.1.

4.4.4 Curriculum Sensitivity

This curriculum learning technique seems to be useful in the different environments that we tested on, even though the gains vary across domains. However, as with many approaches, this method also includes hyper-parameters that need to be chosen by the user. We can assume that the target torque limits are a given, but the starting limits are not fixed. Furthermore, the optimal value for the *NLevels* hyper-parameter is also unknown. An important question to ask is: how sensitive is this method to the hyper-parameters. Therefore, we designed two experiments to answer this question.

First, we look at the starting TLM. We assume that the final TLM of 0.6 is fixed and we can vary the starting TLM value. The results of the experiment on the Walker2D environment can be seen in Figure 4.4. Surprisingly, all initial values seem to work relatively well. Specifically,

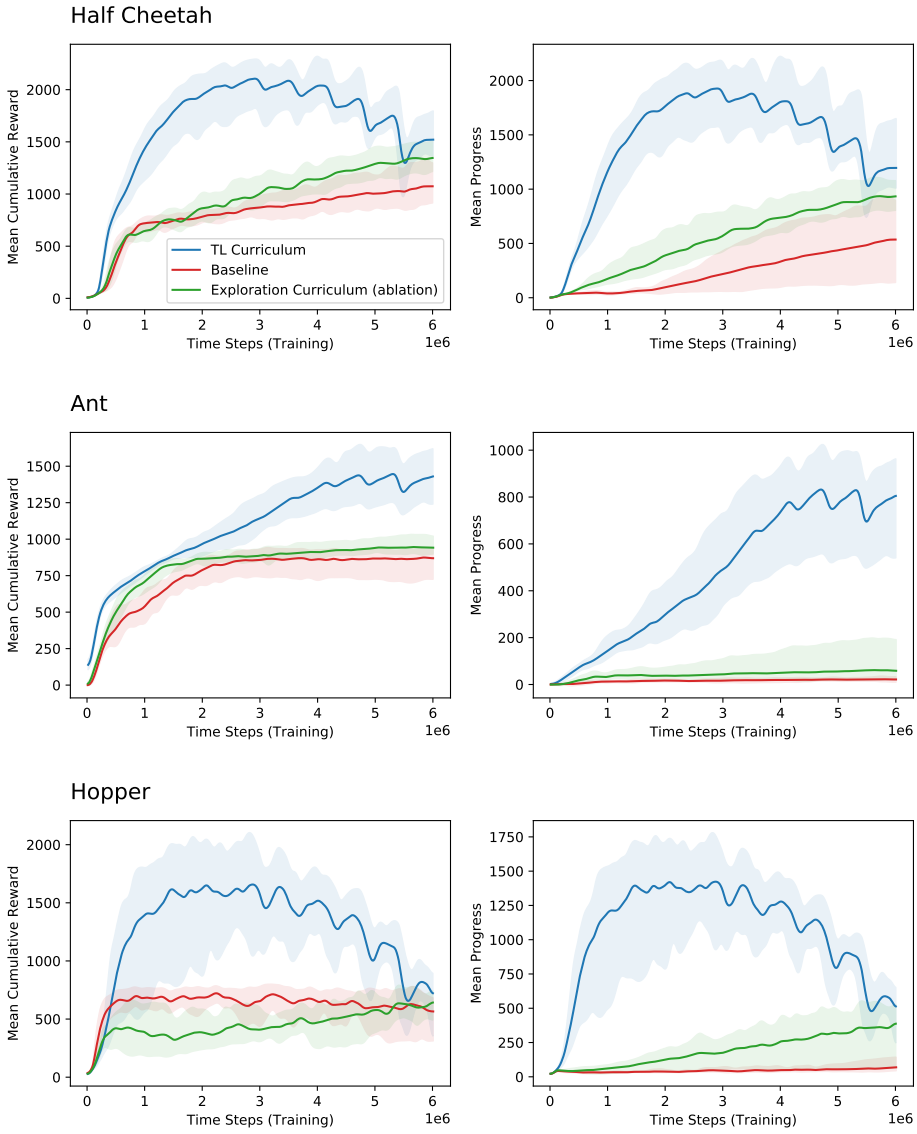


Figure 4.3: Torque limit curriculum results for Half Cheetah, Ant, and Hopper. The dips in performance reveal the points in which the value of TLM was reduced. The final TLM is constant across all methods. The shaded area corresponds to the minimum and the maximum values across five runs.

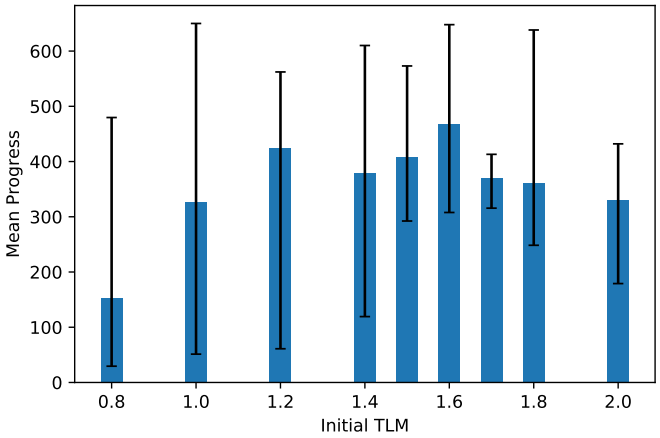


Figure 4.4: Curriculum sensitivity to initial TLM value.
All experiments with $TLM > 1.2$ have an acceptable performance both in terms of the average as well as the worse performance.

by comparing the results for $TLM = 0.8$ with Figure 4.1b, we see that the final performance has increased instead of decreasing even though the final TLM has decreased. This might simply be due to randomness, but it is also possible that the sudden changes in curriculum training let the agent escape local optima regions more easily. More importantly, the results seem to indicate that the method is not sensitive to this hyper-parameter as long as the initial TLM value is high enough to stumble upon a good solution.

Next, we look at the number of curriculum steps required, while keeping the total number of simulation steps fixed. A low number means an abrupt change but a high number would mean changing slowly but frequently. Both approaches have their merits. Changing slowly means the previous solution will still work under the new conditions, but it also means that there might not be enough time to get adjusted to the new situation before the next change. Abrupt changes can also be useful for getting out of a local solution. Again, we run a similar experiment as before on the Walker2D environment with the TLM going from 1.6 to 0.6 with different number of steps. The results are provided in Figure 4.5.

The results in both cases show some variability, which is to be expected, but there is no clear winner or a general trend to be pointed out. This is reassuring, as it shows that the method is not sensitive to small hyper-parameter changes. This makes the method more easily applicable to different settings.

4.5 Conclusions

In this chapter, we showed that the construction and the details of the locomotion environment are important, and the best design for learning need not be the most realistic one. Here, we looked at the effects of torque limits on learning. Torque limits describe how strong the character is and indirectly decides which movements are possible and which ones are not. The

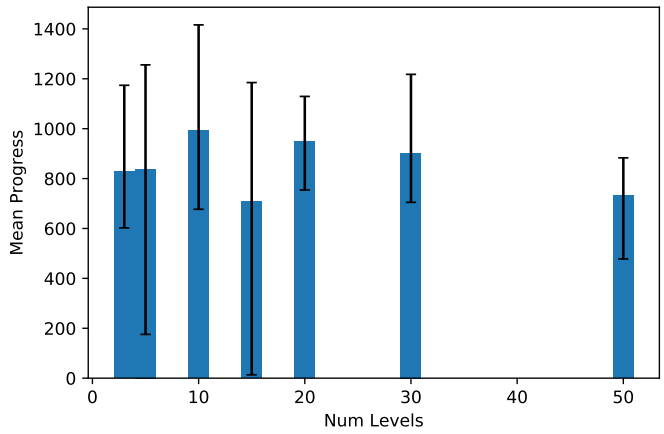


Figure 4.5: Curriculum sensitivity to the number of steps.

robotics and computer graphics communities tend to specify torque limits based on real-life robots and animals, but the machine learning community is less concerned with such details.

The torque limit setting is indeed important, as evident by the decrease in the final performance achieved in different settings. Furthermore, we show that this setting is specifically important in the context of reinforcement learning since with more restrictive configurations the learner tends to get stuck in local optima regions increasingly often. Therefore doing the initial training with higher torque limits is useful for sidestepping local optima and the resulting policy is perhaps more robust as a result of experiencing slightly different versions of the same environment.

Chapter 5

Symmetry Considerations

One obvious path towards faster-and-better learning relies on exploiting the motion symmetry that is a common attribute of human and animal locomotion; gait symmetry is an indicator of healthy outcomes in physiotherapy [29, 30]. Relatedly, while asymmetric gaits are often associated with physical injuries and neural impairments such as stroke. A symmetry constraint or symmetry-favouring bias thus offers a readily available and convenient path towards faster learning and more realistic outcomes. It is also largely orthogonal to most other efficiency improvements.

Naively, exploiting symmetry might be expected to yield a 2× learning speedup, and may help to avoid some of the undesired asymmetric local minima that DRL is prone to exploit. On the other hand, it could also be the case that asymmetric policies and motions serve a useful role as an intermediate path towards finding eventual optimal symmetric motions, and therefore hard symmetry constraints may be problematic. Another important subtlety is that while a symmetric policy helps achieve symmetric motions, it does not guarantee a symmetric outcome. For example, a quadruped gallop and a biped lope are asymmetric gait cycles, as each gait cycle begins with a leading left or right foot, while the underlying policy can still be fully symmetric.

What is the best way to integrate a symmetry bias or other forms of symmetry enforcement into the learning process? How much benefit does it offer in terms of learning speed and learning outcomes? What are other considerations for symmetry-informed methods? The principal contribution of our work is an in-depth analysis of four different methods of incorporating symmetry into the learning process:

DUP Duplicating tuples with their symmetric counterparts.

LOSS Adding a symmetry auxiliary loss.

PHASE Motion phase mirroring.

NET Enforcing symmetry in the network itself.

Two of these methods are new (DUP, NET) and two are already present in existing literature (LOSS, PHASE), albeit without a systematic evaluation of all the issues around symmetry enforcement. The methods incorporate knowledge of symmetry into the policy structure (NET), the learning data (DUP, PHASE), or via the learning loss (LOSS). We also believe that the results are of more general interest because they illustrate (and experimentally validate) various ways that inductive biases can be incorporated into DRL methods.

5.1 Symmetry Enforcement Methods

We now describe four methods for enforcing symmetry, using duplicate tuples, auxiliary losses, a time-indexed motion phase, and architecture-based methods. We begin by formally defining symmetric trajectories and symmetric policies. Two trajectories are symmetric if for each state-action tuple, (s,a), from one trajectory, the corresponding state-action tuple is given by $(\mathcal{M}_s(s), \mathcal{M}_a(a))$ for the other trajectory, where $\mathcal{M}_s$ and $\mathcal{M}_a$ are defined as follows,

$$M_s : \mathcal{S} \rightarrow \mathcal{S} \qquad\qquad M_a : \mathcal{A} \rightarrow \mathcal{A}$$
$$M_s(s) = \text{the mirror of state } s \qquad\qquad M_a(a) = \text{the mirror of action } a$$

Note that the mirroring functions are attributes of the environment and not attributes of the enforcement method or learning pipeline. Here we use *environment* to refer to the combination of the character, its simulated world, and the task, as is common in RL settings. All the symmetry enforcement methods we shall describe require both of these functions as a minimum requirement. Similarly, we can define a symmetric policy to be one where the following holds for all states $s \in \mathcal{S}$:

$$\pi_\theta(M_s(s)) = M_a(\pi_\theta(s)). \tag{5.1}$$

A symmetric policy thus produces the mirrored action when given the mirrored state as input. RL methods such as PPO also use state-value functions during the learning process. The output of these value functions should remain unchanged for any state and its mirrored counterpart. The construction of the mirror functions for our environments (Section 5.3) is further elaborated in Section A.2.

The methods discussed in this section attempt to achieve symmetric gaits by encouraging or constraining the learned policies to be symmetric. However, even if successful, this may be insufficient to guarantee a symmetric gait. In particular, a symmetric policy may learn to favour motions with staggered poses, where the dominant foot is always in front. This may confer advantages concerning balance and agility. Once such a policy is initialized to an initial

asymmetric staggered pose, it can continue with an asymmetric motion. With regard to the policy, it is not always possible to achieve exact symmetry in a parameterized model such as a neural network. For example, regions of the state space may remain unexplored during the learning process, and thus symmetry cannot be enforced for such regions. Therefore, the equality in Equation (5.1) is not always assumed to be strict.

It is possible to directly optimize for gait symmetry with reinforcement learning by including quantitative symmetry measures in the reward function, such as the Symmetry Index [30] or other measures [39]. However, we share the sentiment of previous work [45] that directly optimizing such measures may be ineffective, as they introduce delayed or sparse rewards that may make the learning problem more difficult. Consequently, our work focuses on methods that can be used for obtaining approximately symmetric policies, which are described in the remainder of this section.

5.1.1 Duplicate Tuples (DUP)

This method may be the most intuitive way of achieving symmetry and is a form of data augmentation. In this approach each trajectory tuple is duplicated, mirrored, then added as a valid experience tuple along with the original. More formally, let $\tau = (s_1, a_1, r_1, ..., s_T)$ be a trajectory sampled from the environment. A post-processing step will compute the mirrored trajectory of τ, i.e. $\tau' = (M_s(s_1), M_a(a_1), r_1, ..., M_s(s_T))$, and both τ and τ' will be added to the roll-out memory buffer for learning. Notice that the rewards, $r_1, ..., r_{T-1}$ are the same in both τ and τ'. This is because the reward function $r(s,a)$ is automorphic under the symmetry transformation, namely $r(s,a) = r(\mathcal{M}_s(s), \mathcal{M}_a(a))$.

One drawback of using this approach is that the mirrored tuples are not strictly on-policy, as assumed by policy-gradient RL methods. Thus it could be problematic when used with methods such as PPO [33] and TRPO [32]. The off-policy issue arises because at training time the policy π_θ is not guaranteed to be symmetric, and therefore the probability of sampling action $M_a(a_t)$ from $\pi_\theta(M_s(s_t))$ could be low, effectively corresponding to an off-policy action. However, our results show that this is not necessarily a critical issue in practice.

5.1.2 Auxiliary Loss (LOSS)

In this method proposed by Yu et al.[45], the authors create a symmetry loss defined as follows:

$$L_{sym}(\theta) = \sum_{t=1}^{T} \|\pi_\theta(s_t) - M_a(\pi_\theta(M_s(s_t)))\|^2 \tag{5.2}$$

and optimize this as an auxiliary loss in addition to the default PPO loss:

$$\pi_\theta = \underset{\theta}{\operatorname{argmin}} L_{PPO}(\theta) + wL_{sym}(\theta), \tag{5.3}$$

where w is a scalar hyper-parameter used to balance the gait symmetry loss with the standard policy optimization loss which aims to maximize the original objective. The authors use $w = 4$ for their results. An alternative approach would be to simply include the symmetry loss as an extra reward term. However, the auxiliary loss is generally preferable; the loss term is differentiable and therefore provides a clear signal to optimize rather than being included via the PPO-approximated gradient. Changing the reward function may also induce unexpected behaviours.

Yu et al. [45] showed improvements in the sample efficiency for their four tasks with a factor of approximately two (see Figure 8 in [45]). However, the symmetric loss is shown to be beneficial only in the context of a given curriculum learning algorithm; in its absence, there was no significant improvement over a vanilla-PPO baseline, and in one case (the humanoid) using the symmetric loss proved to be detrimental (please refer to the same plot). The addition of an extra hyper-parameter may generally be seen as undesirable. However, in practice, we find in our experiments that the method is not very sensitive to the choice of w and we end up using the default value in all settings.

5.1.3 Phase-Based Mirroring (PHASE)

To study locomotion, the gait can usually be divided into repeated *gait cycles*, which can then further be parameterized using a phase variable $\phi \in [0, 1)$, which then wraps back to $\phi = 0$ upon reaching $\phi = 1$. A common assumption is to advance the phase linearly with time. Another strategy that can help provide additional robustness is to perform a phase-reset at each bipedal foot strike, e.g., set $\phi = 0$ upon left-foot strike and $\phi = 0.5$ upon right foot strike. To enforce symmetry, a policy is only learned for the first half cycle, and is replaced by the policy with mirrored states-and-actions during the second half cycle:

$$a_t = \begin{cases} \pi_\theta(s_t) & 0 \leq \phi(s_t) < 0.5 \\ M_a(\pi_\theta(M_s(s_t))) & 0.5 \leq \phi(s_t) < 1 \end{cases} \tag{5.4}$$

In our experiments, we strictly advance the motion phase as a function of time and we do not implement phase-resets. For forward-progress tasks, this then corresponds to providing a mandated duration for each half-cycle of the motion. The phase-based method is particularly useful for imitation-guided learning scenarios such as those presented in [27], [28], and [43]. The goal in these cases is to imitate a reference motion capture clip with the help of a phase-indexed reward that measures the distance from the reference motion. The use of the PHASE symmetry in that context is motivated by the potential for faster learning.

The PHASE approach is simple to implement and does not require modifying training in any way since it can be implemented directly within the environment. However, the potential for abrupt changes exists at $\phi = 0.5$ when the phase is strictly computed as a function of time.

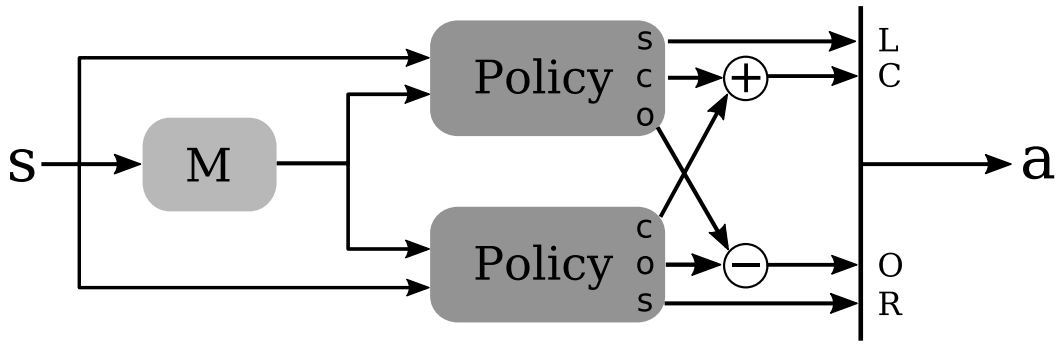


Figure 5.1: A universal method for converting any neural network into a symmetric network policy.
$\mathcal{M}$ block is an environment-dependent state mirroring function. The two policy blocks are the same neural network module, with the output terminals re-order for illustration clarity. The *s, c, o* terminals corresponds to *side*, *common*, and *opposite* joints as described in Section A.2.

5.1.4 Symmetric Network Architecture (NET)

Another approach towards enforcing symmetry is to impose symmetry at the network architecture level. The goal here is to choose a network architecture such that Equation (5.1) holds for all states s and all network parameters θ. There are multiple ways to go about designing such an architecture. However, they may require some knowledge about how the actions and/or states in which case having access to the mirroring functions M_s and M_a is strictly-speaking not enough.

A general case description of this method would be lengthy, and thus we focus only on the key aspects here. The simplest case occurs when we can assume that the action vector is simply divided into two, one corresponding to each side of the body, and that the actions of one side can readily be applied to the other side through a simple swapping operation. This ignores the common parts such as the torso and the head for the time being. More concretely, consider:

$$a = \begin{bmatrix} a_l \\ a_r \end{bmatrix}$$
$$M_a(a) = \begin{bmatrix} a_r \\ a_l \end{bmatrix}$$

where a_l and a_r are vectors of equal size. In this case, we can define a symmetric policy composed of an inner network f as follows:

$$\pi_{side}(s) = \begin{bmatrix} f(s, M_s(s)) \\ f(M_s(s), s) \end{bmatrix}$$

It is easy to show in this case that Equation (5.1) holds:

$$\begin{aligned}
\pi_{side}(M_s(s)) &= \begin{bmatrix} f(M_s(s), M_s(M_s(s))) \\ f(M_s(M_s(s)), M_s(s)) \end{bmatrix} \\
&= \begin{bmatrix} f(M_s(s), s) \\ f(s, M_s(s)) \end{bmatrix} \\
&= M_a \left(\begin{bmatrix} f(s, M_s(s)) \\ f(M_s(s), s) \end{bmatrix} \right) \\
&= M_a \left(\pi_{side}(s) \right)
\end{aligned}$$

When the action space also includes actions for common parts, i.e., those such as the torso and head that have no symmetric counterparts, it is easy to define $\pi_{com}(s) = h(s) + h(M_s(s))$ which is then invariant to left/right mirroring. Finally, the policy is then a combination of the common actions and side actions:

$$\pi_\theta(s) = \begin{bmatrix} \pi_{com}(s) \\ \pi_{side}(s) \end{bmatrix}$$

Please refer to Figure 5.1 for an illustration of the NET method.

A drawback of this method is that it requires knowledge about the state and action symmetry structures to redefine the network. Also, this method is highly sensitive to state and action normalization. The problem is that an ordinary normalization based on past experiences may break the symmetry. Though the other methods introduced here can also suffer from the same problem, this method is much more sensitive to the issue.

5.1.5 Practical Considerations

There are some practical considerations to take into account when working with each of the methods introduced in the previous section. In terms of implementation, the DUP and PHASE methods are the easiest to implement as they required little to no change to the learning pipeline. Architecture-based mirroring (NET) requires the most modification to both the learning pipeline and the environments. The LOSS method is the only approach here that allows us to balance the desire for symmetry with the original learning objective, albeit at the cost of an extra hyper-parameter. The NET method produces a truly symmetric policy, which is not possible with the other methods. The PHASE method is the approach best suited for coping with neutral states, which represent symmetric states where it may become problematic to break symmetry. We revisit this point later. PHASE is also restrictive in that it enforces a predefined walk cycle timing.

One more consideration relates to the application of normalization to network inputs, which is commonly done by using statistics gathered from the data itself. However, this can break some of the mirroring assumptions. The problem is most severe when using a symmetric

network architecture, although other methods are also impacted. Fortunately, developing a normalization scheme that works correctly is relatively straightforward. A simple approach is to duplicate the states (or actions) as in Section 5.1.1 and to compute the statistics based on the aggregated set of states (or actions) and their mirrored states (or actions).

5.2 Gait Symmetry Metrics

All of the methods discussed only provide indirect paths, via the learned policies, for achieving symmetry for the actual motions. Therefore it is important to evaluate how well these methods do at achieving their final goal. Yu et al.[45] uses an established metric in the biomechanics literature known as the Robinson Symmetry Index (SI):

$$SI = \frac{2|X_R - X_L|}{X_L + X_R} \cdot 100, \tag{5.5}$$

where X_R is a scalar features of interest, such as the duration of the stance phase for the right leg, and X_L is its counterpart for the left leg. Previous work using the LOSS method [45] chooses to use the average actuation magnitude as the parameter of interest which leads to $X_R = \sum_{t=1}^{T} \|\tau_{t,R}\|_2$ where $\tau_{t,R}$ is the vector of applied torques at time t for the right leg. We will refer to this as the *actuation symmetry index (ASI)*. In practice, we found that the ASI can be misleading in some circumstances, e.g., a high torque applied to the right hip can be conflated with a high torque applied to the left knee, which is not desirable. ASI also loses information about signs of the applied torques.

The *phase-portrait* is another tool that can be used to qualitatively investigate the symmetry or asymmetry of a gait, as seen in [18]. The phase-portrait is a scatter plot drawn over a period of time, usually over a single gait cycle. The x and y-axes of the 2D plot correspond to the position and velocity, respectively, of a joint of interest, such as the hip flexion, For an asymmetric gait, the phase portraits of the two sides will not fully overlap. To numerically quantify the similarity between two phase-portraits, we propose to use a *phase-portrait index* (PPI). One problem to address is that the left and right limbs usually have a phase offset even for a symmetric motion. This is not a problem when inspecting the phase-portraits visually, but the problem needs to be addressed to compute a meaningful metric. We solve this by finding the best phase offset between the left and the right side through an exhaustive search. We also normalize each axis so that $x, y \in [-1, 1]$ to address the potential discrepancy between magnitudes of different gaits. The final PPI is defined according to:

$$PPI = \frac{1}{C} \min_s \sum_{t=0}^{C-1} \|q_t^R - q_{t+s}^L\|_1 + \|\dot{q}_t^R - \dot{q}_{t+s}^L\|_1, \tag{5.6}$$

where C is the length of a gait cycle, q_t^R and $\dot{q}_t^R$ are the normalized right joint position and velocity at time t. Similarly, q_{t+s}^L is the normalized left joint position at time $t+s$ modulo C, as the elements that are shifted beyond the last position are reintroduced at the beginning.

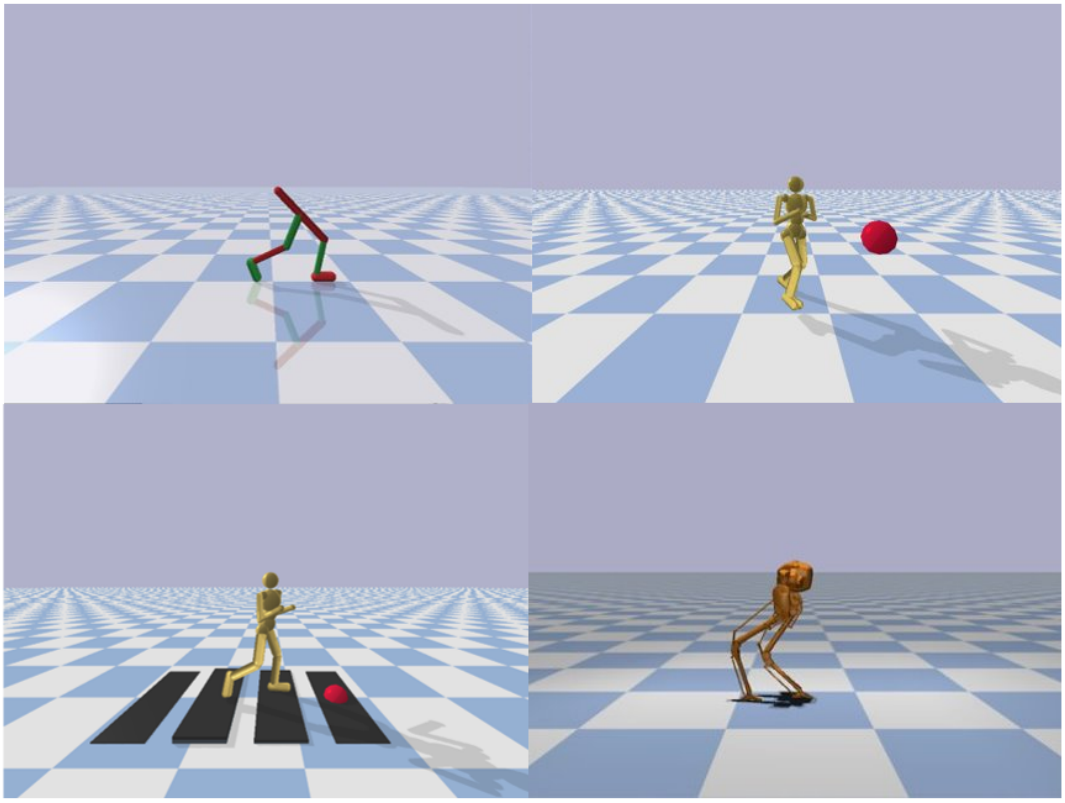


Figure 5.2: Environments.
Top-left: *Walker2D*. Top-right: *Walker3D*. Bottom-left: *Stepper*. Bottom-right: *Cassie*.

5.3 Environments

We evaluate the effectiveness of the enforcement methods described in Section 5.1 on four different locomotion tasks, i.e., RL "environments". The environments were chosen to represent a fairly diverse range of locomotion tasks. They are described in detail below. For each environment, we run each method 5 times and plot the mean results.

Walker2D The implementation of *Walker2D* environment is taken directly from PyBullet [9] without further modification. This character is almost identical to the one used in Section 4.1. The purpose of this environment is to evaluate each symmetry method on a well-established existing reinforcement learning environment. The task is for the character to walk as far as possible in the forward direction in the allotted time. An action is a 6D vector corresponding to a normalized torque at each of the hip, knee, and ankle on both left and right legs. The observation space is 22D and consists of root information (root z-coordinate, x and y heading vector, root velocity, roll, and pitch), joint angles, joint angular velocities, and binary foot contact information.

Walker3D This represents a 3D character simulated in PyBullet, with targets randomly placed, at a distance, in the half-plane in front of the character. The task requires character to navigate towards the target and then stop at the target. A new target will be chosen, in the forward half-plane of the current character orientation, once the target is reached and one second has passed. The 3D character has 21-DoF corresponding to abdomen (x3), hip (x3), knee, ankle, shoulder (x3), and elbow. The observation space is 52D, and is analogous to that provided for *Walker2D*, with an additional 2D vector representing the target location in the character root frame.

Stepper *Stepper* uses the same model as *Walker3D*, and requires it to navigate terrain consisting of a sequence of stepping blocks. The blocks are randomly generated by sampling from the following distributions: spacing $d \sim \mathcal{U}(0.65, 0.85)$ meters and height variation of the next step $h \sim \mathcal{U}(\text{-}25, 25)^\circ$. The character receives information for two upcoming blocks as an (x, y, z) offset in character root space. The stepping block information advances when either foot contacts the immediate next block, which effectively forces the character to step precisely on each step. The precise foot placement requirement, as well as variable terrain height, makes this environment more challenging than *Walker3D*.

Cassie The task requires a bipedal robot Cassie to walk forward at a desired speed while mimicking a reference motion. Since the reference motion is time-indexed, the character receives a phase variable as input. The phase variable varies according to $\phi \in [0, 1)$ in the gait cycle. In addition to phase, the character receives other inputs including the height, the orientation expressed as a unit quaternion, pelvis velocities, angular velocities, and acceleration, joint angles and angular velocities. In total, the Cassie robot has a 10D action space and 47D observation space. Another important distinction between *Cassie* and the other tasks is that it is implemented in MuJoCo [38], while other environments use the Pybullet [9] physics engine. This simulated model has also been validated to be close to the physical Cassie robot [43].

5.4 Results

We compare the four methods, together with an asymmetric baseline, across four different locomotion tasks of varying difficulties[1].

5.4.1 Summary

We begin with a high-level summary of our findings. All symmetry enforcement methods improve motion quality over the baseline, but they cannot be reliably ranked across different environments. In general, DUP is the least effective in enforcing symmetry, while LOSS is the most consistent. For imitation-guided tasks, where the reward is related to imitating a time-

[1]The source code is available at https://github.com/UBCMOCCA/SymmetricRL.

indexed reference motion, such as for *Cassie* and *DeepMimic*, the PHASE method appears to be superior.

Regarding learning speed, the symmetry enforcement methods have no consistent and predictable impact, positive or negative. While this contradicts our initial expectation, it does not provide the full picture. In particular, even though BASE achieves relatively high rewards in *Stepper*, it was unable to make forward progress in any of the five runs. In summary, we suggest symmetry methods be used for producing higher quality symmetric motions, i.e., closer to what we might expect from human and animal movement, but not necessarily for faster learning. We further expand the comparison of the different methods in two sections below.

5.4.2 Effect on Learning Speed

One of our initial hypotheses was that the learning speed can be improved by enforcing symmetry. Symmetry can be considered as domain knowledge that may otherwise be difficult to learn, especially considering its abstract nature. However, our experiments indicated that enforcing symmetry, in general, has no consistent impact on the learning speed. As shown in Figure 5.3, BASE performs well in *Walker2D* and *Walker3D*. In particular, although BASE was not initially the fastest in *Walker3D*, it ultimately achieves a higher return than all mirroring methods. On the other hand, BASE fails to learn the *Stepper* task in all five runs; it often pauses near the beginning without taking a single step. This is consistent with findings by Yu et al., who also find that symmetry enforcement can be crucial when learning more difficult tasks.

For the *Cassie* environment, the benefit of enforcing symmetry is evident because the reward explicitly encourages the character to imitate a symmetric reference motion. We hypothesize that for such a case, symmetry suitably constrains the search space for the symmetric task. However, if symmetry is not rewarded, explicitly or implicitly, then its effects may not be reflected in the learning curve. Finally, between the symmetry methods, there is no clear winner in terms of learning speed.

PHASE and Imitation-Guided Learning In phase-based symmetry experiments, we define a phase variable in correspondence to the gait cycle. For the *Cassie* environment, we use a period of 0.8 s, which is determined based on the reference motion. For all other environments, we assign a period based on a working solution.

We find phase-based symmetry enforcement to be effective for imitation-guided learning, as it outperforms other methods by a significant margin for *Cassie*. When comparing *Cassie* with *DeepMimic* [28], which also uses an imitation objective, we find the results to be consistent. The learning curves for our *DeepMimic* symmetry experiment are presented in Section A.4. We hypothesize that phase-based symmetry is effective for imitation-guided tasks when the motion clips used for training containing suitably-periodic and symmetric motions. On the other hand, PHASE constrains the period of the gait cycle, which can be harmful for non-imitation tasks. PHASE performs poorly in terms of learning speed when used without a reference motion, i.e.,

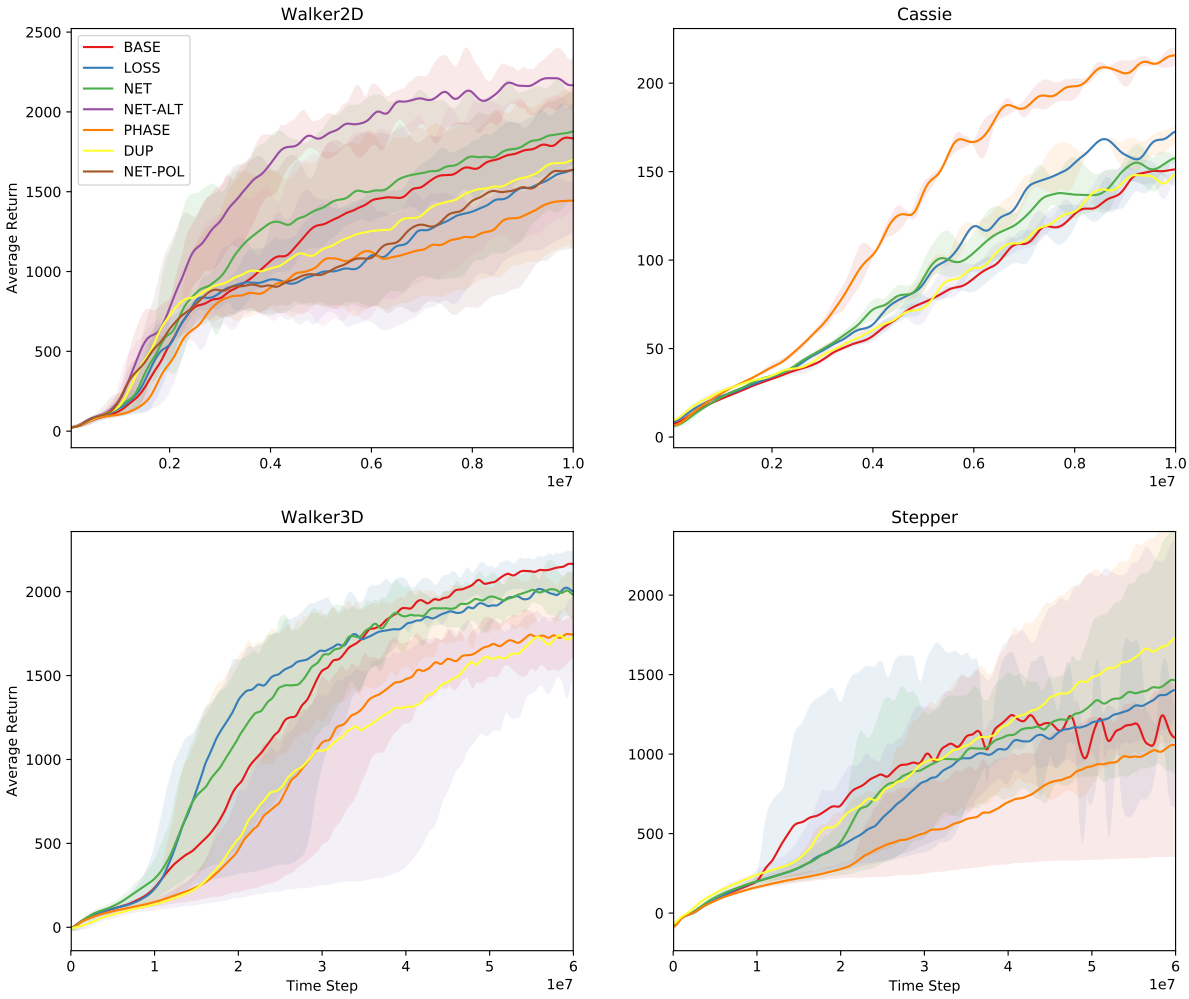


Figure 5.3: Learning curves for different symmetry methods in each of the four locomotion environments (Section 5.3).
The *Walker2D* plot contains two additional experiments aside from the baseline and four symmetry methods. *NET-ALT* uses an alternate formulation of symmetric network architecture described in Section A.3. *NET-POL* is an ablation study on *NET* with symmetry enforcement only on the policy network and not on the value network.

for *Walker2D*, *Walker3D*, and *Stepper*, although it can do well in terms of quality, e.g., for *Walker3D*.

Alternate Symmetric Network The NET method presented in Figure 5.1 is an intuitive way of converting any neural network into a symmetric policy. However, it is perhaps not the immediate solution that one would come up with when tasked to design a symmetric neural network. We include one of our earlier constructions of symmetric policy in Section A.3, which we refer to as NET-ALT. A major difference between NET and NET-ALT is that the latter uses shared weights at the layer level to explicitly enforce the symmetry constraint in Equation (5.1). Despite this, the two architecture-based mirroring methods should, in theory,

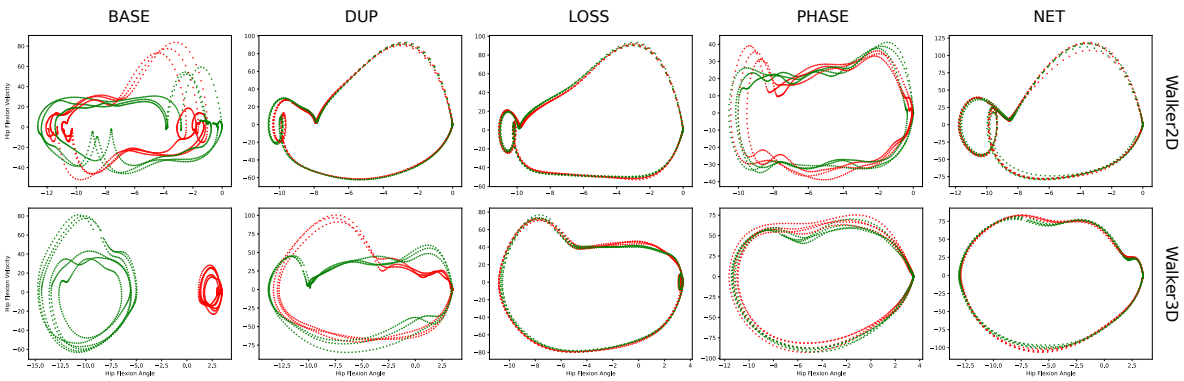


Figure 5.4: Phase-portrait for *Walker2D* and *Walker3D*. The green curve is for the left hip flexion and red for the right side. The more symmetric the motion, the more aligned are the curves.

have similar performance. As can be seen in Figure 5.3, NET-ALT significantly outperforms NET in the *Walker2D* environment, along with the baseline and all other mirroring methods. We believe that the structure of the symmetric layer matrix in Section A.3 may be the key to resolve this gap, which remains to be verified.

Policy Network Ablation Study As an ablation study, we removed the symmetry constraint for the value network in the NET method. Since our goal is to produce a symmetric policy, and the value network is discarded after training, we want to see how enforcing symmetry in the value network during training affects the learning speed. In Figure 5.3, the two curves of interest are NET and NET-POL, where the latter has the symmetry constraint removed for the value network. Our experiment shows that it is beneficial to enforce the symmetry constraint for the value network during training since the difference between the curves is not insignificant.

5.4.3 Symmetry Enforcement Effectiveness

Although learning speed is a major point of interest from the ML perspective, our work is nevertheless motivated by the aesthetics of symmetric gaits that are needed for applications in animation. We measure the effectiveness of each symmetry enforcement methods on the metrics we defined in Section 5.2. In most cases, we find that symmetric gaits are better achieved when any of the enforcement methods are applied, as compared to the baseline. The motions produced by the symmetry methods are also more natural-looking, subjectively speaking, than without mirroring.

Figure 5.4 shows the phase-portraits for *Walker2D* and *Walker3D*. The symmetry metrics for all environments are summarized in Table 5.1 and Table 5.2. To perform consistent measurement for the metrics, we omit the first two strides to limit the influence of the transition period from standing to locomotion. The reported metrics are calculated from the median of the ten subsequent strides after the initial two. For the *Stepper* tasks, we use the median from five

strides to accommodate for the increased difficulty. Also, note the *Stepper* results are missing for BASE because it was unable to produce consistent gait cycles that can be measured. In most cases, the policy either learns to pause at the starting location or falls after taking one or two steps.

As in learning speed, there is not a single best mirroring method across all environments. However, from the overall picture, we found that LOSS and PHASE to be the most consistent among all methods. In general ASI and PPI do not agree on a single best method except for the *Cassie* task where PHASE is the best.

	Walker2D	Walker3D	Stepper	Cassie
BASE	*3.97*	6.36	✗	9.27
DUP	3.77	7.57	7.54	6.58
LOSS	2.56	4.48	6.36	*15.72*
PHASE	3.77	**2.55**	**3.99**	**4.49**
NET	2.00	*10.64*	*28.97*	5.15
NET-ALT	**1.04**	–	–	–
NET-POL	1.71	–	–	–

Table 5.1: Actuation SI. Lower numbers are better.

	Walker2D	Walker3D	Stepper	Cassie
BASE	*1.06*	*2.16*	✗	*0.49*
DUP	0.39	1.61	0.57	0.41
LOSS	0.33	**0.19**	**0.46**	0.31
PHASE	0.57	0.30	0.49	**0.17**
NET	**0.16**	0.58	*0.65*	0.23
NET-ALT	**0.16**	–	–	–
NET-POL	0.28	–	–	–

Table 5.2: Phase-portrait index. Lower numbers are better.

5.5 Discussion

Symmetry can sometimes be harmful, especially when the character begins from or otherwise arrives at a neutral pose, i.e., a symmetric pose where $s = M_s(s)$. The problem is that a symmetric policy is incapable of escaping from a neutral pose since the action that it takes would also be symmetric. When a symmetric action is applied in a symmetric state, the next state is necessarily also symmetric. For instance, a character that starts from the T-pose will likely perform some kind of hopping gait, since the feasible locomotion possibilities which perpetuate symmetric states and actions are limited. To make matters worse, states near the neutral states can also become problematic.

The breaking symmetry problem is most severe when enforcing symmetry through network architecture, as this method is guaranteed to produce true symmetric policies. While DUP and

LOSS methods can suffer from the same issue, they can implement workarounds at an additional cost. This issue, however, does not affect PHASE. A simple workaround to this problem is to always start the character from a non-neutral position. This can be easily achieved by adding some random noise to each joint of the initial pose at the start of the task. In practice, we did notice that on occasion the character would converge on a hopping gait. However, the simple workaround works well for the majority of cases in our experiments.

Our work is motivated by the premise that healthy human gaits are usually symmetric. However, this remains a controversial issue in the biomedical literature [29, 31]. The strongest argument for asymmetry in human motor control is the general belief that humans have a dominant side that is often the preferred choice for manipulating objects. This is also tied with the need for a leading foot to start a walk or run cycle in the neutral state problem. One should, therefore, be aware of the implications when enforcing perfect symmetry. Quadrupedal locomotion, which has six commonly observed gaits as opposed to the three gaits of bipeds [26], is also interesting to examine. Of these six, half are fairly symmetric including walk, trot, and rack. However, the remaining three, also known as the in-phase gaits which are used at high speeds, are often asymmetric. Since the symmetry of gait and policy are not the same, it would be interesting to see whether it is possible to nevertheless achieve these non-symmetric quadrupedal gaits with a symmetric policy.

5.6 Conclusions

In this chapter, we explore the use of symmetry constraints for DRL-based learning of locomotion skills. We compared four different enforcement methods, in addition to a symmetry-free baseline, across four different locomotion tasks of varying difficulty. We find that enforcing symmetry constraints can sometimes be harmful to learning efficiency, but that in general, it produces higher quality motions. When comparing the symmetry methods, we find that the results, both in terms of learning speed and motion symmetry, to be environment-dependent. A notable exception is that the phase-based mirroring method generally performs better than the baseline for imitation-guided reward settings such as for *Cassie* and *DeepMimic*.

The difference between the enforcement methods is more pronounced from the implementation standpoint. LOSS and PHASE methods have the burden of an additional hyperparameter to tune. However, the additional parameter can also be viewed as an advantage in terms of flexibility. In LOSS, the hyperparameter can be used to adjust the strength of symmetry constraint. For PHASE, the phase variable allows us to define a desired locomotion period. Given the similarities across all methods, it is perhaps justifiable to choose one based on the implementation overhead. DUP is the easiest to implement and evaluate since it requires minimal change to the existing RL pipeline and has no hyperparameter to tune. Finally, if the application requires absolute symmetry, then the NET method is guaranteed to produce a symmetric policy.

The application of symmetric policies is not limited to locomotion. Many classical control

tasks may benefit significantly from leveraging symmetry, including acrobat, cart-pole, and pendulum [4]. Furthermore, the notion of symmetry extends beyond left-right symmetry and even character motion. The Sudoku game is an example task that exhibits multiple types of symmetry properties. Whether a learning method can take full advantage of all the symmetries remains an open question. However, this paper lays a foundation for enabling future studies on inductive biases based on symmetry.

Chapter 6

Conclusions

Reinforcement learning provides a promising new path for motion generation, one that can be generalized to new terrains and character morphologies. However the current methods are computationally inefficient and unless motion captured data is used, the motion quality is typically unsatisfactory for applications in computer graphics.

Our work tackles these two problems by investigating two issues: excessively large torque limits and gait asymmetry. We show that more realistic torque limits, though resulting in more natural motions, can hinder the training in the beginning. We propose to use a simple curriculum learning technique that starts with higher torque limits to speed up the training but gradually decreases the limits to arrive at more natural final motions. This way we get the best of both worlds.

Next, we looked at ways of incorporating gait symmetry into the training process. Symmetric motions are generally perceived to be more attractive in humans and asymmetric patterns are commonly associated with disability or injury. We have compared four methods of enforcing symmetry in various environments as well as discussing their advantages and drawbacks in different scenarios.

As with any other work, there remain some questions to be answered. I would be interested in using the torque limit curriculum approach to figure out what is the lowest torque limit that still permits locomotion in a robot, such as Cassie. This can help with designing cheaper robots.

Another promising direction is to look at the time-step used for locomotion. In all of our work, we used a prespecified control frequency. Humans, however, tend to plan at different time horizons depending on the task and their skill level. We have promising initial results showing that using curriculum learning to transfer skills learned with a specific control frequency to another can improve performance. Perhaps the reason for this is that the agent can better focus on the long-term goals while still having the opportunity to refine its movements. This can help bridge the gap between low-level skill learning and long-term planning. This idea is also closely related to hierarchical reinforcement learning.

Bibliography

[1] W. W. Armstrong and M. W. Green. The dynamics of articulated rigid bodies for purposes of animation. *The Visual Computer*, 1(4):231–240, Dec 1985. ISSN 1432-2315. doi:10.1007/BF02021812. URL https://doi.org/10.1007/BF02021812. → page 1

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017. URL http://arxiv.org/abs/1708.05866. → page 7

[3] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1. doi:10.1145/1553374.1553380. URL http://doi.acm.org/10.1145/1553374.1553380. → page 15

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. → pages 5, 34

[5] W. M. Brown, L. Cronk, K. Grochow, A. Jacobson, C. K. Liu, Z. Popović, and R. Trivers. Dance reveals symmetry especially in young men. *Nature*, 438(7071):1148, 2005. → pages 3, 4

[6] A. Bruderlin and T. W. Calvert. Goal-directed, dynamic animation of human walking. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 233–242. ACM, 1989. → page 5

[7] S. Coros, P. Beaudoin, and M. van de Panne. Generalized biped walking control. *ACM Transctions on Graphics*, 29(4):Article 130, 2010. → page 5

[8] 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, 2011. → page 5

[9] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2019. → pages 12, 27, 28

[10] K. S. I. A. A. H. A. G. T. H. L. B. M. L. A. B. Y. C. T. L. F. H. L. S. G. v. d. D. T. G. D. H. David Silver, Julian Schrittwieser. Mastering the game of go without human knowledge. *MNature*, 529, 2016. URL https://doi.org/10.1038/nature16961. → page 11

[11] 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. → page 5

[12] N. Hansen. The cma evolution strategy: A comparing review. In *Towards a New Evolutionary Computation*, pages 75–102, 2006. → pages 1, 5

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

[14] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017. URL http://arxiv.org/abs/1709.06560. → pages 13, 14

[15] 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. → page 5

[16] D. Holden, T. Komura, and J. Saito. Phase-functioned neural networks for character control. *ACM Trans. Graph.*, 36(4):42:1–42:13, July 2017. ISSN 0730-0301. doi:10.1145/3072959.3073663. URL http://doi.acm.org/10.1145/3072959.3073663. → page 5

[17] D. Holden, T. Komura, and J. Saito. Phase-functioned neural networks for character control. *ACM Trans. Graph.*, 36(4):42:1–42:13, July 2017. ISSN 0730-0301. doi:10.1145/3072959.3073663. URL http://doi.acm.org/10.1145/3072959.3073663. → page 4

[18] E. T. Hsiao-Wecksler, J. D. Polk, K. S. Rosengren, J. J. Sosnoff, and S. Hong. A review of new analytic techniques for quantifying symmetry in locomotion. *Symmetry*, 2: 1135–1155, 2010. → pages 4, 26

[19] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Are deep policy gradient algorithms truly policy gradient algorithms? *CoRR*, abs/1811.02553, 2018. URL http://arxiv.org/abs/1811.02553. → page 10

[20] O. Klimov and J. Schulman. Roboschool, open-source software for robot simulation. https://github.com/openai/roboschool, 2017. → page 12

[21] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 473–482, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi:10.1145/566570.566605. URL http://doi.acm.org/10.1145/566570.566605. → page 4

[22] Y. Lee, S. Kim, and J. Lee. Data-driven biped control. *ACM Transctions on Graphics*, 29 (4):Article 129, 2010. → page 5

[23] T. P. Lillicrap, J. J. Hunt, A. e. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, art. arXiv:1509.02971, Sep 2015. → page 7

[24] L. Liu, M. van de Panne, and K. Yin. Guided learning of control graphs for physics-based characters. *ACM Transactions on Graphics*, 35(3), 2016. → page 5

[25] A. Majkowska and P. Faloutsos. Flipping with physics: motion editing for acrobatics. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 35–44. Eurographics Association, 2007. → page 5

[26] T. A. McMahon. *Muscles, Reflexes, and Locomotion*. Princeton University Press, Princeton, New Jersey, 1 edition, 1984. ISBN 069102376X. → page 33

[27] X. B. Peng, G. Berseth, K. Yin, and M. van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017. → pages 5, 23

[28] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (Proc. SIGGRAPH 2018)*, 37(4), 2018. → pages 5, 23, 29, 42

[29] J. L. Riskowski, T. J. Hagedorn, A. B. Dufour, V. A. Casey, and M. T. Hannan. Evaluating gait symmetry and leg dominance during walking in healthy older adults. *Institute of Aging Research, Hebrew SeniorLife, Boston, Usa, Harvad Medical School, USA, School of Public Health, Boston University, Boston*, 2011. → pages 4, 20, 33

[30] R. Robinson, W. Herzog, and B. Nigg. Use of force platform variables to quantify the effects of chiropractic manipulation on gait symmetry. *Journal of manipulative and physiological therapeutics*, 10(4):172–176, 1987. → pages 3, 4, 20, 22

[31] H. Sadeghi, P. Allard, F. Prince, and H. Labelle. Symmetry and limb dominance in able-bodied gait: a review. *Gait & Posture*, 12(1):34–45, 2000. ISSN 0966-6362. doi:https://doi.org/10.1016/S0966-6362(00)00070-9. URL http://www.sciencedirect.com/science/article/pii/S0966636200000709. → pages 4, 33

[32] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477. → pages 9, 22

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347. → pages 9, 13, 22

[34] M. K. Seeley, B. R. Umberger, and R. Shapiro. A test of the functional asymmetry hypothesis in walking. *Gait & posture*, 28(1):24–28, 2008. → page 4

[35] A. Shapiro. Monte carlo sampling methods. In *Stochastic Programming*, volume 10 of *Handbooks in Operations Research and Management Science*, pages 353 – 425. Elsevier, 2003. doi:https://doi.org/10.1016/S0927-0507(03)10006-0. URL http://www.sciencedirect.com/science/article/pii/S0927050703100060. → page 8

[36] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition, 2018. → pages 6, 7

[37] G. Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, Mar. 1995. ISSN 0001-0782. doi:10.1145/203330.203343. URL http://doi.acm.org/10.1145/203330.203343. → page 11

[38] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi:10.1109/IROS.2012.6386109. → page 28

[39] S. Viteckova, P. Kutilek, Z. Svoboda, R. Krupicka, J. Kauler, and Z. Szabo. Gait symmetry measures: A review of current and prospective methods. *Biomedical Signal Processing and Control*, 42:89–100, 2018. ISSN 1746-8094. doi:https://doi.org/10.1016/j.bspc.2018.01.013. URL http://www.sciencedirect.com/science/article/pii/S1746809418300193. → pages 4, 22

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

[41] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. → pages 7, 9

[42] A. Witkin and M. Kass. Spacetime constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 159–168, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6. doi:10.1145/54852.378507. URL http://doi.acm.org/10.1145/54852.378507. → page 1

[43] Z. Xie, P. Clary, J. Dao, P. Morais, J. W. Hurst, and M. van de Panne. Iterative reinforcement learning based design of dynamic locomotion skills for cassie. *CoRR*, abs/1903.09537, 2019. URL http://arxiv.org/abs/1903.09537. → pages 23, 28

[44] K. Yin, K. Loken, and M. van de Panne. Simbicon: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3):Article 105, 2007. → pages 1, 5

[45] W. Yu, G. Turk, and C. K. Liu. Learning symmetric and low-energy locomotion. *ACM Transactions on Graphics (Proc. SIGGRAPH 2018 - to appear)*, 37(4), 2018. → pages 2, 5, 12, 22, 23, 26, 29

[46] H. Zhang, S. Starke, T. Komura, and J. Saito. Mode-adaptive neural networks for quadruped motion control. *ACM Trans. Graph.*, 37(4):145:1–145:11, July 2018. ISSN 0730-0301. doi:10.1145/3197517.3201366. URL http://doi.acm.org/10.1145/3197517.3201366. → page 4

[titletoc]

Appendix

Supporting Materials

A.1 Chapter 4 Hyper-parameters

The following contains the hyper-parameters used for all of the experiments in Chapter 4. The code is also available under https://github.com/farzadab/walking-benchmark/.

Name	Description
Policy LR	$3e^{-3}$
Value Network LR	$5e^{-3}$
Total Environment Steps	6 Million
Clipping Parameter (ϵ)	0.2
Decay (γ)	0.99
GAE λ	0.95
Policy type	Fixed Diagonal Gaussian
Policy stdev	$\exp(-1)$
Network Size	3 Layers of 256 nodes
Network Activations	ReLU
Max Grad Norm	1
Optimization Batch Size	128
PPO Optimization Steps	10

Table 1: Hyper-parameters used in Chapter 4.

A.2 Mirroring Functions

The mirroring functions, $\mathcal{M}_s$ and $\mathcal{M}_a$ as described in Section 5.1 are properties of the environment. Consequently, the environment is responsible for providing the necessary information for policies to perform the mirroring operation on state and action. Although the mirroring functions can be arbitrarily complex, we found that all the environments in Section 5.3 share a similar construction. Using *Walker3D* as an example, the method for deriving mirror functions are described in detail below.

The *Walker3D* character has a total of 21-DoF and each DoF is modelled as a one-dimensional hinge joint. Furthermore, let the x-axis be the forward direction and the z-axis pointing up in the local coordinate frame of the character. For mirroring purposes, the joints can be divided into three categories, common, opposite, and side. The common categories contain joints that are unchanged by the mirroring function, such as $abdomen_y$. In general, joints that rotate about the y-axis should remain unchanged after mirroring. The opposite categories contain joints that are mainly on the torso of the character and they need to be negated for mirroring. In the case of *Walker3D*, only $abdomen_x$ and $abdomen_z$ would fall under this category. The side categories contain joints that are on the limbs. Importantly, for each joint on one side, there must be a corresponding joint on the other side; for instance, the right knee corresponds to the left knee. With the one-to-one mapping, $\mathcal{M}_a$ can simply interchange the applied torques for the respective joints on either side. We found that it is more straightforward if the joint rotation axes are flipped, except for axes aligned on the y-axis, for the left and right limbs. Otherwise, additional negation operations need to be applied after interchanging left and right actions.

$\mathcal{M}_s$ follows a similar pattern as described above. For state information that is derived from the character, such as joint angles and angular velocities, the mirrored counterpart would have negated and interchanged values. Besides, the environment may provide additional information, such as character orientation, velocity, and target location in character root space, as in *Walker3D*. For vector-valued information, such as velocity and target location, the values along the y-axis should be negated; for orientations, values representing roll and yaw should be negated.

A.3 Alternate Symmetric Network Architecture

In Figure 5.1, we presented a universal method for embedding any neural network into a symmetric policy. The NET method effectively uses the same policy module twice with flipped inputs for s and $M(s)$. While this construction is relatively simple to implement, alternative symmetric policy constructions do exist. In this section, we describe the construction used for NET-ALT in Figure 5.3.

Recall that a symmetric policy is one that satisfies Equation (5.1), along with the fact that our mirror functions (Section A.2) essentially perform negation and swapping operation on the state and action vectors. Let us then consider the individual layers of a neural network as matrix operations, in particular, before the application of non-linear activation functions. The full matrix form of the first layer for s and $\mathcal{M}_s(s)$ can be written as,

$$\begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix} = (a_{ij})_{4\times 4} \begin{bmatrix} C \\ O \\ R \\ L \end{bmatrix} \text{ and, } \begin{bmatrix} W \\ -X \\ Z \\ Y \end{bmatrix} = (a_{ij})_{4\times 4} \begin{bmatrix} C \\ -O \\ L \\ R \end{bmatrix}.$$

C, O, R, L represent the portions of the state vector corresponding to common, opposite, right, and left respectively. The uppercase letters for C, O, R, L, W, X, Y, and Z indicate that these are not necessarily scalars. For instance, for *Walker3D*, O contains both $abdomen_x$ and $abdomen_z$. Similarly, the matrix, $(a_{ij})_{4\times4}$, is dimensionally consistent with the corresponding elements in the state vector. For example, a_{2j} is a two-column wide block that matches with the two elements in O for *Walker3D*. In addition, notice the negated O and X, as well as the interchanged R and L are the effect of the mirroring functions. Overall, there are a total of 16 unknowns and 8 equations. A symmetric layer can be obtained by solving this system of equations. In particular, *NetAlt* contains symmetric layers of the following form,

$$(a_{ij})_{4\times4} = \begin{bmatrix} \alpha & 0 & \beta & \beta \\ 0 & \gamma & \beta & -\beta \\ \delta & \epsilon & \zeta & \eta \\ \delta & -\epsilon & \eta & \zeta \end{bmatrix}.$$

To maintain the symmetric policy constraint, the activation function applied to the negation portion, O, must be an odd function such as *tanh* or *softsign*. A similar procedure can be followed for intermediate and output layers, as long as the sizes for each of the portions are correctly maintained. Finally, a symmetric policy network can be constructed by stacking symmetric layers.

A.4 Symmetry in DeepMimic Environment

To evaluate the effectiveness of phase-based mirroring, we ran an experiment for the original DeepMimic environment [28] in additional to the *Cassie* environment. In both cases, our data shows that phase-based mirroring does indeed make the learning faster. However, in the case of *DeepMimic*, the difference in final return is small between BASE and PHASE and only a minor difference can be seen from the video.

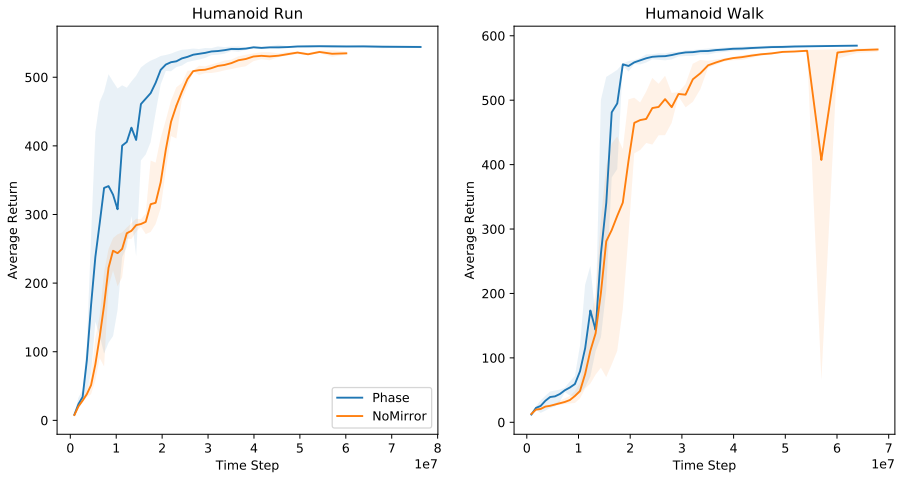


Figure 1: Learning curves for the original *DeepMimic* environment. BASE and *Phase* corresponds to the symmetry enforcement methods in Figure 5.3