Reinforcement Learning Based Mechanical Design and Control of Flexible-Legged Jumping Robots

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Masters of Science

Andrew Albright

Spring 2022

Reinforcement Learning Based Mechanical Design and Control of Flexible-Legged Jumping Robots

Andrew Albright

APPROVED:

Joshua E. Vaughan, Chair Assistant Professor of Mechanical Engineering	Anthony S. Maida Associate Professor of Computer Science
Alan A. Barhorst Department Head of Mechanical Engineering	Mary Farmer-Kaiser Dean of the Graduate School

To all the poor souls using Word, one day you will see the light that is LaTeX.

"Before we work on artificial intelligence why don't we do something about natural stupidity?"

— Steve Polyak

Acknowledgments

I would like to firstly thank my advisor Dr. Joshua Vaughan for his leadership, both academic and personal, during my time here at the university. His support has been invaluable in helping me to understand the underlying ideas behind this material. Additionally, his constant drive to produce high quality material has kept me motivated in my attempts to do the same. I would not be here without him.

Additionally, I would like to thank my committee members, Dr. Alan Barhorst, Dr. Anthony Maida and Dr. Brian Post for their input in regards to this work. Along with my lab members during my time in the C.R.A.W.L.A.B., Gerald Eaglin, Adam Smith, Y (Eve) Dang, Brennan Moeller and Darcy LaFont. Their conversations and advice have greatly assisted me in my efforts to complete this work.

Lastly, I would like to thank the Louisiana Crawfish Board for their financial support in the form of a grant from the University of Louisiana at Lafayette. This grant has allowed me to work knowing my fiscal security was in tack.

Table of Contents

List of Tables

List of Figures

I Introduction and System Description

A legged locomotive robot can have many advantages over a wheeled or tracked one, particularly in regards to their ability to navigate uneven and unpredictable terrain [3, 4]. They can achieve this advantage because of the numerous movement types they can deploy. Abilities such as independently placing their feet within highly rigid terrain and jumping or bounding over obstacles have been shown to be effective ways of locomoting [5]. These advantages do not come at no cost, however. Legged systems are traditionally power inefficient compared to wheeled vehicles making them a less attractive option for applications where power conservation is required. Research has been conducted showing the usefulness of adding flexible components, like the legs seen on the robot in Figure 1, for combating efficiency and other issues [4, 6, 7]. The addition of these components in legged robots has been shown to increase system performance measures such as running speed, jumping capability, and power efficiency [8]. However, the addition of flexible components creates a system that is highly nonlinear, and thus requires a more complex control system.

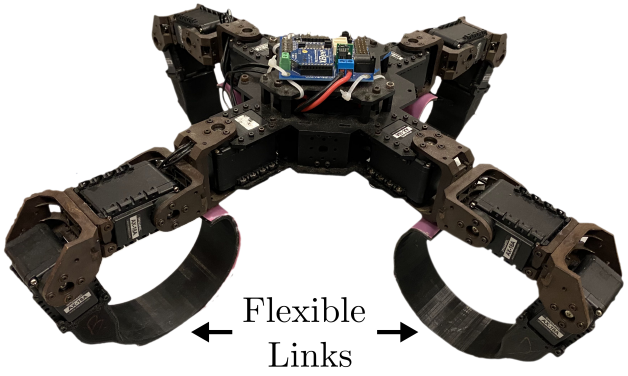


Figure 1. Flexible Robotics System

1.1 Improving Performance with Flexible Components

The use of flexible components within robotic systems has been shown to be an effective way of improving performance metrics such as movement velocity and power efficiency [4, 8]. Of the different techniques that have be deployed, the use of series elastic actuators (SEAs) has been shown to be an effective for increasing energy efficiency [9, 10]. Storing energy in the non-rigid parts of motor joints, such as the elastic element seen in Figure 2, have proven to be an effective way in increasing efficiency. The addition of flexible joints is not the only technique that has been used to improve performance, however; utilizing tendon like elastic members to connect actuators to links has also been shown to be an effective way of improving efficiency [1]. The use of tendons, being an example of replicating what is found in nature, is a common method of finding unique mechanical designs that perform well in the real world. An example of this type of design can be seen in Figure 3. Following a similar idea, research has also been conducted finding the usefulness of including flexibility in the the spine of of 2D running robots where the velocity of the robot was drastically increased [11]. Research studying the effects of flexible links, like the ones shown in Figure 1 is limited though, particularly in the realm of legged-robots. Still, it has been shown as a viable method of increasing performance in these types of robots [12].

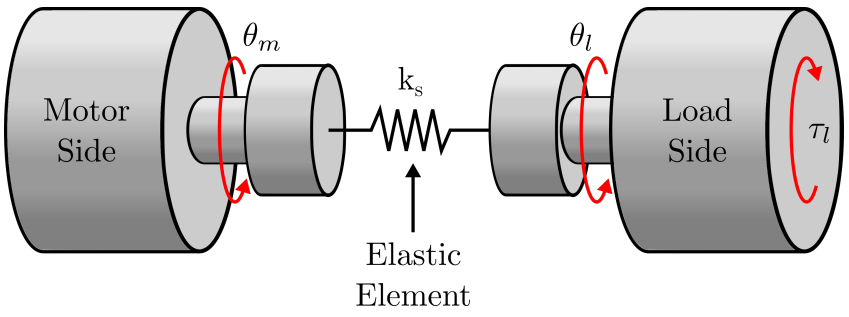


Figure 2. Rotary Style Series Elastic Actuator

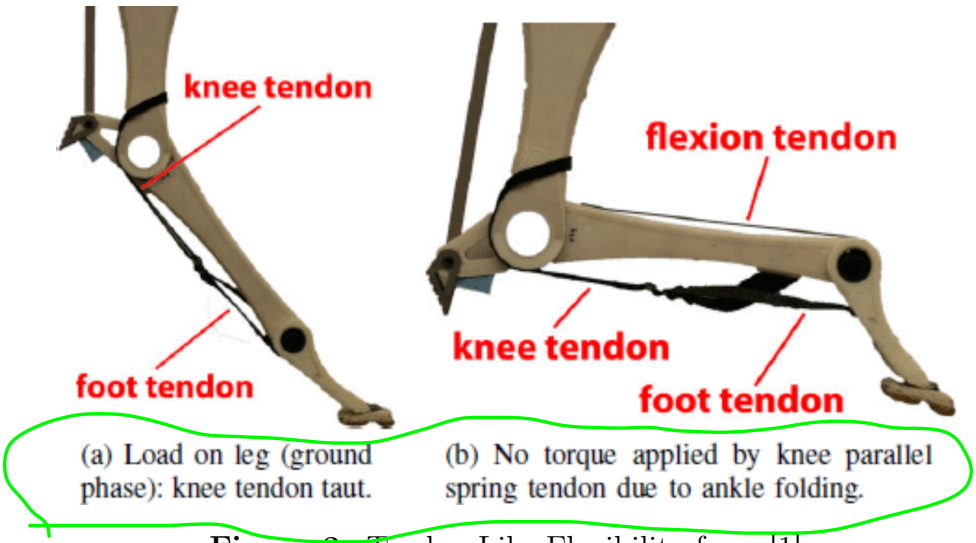


Figure 3. Tendon Like Flexibility from [1]

1.2 Controlling Flexible Systems

Control methods developed for flexible systems have been shown to be effective for position control and vibration reduction [10, 13]. Because of the challenges seen in scaling the controllers, methods utilizing reinforcement learning are of interest. This method has been used in simple planar cases, where it was compared to a PD control strategy for vibration suppression and proved to be a higher performing method [14]. Additionally, it has also been shown to be effective at defining control strategies for flexible-legged locomotion. The use of actor-critic algorithms such as Deep Deterministic Policy Gradient [15] have been used to train running strategies for a flexible legged quadruped [16]. Much of the research is based in simulation, however, and often the controllers are not deployed on physical systems, which leads to the question of whether or not these are useful techniques in practice.

1.3 Concurrent Design

Defining an optimal controller for a system can be difficult due to challenges such as mechanical and electrical design limits. This is especially true when the system is flexible and the model is nonlinear. A solution to this challenge is to concurrently

design a system with the controllers so that the two are jointly optimized. Defining the design process such that the robot's design results in a simple dynamic model has been shown to improve the performance of mechatronics systems [17]. Additionally, in more recent work, the utilization of more complex deep learning methods have shown to be an effective strategy for finding optimal concurrent designs [18]. Deep learning has been used to find concurrent designs for simulated legged robotic systems leading to improved performance in regards to movement velocity [19]. Some research has even been completed where the designs were deployed on physical hardware, validating that this area of research is an effective one for learning how best to define system/controller architectures [20]. Little research exists however, utilizing these techniques on legged-robotic systems, particularly ones ~~that are flexible in nature.~~

1.4 Reinforcement Learning

With the recent successes seen in utilizing reinforcement learning (RL) to define control strategies, design parameters and concurrent designs for robot systems, it is of interest to apply this technique in a unique way to flexible-legged jumping systems. Firstly, it is important to understand the generalities regarding a reinforcement learning problem.

Reinforcement Learning is the process of training a policy to define a series of commands using an environment where those commands can be applied. A policy, often referred to as an agent, from a controls theory perspective, is synonymous with a controller. The environment the controller is deployed in, again from a controls theory perspective, is synonymous with a robotic system. Training the controller requires iteratively deploying the controller's commands, or actions, to the the environment and observing the results. The results are often in the form of the state of the environment and a reward resulting from the action that was applied. The reward is defined by the designer so that the controller is trained to accomplish a desired task. Other than the

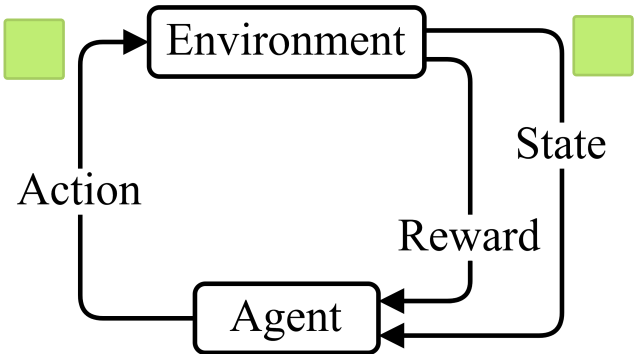


Figure 4. Reinforcement Learning Process

reward, the controller has no way to decern what commands are good when the environment is in some state. The iterative learning process is often described using the block diagram shown in Figure 4.

Learning an optimal control strategy is completed by deploying a gradient-decent-based learning algorithm utilizing information such as the state of the environment and the reward. For general robotics applications, at each discrete time step t, the environment will be in a state $s \in \mathcal{S}$, and the controller will select an action $a \in \mathcal{A}$ according to the current policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ and apply said action within the environment. The environment will transition to a new state s' and will generate a reward r based on the user's definition. The return, being the value the algorithm is trying to optimize, is defined as a discounted sum of rewards, $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, where γ is a discount factor for assigning the level of importance for near-term or long-term rewards.

The challenge of an RL algorithm is to optimize a policy, π_ϕ, with parameters, ϕ, such that actions generated at each time step will maximize the return. Ultimately, an optimized policy will maximize the expected return, $J(\phi) = \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi}[R_0]$.

1.5 Twin Delayed Deep Deterministic Policy Gradient

There are many algorithms used to train a neural-network-based controller in an RL application, some of which have shown their ability to learn high-performing control

strategies for robotic systems [21–23]. Of the different algorithms used in research today, the one selected and tested in this work is Twin Delayed Deep Deterministic Policy Gradient (TD3) [24]. This is an actor-critic type learning algorithm which is widely considered the predecessor to the popular and proven Deep Deterministic Policy Gradient (DDPG) algorithm [15].

Figure 5 displays the flow of information for this algorithm. In general, this algorithm learns both a Q-function and a policy, being the *critic* and the *actor*. For algorithms such as TD3, the ultimate goal is to find a policy, π_θ, which maximizes the expected return:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s\sim p_\pi}\left[\nabla_a Q^\pi(s,a)|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)\right] \tag{1}$$

where $Q^\pi(s,a) = \mathbb{E}_{s_i\sim p_\pi, a_i\sim\pi}[R_t|s,a]$ is the Q-function (sometimes called the value function) and the critic in the case of the TD3 architecture. This function is based on the Bellman Equation [25] and returns a numerical value from being in a state s, taking action a and following policy π from there after:

$$Q^\pi(s,a) = \underset{s'\sim P}{E}\left[r(s,a) + \gamma \max_{a'} Q^\pi(s',a')\right] \tag{2}$$

Using a differentiable function approximator, $Q^\pi(s,a)$ can be represented and estimated by $Q_\phi(s,a)$, with parameters ϕ [26]. Updating the Q-function is accomplished using the temporal difference error between the Q-function and a target Q-function [27, 28]. To maintain a fixed objective over multiple policy updates, the target Q-function approximator is instantiated separately as $Q_{\phi_{targ}}(s,a)$. The target does depend on the same parameters that are being trained, ϕ, so there exists an issue when trying to use it as a target. To solve this issue the target network is updated at a delayed pace following the main Q-function approximator by either matching the parameters or by polyak averaging, $\phi_{targ} \leftarrow \tau\phi + (1-\tau)\phi_{targ}$, where τ is a tunable hyperparameter.

In summary, the critic side of the TD3 algorithm is responsible for minimizing

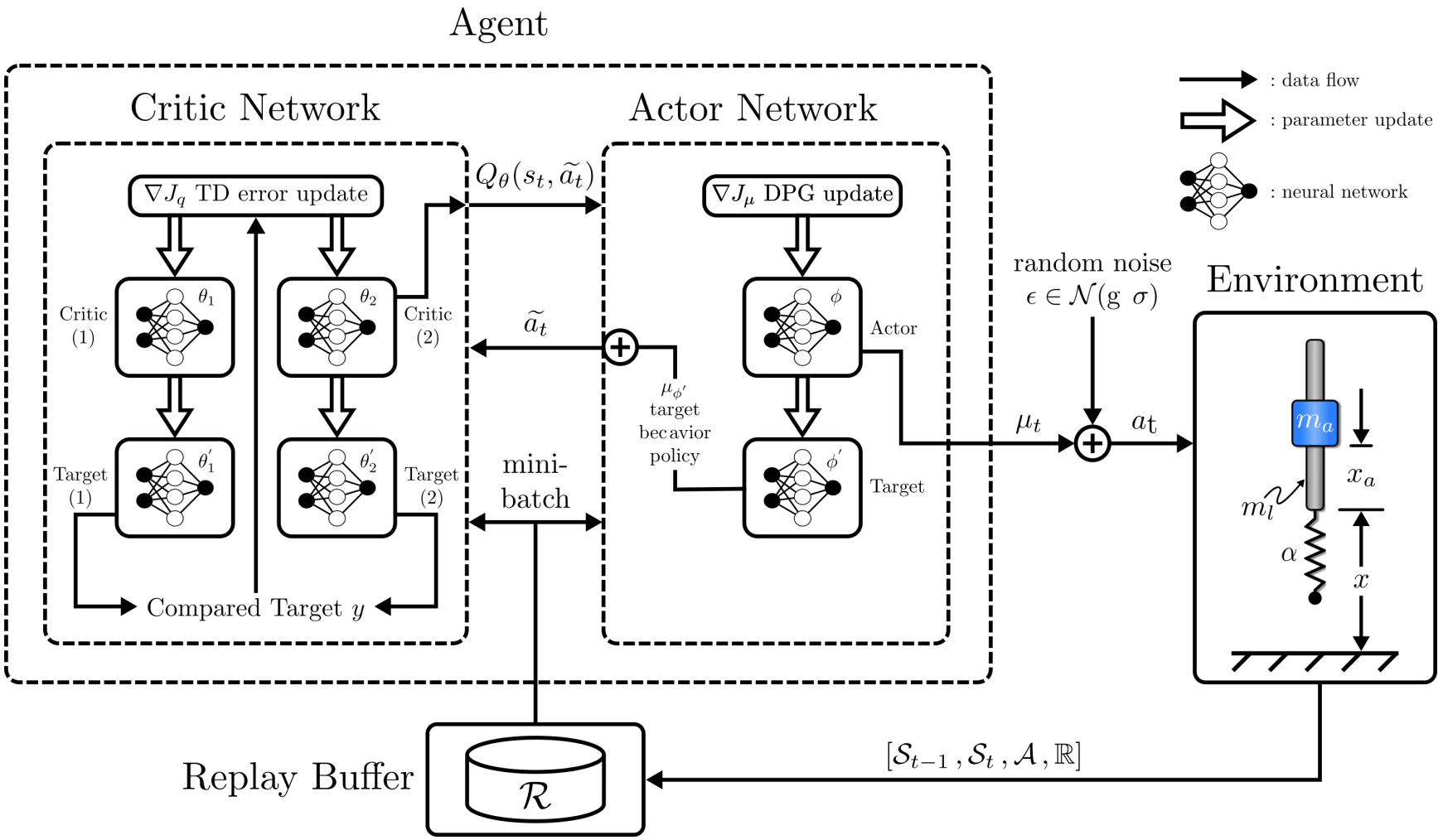


Figure 5. Twin Delayed Deep Deterministic Policy Gradient Block Diagram with Monopode as Environment

the difference between the value of the current state/action pair using the main Q-function, and the reward of the current state/action pair plus the discounted value of the next state/action pair using the target Q-function. The loss function takes the form:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_\phi(s,a) - \left(r(s,a) + \gamma\,(1-d)\,Q_{\phi_{targ}}(s', \pi_{\theta_{targ}}(s'))\right)\right)^2\right] \tag{3}$$

where $\pi_{\theta_{targ}}(s')$ is a target policy that, in a similar manner to the target Q-function, follows the main policy, π_θ, at a delayed pace either by directly copying the values or by polyak averaging. Additionally, d represents a boolean value which depends on the terminal status of the next state, s'.

As for updating the policy for the actor critic-type algorithm, this aspect is rather simple. Because DDPG, and therefore TD3, are built to accommodate only continuous action spaces, the Q-function is assumed to be differentiable with respect to action. Therefore to find optimal policy parameters, θ, for the policy, π_θ, the solution of the following equation must be found:

$$\max_\theta \underset{s\sim D}{E}\left[Q_\phi(s, \pi_\theta(s))\right] \tag{4}$$

The reason that TD3 is considered the successor to DDPG, is that there are some additional tricks deployed in addition to the description thus far. The first being the addition noise to the target policy. It can be seen in 3 that the target policy, $\pi_{\theta_{targ}}$, is required to generate an action to evaluate the the target Q-function. Noise is added to the policy taking the form:

$$a'(s') = \text{clip}\left(\pi_{\theta_{targ}}(s') + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{5}$$

where ϵ represents the noise sampled in some form that the user can specify. This method of adding noise was shown to reduce the issue of the Q-function approximator developing large values for certain state/action pairs, therefore smoothing out the Q-function.

The second trick the TD3 algorithm employs is the addition of a second Q-function approximator and target Q-function approximator. A known potential issue of the Q-function is that is can suffer from overestimation of the value of action/state pairs. This, of course, leads to the policy learning actions that the Q-function assumes are better than they actually are. To alleviate this issue, the authors suggest instantiating two Q-functions and two target Q-functions. Calculating the target Q-function is then completed by evaluating the two target Q-functions:

$$y(r, s', d) = r(s, a) + \gamma\,(1 - d)\,Q_{\phi_{i,targ}}(s', \pi_{\theta_{targ}}(s')), \quad \text{for } i = 1, 2 \tag{6}$$

and taking the lower of the two targets to update both the main Q-functions:

$$L(\phi_1, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_{\phi_1}(s, a) - y(r, s', d)\right)^2\right] \tag{7}$$

$$L(\phi_2, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_{\phi_2}(s, a) - y(r, s', d)\right)^2\right] \tag{8}$$

.

The last trick that the TD3 algorithm employs is the addition of a delay between the update of the Q-functions and the policy. It has been shown that in doing this the Q-function is able to converge to a better solution before updating the policy. Ultimately, the addition of a policy update delay was done to reduce coupling between the Q-function and the policy. The recommended delay is updating the policy every two Q-function updates.

There are many implementations of the TD3 algorithm that are available. Of these, the StableBaselines3 implementation is used to complete the work in this thesis [29]. StableBaselines3 is a widely popular library of RL algorithm implementations and is composed of well written and understandable documentation for the supported implementations. The differentiable function approximators used to estimate the policies and Q-functions are built within StableBaselines3 using PyTorch [30], which is also a widely popular framework for machine learning and more specifically reinforcement learning.

1.6 Contributions

The purpose of the work presented in the remainder of the document is to propose and evaluate the performance of a concurrent design architecture that utilizes RL techniques for flexible jumping systems. A concurrent design system in this work, is one that concurrently learns a mechanical design for a system and an associated control policy for said system. There is a void in the literature surrounding RL based concurrent design, particularly regarding locomotive robotics applications. It is of interest in this work to evaluate if a technique can be developed to generate better performing systems than ones that implements only controller policy learning. The method will be evaluated in simulation on a simplified flexible-legged jumping monopode. The components which build the concurrent design architecture will be split across the next few chapters wherein additional finding will be presented.

In the next chapter, a RL-based controller will be trained on the monopode jumping system to evaluate the effectiveness of training for efficient control. Power use is often considered when designing RL controllers for rigid systems, typically taking the form of a weighted negative reward when deploying an RL algorithm. It is of interest to evaluate if defining strategies, with efficiency being the primary objective, for flexible systems, if the resulting control strategy takes advantage of system flexibility. To determine if the learned policies are approaching what current literature supports regarding optimal control, ~~in Chapter 3~~, the performance will be evaluated against input shaping techniques.

Additionally, it is of interest to incorporate mechanical design into the controller learning process. Therefore, in Chapter 4, a RL problem is defined in a unique way such that the environment the RL policy is deployed in is a simulation of an environment where the actions sent by the policy are mechanical design updates. The method will be evaluated on the monopode jumping system to learn mechanical design parameters related to flexibility. Futhermore, using a single fixed control input, it is of

interest to determine if this technique can be used to define designs to accomplish multiple tasks. Therefore, using a single control input generated from the above input shaping techniques, it will be evaluated if an RL learning technique can be used to learn designs that cause the monopode to jump to multiple heights.

Next, in Chapter 5, the methods of learning a control policy and a design will be combined to create a concurrent design architecture. Two methods of implementing the mechanical design update will be shown, and the effects of the two methods will be discussed. Furthermore, a newly introduced hyperparameter when implementing the constructed concurrent design will be evaluated and the results will be discussed. The methods will be tested to evaluate efficient jumping vs non-efficient jumping concurrent designs for the monopode jumping system. Ultimately, the resulting concurrent design performances will be presented ~~and~~ and compared to the performances of control policies trained on static designs.

Lastly, in Chapter 6, the work presented in this document will be concluded and the results of the proposed concurrent design process will be highlighted. Accompanying the conclusive remarks, future research that has been enabled by the work presented, will be discussed.

II Learning Efficient Jumping Strategies for the Monopode System

Utilizing reinforcement learning to train a neural network based controller has been shown to be useful for controlling many robotic systems [22, 23]. It has been used to successfully control rigid-legged robots both in simulation and on physical hardware [21, 31]. Reinforcement learning has been shown to be capable of defining more effective and efficient jumping techniques for a single-legged robot with SEAs [32]. It has also been shown to be an effective method for controlling multi-legged robots both in simulation and on physical hardware [33–35]. Furthermore, it has been shown to be useful for defining energy efficient strategies for multi-legged robots that have been deployed on physical hardware [36]. However, the body of work demonstrating the use of RL to train controllers for flexible systems is limited, particularly in regards to legged locomotive systems. In this chapter, RL is deployed to define an energy efficient jumping strategy for a monopode jumping system. The purpose of this work is to validate the use of RL for defining the control aspect of a concurrent design architecture for flexible jumping systems.

2.1 Monopode Jumping System

To evaluate the methods discussed in this chapter, a monopode system like the one shown in Figure 6 was used to represent a flexible jumping system. This system has been studied and has been proven to be an effective base for modeling the jumping gaits for many different animals [37].

The monopode is controlled by accelerating the actuator mass, m_a, along the rod mass, m_l, causing a hopping like motion. The system contacts the ground through a nonlinear spring, represented by the variable α in the figure. Also included in the model is a damper parallel with the spring, having a damping coefficient of c, though it is not shown in the figure. Variables x and x_a represent the rod's global position and the actuator's local position with respect to the rod, respectively. The equation of

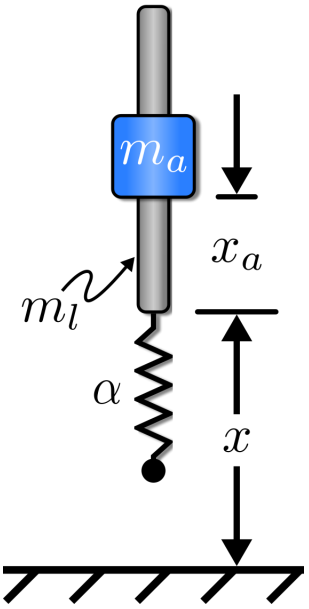


Figure 6. Monopode Jumping System

Table 1. Monopode Model Parameters

Model Parameter	**Value**
Mass of Leg, m_l	0.175 kg
Mass of Actuator, m_a	1.003 kg
Spring Constant, $\alpha_{nominal}$	5760 N/m
Natural Frequency, ω_n	$\sqrt{\frac{\alpha}{m_l+m_a}}$
Damping Ratio, $\zeta_{nominal}$	1e-2 $\frac{\text{N}}{\text{m/s}}$
Gravity, g	9.81 m/s^2
Actuator Stroke, $(x_a)_{\max}$	0.008 m
Max. Actuator Velocity, $(\dot{x}_a)_{\max}$	1.0 m/s
Max. Actuator Acceleration, $(\ddot{x}_a)_{\max}$	10.0 m/s^2

motion for the system is:

$$\ddot{x} = \frac{\gamma}{m_t}\left(\alpha\, x + \beta\, x^3 + c\,\dot{x}\right) - \frac{m_a}{m_t}\,\ddot{x}_a - g \tag{9}$$

where x and $\dot{x}$ are position and velocity of the rod, respectively, the acceleration of the actuator, $\ddot{x}_a$, is the control input, and m_t is the mass of the complete system. Constants α and c represent the linear spring and damping coefficient, respectively, and constant β is set to $1e8$. Ground contact determines the value of γ, so that the spring

and damper do not supply force while the leg is airborne:

$$\gamma = \begin{cases} -1, & x \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Additionally, the spring compression limit, or the systems position in the negative x direction, is limited to 0.008m. The system is also confined to move only vertically in regards to Figure 6 so that controlling ballance is not required. The values of the parameters show in Figure 6 are displayed in Table 1.

2.2 Training Environment

Using the monopode model, an environment aligning with the standards set by OpenAI for a Gym environment was created [38]. The observation and action spaces were defined, respectively, as follows:

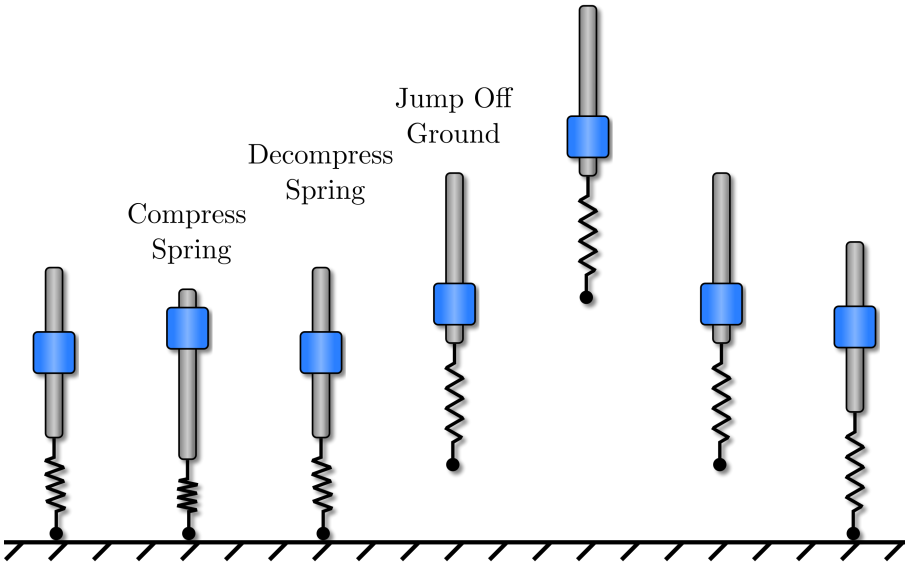
$$\mathcal{S} = [x_{a_t}, \dot{x}_{a_t}, x_t, \dot{x}_t] \tag{11}$$

$$\mathcal{A} = [\ddot{x}_{a_t}] \tag{12}$$

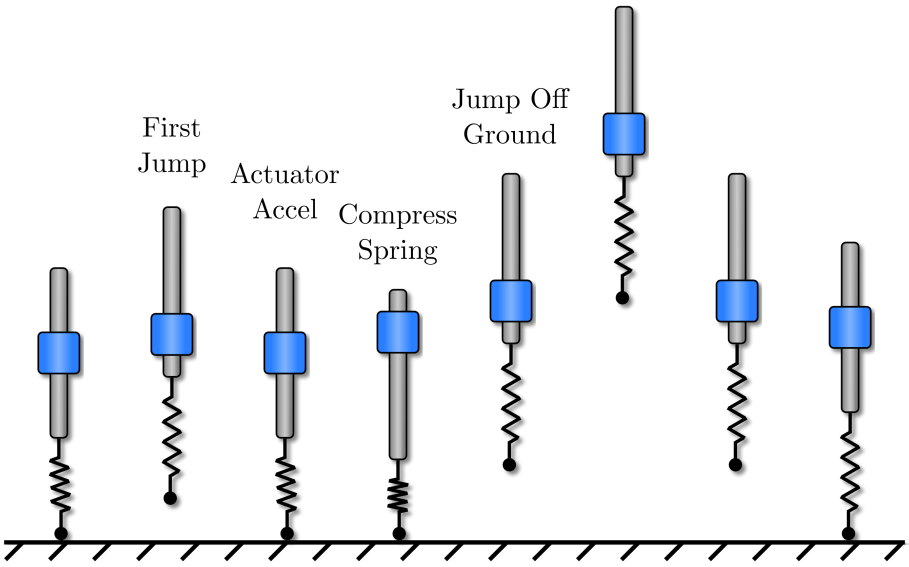
where x_t and $\dot{x}_t$ were the monopode's position and velocity at time t, and x_{a_t}, $\dot{x}_{a_t}$ and $\ddot{x}_{a_t}$ were the actuator's position, velocity and acceleration, respectively.

Two separate stopping conditions were defined for the environment to evaluate two different jump types and therefore two different input commands. The first was defined as the monopode's position being greater than zero than returning to zero once. The second was defined like the first but with the monopode's position being greater than zero and then less than zero twice.

Two different jumps were created from these stopping conditions. The first was referred to as a single jump command, and the second a stutter jump command. The intent of utilizing two different jumping commands was to determine if a RL algorithm was more or less effective in learning differing strategies depending on the complexity of the desired command.



(a) Example Single Jump



(b) Example Stutter Jump

Figure 7. Jumping Types for the Monopode Jumping System

An example single jump can be seen in Figure 7a. The intended command from the learned controller would be one that would jump the monopode once. This type of command would ideally compress the spring/damper by accelerating the actuator in the positive direction. This would allow the system to store energy in the spring that could be used to cause the system to jump. The actuator mass should then accelerate

downward forcing the spring to decompress. At this point, the system should be accelerating upwards and the actuator downwards such that the monopode leaves the ground completing a single jump.

An example stutter jump can be seen in Figure 7b. The intended command from the learned controller would be one that would jump the monopode twice. This type of command would firstly complete an optimal single jump. Following that motion, the actuator should assume an acceleration direction to recompress the spring storing more energy with a farther compression. When the spring is compressed to its maximum value or the system's total acceleration reaches zero, the actuator mass should accelerate downwards forcing the spring to decompress. At this point, the system should be accelerating upwards and the actuator downwards, similar to the single jump, such that the monopode leaves the ground completing a stutter jump.

2.3 Efficient Control Strategies

Efficient control of a robotic system is often one of the most important aspects of a controller's design. Applications where a robotic system is deployed and relies on a limited power source, such as a mobile walking robot, will often require an efficient control strategy. Modern, traditional methods, such as model predictive control, have been shown to produce energy efficient locomotion strategies for wheeled and legged systems [39, 40]. It is of interest in this work to utilize RL, a modern neural network based control method, to find strategies which are designed with power efficiency as the primary objective.

Two different reward functions were designed to accomplish the task of determining how well RL learns efficient jumping strategies. The purpose of defining two different reward functions was to compare the input commands and resulting jumping shapes of the two controller types to determine if the efficient controller was learning to conserve power.

The first reward function was one that ignored power usage and focused solely on the height of the jump:

$$R = x_t \tag{13}$$

where x_t was the height of the monopode system at any given time step. The second reward function was one that was defined to accomplish the same task, but also consider power consumption. It was defined as:

$$R = \frac{x_t}{\sum_{t=0}^{t} P_t} \tag{14}$$

where P_t was the power consumption of the monopode system at any given time step defined mechanically as the product of the actuator's acceleration, velocity and mass:

$$P_t = m_a \, \dot{x}_a \, \ddot{x}_a \tag{15}$$

where m_a was the mass of the actuator, and $\dot{x}_a$ and $\ddot{x}_a$ where the actuators velocity and acceleration, respectively.

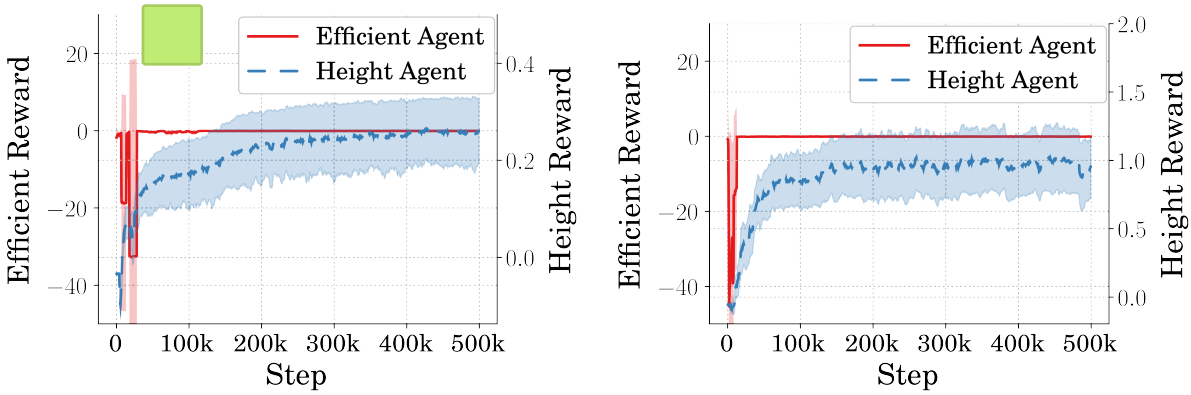
2.4 Deploying TD3

Because training an RL controller does not guarantee that the controller will learn an optimal strategy without finding local optima, training more than one controller is common practice for evaluating performance. In this work, fifty different controllers where trained, each with a different random network initialization. Each controller was trained for a total 500k time steps. The remaining hyperparameters set using the TD3 algorithm are defined in Table 2.

The average and standard deviation of the rewards during training for each of the fifty different algorithms were found and are presented in Figure 8. They represent the controllers being trained to accomplish their respective goals. Looking at Figure 8a, which shows the rewards for learning a single jumping command, it is clear that there are stark differences between the efficient and high jumping reward types. Firstly, the high jumping strategy does converge after 500k steps of training. The reward for the

Table 2. TD3 Training Hyperparameters

Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	1000 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	50 Random Seeds

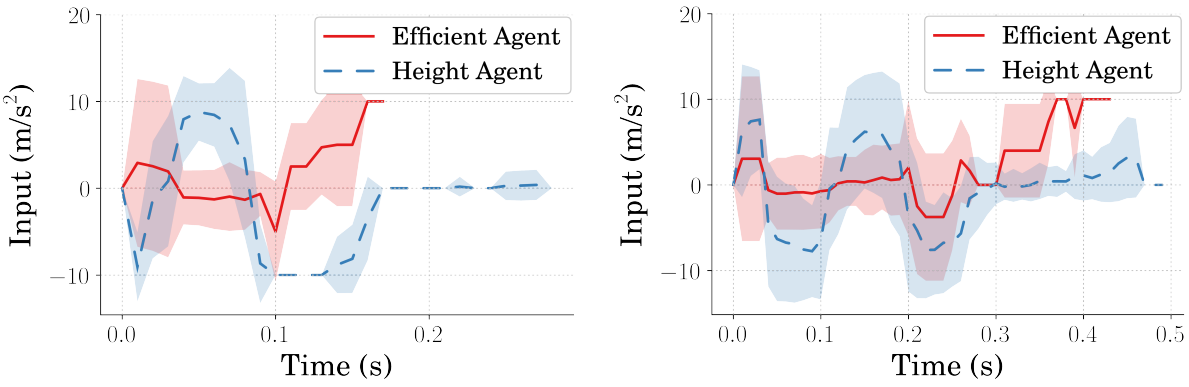


(a) Single Jumping Reward

(b) Stutter Jumping Reward

Figure 8. Reward vs Time Step During Training

efficient jumping controller, having been defined drastically different than the reward from the high jumping controller, not surprisingly, looks drastically different than the high jumping reward. The reward for the efficient agent is heavily punished for using power without gaining height, which is a frequent occupance in the beginning of training. This allows the policy to learn quickly that using less power will result in a higher rewards which is shown in the training reward data. In Figure 8b, it is also apparent that the height controller is converging to a solution. Additionally, the efficient controller can be seen to have learned in the same rapid form as the single



(a) Single Jumping Input **(b)** Stutter Jumping Input

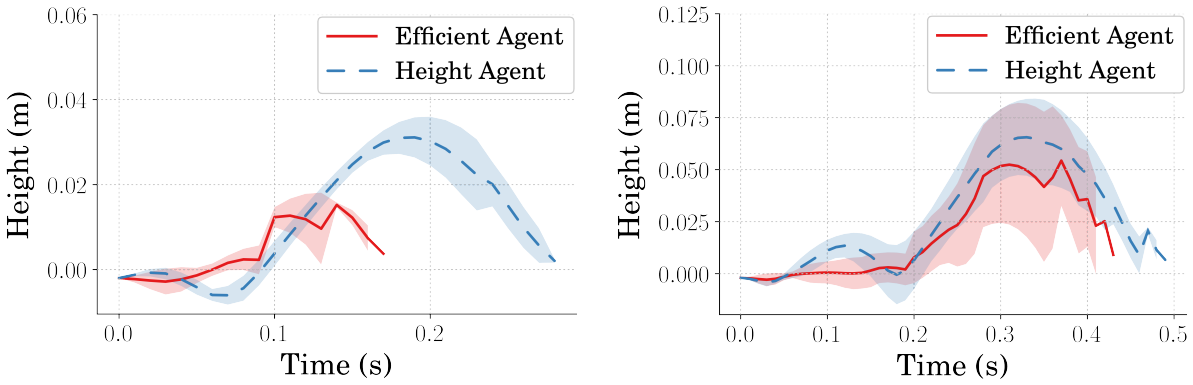
Figure 9. Average and Standard Deviation Inputs to Monopode

jumping command type.

2.5 Average Performance of Network Controller

2.5.1 Input Commands. The average and standard deviation of the final controller's input commands were evaluated for both the single and stutter jumping cases, and the results are shown in Figure 9. The individual subfigures are to compare the controllers that were trained to jump efficiently to those trained to jump high. At first glance, there are obvious differences regarding timing, magnitude, and direction. There are also slight differences in variance seen between the two controller types.

Starting with Figure 9a, which displays the commands for the single jumping case, it is most obvious that the direction for the initial acceleration of the actuator mass differs between the efficient controllers and height controllers. In the case where the controller is learning to jump high, an initial acceleration of the actuator mass in the negative direction is learned, which contrasts the case where the controller is learning an efficient command. Further, the magnitude of the commands is drastically different which may be an indicator for conserving power. For the stutter jumping case shown in Figure 9b, it is immediately apparent that the magnitudes of the commands differ greatly. They are, however, more similar in regards to their timings and directions than the single jumping command.



(a) Single Jumping Height

(b) Stutter Jumping Height

Figure 10. Average and Standard Deviation Heights of Monopode

In both the single and stutter jumping cases, it can be seen that there is upward acceleration command towards the end of the jump, which again, might be an indicator of a more efficient jumping strategy. Furthermore, it can be observed that the in single jumping case, there exists more variance across different instances of the trained efficient controllers in comparison to the height controllers. This does not seem to be the case for the stutter jumping command type, though both cases do seem to generate controllers with high variance inputs across instances.

2.5.2 Jumping Height Performance. The average and standard deviation of the final controller's jumping performance were evaluated for both the single and stutter jumping cases, and the results are shown in Figure 10. In both the single and stutter jumping cases, there are differences in jumping ability when comparing the efficient and height controller types. It is apparent that when increasing the complexity of the command from a single jump to a stutter jump, the efficient controllers are better able to match the performance of the height controllers.

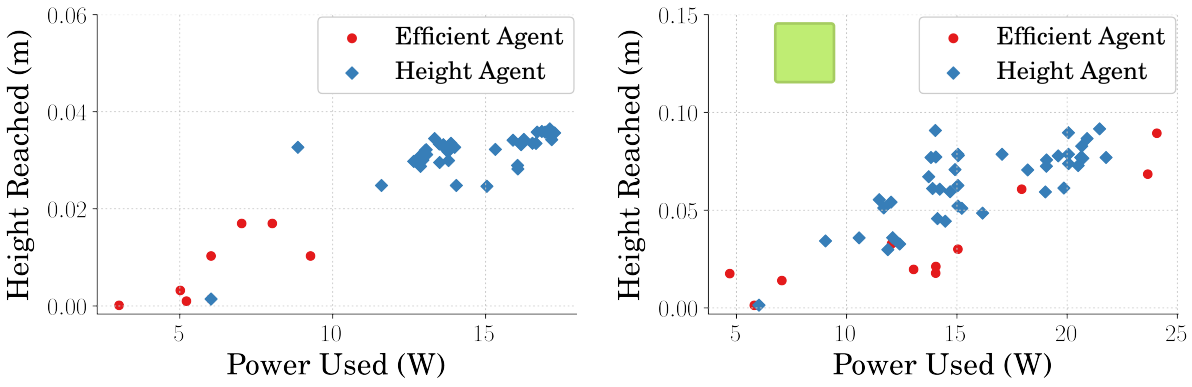
Shown in Figure 10a, the height controllers for the single jumping case learned a command input that outperformed the efficient controllers in terms of jump height. The resulting motion from the input discussed in the previous section can also be seen in that the efficient controller learned to simply compress the spring, then jump the

monopode. The height controllers, in contrast, disregarding power consumption, learned to decompress the spring from its nominal position, keeping it below the point of leaving the ground, then recompressing for a much higher jump. For the single jumping commands, the high jumping strategy trained controllers that jumped the monopode 104.53% higher than the efficient jumping strategy.

Figure 10b compares the jumping performance of the efficient and height strategies for the stutter jumping command. They differ, but less drastically than the single jumping case. The likeness of the command shapes shown in Figure 9b, results in jumping responses that also share a similar form. The large differences seen in the stutter jumping shapes are similar those of the inputs that create them, in that they differ mostly regarding the magnitudes. For the stutter jumping command, the high jumping strategy trained controllers that jumped the monopode 20.69% higher than the efficient jumping strategy.

In both the single and stutter jumping cases, the upward acceleration from efficient controllers toward the end of the command can be seen in that the monopode regains height after the start of its final decent from maximum height. This can be explained in that the efficient control method, which punishes power consumption, discovered a way to maintain height throughout a jump where the utilization of additional power is less costly. It is less costly to influence the monopode's position while airborne because there is no resistance from the spring and damper. Additionally, regarding variance, the single jumping controllers seems to produce jumping shapes with similar levels of variance. Whereas in the stutter jumping case, though similar in high levels of input variance, the jumping height variance for the efficient controllers is noticeably higher that that of the height controller.

2.5.3 Height Reached vs. Power Used. Height reached versus power consumed data, for both the single jumping and stutter jumping cases is shown in



(a) Single Jumping Efficiency

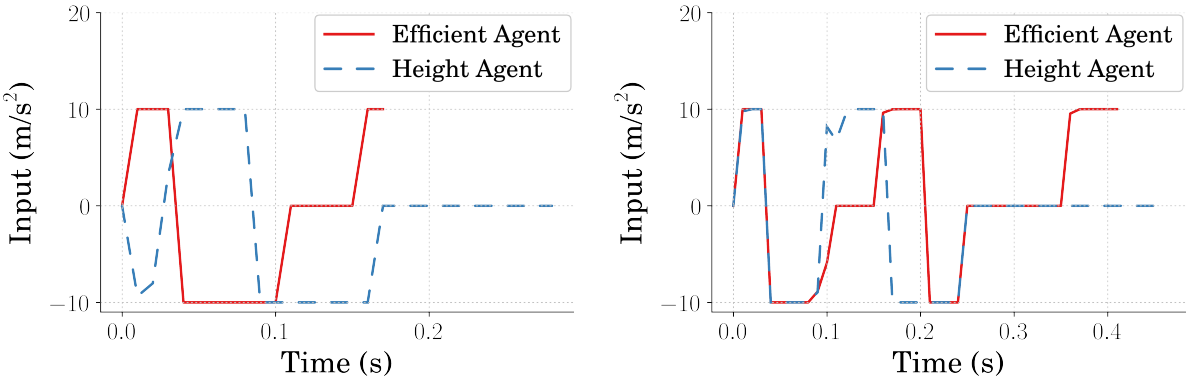
(b) Stutter Jumping Efficiency

Figure 11. Height Reached vs Power Consumed of Monopode

Figure 11. In both the single and stutter jumping cases, the efficient controllers utilized less power and therefore suffered regarding jump height. This matches what was seen in the previous sections regarding input commands and jumping shapes. It was observed however, in both jumping cases, that the power conservation gain is more than the maximum jumping height cost.

In the single jumping case, shown in Figure 11a, there is an apparent separation between the two controller types where the height controllers learn control strategies that use more power and jump higher. On average, for the single jumping command type, the efficient jumping strategy learned jumping commands that were 126.27% more efficient at the cost of 104.53% in average jump height.

As for the stutter jumping case, shown in Figure 11b, the difference in performance is less obvious. This can be explained in that more complex jumping strategies give an RL algorithm more opportunity to learn a control policy that can better take advantage of system flexibility. The variance of the two controller types, being quite high, matches what is seen in the previous sections and results in more mixing of the data. On average, for the stutter jumping command type, the efficient strategy learned commands that were 101.45% more efficient with the average maximum jumping height only being punished by 20.69%.



(a) Single Jumping Input **(b)** Stutter Jumping Input

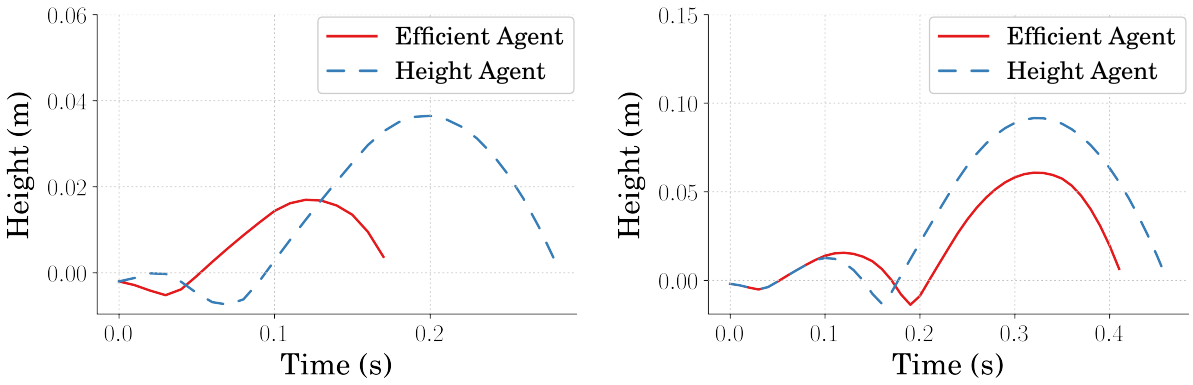
Figure 12. Optimal Inputs to Monopode

2.6 Optimal Performance of the Proposed Controller

2.6.1 Input Commands. Taking the best of the fifty different controllers trained for both the single and stutter jumping cases and comparing the efficient and height controller's performance can show what is possible with a properly defined RL problem. Figure 12 shows the differences in the input commands generated when selecting the highest performing controller in terms of reward received. It can be seen that there are less differences between the the efficient and height controllers in comparison to the average results from Section 2.5. A major similarity being that magnitudes are similar across all cases such that the controllers utilize the actuator's maximum acceleration.

Looking at Figure 12a, which compares the efficient and height controllers for the single jumping input, the major differences are the timing and direction of the commands. This is similar to the average performance evaluation, from Section 2.5.1, in that the efficient controller does not take advantage of the slight decompression of the spring before the monopode leaves the ground. Because of this, the efficient controller learns a different timing for a single jump.

As for the sutter jumping case, shown in Figure 12b, the differences between the efficient and height controller is less drastic. The initial timing is largely the same as



(a) Single Jumping Height **(b)** Stutter Jumping Height

Figure 13. Optimal Heights of Monopode

both controller types learn to utilize the decompression of the spring. The differences begin when decompressing the spring a second time and completing the first jump. The efficient controller learns a command similar in form to a bang-coast-bang command, where, in contrast, the height controller learns a command similar to that of a bang-bang shaped input.

2.6.2 Jumping Height Performance. In line with the previous section, it is of interest to evaluate the jumping performance curves of the best controllers trained. Figure 13 displays the jumping performance for both jump types as well as both controller types when utilizing the inputs shown in Section 2.6.1. These curves show that the efficient controllers, in the best case scenario, do not generate commands that jump the monopode as high.

Figure 13a, which compares the efficient and height controllers for the single jump, shows that the efficient controller does not learn to utilize the allowable decompression in the spring. In Figure 13b, it can be seen that when utilizing a command more similar in form to a bang-coast-bang command, like the efficient controller learned, the timing of the jump sequence is shifted and the resulting final height is less than the height controller who's command is more similar to a bang-bang shaped command. The efficient controller having learned to coast between commands

shows it is a useful method for conserving power, but not a good strategy for optimizing jumping height.

2.7 Conclusion

Two different controller types where trained to generate two different jumping commands for the simplified monopode jumping system. The first type of controller was one that would command the monopode system to jump high where the reward was based on nothing other than system height. The second type of controller was one which controlled the monopode to jump high but at the cost of power consumed, such that high jumps that consumed high amount of power where less desirable than high jumps that consumed less power. It was shown that the rewards passed to RL algorithms that are training controllers can be manipulated so that the the learned input commands take advantage of the spring/damper that exists within the monopode jumping system. Furthermore, the timing of the commands, as well as the input magnitude and direction are all affected when defining a reward strategy that seeks to increase power efficiency. When considering the average performance of the different control strategies, for both the single and stutter jumping cases, the heights reached were less for the efficient strategies. However, they were significantly more efficient, particularly when scaling the complexity of the command from the single jump to the stutter jump. It should be the concluded then, that RL might serve as a useful method for defining control strategies for flexible-legged jumping systems, particularly when energy efficiency is of interest. Additionally, when considering more complex control strategies, which might be difficult to define efficiently, RL might serve as a useful method for effective efficient strategies.

III Using Input Shaping to Validate RL Controller

In utilizing RL to define a control strategy for a robotic system, the resulting commands sent to the system are often described as optimal, or at least approaching optimal. They are described as such due to the nature of RL problems in that the techniques used to learn a policy are optimization theory based so the policy being trained is one that is approaching an optimal solution in regards to the reward defined. Interpreting the commands claimed to be optimal, in the context of control theory, is an important part of utilizing RL for defining control strategies for robotic systems however. In the case of a flexible jumping robot, where the command is to jump the the system as high as possible or as efficient as possible, are the commands generated truly approaching an optimal solution? Methods such as input shaping, having been shown to be an effective method for defining optimal control strategies for flexible jumping systems, can be used to evaluate the commands generated by the RL generated policies [2].

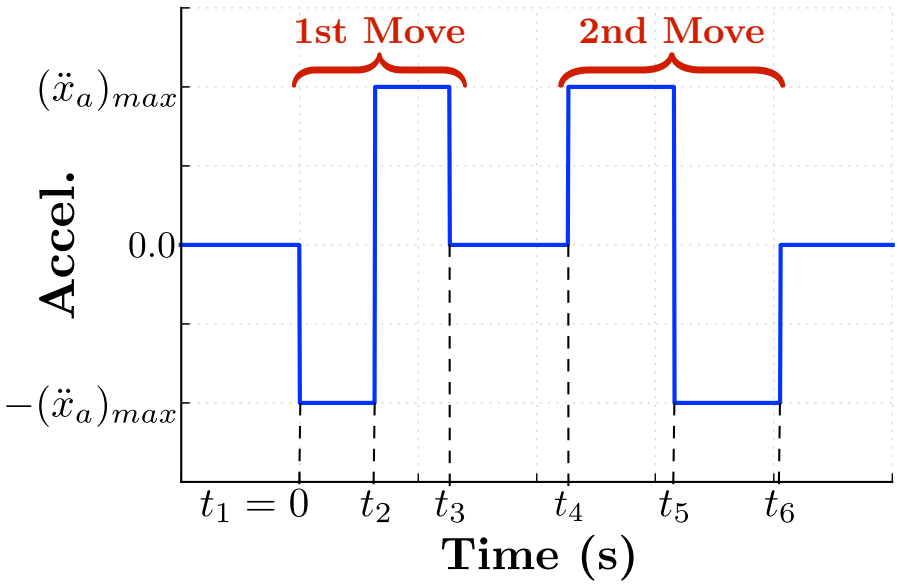


Figure 14. Jumping Command [2]

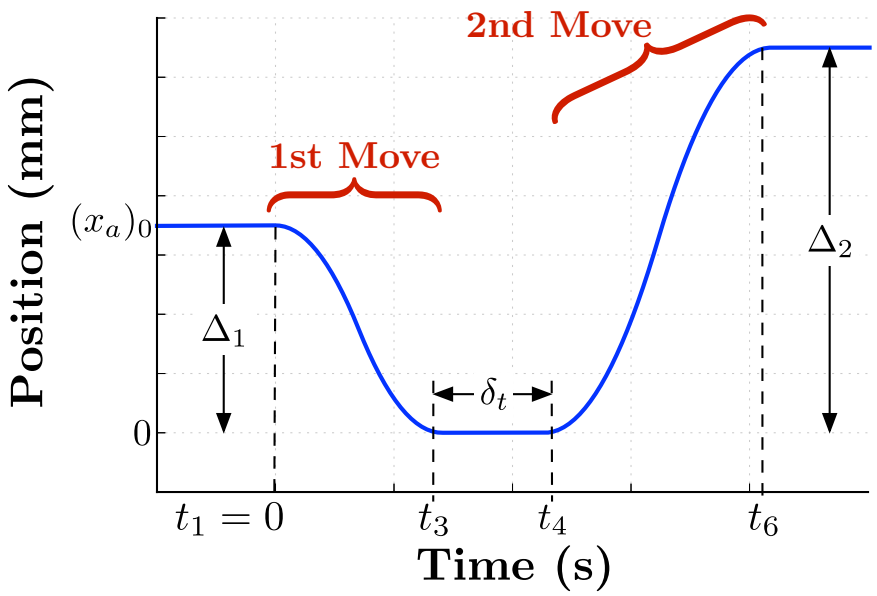


Figure 15. Resulting Actuator Motion [2]

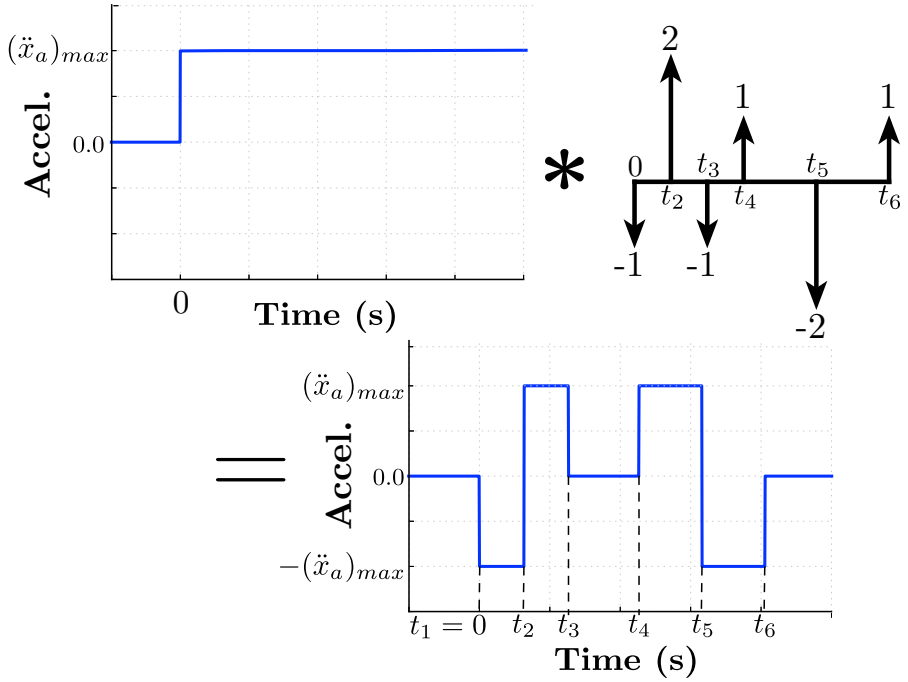


Figure 16. Decomposition of the Jump Command into a Step Convolved with an Impulse Sequence [2]

3.1 Input Shaping Controller Input

Bang-bang based jumping commands like the one shown in Figure 14 are likely to result in a maximized jump height [2]. For these command types, regarding the

monopode jumping system, the actuator mass travels at maximum acceleration within its allowable range, pauses, then accelerates in the opposite direction. Commands designed to complete this motion are bang-bang in each direction, with a selectable delay between them. The resulting motion of the actuator along was rod is shown in Figure 15. Starting from an initial position, x_{a_0}, the actuator moves through a motion of stroke length Δ_1, pauses there for δ_t, then moves a distance Δ_2 during the second portion of the acceleration input.

This bang-bang-based profile can be represented as a step command convolved with a series of impulses, as was shown in Figure 16 [41]. Using this decomposition, input-shaping principles and tools can be used to design the impulse sequence [42, 43]. For the bang-bang-based jumping command, the amplitudes of the resulting impulse sequence are fixed, $A_i = [-1, 2, -1, 1, -2, 1]$. The impulse times, t_i, can be varied and optimal selection of them can lead to a maximized jump height of the monopode system [2]. Commands of this form will often result in a stutter jump like what was shown in Figure 7b of Chapter 2, where the small initial jump allows the system to compress the spring to store energy to be used in the final jump.

3.2 RL Controller Input

Discussion and figures from the inputs defined by the RL algorithms for the monopode system.

...

Waiting for some data from DV.

3.3 Conclusion

Discuss the results.

IV Mechanical Design of a the Monopode Jumping System

Often it is the goal of a controls engineer to design a controller to accommodate and manipulate systems according to the system description provided. However, research has been conducted showing the value of studying the manipulation of mechanical design parameters in order to achieve a desired system behavior [17]. In this chapter, reinforcement learning is shown to be useful as a tool to learn mechanical designs given a predefined system controller for the monopode jumping system. RL has been shown to be an effective strategy for finding optimal concurrent designs for many different types of systems [19, 44, 45]. It has even shown it's ability to define designs that are successfully deployed on physical hardware [18]. It is comparable to work where evolutionary algorithms are deployed to optimize physical parameters of systems for improved energy efficiency [46, 47]. Here, RL is used to define an optimal design for the monopode jumping system described in Chapter 2.

4.1 Learning a Mechanical Design

Section 1.4 describes the common deployment methodology of an RL problem where defining a control policy for a robotic system is often the primary goal. In this chapter, rather than finding a control policy for a defined robotic system, RL is deployed to find a mechanical design for a defined control input. To do this, the general methods in setting up the problem have to change. Figure 17 shows the general flow of information for the algorithm.

The RL problem is similar to the common use case in that the algorithm utilizes the same information types to optimize a design, such as the state of the environment and the reward. The algorithm's interaction with the environment is drastically different, however. The action space, instead of being a command type input, is a range of design choices for a set of parameters within a simulation of a system. Applying these actions within the environment, rather than changing the state of a robotic

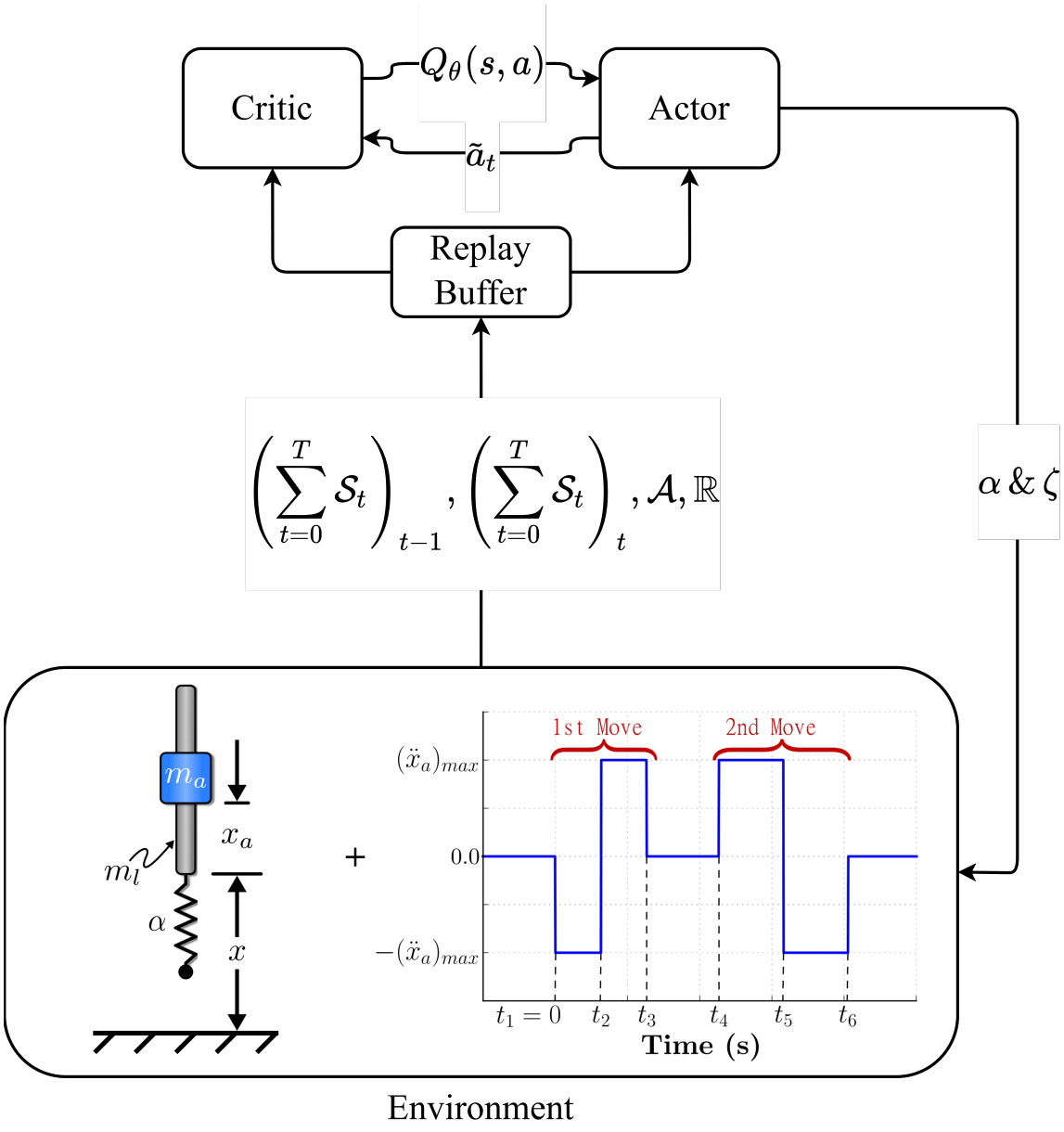


Figure 17. Learning a Mechanical Design

system from state s to s', instead simulates a system within the environment from time $t = 0$ to time T using a predefined control input. Because of this, the state saved as a transition is a matrix of states rather than a single vector state. The reward does not differ greatly in that it is based on the state of the system. A difference being that the information stored in a single state transition is much greater so the reward can be defined to utilize all the information.

For the work presented in this chapter, as is shown in Figure 17, the predefined control input used to simulate the monopode system within the environment at each step was an optimized controller generated using the input shaping techniques discussed in Section 3.1. Having been shown to be a useful technique for generating optimal control strategies, it was of interest to evaluate if an optimal mechanical design could be found to accompany the input.

4.2 Environment Definition

To allow the agent to find a mechanical design, a reinforcement learning environment conforming to the OpenAI Gym standard [38] was created. The monopode model described in Chapter 2 was used as the simulation, and the fixed controller input was based on the work described in Section 3.1. The mechanical parameters the agent was tasked with optimizing were the spring constant and the damping ratio of the monopode system. At each episode during training, the agent selected a set of design parameters from a distribution of available designs. The actions applied, $\mathcal{A}$, and transitions saved, $\mathcal{S}$, from the environment were defined as follows:

$$\mathcal{A} = \{\{a_\alpha \in \mathbb{R} : [-0.9\alpha_{nom}, 0.9\alpha_{nom}]\},\ \{a_\zeta \in \mathbb{R} : [-0.9\zeta_{nom}, 0.9\zeta_{nom}]\}\} \tag{16}$$

$$\mathcal{S} = \left\{ \sum_{t=0}^{t_f} x_t,\ \sum_{t=0}^{t_f} \dot{x}_t,\ \sum_{t=0}^{t_f} x_{at},\ \sum_{t=0}^{t_f} \dot{x}_{at} \right\} \tag{17}$$

where α_{nom} and ζ_{nom} are the nominal spring constant and damping ratio of the monopode, respectively; x_t and $\dot{x}_t$ are the monopode's rod height and velocity steps, and x_{a_t} and $\dot{x}_{a_t}$ are the monopode's actuator position and velocity steps, all captured during simulation. The action space was set to $\pm 90\%$ of the nominal values as that percentage allowed for a large variance in performance across the range of designs.

4.3 Rewards for Learning Designs

The RL algorithm was utilized to find designs for two different reward cases. Time series data was captured during the simulation phase of training and was used to evaluate the designs performance through these rewards. The first reward case used was:

$$R_1 = \left(\sum_{t=0}^{t_f} x_t \right)_{max} \tag{18}$$

where x_t was the monopode's rod height at each step during simulation. The goal of the first reward was to find a design that would cause the monopode to jump as high as possible.

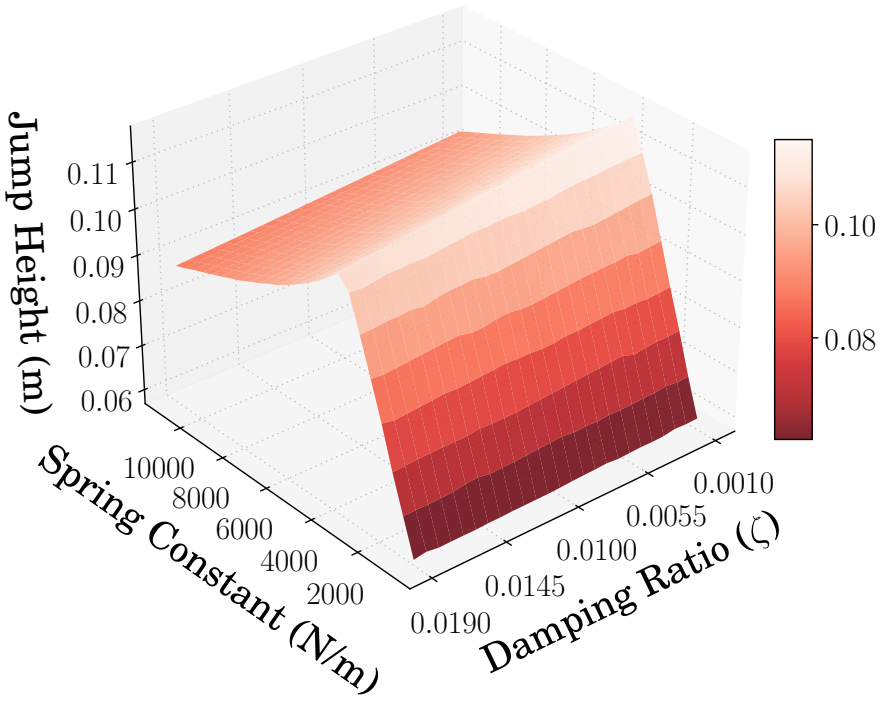
The reward for the second case was:

$$R_2 = \frac{1}{\frac{|\mathbb{R}_1 - x_s|}{x_s} + 1} \tag{19}$$

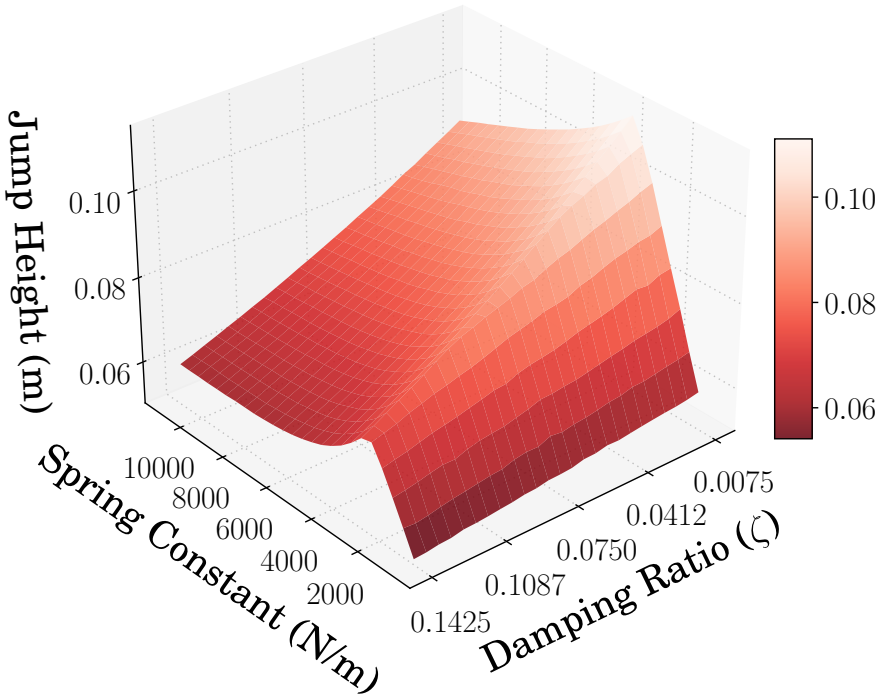
where x_s was the desired jump height, which was set to 0.01 m. The second case was utilized to test RL's ability to find a design that minimized the error between the maximum height reached and the desired maximum height to reach.

4.4 Design Space Variations

Figure 18 represents the heights the monopode could reach for two different design spaces. The design space provided for the first case, shown in Figure 18a, represents a space where the nominal damping ratio, ζ_{nom}, is 0.01, such that $\pm 0.9\,\zeta_{nom}$ creates a more narrow design space. This design space also has more values closer to the optimal value therefore making it more difficult to optimize quickly. The design space provided for the second case, shown in Figure 18b, represents a space where the nominal damping ratio, ζ_{nom}, is 0.075, such that $\pm 0.9\,\zeta$ creates a wider design space. Additionally, there is a more obvious maximum within the design space, making it easer to ascend towards an optimal design. In both cases there are many values that would satisfy the specified height jumping strategy.



(a) Jumping Performance of Narrow Design Space



(b) Jumping Performance of Wide Design Space

Figure 18. Reference Jumping Performance of the Monopode

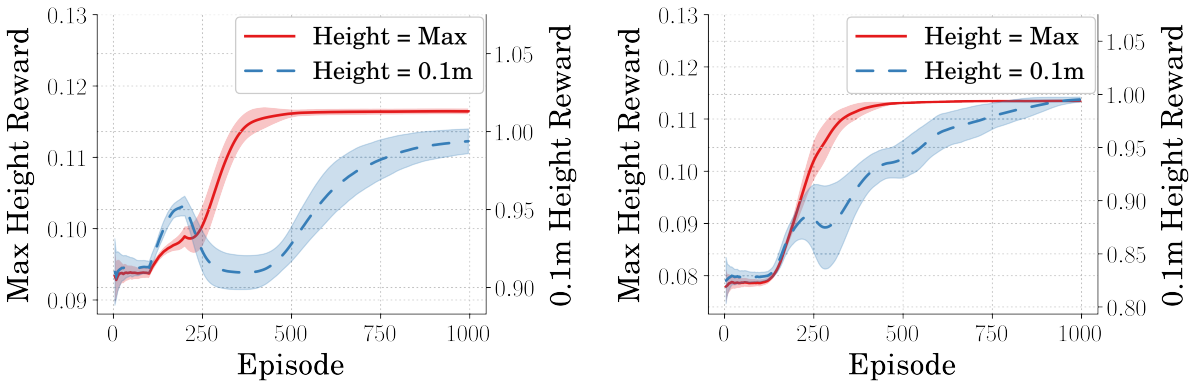
Table 3. TD3 Training Hyperparameters

Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	100 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	100 Random Seeds

4.5 Deploying TD3

The training hyperparameters were selected based on TD3's author recommendations and StableBaselines3 [29] experimental findings and are displayed in Table 3. All of the hyperparameters, with the exception of the rollout (Learning Starts) and the replay buffer, were set according to StableBaselines3 standards. The rollout setting was defined such that the agent could search the design space at random, filling the replay buffer with enough experience to prevent the agent from converging to a design space that was not optimal. The replay buffer was sized proportional to the number of training steps due to system memory constraints.

The average rewards for both the narrow and the wide design space agents are shown in Figure 19. They represent the agents learning a converging solution to the problem of finding optimal design parameters. Looking at Figure 19a, it is apparent that given a more narrow design space, both the high and the specified jumping agents were still able to learn a converging solution. It can also be observed that there exists more variance for the specified-height agent type compared to the agents assigned with maximizing jump height. Looking at Figure 19b, it is apparent that the agents given a



(a) Reward vs. Episode: Narrow Design Space **(b)** Reward vs. Episode: Wide Design Space

Figure 19. Reward vs. Episode for Learning Mechanical Design

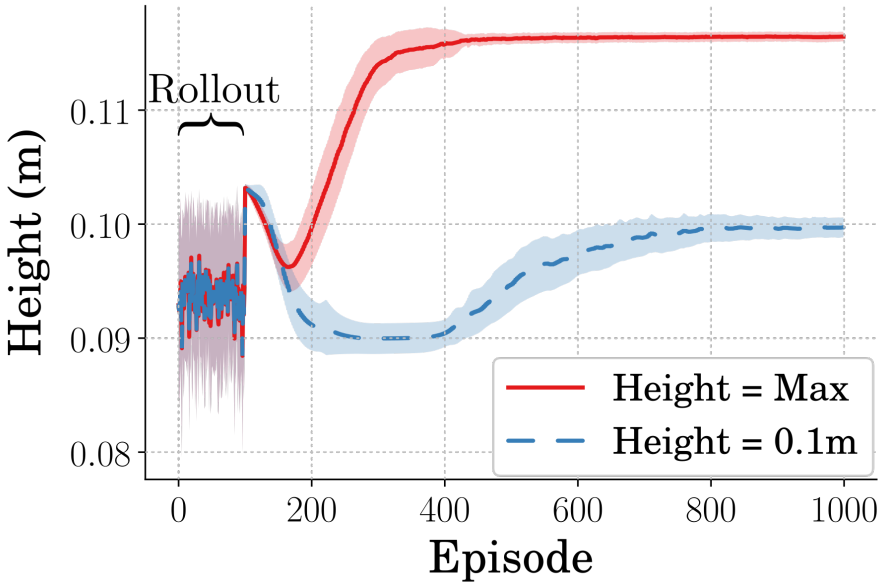
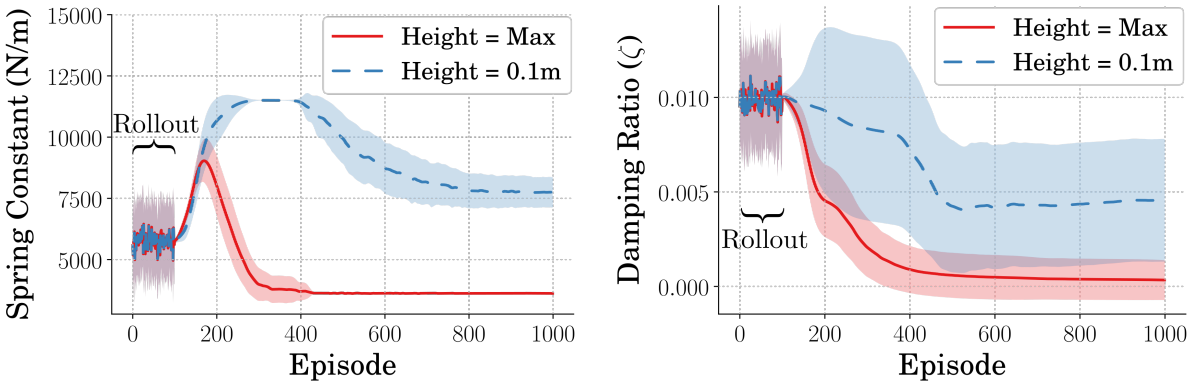


Figure 20. Height Reached During Training Given Narrow Design Space

wider design space where both able to learn a converging design solution. Though, given more designs to choose from, it appears the specified height agents are taking longer to converge. Additionally, once converged, because there are less values that allow the controller/design architecture to accomplish the tasks, there is less variance seen in designs learned.

4.6 Jumping Performance



(a) Spring Constant Selected During Training **(b)** Damping Ratio Selected During Training

Figure 21. Designs Learned for the Narrow Design Space

4.6.1 Narrow Design Space. Figure 20 shows the height achieved by the learned designs for the agents given the narrow range of possible damping ratio values. For the agents learning designs to maximize jump height, Figure 20 can be compared with Figure 18a showing that the agent learned a design nearing one which would achieve maximum performance. Additionally, looking at the agents learning designs to jump to the specified 0.1 m, the designs learned accomplish this with slightly more variance than that of the maximum height case.

The average and standard deviation of the spring constant and damping ratio design parameters the agents selected during training are shown in Figure 21. These plots represent the learning curves for the agents learning design parameters to maximize jump height and the agents learning design parameters to jump to 0.01 m. There is a high variance in both the spring constant and the damping ratio found for the agents that learned designs to jump to a specified height. The agents which were learning designs which maximized height found designs with very little variance in terms of spring constant and significantly less variances in terms of damping ratio.

4.6.2 Wide Design Space. Figure 22 shows the height achieved by the learned designs for the agents given a wide range of damping ratios. For the agents

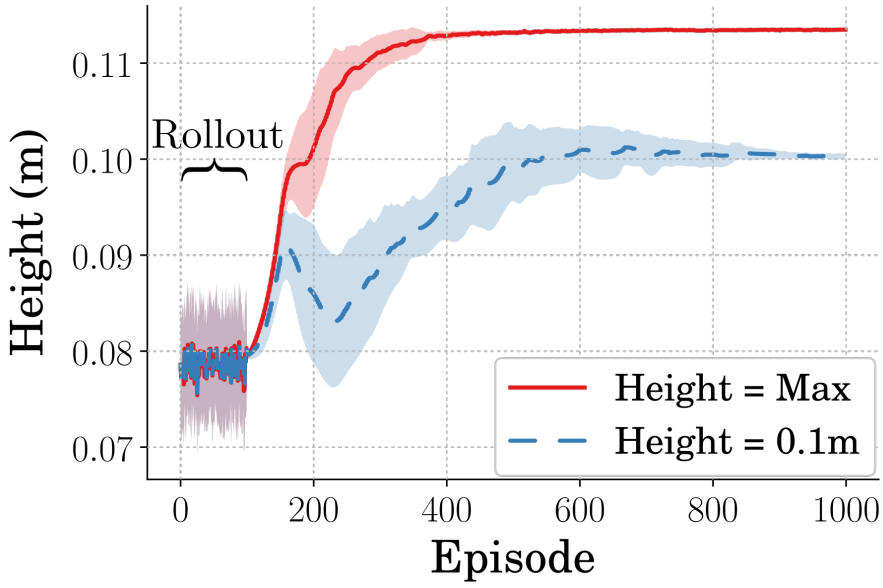
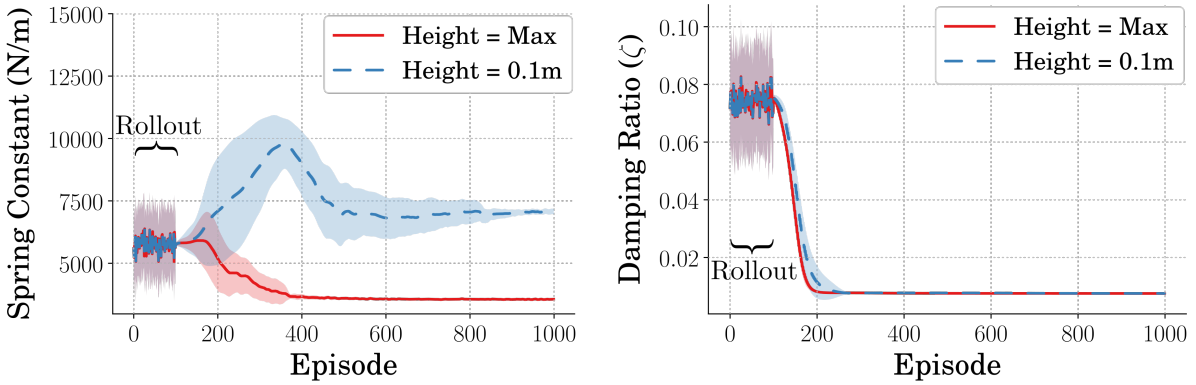


Figure 22. Height Reached During Training Given Wide Design Space



(a) Spring Constant Selected During Training **(b)** Damping Ratio Selected During Training

Figure 23. Designs Learned for the Wide Design Space

learning designs to maximize jump height, Figure 22 can be compared with Figure 18b showing that the agents learned a design nearing one which would achieve maximum performance. Additionally, looking at the agents learning designs to jump to the specified 0.1 m, the designs learned to accomplish this, again, only with slightly more variance than what is seen in the maximum height agents.

The average and standard deviation of the spring constant and damping ratio design parameters the agents selected during training are shown in Figure 23. For the

Table 4. Learned Design Parameters

Training Case		Design Parameter	Mean	STD
Narrow Design Space	Max Height	Spring Constant	3.62e03	3.82e01
		Damping Ratio	3.37e-04	2.11e-03
	Specified Height	Spring Constant	7.74e03	1.24e03
		Damping Ratio	4.55e-03	6.49e-03
Broad Design Space	Max Height	Spring Constant	3.55e03	4.86e01
		Damping Ratio	7.53e-03	8.86e-06
	Specified Height	Spring Constant	7.07e03	2.16e02
		Damping Ratio	7.54e-03	3.27e-05

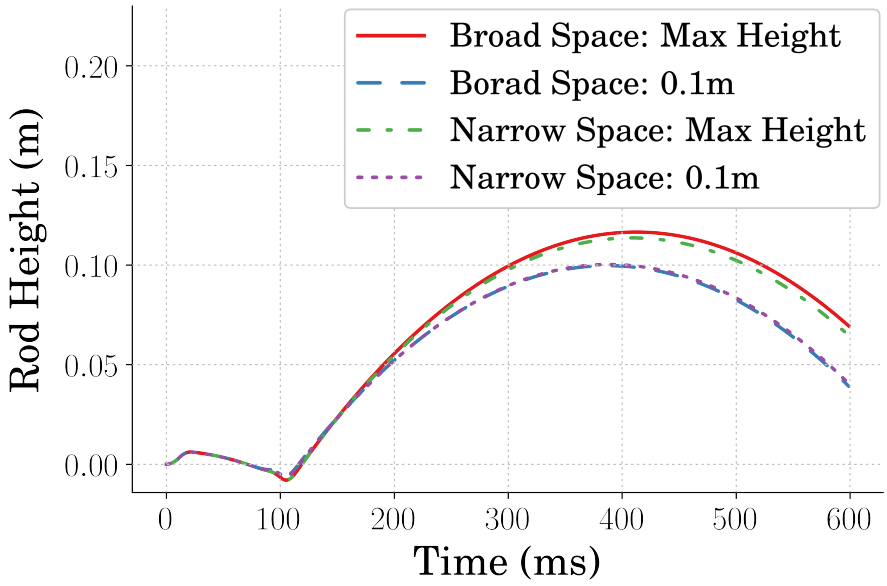


Figure 24. Height vs Time of Average Optimal Designs

agents that learned designs to jump to a specified height, it can be seen that there is a high variance in spring constant throughout training. However, the majority of agents converge to a specific design, lowering the variance. The same can be seen in the damping ratio; however, the variance is mitigated significantly earlier in training. The agents which were learning designs that maximized height found them with very little variance in terms of spring constant and damping ratio.

4.6.3 Average Design Performance.

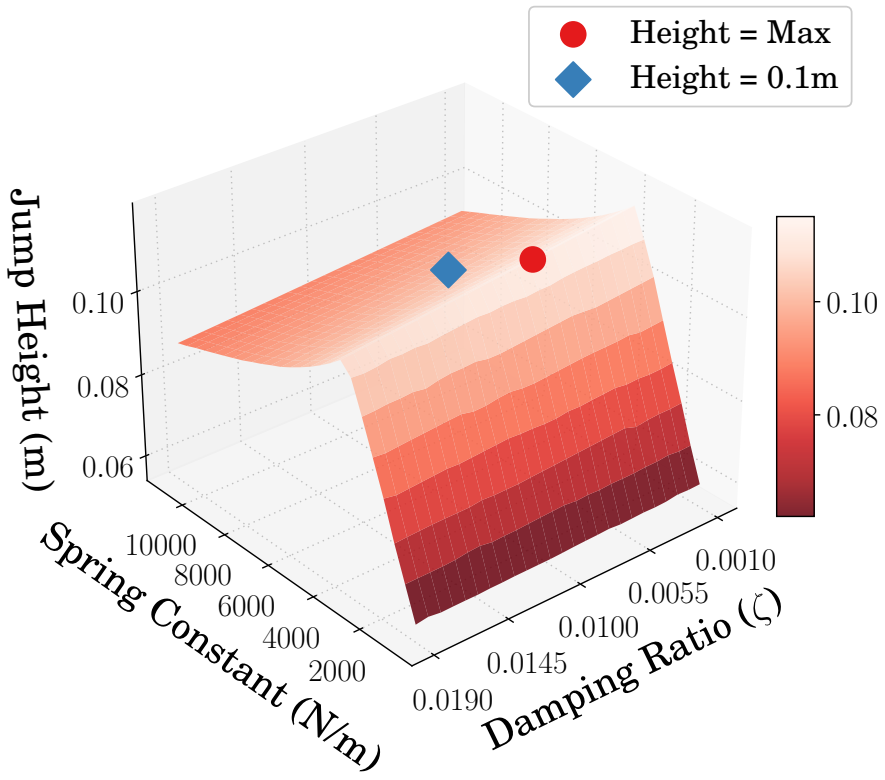
The final mean and standard deviation of the design parameters for the two different cases are presented in Table 4. Figure 24 shows the jumping performance of the mean designs learned for both cases tested. The agents tasked with finding designs to jump to the specified 0.1 m, did so with minimal error. The difference seen in maximum height reached between the two cases represents the difference in the nominal damping ratio within the design spaces. The peak heights from Figure 24 for the maximum height designs can be compared to Figures 18a and 18b to show that the agents learned designs nearing those achieving maximum performance.

Additionally, Figure 25 shows the average designs, and their performance, against the design space performance data. It is apparent that in the case of the narrow design space, where the optimal design for maximizing height is more prominent, the design found is one that is approaching optimal performance. In the case of the wide design space, where the design that optimizes height is less prominent, it is apparent that the optimal design found by the agent is only nearing the optimal design. In both design space cases, the design found to force the monopode to jump to the specified height was found within the higher values of the spring constant range, even though values did exist in the lower range that would satisfy the jumping condition.

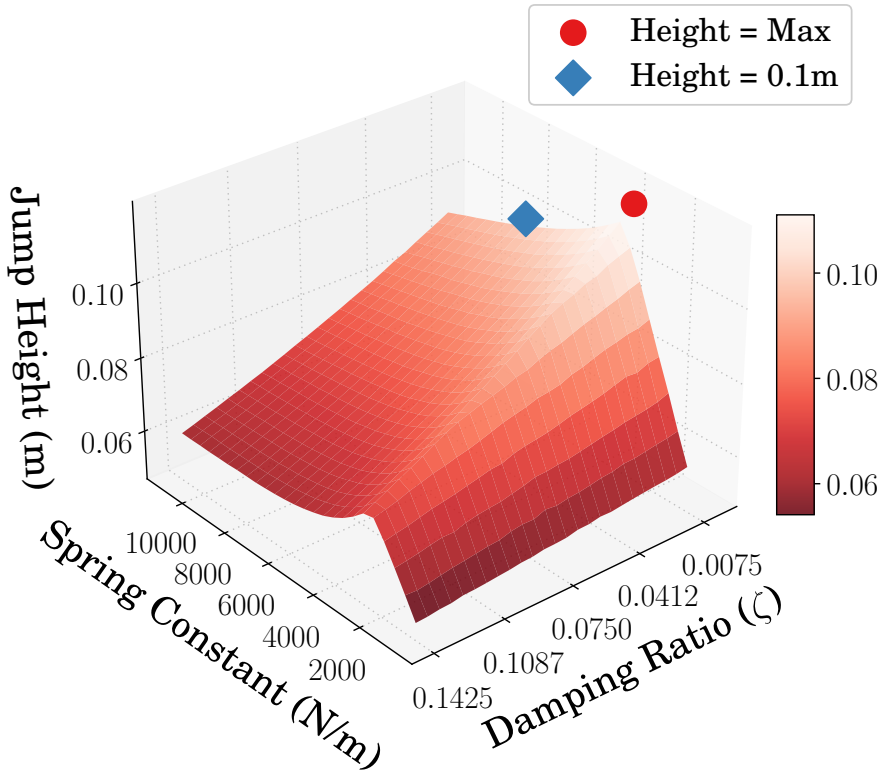
4.7 Conclusion

The monopode model was used in conjunction with a predetermined control input to determine if a reinforcement learning algorithm (TD3) could be used to find optimal performing design parameters regarding jumping performance. This work was done in part to determine if reinforcement learning could be used as the mechanical design learner for an intelligent concurrent design algorithm. It was shown that when providing an agent with a design space that was smaller in size with a more prominent optimal value, the agents performed well in finding design parameters which met the

performance constraints. The designs found were lower in variance as well, even in the in the case where the algorithm was tasked with finding a design for a specific performance within the range of possible performances. It was additionally shown that when provided with larger design space, that additionally had many values closer to the optimal value, the agents still excelled at finding design parameters that performed close to optimal. The parameters found where higher in variance however, as expected, particularly in the case where a design was to be found to generate a specific performance. This was due to the number of viable design options that would satisfy the performance requirements. It should be concluded ultimately that utilizing an RL algorithm, such as TD3, for the mechanical design aspect of a concurrent design method, is a viable solution.



(a) Average Design Performance Within Narrow Design Space



(b) Average Design Performance Within Wide Design Space

Figure 25. Reference Jumping Performance of the Monopode

V Concurrent Design of the Monopode System

Finding a control architecture for a mechanically defined system is often the the workflow for generating a controlled robotic system. However, the mechanical system is not always a simple one and generating a controller for it may require a more complex workflow. It is of interest as well to allow the mechanical parameters of the system, and therefore the system description, to be fluid, allowing for a more optimal mesh between controller and system. Designing the system and control input in unison has been researched and is ofter referred to as concurrent design. This strategy has been used to develop better performing mechatronics systems [17]. More recent work has used advanced methods such as evolutionary strategies to define robot design parameters [46]. In addition to evolutionary strategies, reinforcement learning has been shown to be a viable solution for concurrent design of 2D simulated locomotive systems [45]. This is further shown to be a viable method by demonstrating more complex morphology modifications in 3D reaching and locomotive tasks [19]. However, these techniques have not yet been applied to flexible systems for locomotive tasks. In this chapter, a novel definition of a concurrent design architecture is purposed to find an optimal design and controller for the monopode jumping system defined in Chapter 2.

5.1 Concurrent Design Architecture

To define a concurrent design process utilizing RL, an algorithm is proposed which utilizes two instances of the TD3 algorithm creating an inner and an outer loop. The first instance, being responsible for learning the control policy, will be instantiated in a similar fashion to the what is seen in Chapter 2. It is the outer loop of the concurrent design process. The second instance, being responsible for learning the mechanical design, is instantiated within the outer loop and in a similar fashion to what is seen in Chapter 4. The second instance is the inner loop of the concurrent design process. A key aspect of the inner loop instantiation is that rather than using a

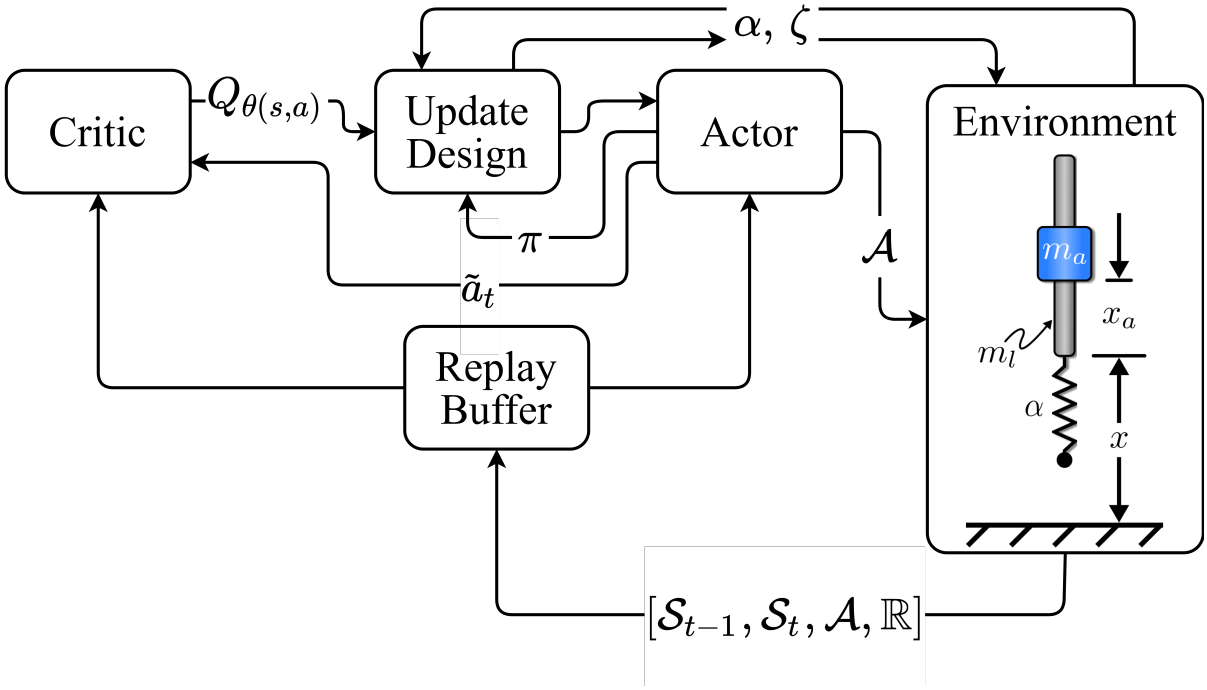


Figure 26. Concurrent Design Architecture

pre-defined control input, like what was shown in Chapter 4, the simulation will be using the input being trained by the outer loop instantiation. Figure 26 shows the general flow of information for the concurrent design algorithm. The inner loop, replicating the work from Chapter 4, is within the "Update Design" block. The description of the inner loop was shown in Figure 17, only the defined input that was used in Chapter 4 is replaced in this diagram by the current policy π. A detailed algorithm, describing the process presented in Figure 26, is presented in Appendix 7.

5.2 Mechanical Design Update

5.2.1 Discrete vs. Continuous. As is further discussed in Appendix 7, there are two different methods for implementing the inner loop of the concurrent design algorithm. The first is called the *discrete* method, where at each environment design update, the model learning the design is instantiated fresh and learns a design from scratch. The second method is called *continuous*, where at each environment

update the model learning the design is saved and reloaded so that the model that is learning a design is the same over the course of the controller model training.

5.2.2 Averaging Design Policies. In Chapter 4, average designs found from 50 different policies are shown, and it can be seen that design choice can vary between policies, depending on factors such as the reward the policy receives and the design space limits. To replicate the results in Chapter 4, and to ensure that the mechanical design inner loop did not suffer from a single policy finding a locally optimal design, n design policies were instantiated at each design update. The average of the n designs found was used to update the environment within the outer loop. This methodology was applied to both the discrete and the continuous methods of mechanical design update. In this work, n was set as five as it proved to resolve one of the policies finding a design which strayed from the average design.

5.2.3 Differing Reward Types. Depending on the reward passed to the design update inner loop, the performance of the control policy may immediately increase or decrease when the design is updated. For example, if the rewards for both inner and outer loops reward the same metric, the control policy within the outer loop should see an increase in performance after a design update. If however, the rewards for the inner and outer loops differ, where the outer loop might reward height for example and the inner rewards efficiency, the control policy may experience an immediate decrease in performance after a design update. Utilizing differing rewards might serve as a tool to generate designs where, as suggested, the control policy is defined to accomplish a task and the design is optimized to allow the control policy to do that task efficiently.

5.2.4 Design Update Rate. When the inner loop makes an update to the environment that the policy within the outer loop is being trained on, the policy should

see an immediate performance change. These step like changes are likely to result in a policy learning less data efficiently. Because of this, a new hyperparameter is introduced being the rate at which the design is updated. As is shown in Appendix 7, the design is updated in line with the control policy only not at the same rate. Regardless, the design update rate is directly tied to the policy update. It is suggested that for this architecture the design is updated every n policy updates where n depends on the complexity of the control policy being trained. For more complex control policies, where the learning process might require more environment interactions to learn, n should be set to a higher value so that the control policy can learn a design without the added difficulty of dynamic environment design parameters.

5.2.5 Design Space Limitations. The work shown in Chapter 4 discussed two design space limits, both with differing nominal values as well as space upper and lower limits. This raises the concern of design space choice for the concurrent design architecture. It was shown in Chapter 4, that the design space had an effect on the final designs learned, both in terms of iterations to converge and in variance seen across network initializations. In the work presented in this Chapter, the design space nominal values and limits were chosen to partially replicate the work in Chapter 4. Regarding the spring constant, the nominal value used was what was presented in Table 1, with the limits being set to $\pm 90\%$ of the nominal value. As for the damping ratio, since it was shown to have learned extremely low damping ratios in most learning cases, the design space was set as a range from 0 to 0.01.

5.3 Environment Definition

5.3.1 Learning the Controller. The outer loop of the concurrent design architecture was defined similar to what was shown in Chapter 2, where a traditional RL environment aligning with the standards set by OpenAI for a Gym environment was created [38]. The monopode, also described in Section 2.1, was used to define the

environment and evaluate the methods discussed in this chapter. The action and observation spaces were defined, respectively, as follows:

$$\mathcal{A} = [\ddot{x}_{a_t}] \tag{20}$$

$$\mathcal{S} = [x_{a_t}, \dot{x}_{a_t}, x_t, \dot{x}_t] \tag{21}$$

where x_t, $\dot{x}_t$ were the monopode's position and velocity at time t, and x_{a_t}, $\dot{x}_{a_t}$ and $\ddot{x}_{a_t}$ were the actuator's position, velocity and acceleration, respectively.

Differing from the evaluation completed in Chapter 2, only the stutter jumping command was evaluated. Therefore, the stopping conditions for the environment were either the monopode completing two jumps or the time step limit. For this work, the time step limit was set to 400 steps at 0.01 seconds per step. Additional information regarding the stutter jump command was provided in Section 2.1. Futhermore, the values of the monopode's nominal constants were shown in Table 1 within Section 2.1.

5.3.2 Learning the Design. To allow the inner loop RL algorithm to find a mechanical design within the outer control loop, a second reinforcement learning environment was defined, again conforming to the OpenAI Gym standard [38], in a similar fashion to what was discussed in Chapter 4. Differing from Chapter 4, however, the control input, rather than being fixed, is captured from the outer loop and used to evaluate the performance of different design choices. The mechanical parameters the algorithm was tasked with optimizing were the spring constant and the damping ratio of the monopode jumping system.

At each episode during training, the algorithm's policy selected a set of design parameters from a distribution of predefined parameter ranges. The actions applied, $\mathcal{A}$, and transitions saved, $\mathcal{S}$, from the environment were defined as follows:

$$\mathcal{A} = \{\{a_\alpha \in \mathbb{R} : [-0.9\alpha, 0.9\alpha]\},\ \{a_\zeta \in \mathbb{R} : [0, 0.01]\}\} \tag{22}$$

$$\mathcal{S} = \left\{ \sum_{t=0}^{t_f} x_t,\ \sum_{t=0}^{t_f} \dot{x}_t,\ \sum_{t=0}^{t_f} x_{at},\ \sum_{t=0}^{t_f} \dot{x}_{at} \right\} \tag{23}$$

Table 5. Outer Loop TD3 Training Hyperparameters

Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	1000 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	5 Random Seeds

where α is the nominal spring constant the monopode, x_t and $\dot{x}_t$ are the monopode's rod height and velocity steps, and x_{a_t} and $\dot{x}_{a_t}$ are the monopode's actuator position and velocity steps, all captured during simulation. The nominal values of the constants were shown in Table 1 within Section 2.1.

5.4 Deploying the Algorithm

As is discussed in Section 5.1, an inner and outer instantiation of the TD3 algorithm are generated to create the concurrent design architecture. Similar to what was practiced in previous chapters, multiple instances of the algorithm were run to evaluate the ability of the architecture to perform with different network initializations. Ten total instances were evaluated so that average performance data could be collected. This proved to be an effective amount needed to evaluate variances seen in performance.

The outer loop was instantiated in a similar fashion as to what was discussed in Chapter 2, only the number if total training steps was increased from 500k to 750k. This was done as the control policy was anticipated to be more difficult to learn given the environments parameters would be changing due to the inner loop. The training hyperparameters used for the outer loop instantiation of the TD3 algorithm are

Table 6. TD3 Training Hyperparameters

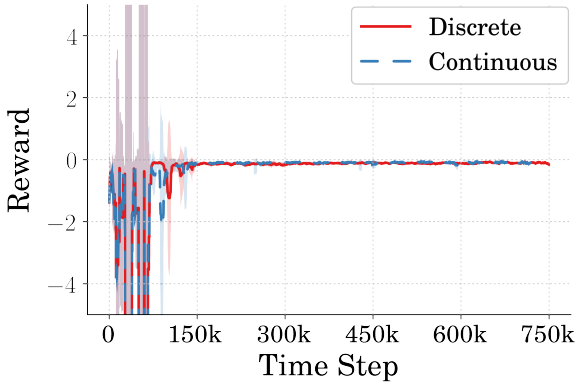
Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	100 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	5 Random Seeds

presented in Table 5.

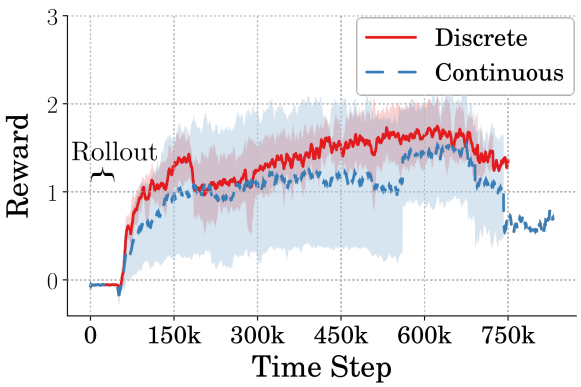
The inner loop, similar to the outer, was instantiated in a similar fashion to what is shown in Chapter 4. This instantiation of the TD3 algorithm was created within the outer instantiation of the TD3 algorithm, using the policy being trained within the outer loop to optimize the environment used in the outer loop. The number of training steps taken by each instantiation was 1000, to best replicate the results from Chapter 4. Additional hyperparameters used for each of the five instantiations of the inner TD3 algorithm are presented in Table 6.

5.5 Discrete vs Continuous Designs

5.5.1 Reward vs Learning Step. It if firstly if interest to evaluate the learning differences between the discrete and continuous methods used to define a concurrent design architecture. Figure 27 shows the reward received during training for the high and efficient jumping strategies comparing the discrete and continuous implementations. The design update rate for this evaluation was set to 1000 for both methods. Figure 27a, comparing the rewards during training for the efficient jumping control strategies show little difference between the two methods. Both the design

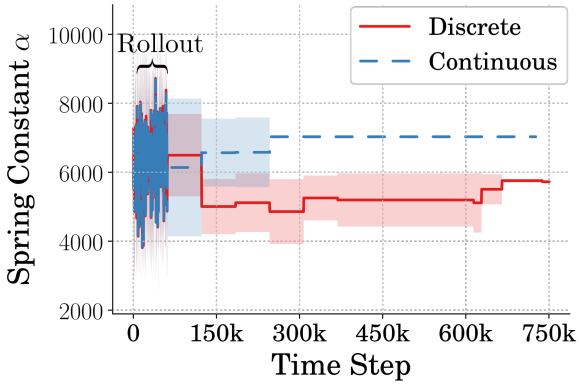


(a) Reward During Training for Efficient Design

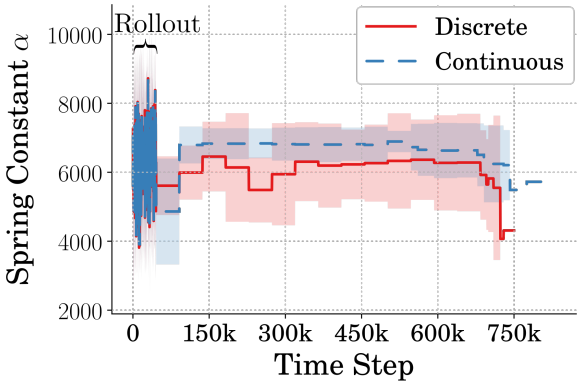


(b) Reward During Training for Height Design

Figure 27. Reward During Training for Discrete and Continuous Implementation Methods of Concurrent Design



(a) Spring Constant Learned During Training for Efficient Design

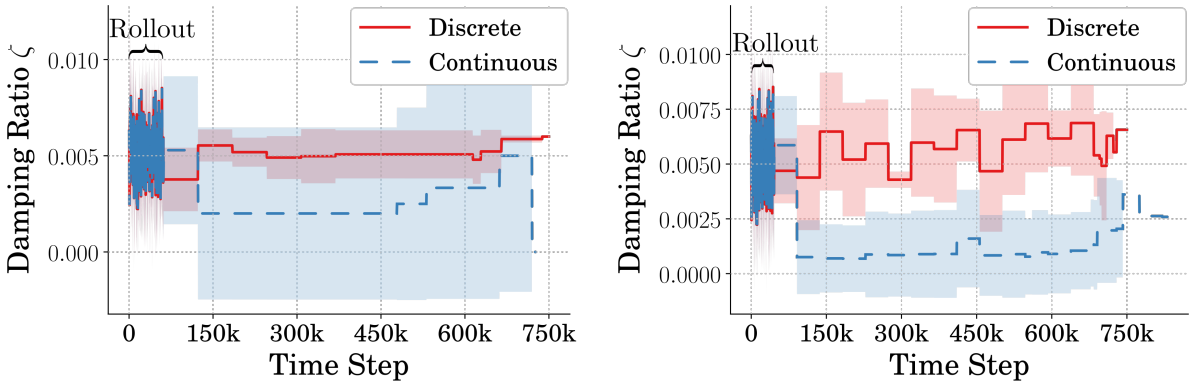


(b) Spring Constant Learned During Training for Height Design

Figure 28. Spring Constant During Training for Discrete and Continuous Implementation Methods of Concurrent Design

architecture display rapid learning capabilities similar to what was shown for the efficient controller types in Chapter 2.

Figure 27b, comparing the rewards during training for the high jumping design architecture, show some differences between the two methods. The discrete method learns a higher performing policy and one that is less susceptible to performance loss due to an over fitting design policy towards the end of training. Both methods however, have policies that are loosing rewards due to over fitting towards the end of training.



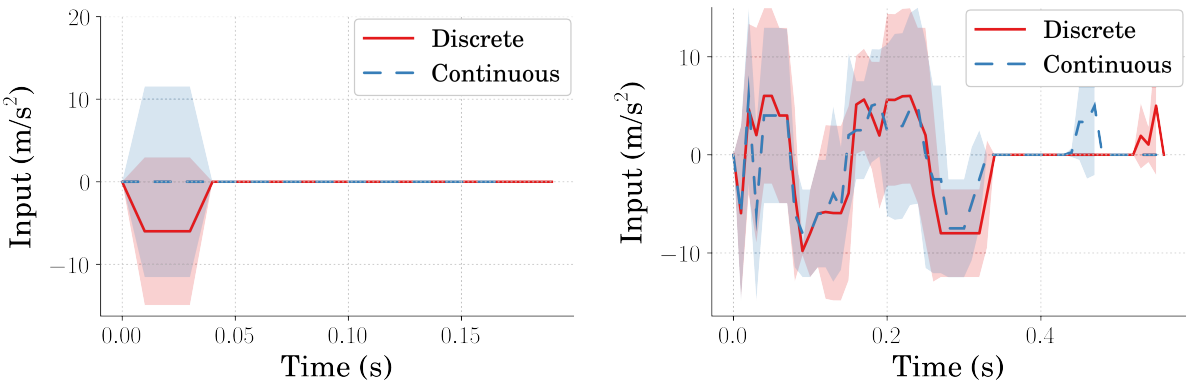
(a) Damping Ratio Learned During Training for Efficient Design

(b) Damping Ratio Learned During Training for Height Design

Figure 29. Damping Ratio During Training for Discrete and Continuous Implementation Methods of Concurrent Design

5.5.2 Designs Learned. It is also of interest to evaluate the differences in the learned designs between the discrete and the continuous mechanical update methods. Figure 28 shows the learning of the spring constant for the efficient and high jumping concurrent design types. For both jumping cases, it is apparent that the continuous method, where the policy for defining a design is kept constant throughout the control policy training, performs in a more continuous manner. In both jumping cases as well, the continuous method learns higher spring rates. Furthermore, there are obvious differences between the methods regarding standard deviation across outer loop instantiations, where the continuous method has much less deviation. This is not a direct translator to performance consistency however since each instantiation of the control policy will have its own design. It does however show that across different concurrent design instantiations, the continuous method learns a more consistent design.

Figure 29 shows the learning of the damping ratio for the efficient and high jumping concurrent design types. Firstly, in both cases, the continuous method learned a lower damping ratio that the discrete method. Similar to the learning of the spring rate, the continuous method seems to learn a more continuous design throughout the control policy training as well. This is particularly obvious in the case where the policy



(a) Average Input for Efficient Concurrent Designs

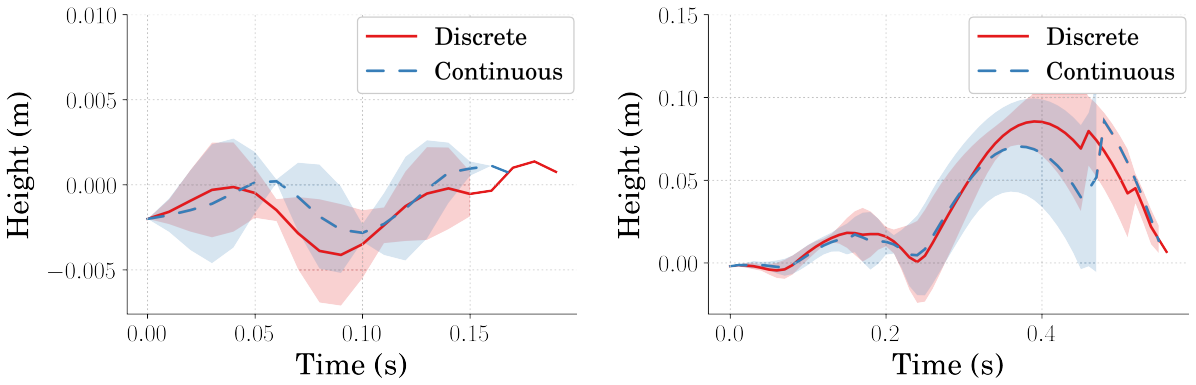
(b) Average Input for High Jumping Concurrent Designs

Figure 30. Average Controller Performance for Efficient Concurrent Designs

is learning a high jumping strategy. Regarding standard deviation, it appears for the efficient jumping strategy, the damping ratios found across concurrent design instantiations vary greatly. Whereas, in the case of the high jumping strategy, the difference in standard deviation is more more similar between the methods. The increase in standard deviation regarding damping ratio is similar to the results shown in Chapter 4 where changes in damping ratio were shown to be less critical than changes in spring rate.

5.5.3 Resulting Input and Jumping Performance. Evaluating the learning process and learned designs, comparing the continuous and discrete methods, can only give some intuition regarding the differences between the methods. Each instantiation of the concurrent design process has its own learned controller and associated design. Because of this, differing designs might not have as great of an effect on final performance because the associated controller was trained for said design specifically. As such, it is of interest to evaluate the differences seen in performance between the two methods discussed.

Starting with the learned inputs for the efficient and high jumping designs, shown in Figure 30, it is apparent that the learned input for the efficient jumping type



(a) Average Jumping Performance for Efficient Concurrent Designs

(b) Average Jumping Performance for High Jumping Concurrent Designs

Figure 31. Average Controller Performance for High Jumping Concurrent Designs

is not one which would accomplish a high jump. Figure 30a, showing the learned commands for the efficient jumping designs, shows that the controllers for the discrete method will accelerate in the negative direction for a short period of time and then stop for the rest of the command. As for the continuous method, across network initializations, the command directions are split such that the average is to stay stationary.

The inputs for the high jumping designs are shown in Figure 30b. For this jump type, the timing and magnitude of the actuators accelerations are seemingly very similar throughout most of the command. In general, the discrete method does appear to learn a command that is less noisy, and, as will be shown, this causes a difference regarding jumping performance. The largest difference that can be seen, is towards the end of the command, where both methods result in a control policy that accelerates the actuator upwards.

Looking at Figure 31, the average timeseries jumping performances of the efficient and high jumping strategies for the discrete and continuous methods can be seen. Figure 31a, showing the jumping performance of the efficient concurrent design, replicates what was seen from the inputs that generated the jumps. It is apparent, that in comparison to the high jumping concurrent design, the increased complexity of the

efficient command makes for a difficult policy to learn. That is, the policy cannot find a consistent control strategy that can overcome the increased complexity of a dynamic environment. The input, only accelerating in one direction for a short period of time, allows the monopode to jump just above the zero point so it can stutter bounce rather than stutter jump.

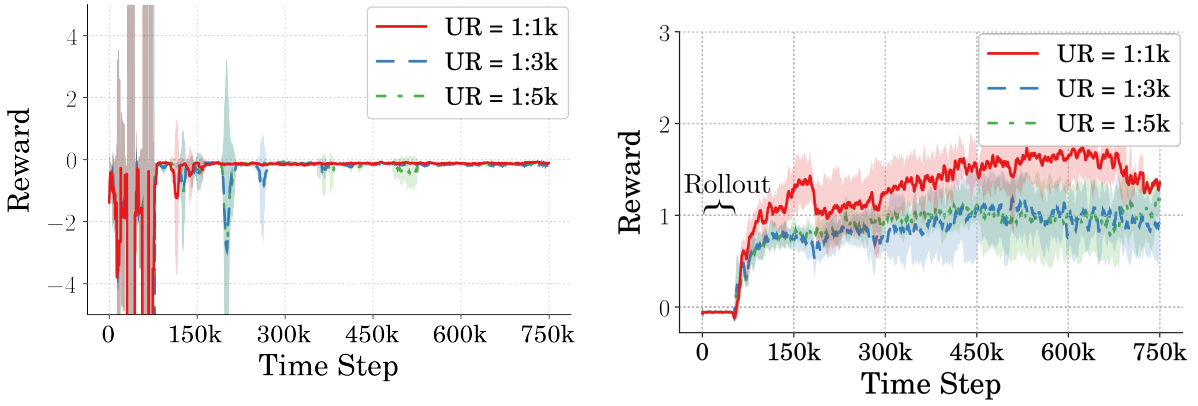
The jumping performance of the high jumping concurrent designs are shown in Figure 31b. Firstly, similar to the inputs that generate these jumps, the two methods discussed create similar stutter jumps. The continuous method, having an input comprised of noisy commands does not generate as much energy within the spring and does not jump the monopode as high using just the spring energy. It instead relies on the final upward acceleration of the actuator to get to its maximum height. The discrete method, though employing the same technique, waits longer and therefore is not able to capitalize on the technique for achieving its maximum height. In regards to consistency, the discrete methods learns concurrent designs that result in commands that more consistently jump higher. Additionally, the discrete methods leads to commands that perform better than the continuous method in the best cases.

5.6 Effects of Differing Update Rate

...

All the data is in this section. I couldn't bring myself to complete the last two paragraphs. Most of this will likely need re-writing to make it cleaner and less blah.

In addition to discovering the differences in learning and final design performances due to employing the discrete/continuous method, it is also of interest to discover the difference when changing the design update rate. The update rate number, again, is tied to the rate that the control policy is updated in the outer loop. In the previous section, the results presented where trained using a design update rate of 1000, i.e. the design was updated every 1000 control policy updates. It was shown that in general the efficient designs could not learn a concurrent designs that successfully



(a) Reward During Training for Efficient Design

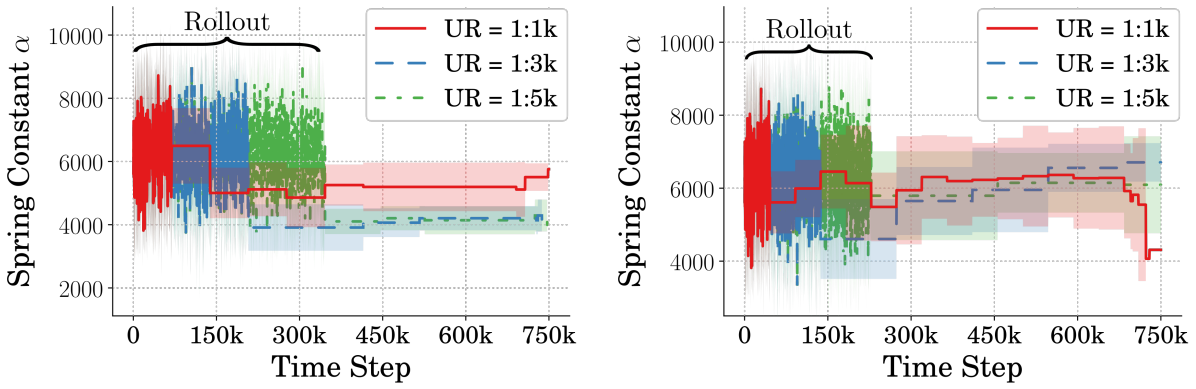
(b) Reward During Training for Height Design

Figure 32. Reward During Training for Discrete and Continuous Implementation Methods of Concurrent Design

completed a stutter jumping command. It was also shown that the discrete method produced designs that performed more consistently and better in general regarding the best case for the high jumping strategy. As such, in this section, design update rates of 1000, 3000, and 5000 are evaluated and compared using the discrete method of mechanical design updating.

5.6.1 Reward vs Learning Step. Figure 32 shows the rewards during training for each of the three mechanical update rates evaluated for the high and efficient jumping strategies for the discrete update method. In the case of the efficient strategy, seen in Figure 32a, there are little differences found when altering the update rate. However, it seems that lowering the rate at which the mechanical design is updated, results in the policy drastically changing in the middle of learning. This is likely because, as discussed earlier, as the design changes, the policy will see an immediate change, and in the when lowering the update rate, the design changes less often and more drastically.

As for the high jumping strategies, seen in Figure 32b, there are more obvious differences. Firstly, it is apparent that lowering the update rate decreases policy



(a) Spring Constant Learned During Training for Efficient Design

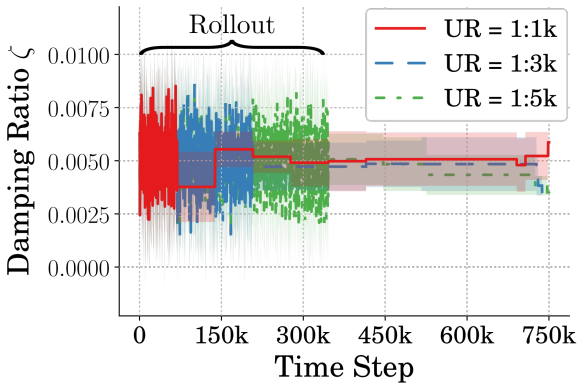
(b) Spring Constant Learned During Training for Height Design

Figure 33. Spring Constant During Training for Discrete and Continuous Implementation Methods of Concurrent Design

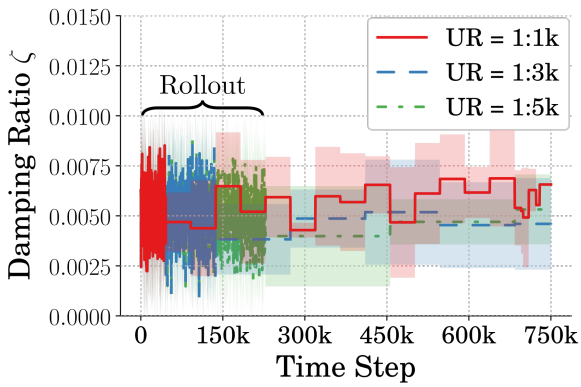
performance regarding reward received. Additionally, in the beginning of training, there is less variance in the performance of the policies across network instantiations. This is directly related to how often the mechanical design is updated, and will be further explain when evaluating the mechanical designs learned. Finally, it should be noted that over fitting within the design policy becomes less apparent as the update rate decreases.

5.6.2 Designs Learned. The learned spring constant value during training, for both the efficient and high jumping strategies, is displayed for the three different update rates being evaluated in Figure 33. It is most obvious that in both cases, the increase in update rate directly translates to an increase in rollout time before the design update policy can begin learning designs. This was shown to translate to the rewards in that the concurrent design instantiations with lower update rates required more steps to learn better performing policies.

The learning of the spring constants for the three different update rate being evaluating are shown in Figure 34. In the case of the efficient strategy, shown in Figure 34a, there is little difference outside of the increasing rollout period. The As for the high jumping strategies, shown in Figure 34b, the differences are more pronounced.

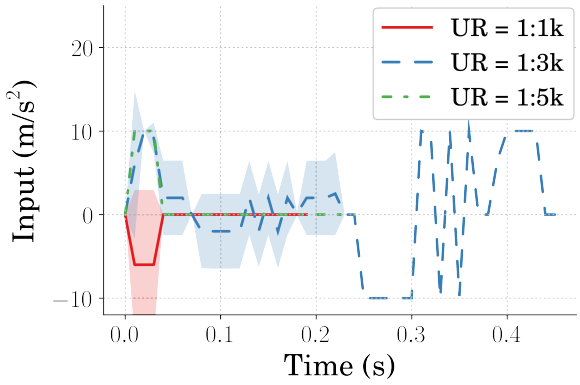


(a) Damping Ratio Learned During Training for Efficient Design

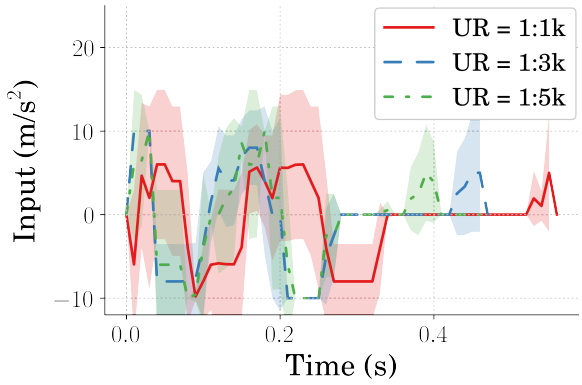


(b) Damping Ratio Learned During Training for Height Design

Figure 34. Damping Ratio During Training for Discrete and Continuous Implementation Methods of Concurrent Design



(a) Average Input for Efficient Concurrent Designs

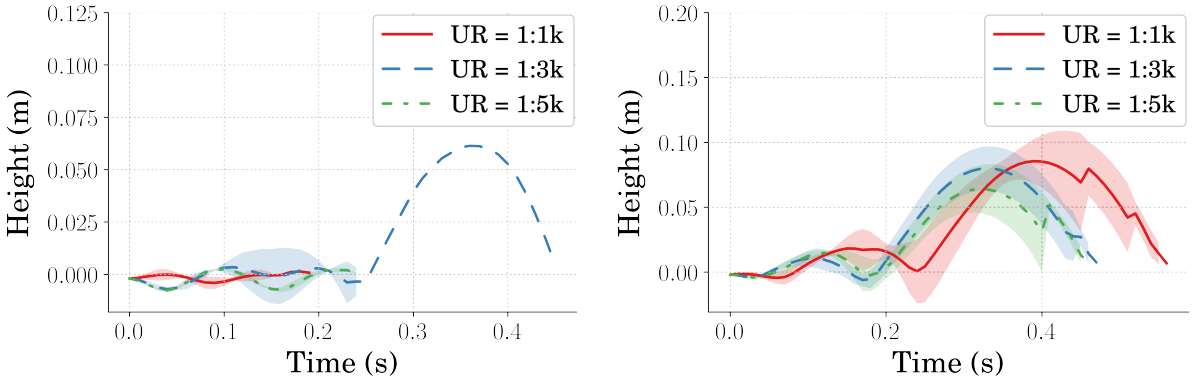


(b) Average Input for High Jumping Concurrent Designs

Figure 35. Average Controller Performance for Efficient Concurrent Designs

Primarily, as the update rate decreases, the consistency of the designs learned throughout training increases. This can be explained in that the control policy has more time to train a high performing policy before the design changes and therefore is able to better train a controller on the most current design.

5.6.3 Resulting Input and Jumping Performance. Average learned input for the efficient and high jumping strategies for the three different update rates evaluated are shown in Figure 35.



(a) Average Jumping Performance for Efficient Concurrent Designs

(b) Average Jumping Performance for High Jumping Concurrent Designs

Figure 36. Average Controller Performance for High Jumping Concurrent Designs

Average jumping performance resulting from the learned input for the efficient and high jumping strategies for the three different update rates evaluated are shown in Figure 35.

5.7 Conclusion

A concurrent design architecture was defined and evaluated using the monopode jumping system discussed in Chapter 2. The architecture consists primarily of an inner and outer loop where the outer is defined to learn a control policy for an environment design that can be updated within the inner loop. The architecture was evaluated to generate an efficient and a high jumping concurrently designed system and controller. Two methods for implementing the inner loop mechanical design update, being discrete and continuous, were discussed. The differences in learning and final concurrent design performance between these two methods were shown and the use cases for each were discussed. Additionally, a new hyperparameter, being the rate at which the mechanical design is updated within the outer loop, was introduced. It was shown that changes to the update rate are of primary importance when altering the complexity of the control policy being trained. In general the concurrent design architecture struggled to find designs for the efficient jumping strategy due the increased complexity of the control

policy. Regarding the high jumping strategy, the designs found averaged higher performance than control policy trained on static mechanical designs. Additionally, in the best case, the high jumping design learned also outperformed the best case of a policy trained on a static environment. It should be concluded than, that this type of architecture could be used where finding a concurrent design is of interest. However, the reward shape passed to the control policy should be considered carefully, as fragile policies become more fragile with dynamically changing environments.

VI Conclusion and Discussion

6.0.1 Conclusion. ...

Highlight the work in the thesis.

6.0.2 Future Research. ...

Discuss the work enabled by the work completed in this thesis.

VII Appendix: Concurrent Design Algorithm

The algorithm presented, is in simple terms, an instantiation of the TD3 algorithm within an instantiation of the TD3 algorithm. Line 18, highlighted with green text, denotes the start of the inner loop and is responsible for learning a mechanical design similar to what is shown in Chapter 4. This inner loop runs before the control policy is updated depending on a hyperparameter that is related to how often the mechanical design should be updated. The lines on the upper and lower end of the green text, create what is considered the outer loop, which is responsible for learning a control policy.

The inner and outer loop are concurrent in that the inner loop takes the policy being trained in the outer loop and uses it to simulate the environment to learn an optimal mechanical design. Once the design has been learned, the inner loop passes the design back to the outer loop so that it can modify the environment which it is using to train a control policy.

7.0.1 Averaging n Learned Designs. To best replicate the results shown in Chapter 4, rather than instantiating a single instance of the TD3 algorithm within the outer loop to learn a design, n instances are created and the average design found is used. Line 19, highlighted in blue text shows the start of the looping through n instances of the learning of mechanical designs. Line 45, also highlighted in blue, shows the averaging of the n designs before the environment within the outer loop is modified on line 47.

7.0.2 Discrete vs. Continuous. There are two methods for implementing the inner loop. The first being at every instantiation, the policy parameters learning a design (line 20) are initialized from nothing creating a network without learned intuition regarding good designs. Removing the *load* command on line 20 of the

algorithm replicates this method. As for the second method, rather than starting with an untrained policy every design update, the policy is saved and then reloaded the next time the design is updated. During the first design update, n policies are created and learn a design. They are then saved along with their replay buffers so they can be reloaded to continue updating the mechanical design. The differences between these two methods are presented in Section 5.5.

7.0.3 Design Update Rate. This algorithm presents an additional hyperparameter on top of the ones already present, being the rate at which the design is updated within the outer loop. This decision is displayed on line 18 in green text. The findings of altering this hyperparameter are presented in Section 5.6.

1: Input: initialize policy parameters θ_{ctr}, Q-function parameters $\phi_{1,ctr}$ and $\phi_{2,ctr}$ and empty replay buffer, $\mathcal{D}_{ctr}$

2: Set target parameters equal to main parameters: $\theta_{ctr,targ} \leftarrow \theta$, $\phi_{1,ctr,targ} \leftarrow \phi_{1,ctr}$, $\phi_{2,ctr,targ} \leftarrow \phi_{2,ctr}$

3: **while** Not Converged **do**

4: Observe system state s_{ctr} and select action $a_{ctr} = \text{clip}(\pi_{\theta_{ctr}}(s_{ctr}) + \epsilon, a_{ctr,low}, a_{crt,high}), \quad \epsilon \sim \mathcal{N}$

5: Execute the action a in the environment

6: Observe the next state s'_{ctr} and the reward r_{ctr} (verify if the state s'_{ctr} is a terminal state d_{ctr})

7: Store $(s_{ctr}, a_{ctr}, r_{ctr}, s'_{ctr}, d_{ctr})$ in the replay buffer $\mathcal{D}_{ctr}$

8: **if** s'_{ctr} is terminal **then**

9: Reset environment

10: **end if**

11: **if** Update Parameter % Update Frequency **then**

12: **for** j in range number of updates **do**

Sample random batch of transitions from buffer $\mathcal{R}_{ctr}$

Compute target actions:

$$a_{ctr} = \text{clip}(\pi_{\theta_{ctr,targ}}(s'_{ctr}) + \text{clip}(\epsilon, -c, c), a_{ctr,low}, a_{ctr,high}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

Compute targets:

$$y(r_{ctr}, s'_{ctr}, d_{ctr}) = r_{ctr} + \gamma\,(1 - d_{ctr}) \min_{i=1,2} Q_{\phi_{ctr,targ,i}}(s'_{ctr}, a_{ctr}(s'_{ctr}))$$

Update the Q-function by way of gradient decent:

$$\nabla_{\theta_{ctr}} \frac{1}{|B|} \sum_{(s_{ctr}, a_{ctr}, r_{ctr}, s'_{ctr}, d_{ctr}) \epsilon B} \left(Q_{\phi_{ctr,i}}(s_{ctr}, a_{ctr}) - y(r_{ctr}, s'_{ctr}, d_{ctr})\right)^2 \quad \text{for } i = 1, 2$$

if j % Policy Delay is 0 **then**

  if Update Design % Update Frequency **then**

    for i in range n instantiations of a mechanal design policy **do**

      Input: initialize/load policy parameters θ_{des}, Q-function parameters $\phi_{1,des}$ and $\phi_{2,des}$ and empty replay buffer, $\mathcal{D}_{des}$

      Set target parameters equal to main parameters: $\theta_{des,targ} \leftarrow \theta$, $\phi_{1,des,targ} \leftarrow \phi_{1,des}$, $\phi_{2,des,targ} \leftarrow \phi_{2,des}$

      while Not Converged **do**

        Observe system state s_{des} and select a design $a_{des} = \text{clip}(\mu_{\theta_{des}}(s_{des}) + \epsilon, a_{des,low}, a_{crt,high}), \quad \epsilon \sim \mathcal{N}$

        Simulate the design a in the environment using $\pi_{\theta_{ctr}}$

        Observe the simulation s'_{des} and the reward r_{des} (verify if the state s'_{des} is a terminal state d_{des})

        Store $(s_{des}, a_{des}, r_{des}, s'_{des}, d_{des})$ in the replay buffer $\mathcal{D}_{des}$

        if s'_{des} is terminal **then**

          Reset environment

        end if

        if Update Parameter % Update Frequency **then**

          for j in range number of updates **do**

Sample random batch of transitions from buffer $\mathcal{R}_{des}$

Compute target actions:

$$a_{des} = \text{clip}(\mu_{\theta_{des,targ}}(s'_{des}) + \text{clip}(\epsilon, -c, c), a_{des,low}, a_{des,high}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

Compute targets:

$$y(r_{des}, s'_{des}, d_{des}) = r_{des} + \gamma\,(1 - d_{des}) \min_{i=1,2} Q_{\phi_{des,targ,i}}(s'_{des}, a_{des}(s'_{des}))$$

Update the Q-function by way of gradient decent:

$$\nabla_{\theta_{des}} \frac{1}{|B|} \sum_{(s_{des}, a_{des}, r_{des}, s'_{des}, d_{des}) \epsilon B} (Q_{\phi_{des,i}}(s_{des}, a_{des}) - y(r_{des}, s'_{des}, d_{des}))^2 \quad \text{for } i = 1, 2$$

if j % Policy Delay is 0 **then**

Update policy by one step of gradient decent:

$$\nabla_{\theta_{des}} \frac{1}{|B|} \sum_{s_{des} \epsilon B} Q_{\phi_{des,1}}(s_{des}, \mu_{\theta_{des}}(s_{des}))$$

Update target networks by:

$$\phi_{des,targ,i} \leftarrow \rho \phi_{des,targ,i} + (1 - \rho)\phi_{des,i} \quad \text{for } i = 1, 2$$

$$\theta_{targ} \leftarrow \rho \theta_{targ} + (1 - \rho)\theta$$

end if

end for

end if

end while

Capture the final design that was learned as d_i

end for

Compute the average of the designs:

$$\mathbb{D} = \frac{\sum_{i=1}^{n} \mathrm{d}_i}{n}$$

if Update Method is *continuous* **then**

Save the policies, μ_i, for $i = 0 \ldots n$,

end if

end if

Update the environment with the learned design $\mathbb{D}$

Update policy by one step of gradient decent:

$$\nabla_{\theta_{ctr}} \frac{1}{|B|} \sum_{s_{ctr} \epsilon B} Q_{\phi_{ctr,1}}(s_{ctr}, \pi_{\theta_{ctr}}(s_{ctr}))$$

Update target networks by:

$$\phi_{ctr,targ,i} \leftarrow \rho \phi_{crt,targ,i} + (1-\rho)\phi_{ctr,i} \quad \text{for } i = 1, 2$$

$$\theta_{crt,targ} \leftarrow \rho \theta_{ctr,targ} + (1-\rho)\theta_{ctr}$$

end if

end for

end if

end while

Bibliography

[1] Folkertsma, G. A., Kim, S., and Stramigioli, S., "Parallel stiffness in a bounding quadruped with flexible spine," *IEEE International Conference on Intelligent Robots and Systems*, pp. 2210–2215, 2012.

[2] Vaughan, J., "Jumping Commands For Flexible-Legged Robots," 2013.

[3] Park, H. W., Wensing, P. M., and Kim, S., "High-speed bounding with the MIT Cheetah 2: Control design and experiments," *International Journal of Robotics Research*, vol. 36, no. 2, pp. 167–192, 2017.

[4] Seok, S., Wang, A., Chuah, M. Y., Hyun, D. J., Lee, J., Otten, D. M., Lang, J. H., and Kim, S., "Design principles for energy-efficient legged locomotion and implementation on the MIT Cheetah robot," *IEEE/ASME Transactions on Mechatronics*, vol. 20, no. 3, pp. 1117–1129, 2015.

[5] Blackman, D. J., Nicholson, J. V., Pusey, J. L., Austin, M. P., Young, C., Brown, J. M., and Clark, J. E., "Leg design for running and jumping dynamics," *2017 IEEE International Conference on Robotics and Biomimetics, ROBIO 2017*, vol. 2018-Janua, pp. 2617–2623, 2018.

[6] Sugiyama, Y. and Hirai, S., "Crawling and jumping of deformable soft robot," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 4, no. c, pp. 3276–3281, 2004.

[7] Galloway, K. C., Clark, J. E., Yim, M., and Koditschek, D. E., "Experimental investigations into the role of passive variable compliant legs for dynamic robotic locomotion," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1243–1249, 2011.

[8] Hurst, J., "The Role and Implementation of Compliance in Legged Locomotion," *The International Journal of Robotics Research*, vol. 25, no. 4, p. 110, 2008.

[9] Pratt, G. A. and Williamson, M. M., "Series elastic actuators," *IEEE International Conference on Intelligent Robots and Systems*, vol. 1, pp. 399–406, 1995.

[10] Ahmadi, M. and Buehler, M., "Stable control of a simulated one-legged running robot with hip and leg compliance," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 1, pp. 96–104, 1997.

[11] Kani, M. H. H. and Ahmadabadi, M. N., "Comparing effects of rigid, flexible, and actuated series-elastic spines on bounding gait of quadruped robots," pp. 282–287, 2013.

[12] Horigome, A., Qudsi, Y., Fisher, E., and Vaughan, J., "Robot Jumping with Curved-Beam Flexible Legs Orange marker Blue marker Tether,"

[13] Luo, Z.-h., "Direct Strain Feedback Control of Flexible Robot Arms: New Theoretical and Experimental Results," vol. 38, no. 11, 1993.

[14] He, W., Gao, H., Zhou, C., Yang, C., and Li, Z., "Reinforcement Learning Control of a Flexible Two-Link Manipulator: An Experimental Investigation," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–11, 2020.

[15] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D., "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[16] Dwiel, Z., Candadai, M., and Phielipp, M., "On Training Flexible Robots using Deep Reinforcement Learning," *IEEE International Conference on Intelligent Robots and Systems*, pp. 4666–4671, 2019.

[17] Li, Q., Zhang, W. J., and Chen, L., "Design for control - A concurrent engineering approach for mechatronic systems design," *IEEE/ASME Transactions on Mechatronics*, vol. 6, no. 2, pp. 161–169, 2001.

[18] Chen, T., He, Z., and Ciocarlie, M., "Hardware as Policy: Mechanical and computational co-optimization using deep reinforcement learning," *arXiv*, no. CoRL, 2020.

[19] Schaff, C., Yunis, D., Chakrabarti, A., and Walter, M. R., "Jointly learning to construct and control agents using deep reinforcement learning," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 9798–9805, 2019.

[20] Whitman, J., Bhirangi, R., Travers, M., and Choset, H., "Modular Robot Design Synthesis with Deep Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 06, pp. 10418–10425, 2020.

[21] Zhao, W., Queralta, J. P., and Westerlund, T., "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: A Survey," *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*, pp. 737–744, 2020.

[22] Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., and Riedmiller, M., "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards," pp. 1–10, 2017.

[23] Plappert, M., Andrychowicz, M., Ray, A., McGrew, B., Baker, B., Powell, G., Schneider, J., Tobin, J., Chociej, M., Welinder, P., Kumar, V., and Zaremba, W., "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research," pp. 1–16, 2018.

[24] Fujimoto, S., Van Hoof, H., and Meger, D., "Addressing Function Approximation Error in Actor-Critic Methods," *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2587–2601, 2018.

[25] Sammut, C. and Webb, G. I., eds., *Bellman Equation*, p. 97. Boston, MA: Springer US, 2010.

[26] Mnih, V. and Silver, D., "Playing Atari with Deep Reinforcement Learning," pp. 1–9.

[27] Sutton, R. S. and Matheus, C. J., "Learning Polynomial Functions by Feature Construction," *Machine Learning Proceedings 1991*, pp. 208–212, 1991.

[28] Watkins, C., "Learning From Delayed Rewards," 1989.

[29] Dormann, A. R., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Noah, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, vol. 22, no. 22, pp. 1–8, 2021.

[30] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Kopf, Andreas and Yang, Edward and DeVito, Zacha, S., *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates, Inc., 2019.

[31] Ha, S., Xu, P., Tan, Z., Levine, S., and Tan, J., "Learning to walk in the real world with minimal human effort," *arXiv*, no. CoRL, pp. 1–11, 2020.

[32] Fankhauser, P., Hutter, M., Gehring, C., Bloesch, M., Hoepflinger, M. A., and Siegwart, R., "Reinforcement learning of single legged locomotion," *IEEE International Conference on Intelligent Robots and Systems*, pp. 188–193, 2013.

[33] Xiao, Q., Cao, Z., and Zhou, M., "Learning locomotion skills via model-based proximal meta-reinforcement learning," *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, vol. 2019-Octob, pp. 1545–1550, 2019.

[34] Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., and Levine, S., "Learning to Walk Via Deep Reinforcement Learning," 2019.

[35] Tsounis, V., Alge, M., Lee, J., Farshidian, F., and Hutter, M., "DeepGait: Planning and control of quadrupedal gaits using deep reinforcement learning," *arXiv*, no. 1, pp. 1–8, 2019.

[36] Da, X., Xie, Z., Hoeller, D., Boots, B., Anandkumar, A., Zhu, Y., Babich, B., and Garg, A., "Learning a Contact-Adaptive Controller for Robust, Efficient Legged Locomotion," vol. 1, no. c, 2020.

[37] Blickhan, R. and Full, R. J., "Similarity in multilegged locomotion: Bouncing like a monopode," *Journal of Comparative Physiology A*, vol. 173, no. 5, pp. 509–517, 1993.

[38] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W., "OpenAI Gym," pp. 1–4, 2016.

[39] Harper, M. Y., Nicholson, J. V., Collins, E. G., Pusey, J., and Clark, J. E., "Energy efficient navigation for running legged robots," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 6770–6776, 2019.

[40] Pace, J., Harper, M., Ordonez, C., Gupta, N., Sharma, A., and Collins, E. G., "Experimental verification of distance and energy optimal motion planning on a skid-steered platform," *Unmanned Systems Technology XIX*, vol. 10195, p. 1019506, 2017.

[41] Sorensen, K. L. and Singhose, W. E., "Command-induced vibration analysis using input shaping principles," *Autom.*, vol. 44, pp. 2392–2397, 2008.

[42] Singer, N. C. and Seering, W. P., "Preshaping Command Inputs to Reduce System Vibration," *Journal of Dynamic Systems, Measurement, and Control*, vol. 112, no. 1, pp. 76–82, 1990.

[43] Singhose, W., Seering, W., and Singer, N., "Residual Vibration Reduction Using Vector Diagrams to Generate Shaped Inputs," *Journal of Mechanical Design*, vol. 116, no. 2, pp. 654–659, 1994.

[44] Luck, K. S., Amor, H. B., and Calandra, R., "Data-efficient co-adaptation of morphology and behaviour with deep reinforcement learning," *arXiv*, no. CoRL, 2019.

[45] Ha, D., "Reinforcement learning for improving agent design," *Artificial Life*, vol. 25, no. 4, pp. 352–365, 2019.

[46] Wang, T., Zhou, Y., Fidler, S., and Ba, J., "Neural graph evolution: Towards efficient automatic robot design," *arXiv*, pp. 1–17, 2019.

[47] Hu, S., Yang, Z., and Mori, G., "Neural fidelity warping for efficient robot morphology design," *arXiv*, 2020.

Albright, Andrew. Bachelor of Science, North Carolina State University, Spring 2020;
Masters of Science, University of Louisiana at Lafayette, Spring 2022
Major: Mechanical Engineering
Title of Thesis: Reinforcement Learning Based Mechanical Design and Control of Flexible-Legged Jumping Robots
Thesis Director: Dr. Joshua E. Vaughan
Pages in Thesis: 125; Words in Abstract: 185

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus nec tellus eget elit aliquet accumsan sit amet in lacus. In hac habitasse platea dictumst. Ut sit amet elit odio. Aenean lobortis mollis metus, sed consequat neque tristique in. Curabitur nec hendrerit metus. Praesent non scelerisque urna, vitae iaculis diam. Aliquam nisl est, imperdiet eu nulla sed, bibendum pulvinar arcu. In ultricies purus purus, vulputate congue justo volutpat ut. Donec nunc magna, rutrum nec turpis et, viverra efficitur lorem. In hac habitasse platea dictumst. Vestibulum maximus lobortis nisl, eget molestie sem sollicitudin nec. Mauris ut enim eu ipsum auctor rhoncus ac vel eros.

Vivamus tincidunt, tortor eu rutrum dapibus, orci turpis porta metus, ac iaculis quam eros sollicitudin nisl. Nam id massa elementum, commodo mi at, lobortis nisl. Fusce vestibulum eu lorem non aliquam. Morbi eleifend tortor id metus elementum, ac tincidunt lorem commodo. Pellentesque vestibulum, erat in tempus vehicula, ex urna auctor leo, ut lobortis eros mauris nec erat. Aliquam erat volutpat. Sed sed pretium risus.

Biographical Sketch

Forrest Montgomery was born in Lafayette, Louisiana for all intents and purposes. He began his academic career at the University of Louisiana with an internal struggle between majoring in Mechanical Engineering or Industrial Design. This thesis is evident of the choice he made. After earning his Bachelor's degree at the University of Louisiana at Lafayette in the Spring of 2015, he joined the CRAWLAB and conducted research in dynamics, controls, and robotics under the tutelage of Dr. Joshua Vaughan. This research culminated with earning a Master's degree in Mechanical Engineering again at the University of Louisiana at Lafayette in the Summer of 2017.