V Concurrent Design of the Monopode System

The workflow for creating a mechanical system and associated control strategy is often sequential. Typically, the mechanical and electrical hardware are developed, creating a set of confinements for the controller to be designed from. However, the mechanical/electrical system description is not always a simple one, and generating a controller for it might be unnecessarily challenging. It is of interest to allow the mechanical/electrical parameters of the system, and therefore the system's description, to be changeable. This would allow for optimization of the complete system rather than optimizing the mechanical/electrical elements and control in isolation. Designing the system and control input in unison has been researched and is often 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 concurrent design architecture is proposed 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, the proposed algorithm uses two instances of the TD3 algorithm, creating an inner and an outer loop. The first instance, which is responsible for learning the control policy, will be instantiated in a similar fashion to the what was seen in Chapter 2. It is the outer loop of the concurrent design process. The second instance, which is responsible for learning the mechanical

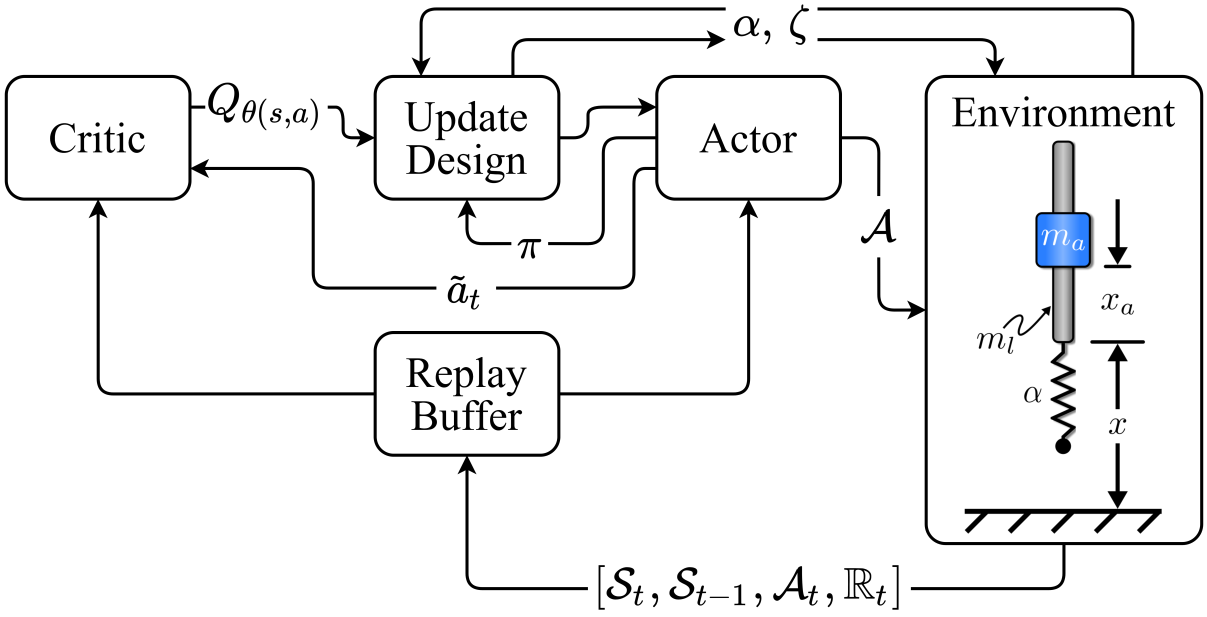


Figure 26. Concurrent Design Architecture

design, is instantiated ~~within the outer~~ in the inner loop and in a similar fashion to what was 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 pre-defined control input, like what was shown in Chapter 4, the simulation used the input that was 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 was 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 were shown, and it was 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 design found was used to update the environment within the outer loop. This methodology was applied to both the discrete and the continuous methods of the 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, which is the rate at which the design is updated. As is shown in Appendix 7, the design is updated in line with the control policy but 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 δ policy updates where δ 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, δ 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 spaces, both with differing nominal values as well as 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 the number of iterations required 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 applied to the environment and observation saved to the replay buffer 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. The observation space was defined as:

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. 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 and saved the timeseries

simulation information in the replay buffer. The actions applied, $\mathcal{A}$, within the environment were the designs selected and were defined as follows:

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

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. The transitions saved, were the timeseries information, and were defined as:

$$\mathcal{S} = \{\mathbf{X}, \dot{\mathbf{X}}, \mathbf{X}_a, \dot{\mathbf{X}}_a\} \tag{23}$$

where $\mathbf{X}$, $\dot{\mathbf{X}}$, $\mathbf{X}_a$ and $\dot{\mathbf{X}}_a$ are vectors of time evolution for the rod's position and velocity and the actuator's positions and velocity, respectively.

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.

The outer loop was instantiated in a similar fashion as to what was discussed in Chapter 2, except that the number of 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 environment's parameters would be changing due to the inner loop. The training hyperparameters used for the outer loop instantiation of the TD3 algorithm are presented in Table 5.

The inner loop, was instantiated in a similar fashion to what is shown in Chapter 4. This instantiation of the TD3 algorithm was created within the outer

Table 5. Outer Loop TD3 Training Hyperparameters

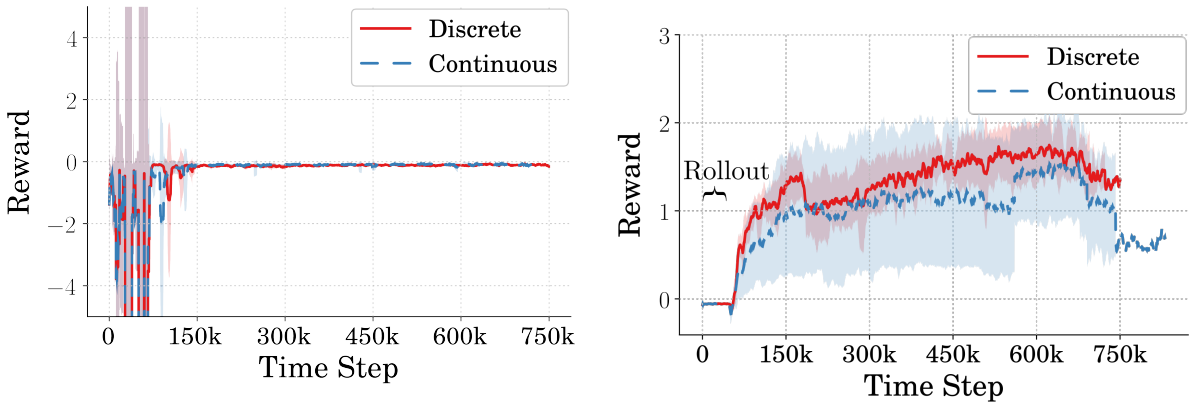
Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	1000 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip,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

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

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



(a) Reward During Training for Efficient Design

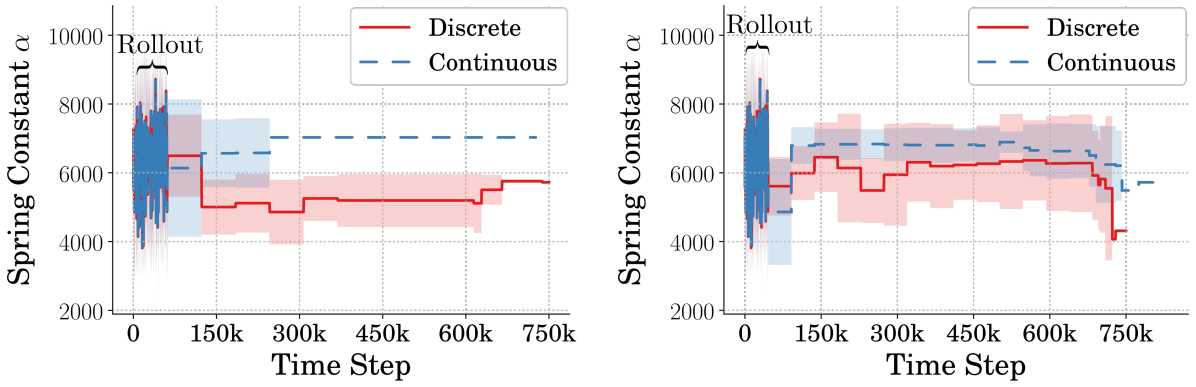
(b) Reward During Training for Height Design

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

5.5.1 Reward vs Learning Step. It is firstly of 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 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 losing rewards due to over fitting towards the end of training.

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



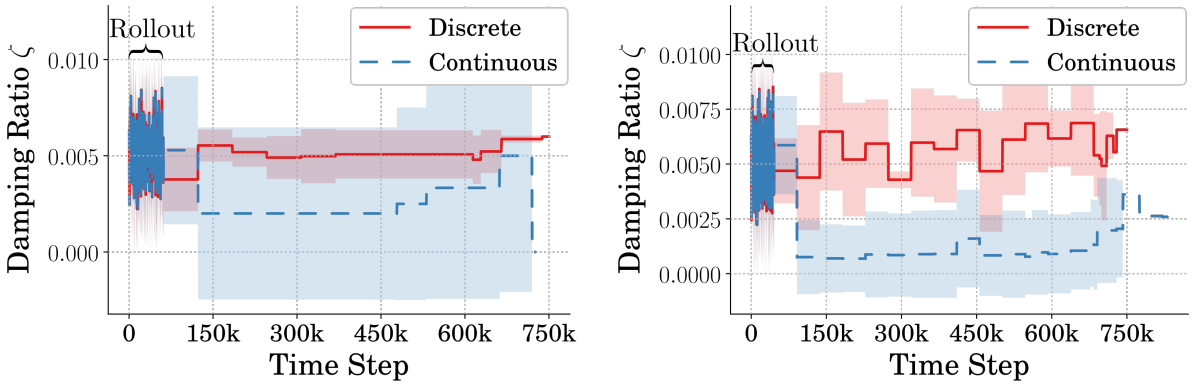
(a) Spring Constant Learned During Training for Efficient Design

(b) Spring Constant Learned During Training for Height Design

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

methods. Figure 28 shows the learning of the spring constant for the efficient and high jumping concurrent design types. It is apparent that the learning of the designs for both jumping cases is more continuous for the continuous design update method, where the policy for learning a design is kept constant. In both jumping cases, the continuous method learns higher spring rates on average across different instantiations of the concurrent design architecture. This is likely because higher spring rates create higher jumps for poorly trained policies which are learned early in training. For the continuous update method, the designs learned have a greater effect throughout the entirety of the learning process. Furthermore, there are obvious differences between the methods regarding standard deviation across outer loop instantiations, where the continuous method has much less deviation. This does not directly translate to performance consistency, 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 converges to a more consistent design.

Figure 29 shows the learning of the damping ratio for the efficient and high jumping concurrent design types. In both cases, the continuous method learned a lower damping ratio than the discrete method. Similar to the learning of the spring rate, the



(a) Damping Ratio Learned During Training for Efficient Design

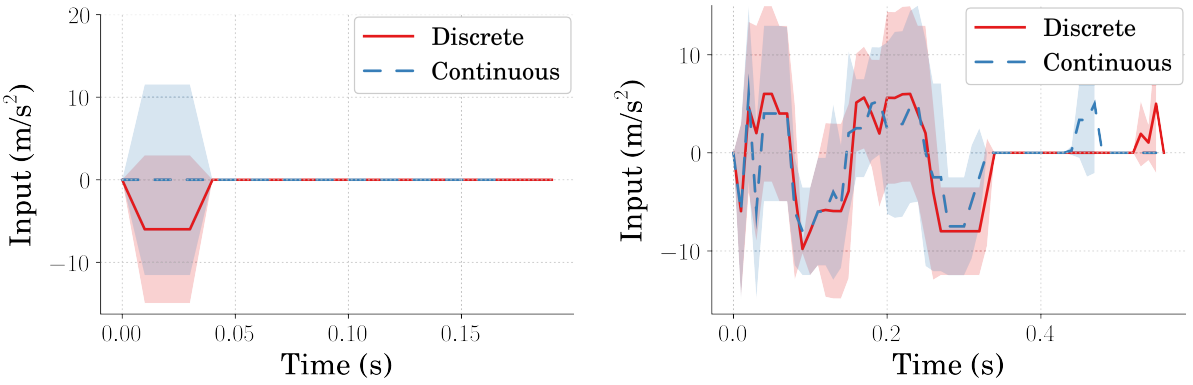
(b) Damping Ratio Learned During Training for Height Design

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

continuous method learns a design with less discrete value changes throughout the control policy training. This is particularly obvious in the case where the policy is learning a high jumping strategy. For the efficient jumping strategy, the damping ratios found across concurrent design instantiations vary greatly, as is shown by the large standard deviation among the learned ratios. Whereas, in the case of the high jumping strategy, the difference in standard deviation is lower. The increase in standard deviation of learned damping ratios 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. Comparing the learning processes of 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 necessary to evaluate the differences seen in performance between the two methods discussed.

Starting with the learned inputs for the efficient and high jumping designs,



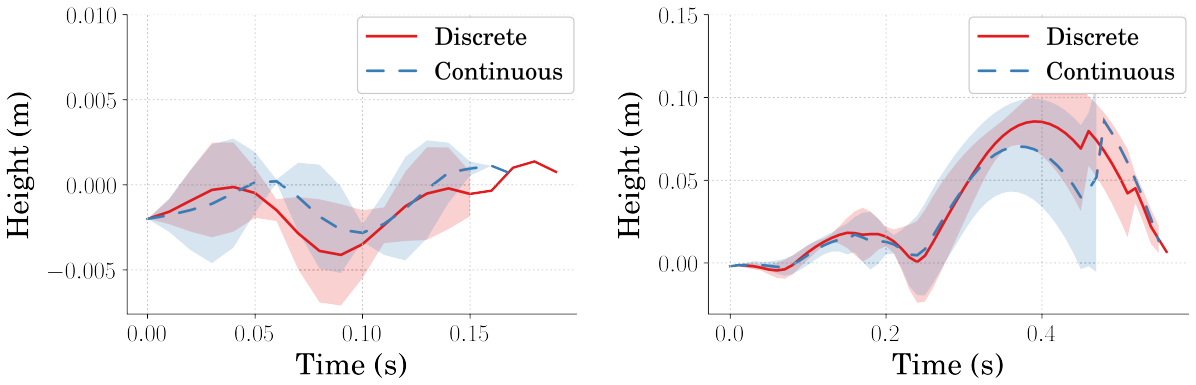
(a) Average Input for Efficient Concurrent Designs

(b) Average Input for High Jumping Concurrent Designs

Figure 30. Average Controller Performance for Efficient Concurrent Designs

shown in Figure 30, it is apparent that the learned input for the efficient jumping type 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. This learned input is not consistent with what was shown to be learned in Chapter 2, and is likely the result of increasing the complexity of the learning process where already there existed a fragile reward function.

The inputs for the high jumping designs are shown in Figure 30b. For this jump type, the timing and magnitude of the inputs of the discrete and continuous methods are very similar throughout most of the command. However, as will be discussed later, the differences do create differing jumping performances. The largest difference between the methods, regarding the inputs learned, are towards the end of the command, where both methods result in a control policy that accelerates the actuator upwards. This resulted from the definition of the reward where the policy was trying to maximize the height at every time step. The differences in the timing seen in the final command are not entirely clear regarding how the discrete and continuous methods effected the



(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

learned command.

The average timeseries jumping performances of the efficient and high jumping strategies for the discrete and continuous methods can be seen in Figure 31. Figure 31a shows the jumping performance of the efficient concurrent design. It is apparent 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. A potential solution to this poorly learned policy could be to modify the reward such that it returned a normalized score to better equip the policy with usable information.

The jumping performance of the high jumping concurrent designs are shown in Figure 31b. The learned inputs for the discrete and continuous methods, shown in Figure 30b, being similar in timing and magnitude, create similar jumping performances. The continuous method, having an input comprised of noisy commands does not store as much energy within the spring and therefore does not jump the monopode as high. Rather than storing increased energy within the spring, it instead relies on the final upward acceleration of the actuator to get to its maximum height.

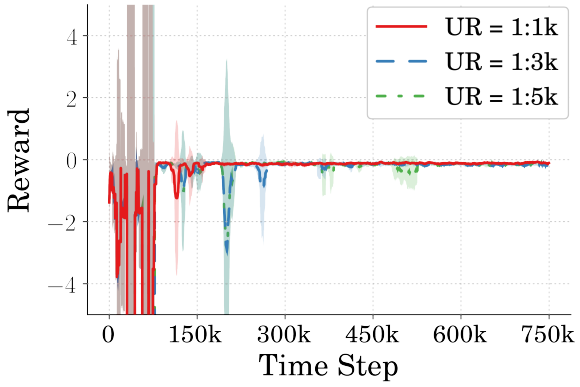
The discrete method, though employing the same final upward acceleration, does so later than the continuous method and therefore does not capitalize on it for achieving it's maximum height. Regarding the consistency across instantiations of the concurrent design architecture, the discrete methods learn 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

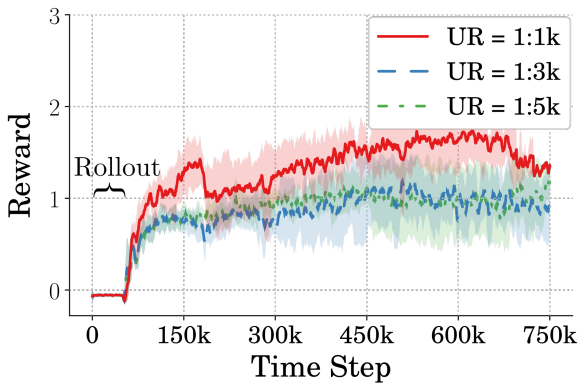
In addition to discovering the differences in learning and final design performance due to employing the discrete/continuous method, it is also of interest to discover the difference when changing the design update rate. The update rate value is tied to the rate that the control policy is updated in 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 design that successfully completed a stutter jumping command. It was also shown that the discrete method produced designs that more consistently resulted in high jumping performance 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 were 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 when lowering the update rate, the design changes less often



(a) Reward During Training for Efficient Design



(b) Reward During Training for Height Design

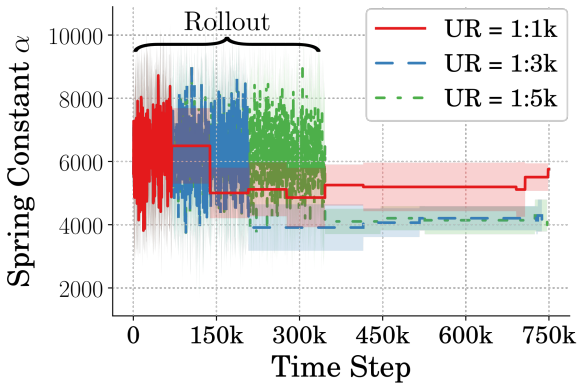
Figure 32. Reward During Training for Discrete and Continuous Implementation Methods of Concurrent Design

and more drastically.

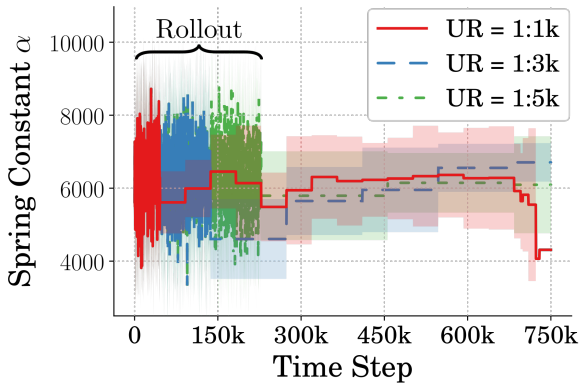
As for the high jumping strategies, seen in Figure 32b, there are more obvious differences. It is apparent that lowering the update rate decreases policy performance with respect to the reward received. Additionally, in the beginning of training, there is less variance in the performance of the policies across network instantiations. This is directly related to how often the mechanical design is updated and will be further explained when evaluating the mechanical designs learned. Finally, it is shown 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. 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 performant policies.

The learning of the damping ratios for the three different update rates being

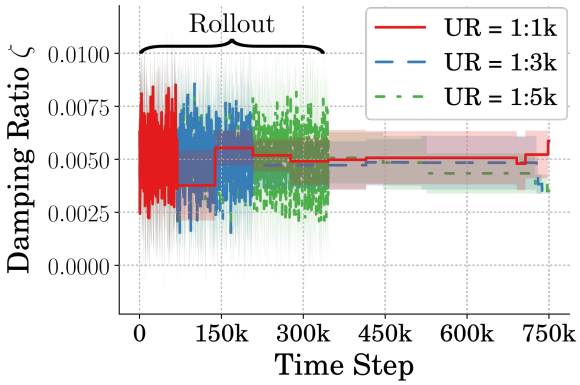


(a) Spring Constant Learned During Training for Efficient Design

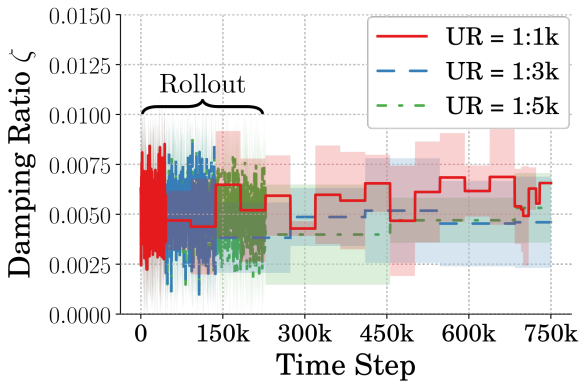


(b) Spring Constant Learned During Training for Height Design

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



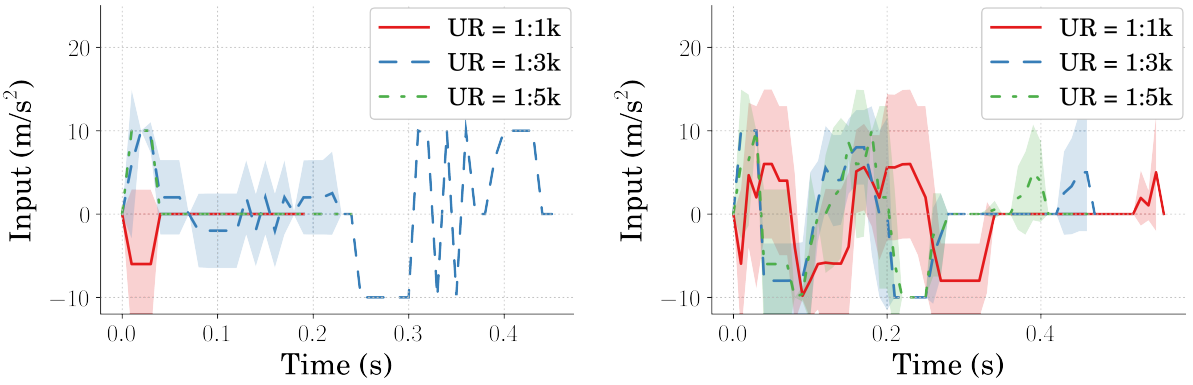
(a) Damping Ratio Learned During Training for Efficient Design



(b) Damping Ratio Learned During Training for Height Design

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

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. As for the high jumping strategies, shown in Figure 34b, the differences are more pronounced. 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.



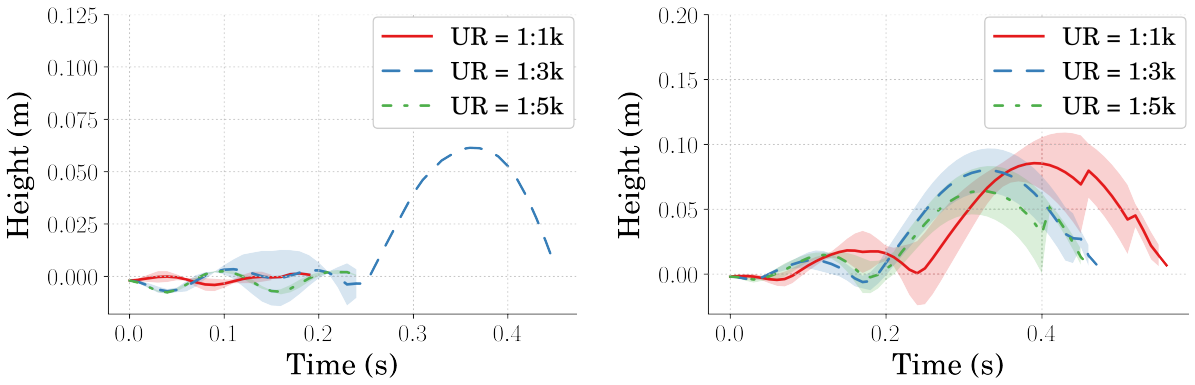
(a) Average Input for Efficient Concurrent Designs

(b) Average Input for High Jumping Concurrent Designs

Figure 35. Average Controller Performance for Efficient Concurrent Designs

5.6.3 Resulting Input and Jumping Performance. Average learned input for the efficient and high jumping concurrent designs for the three different update rates evaluated are shown in Figure 35. There are clear differences regarding the policies learned across the different design update rates. For the efficient jumping strategy, shown in Figure 35a, reducing the rate that the design is updated allows the policies to learn a strategy beyond a single bang command. The lower update rates also caused the policies to learn an input that accelerated the actuator in the opposite direction for the initial acceleration. Additionally, for the middle update rate evaluated, where the design was updated every 3000 policy updates, the policy for one of the 5 instantiations learned a command that performed what would be considered a stutter jump. This inconsistency might be solved by either generating a more stable reward condition, or by better tuning the learning hyperparameters within the inner/outer loops. Regardless, the differences in input shapes show that changes in update rate can have an impact on learned policies.

Figure 35b shows the inputs learned for the high jumping concurrent designs. Similar to the inputs learned for the efficient concurrent designs, there are differences that can be seen both in timing and magnitude between the different update rates



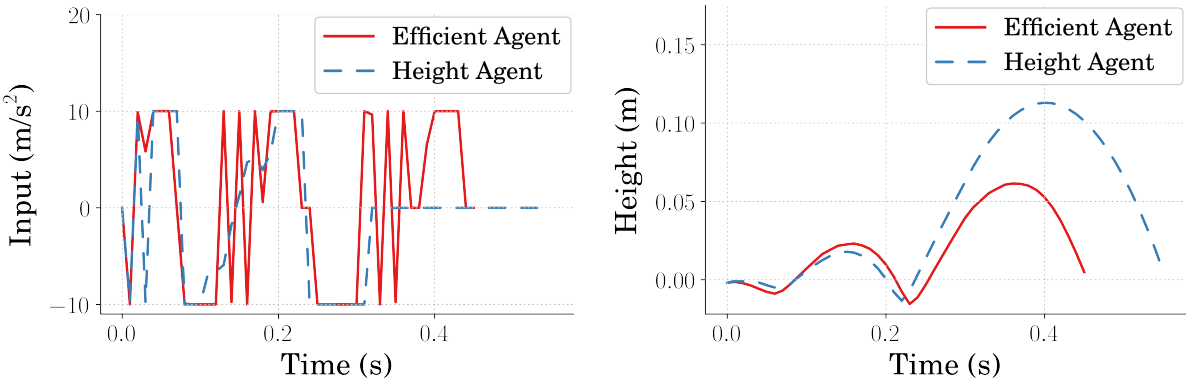
(a) Average Jumping Performance for Efficient Concurrent Designs

(b) Average Jumping Performance for High Jumping Concurrent Designs

Figure 36. Average Controller Performance for High Jumping Concurrent Designs

evaluated. In both the 1:3k and the 1:5k update rates, the average commands learned have higher magnitude accelerations in the positive direction and in the last negative direction. Additionally, these update rates produce commands that, like the lower update rates for the efficient concurrent designs, accelerate in the positive direction for the initial acceleration command. Finally, the commands with lower update rates learn more consistent commands across different instantiations of the concurrent design architecture.

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 36. Figure 36a, showing the resulting jumping performance from the efficient command input, shows the effects of the differing inputs. Firstly, looking at the performance of the 1:3k update rate, it is clear that changes to the update rate can result in a policy that learned a design to accomplish a stutter jump. However, as was pointed out in the input discussion, this behavior is highly inconsistent and exists within a set of commands that are already inconsistent in general. Regardless, the effects of the different input directions can be seen in that the lower update rate learned concurrent designs do learn to initially compress the spring to create the energy to jump. It appears, however, that the rewards punish power in a way that forces the



(a) Best Case Efficient and High Jumping Inputs Learned

(b) Best Case Efficient and High Jumping Heights

Figure 37. Best Case Performance for Concurrent Designs

commands not to learn to fully complete a stutter jump, as they rely too much on the spring energy alone to complete the jumping commands.

Figure 36b, shows the average jumping performance of the high jumping concurrent designs, and like the efficient jumping performance, represent the performance of the inputs discussed previously. It is apparent that, in the case of the high jumping strategies, the changes in the update rate have a direct correlation to jumping performance. That is, lower update rates ultimately result in lower average final jumping performance. It appears as well, that the update rate directly effects the consistency in learning where the 1:3k update rate, although not having learned the best average performance, did learn a more consistent performing design across different instantiations of the concurrent design architecture. For the 1:1k update rate, the concurrent designs learned, on average, outperformed the controller trained on a static environment which was shown in Chapter 2.

5.7 Best Case Performance

Figure 37 shows the best concurrent design performance for the efficient and high jumping strategies. The performance of the efficient strategy comes from a discrete mechanical design update method. Additionally, the design update rate that resulted in

a properly learned stutter jump was the 1:3k update rate. For the high jumping strategy, the design update method that produced the highest performing design was the continuous method. The update rate that produced the highest jumping strategy was the 1:5k update rate. Comparing the results found using the concurrent design method to those seen in Chapter 2, where a policy was trained on a static environment, the high jumping strategies see an increase in jump height of 18.96% and the efficient strategies see a decrease in power efficiency of 51.23%.

5.8 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. In general, the continuous method showed more consistent learning across multiple instantiations of the concurrent design architecture, making it the more attractive option for the monopode system. 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 to the increased complexity of the control policy. As for the high jumping strategy, the designs found averaged higher performance than control policies trained on static mechanical designs like what shown in Chapter 2. Additionally, in the best

case, the high jumping concurrent design also outperformed the best case of a policy trained on a static environment. It should be concluded then, 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

6.1 Conclusion

The work presented in this thesis presented and evaluated the performance of a concurrent design architecture that utilized reinforcement learning techniques, for flexible jumping systems. The RL algorithm used throughout the contributions within the document was the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm. The architecture presented was one that concurrently learned a mechanical design and an associated control policy. A monopode jumping system was described and used in simulation throughout the work to evaluate the steps leading to and the final performance of the concurrent design architecture. The proposed method was divided into multiple sections presented across several chapters, wherein, additional discoveries were presented.

In Chapter 2, a RL-based controller was trained on the monopode jumping system to evaluate the effectiveness of training for power efficient control. Two jump types were evaluated to determine if learning efficient control could scale to more complex inputs. This work was completed, in part, to evaluate if rewarding an RL algorithm in a manner solely focusing on efficiency would result in more efficient jumping strategies that learned to utilize flexible components within a system. Additionally, the work presented in Chapter 2 was completed to build the controller component for a flexible jumping system, which is half of what is required for the proposed concurrent design architecture. It was shown that when training for efficiency, the learned commands for the monopode are drastically changed, particularly in the optimal case. The commands learned did utilize the spring within the system to store energy to use efficiently. Regarding the differences in efficiency when scaling the complexity of the jump, the TD3 algorithm, given more opportunity, was better able to optimize power consumption for more complex jump types. Additionally, in the case of training to jump to maximum height, the commands learned were shown to be ones

which aligned with optimal inputs for both jump types. This was shown in Chapter 3, where input shaping techniques were used to evaluate the learned polices. It was concluded that RL is a useful method for defining control strategies for flexible-legged jumping systems, particularly when energy efficiency is of interest.

In Chapter 4, a method for utilizing the TD3 RL algorithm to define mechanical designs was introduced. Changes to the traditional ways in which an RL environment is utilized were discussed and the implementation of the algorithm was shown. The monopode jumping system, combined with a control strategy defined using the input shaping techniques discussed in Chapter 3, was used to generate a simulation. The simulation was deployed such that an instantiation of the TD3 algorithm was used to learn mechanical designs related to system flexibility. Two rewards were defined to evaluate if the algorithm could find multiple designs to accomplish multiple tasks given a single input. The first reward learned a design that maximized jump height and the second learned a design that forced the monopode to jump to a specified height. Additionally, two design spaces were created to evaluate the algorithms performance in finding a design within different design space configurations. The two design spaces had differing nominal values and differing space limits. It was shown that different designs could be learned within different design space constraints to accomplish multiple tasks utilizing the same input. This work showed that the TD3 RL algorithm could be utilized as a method for optimizing design parameters and therefore could be used as the second half to the concurrent design architecture presented.

Finally, in Chapter 5, the methods of learning a control policy and a design were combined to create the concurrent design architecture proposed. The discrete and continuous methods of implementing the mechanical design update were shown, and the effects of each method was evaluated. Additionally, a new hyperparameter, being the design update rate, was evaluated and the results were shown. The methods were tested to evaluate efficient jumping and non-efficient jumping concurrent designs for the

monopode jumping system. The discrete method proved to learn more consistent designs across different instantiations of the concurrent design architecture. Additionally, the final performance of the learned designs were better on average for the discrete method. These results suggest that the discrete method is the more attractive option for the monopode jumping system. Changes to the update rate also affected the learning process and the performance of the learned designs. It was shown that this hyperparameter was highly dependent on the complexity of the command being learned, where more complex commands showed difficulty learning with higher design update rates. In general, the concurrent design architecture struggled to find good concurrent designs for the efficient strategy due to the increased complexity of the reward, along with the natural fragility of the reward.

The performance of the high jumping strategies concurrent designs were compared to the systems where a controller was trained on a static environment. The concurrent designs, on average and in the best case for both architectures, outperformed the controller trained on static environments. It can be concluded then, 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.

6.2 Future Research

The work presented in this thesis explores the use of reinforcement learning for building concurrent design architectures. This type of architecture is of interest for developing systems that are optimized regarding both the mechanical structure and the control policy. It has been shown that RL can be used as a base for building the parts of the concurrent design architecture for the simple monopode jumping system. The immediate steps should be scaling the architecture to a more complex system. Additionally, different RL algorithms such as SAC, PPO, and TRPO should be

evaluated wherein architectures that can learn discrete mechanical designs should be studied.

Different network-based learning methods, such as evolutionary programming, should be applied for learning a mechanical design for flexible systems. These have shown promise in the studies of concurrent design, but have not been applied to flexible systems. Finally, scaling the simulation to return information such as stress within the links would assist in performing more complex mechanical design updates. Ultimately, a concurrent design system should be one where the architecture can learn to shape objects and select components, effecting all manner of mechanical parameters (mass, flexibility, damping, motor parameters) while also learning a control policy suited for those parameters.

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 method being that for 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.