Deep RL for Serial Stewart Platform Control

Jakob Frabosilio, E.I.T, M.S. Mechanical Engineering
Computer Science Department, Cal Poly SLO
jfrabosi@calpoly.edu

Abstract— **A Serial Stewart platform (SSP) is a complex robotic actuator made of stacked Stewart platforms or hexapods. This paper proposes a simplified version of an SSP representing each platform using a position and an orientation vector. This SSP is then trained in a custom Gymnasium environment using deep reinforcement learning through proximal policy optimization in Stable Baselines3. The simulation environment has flaws in modeling the constraints / range of motion of each individual Stewart platform which causes the reinforcement learning agent to fail to converge on a solution. Future work is proposed on how to improve the representation of an SSP for deep reinforcement learning.**

Keywords— **deep reinforcement learning, proximal policy optimization, Stewart platform, neural networks, control systems**

I. Introduction

Artificial intelligence has become more and more prevalent throughout my college career; from large language models capable of holding realistic conversations to autonomous taxis transporting humans across cities in the US, the world is a different place from when I began my studies in 2018. Though my degrees are in the field of mechanical engineering, I have been taking numerous electrical engineering, probability / statistics, and computer science courses to adapt to this changing world. As part of CSC 580 – Intelligent Agents, I have immersed myself in the field of deep reinforcement learning. I foresee a future where robots and complex mechatronic systems are controlled by reinforcement learning algorithms, and I wish to be present in that future.

In this paper, I present my work on modeling stacked Stewart platforms in a virtual environment and controlling the position of each platform using a deep neural network policy trained with proximal policy optimization (PPO). In Section II, I delve into the basics of deep reinforcement learning and the particular algorithm that I chose to train the system with. In Section III, I explain what a Stewart platform is, and the theory of stacking platforms to create one long, complex actuator. In Section IV, I detail the creation of the custom simulation environment and the existing libraries that I took advantage of. In Section V, I visualize the neural network architecture controlling the agent. In Section VI, I process the results of deep RL training on the agent.

II. Deep Reinforcement Learning (Deep RL)

Reinforcement learning is the process of training an agent to perform actions that maximize reward from an environment. An agent acts according to a policy, which takes an observation of the current state of the environment and produces an action [1].

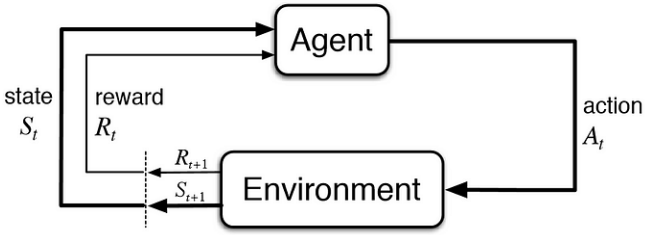

Fig. 1 Basic reinforcement learning architecture, from [1]

Variations on how to train this policy (and how the policy is represented) are one place where RL algorithms diverge. Q-learning uses an evaluation function on state-action pairs (in basic terms, taking a certain action given that you're in a certain state) to choose an optimal policy. It selects the action that maximizes Q, a weighted estimate of reward for immediately following that policy plus the estimated reward for following that policy for the rest of the simulation. A Q-table can be built to store results of iterative trials, i.e. a record of Q-values from testing state-action pairs and policies. [2]. In Q-learning, we do not build a model of the environment (as we do in some other RL algorithms), but we merely follow a policy that maximizes our reward. Q-learning and other algorithms suffer from the "curse of dimensionality," where the state space is so large that it is impractical to calculate and store these Q-values explicitly, as the Q-table gets increasingly large [2].

This is where the "deep" part comes in – deep neural networks can be used in place of deterministic Q-learning tabulation. These neural networks

generate an abstraction of the state, and this abstraction is used to generate an action; we can use these neural networks in place of querying a Q-table to choose an action given the current state. Though the process of abstraction results in information loss, these deep Q-networks (DQN) have been used in multiple cases to solve complex RL problems. In 2015, a team of DeepMind engineers revolutionized the field of RL by using a DQN to train a single agent to play nearly 50 Atari games. The agent takes in the pixel screen as an input (along with some preprocessing) and returns an action to perform in the game. The agent's policy is stored within the neural network weights and biases, and the policy can be trained using backpropagation and Q-learning. Given a particular action in a particular state, the agent receives a reward, and the process of Q-learning is used to modify the agent's neural network parameters to produce the largest possible reward [3].

A. Proximal Policy Optimization (PPO)

Though DQN may have been state-of-the-art in 2015, many new algorithms have been developed and honed for use in deep RL. At the time of writing, proximal policy optimization (PPO) is a state-of-the-art algorithm that works for both discrete and continuous action spaces, is relatively simple to implement, and uses a model-free architecture like DQN [4]. Given that I am still a novice when it comes to reinforcement learning, I chose to use PPO as the deep RL training algorithm.

Comparing PPO to DQN, PPO is an on-policy algorithm, meaning that it trains in real-time (as opposed to DQN, which can be trained on past data) [5]. PPO uses policy optimization to train its agent, whereas DQN uses Q-learning as described previously. Policy optimization can work in continuous or discrete action spaces (such as controlling the lengths of many linear actuators), while Q-learning is limited to discrete action spaces (such as pressing up, down, left, or right on a keyboard) [6]. Policy optimization attempts to modify the policy to maximize the estimated reward in a given state, while Q-learning uses a table (or neural network) to lookup (or generate) Q-values from a given state and chooses the action that produces the highest (estimated) Q-value.

The algorithm for PPO-Clip, the most common variant of PPO, can be seen below. In basic terms, PPO-Clip functions as such:

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Fig. 2 PPO-Clip algorithm, from [4]

First, the algorithm runs multiple simulations following an arbitrary/random starting policy. It then saves the rewards for each of those simulations, giving higher weight to rewards that were received in the beginning of each simulation (discounting the weight of long-term rewards, like humans do). Then, the advantage of each simulation is calculated. The advantage of a policy is the projected value of acting according to that policy, versus following some random policy (mathematically, it is equal to the Q-value of following that policy given a particular state minus the expected value of being in that state and following a random policy). Next, the policy is updated by maximizing the objective. With a very big oversimplification, this step modifies the policy by comparing its current actions to its previous actions (and the advantage that the policy had in those states) and adjusting the policy towards policies that had a large advantage. The "clip" part comes in by restricting the policy adjustment by a parameter ϵ, preventing the policy from making too large of a change, even if a particular policy had a very large advantage. This results in our policy taking small steps in the direction that maximizes our advantage by modifying the weights of our actor neural network. In the following step, the critic neural network is updated to produce the best possible value approximation for a state [6].

Small aside: PPO uses two neural networks, an actor network and a critic network. The actor network is the one that takes in an observation of a state and returns an action, while the critic network

takes in an observation and the most recent reward to determine how good of an action the actor network made in the previous timestep. The graphic below shows a great explanation of this process.

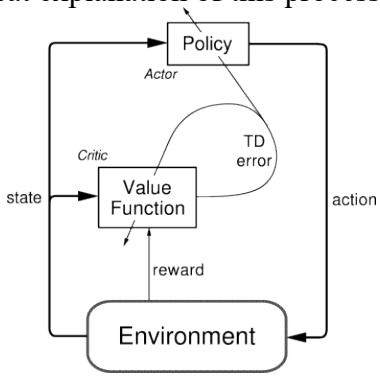

Fig. 3 Actor-critic framework, from [7]

III. Stewart Platforms

Now that we know the basics of deep RL and the algorithm we'll be using, let's discuss the actuator that we'll be controlling. A Stewart platform, also known as a Gough-Stewart platform or a "hexapod", is a 6DOF system that uses six parallel linear actuators connected by joints to two planes [8]. By controlling the length of the six actuators, the top plane of the Stewart platform can be positioned relative to the bottom plane in six degrees of freedom (translation and rotation in 3D). A model of a Stewart platform can be seen below.

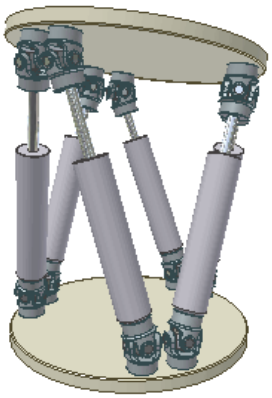

Fig. 4 Stewart platform animation, from [8]

The platform was designed by V. E. Gough in 1954 and publicized in a 1965 paper by D. Stewart. Many variants of the platform exist, with variations in top and bottom platform geometry, joint usage and positioning, and linear actuator type. The platforms are known for their reliability, well-established inverse kinematics, and their fine position control and stability. Stewart platforms can be found in flight simulators, animatronics, and surgical robots [8].

A. Serial Stewart Platforms (SSP)

Taking multiple Stewart platforms and attaching them end-to-end results in a complex linear actuator henceforth known as a serial Stewart platform (SSP). SSPs are capable of a far greater range of motion than single Stewart platforms, but can suffer from the increased complexity of controlling them and the load on subsequent platforms increasing as more platforms are added. SSPs are being explored for use in space robotics, where the compounding loading conditions on platforms near the base are mitigated due to the lack of gravity in space [9].

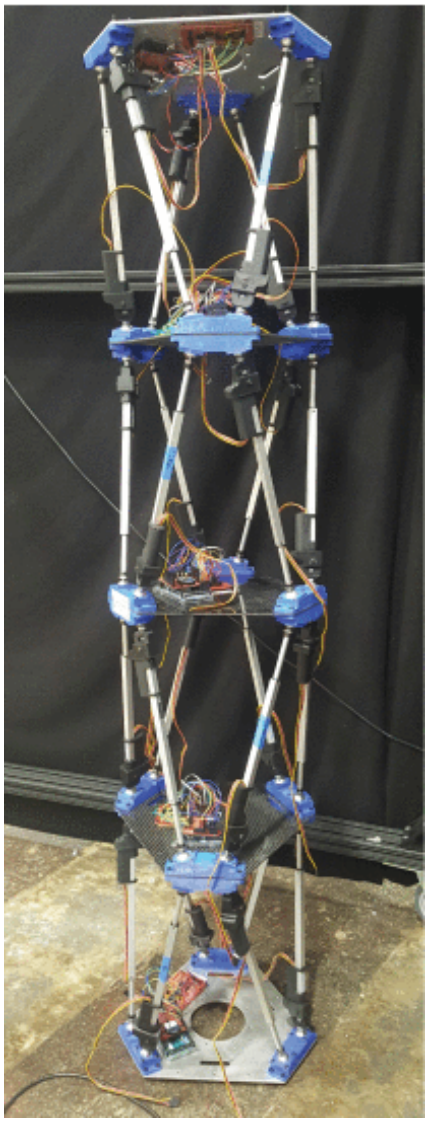

Fig. 5 Serial Stewart platform, from [9]

Some work has been done in controlling Stewart platforms using reinforcement learning, but this work has only considered single Stewart platforms, not SSPs. The inverse kinematics of the Stewart platform, while solvable and well-known, can be computationally intense and tend to be over-simplified. Yadarari et. al. presented a reinforcement-learning-based approach for control of a single Stewart platform earlier this year [10]. In this paper, I propose and explore a very introductory framework for modeling and controlling SSPs using deep reinforcement learning.

IV. Simulation Environment

In my model, I will be representing SSPs using a series of planes and vectors. The plane of the bottom platform is fixed to the ground (lying on the z-plane). The first platform is represented by a position vector that specifies the distance between the center of bottom plane and the center of the first actuating platform, as well as an orientation vector that specifies the orientation of the first actuator platform plane. Referring to Figure 4, the first position vector would point from the center of the bottom platform to the center of the top platform; the first orientation vector would extend from the center of the top platform and would be normal to the orientation of the top platform.

I have chosen Python for this research because of my existing knowledge of the language and the number of libraries available to simplify the deep RL implementation. I am using Stable Baselines3 [11] to train the agent due to its ease-of-use, proven results, and broad community support. Stable Baselines3 uses Gymnasium [12] environments for simulation, which is well-documented and easy to create custom environments for deep RL implementations.

A. Custom Environment Modeling

I created a custom environment called SerialStewartPlatform to simulate the SSPs. All documentation and code for this environment (and implementation of deep RL training) can be found in the GitHub repository in [13]. In this section, I will describe the __init__(), reset(), step(), and render() methods of this custom environment class.

1) *__init__()*: This method initializes the environment. The user specifies the rendering mode (choices are none, which doesn't render anything for training purposes, and human, which renders the visualization seen later in this paper), the number of Stewart platforms to stack, and the physical size and limitations of each Stewart platform. I have chosen to simplify the modeling of each platform by assigning each platform a default length (the "neutral" length of a platform, i.e. the height of a platform if all six actuators are extended halfway) and a maximum allowed extension and maximum allowed angular deviation. By default, I am using a default length of 0.6m, max extension of 0.2m, and max angle of 20 degrees.

2) *reset():* This method resets the environment for simulation and returns an initial observation. When this method is called, the vectors representing the SSP are initialized with their default values. In this initial configuration, all platforms have their position vector set to the default length of the platform, pointing straight up; the orientation vectors are all initialized pointing up. This configuration is like that of Figure 5. Then, a random position and orientation vector are generated; these vectors are "goals" and correspond to the desired position and orientation of the top platform. The reinforcement learning algorithm should train the SSP agent to reach a desired position and orientation for the top platform given an (ideally arbitrary) starting position. We return an observation to the SSP agent which includes all of the position and orientation vectors, as well as the desired position and orientation vector. This observation describes all the necessary information from our state that the agent will use to train.

3) *step():* This method simulates motion of the SSP given an input action, and returns an observation and a reward. First, the input action (which is a N by 3 by 6 transformation array, where each platform has its position vector multiplied by a 3x3 matrix for rotation and scaling and has its orientation vector multiplied by a 3x3 matrix for rotation) is applied to each of the position and orientation vectors of the SSP. If either of the transformations would cause a single Stewart platform to exceed the bounds specified in __init__(), then the new transformation for that platform is discarded and a large penalty is applied to the reward. Otherwise, the new position and orientation vectors are saved. The (x, y, z) coordinates of each platform's center are calculated by summing the subsequent position vectors for each platform. Then, the distance from the desired position and orientation (the "goals" from reset()) are calculated and added to the reward as a penalty, with the penalty being lower for how close the SSP is to achieving the goals. If the SSP reached the goals in this step, it gets a large reward and a new desired goal is generated. Once the SSP reaches its first goal, its position does not reset; this is how we achieve the "arbitrary starting position" described in reset(). We then return the observation and the reward for the current step. Note: within this method, we frequently have to find the angle between two vectors; the calculation to find this angle is from [14].

4) *render():* This method renders the SSP in a Matplotlib [15] window if render_mode is set to "human" when simulating the environment. Each Stewart platform is rendered as a 2D circle in a 3D plot, with a vector pointing from the center of each platform to the next. The randomly generated "desired

position and orientation" is rendered as a 2D circle with no vectors connecting it to the main SSP structure. Additionally, the current reward for the simulation is displayed above the graph. A sample render can be seen on the following page.

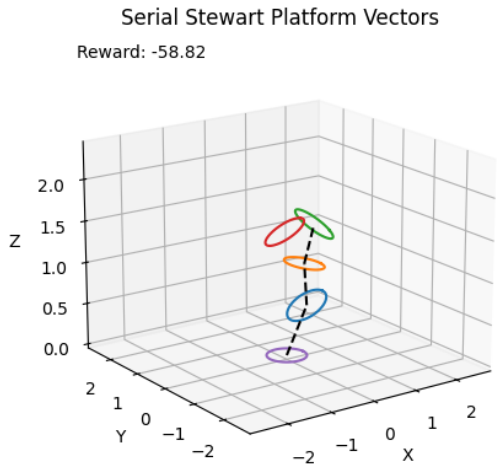

Fig. 6 Serial Stewart platform Python simulation

V. Neural Network Architecture

For the neural network architecture for the agent (which is really two neural networks, one for the actor and one for the critic), I chose to use a feed-forward deep neural network whose size is dependent on the number of Stewart platforms. The input layer for the actor and the critic networks is an integer multiple of 18 times the number of SSPs (a multiple of the observation size for the environment). There are a number of hidden layers with the same size as the input. Lastly, the output layer for the actor also happens to be an integer multiple of 18 times the number of SSPs, since the action returned by the agent is a n by 3 by 6 array. The output layer for the critic is simply a single node. Each of the hidden and output nodes uses a rectified linear unit output, a common choice for basic neural networks. A diagram of the NN architecture for an SSP with one Stewart platform is shown to the right.

To find the optimal network architecture, I chose four factors with three levels each. I ran tests using a Taguchi array, which is a type of experimental procedure that allows for the testing of multiple factors with multiple levels while minimizing the number of tests to run [18]. The factors and levels can be seen in Table I, while the Taguchi array used can be seen in Table II. I ran each of the nine tests in the Taguchi array a total of three times for a total of 27 tests.

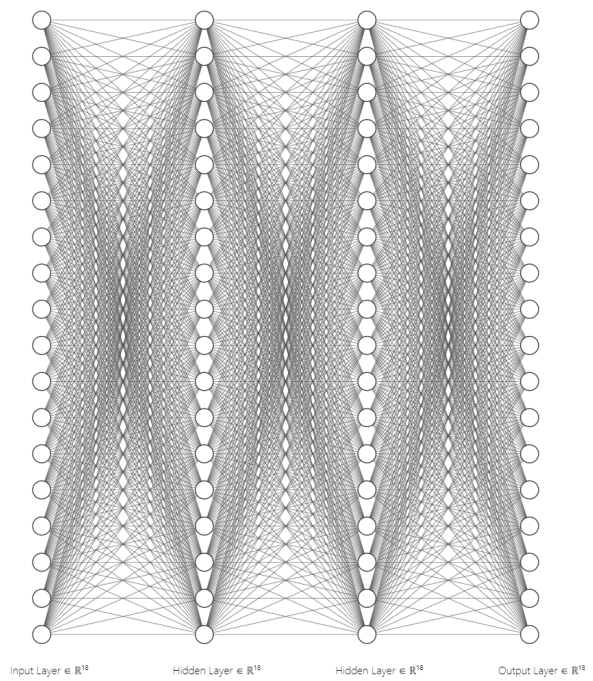

Fig. 7 Actor neural network for num_SSP = 1, num_hidden_layers = 2, size_of_hidden_layer = 18, made using [23]

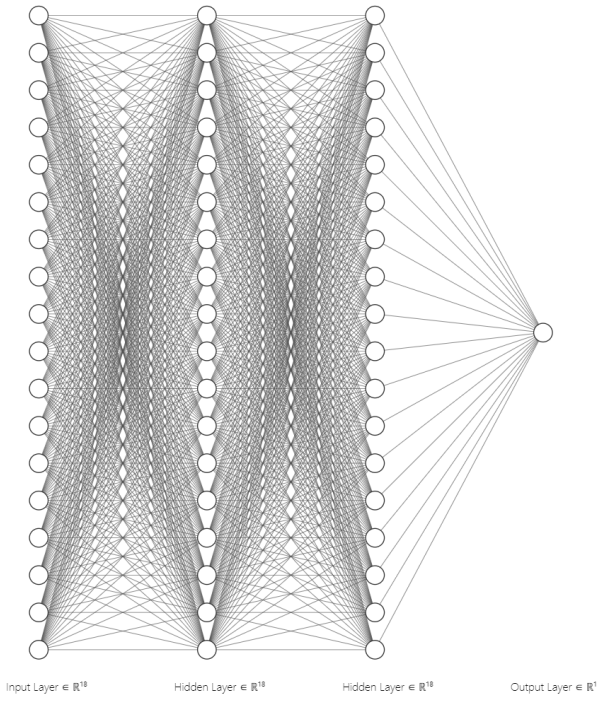

Fig. 8 Critic neural network for num_SSP = 1, num_hidden_layers = 2, size_of_hidden_layer = 18, made using [23]

TABLE I
FACTORS AND LEVELS FOR SIMULATIONS

Factor	Level		
	Low (0)	Med (1)	High (2)
(A) Max timesteps per episode	1k	2.5k	5k
(B) Total timesteps for training	100k	200k	400k
(C) Number of hidden layers in NN	2	4	6
(D) Size of hidden layers in NN	18x2	18x4	18x6

TABLE III
4X3 TAGUCHI ARRAY

Test Number	(A)	(B)	(C)	(D)
1	0	0	0	0
2	0	1	1	1
3	0	2	2	2
4	1	0	1	2
5	1	1	2	0
6	1	2	0	1
7	2	0	2	1
8	2	1	0	2
9	2	2	1	0

VI. TRAINING RESULTS

Before simulating the custom SSP environment, I trained a basic example environment included with Gymnasium, "LunarLander-v2" [12]. The observation and action state and neural network architecture are much smaller in this example environment than my custom SSP environment, but it will provide a good baseline on what to look for. The training process converged above around 200k timesteps. The training results can be seen below.

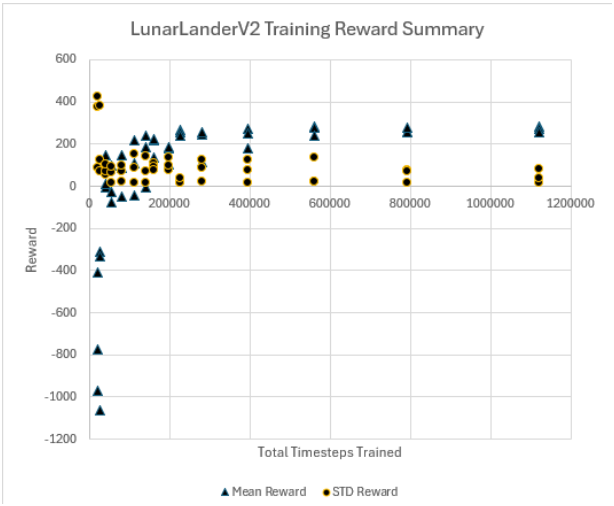

Fig. 9 LunarLander-v2 training reward summary

After each model is trained, the simulation is run for 10 episodes and the mean and standard reward are calculated. The spread of mean and standard deviation of rewards appears to converge as the number of timesteps is increased; there is not much difference between models trained at 800k timesteps versus those trained at 1.1m timesteps for the LunarLanderV2 environment. When training the SSP agent, we will be looking for a similar convergence as the number of timesteps increase.

Below are the results for training the SSP environment following the Taguchi array experimental process as described earlier.

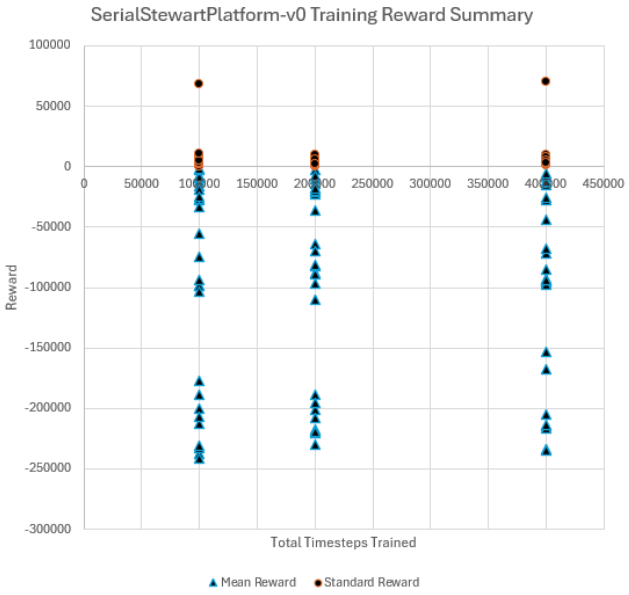

Fig. 10 SerialStewartPlatform-v0 training reward summary

These results are unfortunately less clear than that of the LunarLander-v2 environment, but we can extract some meaning, nonetheless. It appears that the model's performance does not depend on the number of timesteps trained for the range we've selected. However, we have three other factors to look at; additionally, we can consider the interaction between factors. We will initially use the model:

$$y_{ijklm} = \mu + \alpha_i + \beta_j + \gamma_k + \delta_l + \epsilon_{ijklm}$$

where i = 0:2, j = 0:2, k = 0:2, l = 0:2, and m = 0:2. This model assumes that the reward of each training/test can be modeled as the sum of the grand mean (the overall average for every test) plus the affects from the particular level of each factor (e.g.

level 0 for factor A corresponds to the effect that using 1000 max timesteps per episode has on the reward, which appears to be a large positive effect), plus some random variation modeled as ϵ_{ijklm}, where m is the particular replicate for that test [22]. The results of fitting the data to this model in JMP, a statistical analysis software, can be seen below.

Tests wrt Random Effects

Source	SS	MS Num	DF Num	F Ratio	Prob > F
Max Timesteps per Episode	5e+11	5e+11	1	675.3789	<.0001*
Total Timesteps	7.78e+7	7.78e+7	1	0.1049	0.7470
NN Layer Count	2.23e+9	2.23e+9	1	3.0067	0.0871
NN Layer Size	1.48e+9	1.48e+9	1	1.9894	0.1626
Replicate #&Random	1.56e+9	7.82e+8	2	1.0539	0.3538

Fig. 11 Initial model fitting results, no interaction terms

Without going into too much detail on the statistics, the test reveals that the only factor in our model that has a statistically significant effect on the reward for this range of levels is the number of maximum timesteps per episode. If we plot each of the four factors versus the average reward, we can see that this appears to be believable.

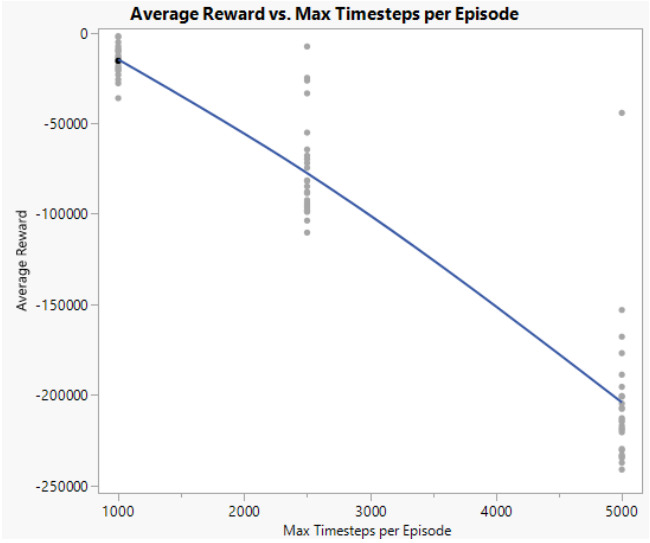

Fig. 11 Average reward as a function of max timesteps per episode

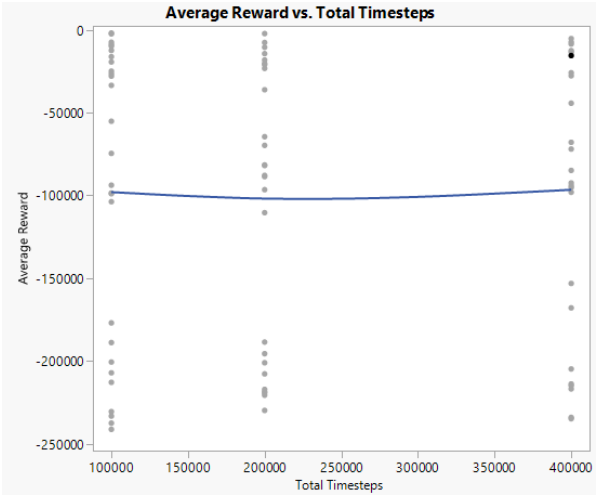

Fig. 12 Average reward as a function of total timesteps

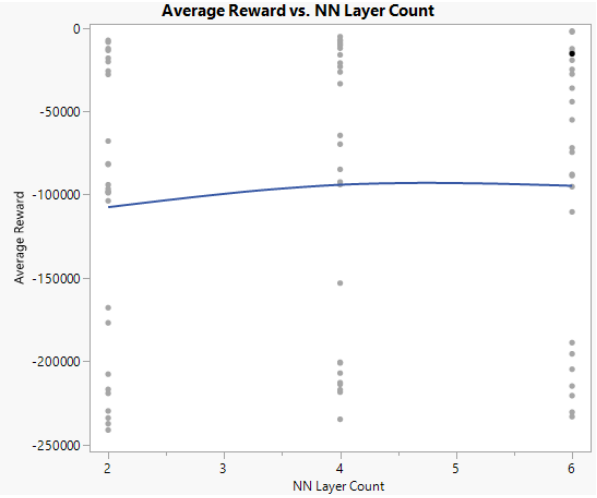

Fig. 13 Average reward as a function of neural network hidden layer count

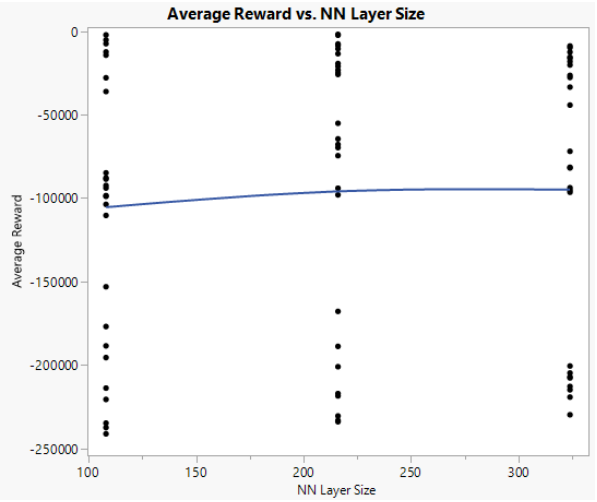

Fig. 14 Average reward as a function of neural network hidden layer size

Now, let's look at possible interactions. We'll modify our model to include any possible two-way interactions between factors. The results of fitting this model in JMP can be seen below.

Tests wrt Random Effects

Source	SS	MS Num	DF Num	F Ratio	Prob > F
Max Timesteps per Episode	5e+11	5e+11	1	689.2105	<.0001*
Total Timesteps	1.08e+7	1.08e+7	1	0.0149	0.9032
NN Layer Count	2.52e+9	2.52e+9	1	3.4695	0.0668
NN Layer Size	1.57e+9	1.57e+9	1	2.1580	0.1464
Max Timesteps per Episode*Total Timesteps	2.83e+9	2.83e+9	1	3.8974	0.0524
Max Timesteps per Episode*NN Layer Count	9.79e+8	9.79e+8	1	1.3464	0.2500
Max Timesteps per Episode*NN Layer Size	3.93e+8	3.93e+8	1	0.5413	0.4644
Total Timesteps*NN Layer Count	1.11e+8	1.11e+8	1	0.1521	0.6978
Total Timesteps*NN Layer Size	5.03e+8	5.03e+8	1	0.6917	0.4085
NN Layer Count*NN Layer Size	1.09e+9	1.09e+9	1	1.5042	0.2243
Replicate #&Random	1.56e+9	7.82e+8	2	1.0754	0.3469

Fig. 15 Model fitting results, all interaction terms

Though none of the interactions have a significance level below 5%, one does come fairly close: the interaction between max timesteps and total timesteps. This tells us that we may want to consider it in our model. Plotting average rewards as a function of total timesteps while blocking for the max number of timesteps per episode produces the following graph.

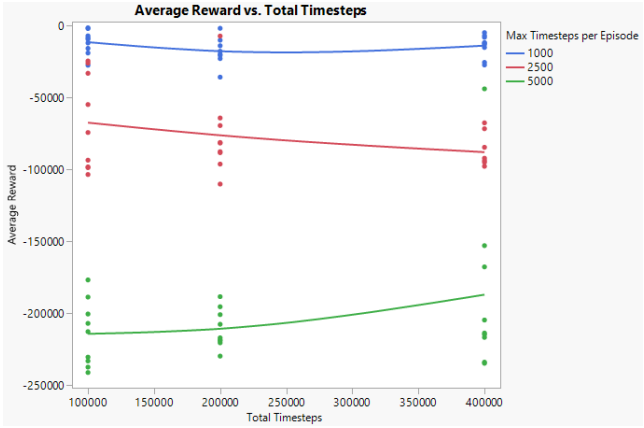

Fig. 16 Average reward as a function of total timesteps, grouped by max number of timesteps per episode

It appears that, though slight, there may be an effect from crossing these two factors. Through all of this analysis, we can conclude that training a model with a lower number of max timesteps per episode (1000) tends to produce a higher average reward; additionally, for tests using a high max number of timesteps per episode (5000), the average reward tends to increase as the total number of timesteps increases. Intuitively, this makes sense; the largest negative penalty comes from overexertion, which is defined as excessive deviation from the neutral position. If each episode is allowed to run for longer, then we would expect the SSP to overexert itself more frequently. Also, this increase in overexertion (and the steep negative penalties that come with it) may cause the model to be more conservative with its exploration. This does not necessarily mean that training with less max timesteps per episode is better, just that it produces a lower average reward. This is a fundamental flaw with this simulation and is addressed in the following section.

We can still use this data to our advantage, however. Based on the results of the analysis, it seems like the best performing model from our tests (highest average reward and lowest standard deviation) was test #7 as seen below. Following a naïve approach, we'll train an agent using similar parameters in hopes of maximizing our average reward.

Test #	Max Timesteps per Episode	Total Timesteps	NN Layer Count	NN Layer Size	Average Reward across Replicates	Standard Deviation across Replicates
1	1000	100000	2	108	-15906	2857
2	1000	200000	4	216	-18261	2067
3	1000	400000	6	324	-18557	2885
4	1000	100000	4	324	-11602	2671
5	1000	200000	6	108	-17549	1332
6	1000	400000	2	216	-15959	5964
7	1000	100000	6	216	-7870	828
8	1000	200000	2	324	-18750	2824
9	1000	400000	4	108	-8216	2570
10	2500	100000	2	108	-100358	4578
11	2500	200000	4	216	-47266	2234
12	2500	400000	6	324	-87429	4217
13	2500	100000	4	324	-51213	4157
14	2500	200000	6	108	-95579	2019
15	2500	400000	2	216	-86636	5264
16	2500	100000	6	216	-51529	2624
17	2500	200000	2	324	-86605	3004
18	2500	400000	4	108	-90375	2879
19	5000	100000	2	108	-218471	27048
20	5000	200000	4	216	-212150	5611
21	5000	400000	6	324	-154578	3413
22	5000	100000	4	324	-206774	4694
23	5000	200000	6	108	-201491	5535
24	5000	400000	2	216	-206172	5181
25	5000	100000	6	216	-217448	6833
26	5000	200000	2	324	-218854	5889
27	5000	400000	4	108	-200514	26169

Fig. 17 Taguchi test results

On the following page is a rendering of an agent trained with 750 max timesteps per episode, 2000000 total timesteps, and 6 hidden layers each with 216 neurons. While the results are not perfect, they are satisfying to watch and are a major improvement over initial testing results.

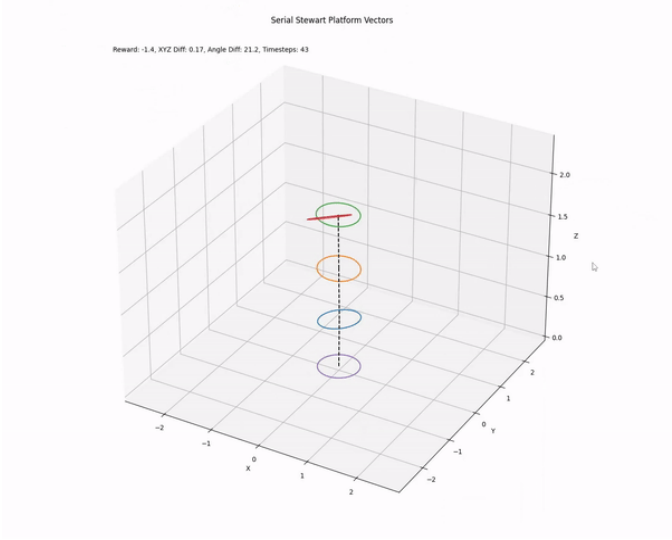

Fig. 18 Best simulation results after optimization

VII. Conclusion

Despite the training results being less than ideal, I am still very happy with this research. As a mechanical engineering student, most of my time is spent designing robotic systems, writing code for microcontrollers, and developing PCBs and other electronic circuits. This is my first "large" coding project not related to a physical mechatronic system. I had to do a lot of background research on deep reinforcement learning theory and implementation, topics that I've heard of but have never tried to recreate. Now, thanks to some helpful documentation on the Gymnasium website [16] and other online resources, I was able to design a completely custom simulation environment for a simplified SSP system. Even though the training did not converge to a usable state, I now have a great foundation for the next deep RL environment that I bring to life.

I have some theories as to what went wrong in the simulation, i.e. what caused the system to not converge on a solution. First, I believe that the constraint system for limiting the range of motion of each Stewart platform was flawed. Instead of prohibiting the platform from overextending, I attempted to heavily punish the agent for overextending. Having the ability to overextend (and incur a huge negative reward) may have conditioned the agent to explore more cautiously and make more conservative changes to the SSP's orientation. Second, some major oversimplifications were made in the design of the SSP simulation. The motion of individual platforms should ideally be simulated by changing the lengths of six linear actuators and having the platform's orientation and position change as a result, as opposed to the current method of applying two rotation matrices to each platform to approximate the platform's controls and motion. Thankfully, both issues can be solved in a fairly straightforward (yet laborious) way: simulating the SSP using real joints and linear actuators in a physics simulation engine like MuJoCo [17] or Gazebo [19], or simply building and training a real-life SSP system by controlling the lengths of real linear actuators. After finishing training in my custom environment, I started researching physics simulation engines that are compatible with Stable Baselines3 and Gymnasium and found some great projects: one that created a Stewart platform in Gazebo [20], and another that used Gazebo with Stable Baselines3 and Gymnasium to train a robot to navigate a hospital environment [21]. The learning curve for Stable Baselines3 and Gymnasium was relatively soft for a first-timer like me, but the learning curve for Gazebo, MuJoCo, and similar engines appears to be much steeper. Given my background, it may be easier (though less efficient) to simply build and train a physical system in real-time.

Overall, I am very proud of myself for plunging into this field and coming out with a reasonable product. There is still a lot of room to grow, but through this project I have built a solid foundation for creating reinforcement learning environments that can be trained and simulated using state-of-the-art algorithms in Stable Baselines3.

REFERENCES

[1] S. Bhatt, "Reinforcement Learning 101," *Medium*, Mar 19, 2018. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292. Accessed: Mar 19, 2024.

[2] W. Ertel, *Introduction to Artificial Intelligence*, 2nd ed. Cham, Switzerland: Springer Nature, 2017, doi: 10.1007/978-3-319-58487-4.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, *et al,* "Human-level control through deep reinforcement learning," *Nature*, no. 518, pp. 529-533, Feb 25, 2015. [Online]. Available: https://www.nature.com/articles/nature14236. Accessed: Mar 19, 2024.

[4] OpenAI, *Spinning Up*, 2024. [Online]. Available: https://spinningup.openai.com/en/latest/. Accessed: Mar 19, 2024.

[5] J. Liu, "DQN vs PPO. Discussion with my mentors," *jerrickliu.com*, May 3, 2021. [Online]. Available: https://jerrickliu.com/2020-07-13-FourthPost/. Accessed: Mar 19, 2024.

[6] E. Yu, "Coding PPO from Scratch with PyTorch (Part 1/4)," *Medium*, Sep 17, 2020. [Online]. Available: https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8. Accessed: Mar 19, 2024.

[7] R. Sutton, A. Barto, "Actor-Critic Methods" in *Reinforcement Learning: An Introduction*, Cambridge, Massachusetts, United States: The MIT Press, 2005, ch. 6, sec. 6. [Online]. Available: http://incompleteideas.net/book/first/ebook/node66.html. Accessed: Mar 19, 2024.

[8] "Stewart platform," *Wikipedia*, Oct 24, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Stewart_platform. Accessed: Mar 19, 2024

[9] D. Balaban, J. Cooper and E. Komendera, "Inverse Kinematics and Sensitivity Minimization of an n-Stack Stewart Platform," *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China, 2019, pp. 6794-6799, doi: 10.1109/IROS40897.2019.8968190.

[10] H. Yadavari, V. Tavakol Aghaei, and S. Ikizoglu, "Addressing Challenges in Dynamic Modeling of Stewart Platform using Reinforcement Learning-Based Control Approach," *J. Robotics Control (JRC)*, vol. 5, 2024, Art. no. 20582. doi: 10.18196/jrc.v5i1.20582.

[11] A. Raffin, *Stable Baselines3*, 2024. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/. Accessed: Mar 19, 2024.

[12] Farama Foundation, "*Gymnasium*," 2024. [Online]. Available: https://gymnasium.farama.org/index.html. Accessed: Mar 19, 2024.

[13] J. Frabosilio, "SB3-PPO-Test," *GitHub*, 2024. [Online]. Available: https://github.com/jfrabosi/SB3-PPO-Test,

[14] "Angle between Two Vectors Formula", *GeeksForGeeks*, Feb 4, 2024. [Online]. Available: https://www.geeksforgeeks.org/angle-between-two-vectors-formula/**.** Accessed: Mar 19, 2024.

[15] "Matplotlib," *matplotlib.org*, 2024. [Online]. Available: https://matplotlib.org/. Accessed: Mar 19, 2024.

[16] "Make your own custom environment," *Gym*, 2024. [Online]. Available: https://www.gymlibrary.dev/content/environment_creation/. Accessed: Mar 19, 2024.

[17] "MuJoCo," *mujoco.org*, 2024. [Online]. Available: https://mujoco.org/. Accessed: Mar 19, 2024. Accessed: Mar 19, 2024

[18] R. N. Kacker, E. S. Lagergren, and J. J. Filliben, "Taguchi's Orthogonal Arrays Are Classical Designs of Experiments," *J. Res. Natl. Inst. Stand. Technol.*, vol. 96, no. 5, pp. 577–591, 1991. doi: 10.6028/jres.096.034.

[19] "Gazebo," *gazebosim.org*, 2024. [Online]. Available: https://gazebosim.org/home. Accessed: Mar 19, 2024.

[20] D. Ingram, "Stewart," *GitHub*, 2018. [Online]. Available: https://github.com/daniel-s-ingram/stewart. Accessed: Mar 19, 2024.

[21] T. Vandermeer, "Hospitalbot path planning," *GitHub*, 2023. Available: https://github.com/TommasoVandermeer/Hospitalbot-Path-Planning/tree/humble. Accessed: Mar 19, 2024.

[22] G. Oehlert, *A First Course in Design and Analysis of Experiments*. New York, New York, United States: W. H. Freemen and Company.

[23] A. Lenail, "NN-SVG," *alexlenail.me*, 2024. [Online]. Available: https://alexlenail.me/NN-SVG/index.html**.** Accessed: Mar 19, 2024.