Control of Simulated Under-damped Series Elastic Actuated Robotic Arm with Reinforcement Learning Algorithms

Landon Clark, Huitao Guan, Sumit Kamat

University of Kentucky

Abstract

In order to better engage with humans cooperants, a robotic arm can be designed to absorb impacts with its surroundings - one such example being a Series Elastic Actuated (SEA) robot. In the pursuit of such a design, our work places a rotary spring between the motor and the link of each actuator, with the spring being attached to freely rotating plates on either side (see fig. 1). Moreover, the springs in the series elastic joints store and release energy during motion and thus this approach can even serve to increase power efficiency of our actuators and provide a non-rigid response to external impacts. Due to the nature of this specific design, the model of our system will be highly non-linear, adding control complexity when compared to standard direct-drive robots. This paper is partially inspired by [1], the novelty of our study is to expand to 3 degrees-of-freedom (DOF) and build a sufficient control algorithm. In this study, we introduce the simulation of a 3-DOF robotic arm with SEAs, using PyBullet[2] as our physics environment of choice. To accomplish this control problem, we consider two reinforcement learning algorithms (DDPG[3] and PPO[4]) - comparing the performance of both. We have chosen machine learning algorithms over traditional control methods as a means to produce accessible and simple controllers, implementable for a wide array of SEA robots given proper training parameters. Our results show a significant difference in learning behavior between DDPG and PPO, with the PPO algorithm exploring the action space much more effectively than DDPG. With proper tuning and more time to train, it could be possible for the PPO algorithm to produce an effect control policy.

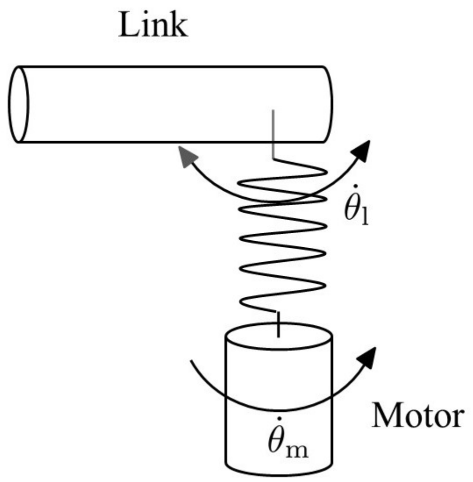

Fig. 1. Sketch of series elastic actuator model simulated in this work

I. Introduction

The design of a robot's structure in today's world typically falls into one of two major categories, rigid or elastic. In recent years, more and more research has been focused on Series Elastic Actuated (SEA) robotic arms. SEAs can create a safer work environment for human cooperants as any collisions with an SEA robot would be absorbed by the robot[1], as they are typically more flexible and lightweight [5]. As such, there is a wide array of applications for SEA robots in certain rehabilitation, collaborative robotic arm, exo-skeleton, and muscle training fields[6], [7].

Elastic components inherently provide a "soft" configuration compared to rigid direct-drive joints by virtue of their energy-absorbing properties. As such, traditional control algorithms fail to precisely achieve the desired trajectory and usually cause large tracking errors [8]. This problem has sparked much interest in the field of SEA torque control. [9] proposes a control method utilizing an optimized Radial Basis Function (RBF) neural network adaptive control algorithm. In [10], feedback linearization control is used for a compliant robotic arm. [11] presents an adaptive neural network tracking control of a robotic manipulator with input deadzone and output constraint, combined with a barrier Lyapunov function to deal with the output constraints. In this study, we present two seperate reinforcement learning algorithms as possible controllers. The first is a widely adopted deep reinforcement learning method, Deep Deterministic Policy Gradient (DDPG)[3]. In addition, a Proximal Policy Optimization (PPO)[4] algorithm is implemented - providing a sense of how effective reinforcement learning algorithms can be at solving this control problem.

The remainder of this paper is organized as follows: Section II describes our simulation methodology including robot design and PyBullet usage. Section III presents the two deep reinforcement learning algorithms used, DDPG and PPO. Section IV describes the results of our control methods given our 3-DOF model.

II. Simulation Model

Before we can simulate a robot, we must first make sure we have an adequate physical model developed. [12] gives the following state space definition:

$$x_1 = \theta_l, x_2 = \dot{\theta_l}, x_3 = \theta_m, x_4 = \dot{\theta_m}$$

where θ_l is the position of the link side of the actuator, and θ_m is the position of the motor portion of the actuator. When considering our physical system as a whole, we can model the link side and motor side forces as

$$\begin{gathered} M\dot{x_2} + Cx_2 + G\sin x_1 = K(x_3 - x_1) \\ J\dot{x_4} + K(x_3 - x_1) = \tau_m + d \end{gathered} \tag{1}$$

respectively, where M is the link inertia, C is the link damping coefficient, G is the gravity vector on the link side, K is the spring constant of a given actuator, J is the inertia on the motor side, τ_m is the motor input torque, and d is the systems lumped disturbances. These equations give us all we need to fully develop our system state space:

$$\begin{aligned}
\dot{x_1} &= x_2 \\
\dot{x_2} &= -\frac{K}{M}x_1 - \frac{G}{M}\sin x_1 - \frac{C}{M}x_2 + \frac{K}{M}x_3 \\
\dot{x_3} &= x_4 \\
\dot{x_4} &= \frac{K}{J}x_1 - \frac{K}{J}x_3 + \frac{1}{J}(\tau_m + d)
\end{aligned} \tag{2}$$

When implementing this system in PyBullet, we let the physics engine account for the M, C, G, J, and d terms. In addition to this, our goal is to apply suitable τ_m values, so this does not fall into our modelling problem either. Thus, we only need to design a model that effectively simulates the spring force both across our motor and across our link. To do so, we create two software joints, one with a dummy link (the motor and a rough estimate of the body it acts on) and the other connected to the remainder of the robot's body. As we are modelling a 3-DOF robot, we repeat this process for each actuator. When we apply a torque to our robot, it is applied to the motor side, with no immediate impact on the link. We then step our simulation, update motor and link positions, and determine what force is to be applied to the link (the same force is applied in the opposing direction to the motor) - determined by $K(x_3 - x_1)$. It should be noted that we make the simulation step size very small, as to accurately represent a continuous and quickly varying force across the spring. Thus our software model is abstracted to three steps per iteration: apply a controlling motor torque, update our model's state, and determine the force across our spring. Once the simulation model was developed, we worked to make it easy to interface with already existing reinforcement learning algorithms. To do so, we implemented some common functions used in reinforcement learning, such as step, state, and reset functions. We should make note of our reward function as well:

$$r_t(a_t|s_{t+1}) = \frac{1}{\sum_i(|s_{i,t+1} - g_i|)} + h(\sum_i(|s_{i,t+1} - g_i|)) \tag{3}$$

where

$$h(x) = \begin{cases} 2.5 - x & x \leq 2.5 \\ 0 & x \geq 2.5 \end{cases}$$

g is our goal state and 2.5 is roughly the initial value of x in $h(x)$ when beginning the simulation.

III. Algorithms

In this section, we will briefly describe the algorithms that we have used to learn the control policies.

A. Deep Deterministic Policy Gradient (DDPG)

Standard reinforcement learning methods adopt an actor interacting with the environment to obtain a maximum long-term reward. However, most early reinforcement learning algorithms required discrete state spaces and action spaces. DDPG was developed to solve this problem, providing the ability to handle continuous problems with the familiar actor-critic model. The DDPG algorithm utilizes four distinct neural networks, a Q-Network, a deterministic policy network, a target Q network, and a target policy network. The Q-Network and deterministic policy network are very similar to the simple Advantage Actor-Critic. A major difference is that DDPG provides a deterministic result, with the action being directly obtained from state through the actor instead of a probability distribution across a discrete action space. The corresponding target networks on the other hand, are delay copies of the original neural networks that can slow-track the learned networks. By using these target networks, the stability of the learning process is greatly increased.

The steps of the DDPG algorithm can be boiled down as follows. First, a deterministic exploration policy is obtained by combining the action $\mu(s_t)$ given by the actor network with a noise given by an Ornstein-Uhlenbeck process[3], $\mathcal{N}$. Then with the action applied in the environment, we get the next state s_{t+1} and the reward r_t. All the transition information (s_t, a_t, r_t, s_{t+1}) is recorded in an experience replay buffer. We then train our networks by sampling the transition information using a minibatch approach, sampling only a select number of elements from the experience replay. Subsequently, we update the critic network's weights with the appropriate loss relative to the reward we received at transition i and the policy network's weights using the policy gradient. Finally, the target networks are updated using a soft update strategy. A soft update strategy consists of slowly blending the initial network weights with target network weights. Updating the weights of the target networks slowly promotes greater stability of the learning process.[3] (see Appendix A1 for the DDPG algorithm)

B. Proximal Policy Optimization (PPO)

Let the stochastic policy be $\pi_\theta(a_t|s_t)$. Next, consider the probabilistic ratio given by

$$r(t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}. \tag{4}$$

where θ_{old} is the vector of the policy parameters before the update t. In Trust Region Policy Optimization, the aim is to maximize a surrogate objective

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t[r(t)\hat{A}_t] \tag{5}$$

where $\hat{A}_t$ is an advantage function. The superscript CPI stands for conservative policy iteration as without a constraint or penalty maximization of $L^{\text{CPI}}(\theta)$ can lead to a very large policy update.

Thus, we consider PPO algorithms. There are two PPO algorithms, PPO clipped, and PPO with penalty. In PPO with penalty we aim to maximize the surrogate objective

$$L^{\text{penalty}}(\theta) = \hat{\mathbb{E}}_t[r(t)\hat{A}_t - \beta \text{KL}[\pi_{\theta\text{old}}(., s_t), \pi_\theta(., s_t)] \quad (6)$$

where $\hat{A}_t$ is an advantage function and β is some coefficient. The surrogate objective is determined by computing the max Kullback-Leibler (KL) divergence over the states forms a lower bound on the policy π. Note, that we have implemented PPO with penalty.

In PPO clipped we use a clipped surrogate objective function $L(\theta)$ which is given by

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \quad (7)$$

where $\hat{A}_t$ is an advantage function which is dependent on the current probabilistic ratios and all the prior probabilistic ratios, and ϵ is a hyperparameter.

In PPO clipped, we clip the probabilistic ratio $r_t(\theta)$, such that it is bounded within a small interval around 1, i.e., $[1-\epsilon, 1+\epsilon]$. By doing so we limit the difference between θ_{old} and θ, which results in better stability. Finally, we take the minimum of the clipped and unclipped (CPI) objective function, so the final objective is a lower bound of the unclipped objective. [4] (See Appendix A.2 for the PPO algorithm)

IV. Results

Given the inherent complexity of this control problem, it is not surprising that our control methods were generally unsatisfactory. Let us first consider the DDPG results

A. DDPG Results

The DDPG algorithm produced very stable, but generally worthless results. The learning algorithm converged almost immediately on every training attempt to torque outputs of only maximum or minimum values (a common DDPG problem discussed in [13]). This would cause the arm to stay stationary and achieve some reasonable rewards, but it never effectively explored its action space. The following figure shows the torque output during the first training episode of a DDPG run:

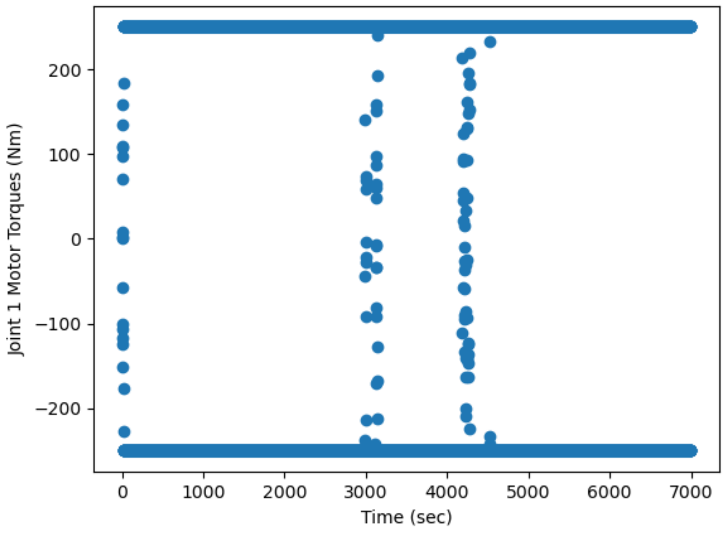

Fig. 2. This figure demonstrates the immediate convergence of the DDPG algorithm to produce either minimum or maximum torque values.

While the torque outputs were unideal, the rewards at each episode would make the DDPG algorithm appear effective as seen in the following figure:

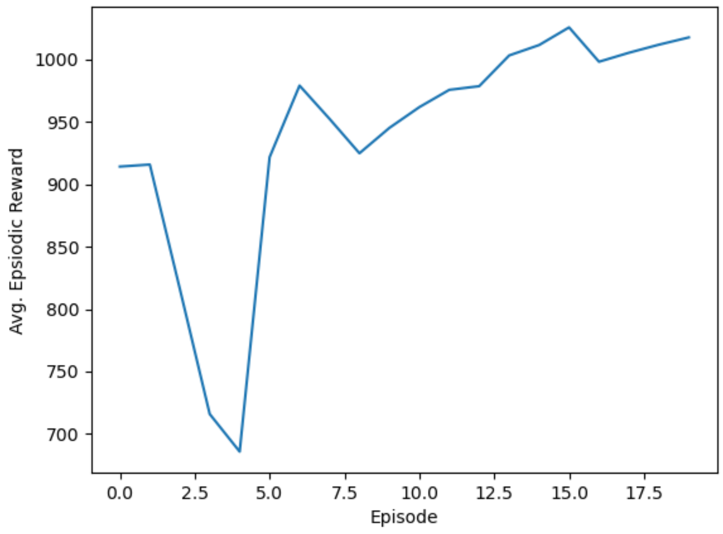

Fig. 3. This figure shows consistent reward increases despite ineffective exploration.

However, as the rewards are consistent only because the robot was able to stay stationary in a rewarding region, we can view these results as insignificant.

B. PPO Results

While proving to be ineffective in our implementations, the PPO algorithm showed promise with its exploration ability. However, while it did explore well, it typically failed to converge on policies that created high long-term rewards. The following figure shows a plot of three separate learning attempts using PPO:

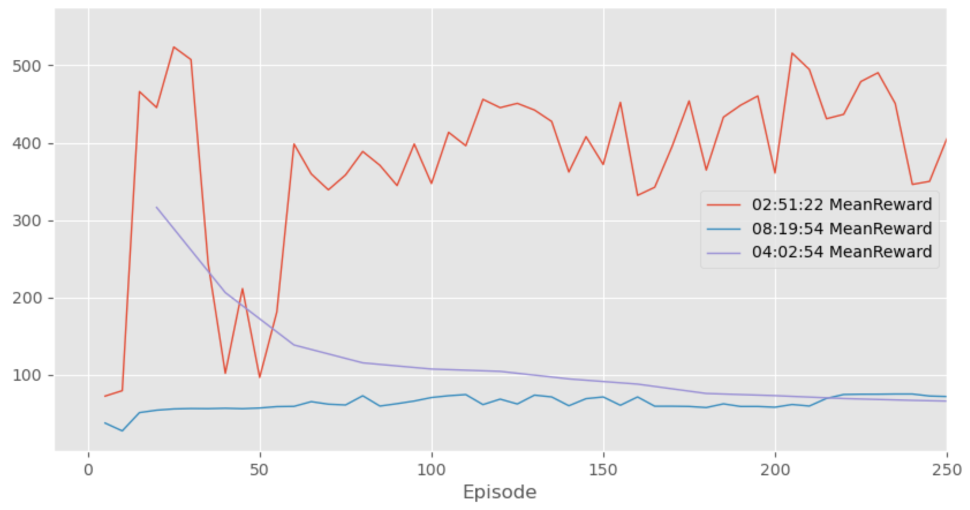

Fig. 4. This figure shows some ability for the PPO algorithm to find semi-effective policies, although tending to opt for ineffective policies as time goes on.

The most likely cause of divergence in our training was the inability to explore actions that brought our robot out of high velocity states. Once our robot became unstable, it was unable to learn policies that would apply large opposing forces to stabilize.

V. Conclusions and Future Work

For the most part, our results proved to be less than satisfactory, with neither one of our algorithms producing an effective control policy. It appears that the PPO algorithm at least shows some promise in this application. Given improved tuning, longer training times, and perhaps some variations to the traditional PPO algorithm, PPO could prove to be the algorithm of choice for this control problem. However, perhaps our work is best used as an insight into some serious limitations of reinforcement learning algorithms in the controls field. Given reasonable constraints on time and computing power, our work would suggest it is highly unlikely that any practical results be attainable. As such, further long-term study and testing would need to be done to guarantee whether or not reinforcement learning can be an effective, simple solution to this problem.

Appendix

A1. Project Repositories

- Our work can be found at [14]
- PyBullet [2]
- DDPG implementation [15]
- PPO implementation [16]

A2. Deep Deterministic Policy Gradient (DDPG) Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, ... , $M$ do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for $t$ = 1, ... , $T$ do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    end
end
```

A3. Proximal Policy Optimization (PPO) Pseudocode

Algorithm 2 Proximal Policy optimization

```
for iteration=1,2,..., do
    for actor=1,2,...,$N$ do
        Run policy $\pi_{\theta_{old}}$ in environment for $T$ time steps Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
    end
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$ $\theta_{old} \leftarrow \theta$
end
```

References

[1] C. Liao, H. Ma, and H. Wu, "Adaptive control of series elastic actuator based on rbf neural network," in *2019 Chinese Automation Congress (CAC)*, 2019, pp. 4365–4369.

[2] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[5] Q. Zhu, Y. Mao, R. Xiong, and J. Wu, "Adaptive torque and position control for a legged robot based on a series elastic actuator," *International Journal of Advanced Robotic Systems*, vol. 13, no. 1, p. 26, 2016. [Online]. Available: https://doi.org/10.5772/62204

[6] T. Chen, R. Casas, and P. S. Lum, "An elbow exoskeleton for upper limb rehabilitation with series elastic actuator and cable-driven differential," *IEEE Transactions on Robotics*, vol. 35, no. 6, pp. 1464–1474, 2019.

[7] S. Li, J. Li, G. Tian, and H. Shang, "Stiffness adjustment for a single-link robot arm driven by series elastic actuator in muscle training," *IEEE Access*, vol. 7, pp. 65 029–65 039, 2019.

[8] L. Sun, M. Li, M. Wang, W. Yin, N. Sun, and J. Liu, "Continuous finite-time output torque control approach for series elastic actuator," *Mechanical Systems and Signal Processing*, vol. 139, p. 105853, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888327018308082

[9] Z. Q. Shao N. and S. C, "Adaptive control of robot series elastic drive joint based on optimized radial basis function neural network. int j of soc robotics," *International Journal of Social Robotics*, vol. 139, p. 105853, 2021.

[10] J. C. Cambera and V. Feliu-Batlle, "Input-state feedback linearization control of a single-link flexible robot arm moving under gravity and joint friction," *Robotics and Autonomous Systems*, vol. 88, pp. 24–36, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889016303840

[11] W. He, A. O. David, Z. Yin, and C. Sun, "Neural network control of a robotic manipulator with input deadzone and output constraint," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 6, pp. 759–770, 2016.

[12] W. Yin, L. Sun, M. Wang, and J. Liu, "Position control of a series elastic actuator based on global sliding mode controller design," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 3, pp. 850–858, 2019.

[13] G. Matheron, N. Perrin, and O. Sigaud, "The problem with DDPG: understanding failures in deterministic environments with sparse rewards," *CoRR*, vol. abs/1911.11679, 2019. [Online]. Available: http://arxiv.org/abs/1911.11679

[14] L. Clark, H. Guan, and S. Kamat, "Series elastic actuated controller, rl controller for 3-dof sea robot," https://github.com/landonclark97/ME699_RMC/sea_controller, 2021.

[15] H. Singh, "Deep deterministic policy gradient (ddpg)," https://github.com/keras-team/keras-io/blob/master/examples/rl/ddpg_pendulum.py, 2020.

[16] D. Chen, "Ppo tensorflow," https://github.com/Derekabc/ppo_tf, 2020.