RL accelerated control of a 6-DOF robotic arm

AE 494: BTP-II

by

Pranav Malpure (200010055)

under the guidance of

Prof. Mayank Baranwal

April 20, 2024

Department of Aerospace Engineering

Indian Institute of Technology, Bombay

Contents

Acknowledgements

I would like to express my deep gratitude to Prof. Mayank Baranwal, my guide, for his patient guidance, encouragement and valuable feedback throughout the duration of this project. I would also like to thank Prof. Rohit Gupta, my co-guide, for supporting my BTP throughout the year.

Chapter 1

Introduction

Effectively controlling a robotic arm and making it do general tasks is a major challenge to tackle in robotics. While the theory of robotics based upon kinematic equations and linear algebra control is very robust and help us to solve the challenges in a very formal way, it is not possible to model each and every scenario in the environment all the time. This calls for a data based approach to solve the challenges. In this project, I analyse the use of reinforcement learning based control of robotic arm to guide it to a specific point in 3-D space. Although there are various inverse kinematics based approaches, it is interesting to see how can we use RL to perform this task and then further develop on doing specific tasks. The report is structured in the following way – I firstly describe the environment which I am using to simulate my algorithms, then I discuss about the RL algorithm which I use to approach the problem, then I discuss the results which I achieved using my algorithm, and then the future scope of improvements.

Chapter 2

Methodology/Theory

2.1 Modelling the environment & tasks using Google Deepmind's Control Library

To effectively implement reinforcement learning (RL) for controlling the robotic arm in a dynamic and unpredictable environment, it is crucial to accurately model both the surroundings and the specific tasks at hand. Due to the already present vast number of state of the art environments simulators available, it was imperative that I pick one of them according to my needs and proceed with that. After surveying a number of possible such environments, I decided to use the Google Deepmind's control library - *dm_control*, which works in an integrated way on the top of mujoco, a physics engine, and provides a set of environments and tasks to implement RL algorithms on.

Figure 2.1: The kinova Jaco arm

2.1.1 Using *dm_control*

To get familiar with the library, I began with understanding the code structure and then performed a few basic tasks and simulations. The main components of *dm_control* include a Mujoco wrapper which provides convenient bindings between physics simulator and RL code. It provides a set of physics-based environments that can be used for reinforcement learning tasks. These environments include tasks such as humanoid and quadruped locomotion, manipulation, and navigation.

The Environment class is a lower-level abstraction that represents the physical simulation environment. It is responsible for interacting with the underlying MuJoCo simulation engine and provides the necessary infrastructure for defining and working with tasks. The Environment class can be instantiated using the load method, which takes an optional seed argument for seeding the random number generator used by the environment. It provides a number of method functions such as: *reset()* Initialises the state, sampling from some initial state distribution. *step()* Accepts an action, advances the simulation by one time-step, and returns a TimeStep namedtuple. The step function accepts an action as input, and computes the next state of the environment based on the dynamics defined in the underlying MuJoCo physics engine. *action_spec()* describes the actions accepted by an Environment. The method returns an ArraySpec, with attributes that describe the shape, data type, and optional lower and upper bounds for the action arrays.

The Composer framework organises RL environments into a common structure and endows scene elements with optional event handlers. At a high level, Composer defines three main abstractions for task design: The *composer.Entity* serves as a modular and self-contained component, encapsulating an MJCF model, observables and potentially callbacks executed at specific stages of the environment's lifecycle. These entities can be organized into a hierarchical tree structure by attaching child entities to a parent. Conventionally, the top-level entity is referred to as an "arena" and provides a fixed ¡worldbody¿ for the consolidated MJCF model.

A composer.Task comprises a tree of composer.Entity objects that populate the physical scene and offer reward, observation, and termination methods. Additionally, a task can define callbacks to implement "game logic," such as modifying the scene in response to various events, and providing additional observables specific to the task.

The composer.Environment wraps an instance of composer.Task within an RL environment that agents can interact with. This entity takes on the responsibility of compiling the MJCF model, triggering callbacks at relevant points in an episode and determining termination conditions, whether through task-defined criteria or a user-defined time limit. It also manages a random number generator state used by the callbacks, ensuring reproducibility in experiments.

Figure 2.2 illustrates the sequence of callbacks in the Composer framework, organized into events occurring during the reset of an RL episode and those occurring during the stepping of the environment. Each callback is executed in a specific order, starting with the one defined in the Task, followed by those defined in each Entity during a depth-first traversal of the Entity tree, starting from the root (conventionally the arena) and following the order of attachment.

During the reset phase, the first of the two callbacks is *initialize_episode_mjcf*. This callback enables modifications to the MJCF model between episodes, allowing changes to quantities that are fixed once the model is compiled. These modifications impact the generated XML, which is then compiled into a Physics instance and passed to the *initialize_episode* callback, where the initial state can be set.

The Environment.step sequence begins with the *before_step* callback, playing a crucial role in translating agent actions into the Physics control vector. To ensure stability, a sub-step loop is introduced to decouple potentially very small physics simulation steps from the agent's control time-step. Each Physics substep is preceded by *before_substep* and followed by *after_substep*. These callbacks are useful for detecting transient events in the middle of an environment step, such as a button press.

The internal observation buffers are updated according to the configured *update_interval* of each Observable unless the substep is the last one in the environment step. In this case, the *after_step* callback is called before the final update of the observation buffers. The internal observation buffers are then processed based on the delay, *buffer_size*, and aggregator settings of each Observable to generate "output buffers" that are returned externally.

At the conclusion of each Environment.step, the Task's *get_reward, get_discount*, and *should_terminate_episode* callbacks are invoked to obtain the step's reward, discount, and termination status, respectively. It's recommended to compute these values in the *after_step* callback, cache them in the Task instance, and return them in the respective callbacks, as they are often interdependent.

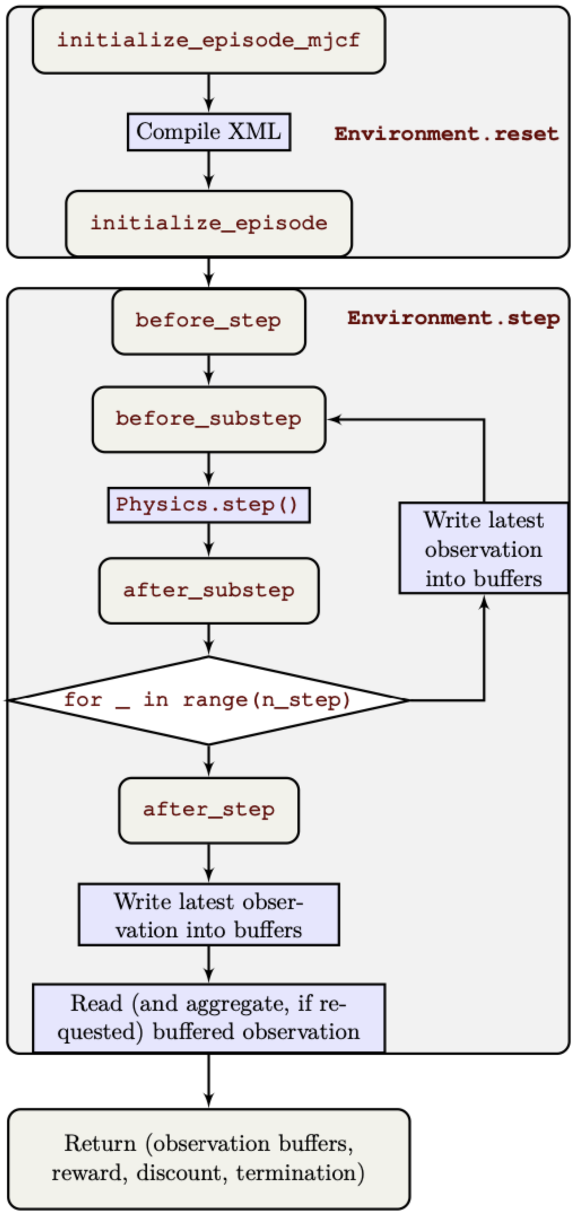
initialize_episode_mjcf
Compile XML
Environment.reset
initialize_episode
before_step
Environment.step
before_substep
Physics.step()
Write latest observation into buffers
after_substep
for _ in range(n_step)
after_step
Write latest observation into buffers
Read (and aggregate, if requested) buffered observation
Return (observation buffers, reward, discount, termination)

Figure 2.2: Diagram showing lifecycle of composer callbacks. Rounded rectangle represent that Tasks and entities may implement. Blue rectangles represent built-in Composer operations.
source: https://arxiv.org/pdf/2006.12983.pdf

2.2 Selecting the task and the environment

Building custom tasks and mujoco models is an extensive work, and would have occupied much of my time just to set things up, defeating the main purpose of the project. Thus I decided to go with pre-defined kinova robotic arm based environment, which also has a lot of pre-defined structures for general tasks. It provides 6 degrees of freedom for the robotic arm, and further 3 for its gripper claws, one in each. See Fig 2.1. For further experiments, this environment also provides compatibility and easy scalability.

Now that once the environment and task to be done is finalised, we are all set to implement RL algorithms.

2.3 Selecting an RL policy

Choosing an RL policy involves numerous considerations, with various options available, each with slightly different objectives. In this case, a policy which will explore significantly will be better suited due to the highly complex and continuous action space involved in guiding the robotic arm to a specific point in space.

Actor-Critic methods offer a distinct advantage in this context by providing a framework that combines the benefits of both policy learning for allowing exploration, and value learning enabling effective evaluation of actions in the given environment. This dual approach enhances the adaptability of the RL algorithm to the complexities of our task giving more efficient learning and convergence towards optimal solutions. In an Actor-Critic method, the actor updates the policy based on θ parameter, and the critic evaluates the policy and provides input for gradient descent. The policy is referred to as the actor that proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy.

$$\theta_{new} \leftarrow \theta + \alpha \sum_{t=0}^{T-1} (r^t + \hat{V}(s^{t+1}) - \hat{V}(s^t)) \nabla_\theta ln[\pi_\theta(s^t, a^t)]$$

2.4 Soft-Actor Critic

Soft Actor Critic, or SAC, is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework. In this framework, the actor aims to maximize expected reward while also maximizing entropy. That is, to succeed at the task while acting as randomly as possible. Prior deep RL methods based on this framework have been formulated as Q-learning methods. SAC combines off-policy updates with a stable stochastic actor-critic formulation. A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration - exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

$$J(\pi) = \sum_{t=0}^{\infty} \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[\sum_{l=t}^{\infty} \gamma^{l-t}\, \mathbb{E}_{\mathbf{s}_l\sim p,\mathbf{a}_l\sim\pi} \left[r(\mathbf{s}_t,\mathbf{a}_t) + \alpha\mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t)) | \mathbf{s}_t,\mathbf{a}_t \right] \right]$$

2.5 Hyperparameters tuning

Now once we fix a policy, the next thing to select are the hyperparameters. These include architecture of the neural network used for the policy and Q value function, the number and type of observations, the reward structure, episode terminating condition, control actions and entropy temperature to name a few.

Chapter 3

Experiments and Results

Model was trained on episodes lasting 15 seconds of duration. Every episode started with a random start and target point for the end effector. If the previous episode ended in a success, the start and end points were randomly sampled from the reachable space of the end effector. The control actions to the model were values between 0 to 2π, subsequently mapped to respective control signals internally by the simulator. The success of an episode is defined in a way that if the end effector reaches the target sphere(a sphere of radius 0.15m around the target point) and stays there for 1.5 seconds. Past learnings from BTP-I has helped in tuning some parameters in an efficient way.

3.1 Neural Network Architecture

The size of neural network was reduced from 6x256x128x6 to 15x12x8x6 for the policy actor, and 21x15x9x1 for the critic q function. This led to faster training period and reduced the heavy underfitting happening before. The below plots 3.1 and 3.2 show the comparison between the higher number of neurons and lesser number of neurons based architecture. In the lesser neurons architecture, the episode is successful more number of times, while for the more neurons case, only once did the episode ended in a success.

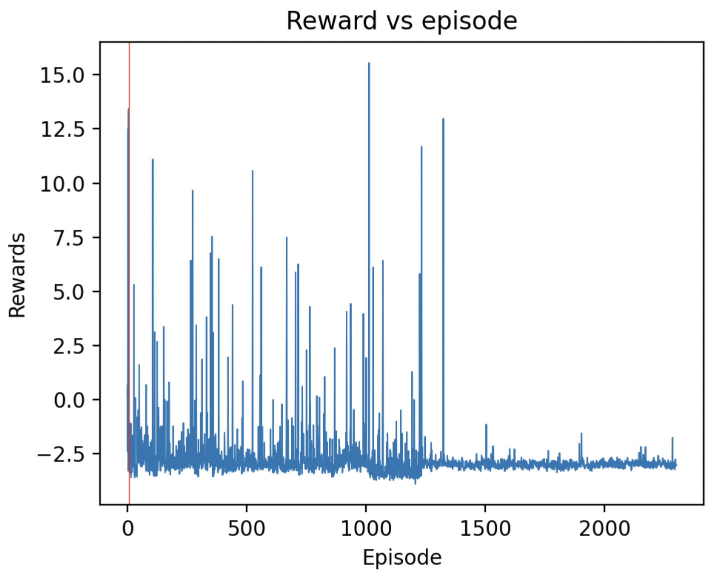

Figure 3.1: Before decreasing the number of neurons per layer

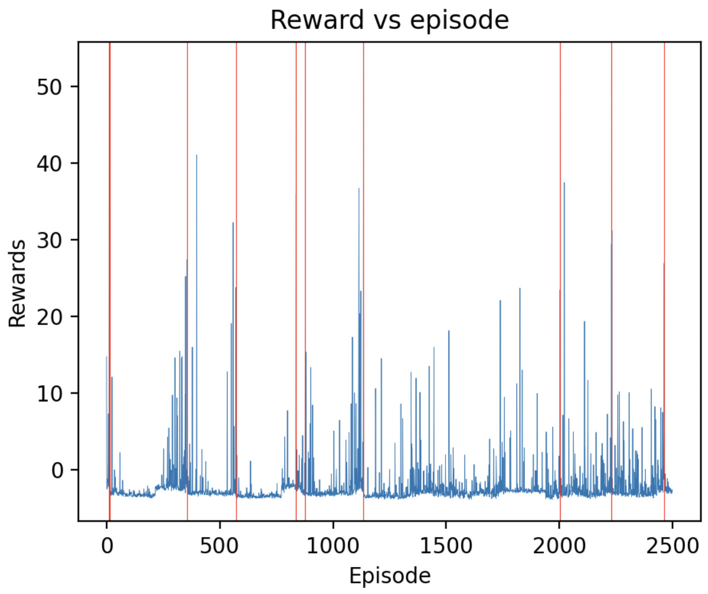

Figure 3.2: After decreasing the number of neurons per layer

3.2 Observation values

An important decision was to make was the observation values to be chosen to provide to the agent. The obvious observation values are that the agent should know about the current joint positions. But there can be several ways of reaching those joint positions. The observation values should be such that when the agent has to guide the arm from point A to point B in space, it should also be able to learn to do it from A to B' provided A' and B' are equivalent of A and B in some other coordinate frame, or just shifted with the spatial geometry remaining the same. To induce this behaviour in the agent, additional observations were provided in the form of the current joint velocities, and the current x,y,z location of the end effector in space. The below plots 3.3 and 3.4 reflect the impact of adding the end effector pos in the observation values. The addition of the three observation values, lead to more positive episodic rewards.

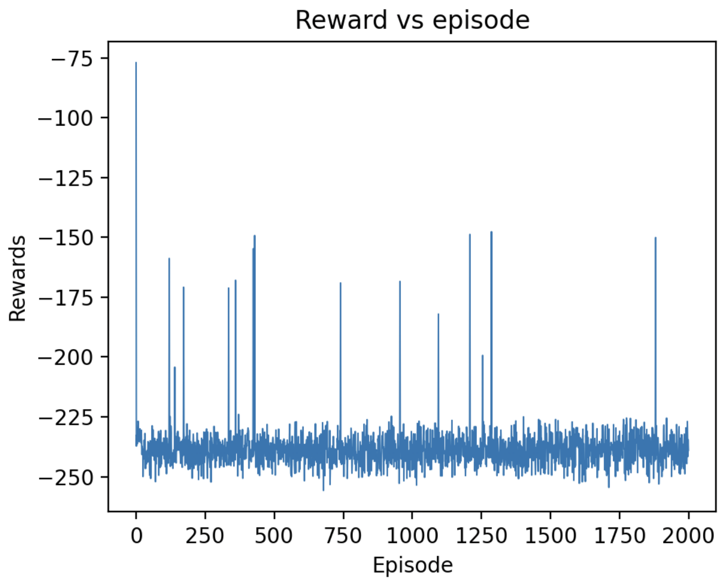

Figure 3.3: Before adding end effector position as an observation value

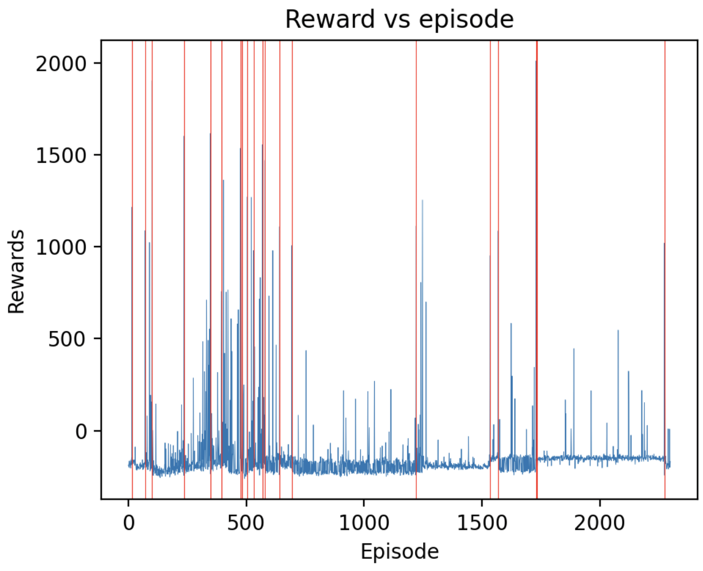

Figure 3.4: After adding end effector position as an observation value

3.3 Entropy coefficient

An advantage of using the Soft Actor critic method is the availability of the control of the entropy term. The entropy term controls the amount of exploration, which can be adjusted by changing the entropy coefficient known as temperature. The $\alpha\mathcal{H}(\pi(\cdot|s_t))$ is the entropy coefficient in the expression for $J(\pi)$. This parameter was tuned dynamically during the training to get better results. It started with fixing the target entropy to a fix value of 0, or log(6) signifying the 6 different control actions, but the results did not meet expectations. After a lot of experimenting, the entropy coefficient was set as follows. It was started with 1 in the beginning, and as the episodes progressed, the entropy coefficient was discretely increased to make the agent explore more until an episode resulted in a successful path to target. Once target was reached the entropy temperature was again set to zero. This pattern resulted in a more converged training compared to other experiments and hence was followed for the first 12000 episodes in the plot above. Once enough training was done, this tuning was done away with and the temperature remained the same, i.e. 1. The plots 3.5 shows the variations of entropy in the final training.

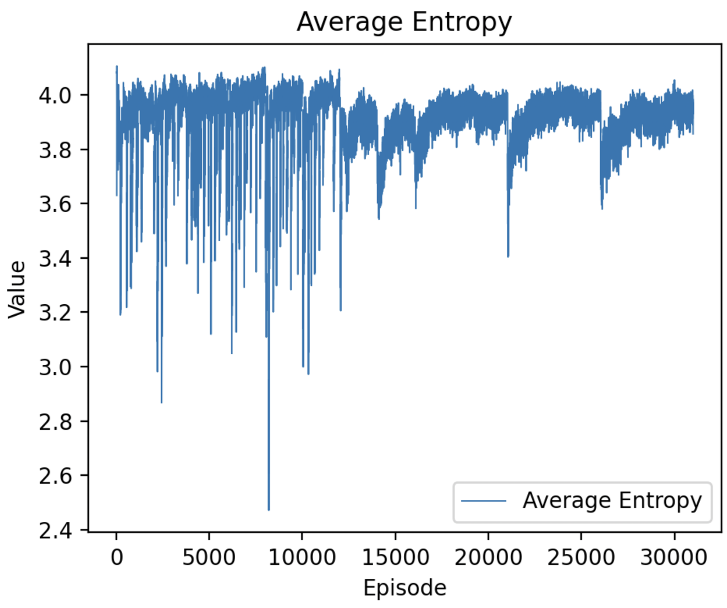

Figure 3.5: The varying entropy during the final training

3.4 Reward Structure

The reward structure was modified to give the policy a better training incentive. The reward was defined as:

If the end effector is inside the target sphere:

$$Reward = \frac{time\_inside\_radius * 100 * Target\_radius}{2 * distance * 40}$$

The above reward structure provides a greater reward for staying inside the target sphere for a longer time, as well as incentivises it to go further towards it centre by providing a greater reward for lesser distance to the centre. The other factors in the denominator are to reduce the magnitude of the total episodic reward and limit it to around 20 on the upper side.

If the end effector outside the target sphere:

$$Reward = \frac{-1 + e^{-distance}}{65}$$

Here, if the end effector is outside the target sphere, then this gives a negative reward while conditioning that the further away the end effector is from the target sphere, the more negative reward it gets. As we can see that -1 + exp will always be negative between 0 and 1, and as distance gets closer to zero, the magnitude of e term increases, thus bringing the reward closer to zero. The factor of 65 is obtained through experiments which limits the total episodic reward to upto min of -4.

The reward plot sans the factors is shown below in 3.6. As is seen, the rewards structure did not encourage the right actions. There were many other reward structures tried apart from this to reach the current final reward structure.

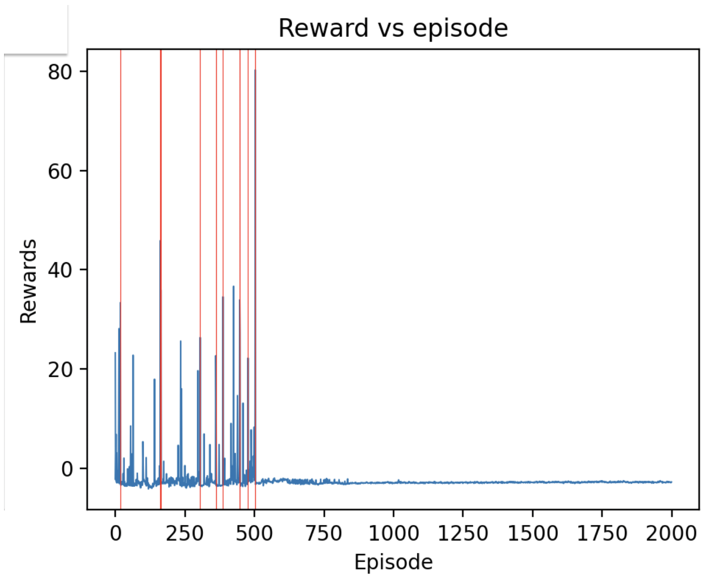

Figure 3.6: The lesser red lines implying that reward structure not efficient

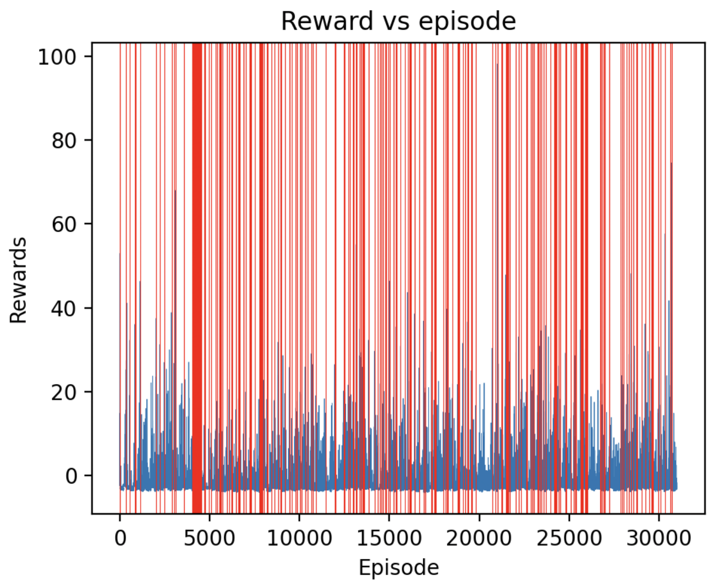

Figure 3.7: The red lines become slightly more dense as we train more episodes

3.7 is the final training plot. The red lines corresponding to the episode number corresponds to a successful episode. The below reward vs episode plot were reached by following the above reward structure.

Conclusion

With the plot in 3.7, we can observe that the density of red lines(representing episode success) is stagnated towards the end, implying that the model has converged. The model when successful, finds a trajectory to the final target sphere and stays inside the sphere for 1.5 seconds, which is defined to be as a successful episode. The problem being solved is a 'control' problem, and one may wonder if is this already acheived through inverse kinematics' numerical solutions. Although numerically solving inverse kinematics presents us with the values of the final joint positions required, it doesn't reveal the trajectory from the current to the final joint positions. Similarly, it doesn't consider the physical effects of the environment and self collisions as well.

At the current stage of the model, further training is unlikely to significantly increase accuracy, as evidenced by the saturation of the success rate depicted in the 3.7. Achieving higher accuracy would require a different approach, particularly regarding the observation values utilized and the corresponding rewards. It's important to note that in our case, episode success is defined by the duration the end effector remains inside the sphere. However, upon reviewing the reward plot 3.7, we observe that in every episode where the episodic reward is positive (the blue plot), the end effector enters the target sphere for a period before moving out. This outcome serves as an initial stage, and subsequent tasks, such as object manipulation and gripping, can be pursued once the end effector successfully reaches the sphere, at our current stage of the model.

Further Scope

The project started out with an aim to pick up random objects from an unknown surrounding. But the task of the navigating it to a random target posed multiple problems which occupied up the future task in line. Now that the arm is robustly able to navigate to a point in 3D space, it can be left to see the possibilities ahead to grip and pick up objects. The next task can be achieved through a mix of vision and force based inputs by placing force sensors on various positions on the gripper. This blend of vision and force inputs will enhance the arm's ability to perceive and interact with its surroundings, paving the way for more sophisticated object manipulation tasks.

References

1. dm_control: Software and Tasks for Continuous Control
2. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor
3. PFRL Docs
4. Proximal Policy Optimization Algorithms
5. Trust Region Policy Optimization
6. Continuous Control with Deep Reinforcement Learning