Physics-Informed Proximal Policy Optimizer for Space Game navigation

Benjamin Bradley, Sofia Tazi

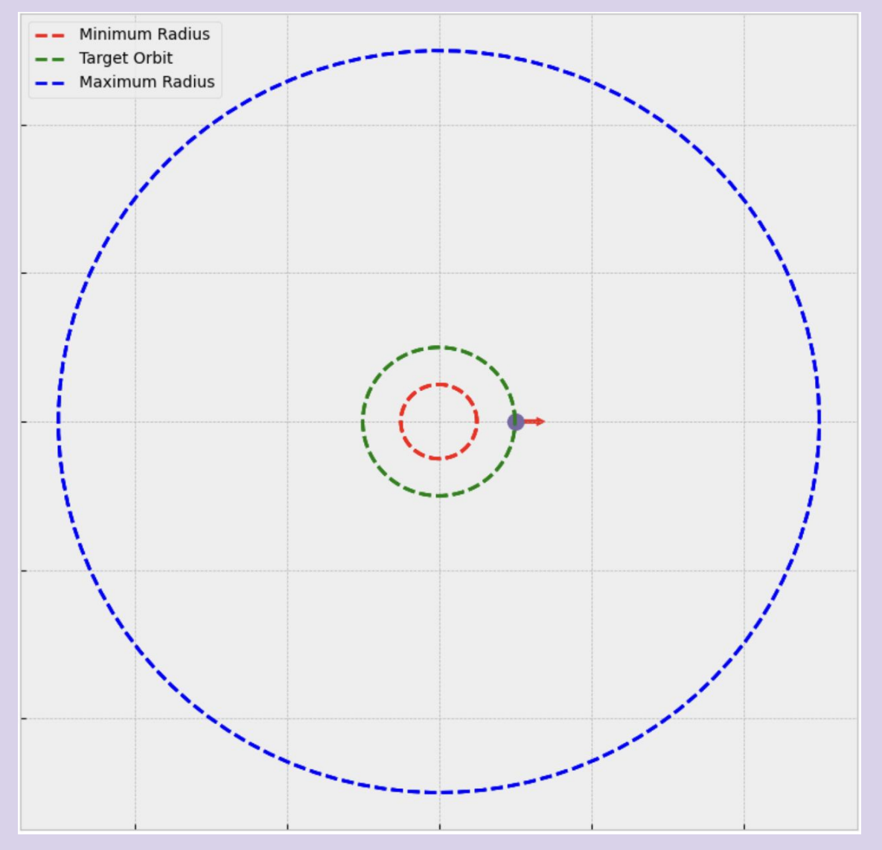
BROWN

Introduction

Our **goal** was to develop a educational and competitive physics-informed RL framework capable of piloting a spacecraft around a *gravitational game space* into stable and sustainable **orbit**. In this game the player is represented by a spacecraft spawned in orbit around a black hole, the player is given control over **thrusting** the spacecraft in order to achieve a variety of goals such as: *navigating* into a different orbit, *surviving* upcoming collisions, and *minimizing* fuel usage to survive as long as possible.

This project functions largely to produce educational content for the Center for Fundamental Physics of the Universe's (CFPU) Student Machine Learning Initiative (SMLI) as part of their Machine Learning Challenge for Physics Students of Brown's Winter School.

Our research these past months has focused on **training** a RL agent capable of navigating from orbit at random radii to stable orbit at r=1.



Environment

Defining Simulation Parameters

1

Specified

G	1
M	1
r_target	1
v_target	$1 = \frac{1}{\sqrt{r}}$
T	2Π
L/m	1

μ = GM = 1

$$\frac{GMm}{r^2} = \frac{mv_t^2}{r}$$

F_gravitational F_centropidal

Table 1: Our natural number unit system

Description : Our environment simulates the motion of a spaceship around a central gravity well of mass M with applied thrust by the spacecraft to alter it's trajectory. Simulation runs until the spaceship either reaches a boundary or a maximum step count. The objective is to reach a stable orbit of radius 1 AU, within this environment

Based on seeking an orbit of 1, the angular momentum L can be determined :

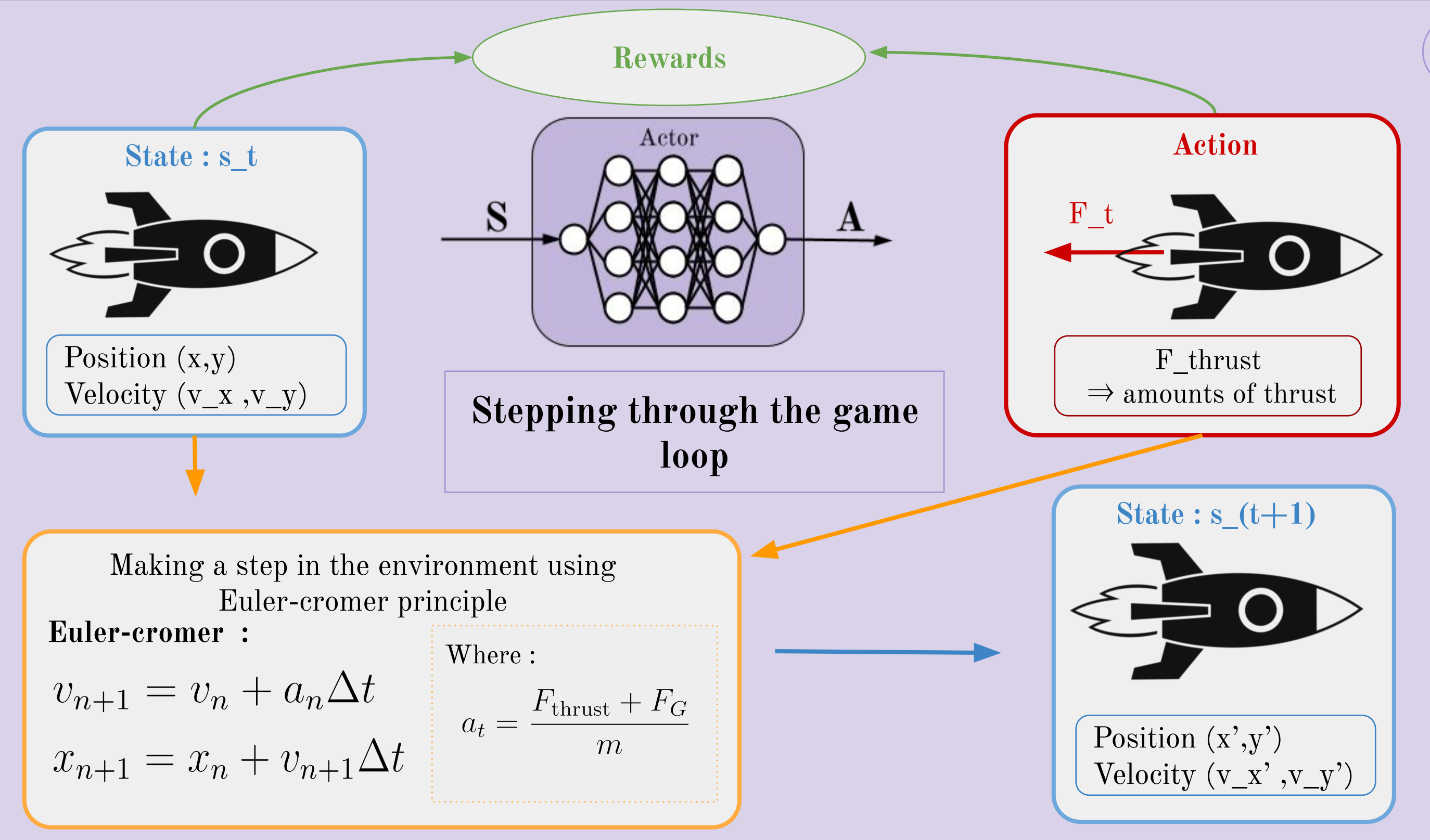
A reminder :

$$\mathbf{r} = r\hat{r}$$
$$\mathbf{v} = \dot{r}\hat{r} + r\dot{\theta}\hat{\theta}$$
$$\mathbf{a} = (\ddot{r} - r\dot{\theta}^2)\hat{r} + (r\ddot{\theta} + 2\dot{r}\dot{\theta})\hat{\theta}$$

Letting us find L :

$$L = r\hat{t} \times m(v_r\hat{r} + v_t\hat{\theta}) = rmv_\theta\hat{z} = mrv_t$$

2



$$v_{n+1} = v_n + a_n\Delta t$$
$$x_{n+1} = x_n + v_{n+1}\Delta t$$
$$a_t = \frac{F_{\text{thrust}} + F_G}{m}$$

The reward function

3

Overview : in Reinforcement Learning, the choice of reward function is key. To train our agent to achieve a optimal outcomes we give feedback in the form of rewards based on the action it takes in the following way.

First function h(*s*)

The target state that we are trying to reach can be described by :

- Radius = **1**
- L/m = **1**
- Thrust = **0**

We are looking for a convergence to this target state which leads to our chosen reward function:

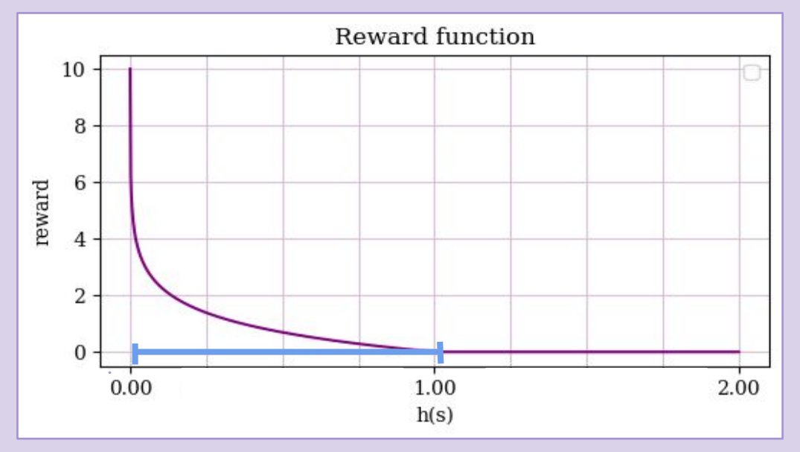
$$h(r,v_t,a) = (r-1)^2 + (vr-1)^2 + (a-0)^2$$

At convergence, h(s) = 0

Maximizing the reward r when h(*s*) = 0

In order to do so, we will wrap h(*s*) in a -ln() function as follows :

$$r = \begin{cases} \text{if } -\ln(h(s)) < 0 : r = 0 \\ \text{if } -\ln(h(s)) > 10 : r = 10 \\ \text{else: } r = -\ln(h(s)) \end{cases}$$



Weighting h(*s*)

In order for h(*s*) to be significant, *s* needs to be in the range [0, 1], we assign a weight each error term.

$$h(r,v_t,a) = w_r(r-1)^2 + w_l(vr-1)^2 + w\_t(a-0)^2$$

Varying weights for the areas of error allows us to explicitly adjust the relative importance of each variable.

Understanding PPO

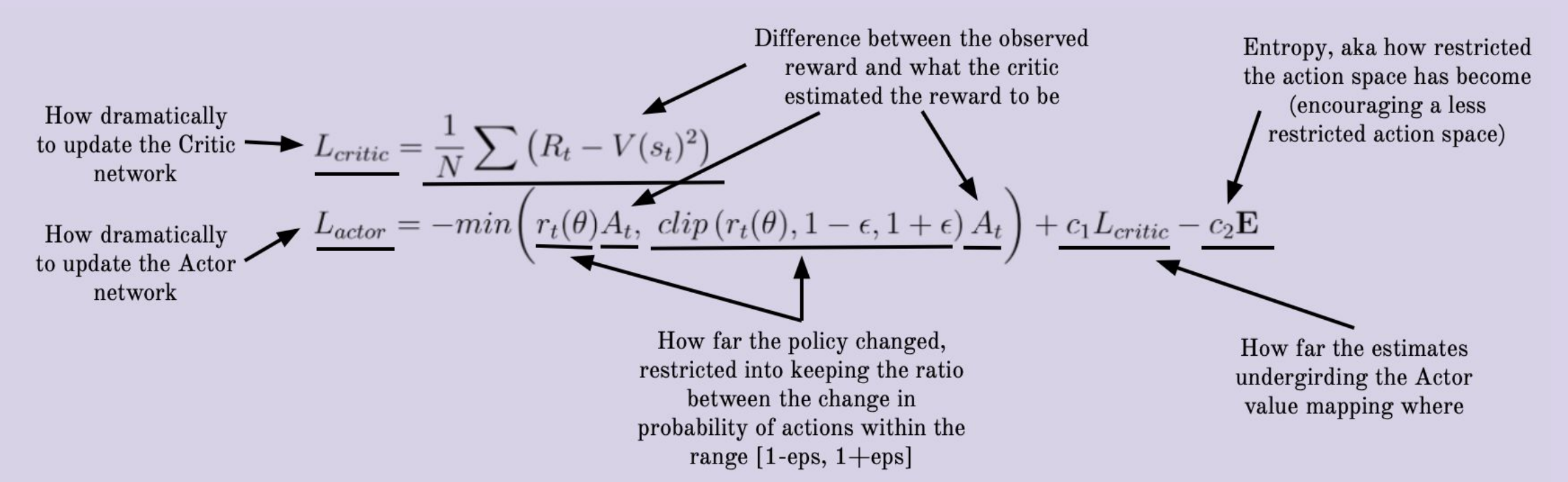
Reinforcement learning (RL) studies how to formulate a self-guiding agent to take actions in a dynamic environment for maximum performance. The challenge becomes (as always) tailoring the model to best suit this particular task. Most RL approaches focus on either:

A. Estimating the *value* of the next world states your agent can take actions to arrive at, or,
B. Directly developing a *policy* for how to, based on the world state your model is viewing, navigate.

In our case we understood our environment had relatively low degrees of state-space complexity and is even, in some ways, a solved problem. We spent a few weeks looking into existing Control Systems such as *PID*s and the Hohmann transfer to inform our understanding but understood our approach would need to scale to more complicated *N*-body state spaces. For these reasons we chose to implement and tinker with PPO models which blend together both estimating state values and a network to learn a policy informed by that map.

PPO models engage two parallel Neural Networks: the first network a *Critic* model tasked with estimating the value of any given state, the second a *Actor* used to generate which action to take from the given state. Updating the *Actor* and *Critic* networks entails a large amount of lives played utilizing the current networks after which that whole experience is looked through to calculate for the many states observed how far the *Critic*'s estimate of state values was from the actual state values in order to update the Critic and which actions over or underperformed, minding to not update the probability of taking an action too dramatically, in order to update the Actor).

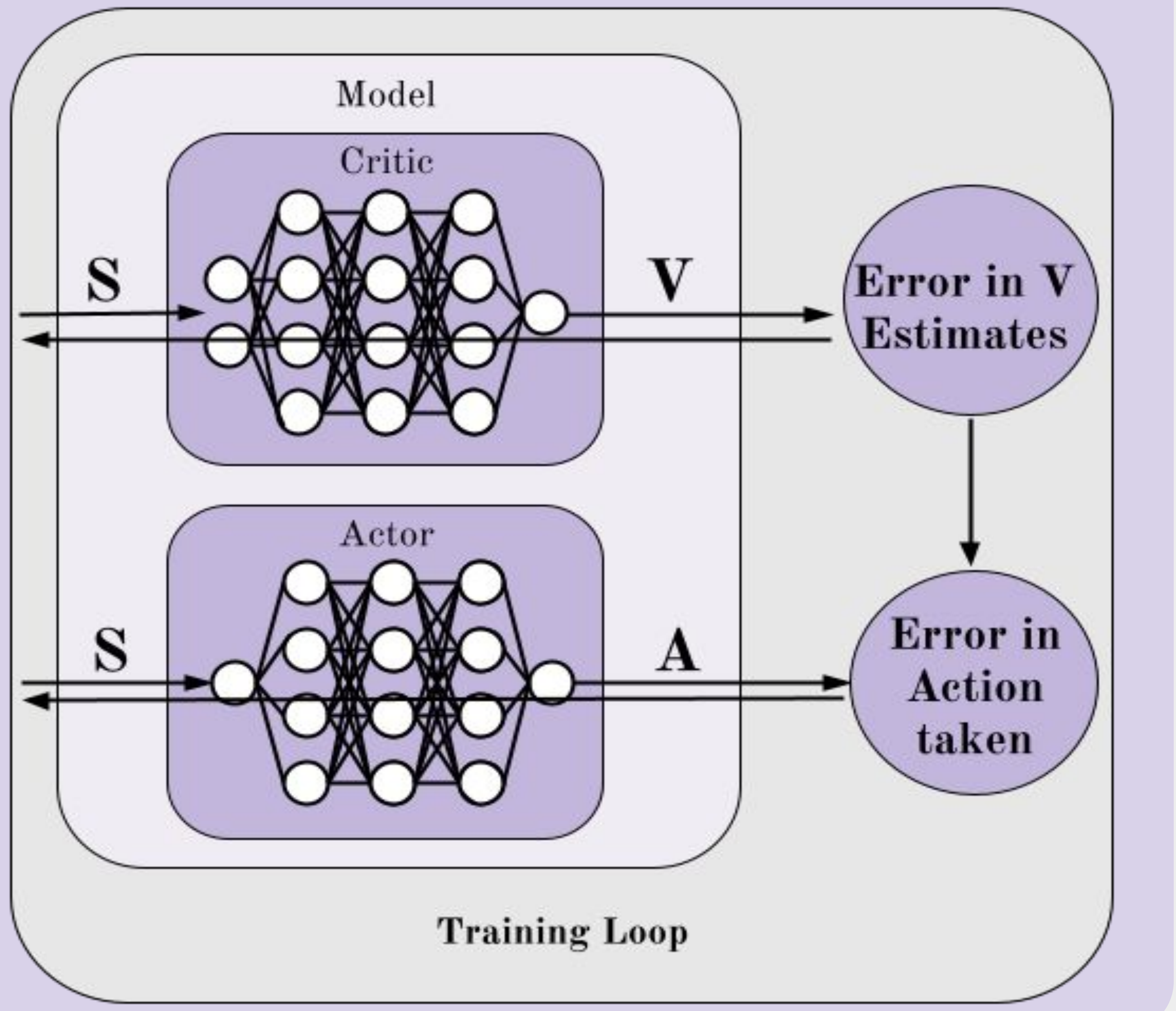
$$L_{critic} = \frac{1}{N}\sum (R_t - V(s_t)^2)$$
$$L_{actor} = -min\left(r_t(\theta)A_t,\ clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)A_t\right) + c_1 L_{critic} - c_2\mathbf{E}$$


How dramatically to update the Critic network
How dramatically to update the Actor network
Difference between the observed reward and what the critic estimated the reward to be
Entropy, aka how restricted the action space has become (encouraging a less restricted action space)
How far the policy changed, restricted into keeping the ratio between the change in probability of actions within the range [1-eps, 1+eps]
How far the estimates undergirding the Actor value mapping where

Hyperparameters

```
gamma = 0.9999
lr_actor = 1e-3
lr_critic = 2e-3
eps_clip = 0.2
min_action_std = 0.1
max_ep_len = 25_000
```

That core *Actor-Critic* loop of generating estimates for the value of states, using that map of states to values for informing a policy, then playing through many lives and timesteps to measure how far wrong both networks were altogether composed the backbone of the PPO. Our particular *Actor* and *Critic* model's were a rather standard 3 layers of 12 neurons linked by `nn.LeakyReLU()`and `nn.Tanh()`activations respectively. We did tease some benefit with a custom activation function to take in the usual `nn.Tanh()` output of the *Actor* and feed it into a f(x)=x^7 function so as to squish any output from the final linear layer within the x domain of roughly -1 to 1 into an action value of 0 and give the actor theoretically more area of output over which to pick the 0 actions implicit to the sparse-action Hohmann solution policy.



Results

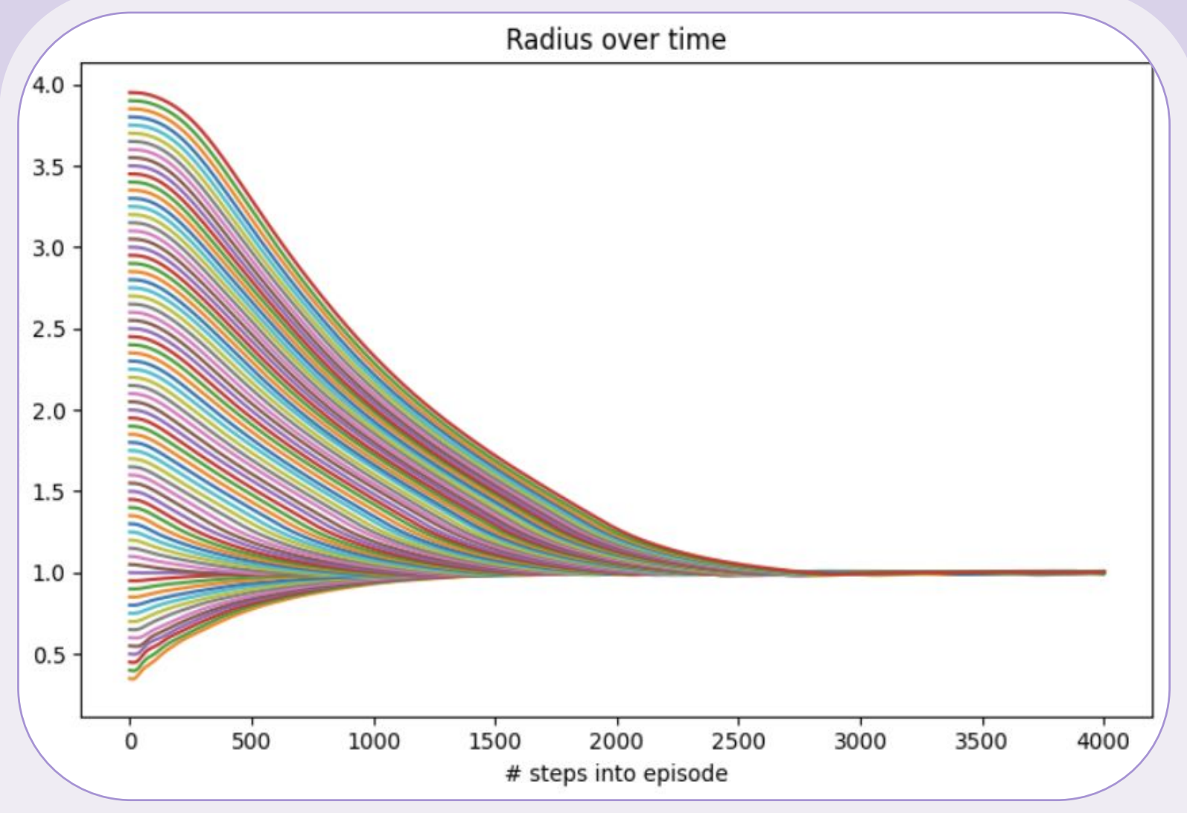
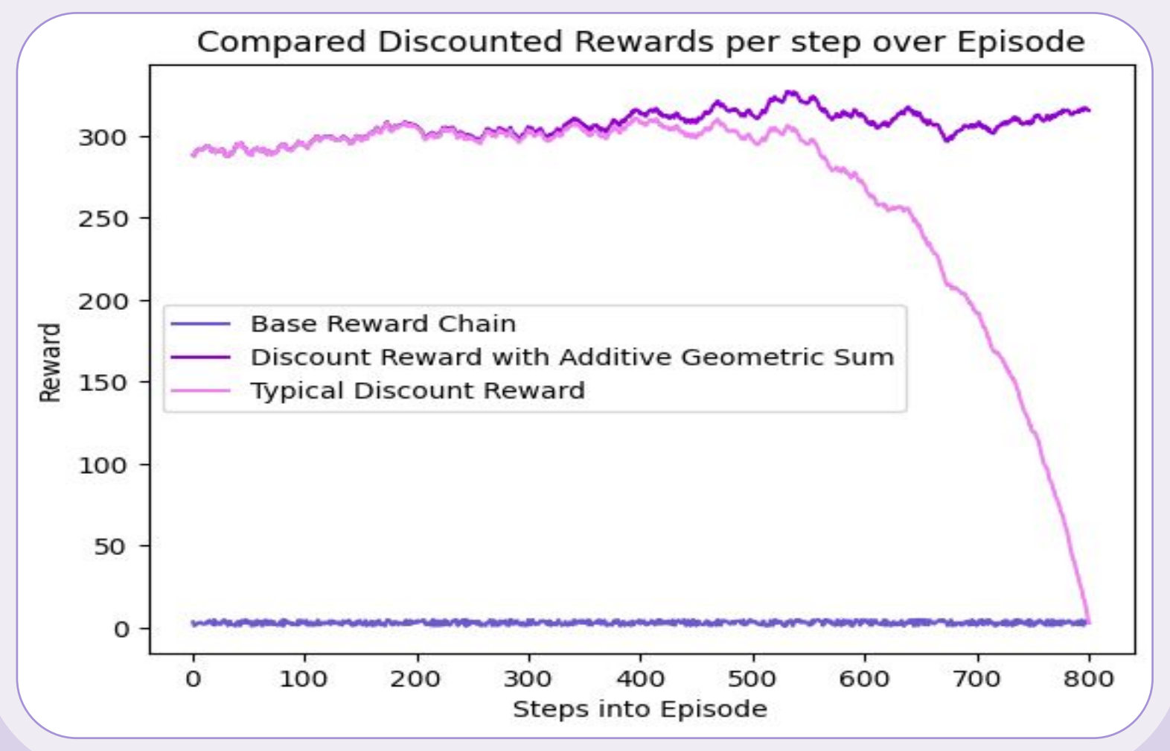


Fig 1 : Consistent navigation was observed towards r_target = 1.0 from radii across the range [0.3, 4.0] showing applicability of the PPO format, an important takeaway given the nonlinear relationships between velocity and radius. We didn't observe strong generalization outside of the trained on radial region, implying overfitting and highlighting the nonlinear relationships involved.

Fig 2 : We corrected issues with quantifying rewards late into each trajectory which indicated huge amounts of future rewards post reaching the timestep cutoff of each episode. We solved it by adding the geometric sum of projecting forward 800 steps to the later states.Another change was the model the state variables being sqrt(r), rv, and 1/sqrt(r) to compensate for some of the nonlinear relationships and help with the learnability of the space.



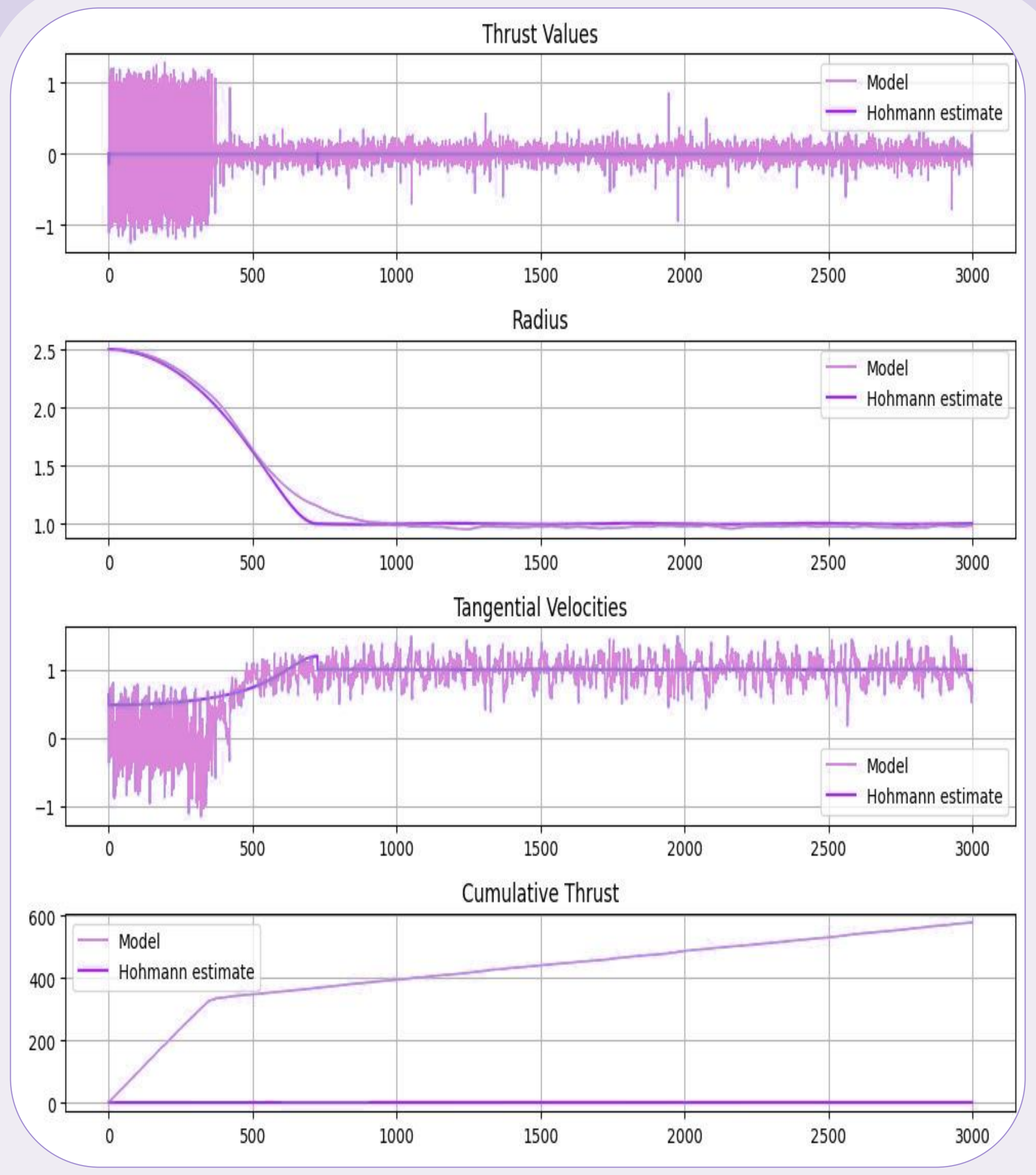


Fig 3 : We observed our model holding to paths of remarkably similar to the paths we would expected from the analytic Hohmann transfer solution, indicating a fascinating and satisfying rediscovery of the analytic solution via the PPO model. The primary area where our model differed from the Hohmann transfer was in it's heavy propensity to oversteer causing our spacecraft to burn through far more fuel (up to 1000x) the mathematically necessary amount.

Discussion

Challenges faced

While the transfer of orbit has been achieved, our observations indicate a suboptimal performance in thrust optimization. The final slope characterizing the cumulative thrust suggests a minimal yet continuous compensation at each step, even though the stable orbit characteristics are supposedly attained. Furthermore, the initial steps of the episodes exhibit overthrusting.

Insights

Continuous Compensation: Because of the Euler-Cromer discretization of our environment, we observe small oscillations in the orbit radius even under the ideal Hohmann transfer, a inherent error which our current reward function does not account for and tries in vain to trim out. Additionally, the penalty for thrust is linked to the impulse at a given instant *t*, making these small impulses nearly negligible due to the term associated with their penalty.

Overthrusting: The initial weighting of the reward function was designed for smaller radius transfers. For radii sufficiently far from our target however the reward function consistently yields a value of h(s)=0 regardless of action taken and cannibalizes the ability to learn to distinguish good actions from bad ones at extreme radii.

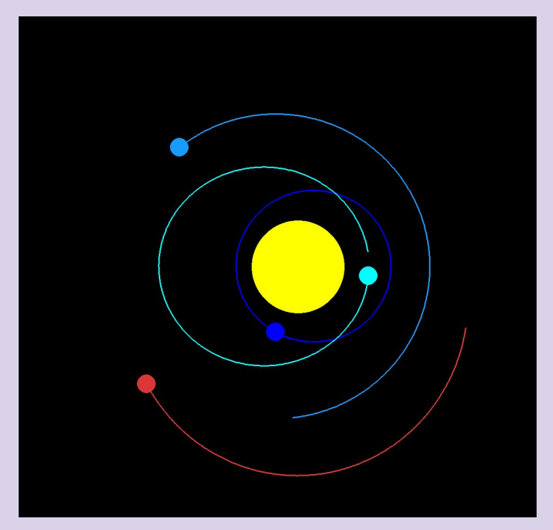
Going Forward

Immediate Next Steps

In the short term, we aim to optimize the thrust by refining its term in the reward function and carefully reevaluating the weighting of each term to ensure significant and appropriate scaling.

Future direction

Moving toward an *n*-body problem and reshape the problem to enhance its educational value



Acknowledgements

We would be remiss not to mention some of the incredible talented people who we've worked with and built on the work of to make this work possible:

1. Yizhong Hu '24 wrote most of the codebase for running the gravitational gamespace
2. Prof. Richard Gaitskell, Shawn Dubey, and Austin Vaitkus worked with us weekly to guide our work, set straight the direction of our work, and inquire into clues in our results which could help iterate.
3. Our PPO implementation is drawn from Nikhil Barhate's "PPO-Pytorch" repo on GitHub