12-2019

A Reinforcement Learning Approach to Spacecraft Trajectory Optimization

Daniel S. Kolosa
Western Michigan University, danieljr1@aol.com

A REINFORCEMENT LEARNING APPROACH TO SPACECRAFT TRAJECTORY OPTIMIZATION

by

Daniel S. Kolosa

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Mechanical and Aerospace Engineering
Western Michigan University
December 2019

Doctoral Committee:

Jennifer S. Hudson, Ph.D., Chair
Richard M. Meyer, Ph.D.
Kapsong Ro, Ph.D.
Robert G. Trenary, Ph.D.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Professor Jennifer Hudson, for her patience, guidance, and encouragement throughout my graduate career. I would also like to thank Professor Richard Meyer and Professor Robert Trenary for their invaluable support and advice. I also want to thank Professor Kapsong Ro for being a part of my education since my undergraduate career.

I would like to thank the Mechanical and Aerospace engineering department for providing assistance and support throughout my graduate career. I am grateful to the Space Flight Dynamics and Control Laboratory as well as Justin Rittenhouse and Chris Proctor for their insight, company, and comradery.

I would like to thank my parents and siblings for their continued support and encouragement to pursue a graduate career. Finally, I want to sincerely thank my wife Yihan, for encouraging me to be persistent and keeping me grounded in moments when it is darkest before dawn.

Daniel S. Kolosa

A REINFORCEMENT LEARNING APPROACH TO SPACECRAFT TRAJECTORY OPTIMIZATION

Daniel S. Kolosa, Ph.D.

Western Michigan University, 2019

This dissertation explores a novel method of solving low-thrust spacecraft targeting problems using reinforcement learning. A reinforcement learning algorithm based on Deep Deterministic Policy Gradients was developed to solve low-thrust trajectory optimization problems. The algorithm consists of two neural networks, an actor network and a critic network. The actor approximates a thrust magnitude given the current spacecraft state expressed as a set of orbital elements. The critic network evaluates the action taken by the actor based on the state and action taken. Three different types of trajectory problems were solved, a generalized orbit change maneuver, a semimajor axis change maneuver, and an inclination change maneuver. When training the algorithm in a simulated space environment, it was able to solve both the generalized orbit change and semimajor axis change maneuvers with no prior knowledge of the environment's dynamics. The robustness of the algorithm was tested on an inclination change maneuver with a randomized set of initial states. After training, the algorithm was able to successfully generalize and solve new inclination changes that it has not seen before.

This method has potential future applications in developing more complex low-thrust maneuvers or real-time autonomous spaceflight control.

TABLE OF CONTENTS

Table Of Contents–Continued

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

α_m	prioritized memory hyperparameter
α	learning rate
α_a	reward value weight for semimajor axis
α_{e_x}	reward value weight for first component of equinoctial eccentricity
α_{e_y}	reward value weight for second component of equinoctial eccentricity
α_{h_x}	reward value weight for second component of equinoctial inclination
β	prioritized memory selection hyperparameter
δ	TD-error
ε	priority replay value weight
γ	discount factor
μ_E	Gravitational constant of a central body
μ	action
μ^l	layer mean
Ω	Right ascension of the ascending node
ω	perigee
π	policy
σ^l	layer standard deviation
τ	target network soft update
$\bar{\theta}$	normalized angle
$\dot{\theta}$	angular velocity
θ_t^μ	parameter policy
θ	neural network weight parameter
θ_L	Mean Longitude
A	action value
a	semimajor axis
a_i	action at a sampled state
a_t	action at a given time
E	Eccentric Anomaly
E_π	Environment given a policy
e	eccentricity
e_t	experience memory
e_x	first component of equinoctial eccentricity
e_y	second component of equinoctial eccentricity
F_R	Thrust in the radial direction
F_S	Thrust in the circumferential direction
F_t	Thrust acceleration

F_W	Thrust acceleration in the normal direction
G_t	Return following time t
h	angular momentum
h_x	first component of equinoctial inclination
h_y	second component of equinoctial inclination
I_{sp}	specific Impulse
i	inclination
J	cost function
K	normal direction unit vector
L	Loss function
l	length
M	Mean Anomaly
m	mass
m_f	fuel mass
N	Node line unit vector
N_b	batch size
N_t	action noise
N_d	decay rate
p_t	priority value
Q	Q-value
q_π	q-value from a policy
$\mathbf{r}$	position vector
R	total reward
r	Reward
r_a	reward value semimajor axis
r_{ex}	reward value first component of equinoctial eccentricity
r_{ey}	reward value second component of equinoctial eccentricity
r_{hx}	reward value third component of equinoctial inclination
r_{hy}	reward value fourth component of equinoctial inclination
S_t	state at a given time
s	state
T	Torque
t	time
V	value function
$\mathbf{v}$	velocity vector
w_i	neural network weight
$\mathbf{x}$	state vector
y_i	expected reward

CHAPTER 1

INTRODUCTION

Reinforcement learning has recently become an exciting topic in machine learning. With recent advancements in computer hardware and architecture, it has become feasible to train reinforcement learning algorithms in a reasonable amount of time. Reinforcement learning algorithms are being used in the development of self-driving cars, robotics, finance, medicine, games, and mechanical systems.

This dissertation proposes the use of a reinforcement learning algorithm to solve low-thrust spacecraft targeting problems. The objectives of this research are as follows; apply a reinforcement learning algorithm as a method to determine the piecewise continuous thrust of a spacecraft to reach a desired target orbit using only the current state of the spacecraft at a given time, generalize the algorithm to be able to solve different orbital targeting problems.

The methodological goal of this research is to approach spacecraft trajectory control and optimization through a novel perspective. Low-thrust trajectory dynamics are nonlinear and solving these types of trajectories often requires optimization methods. The novel approach presented here uses a set of neural networks to control the thrust vector of a spacecraft at a given time. Neural networks are known as nonlinear function approximators, that only require an input and generate a representation of a nonlinear function. Without training and tuning, a neural network will not sufficiently be able to generate a meaningful output. This research uses model-free reinforcement learning as the optimization model. Model-free reinforcement learning algorithms do not require domain knowledge of the application. This approach allows for the representation of complex dynamics without having to compromise on fidelity by linearization. Linearization of the dynamics are limited by the perspective of the designer whereas reinforcement learning can generate and simulate complex control laws in a relatively short amount of time.

The method presented here is formulated as a Markov Decision Process. This process has an agent (satellite), placed in an environment, given a set of goals, and propagated through the environment according to a policy that attempts to maximize the reward obtained from the environment. The policy is a function based on a form of the Bellman equations, which attempts to maximize the reward or quality Q, given a current state and possible action. The quality function Q, can be represented as a non-linear function in terms of the state-action pair and can be estimated using a neural network. The neural network based policy is analogous to a non-linear thrust control law. Using an iterative method of exploring the environment, the spacecraft can train the neural network to find an optimal control law to allow the spacecraft to reach a target state.

There are many possible applications of this research within the domain of orbital trajectory optimization. In a problem with a known solution, this method can allow for further optimization of fuel usage by allowing a more complex thrust controller. The research presented here can also be applied to real-time autonomous spacecraft navigation. Deep space missions with large time delays or interference in communications could use this method to allow a spcecraft to safely navigate. Also this algorithm can be used for satellites to re-route a trajectory for obstacle avoidance like space debris.

1.1 Contributions

The primary contributions of this dissertation are:

- A generalized machine learning framework for solving low-thrust spacecraft trajectory optimization problems.
- An algorithm that can be applied flexibly to any two-body space dynamics system regardless of the complexity of the trajectory problem.
- An alogrithm to tune neural networks for common orbital targeting problems that can be used for preliminary low-thrust mission design.

CHAPTER 2

LITERATURE REVIEW

The following sections discuss different approaches used to solving and modeling low-thrust trajectory problems. The second section discusses machine learning methods for low-thrust trajectory design. The final section discusses the use of reinforcement learning to solve complex control problems.

2.1 Low-thrust Trajectory Optimization

For an optimization model, good initial guesses are more important for one type of optimization problem than another to reach a good approximation. Two major approaches to solving an optimization problem are by taking a direct or an indirect approach. A direct approach solves a nonlinear programming problem using a penalty function or augmented Lagrangian functions to find the control parameters. An indirect approach finds an approximate solution using an associated two-point boundary value problem. To obtain a good approximate solution, many people have developed methods that generate good initial guesses.

For low-thrust trajectory optimization and control, many analytical and numerical approaches exist. Methods like [1] use the sinusoidal logarithmic shape to represent the trajectory. Fang, Wang, Sun, and Yuan [2], use a finite Fourier series to approximate a low-thrust trajectory. Euler [3], solves a power-limited low-thrust rendezvous problem using polar coordinates. Ocampo [4] discusses the tradeoffs between scope and depth of designing a software system for spacecraft trajectory optimization. He then designs two missions, one is an lunar-free trajectory implusive maneuver, and the second mission is a low-thrust maneuver to place a spacecraft in a polar lunar orbit. Using evolutionary algorithms is another popular approach to solving low-thrust optimization problems. Zeng, Geng, and Wu [5], developed a shape-based method that can be applied to both transfer and rendezvous problems. This method does not assume a fixed shape for the trajectory and optimizes for minimum fuel consumption. Their approach may reduce computation time by avoiding numerically integrating the motion equations. Vasile [6], implemented an evolutionary and branching algorithm to solve two-body low-thrust optimization problems. Zhuang and Huang use a combination of particle swarm optimization and Legendre pseudospectral method to find a time-optimal low-thrust trajectory. The method switches between the two algorithms to mitigate their individual disadvantages to find better global optimum solutions than using the algorithms separately.

2.2 Machine Learning Methods for Spacecraft Trajectory Analysis

Work done by Yang, Xu, and Zhang[7], modeled low-thrust spiral trajectories of an electric propulsion system using Lyapunov-based guidance and an artificial neural network (ANN) to implement control gains. The ANN used an evolutionary algorithm for learning and training. The dynamic model used the orthogonal radial-horizontal frame (RSW) as the coordinate system. Lyapunov-based guidance was used to model minimum-time and time-fixed minimum-propellant low-thrust orbit transfers. For both minimum-time and time-fixed minimum-propellant models, the satellite was within the threshold, concluding that this model can be used as an autonomous guidance scheme.

Ampatzis and Izzo [8], integrated a neural network in an evolutionary algorithm to approximate the objective function of a low-thrust trajectory. The neural network was adaptively trained by employing the original objective function and collecting input/output data. The neural network was created using feed-forward networks and trained using the Levenberg Marquardt algorithm. The model was then benchmarked using a Multiple Gravity Assist (MGA) Problem, modeled after the Cassini 1 trajectory to Saturn. The objective function was the total deltaV accumulated during the mission. The second benchmark was modeled after the Cassini 2 mission. For this mission, the target planet was Saturn and the planetary flyby sequence was Earth-Venus-Venus-Earth-Jupiter-Saturn and deep space maneuvers(DSM) were allowed in between each one of the planets. The third benchmark used a MGA-DSM mission to Mercury based on the Messenger mission. The results showed that an objective function using an ANN gave accurate results.

Machine learning techniques were employed by [8], to optimize fuel usage of a spiral low-thrust optimization problem. This research uses a simple optimal control problem to initially calculate the fuel mass of an orbit transfer then further optimizes the result with a regression algorithm. The method presented in this research does not implement an optimal control approach. The reinforcement learning approach is a model-free approach which does not require any domain knowledge of the problem to find an optimal solution. By not being restricted by the domain knowledge of the problem, it is possible to generalize the algorithm to solve trajectory problem where the domain may be very complex.

Izzo, Sprague, and Tailor[9], implement a similar approach where a simplified optimal trajectory optimization problem is solved first then optimized using machine learning. Their method uses a deep neural network represent the optimal guidance profile of an interplanetary mission. Th test case implemented was a interplanetary trajectory problem from Earth to Mars. The neural network was trained using a dataset of optimal trajectory problems that were solved as two-point boundary value problems.

2.3 Reinforcement Learning Approaches to Complex System Dynamics

In the realm of control problems, reinforcement learning can be used to control complex systems. Kumar, Paul, and Omkar designed a bipedal robot and used reinforcement learning train it in a simulated environment to successfully walk 10 meters without falling[10]. Lingli, Xuanya, yadong, and Kaijun used a deep deterministic policy gradient algorithm and a simulated environment to determine the optimal driving maneuvers of a land-vehicle [11]. Their trained model is then transferred to another virtual environment calculated a sequence of trajectories. The trajectories are evaluated and the optimal is selected.

CHAPTER 3

ORBITAL MECHANICS REVIEW

In a restricted two-body problem, a small object with negligible mass orbits a central body with no other bodies affecting the system. The dynamics of the restricted two body problem can be described by Newton's equation of planetary motion,

$$\ddot{\mathbf{r}} = \frac{\mu_E}{|\mathbf{r}|^3}\mathbf{r} \tag{3.1}$$

where $\mathbf{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$, is the position and μ_E is the standard gravitational constant of the central body. In spacecraft trajectory analysis, the small secondary object is a spacecraft, which many have an onboard propulsion system. In the case of general continuous thrust, thrust is modeled as a continuous vector that changes its magnitude and direction over time. The dynamic model of a continuous thrust maneuver can be modeled as,

$$\dot{\mathbf{r}} = \frac{\mu_E}{|\mathbf{r}|^3}\mathbf{r} + F_t \tag{3.2}$$

where F_t is the thrust acceleration. Over a long duration thrust arc, a small change in the magnitude or direction of the thrust may result in a large change of the trajectory.

3.1 Orbital Elements

In orbital mechanics, several different coordinate systems are used. One method to express the position and velocity of an orbiting object with respect to a central body is by using Cartesian

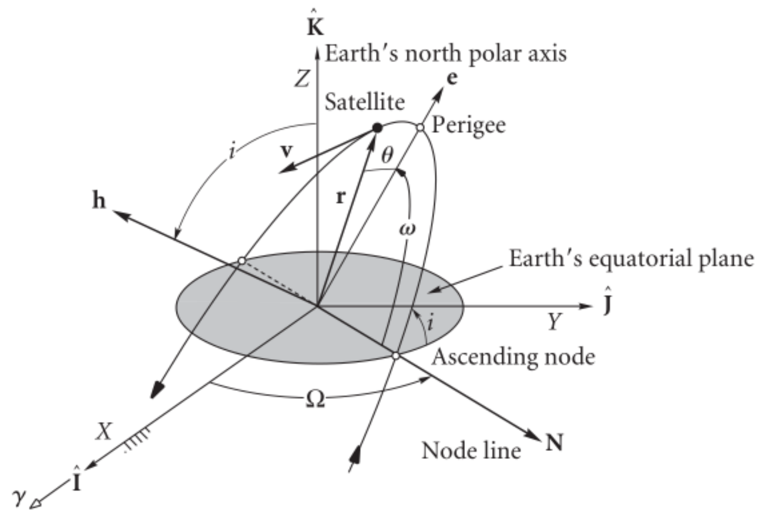

Figure 3.1: Keplerian Orbital Elements [12]

vectors defined in Equations 3.3-3.4.

$$\mathbf{r} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix}^T \tag{3.3}$$

$$\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T \tag{3.4}$$

The state vector at any point in time can be obtained by solving the two-body differential equations for Newton's equation of motion Equation 3.1.

Another coordinate system commonly used is the Keplerian orbital elements. These orbital elements describe the shape, size, and orientation of the orbit using six elements as shown in Figure 3.1.

The semi-major axis (a), is the average of the periapsis and apoapsis radii. The eccentricity (e), describes the shape of the orbit, with an eccentricity of zero being a circular orbit and as the eccentricity approaches 1 the orbit becomes more elliptical. Beyond an eccentricity of one, the orbit is then hyperbolic. The inclination (i) is the vertical tilt of the orbit. The longitude of the ascending node (Ω) is the intersection point between the Earth's equatorial plane and the orbit plane. The argument of perigee (ω) is the angle from the node line to the eccentricity vector in the orbital plane. The node line is a line where the orbit intersects a point of reference where the orbit is inclined. The true anomaly θ, is the angle of the position of a body measured in the plane of the orbit from periapsis. A state vector (r, v) described in Cartesian coordinates can be described equivalently by the set of six orbital elements $(a, e, i, \Omega, \omega, \theta)$. Appendix B details the transformation from Cartesian to Keplerian orbital elements. Continuous thrust maneuvers can be represented in the RWS coordinate frame, where R is along the radial direction of the orbit, W is the circumferential direction, and S along the angular momentum vector. The non-linear dynamics of low-thrust maneuvers can be expressed using Gauss's Variational equations [13] (3.5 - 3.10).

$$\frac{da}{dt} = 2\sqrt{\frac{a}{\mu_E}}[F_R \frac{ae}{\sqrt{1-e^2}} \sin\theta + F_S \frac{a^2\sqrt{1-e^2}}{a(1-e\cos E}] \tag{3.5}$$

$$\frac{de}{dt} = \frac{h}{\mu_E} \sin\theta F_R + \frac{1}{\mu_E h}[(h^2 + \mu_E r)\cos\theta + \mu_E r]F_S \tag{3.6}$$

$$\frac{di}{dt} = \frac{r}{h}\cos(\omega + \theta)F_W \tag{3.7}$$

$$\frac{d\Omega}{dt} = \frac{r}{h\sin i}\sin(\omega + \theta)F_W \tag{3.8}$$

$$\frac{d\omega}{dt} = -\frac{1}{eh}[\frac{h^2}{\mu_E}\cos\theta F_R - (r + \frac{h^2}{\mu_E})\sin\theta F_S] - \frac{r\sin(\omega + \theta)}{h\tan i}F_W \tag{3.9}$$

$$\frac{d\theta}{dt} = \frac{h}{r^2} + \frac{1}{eh}[\frac{h^2}{\mu_E}\cos\theta F_r - (r + \frac{h^2}{\mu_E})\sin\theta F_s] \tag{3.10}$$

Where F_R, F_W, F_S are the thrust acceleration, in the $\hat{R}, \hat{W}, \hat{S}$ directions. One disadvantage of using Keplerian orbital elements are singularities at zero inclination or eccentricity for Equation 3.8 and Equation 3.9. By transforming Keplerian orbital elements to equinoctial elements shown in Equations (3.11 - 3.15), singularities can be avoided.

$$a = a \tag{3.11}$$

$$e_x = e\cos(\Omega + \omega) \tag{3.12}$$

$$e_y = e\sin(\Omega + \omega) \tag{3.13}$$

$$h_x = \tan(\frac{i}{2})\cos(\Omega) \tag{3.14}$$

$$h_y = \tan(\frac{i}{2})\sin(\Omega) \tag{3.15}$$

Equations (3.5 - 3.10) can also be expressed in equinoctial form [14]

$$\frac{da}{dt} = \frac{2a^2}{h}\{(e_x \sin\theta_L - e_y \cos\theta_L)F_R + fracprF_s\} \tag{3.16}$$

$$\frac{de_x}{dt} = \frac{r}{h}[-\frac{p}{r}\cos\theta_L F_R + \{e_y + (1+\frac{p}{r})\sin\theta_L\}F_s - e_x(h_y\cos\theta_L - h_x\sin\theta_L)F_w] \tag{3.17}$$

$$\frac{de_y}{dt} = \frac{r}{h}[-\frac{p}{r}\sin\theta_L F_R + \{e_y + (1+\frac{p}{r})\cos\theta_L\}F_s + h_y(h_y\sin\theta_L - h_x\cos\theta_L)F_w] \tag{3.18}$$

$$\frac{dh_x}{dt} = \frac{r}{2h}(1+h_y^2+h_x^2)\sin\theta_L F_w \tag{3.19}$$

$$\frac{dh_y}{dt} = \frac{r}{2h}(1+h_y^2+h_x^2)\cos\theta_L F_w \tag{3.20}$$

3.2 Orbital Trajectory Problem

Two types of low-thrust orbit maneuvers will be discussed: the spiral out maneuver, the plane change maneuver. A spiral out maneuver, is defined as an orbit maneuver where the semi-major axis changes while keeping the eccentricity and inclination constant. Figure 3.2, shows an example of spiral out maneuver.

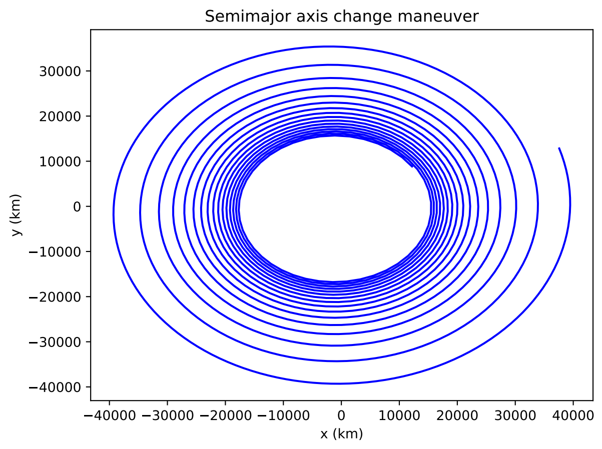

Figure 3.2: Spiral Out Maneuver

A plane change maneuver is where only the inclination changes, while the other orbit elements are held constant. Figure 3.3, shows an example of a plane change maneuver.

The low-thrust spacecraft targeting problem is formulated as follows: Given an initial

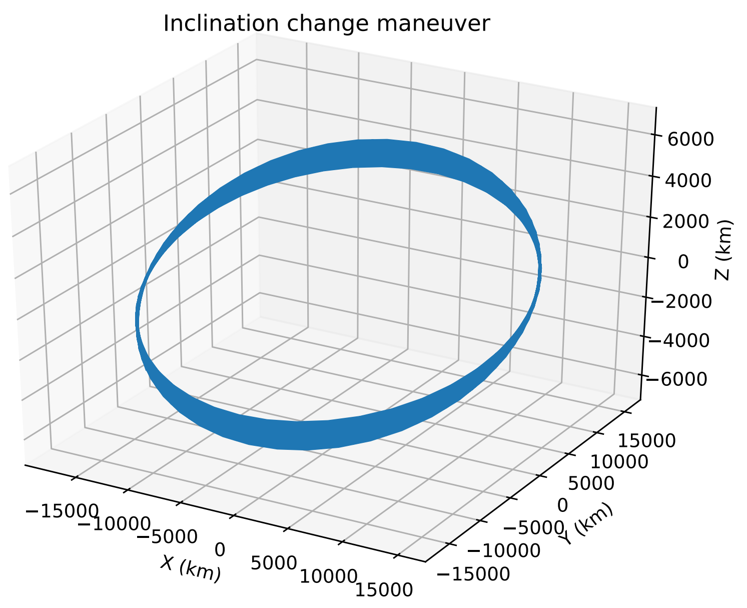

Figure 3.3: An Inclination Change Maneuver

spacecraft state

$$\mathbf{x}_o = \begin{Bmatrix} \mathbf{r}(t_o) \\ \dot{\mathbf{r}}(t_o) \end{Bmatrix}$$

A target state

$$\mathbf{x}_f = \begin{Bmatrix} \mathbf{r}(t_f) \\ \dot{\mathbf{r}}(t_f) \end{Bmatrix}$$

and a transfer time

$$t = t_f - t_o$$

find the piecewise-continuous thrust acceleration control

$$\mathbf{F}_t = \begin{Bmatrix} F_{R1} & F_{w1} & F_{s1} \\ F_{R2} & F_{w2} & F_{S2} \\ \vdots & \vdots & \vdots \\ F_{Rn} & F_{wn} & F_{Sn} \end{Bmatrix}$$

such that $\overrightarrow{x}(t_f) = \overrightarrow{x}_f$, where the trajectory dynamics are given by Equations 3.5 - 3.10 where $i = 1, \ldots n$, and $\frac{t}{n}$ is the time duration of each time interval.

CHAPTER 4

REINFORCEMENT LEARNING

In machine learning, three major types of learning strategies exist; supervised, unsupervised, and reinforcement learning. For supervised learning, a model is trained given a set of inputs and expected responses. Then the model is evaluated by only giving the model a set of inputs and comparing the response from the model with the true response. For unsupervised learning, a data set for a model is given where the response is not known. The model then uses pattern matching or other techniques to find associations between the features. Both supervised and unsupervised learning algorithms are trained on large sets of data. Reinforcement learning differs from the previous two learning strategies because the learning algorithm is trained in an environment. The goal of a reinforcement learning algorithm is to maximize the final reward in a given environment. Reinforcement learning is more closely related to unsupervised learning than supervised learning because the target value of every possible state is not known.

4.1 Markov Decision Process

The Markov Decision Process (MDP) is a framework used to define reinforcement learning problems. In reinforcement learning, an agent is placed inside an environment with the goal of maximizing a reward value. The agent interacts with the environment by performing an action determined by a policy. A policy is a general term used to describe a function that an agent follows to take an action in an environment. A reward is then given by the environment to the agent when particular state is attained. A reward value is a number that indicates to an agent how good an action is in a good stae transition. The Markov Decision Process is detailed in Figure 4.1.

The state describes the current situation that an agent is in at a particular point of time. For this project the state is defined as the satellite's current position and velocity expressed as

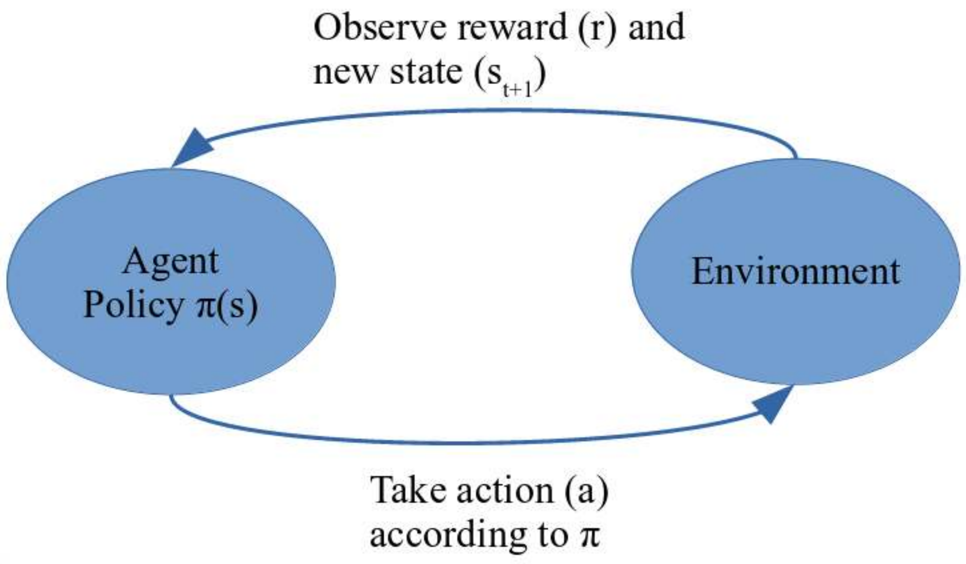
Observe reward (r) and new state (s_{t+1})
Agent Policy π(s)
Environment
Take action (a) according to π

Figure 4.1: Markov Decision Process

equinoctial orbital elements. The action is the possible operation that an agent can perform a particular point in time to transition from one state to another. The actions that can be taken by the satellite are a set of discrete throttle points in the radial, tangential, and normal directions. To determine the effect of an action at a particular state, a value function is used to estimate the reward of a state-action pair.

The value is defined as the expected return in a state s when following a policy π. The value function is derived from the Bellman equation, which is taken from dynamic programming expressing the total reward given by taking an action at a particular state. In terms of a MDP, the value function can be defined as[15]

$$v_\pi(s) = E_\pi\left[G_t | S_t = s\right] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \tag{4.1}$$

where the policy π, defines which action the agent will take at a given state. E_π is the expected value of a random variable when an agent follows a particular policy π, R is the reward value at a state at time t, and γ is a discount factor whose value is between zero and 1. The discount factor is an indication of how much influence future rewards have on the value function. For the spacecraft, the value function is a function of the difference between the current and target spacecraft state.

Value functions can be represented by a linear or non-linear function.

When an agent reaches a particular termination state, the current episode terminates. The agent replays multiple episodes until a reward value, maximum number of episodes, or a particular final state is obtained. With RL, the agent does not have to be aware of the dynamics of the environment to reach its goal, making RL algorithms capable of solving problems too complex to model analytically.

4.2 Temporal Difference

Temporal difference (TD) is a method of reinforcement learning that predicts the value of a variable based on the values of an input signal at every time step. This method is a combination of both dynamic programming and Monte Carlo methods. Dynamic programming is a method for solving problems by breaking one problem up into multiple subproblems. With dynamic programming, the model of the dynamics of the environment must be known. Monte Carlo methods are algorithms that learn using experience and then update their value function at the end of each episode but they do not require a model of the environment to be known. Temporal difference uses a model-free base of learning. A model-free system can learn from experience and without a model of the dynamics of the environment. Unlike Monte Carlo methods, which update the value function at the end of every episode, TD methods update the value function at every time step, which may result in faster convergence. The value function for a TD model looking one step ahead, TD(0), is shown in the equation below:

$$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)] \tag{4.2}$$

where α is the learning rate, and γ is the discount factor that affects the impact of the reward value over time. $V(s)$ is the value of the current state and $V(S')$ is the value of the next state.

4.3 Q-Learning

Q-learning is an off policy TD control algorithm. An off-policy algorithm learns the optimal policy that is different from the policy that is used to generate data. Q-learning algorithms are used in discrete action-space environments. Instead of a value function, Q-learning uses expected reward values called Q values to predict the future reward. Where the value function is defined in terms of only the state of the agents, the Q-function is defined in terms of the state and action of the agent. The Q action-value function is defined as [15]:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{4.3}$$

The Q function is determined by using the current Q value, the reward of the next state and the maximum value of the difference between the Q-value of the current and next state-action pair. The Q-values are stored in a table for every possible state-action pair. The rows of the table are all the possible states and the columns are all the possible actions. The Q-learning algorithm is shown below[15].

```
Initialize Q;
for each episode do
    Initialize S;
    while S is not terminal do
        Choose an action given the current state using the policy $\pi$ derived from max
          Q;
        Take action A;
        Observe Reward R and S';
        $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1}\gamma max_a Q(S_{t+1},a) - Q(S_t,A_t)]$;
        S = $S_{t+1}$ ;
    end
end
```

Algorithm 1: Q-Learning

The first step of the algorithm initializes all q-table values with zero. The episode begins by initializing the state of the environment. The while loop is used to check for a terminal state. The terminal state can be a maximum number of iterations or a particular state. This prevents the agent from exploring a policy that steers too far from reaching the target state or takes too long to reach the target state. The next step is to select an action. The action selection process is based on the maximum Q-value at the given state. To prevent the algorithm from always taking a path that yields the same value, a random action can be taken based on a random chance percentage. The action is taken and the state and the reward for the next time step is updated. Then the q-value is updated, the next state is set to the current state, and the next iteration begins. This process continues until the termination criteria is met and the next episode begins. At the beginning of the next episode the state and agent are reset and the q-table remains persistent.

4.4 Policy Gradient Theorem

Unlike Q-learning, policy gradient methods do not use a value function to select an action; policy gradients select an action based on a parameterized policy without learning a value function. With policy gradient methods, the policy must be continuous and differentiable; this allows for policy gradients methods to be applied in continuous action-spaces. The performance of a policy-gradient algorithm can be measured by Equation 4.4,

$$J(\theta) = v_{\pi\theta}(s_0) \tag{4.4}$$

Where v is the value function determined by following the policy π with parameters θ at an initial state s_0. The cost function J can be difficult to determine, but the effect of the policy on the state given a set of parameters can be determined by applying the gradient policy theorem as shown by Sutton and Barto [15],

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|,s,\theta) \tag{4.5}$$

The policy gradient theorem uses gradient ascent to maximize the performance value. The two major algorithms that apply policy gradients are REINFORCE, a Monte Carlo based policy gradient and actor-critic methods. These algorithms can be applied to continuous action spaces, but the policy does not update until the episode is completed.

4.5 Actor-Critic and Deep Deterministic Policy Gradients

Temporal difference algorithms have demonstrated promising results when applied to low-dimensional discrete action-spaces[16]. When scaling to higher dimensional action-spaces or continuous action-spaces, they can become computationally expensive. In contrast, policy gradient methods perform well in continuous action-spaces, but can lead to difficulties in computing a good performance function. Actor-critic algorithms combine temporal difference and policy gradient methods to mitigate the drawbacks of the two algorithm types.

Actor-critic algorithms extend from policy gradient methods and temporal difference[15]. Actor-critic algorithms consist of an actor that selects an action based on a policy while the policy is evaluated by the critic using a value function. Deep Deterministic Policy Gradients (DDPG) is one of the many actor-critic algorithms.

The DDPG algorithm is an extension of the deterministic policy gradient algorithm[17], which use a combination of Q-learning and policy gradients to train a critic and actor model. The critic model, is represented as a neural network whose inputs are the state and action of an agent and the output is a single q-value. The loss function for the critic is defined in Equation 4.6 - 4.7,

$$y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'} \tag{4.6}$$

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{4.7}$$

where Q', μ', and θ' denote the target critic, target actor, and target neural network parameters, respectively. Target networks are used in Q-learning to stabilize neural networks by providing a fixed point preventing the estimated q-values from diverging. The actor network is also represented by a neural network with current agent state as the input. To encourage exploration, a noise parameter is added to the policy. The actor network is updated using the policy gradient method shown below.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N_b}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i} \tag{4.8}$$

DDPG updates the target networks using a conservative update, $\tau << 1$ to allow the target networks to change slowly to stabilize learning shown in Equations 4.9 - 4.10.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{4.9}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \tag{4.10}$$

Using a deep-q network allows the use of techniques such as replay memory and target networks. The algorithm for DDPG is shown below[18].

```
Initialize actor and critic networks and respective target networks;
Initialize replay buffer ;
for each episode do
    Initialize S;
    while S is not terminal do
        Choose an action according to the current policy;
        Take action A;
        Observe Reward R and S';
        Store in memory buffer (s,a,r,d,s') ;
        Sample a random mini-batch of $N_b$ transitions;
        Update the critic by minimizing the loss
          $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1}\gamma max Q(S_{t+1},a) - Q(S_t,A_t)]$;
        update the actor policy using policy gradient;
        update actor and critic target network ;
    end
end
```

Algorithm 2: Deep Deterministic Policy Gradient

4.6 Neural Networks

Neural networks are computational units inspired from biological neurons. An artificial neural network is a combination of interconnected artificial neurons or nodes that take in an input and produce an output. The output of an artificial neuron is calculated by an activation function which is a linear or non-linear function of the sum of the artificial neuron's inputs. An output is generated and sent the other nodes connected to it. Artificial neural networks were originally inspired by biological neural networks.

Artificial neural networks began as a system that can perform NOT, OR, and AND logical operations. The simplest of neural networks is the McCulloch-Pitts Neuron Model. The output of this model can be expressed in the equation 4.11 and Figure 4.2.

$$O^{k+1} = r\begin{cases} 1 \quad if \quad \Sigma_{i=1}^{n} w_i x_i^k \geq Threshold \\ 0 \quad if \quad \Sigma_{i=1}^{n} w_i x_i^k \leq Threshold \end{cases} \tag{4.11}$$

A feed-forward neural network is made up of layers connected to each other via weights. It is generally composed of an input layer, one or multiple hidden layers, and an output layer as shown in Figure 4.3. The outputs of the neural network are determined by the weights of the

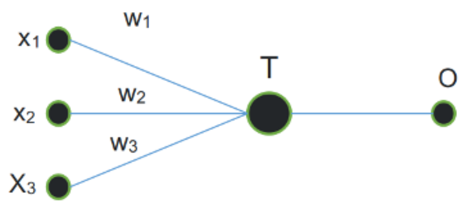

Figure 4.2: Diagram of McCulloch-Pitts Neuron Model [19]

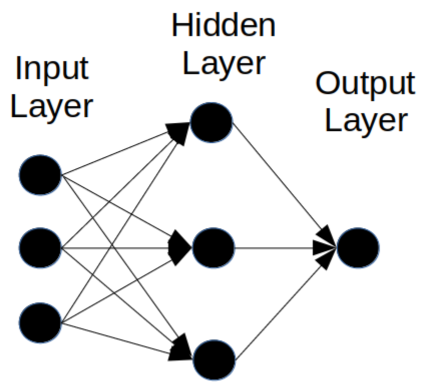

Figure 4.3: Feed-forward Neural Network

input to the nodes. A node is composed of the sum of the incoming weights of the node and an activation function as seen in Figure 4.4. where x is the sum of the input weights and y is output of the sigmoid function For a 2x2 example, the sum of the weights take the matrix form of,

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$
$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.12}$$

where **I** is the input vector and **W** is the weight matrix.

There are three common activation functions used for neural networks; sigmoid, hyperbolic tangent (tanh), and the rectified linear unit (RELU). Cybenko showed that any continuous function of n real values be approximated by adding a sigmoid function to a continuous feedforward neural network with one hidden layer and a sigmoid activation function[20]. The sigmoid function, Figure 4.5, is a popular choice with neural networks because it is asymptotic, where outputs can not be

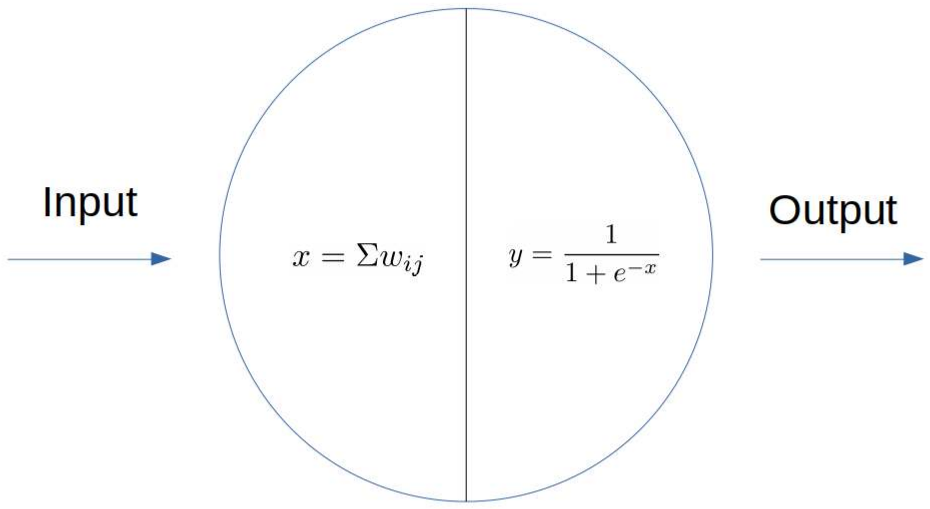

Figure 4.4: Neural Network Node with a Sigmoid Activation

either zero or one, and differentiable. Similar to the sigmoid function, tanh, shown in Figure 4.6

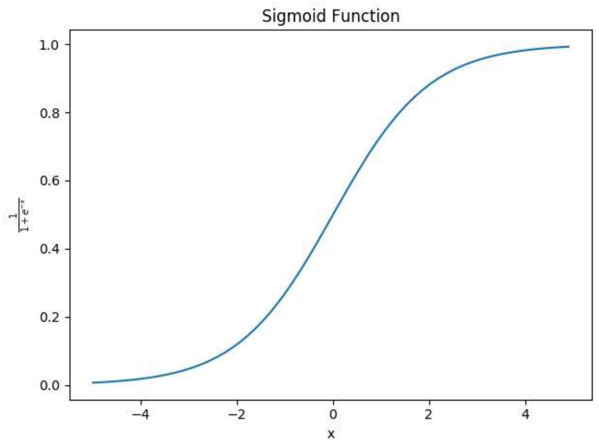

Figure 4.5: Sigmoid Activation Function

has a similar shape to sigmoid but has a range between -1 and 1 as shown in Figure 4.6. The activation functions above can run into the vanishing gradient problem. This can occur during the training phase of the neural network when the weight values are updated. As the number of hidden layers increase, the update of the weights along the layers decrease. This results in exponentially small updates during training causing the first layers in the hidden layer to update exponentially slower than the layers closer to the output layer.

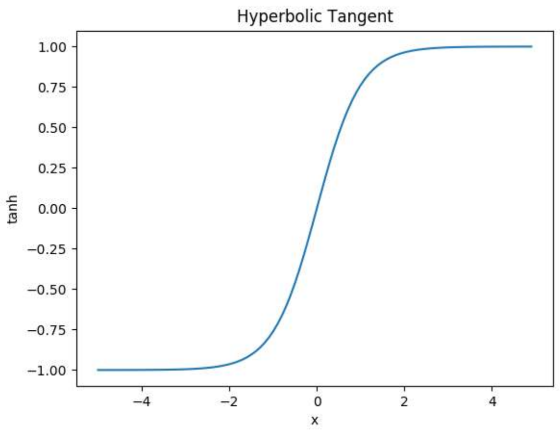

Figure 4.6: Hyperbolic Tangent Activation Function

Recently, the RELU activation function has become one of the most popular activation functions for neural networks as shown in Figure 4.7. Relu is also referred to as the ramp input and is expressed as

$$f(x) = max(0,x) \tag{4.13}$$

The gradient of this activation function is either 0 or 1, reducing the computational complexity and unlike sigmoid or tanh, relu has no upper limit preventing the vanishing gradient problem during training.

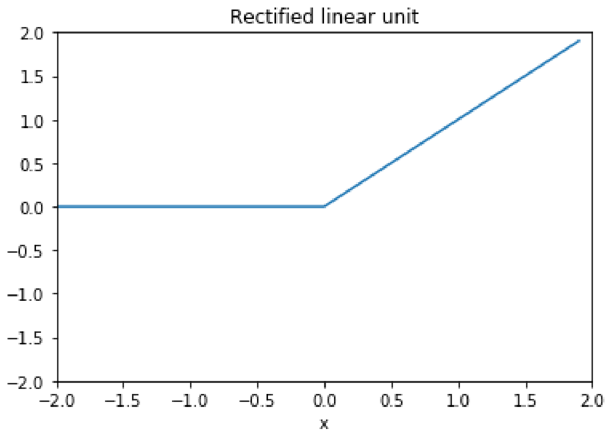

Figure 4.7: Rectified Linear Unit Activation Function

CHAPTER 5

IMPLEMENTATION

The work with this dissertation uses existing machine learning and space dynamics libraries because they have been tested and verified with other applications. Also this research aims to provide a general framework that can be applied to any space dynamics and machine learning library.

This section begins by discussing implementation of the spacecraft dynamics problem proposed in Chapter 3 in the machine learning framework described in Chapter 4. First the high-fidelity space dynamics simulation environment will be introduced.

Then we discuss how the reinforcement learning algorithm is implemented in the simulated space environment. Next we discuss the neural network architecture, the reward function, and the memory buffer used. Then we discuss how the reinforcement learning algorithm was validated using other environments. Chapters 6 and 7 discuss the three missions implemented and the conclusions drawn from the results.

5.1 Space Dynamics

This dissertation uses the Orekit space dynamics library. Orekit is a free and open-source software developed and maintained by the CS Group, a software company based out of France. This library is currently being used by Airbus Defense and Space, US Naval Research Laboratory, the Swedish Space Corporation, Thales Alenia Space, and the European Space Agency. The library contains both high and low-fidelity models for forces, propagation, and central bodies, and allows users to quickly and easily setup complicated space dynamics models.

Orekit defines a spacecraft as an object with many properties. The spacecraft properties used in this research are, date, inertial frame, mass, and specific impulse (ISP). The mass of the spacecraft is defined as the total mass. The user cannot change the mass after the spacecraft is defined, but Orekit does reduce the mass based on the ISP and thrust applied when a maneuver is implemented. The initial state is defined as a set of Keplerian orbital elements with an Earth-centered inertial frame. This inertial frame is defined as the x-axis aligned with the mean equinox, the y-axis rotated 90 degrees east along the Earth's equator, and the z-axis is aligned with the Earth's spin axis.

Orekit provides many different propagators with varying levels of fidelity. Propagators are used to calculate the dynamics of the spacecraft. The keplarian propagator uses a simple two-body model. The Eckstein-Hechler propagator is used in situations where the eccentricity is near zero. The Draper Semi-analytical Satellite Theory (DSST), is a combination of fast computation and accuracy. The propagator used here is the numerical propagator. This propagator uses numerical integration to approximate the system dynamics. The integrator is a Dormand–Prince Runge-Kutta method modified by Hairer, Norsett and Wanner[21]. The numerical propagator was selected because it has the largest selection of force models available.

Orekit also provides various force models to simulate different forces that a spacecraft may experience. The force models are implemented as perturbing accelerations and must be defined before propagation. To model central body forces, Newtonian central body attraction and a more complex Holmes and Featherston model is available. The force model selected is an implementation developed by Holmes and Featherstone[22]. This model approximates the Earth as a gravity field based on the shape of the Earth rather than the restricted two-body model 3.1. To implement the thrust, the constant thrust force model is used. This force model takes an input of thrust in Newtons, ISP in seconds, thrust start date as a date object, thrust duration in seconds, and a thrust direction as a vector. The attitude model uses the local vertical local horizontal (LVLH) frame which is similar to the RSW frame discussed in Chapter 3.

5.2 Actor-Critic Network

The reinforcement learning algorithm uses a total of four neural networks but two on them are unique. The actor network is a feed-forward neural network whose input is spacecraft's state at a given time and its outputs are three thrust values as seen in Figure 5.1. There are two hidden layers, each use the RELU activation function and layer normalization.

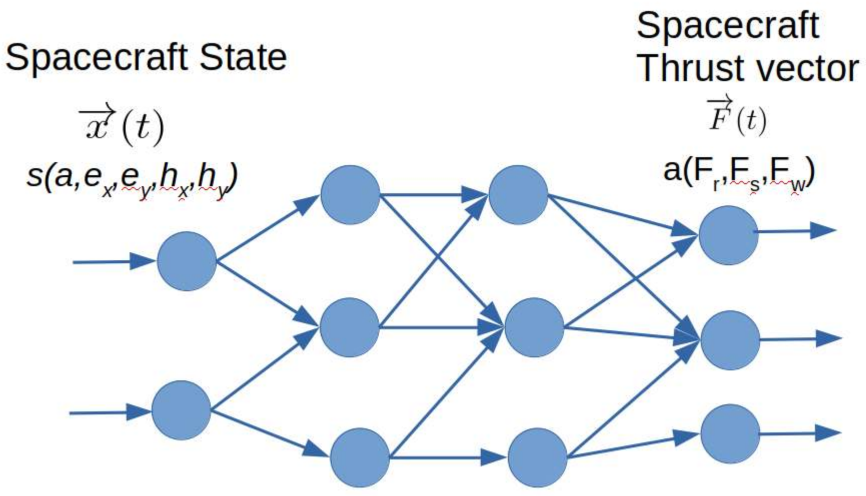

Figure 5.1: Actor Network

The RELU activation function, Figure 4.7, is used to prevent the vanishing gradient problem and the output from becoming saturated. Vanishing gradient occurs as the number of layers increase throughout the neural network as the gradient of the activation function is reduced to zero.

Inputs to a neural network may consist of values of different dimensions and scales. In the case of orbital elements, the semi-major axis is given in meters, the eccentricity is dimensionless, and the other elements are in radians. Normalization techniques are used to scale the weights and inputs in neural networks. Layer normalization normalizes the values across nodes of a hidden layer by the mean and standard deviation of the node values as shown below,

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{5.1}$$

where H is the number of nodes in the hidden layer, and a is the value of a hidden layer node. This method shows improved performance to batch normalization as demonstrated by Ba, Kiros, and Hinton [23], due to the reduction of covariate shift. The phenomenon is due to the potential for high output values from an activation layer. Also layer normalization can be used with any batch size. The batch size is the number of smaples that will be passed through the neural network at a given time.

The output layer consists of three nodes, each one representing a thrust direction defined in the radial, tangential, and normal directions. The hyperbolic tangent function, Figure 4.6, is used as the activation function to give an output between -1 and 1. Each of the three outputs are multiplied by a scale factor representing the maximum thrust value.

The second neural network used is the critic network. The architecture of the critic network is similar to the one developed in by [18]. The critic network takes the state and action as the input, but the action input connects to the second hidden layer directly whereas the state input connects to the first hidden layer as shown in Figure 5.2. For the critic network only the first hidden layer

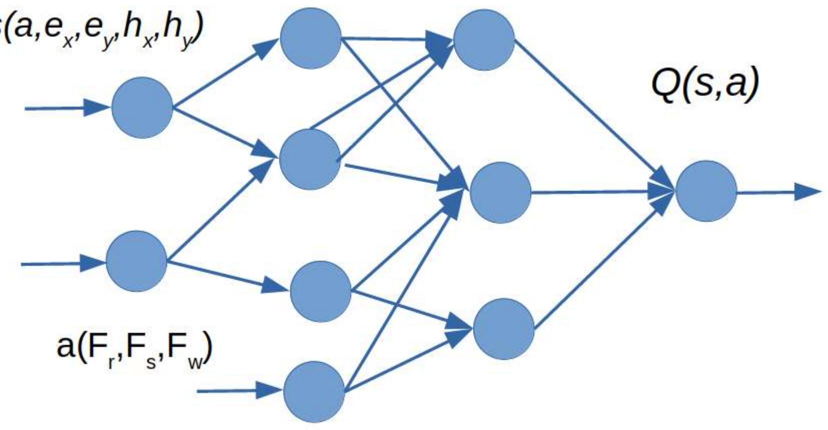

Figure 5.2: Critic Network

uses layer normalization and both hidden layers use the RELU activation function. The output of the critic network is a single node with a linear activation function, meaning the output is not multiplied by a value.

5.3 Reward Function

The reward function can be one of the most challenging choices when designing an environment. The quality of the reward function can dictate how quickly the agent converges to a solution. Making a reward function too specific will cause the agent to quickly converge to a solution, but it will prevent the agent from adequately exploring the environment and achieve a more optimal solution. Giving the agent a sparse reward may significantly increase the runtime, making it difficult for the agent to converge to a solution. Using a negative based reward function, the agent will tend to try to obtain a less negative reward value as fast as possible. For a positive based reward function, the agent will try to accumulate as much reward as possible per time step. The reward function in this research is based on the squared difference between the target state and current state shown below,

$$r_a = \frac{\sqrt{(a_{targ} - a(t))^2}}{a_{targ}} \tag{5.2}$$

$$r_{ex} = \sqrt{(e_{x,targ} - e_x(t))^2} \tag{5.3}$$

$$r_{ey} = \sqrt{(e_{y,targ} - e_y(t))^2} \tag{5.4}$$

$$r_{hx} = \sqrt{(h_{x,targ} - h_x(t))^2} \tag{5.5}$$

$$r_{hy} = \sqrt{(h_{y,targ} - h_y(t))^2} \tag{5.6}$$

$$r = -(r_a\alpha_a + r_{e_y}\alpha_{e_x} + r_{e_x}\alpha_{e_y} + r_{h_x}\alpha_{h_x} + r_{h_y}\alpha_{h_y}) \tag{5.7}$$

the orbital element corresponding to the semi-major axis a is normalized by the target state to scale accordingly with the other orbital elements. The variables α_a, α_{e_x}, α_{e_y}, α_{h_x}, α_{h_y} are scaling factors to scale the relative weights of the orbital elements. Proper scaling of the reward function is required for the agent to learn a good policy.

When the agent reaches the target state the episode terminates and the agent is reward with a value of 1. Giving a high positive reward value when the agent reaches a target state would result in the agent never reaching the target state in the following episodes. A high positive reward value when the agent releases the target state, cause the q-values to become unstable and continuously increase regardless of the action taken in future episodes.

5.4 Experience Replay

Replay memory is used in the training of deep-q neural networks. It is used to break the correlation between consecutive samples improving learning efficiency. The experiences of the agent are stored as a tuple in a double-ended queue describing the state-action at a time t.

$$e_t = (s_t, a_t, r_t, s_{t+1}, terminal) \tag{5.8}$$

A simple method to implement replay memory is to uniformly sample experiences from the replay memory buffer. Uniform sampling is done by randomly selecting a batch of experiences from the memory buffer. The drawbacks of this approach is that it is not an efficient way to sample experiences; the chance of sampling positive experiences is the same as the chance of sampling negative experiences, which reduces efficiency of training.

A more efficient method to sample experiences is with prioritized experience replay developed by Schaul, Quan, Antonoglou, and Silver [24]. This method organizes experiences based on a priority value p_t, and the experiences with higher values have a greater chance of being sampled more often.

$$e_t = ((s_t, a_t, r_t, s_{t+1}, terminal), p_t) \tag{5.9}$$

The priority values are stored in a binary heap for efficient sorting and selection. The priority value

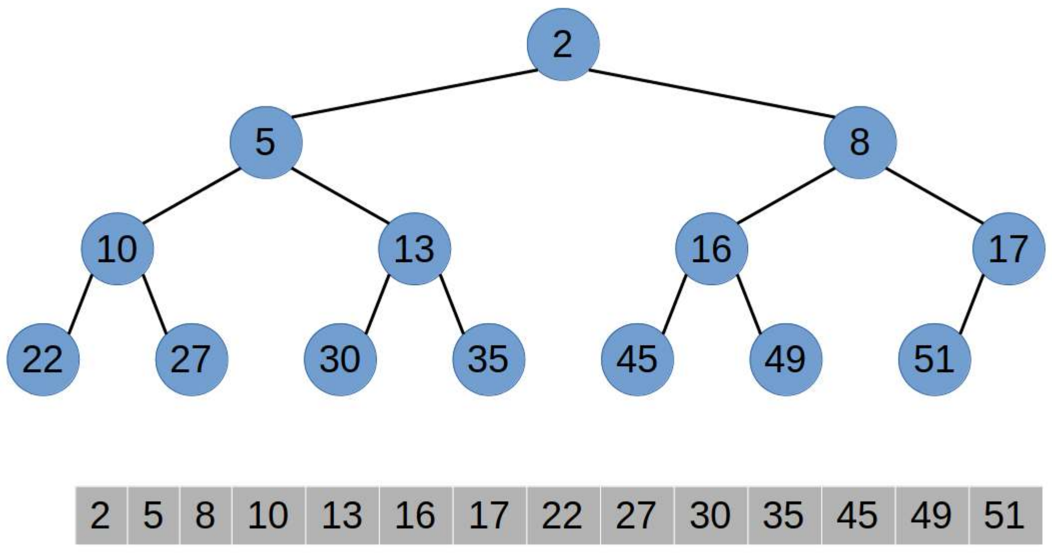

Figure 5.3: Binary Heap Data Structure

for the experience is updated according to the TD-error,

$$p_t = |\delta_t| + \varepsilon \delta_t = \qquad R + \gamma Q(S_{t+1}, a) - Q(S_t, A_t) \tag{5.10}$$

where ε is a constant to ensure no priority has a value of zero. The priority values are then sorted and stored in the heap binary. During training, the priority values are sampled according to a probability value given as,

$$P(i) = \frac{p_t^\alpha}{\sum_k p_k^\alpha} \tag{5.11}$$

where α is a hyperparameter to introduce randomness in the selection process where a higher value encourages sampling higher priority values and a lower value encourages random sampling, and k is the batch size. The probability sampling introduces a bias to replay high priority experiences; to reduce the bias, importance sampling weights are used shown in Equation 5.12,

$$w_t = (\frac{1}{N_b} \cdots \frac{1}{P(i)})^\beta \tag{5.12}$$

where N is the memory buffer size and β is a hyperparameter that determines how much the weight value effect learning. As training progresses, β should increase to one because near the end of training the q values begin to converge, making the weight value more important. The weight value is passed into the critic network's loss function during training as shown in Equation 5.13,

$$L = \frac{1}{N_b} \sum_i w_i \delta_i^2 \tag{5.13}$$

where N_b is the batch size.

5.5 Reinforcement Learning Implementation

The reinforcement learning algorithm used in this research is based on the DDPG algorithm discussed in Section 4.5. This version of the algorithm differs from Algorithm 2 by implementing actor noise decay and prioritized experience replay. Figure 5.4, shows a high level overview of the algorithm.

Initially, an instance of Orekit is started and loaded. The spacecraft specifications, propagator, central body force model, and initial and target orbits are initialized. The neural networks and their respective target networks are initialized. The memory buffer is initialized by allocating a double ended queue (deque) with a maximum memory buffer size.

The episode begins by setting the current orbit state to the initial orbit state. A thrust value is obtained by passing the current state into the actor network according to its policy and a noise

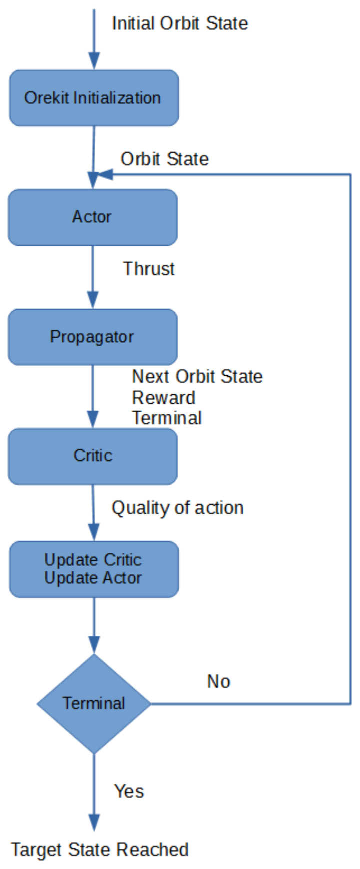

Figure 5.4: Software Architecture

value is updated as shown in Equation 5.14,

$$\mu_t(s) = a_t(s) + N_d N_t \tag{5.14}$$

where N_t is the exploration noise based on the Ornstein-Uhlenbeck process[25], and N_d is a decay rate. A decay rate was introduced to reduce noise as the agent explores the environment. The thrust value is then passed into a step function where it is added as a constant thrust force. Orekit then propagates the orbit for over a specified time step and returns a new spacecraft state. The new state along with the thrust value is passed into a reward method to compute a reward value and determine if a terminal state is reached for the current state-action pair. The step is completed and returns the state, reward value, and terminal state status back to the reinforcement learning algorithm.

The algorithm then adds these values into its replay buffer and assigns a priority value. A batch of experiences are then sampled from the replay buffer and the TD-error is computed. The critic is updated by minimizing the loss function which using equation 4.6. The loss is then backpropagated through the neural network using the adaptive momentum estimation (adam) algorithm. The TD-error is the difference between the actual reward and the expected reward, in the case of the spacecraft, the TD-error decreases as the spacecraft takes actions that will lead it closer to the target state.

The actor is then updated by sampling the gradient of the actor's policy and the gradient of the critic network output. The target critic and actor networks use a soft update method according to the updated versions of the actor and critic networks and a hyperparameter τ.

This process repeats itself until a terminal state is reached. A terminal state occurs if the spacecraft reaches the target state, or if the maximum duration of the mission is exceeded. When a new episode begins. A maximum duration was selected as a terminal state to prevent the agent from exploring portions of the environment where it will never reach the target state. The spacecraft and the propagator is reset to the initial state. Algorithm 3 illustrates the modified DDPG algorithm.

```
Initialize Orekit environment ;
Initialize actor $\mu$ and critic $Q$ neural networks with weights $\theta^{\mu}$, $\theta^{Q}$ ;
Initialize respective target network $\mu'$ and $Q'$ weights $\theta'^{\mu} \leftarrow \theta^{\mu}$, $\theta'^{Q} \leftarrow \theta^{Q}$ ;
Initialize priority replay buffer ;
for each episode do
    Initialize State s;
    while $t <\ t_{max}$ or not terminal do
        Choose an action $a_t(s)$ according to the current policy $\pi(\theta^{\mu})$;
        Take action $\mu_t(s) = \mu_t(s_t) + N_d N_t$;
        Observe Reward $r$ and new state $s_{t+1}$;
        Set the priority queue value $p_t = |r_t| + \varepsilon$ ;
        Store in memory buffer $((s_t, a_t, r_t, s'_{t+1}, terminal), p_t)$ ;
        Sample from memory a batch of $N_b$ transitions based on importance sampling
          $w_i$ $((s_i, a_i, r_i, s_{i+1}, terminal), w_i)$;
        Calculate TD-error ;
        $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ ;
        $\delta_i = y_i - Q(s_i, a_i|\theta^{Q})$ ;
        Update the critic by minimizing the loss: ;
        $L = \frac{1}{N_b} \sum_i (\delta)^2 w_i$ ;
        Update the priority values based on the TD-error: ;
        $p(i) = |\delta_i| + \varepsilon$ ;
        update the actor policy using policy gradient;
        $\nabla_{\theta^{\mu}} J \approx \frac{1}{N_b} \sum_i \nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}$
        update actor and critic target network parameters ;
        $\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$ ;
        $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$;
        $s = s_{t+1}$;
    end
end
```

Algorithm 3: Modified DDPG Algorithm

5.6 Algorithm Validation

Testing to ensure all components are working correctly is a vital step in ensuring accurate results. A difficult task in reinforcement learning is model validation. Model validation is testing to ensure that neural networks and the algorithm are setup correctly. Proper validation ensures that both the critic and actor are learning correctly, which may save time by not training an incorrect model. Validation of a neural network is difficult because currently the weight values can not be interpreted or predicted. One validation method is to run the algorithm in a known environment and monitor its performance during training. The metric used to measure the performance the total reward per episode. If the maximum reward steadily increases over each episode, it can be assumed that the agent is learning the correct policy since the goal of the agent is to attain the maximum reward.

When developing the algorithm, multiple test environments were used to validate proper training. The test environments are part of OpenAI gym, a Python package that contains environments for testing reinforcement learning algorithms. The environments are standardized to easily interchange the environments and test reinforcement learning algorithms. Three test environments were used in testing the algorithm developed in this paper; the pendulum, lunar lander continuous, and bipedal-walker environments.

These three environments were chosen because of their increasing complexity and they are continuous. In the first steps of algorithm development, the pendulum environment was used to quickly test development. When making changes to the algorithm or neural networks, the pendulum environment was used extensively to ensure the algorithm can converge to a solution. When the algorithm became more complex, the other two environments were used to test the performance. The test environments are described in Appendix A.

CHAPTER 6

STATIC MISSION RESULTS

Three representative spacecraft targeting problems are defined to test the algorithm: a generalized orbit raising maneuver, a semimajor axis change a Medium-Earth (MEO) to Geosynchronous Earth orbit (GEO) transfer.

These three problems were selected because they encompass some of the missions that a low-thrust spacecraft might perform. It is assumed that the thruster on the spacecraft can produce the given thrust instantaneously and with high efficiency. For all three scenarios, the same spacecraft parameters shown in Table 6.1 are used.

Table 6.1: Spacecraft Parameters

Dry Mass (kg)	750
Fuel Mass (kg)	150
$I_{sp}(s)$	3100
Max Thrust (N)	0.6

To ensure best potential results for all three missions, three different models are trained. Each model is optimized to solve a particular type of mission, but parameters in Table 6.2 of the RL algorithm parameters remain the same.

Table 6.2: Algorithm hyperparameters, Algorithm 3

step size (seconds)	500.0
ε	0.01
$\dot{\varepsilon}$	0.0001
γ	0.99
τ	0.01

The propulsion system is based on ion thrusters that have a moderate I_{sp} but a higher level of thrust similar to High Power Electric Propulsion (HiPEP) or magnetoplasmadynamic thruster

(MPDT) experimental ion thrusters [26]. For each mission, the agent was updated at an interval of 500 seconds per step, approximately 8.33 minutes per step. The value ε, The replay memory parameters in Table 6.3 remain the same among all missions. The parameters α and β defined in the experience replay section, are selected based on the paper by Schaul, Quan, Antonoglou, and Silver [24].

Table 6.3: Memory hyperparameters Equations 5.11, 5.12

Memory buffer	1,000,000
α	0.10
β	0.01

In Table 6.3, the memory buffer is the maximum number of experiences stored, and α and β are hyperparameters to determine the probability of selecting higher value experiences. The true anomaly orbital element θ, is not targeted or included in the reward function in any of the missions. Since there are no rendezvous maneuvers and only the orbit state is considered, θ can be omitted. It is necessary to establish a tolerance values for the orbital elements being targeted. When the difference between the target orbit state and current orbit state are within the tolerance values, the episode terminates successfully. The orbit element tolerances are listed in Table 6.4.

Table 6.4: Orbital element tolerances

a km	10
e_x (rad)	0.01
e_y (rad)	0.01
h_x (rad)	0.001
h_y (rad)	0.001

The duration set for all missions is not the target date, but rather the maximum duration the agent is given to reach the target state. If the episode exceeds the maximum duration or if the agent reaches the target state before the maximum duration, the current episode terminates. The maximum duration is estimated by applying a constant thrust in the radial, tangential, or normal directions until the current semimajor axis or inclination reaches or exceeds the corresponding target orbital element. This process does not solve the orbit maneuver problem since only two elements are tested. An additional day is added to the estimated maximum duration to prevent the agent from applying only a maximum thrust.

To determine which thrust direction to apply to reach a target orbital element, Gauss's Variational Equations (3.5-3.10), are used to determine which thrust acceleration influences which change in Keplerian orbital elements. By inspection of Gauss's Variational equations in the equinoctial frame (3.16 - 3.20), we can form an understanding on how the thrust direction relates to the change in each orbital element.

$$\frac{da}{dt} = f(F_S, F_R) \tag{6.1}$$
$$\frac{de_x}{dt} = f(F_R, F_S, F_W) \tag{6.2}$$
$$\frac{de_y}{dt} = f(F_R, F_S, F_W) \tag{6.3}$$
$$\frac{dh_x}{dt} = f(F_w, F_R, F_S) \tag{6.4}$$
$$\frac{dh_y}{dt} = f(F_W, F_R, F_S) \tag{6.5}$$

where the magnitude of the first thrust direction listed has the largest influence on the corresponding orbital element.

6.1 General Orbit Change

A general orbit change maneuver includes changes in all of the orbital elements a, e_x, e_y, h_x, h_y.

Table 6.5: General orbit change mission

Orbit Parameters	Keplerian Initial State	Keplerian Target State	Orbit Parameters	Equinoctial Initial State	Equinoctial Target State
a (km)	5500	6300	a (km)	5500	6300
e	0.20	0.23	e_x (rad)	0.766	0.669
i (deg)	5.0	5.3	e_y (rad)	0.642	0.743
ω (deg)	20.0	24.0	h_x (rad)	0.041	0.042
Ω (deg)	20.0	24.0	h_y (rad)	0.014	0.018
Duration (day)	4		Duration (day)	4	

Table 6.5 is a Low-Earth orbit (LEO) mission with a slight eccentricity and inclination change. For this mission, the agent has the following hyperparameters shown in Table 6.6 The number and size of the hidden layers are the same for both the actor and the critic networks. The algorithm terminated when the agent successfully reached the target state consecutively, which

Table 6.6: Neural Network hyperparameters

layer 1	500
layer 2	450
learning rate (actor)	0.0001
learning rate (critic)	0.001
τ	0.01

took 600 episodes. A 3D figure of the trajectory can be seen in Figure 6.1, the blue line shows the

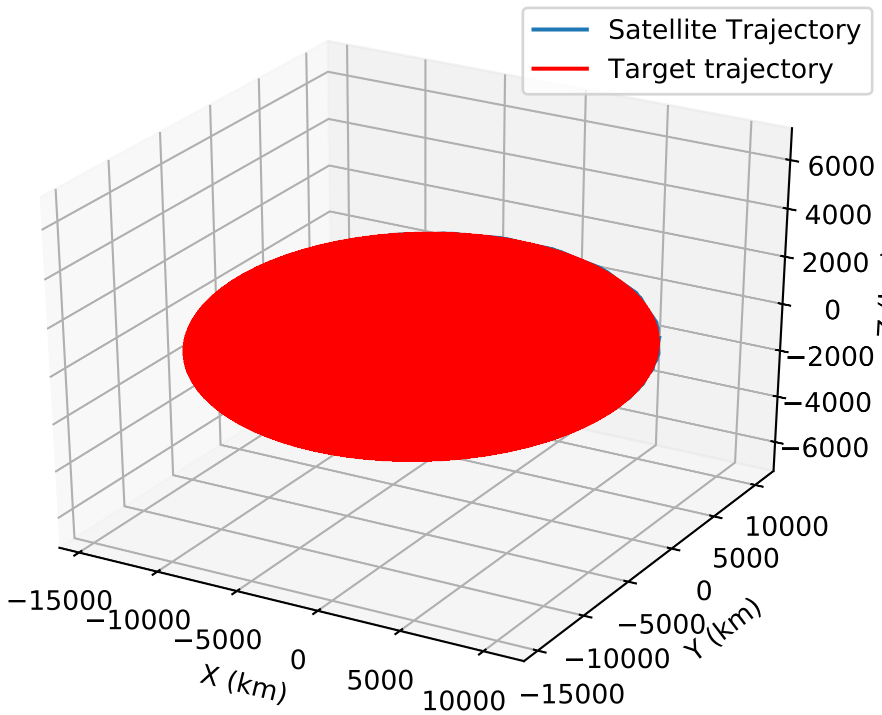

Figure 6.1: General Orbit Change Maneuver

trajectory of the spacecraft and the red shows the target orbit state. Figure 6.2, shows a successful episode of the agent trained to perform the general orbit change maneuver.

Figure 6.2 shows the trajectory of the spacecraft in equinoctial orbital elements. The agent reached the target a, h_x, and h_y around step 350. An interesting observation is that the semimajor axis remained near the initial value until the agent reached the target e_y, h_x, and h_y states. More

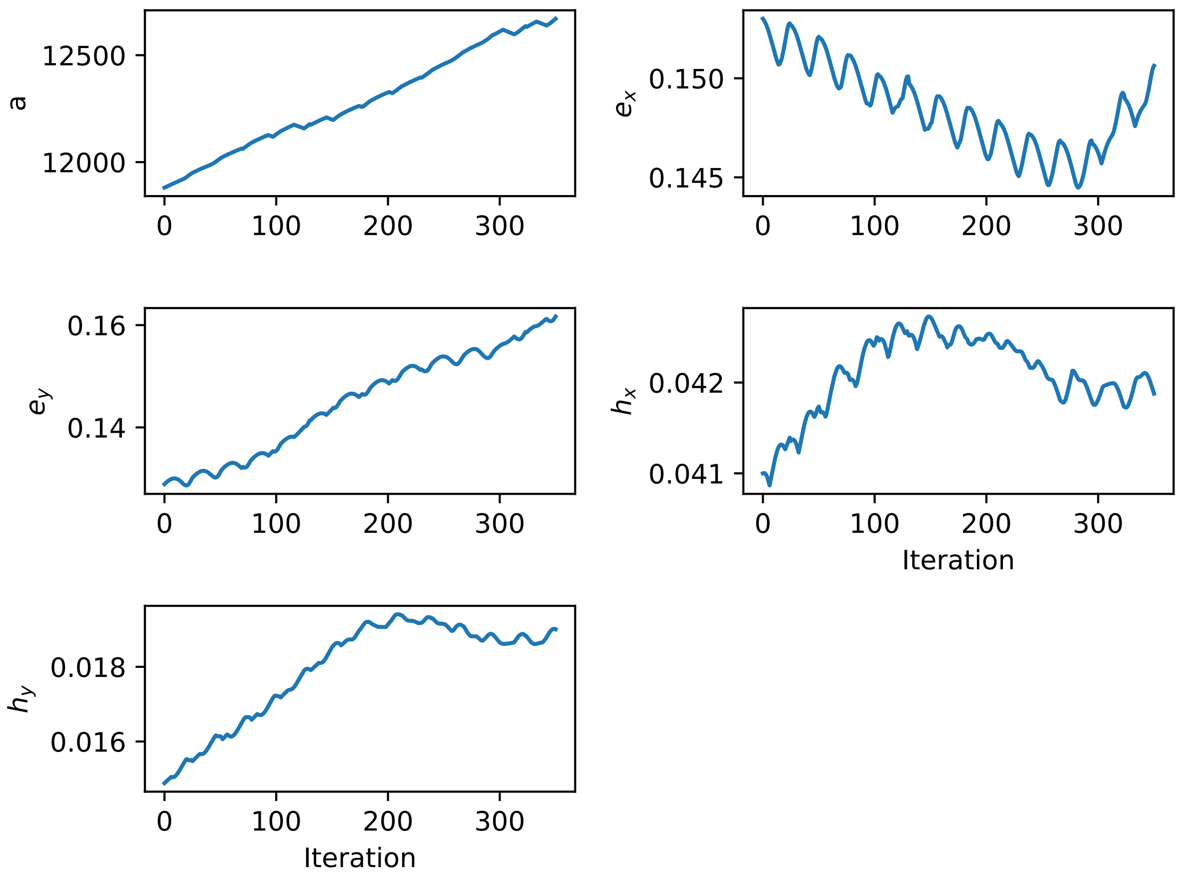

Figure 6.2: Orbital Elements of the General Orbit Change Maneuver

insight on this behavior can be seen in Figure 6.3 showing the thrust profile. When setting the weights α as shown in Table 6.7 for the reward function, the values are adjusted to ensure that each change in orbit element has the same scale.

Table 6.7: General orbit element change reward function coefficients, Equation 5.7

α_a	1
α_{e_x}	1
α_{e_y}	1
α_{h_x}	10
α_{h_y}	10

The reward weight values are used to prevent the agent from ignoring the orbital elements that have a small delta in the targeting problem. The results of the reward function can be seen in Figure 6.4. Initially, the reward values are very low, then there is a jump in the reward values

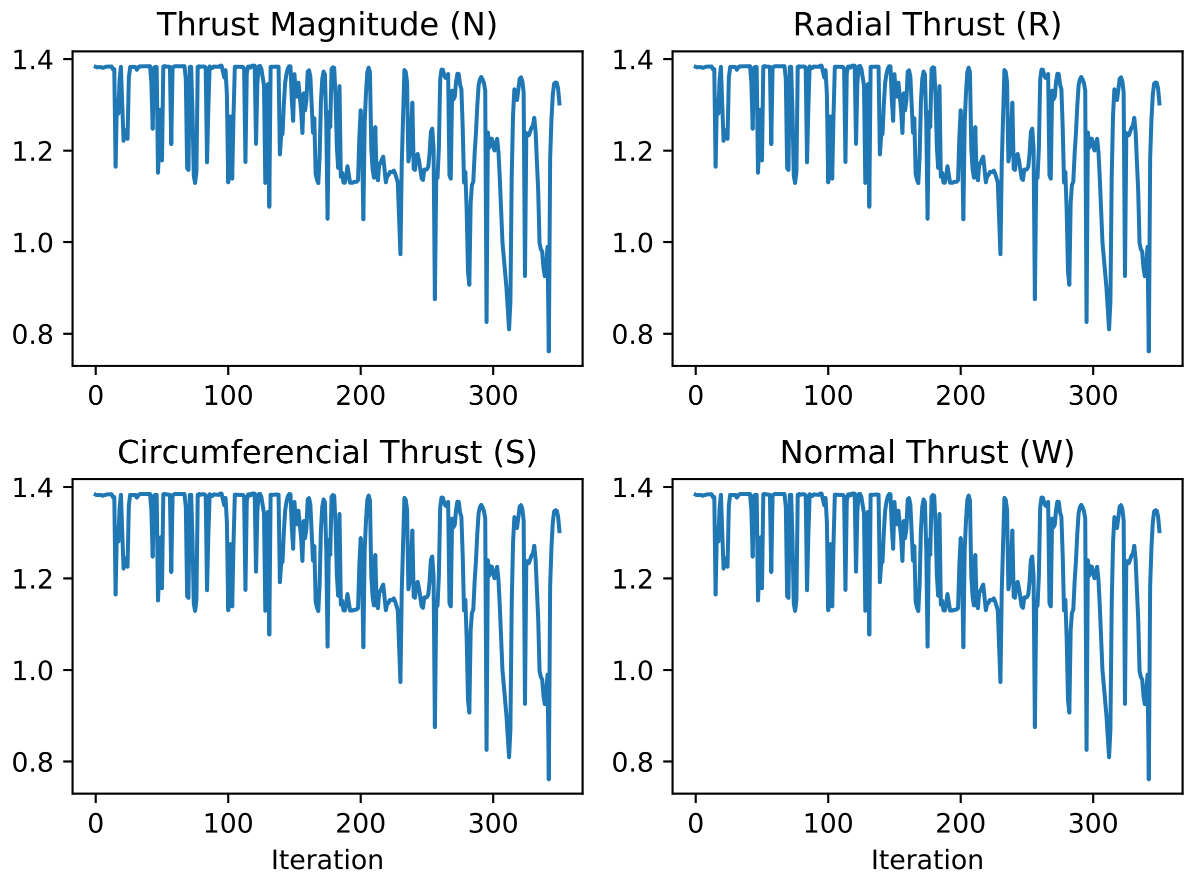

Figure 6.3: General Orbit Change Maneuver Thrust Profile

around 100 episodes. From episode 100 to 350, the agent nears the target state but does not reach that state. After around 350 episodes, the agent is learning the policy well because there is a jump in the reward function.

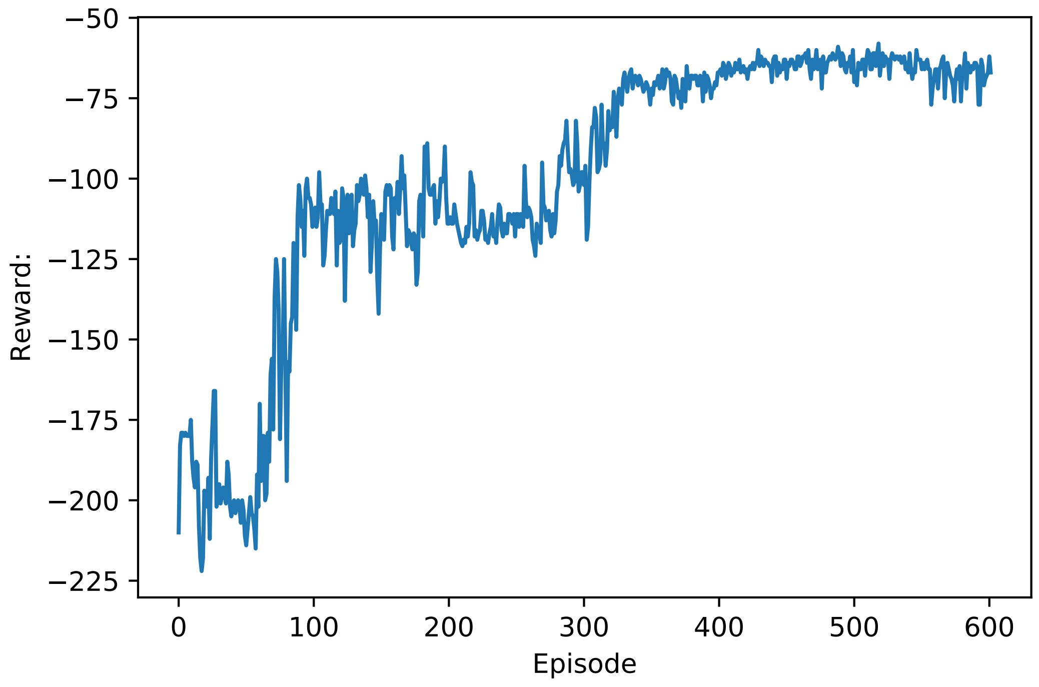

Figure 6.4: Reward Function Output for Generalized Orbit Change Mission

6.2 Semimajor Axis Change and MEO to GEO Missions

A semimajor axis change maneuver increases the semimajor axis of an orbit while keeping other orbital elements fixed. The first semimajor axis change example demonstrates a smaller semimajor axis change of 5000 km as shown in Table 6.8. The neural network model for this

Table 6.8: Semimajor axis change mission

Orbit Parameters	Keplerian Initial State	Keplerian Target State	Orbit Parameters	Equinoctial Initial State	Equinoctial Target State
a (km)	5000	10000	a (km)	5000	10000
e	0.10	0.10	e_x (rad)	0.939	0.939
i (deg)	1.0	1.0	e_y (rad)	0.342	0.342
ω (deg)	10.0	10.0	h_x (rad)	0.00859	0.00859
Ω (deg)	10.0	10.0	h_y (rad)	0.00151	0.00151
Duration (day)	10		Duration (day)	10	

mission was enlarged to using three hidden layers instead of two to allow the agent to converge to the target state, and the number of nodes was increased as shown in Table 6.9. When training

Table 6.9: Neural Network hyperparameters for semimajor axis change mission

layer 1	1028
layer 2	850
layer 3	200
learning rate (actor)	0.0001
learning rate (critic)	0.001
τ	0.01

for small semimajor axis change maneuvers, the reward function parameters are the same values used in Table 6.7. The semimajor axis changes linearly over the mission duration while it is

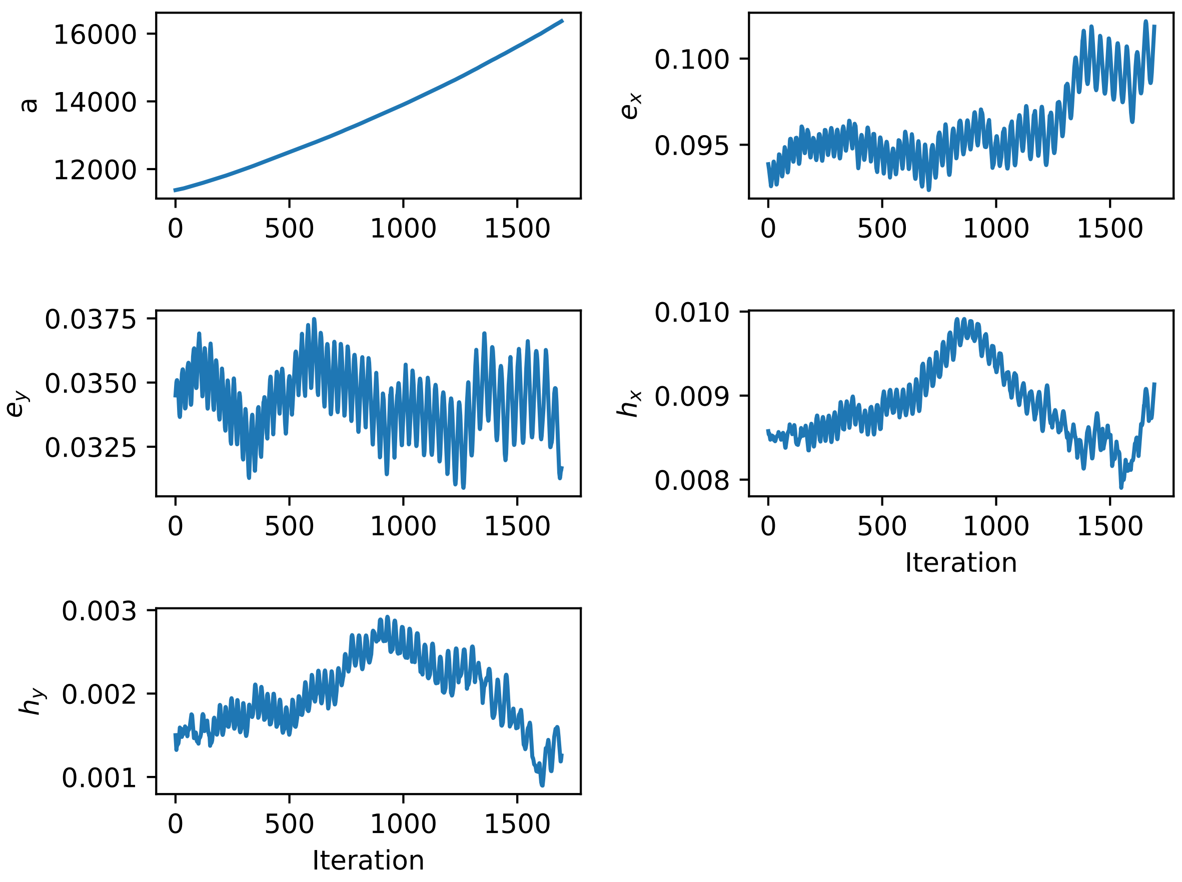

Figure 6.5: Orbital Elements for Semimajor Axis Change Maneuver

attempting to maintain all other orbit elements near the target state as shown in Figure 6.5. The thrust output for this mission is given in Figure 6.6. The spacecraft applies a maximum thrust in the tangential direction for the duration of the mission time, while the radial and normal directions heavily fluctuate between ± 0.6 N to maintain the other orbital elements. The reward function for

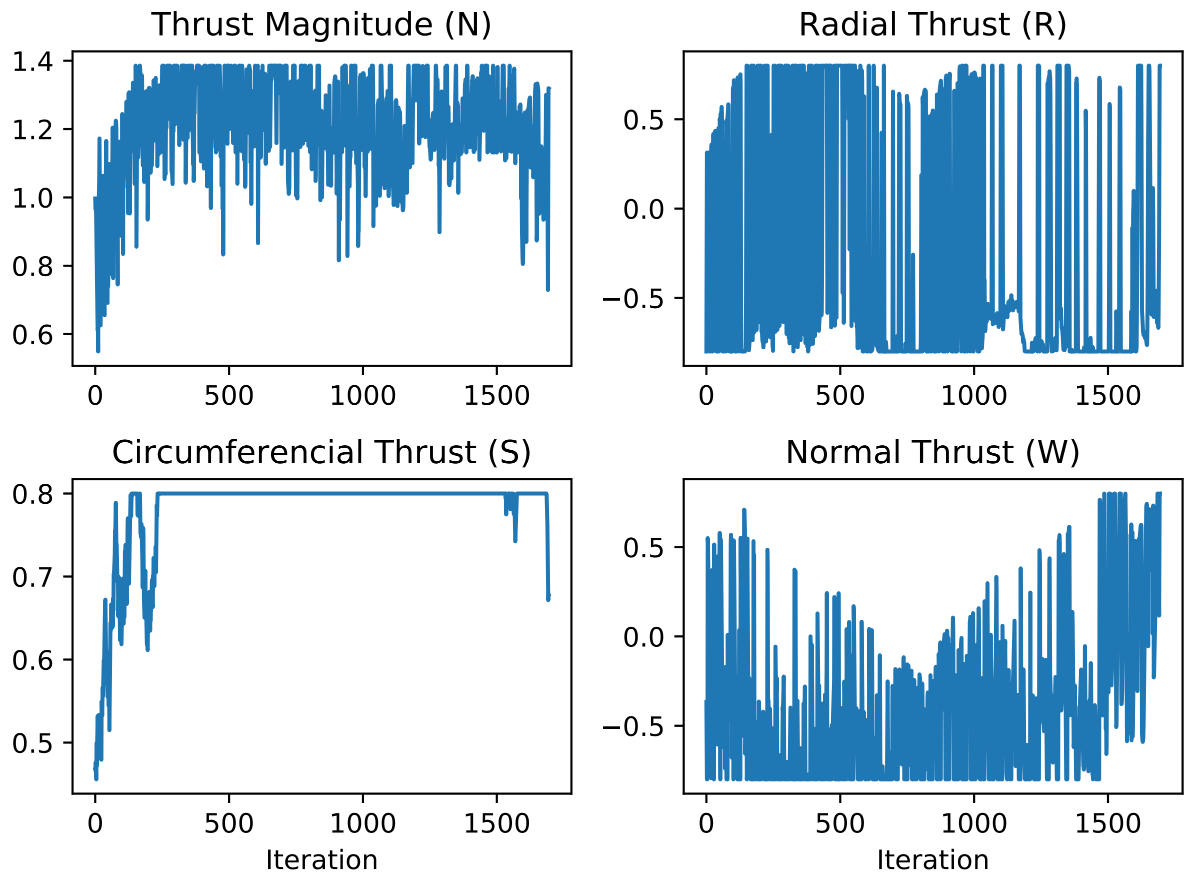

Figure 6.6: Thrust Output for Semimajor Axis Change Mission

the semimajor axis change steadily increases with training time as shown in Figure 6.7.

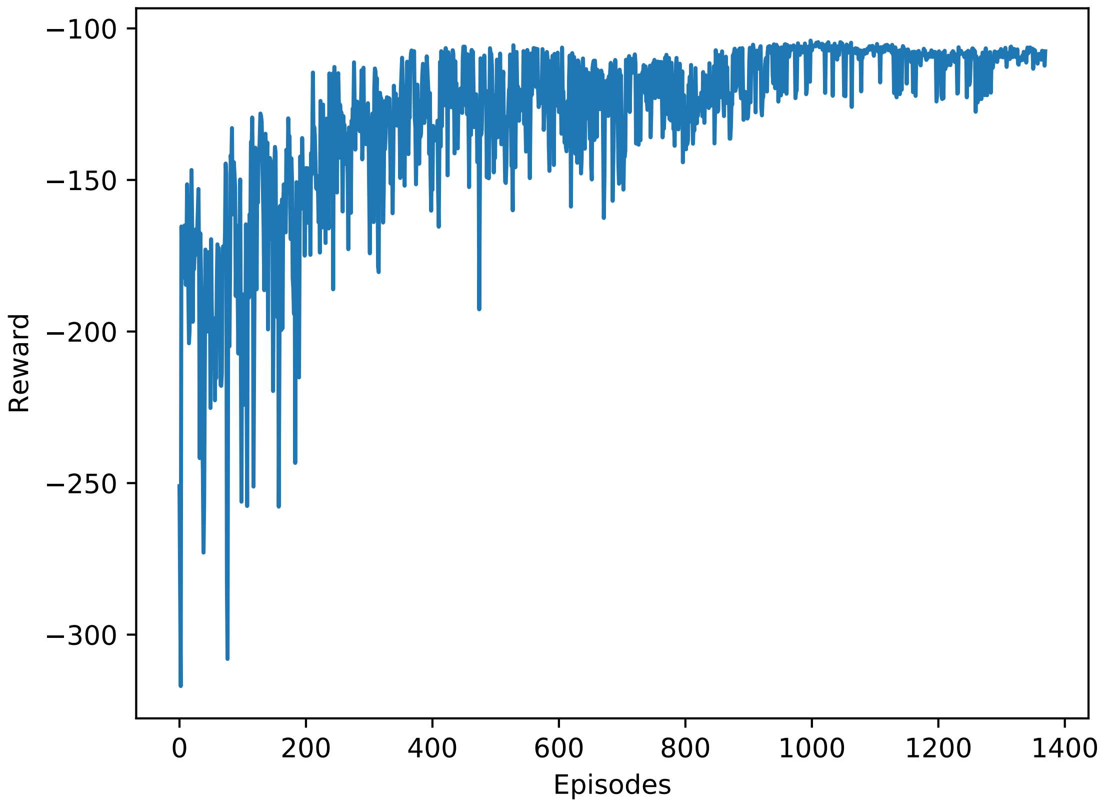

Figure 6.7: Reward Function for a Semimajor Axis Change Mission

For a MEO to GEO mission, the orbit periods are longer, making it possible to perform a larger orbit change during each revolution. The mission parameters for the MEO to GEO orbit is given in Table 6.10 below, For this mission, the maximum duration is significantly higher than the other previous missions, but e, i, ω, and Ω, remain the same as the LEO semimajor axis change. The neural network hyperparameters were kept constant as in the semimajor axis change maneuver, except for the third hidden layer which was increased from 200 to 500 nodes as shown in Table 6.11 below. The agent was trained for 1000 episodes, with the successful mission trajectory shown in Figure 6.8. In Figure 6.9, the agent was attempting to maintain the target e_x, e_y, h_x, and h_y elements while linearly increasing the semimajor axis. This can be better observed in the thrust output in Figure 6.10. For the entire mission duration, the agent applies a near maximum thrust in the tangential direction, increasing the semimajor axis, and then attempts to stabilize the other orbital elements by thrusting in bursts in the radial and normal directions. The coefficients for the reward function are given in Table 6.12. All the components except for h_y are weighted by 10. From testing various weight values, giving a small weight value to the eccentricity components would cause the orbit to reach the correct semimajor axis but the target orbit would have a high

Table 6.10: MEO to GEO mission parameters

Orbit	Keplerian		Orbit	Equinoctial	
Parameters	Initial State	Target State	Parameters	Initial State	Target State
a (km)	26000	41000	a (km)	26000	41000
e	0.10	0.10	e_x (rad)	0.939	0.939
i (deg)	1.0	1.0	e_y (rad)	0.342	0.342
ω (deg)	10.0	10.0	h_x (rad)	0.00859	0.00859
Ω (deg)	10.0	10.0	h_y (rad)	0.00151	0.00151
Duration (day)	15		Duration (day)	15	

Table 6.11: Neural Network hyperparameters

Hyper parameters	
layer 1	1028
layer 2	850
layer 3	500
learning rate (actor)	0.0001
learning rate (critic)	0.001
τ	0.01

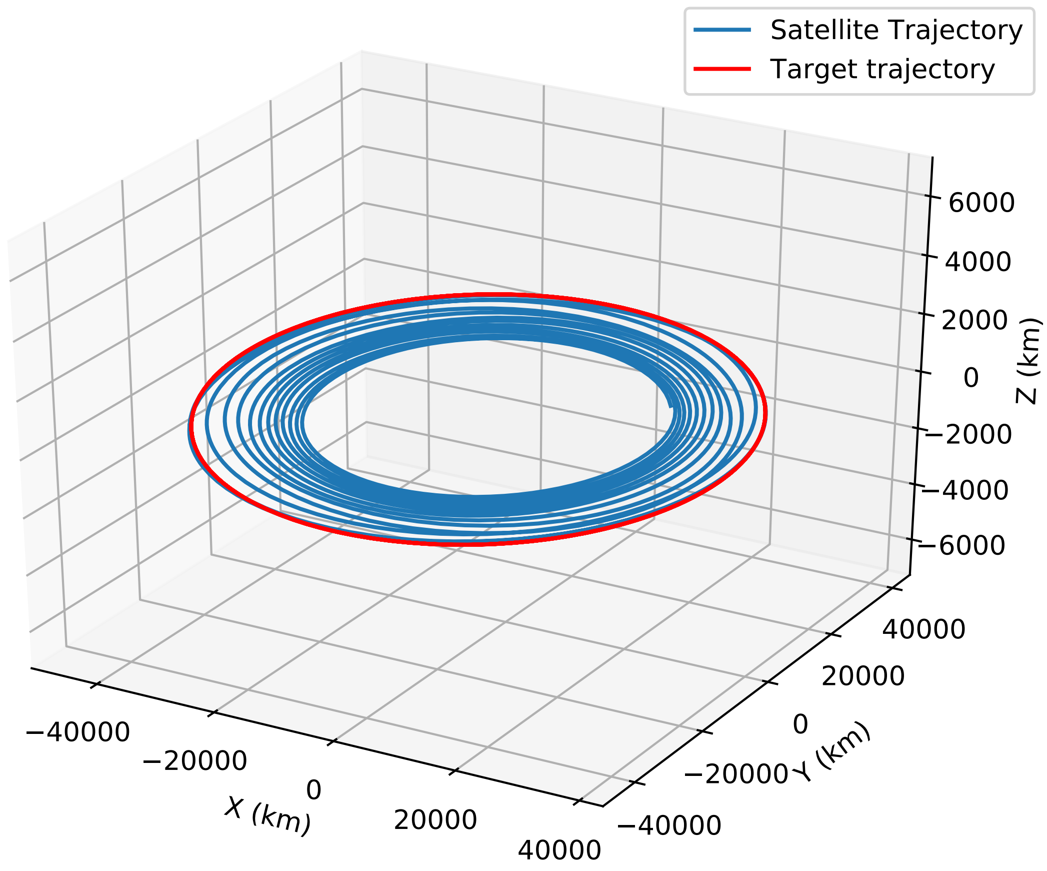

Figure 6.8: Trajectory of MEO to GEO Mission

eccentricity value. Figure 6.11 shows large spikes in the reward values, this was due to the agent reaching a small semimajor axis at the end of the mission. Around episode 900, the agent begins to converge to the target state.

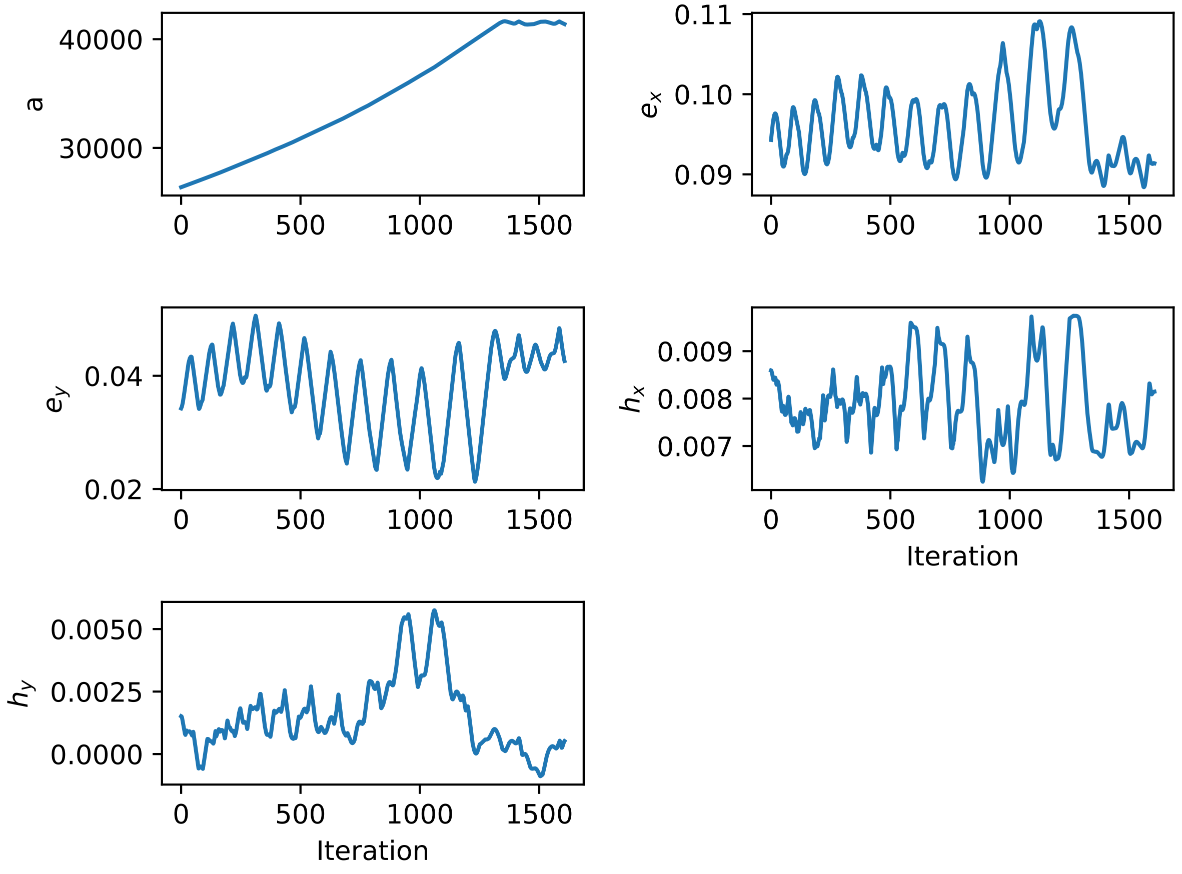

Figure 6.9: Orbital Elements for MEO to GEO Mission

Table 6.12: MEO to GEO reward parameters, , Equation 5.7

Reward function coefficients	
α_a	10
α_{e_x}	10
α_{e_y}	10
α_{h_x}	10
α_{h_y}	1

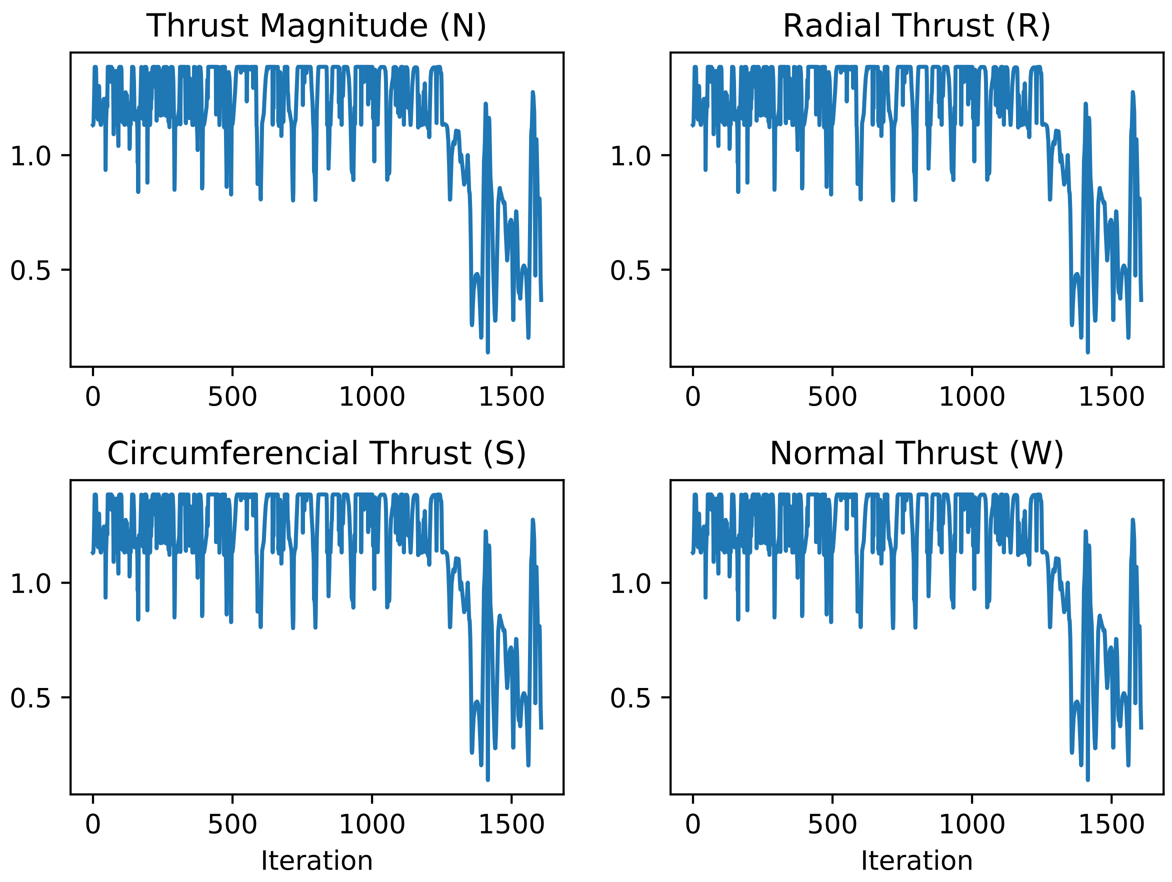

Figure 6.10: Thrust Output for MEO to GEO Mission

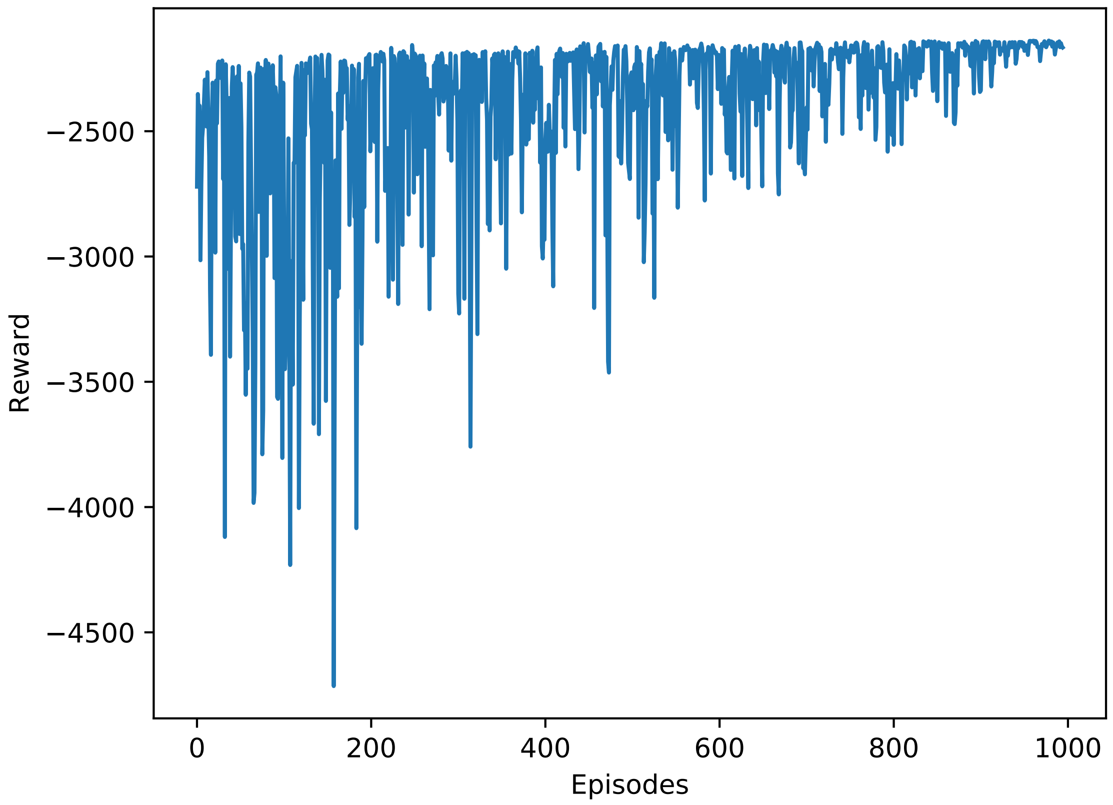

Figure 6.11: Reward Function Output for a MEO to GEO Mission

CHAPTER 7

PARAMETER VARIATION RESULTS

In the previous chapter, only static missions where a new model per mission was created. In this chapter, missions with changing orbit states are discussed. These more complex problems demonstrates the robustness of this method. The first problem will introduce the inclination change maneuver with a new initial orbit state at each episode. The second problem revisits the generalized orbit change maneuver.

7.1 Variable Inclination Change

The inclination change maneuver keeps all of the orbital elements constant except the inclination. This maneuver can be difficult due to the high Δv requirement, so large changes of inclination are generally avoided. This mission differs slightly from the other two missions presented earlier. Similar to the pendulum test environment described in Appendix A, the initial state will vary for each episode. The parameters of the inclination change maneuvers are given in table 7.1 below,

Table 7.1: Inclination change mission parameters

Orbit	Keplerian		Orbit	Equinoctial	
Parameters	Initial State	Target State	Parameters	Initial State	Target State
a (km)	20000	20000	a (km)	20000	20000
e	0.10	0.10	e_x (rad)	0.939	0.939
i (deg)	13.0 ± 0.3	14.3	e_y (rad)	0.342	0.342
ω (deg)	10.0	10.0	h_x (rad)	0.109-0.115	0.123
Ω (deg)	10.0	10.0	h_y (rad)	0.0193-0.202	0.0217
Duration (day)	4		Duration (day)	4	

During training, only the inclination varies by ± 0.3 degrees with random values based on a normal distribution. By varying the initial orbit state, this allows the policy to become more robust, allowing it to solve a more generalized set of mission parameters.

The same neural network hyperparameters were used in this mission as were used for the orbit raising mission, Table 6.7. The model was trained for 1400 episodes it was determined that the agent to adequately trained. Figures 7.1 and 7.2 below, shows the trajectory of the spacecraft during a successful mission. It can be seen from Figure 7.2, that the satellite reached all the target

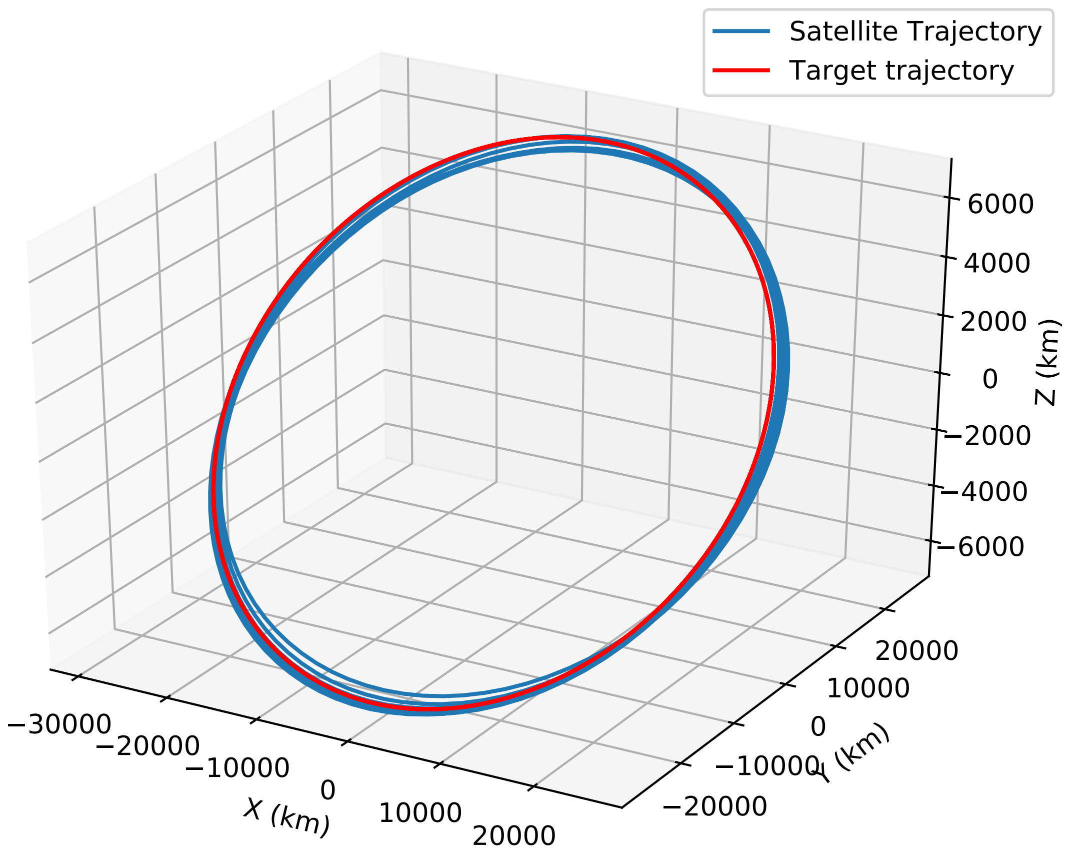

Figure 7.1: Inclination Change Mission Trajectory

states with the specified tolerances. An interesting observation is the behavior of the semimajor axis. The semi-major axis begins to increase during the episode and then decreases to the target state. An efficient way to raise inclination is to increase the semimajor axis and inclination simultaneously and then decrease the semimajor axis; the agent learned this behavior through exploration. Another interesting behavior involves the thrust as shown in Figure 7.3. It can be seen in Figure 7.3, that the agent attempts to reduce the oscillating behavior of the inclination change maneuver by applying a thrust in a normal direction in a step-wise behavior. When performing inclination

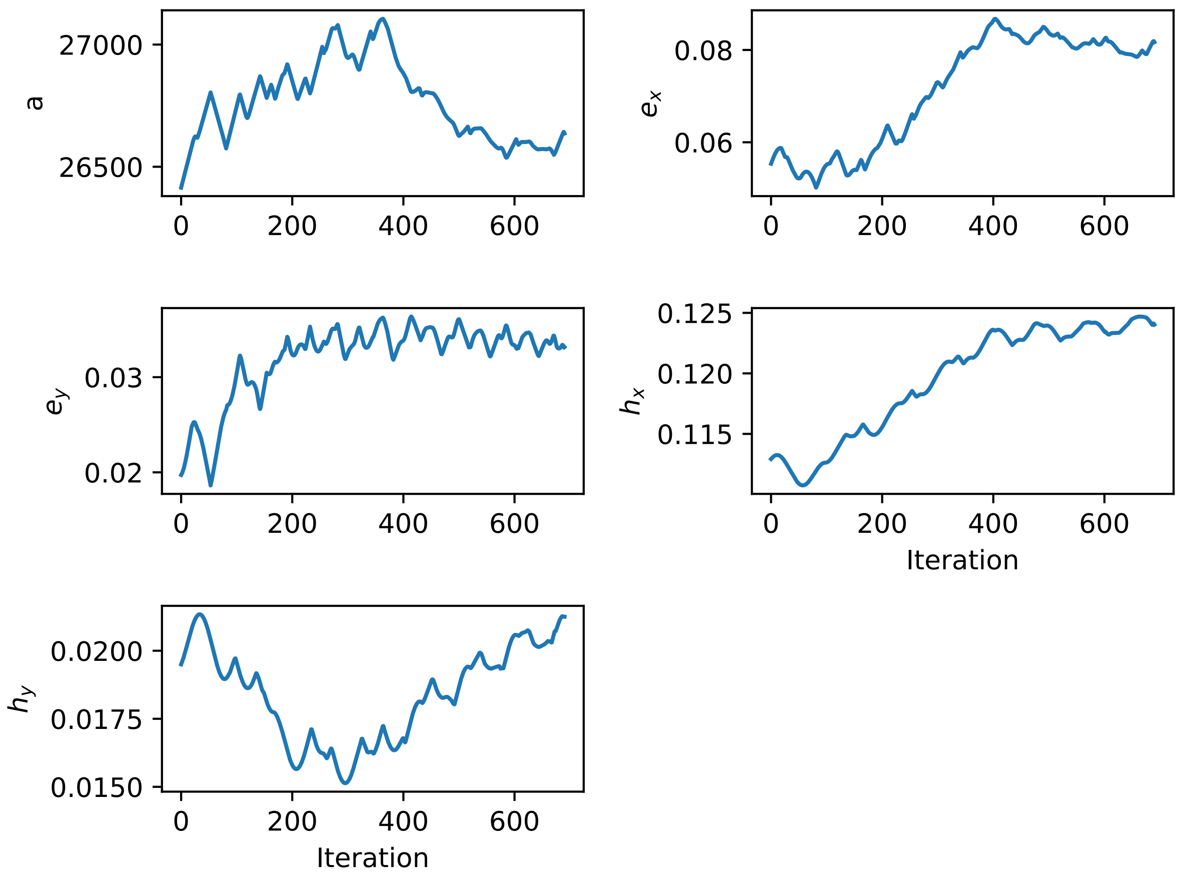

Figure 7.2: Inclination Change Mission

change maneuvers oscillations can be introduced in the orbit change by applying a thrust in the normal direction as shown in Figure 7.4. This ”see-saw” effect may result in more fuel consumption since inclination is decreasing in the tough portion on the oscillation. The reward function used for

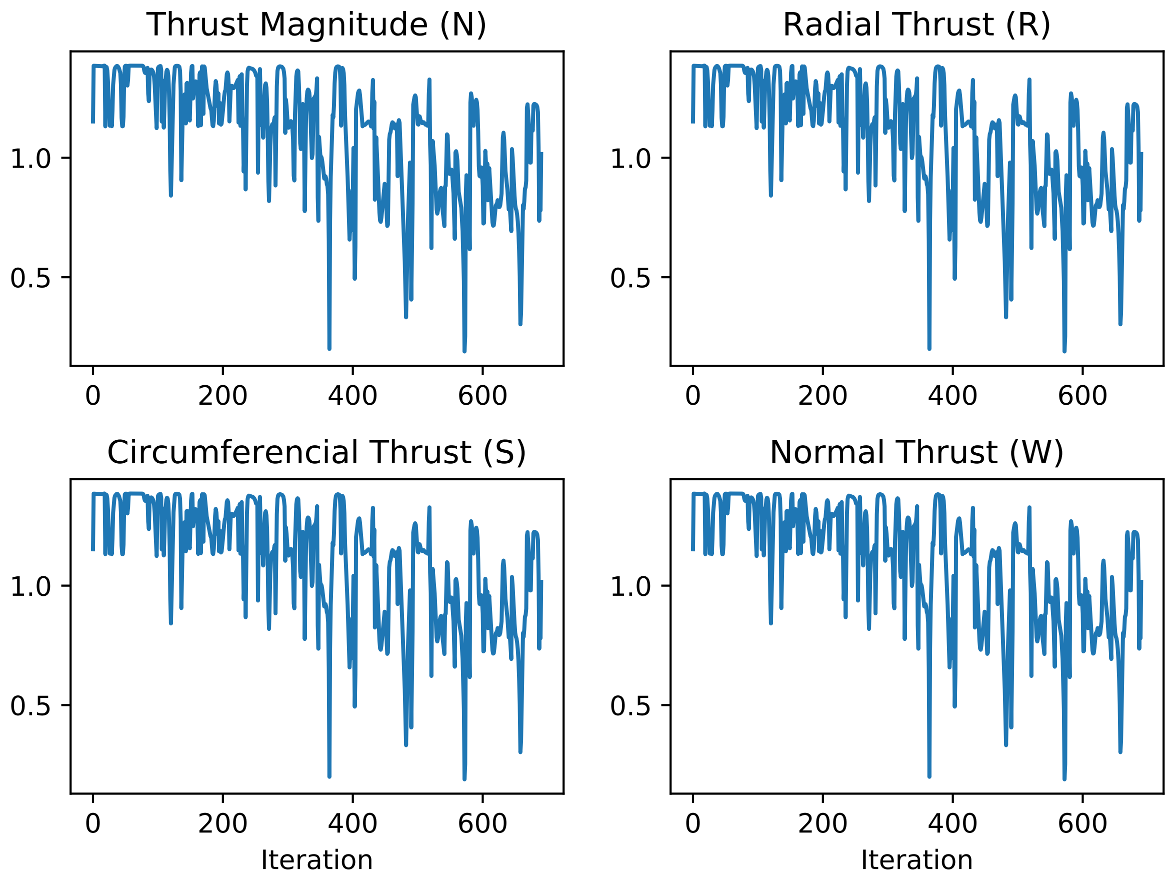

Figure 7.3: Inclination Change Thrust Profile

the inclination change is the same one used for the orbit raising maneuver, Table 6.7. Figure 7.5, shows performance of the agent during training. At approximately episode 700, the agent reaches a maximum reward value which is also the time when the target state was reached consistently.

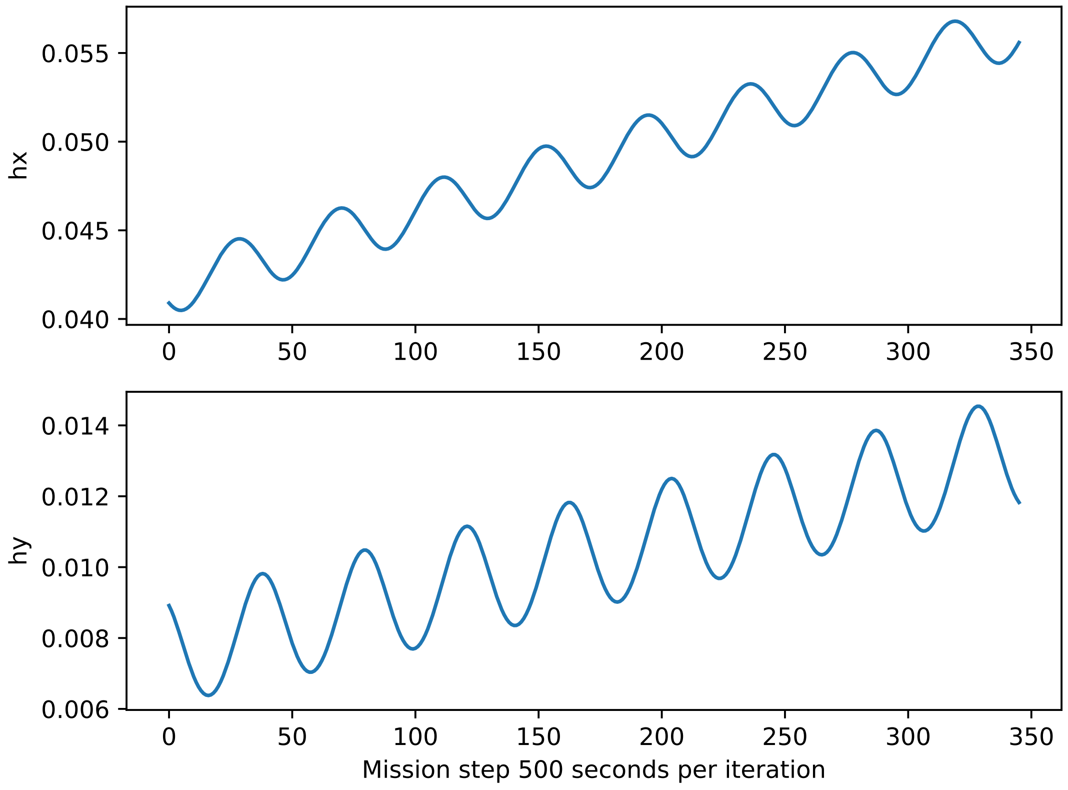

Figure 7.4: Inclination Change Seesaw Effect

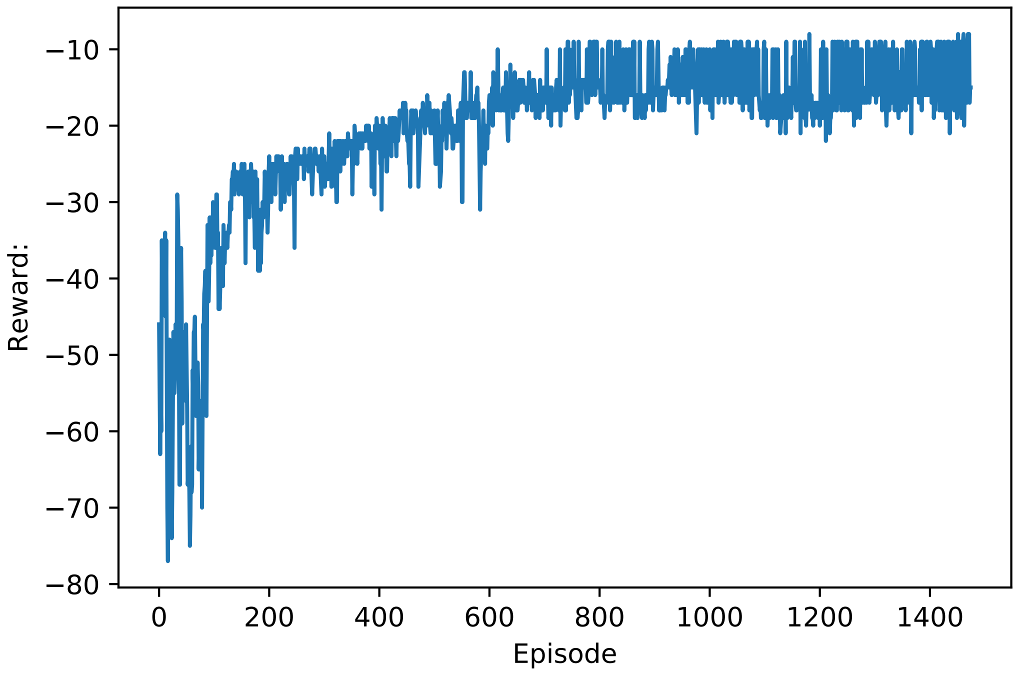

Figure 7.5: Reward Function Output for the Inclination Change Mission

7.2 Variable Generalized Orbit Change

The following section looks at the flexibility of the generalized orbit change model. In the following cases, the trained generalized orbit change model discussed in Section 6.1 is applied to a new set of target states. When the pre-trained actor and critic models are loaded, only the weights and architecture of the actor and critic networks are saved. Table 7.2, show the target states of maneuver. After running the test cases, we can see from Figures 7.6 and 7.7 the agent reaches the

Table 7.2: Generalized orbit change mission

Orbit	Keplerian			Orbit	Equinoctial		
Parameters	Initial State	Test 1	Test 2	Parameters	Initial State	Test 1	Test 2
a (km)	5000	6300	6500	a (km)	5000	6300	6500
e	0.2	0.23	0.24	e_x	0.766	0.615	0.500
i (deg)	5.0	5.4	5.6	e_y	0.642	0.788	0.866
ω (deg)	20.0	26.0	30.0	h_x	0.0410	0.0423	0.0423
Ω (deg)	20.0	26.0	30.0	h_y	0.0149	0.0206	0.0244
Duration (day)	4			Duration (day)	4		

target state for all test cases using the same base model. Referring back to Figure 6.4, the reward function for the base model, it took approximately 350 episodes for the agent to be trained. When looking at the reward function for test case 1 and 2, the agent was trained in about 100 episodes.

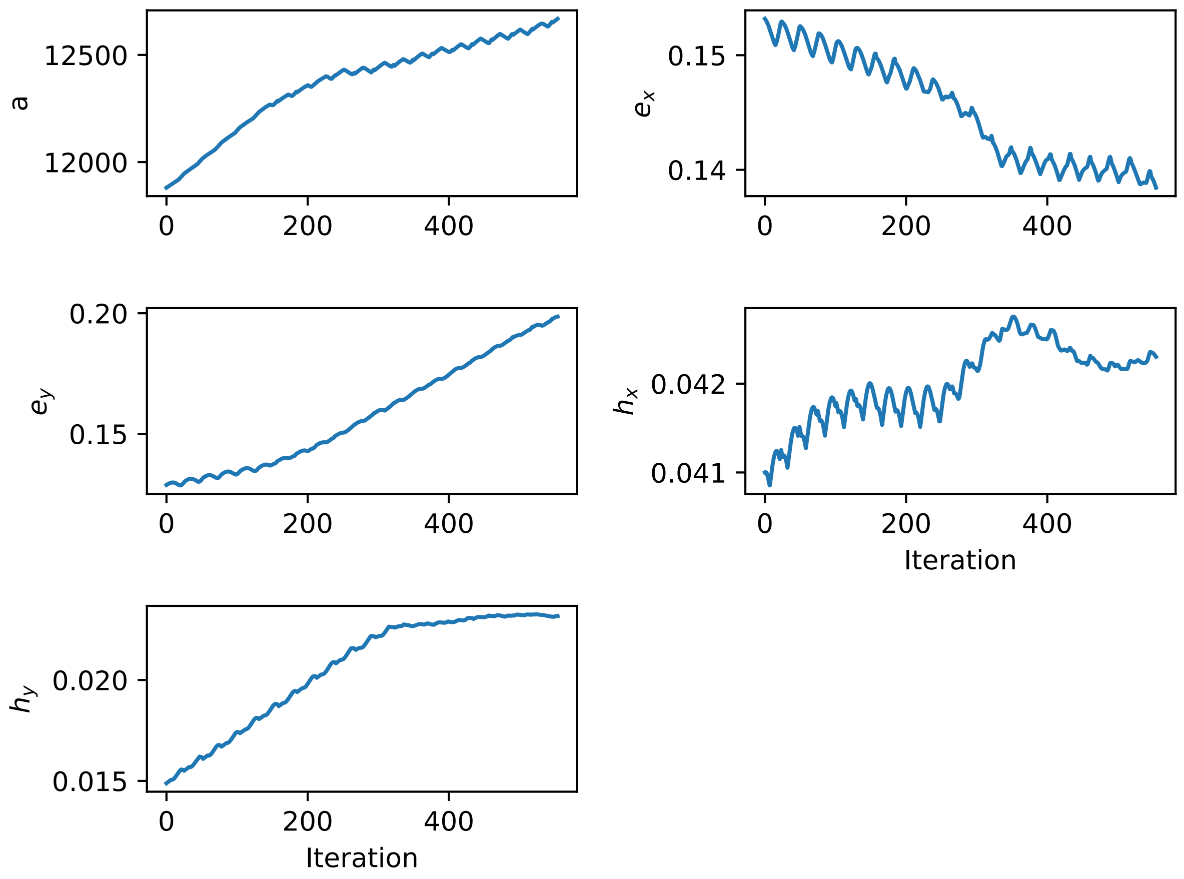

Figure 7.6: General Orbit Change Maneuver Test Case 1

The reward functions for both test cases, Figure 7.8 and Figure 7.9 show that the agents converge to the target states faster than in the initial orbit change maneuver case.

When a third test case was attempted with a higher eccentricity value, the agent was not able to converge to a target state after 500 episodes. Possible methods to hae the model converge for the third test case would be to increase the size of the model or increase the duration.

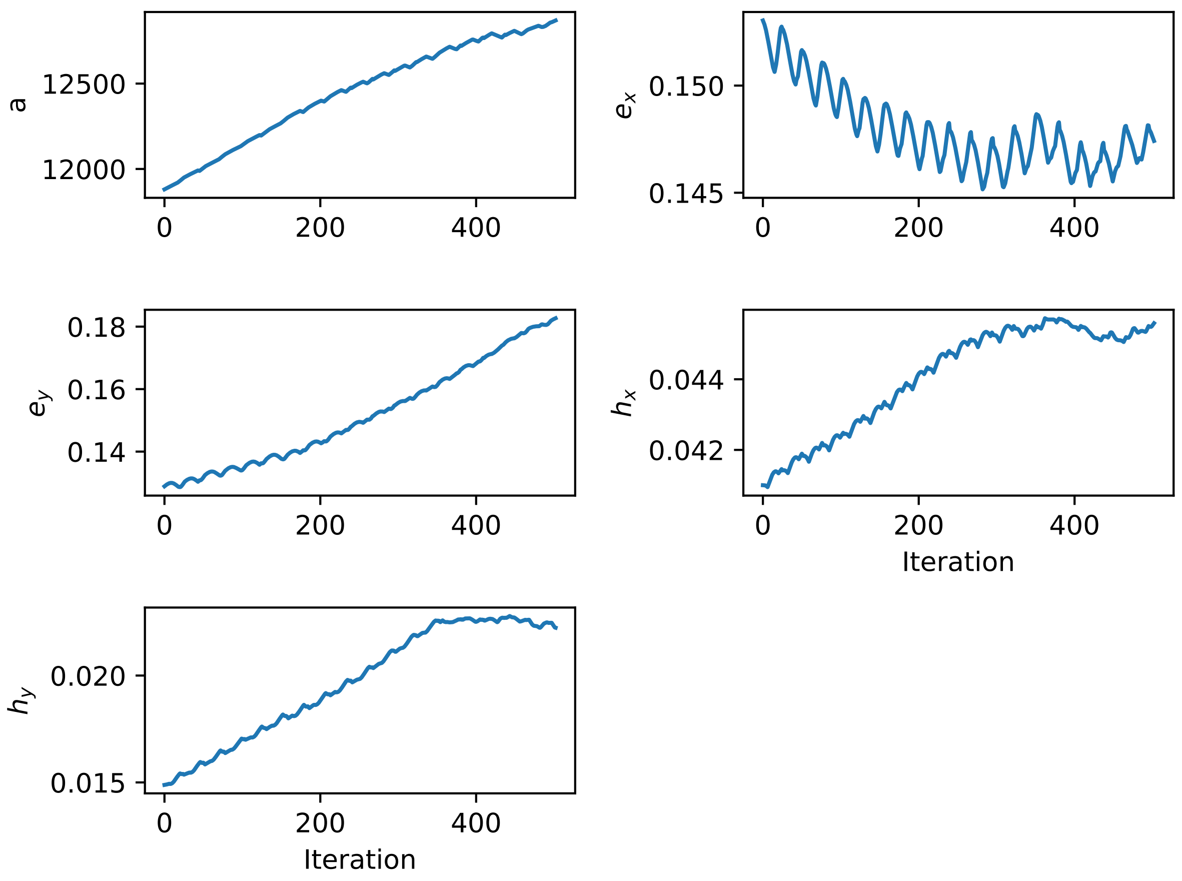

Figure 7.7: General Orbit Change Maneuver Test Case 2

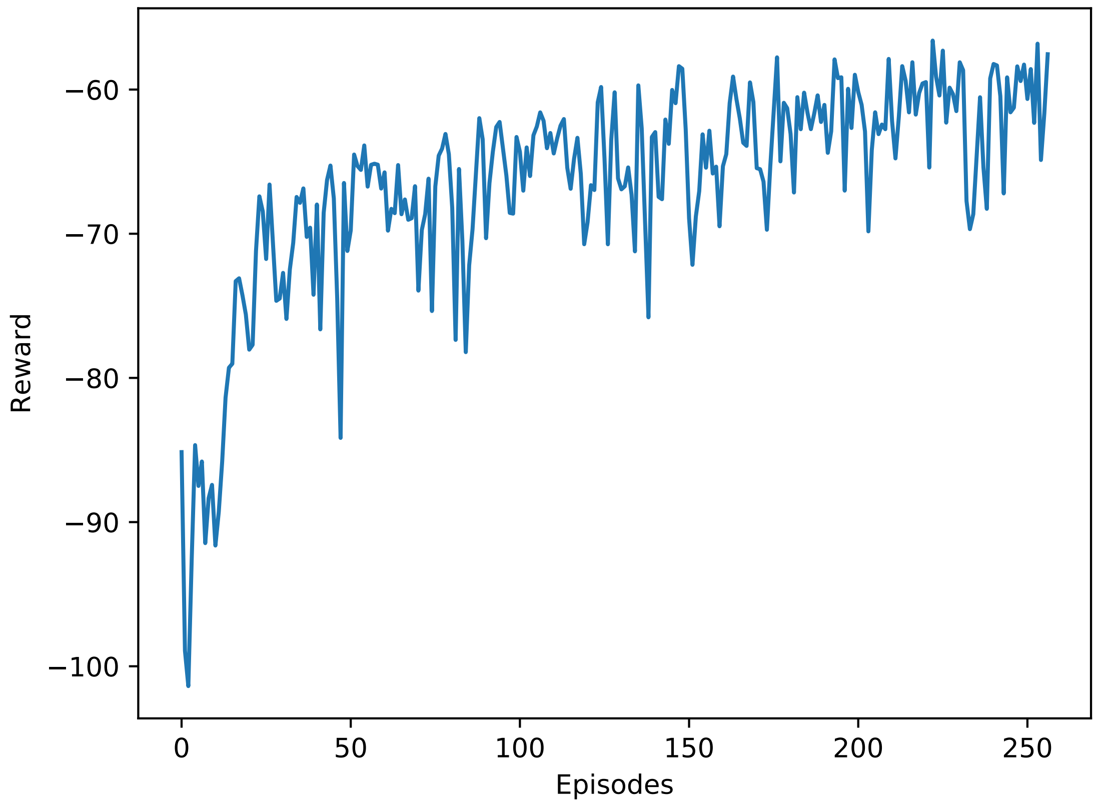

Figure 7.8: Reward Function for General Orbit Change Maneuver Test Case 2

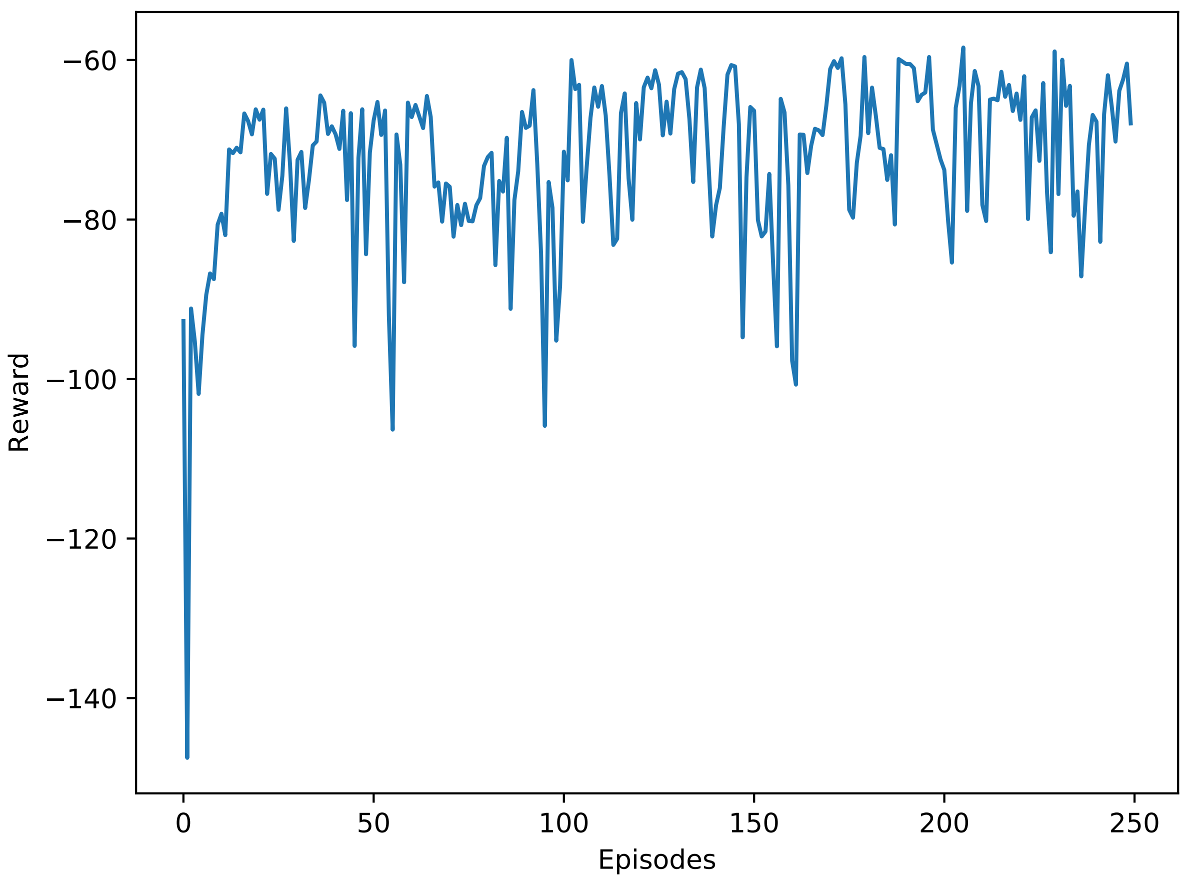

Figure 7.9: Reward for orbit raising Maneuver Test Case 2

CHAPTER 8

CONCLUSION

A novel method was developed to train a reinforcement learning model to control the thrust magnitude and direction of a low-thrust spacecraft. A simulated space environment was created using a low-level space dynamics library to allow the reinforcement learning model to traverse the environment. The reinforcement learning model is based on the actor-critic deep deterministic policy gradient algorithm. An actor-critic model consists of an actor that takes an action in an environment based on the current state agent, and the critic evaluates the actor based on state and action taken by the actor. Both the actor and critic models are represented using feed-forward neural networks with RELU activation functions in the hidden layers, and hyperbolic tangent functions in the output layers.

Three unique low-thrust maneuvers were analyzed: generalized orbit change, semimajor axis change, and inclination change maneuvers. The generalized orbit change maneuver increases all orbital elements to a higher orbit state. The agent was shown to reach a given target within the mission duration time. The same model was used to solve two new generalized orbit change maneuvers. For each new maneuver, the pre-train model was able to adpat two solve each of the two new maneuvers in a smaller number of episodes than the original baseline general orbit change maneuver.

The semimajor axis change mission increased the only the semimajor axis of the orbit. It was shown that it took four days for the agent to change the semimajor axis by 10,000 km. Another model tested the algorithm's capability for an agent to go from a MEO to a GEO orbit. After training the agent, it was successfully able to reach its target state within the allotted mission time.

The final mission demonstrated was the inclination change maneuver. This mission randomly placed the agent in a different initial inclination that is within $\pm$ 0.3 degrees of a baseline orbit state. After training for 1400 episodes, the agent was able to generalize and successfully reach the target orbit from any initial inclination within the given range.

These solutions demonstrated that a single algorithm can be used to solve a variety of trajectory problem with various levels of complexity. A framework using machine learning models and a simulated space environment to solve low-thrust optimization problems was created. Four neural network models were tuned to provide a preliminary low-thrust mission.

Future work in this area can focus on many different applications. One aspect is mission trajectory planning, this algorithm can be used as a preliminary analysis tool. In the same way the trained network was designed to solve generalized inclination change problems in Section 7.1, this approach can be applied to various other missions. This can allow a quick way to generate a successful mission trajectory without having to formulate and solve many unique control optimization problems.

Another future work can explore autonomous real-time control in deep space and interplanetary flight. Due to the vast distances between planets, and the limitations of the speed of light, communication delays may occur. To ensure the spacecraft is moving along the correct trajectory, a pre-trained model can loaded onto flight computer and activate if there are any communication failures or system malfunctions.

Other future work can include exploring novel types of neural network architectures or reinforcement learning algorithms. Recurrent neural networks and Long-short term memory (LSTM) networks store previous weights within the nodes of the network. These types of neural networks work well with long time histories and may perform more complex orbital maneuvers. Distributed Distributional Deep Deterministic Policy Gradient (D4PG) uses multiple parallel actors to generate more data for the replay buffer, calculates multiple steps of the TD-error and uses a distribution parameter for the critic.

REFERENCES

[1] B.J. Wall and B.A. Conway. Shape-based approach to low-thrust rendezvous trajectory design. *Journal of Guidance, Control*, 32, 2009.

[2] Q. Fang, X. Wang, C. Sun, and J. Yuan. A shape-based method for continuous low-thrust trajectory design between circular coplanar orbits. *International Journal of Aerospace Engineering*, 2017.

[3] E. A. Euler. Optimal low-thrust rendezvous control. *AIAA Journal*, 7, 1969.

[4] Ocampo C. Elements of a software system for spacecraft trajeectory optimization. In *Spacecraft Trajectory Optimization*. Cambridge University Press.

[5] Q. Zeng, X. Geng, and C. Wu. Shape-based analytic safe trajectory design for spacecraft equipped with low-thrust engines. *Aerospace Science and Technology*, 2017.

[6] M. Vasile, editor. *A Global Approach to Optimal Space Trajectory Design*, 2003.

[7] D. Yang, B. Xu, and L. Zhang. Optimal low-thrust spiral trajectories using lyapunov-based guidance. *Acta Astronautica*, 2016.

[8] C. Ampatzis and D. Izzo. Machine learning techniques for optimization of objective functions in trajectory optimization. *European Space Agency*, 2017.

[9] D. Izzo, S. Sparague, and D. Tailor. Machine learning and evolutionary techniques for interplanetary trajectory design. *Optimization in Space Engineering*, 2018.

[10] A. Kumar, N. Paul, and M. Omkar, S. Bipedal walking robot using deep deterministic policy gradient. *arXiv:1807*, 2018.

[11] Lingli Y., Xuanya S., Yadong W., and Kaijun Z. Intelligent land-vehicle model transfer trajectory planning method based on deep reinforcement learning. *arXiv:1807*, 2018.

[12] H.D. Curtis. *Orbital Mechanics for Engineering Students*. Elsevier Butterworth-Heinemann, 2 edition, 2005.

[13] Danby J. *Fundamentals of Celestial Mechanics*. Willmann-Bell, Richmond, Virginia, 2 edition, 2003.

[14] Ozawa Y. An analytical solution for multi-revolution transfer trajectory with periodic thrust and non-singular elements. *International Symposium on Space Flight Dynamics*, 2017.

[15] R.S Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *Deepmind Technologies*, 2013.

[17] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. *Google Deepmind*, 2014.

[18] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *Google Deepmind*, 2015.

[19] J.M. Zurada. *Introduction to Artificial Neural Systems*. West Publishing Company, 1992.

[20] Cybenko G. Approximations of superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 1989.

[21] Hairer E., P. Norsett, S., and G. Wanner. *Solving Ordinary Differential Equations I Nonstiff Problems*, volume 1. Springer, 2 edition, 1993.

[22] A. Holmes, S. and E. Featherstone, W. A unified approach to the clenshaw summation and the recursive computation of very high degree and order normalised associated legendre functions. *Journal of Geodesy*, 2002.

[23] J. Ba, J. Kiros, and G. Hinton. Layer normalization. *arxiv:1607*, 2016.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arxiv:1511*, 2015.

[25] Uhlenbeck G. E. and Ornstein L. S. On the theory of the brownian motion. *Physical review*, 1930.

[26] NASA Glenn Research Center. Magnetoplasmadynamic thrusters. *NASA Facts*, 2013.

APPENDIX

Reinforcement Learning Testing Environments

.1 Pendulum Environment

The pendulum environment consists of a swinging pendulum that starts at random initial positions as shown in 1.

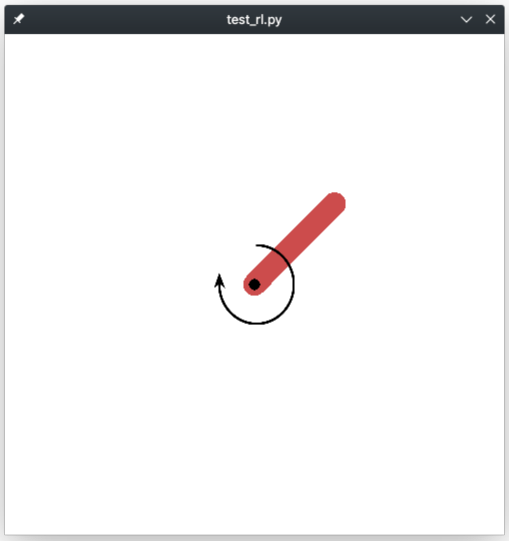

Figure 1: Pendulum Environment

The goal is for the agent to apply a continuous torque value until the pendulum is at a 90 degree angle with no angular velocity. The agent is penalized until it either achieves the goal or reaches the maximum number or steps. This environment can be expressed as the following optimization problem:

Given a pendulum that is allowed to freely move subject to:

$$\dot{\theta}_{t+1} = \dot{\theta} + \frac{-3g}{2l}\sin(\theta + \pi) + \frac{3}{mL^2T}dt$$

$$\theta_{t+1} = \theta_t + \dot{\theta}dt$$

where g is the gravitational force, T is the input torque, L and m are the length and mass of the pendulum, respectively.

Find a control control law that maximizes the reward function:

$$r = \bar{\theta}^2 + .1\dot{\theta}^2 + .001u^2$$

$$\bar{\theta} = \frac{\theta + \pi}{2\pi} - \pi$$

where $\bar{\theta}$ is the normalized angle. This environment with the algorithm under these hyperparameters.

Table 1: Neural Network parameters

Hyperparameters	
layer 1	128
layer 2	80
learning rate	0.001
τ	0.01

The pendulum environment ran for 200 episodes, the reward function output over the training time is shown in the plot below.

.2 Lunar Lander

The second test environment is the lunar lander continuous. In this environment the agent must land the lunar lander in between the two flagpoles with almost zero vertical velocity. The lander is constrained in two dimensions, x and y , and can move the thruster at an angle and thrust magnitude.

The agent receives a high negative reward if it crashes into the ground, a positive reward if lands on the ground without going too fast or if one of the two legs touches the ground, and a high positive reward if the successfully lands in between the flags. The lander is also penalized based on the time it takes to land and applying too much thrust simulating fuel loss. This agent uses reward shaping where the reward is based on a function of the position and velocity of the lander.

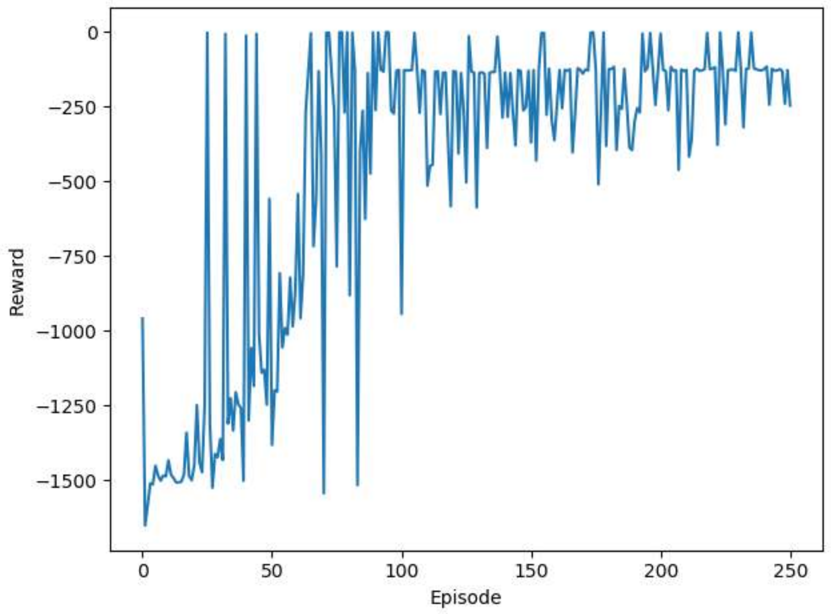

Figure 2: Pendulum Reward Output

Given a lunar lander whose dynamics are subject to:

$$F = m\ddot{x}$$

Find a control law that maximizes the reward function:

$$r = -100\sqrt{x^2+y^2} - 100\sqrt{\dot{x}^2+\dot{y}^2} - 100|\theta| + 10leg_1 + 10leg_2 - 0.3u$$

where θ is the orientation angle of the spacecraft, and leg_1 and leg_2 are either 1 or 0 if the lander's legs touch the ground or not, respectively. The dynamics of the environment are dictated by Newton's 2nd law of motion.

The reinforcement learning algorithm was tested using the hyperparamters shown in Table 2.

The algorithm ran for 500 episodes and the values of the reward function are shown in Figure 4. The results demonstrate that the algorithm is capable of learning a targeting problem given an initial state and a target state, where the target state must meet velocity requirements i.e. the agent can't crash.

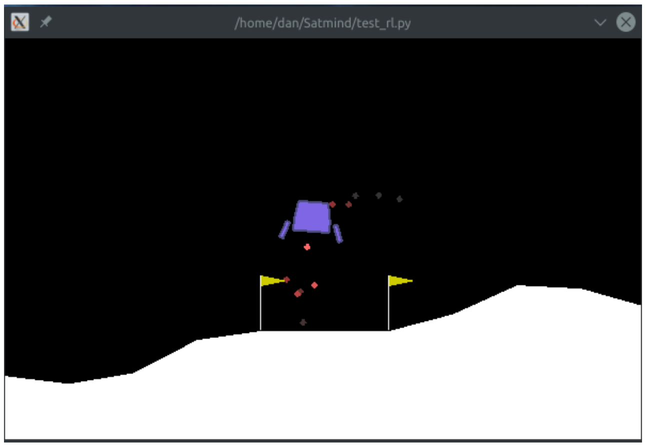

Figure 3: Lunar Lander Environment

Table 2: Pendulum neural network hyperparameters

Hyperparameters	
layer 1	300
layer 2	250
learning rate	0.0001
τ	0.01

.3 Biped Walker

The most complex of the three test environments is the biped walker. In this environment, a two-legged robot must traverse the ground without falling over as shown in Figure 5 below.

The agent has 24 states, the angles and velocities of all the joints, and the body, ground contact, and lidar information. There are four possible actions, the torque values of of the two hip and knee joints. The agent is penalized for the body touching the ground and applying motor torque, and it is rewarded for walking across the environment. The environment episode ends when the agent falls of when the end of the environment is reached. The reward function for the biped walker is given in Equation 1

$$r = \frac{130x}{SCALE} - 5.0|\theta| - 0.00035T_{max}u \qquad (1)$$

where $SCALE$ is the scaling factor of how fast the environment runs, x is the horizontal distance traveled, T_{max} is the maximum input torque allowed, u is the applied torque, and θ is the

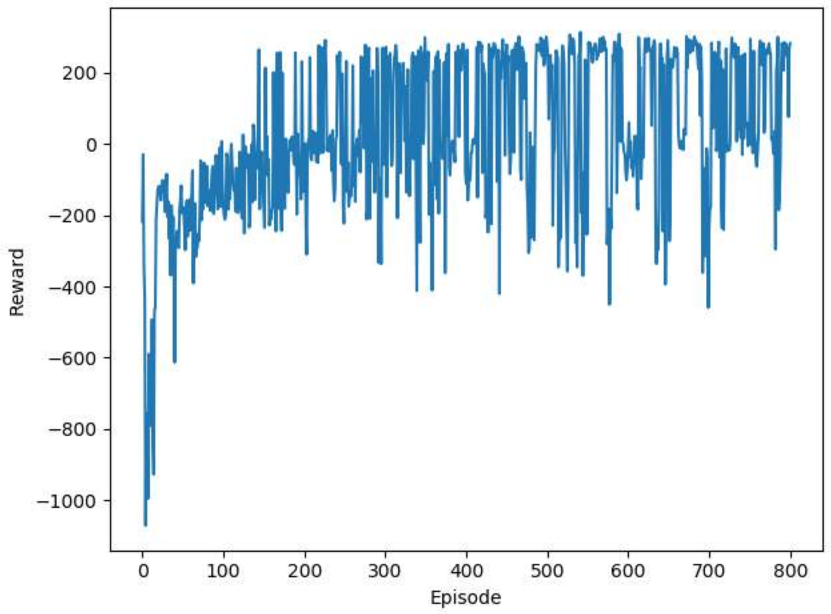

Figure 4: Lunar Lander Reward Output

angle of the head position, encouraging the agent to maintain a head position parallel with the ground. The hyperparamters for the reinforcement learning algorithm are shown in Table 3.

Table 3: Lunar Lander network parameters

Hyperparameters	
layer 1	500
layer 2	450
learning rate	0.001
τ	0.01

The number of layers in each of the hidden layers have increased dramatically compared to the other two environments. A larger number of nodes per layers allows for the neural networks to better capture the dynamics but it comes at a cost of computation speed and increased statistical variance.

After running the algorithm for 800 episodes, the results of the reward function are shown in Figure 6. It can be seen from Figure 6, that the agent requires a significantly longer period of the time to successfully navigate the environment compared to the previous two environments. When

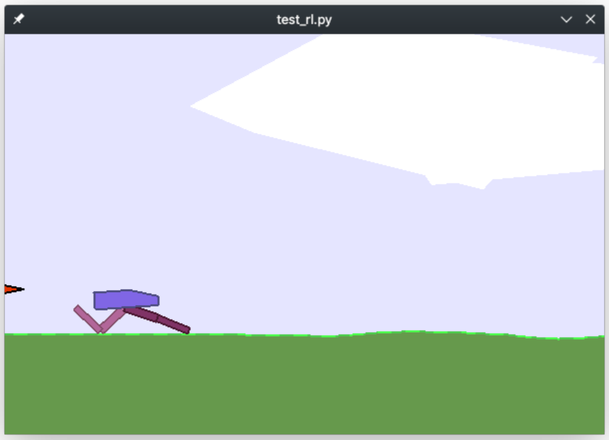

Figure 5: Biped Walker Environment

inspecting the agent during training, it was observed that the agent was walking successfully but would gain too much momentum and fall over, or would attempt to conserve the applied torque by hopping on one leg.

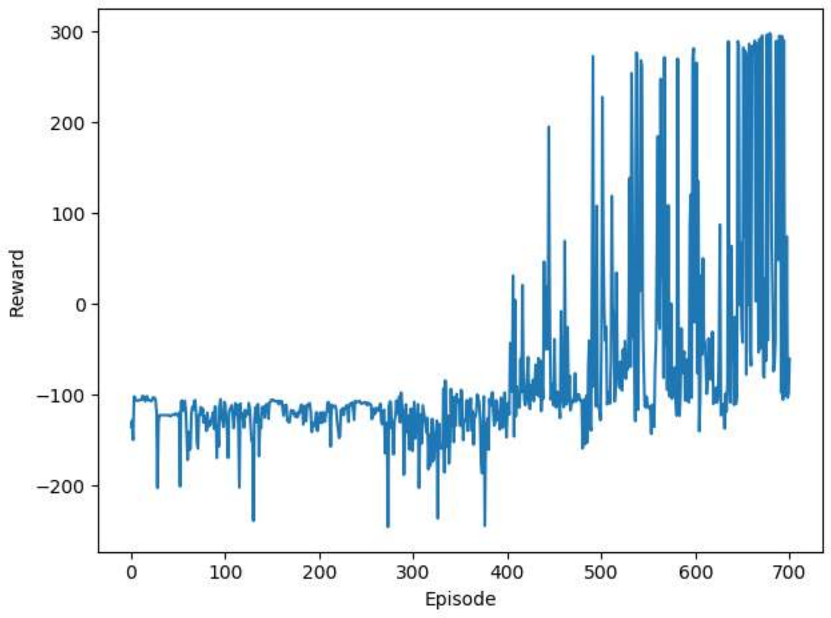

Figure 6: Biped Walker Reward Output