Deep Reinforcement Learning for Robust Spacecraft Rendezvous Guidance

Courtney Bashnick
Mechanical & Aerospace Engineering
Carleton University
Ottawa, Canada
courtneybashnick@cmail.carleton.ca

Lakshay Arora
Mechanical & Aerospace Engineering
Carleton University
Ottawa, Canada
lakshayarora@cmail.carleton.ca

***Abstract*—This paper uses reinforcement learning to train a deep neural network to act as the guidance strategy for spacecraft proximity operations. Specifically, the neural network takes the state error of the spacecraft as input and outputs velocity commands. An actor-critic training scheme is implemented, similar to the reference paper that first proposed this problem, where the actor network is responsible for mapping state errors to velocity commands, and the critic network is responsible for assessing the value of state-action pairs and outputs the probability of its reward falling within a specified range. The scenario under consideration is the pose tracking and docking of a chaser spacecraft to a stationary target spacecraft. After several hours of training, the policy network learns to reduce the chaser spacecraft error from the desired positions. The episodic rewards, losses, and training trajectories of the trained network are compared to those of the reference paper.**

I. Introduction

Autonomous spacecraft missions are enabled by the guidance, navigation, and control system software responsible for planning and executing a safe, feasible trajectory to a targeted location. Rendezvous missions consist of a "chaser" spacecraft that actively maneuvers to a functional spacecraft, artificial satellite, or space debris considered the "target." In such space missions it is additionally beneficial to limit the maneuver time or the amount of fuel consumed by the chaser spacecraft and so an optimization problem is posed.

Typically, the optimization problem consists of nonlinear orbital dynamics and constraint equations that create a nonlinear programming problem that is computationally expensive to solve. A longer computation time limits the real-time implementability of the guidance algorithm and has negative effects on the robustness and stability of the solution. Furthermore, as more complex missions are designed (e.g., detumbling spinning space debris with a net or stabilizing a tumbling object with a manipulator to minimize angular momentum) it may be infeasible for an engineer to design an accurate system model using traditional methods. This is especially true when the environment, dynamics, or disturbances are time-varying.

Artificial intelligence (AI), specifically machine learning and artificial neural networks, can be utilized to solve these complex space problems.

II. Machine Learning in Aerospace Guidance and Control

The application of AI, and more specifically deep learning, has proven to be an efficient and successful method for solving control problems in a variety of fields of study, most notably robotics. In astrodynamics, particularly spacecraft guidance, a deep neural network (DNN) is served as a parametric model that maps observations of the spacecraft state to control actions that specify the amplitude and direction of the required force. DNNs have been used in both supervised- and reinforcement-learning-based architectures for spacecraft guidance.

A survey on the recent developments of AI in spacecraft guidance and control can be found in Izzo et al. [1], which focuses on evolutionary optimization, tree searches, and machine learning including deep learning and reinforcement learning. A survey on machine learning techniques in spacecraft control and design can be found in Shirobokov et al. [2], particularly focusing on stochastic and deterministic methods for supervised learning as well as direct value-based approaches for reinforcement learning.

A. Supervised Learning

To train neural networks via supervised learning, a dataset must be obtained. In the context of optimal spacecraft trajectory control, indirect optimization algorithms are used to compute these optimal spacecraft trajectories using stochastic methods, e.g., starting from randomized initial conditions.

Cheng et al. [3] train three DNNs offline using the state-action pair solutions of trajectory control generated by solving an optimal control problem for orbital transfer maneuvers using an indirect method. The three neural networks for the costates, flight time, and control variables cooperate to generate real-time optimal guidance commands.

Similarly, Yin et al. [4] use DNNs trained via a dataset from an indirect method to provide intelligent initial guesses for solving an orbital transfer mission. It is well-known that indirect optimization methods are particularly susceptible to the chosen initial guess for the solution. The authors train a DNN using samples from the indirect method and the trained DNN is initially used to predict optimal control actions. Afterwards, the output parameters from the DNN are used as optimized initial guesses for the indirect method to produce even further optimal solutions.

Supervised learning methods for spacecraft guidance are not very popular due to limitations associated with the generated training dataset. Reinforcement learning, as described below, is used much more extensively.

B. Reinforcement Learning and Deep Reinforcement Learning

Reinforcement Learning (RL) is a sub-discipline of machine learning whereby an implicit environmental model is developed using repeated simulations together with a reward function. The goal is to determine an optimal policy that achieves the maximum reward value. The classification of reinforcement techniques into policy- and value-based methods, i.e., primarily seeking the optimal policy or optimal value function, is equivalent to the classification of direct and indirect methods in optimal control theory [2], [5], [6].

The policy that maps input states to control actions can be modelled as a DNN, as can the value function that maps the states to rewards. When a DNN is trained via RL, this is called Deep Reinforcement Learning (DRL). The DRL framework has been used in a variety of applications including gameplay [7], [8], autonomous wild-fire monitoring aircraft [9], spacecraft mapping missions [10], and optimal thrust control for lunar landing [11].

A commonly used algorithm to train DNNs is Proximal Policy Optimization (PPO) [12]. PPO is considered state-of-the-art for solving continuous control problems within RL and has been used for a wide range of spacecraft guidance missions including cislunar transfers [13], planetary soft-landing maneuvers [14], and rendezvous maneuvers [15]–[17]. PPO must be described in the context of Trust Region Policy Optimization (TRPO) [18]. TRPO was introduced to improve training stability by ensuring the updated policy is within a certain region ("trust region") of the previous policy. It does so by implementing a hard constraint in the optimization problem that limits Kullback–Leibler (KL) divergence or relative entropy, i.e., a measure of the difference between probability distributions, in this case, the old and updated policies. The algorithm alternates between sampling information from the environment and optimizing a surrogate objective function using a stochastic gradient ascent.

PPO simplifies the TRPO algorithm. It modifies the optimization problem in the TRPO method by either (1) introducing a clipped surrogate objective to constrain the ratio of the old and new policy using a hyperparameter, or (2) using an adaptive KL penalty term in the objective function [12].

PPO is used by Gaudet et al. [14] to train a DNN applied to the guidance and control of planetary landing missions with six-degrees-of-freedom. The authors use the reward function to impose constraints on the lander's attitude and thrust during the powered descent whilst encouraging the agent to minimize the required thrust and errors in the terminal position, attitude, and velocity. The Adam optimizer [19] is used in the training of the policy and value functions employing an adaptive learning rate.

Broida and Linares [15] use PPO in the context of RL for spacecraft rendezvous and docking. The policy is applied to three-degree-of-freedom closed-loop docking in an R-bar and V-bar approach given in the local-vertical-local-horizontal frame. Both the policy and value neural networks consist of three hidden layers and the system is trained for 200,000 episodes taking about 24 hours of wall-clock time. Simulations show robustness of the trained model when started in a locus of uncertainty.

A feedback control law for six-degree-of-freedom spacecraft rendezvous and docking maneuvers is presented by Oestreich et al. [16]. The authors use the PPO method to train standard, feedforward neural networks. The training itself is initialized with a certain variance in the policy's probability distribution to encourage exploration of the action space. As the model training progresses, the variance is decreased to encourage more exploitation of the policy. After training is complete (600,000 episodes), the variance is set to zero, which results in a deterministic feedback control law for implementation. The performance of the trained network is compared to the solution of the optimal control problem solved using the GPOPS-II software [20] to show that the RL-based policy achieves comparable, although sub-optimal, results.

As an alternative to using PPO, Hovell and Ulrich [21]–[23] use the distributed distributional deep deterministic policy gradient (D4PG) [24] framework to train a spacecraft rendezvous and docking guidance strategy. The D4PG algorithm is an actor-critic algorithm, which is a combination of a policy-based and value-based approach: The "actor" is a neural network that drives the policy (i.e., decides the action that should be taken) and is updated based on a policy gradient approach from a "critic" neural network that critiques the policy (i.e., informs the actor on how good its taken action was and how to adjust) with its own trainable weights.

For their guidance algorithm, Hovell and Ulrich [21]–[23] use two hidden layers in each of their policy and value networks. After every five training episodes, the policy is deployed to an external dynamics simulation for evaluation, The policy is trained to generate the desired spacecraft velocity commands for proximity operations that are subsequently sent to a conventional controller to track. This guidance algorithm is capable of real-time implementation and was experimentally validated on a granite table testbed for docking and berthing between a chaser and target spacecraft in three-degrees-of-freedom under planar dynamics. Furthermore, experiments [23] prove the robustness of the algorithm to perturbations to both the chaser and the target states in the form of manual interference.

The performance of supervised- and reinforcement-learning-based algorithms for terminal rendezvous missions are compared directly in Federici et al. [17] whereby a DNN is trained using a dataset computed using a second-order cone programming formulation ("behavioural cloning") with scattered initial conditions and compared to a DNN trained in an RL framework using PPO. In both approaches, the training is organized in batches to improve speed and the parameters of the DNN (with three hidden layers) are optimized using stochastic gradient descent. The authors note that the training of the behavioural cloning approach took 45 minutes whereas the training of the RL approach took about 7 hours on the same hardware. Overall, the authors found that the RL approach achieved a network that was more robust to off-nominal initial conditions as compared to the policy trained using supervised learning.

III. Problem Statement

This work aims to use reinforcement learning to produce a robust guidance algorithm for spacecraft rendezvous and docking operations. Specifically, the authors will utilize DRL and reproduce the results of Ref. [22] for pose tracking and docking with a static target spacecraft.

The objective of any rendezvous problem is to nullify the relative separation and velocity of the chaser and target spacecraft whilst considering operational limitations (e.g., maximum thrust limits) and safety requirements (e.g., collision avoidance). The spacecraft guidance problem for rendezvous can be rendered into a reinforcement learning framework quite elegantly through training a policy that is rewarded by reducing the state error of the spacecraft and punished for large control efforts and for collisions between the spacecraft and other objects.

The scenario considered is shown in Fig. 1. Over the course of the mission, the chaser is set to maneuver from its initial position (at rest) to a desired state: for the first part of the mission, the desired state is the holding point, directly in front of the docking mechanism of the target spacecraft; afterward, it is switched to the docking point such that the chaser and target spacecraft make contact.

To complete this mission, a guidance strategy [22] as given in Fig. 2 is proposed. Here, the guidance algorithm called "Deep Guidance" is trained to output velocity commands $\mathbf{v}_t$ based on the current state of the chaser spacecraft $\mathbf{x}_t$ (particularly, its error $\mathbf{e}_t$ from the desired state $\mathbf{x}_d$). The process repeats every sampling instant t whereby the new current state error of the chaser, $\mathbf{e}_t \leftarrow \mathbf{e}_{t+1}$, is fed into the network to generate the next set of velocity commands.

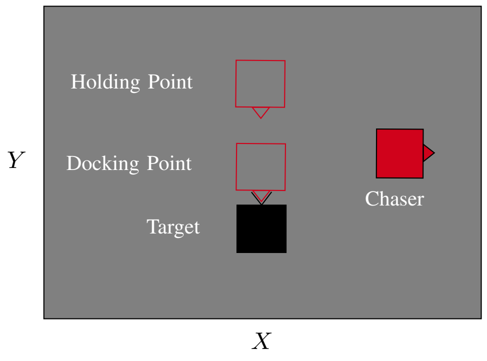


Fig. 1: Pose tracking and docking task of a spacecraft. Adapted from [22].

In this work, an ideal controller is assumed, which guarantees that the guidance model does not overfit to any particular controller. Consequently, the dynamics model and the ideal controller blocks from Fig. 2 may be merged into a single kinematics model as shown in Fig. 3 because the ideal controller gives the dynamics exactly what it needs to achieve the required velocity $\mathbf{v}_t$. As a result, the output velocity commands from the guidance algorithm can be directly integrated (using, e.g., Scipy's integrate sub-package[1]) to achieve the next state $\mathbf{x}_{t+1}$.

Assuming an ideal controller is not realistic, and so there is likely to be performance degradation when the model is exposed to an actual (non-ideal) controller. To circumvent this issue, Ref. [22] introduces Gaussian noise to the kinematics output while training the model, which thereby allows the system to achieve *unseen* states–an expected result from using a non-ideal controller. For simplicity and to reduce training time, this work does not consider noise on the kinematics.

Once trained, Ref. [22] assumes a non-ideal controller to generate a control effort $\mathbf{u}_t$, e.g., a simple proportional velocity controller with gain K_p,

$$\mathbf{u}_t = K_p(\mathbf{v}_t - \dot{\mathbf{x}}_t), \tag{1}$$

that is input to the dynamics of the spacecraft rendezvous problem to propagate the chaser state. The Clohessy-Wiltshire (CW) equations [25] model the relative motion of spacecraft under small relative separations in a circular target orbit. Under the assumption that the rendezvous period is short compared to the orbital period, the CW equations are reduced to double-integrator planar dynamics for two-degree-of-freedom problems, typically considered in the x- and y-axes. An additional degree-of-freedom in ψ can be added to consider the double-

[1]https://docs.scipy.org/doc/scipy/tutorial/integrate.html

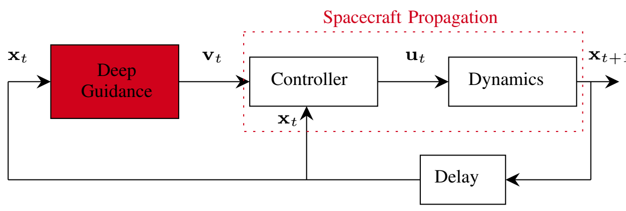


Fig. 2: Guidance and control loop using "Deep Guidance". Adapted from [22].

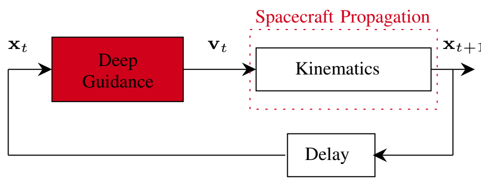


Fig. 3: Guidance and control loop with an ideal controller. Adapted from [22].

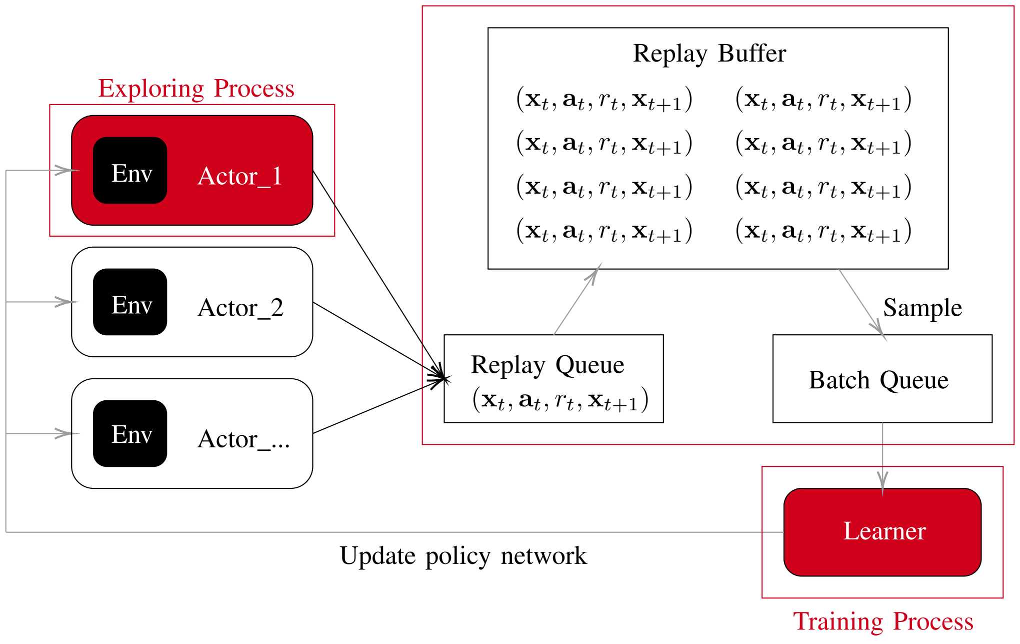


Fig. 4: Visualization of the D4PG process.

integrator rotational motion of the chaser spacecraft about the z-axis:

$$\begin{cases} \ddot{x} &= \frac{u_x}{m} \\ \ddot{y} &= \frac{u_y}{m} \\ \ddot{\psi} &= \frac{\tau}{I_z} \end{cases} \tag{2}$$

Here, m denotes the mass of the chaser platform, I_z is the moment of inertia about the vertical axis, and u_x, u_y, and τ are the control forces and torque, respectively. For simplicity and to reduce training time, this work assumes an ideal controller for testing the trained network and therefore uses the kinematics equations to propagate the chaser state instead of the spacecraft dynamics equations.

IV. Methodology

A. Framework Overview

As previously described, the Deep Guidance strategy in [22] is built in TensorFlow around the D4PG algorithm [24] which at its core consists of an actor-critic framework: a policy (actor) network outputs the actions of the spacecraft based on state information, and the value (critic) network aids in the training of the policy network by estimating the reward that state-action pairs generate. Specifically, and as pictured in Fig. 4, the D4PG algorithm uses many "actors" in parallel that generate reward information for state-actions. The data is saved in a "replay buffer," which is sampled in batches by a "learner" that updates the weights of both the policy and value networks. The specific training algorithm as described in [21], [22], [24] is given in Algorithm 1 and is elaborated upon in the remainder of this section.

B. Initializing the Networks

The D4PG algorithm uses four networks: two *main* policy and value networks, and two *target* policy and value networks that use a slower, smoother parameter update to reduce oscillations during training and improve the training stability.

A general policy π_θ has trainable weights θ that map states to actions. Specifically for Deep Guidance, the policy network takes as input the system state error $\mathbf{e}_t \triangleq \mathbf{x}_d - \mathbf{x}_t = [e_x \ \ e_y \ \ e_\psi]^T$ and outputs as the commanded action the velocity $\mathbf{v}_t = \begin{bmatrix}\dot{x} & \dot{y} & \dot{\psi}\end{bmatrix}^T$ of the chaser spacecraft, i.e., $\mathbf{v}_t = \pi_\theta(\mathbf{e}_t)$.

A value network $Z_\phi(\mathbf{e}, \mathbf{a})$ has trainable weights ϕ that map the state-action pairs to a probability distribution of the expected rewards for the remainder of the simulation (episode). In [22], the distribution is separated into $B = 51$ evenly spaced bins along the empirically determined interval [-1000, 100].

The main weights θ and ϕ are randomly initialized at the start of training, and are used to initialize the target network weights, θ' and ϕ', which are updated more slowly than the regular network weights. An exponential moving average of the main network weights is used to update the smoothed weights

$$\theta' = (1-\epsilon)\theta' + \epsilon\theta \tag{3}$$
$$\phi' = (1-\epsilon)\phi' + \epsilon\phi \tag{4}$$

where $\epsilon = 0.001 \ll 1$.

The architecture for both the policy and value networks is given in Table I and visualized in Figs. 5 and 6. Notably, each network has two hidden layers using the Rectified Linear Unit (ReLu) activation function. The output of the policy network is cast using a hyperbolic tangent to ensure a bounded velocity.

Algorithm 1: Training Deep Guidance using D4PG [21], [24]

```
Data: Batch size $M$, trajectory length $N$, number of actors $K$, replay buffer size $R$, exploration rate $\sigma$, discount factor
      $\gamma$, policy and value network learning rates $\alpha$ and $\beta$
Learner
1 Randomly initialize policy and value network weights, $\theta$ and $\phi$
2 Initialize smoothed policy and target value network weights, $\theta' = \theta$ and $\phi' = \phi$
3 Launch $K$ actors and copy policy weights $\theta$ to each actor
  repeat
4     Sample a batch of $M$ transitions $(\mathbf{e}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ from the replay buffer
5     Compute the target value distribution from Eq. (10), which is used to train the value network
6     Update the value network weights $\phi$ by minimizing the loss function in Eq. (9) using learning rate $\beta$
7     Compute the policy gradients using Eq. (13) and update policy weights using Eq. (14)
8     Update the smoothed network weights slowly using Eqs. (3) and (4)
  until acceptable performance
Actor
  repeat
1     For each episode $E$, obtain the most up-to-date version of the policy from the Learner and reset the environment
      for each timestep $t$ within episode do
2         Sample an action from the policy and add exploration noise via Eq. (6), i.e., obtain $\mathbf{a}_t$ via Eq. (5)
3         Step environment forward one timestep using the action $\mathbf{a}_t$; observe the reward $r_t$ and next state $\mathbf{e}_{t+1}$
4         Record $(\mathbf{e}_t, \mathbf{a}_t, r_t = \sum_{n=0}^{N-1} \gamma^n r_n, \mathbf{e}_{t+1})$ and store in the replay buffer
      end
  until acceptable performance
```

TABLE I: Structure of the policy and value networks

	Policy Network		Value Network	
Layer	Neurons	Activation	Neurons	Activation
Layer 1 (Input)	3	–	6	–
Layer 2 (Hidden)	400	ReLu	400	ReLu
Layer 3 (Hidden)	300	ReLu	300	ReLu
Layer 4 (Output)	3	tanh	B	softmax

The output layer of the value network uses the softmax function to produce a probability distribution.

C. Filling a Replay Buffer

In the D4PG framework, the data used to train the policy and value networks is generated by K actors that run the most up-to-date policy in separate threads from the main training program, i.e., parallel programming. Each actor repeatedly runs episodes whereby actions at the current state are generated by adding the output of the policy network with a term that promotes exploration of the environment [22],

$$\mathbf{a}_t = \pi_\theta(\mathbf{e}_t) + \mathcal{N}(0,1)\sigma \tag{5}$$

where $\mathcal{N}(0,1)$ is the standard normal distribution and σ is the noise standard deviation. In [21], the exploration noise is based on the limits of the velocity and exponentially decays in value as the episode number E increases

$$\sigma = \frac{1}{3}\left[\max(\mathbf{v}) - \min(\mathbf{v})\right](0.9997)^E, \tag{6}$$

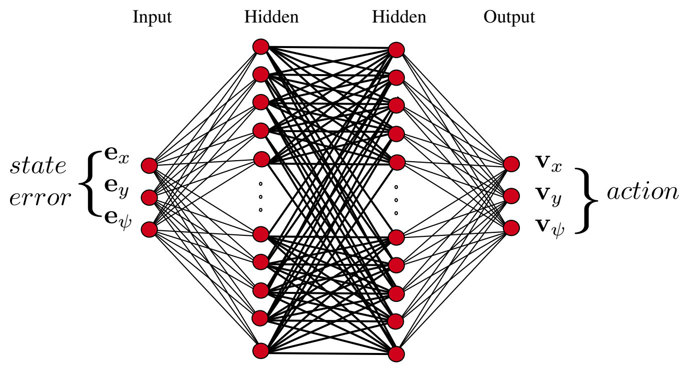


Fig. 5: Policy network architecture.

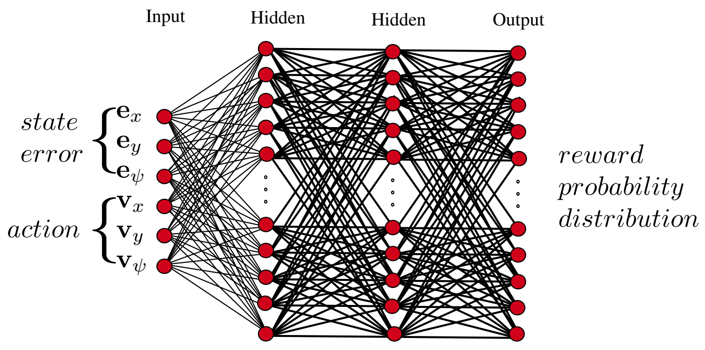


Fig. 6: Value network architecture.

thereby reducing the exploration of the environment for later trials.

After a velocity command is generated, the environment is stepped forward one timestep and the next state is determined. Moreover, the reward r_t that is given to the agent must be calculated. In [22], the reward is defined by the difference in the chaser state error of the current and previous timesteps,

$$r_t = \|\mathbf{K}\left(-|\mathbf{e}_t| + |\mathbf{e}_{t-1}|\right)\|, \tag{7}$$

for a weighting matrix $\mathbf{K}$ such that the reward is positive for actions that move the chaser closer to the desired position, and is negative otherwise. Additional terms may be added to the reward equation that penalize large velocities in the vicinity of the desired state or collisions between the spacecraft. In this work, the reward is a function of the differential chaser state error and a collision term,

$$r_t = \|\mathbf{K}\left(-|\mathbf{e}_t| + |\mathbf{e}_{t-1}|\right)\| - r_{\text{collide}}. \tag{8}$$

After each timestep, the state error $\mathbf{e}_t$, action $\mathbf{a}_t$, reward r_t, and next state error $\mathbf{e}_{t+1}$ are stored in a replay buffer. The replay buffer stores the $R = 10^6$ most recent data points and is used asynchronously by a learner to train the value network one step and then train the policy network one step. At the end of the episode, the environment is reset and the most up-to-date policy weights are obtained from the learner to be used in the next episode of each actor instance.

D. Network Training

To train the policy and value networks, batches of size $M = 256$ of simulated data is randomly sampled from the replay buffer. The value network is trained to minimize the cross entropy loss function

$$L(\phi) = \mathbb{E}\left[-Y \log(Z_\phi(\mathbf{e}, \mathbf{a}))\right] \tag{9}$$

where $\mathbb{E}$ is the expectation operator and Y is the target value distribution as predicted from the sampled data given by [22]

$$Y = \gamma^N Z_{\phi'}(\mathbf{e}_N, \pi_{\theta'}(\mathbf{e}_N)) + \sum_{n=0}^{N-1} \gamma^n r_n \tag{10}$$

where $\gamma = 0.99$ is the discount factor, N is the horizon length accounted for in the prediction, $Z_{\phi'}(\mathbf{e}_N, \pi_{\theta'}(\mathbf{e}_N))$ is the value distribution evaluated by the target network at the horizon length, and r_n is the reward at each intermediate step n. In [22], the learning rate is set as $\beta = 0.0001$.

The total rewards expected to be received for a state-action pair depends on the policy weights θ [21],

$$J(\theta) = \mathbb{E}\{Z_\phi(\mathbf{e}, \pi_\theta(\mathbf{e}))\}. \tag{11}$$

To increase the expected reward, the policy network is trained using the gradient of J,

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J(\theta)}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \theta} \tag{12}$$

or otherwise written as [22]

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \pi_\theta(\mathbf{e}) \mathbb{E}\left[\nabla_\mathbf{a} Z_\phi(\mathbf{e}, \mathbf{a})\right] \big|_{\mathbf{a}=\pi_\theta(\mathbf{e})}\right] \tag{13}$$

where it is clear that this expression is differentiating both the policy network and the value network. From the gradient, the policy weights are updated as

$$\theta = \theta + \nabla_\theta J(\theta)\alpha \tag{14}$$

where $\alpha = 0.0001$ is the learning rate.

V. Code Implementation

The code as developed for this project takes inspiration from the open source D4PG implementation[2] and Deep Deterministic Policy Gradient (DDPG) implementation[3] on GitHub by Mark Sinton (written in TensorFlow v1), and the DDPG implementation on the Keras examples website[4] (written in TensorFlow v2). It can be noted that TensorFlow v1 and TensorFlow v2 have fundamentally different paradigms where, notably, the software has been upgraded for "eager execution" and no longer requires manual compilation and execution of the networks to generate output. Furthermore, many application programming interfaces (APIs) supporting, for example, building the layers of neural networks have been depreciated and so many architectures and functions from TensorFlow v1 have to be re-written for TensorFlow v2.

A. Implementation

The training algorithm, D4PG, as previously described in Sec. IV was used by Hovell and Ulrich [22] to generate their version of Deep Guidance. In this work, the authors have implemented a very similar algorithm called DDPG [26]. DDPG is the predecessor of D4PG and trains the policy to operate over continuous action spaces using an experience replay buffer for batch training with smoothed target weights to promote stability. Unlike D4PG which uses a *distributional* Deep Q-network during training, DDPG uses the vanilla Deep Q-network. One other difference between D4PG and DDPG is that DDPG executes a single actor and learner sequentially, whereas D4PG implements multiple actors, environments, and a single learner in different processing threads for faster, parallel programming. The original D4PG algorithm as provided in Algorithm 1 can be re-written to be executed in a single-thread and is given in Algorithm 2. Algorithm 2 can be further modified to more closely resemble DDPG if the prediction horizon in Eq. (10) is set to $N = 1$ or one time-step return, i.e., removing the N-step return.

B. Architecture

The implementation of the program is written in an object-oriented programming (OOP) paradigm. Each of the major blocks of the program are bundled into separate `classes`: the environment and the replay buffer, in particular. The training process containing the main loop from Algorithm 2 is written as an executable program that instantiates and calls the methods of each class as necessary. For a parallel programming approach as in Algorithm 1, the agent and

[2]https://github.com/msinto93/D4PG/
[3]https://github.com/msinto93/DDPG/
[4]https://keras.io/examples/rl/ddpg_pendulum/

Algorithm 2: Modified DDPG [26] with D4PG [24] traits

```
Data: Batch size $M$, trajectory length $N$, replay buffer size $R$, exploration rate $\sigma$, discount factor $\gamma$, policy and value
      network learning rates $\alpha$ and $\beta$
1  Randomly initialize policy and value network weights, $\theta$ and $\phi$
2  Initialize smoothed policy and target value network weights, $\theta' = \theta$ and $\phi' = \phi$
3  Initialize replay buffer
   while episode $E <$ maximum number of episodes do
4      Reset the environment
       while timestep $t <$ maximum number of timesteps do
5          Sample an action from the policy and add exploration noise via Eq. (6), i.e., obtain $\mathbf{a}_t$ via Eq. (5)
6          Step environment forward one timestep using the action $\mathbf{a}_t$; observe the reward $r_t$ and next state $\mathbf{e}_{t+1}$
7          Record $(\mathbf{e}_t, \mathbf{a}_t, r_t = \sum_{n=0}^{N-1} \gamma^n r_n, \mathbf{e}_{t+1})$ and store in the replay buffer
8          Sample a batch of $M$ transitions $(\mathbf{e}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ from the replay buffer
9          Compute the target value distribution from Eq. (10), which is used to train the value network
10         Update the value network weights $\phi$ by minimizing the loss function in Eq. (9) using learning rate $\beta$
11         Compute the policy gradients using Eq. (13) and update policy weights using Eq. (14)
12         Update the smoothed network weights slowly using Eqs. (3) and (4)
       end
   end
```

learner would also be bundled into separate classes that are then called by the main (parent) training program.

In this report, the authors have developed individual versions of the program.

1) Program 1: C. Bashnick has written a Python program implementing the sequential programming approach as given in Algorithm 2 using TensorFlow v2. Critically, the author notes that the cross entropy loss function is used in training the critic network. This approach avoids threading and, consequently, the development of a communication architecture between those threads. A sequential programming approach was chosen since the author is not familiar with parallel programming techniques.

2) Program 2: L. Arora has written a program using the PyTorch framework in which the Mean Squared Error (MSE) loss function is used to train the value network. PyTorch was developed by Facebook and was first publicly released in 2016. It was created to offer production optimizations similar to TensorFlow while making models easier to write. It later became open-sourced on Github in 2017. Since Python programmers found it so natural to use, PyTorch rapidly gained users, inspiring the TensorFlow team to adopt many of PyTorch's most popular features in TensorFlow 2.0.

For the PyTorch framework, additional features were used for the rendezvous environment as compared to the TensorFlow environment, which was explained earlier. This customized environment is very similar to the environment in Hovell and Ulrich [22]. This means that some features, like using a controller and different types of reward functions, were not considered in the TensorFlow environment. The aim of this section is to show the distinction between PyTorch and TensorFlow. One main feature that distinguishes PyTorch from TensorFlow is data parallelism. PyTorch optimizes performance by taking advantage of native support for asynchronous execution from Python. In TensorFlow, one has to manually code and fine tune every operation to be run on a specific device to allow distributed training.

C. Hyperparameters

For ease of reproducibility, the parameters used for training the networks and testing the performance of the trained policy network are summarized in Table II.

TABLE II: Overview of program hyperparameters.

Parameter	Symbol	Value
Number of states	–	3
Number of actions	–	3
Timestep [s]	Δt	0.2
Max number of timesteps	t^{max}	450
Max number of episodes	E^{max}	2,000
Learning rate	α, β	0.0001
Horizon, N-step return	N	1
Batch size	M	256
Target network smoothing factor	ϵ	0.001
Discount factor	γ	0.999
Buffer size	R	10^6
Hidden Layer 1 size	–	400
Hidden Layer 2 size	–	300
Critic Network bin number	B	51
Network value bounds	–	[-1000,100]
Max action bounds [m/s,m/s,rad/s]	$\mathbf{v}^{max}$	$[0.05, 0.05, \pi/18]$
Min action bounds [m/s,m/s,rad/s]	$\mathbf{v}^{min}$	$[-0.05, -0.05, -\pi/18]$
Max state bounds [m,m,rad]	$\mathbf{x}^{max}$	$[3.5, 2.4, 8\pi]$
Min state bounds [m,m,rad]	$\mathbf{x}^{min}$	$[0, 0, -8\pi]$

D. Building Neural Networks

To build the neural networks in Program 1 by C. Bashnick, the Keras Functional API from TensorFlow v2 is utilized, which allows the user to build non-linear models, e.g., having shared layers or multiple inputs or outputs that must be concatenated. In TensorFlow v1, the function call used to create a dense layer was:

```
dense_layer = tf.layers.dense(inputs,units,...
                activation)
```

where `inputs` gives the tensor input of the previous layer, `units` is the dimensionality of the output space, and `activation` specifies the layer's activation function. In the Keras Functional API of TensorFlow v2, an additional step is required to first funnel the input (e.g., the state error of the chaser) into the network before the dense layers may be generated:

```
input_layer = tf.keras.layers.Input(shape)
dense_layer = tf.keras.layers.Dense(units,...
                activation)(input_layer)
```

where `shape` specifies the size of the neural network input. The structure of the actor network as described in Table I is therefore generated as:

```
input_layer = tf.keras.layers.Input((STATE_SIZE,))
layer1 = tf.keras.layers.Dense(
                ACTOR_NETWORK_SIZE[0],...
                activation="relu")(input_layer)
layer2 = tf.keras.layers.Dense(
                ACTOR_NETWORK_SIZE[1],...
                activation="relu")(layer1)
layer3 = tf.keras.layers.Dense(ACTION_SIZE,...
                activation="tanh")(layer2)
```

The actions output by the tanh activation function are in the range [-1,1] and must be scaled to the action range:

```
output = tf.math.multiply(0.5, tf.math.multiply(layer3,
        (ACTION_BOUND_HIGH - ACTION_BOUND_LOW))
            + (ACTION_BOUND_HIGH + ACTION_BOUND_LOW))
```

A policy network model is generated from the input and output layers:

```
model = tf.keras.Model(input_layer, output)
```

For the value network, the states are fed into the first layer and the actions are fed directly into the second layer of the network, as described in [22]. The structure for the value network outlined in Table I is therefore generated as:

```
input_layer1 = tf.keras.layers.Input((STATE_SIZE,))
layer1 = tf.keras.layers.Dense(
                CRITIC_NETWORK_SIZE[0],...
                activation="relu")(input_layer1)
layer2a = tf.keras.layers.Dense(
                CRITIC_NETWORK_SIZE[1],...
                activation=None)(layer1)
input_layer2 = tf.keras.layers.Input((ACTION_SIZE,))
layer2b = tf.keras.layers.Dense(
                CRITIC_NETWORK_SIZE[1],...
                activation=None)(input_layer2)
layer2c = tf.keras.layers.Concatenate()(
                [layer2a, layer2b])
layer2 = tf.keras.layers.Activation("relu")(layer2c)
output_logits = tf.keras.layers.Dense(
                CRITIC_NETWORK_SIZE[2],
                activation=None)(layer2)
output_probs = tf.keras.layers.Activation("softmax")
                (output_logits)
```

A value network model is generated from tuples of the input and output layers:

```
model = tf.keras.Model([input_layer1,input_layer2],
                        [output_logits, output_probs])
```

E. Environment

The `Environment` class has three methods: The `__init__` method is used to initialize the environment parameters including the timestep and the nominal positions of the chaser and target; The `reset` method is called once an episode is completed and will reset the positions of the chaser and target, re-define the positions of the holding and docking points in relation to the target position, and reset the time; The `step` method takes as input the action of the chaser as generated by Eq. (5) and will observe the next state by solving the kinematics equations using the built-in differential equation integrator from Scipy:

```
next_state = scipy.integrate.odeint(kinematics_func,
                                    state,
                                    time_array,
                                    args = (action,))
```

Based on the time within the episode, the environment also determines whether the chaser is reducing its error from the holding or docking point. The `step` method outputs the transition (state error, action, reward, and next state error) to be sent to the replay buffer.

F. Replay Buffer

The algorithm uses a replay buffer to sample experiences to train neural network parameters. During each timestep within an episode, an experience tuple of the state error, commanded action, generated reward, next state error is saved and stored in a finite-sized cache: a replay buffer. Then, random mini-batches of experiences from the replay buffer are sampled to train the value and policy networks.

The `ReplayBuffer` class as used in Program 1 by C. Bashnick has four methods: The `__init__` method is used to initialize the buffer parameters and pre-allocate memory for the incoming buffer queue information from the actor, i.e., the state error, action, reward, and next state error; The `record` method is called after each timestep within an episode to input the observed data into the replay buffer; The `learn` method randomly samples transitions in the buffer to generate a mini-batch of experiences, which it then inputs into the `update` method, which trains the main network parameters using the equations defined in Sec. IV-D. This method outputs the mean loss of the critic network over the batch size such that the progress of the training can be monitored.

The code implementation for network training is more specifically outlined in the following subsection.

G. Network Training

In Program 1 by C. Bashnick, the loss that trains the critic network is the cross entropy loss function given in Eq. (9), designed for "measuring the probability error in discrete classification tasks in which the classes are mutually exclusive", as stated in the TensorFlow v2 documentation. This loss is calculated using the TensorFlow function where the `logits` parameter specifies the output of the critic network (before going through a softmax activation function) and `labels` is the target value distribution given by Eq. (10) that must be projected onto the bounds of the critic network's output distribution, i.e., the B bins ranging [-1000,100]:

```
loss = tf.nn.softmax_cross_entropy_with_logits(
        logits=output_logits,
        labels=tf.stop_gradient(target_distribution))
```

The mean of the loss is additionally calculated over the batch size,

```
mean_loss = tf.math.reduce_mean(loss)
```

before being used to train the critic network parameters,

```
critic_grad = tape.gradient(mean_loss,
                critic_model.trainable_variables)
critic_optimizer.apply_gradients(zip(critic_grad,
                critic_model.trainable_variables))
```

where `tf.GradientTape()` is used to record operations to a 'tape' such that automatic differentiation done with the `tape.gradient` method can be used to calculate the gradients of the loss with respect to the network parameters.

The policy network is trained using gradients of its performance compared to the output of the critic network. Thus, the gradients of critic network output probabilities (through the softmax function) over the chaser actions are calculated,

```
dQ_dAction = tf.gradients(output_probs, actions, bins)
```

where the `bins` are generated from the empirical bounds, i.e., [-1000,100], and the known number of bins B:

```
bins = tf.linspace(VALUE_DISTRIBUTION_BOUND[0],...
                   VALUE_DISTRIBUTION_BOUND[1],...
                   CRITIC_NETWORK_BIN_NUMBER)
```

The gradient of the policy network actions over the network parameters are then calculated where the gradients are initialized using the negative `dQ_dAction` for gradient ascent:

```
grads = tape.gradient(actions,...
             actor_model.trainable_variables,...
             -dQ_dAction)
```

The mean of the gradients are found over the batch size,

```
grads_scaled = list(map(lambda x: tf.divide(x,
                   BATCH_SIZE), grads))
```

and these scaled gradients are used to train the policy network parameters:

```
actor_optimizer.apply_gradients(zip(grads_scaled,...
                critic_model.trainable_variables))
```

VI. Simulation Results

A. Results of Program 1

Using the previously discussed methodology and code implementation, the networks were trained for 2000 episodes taking roughly 10 hours of wall-clock time on an Intel i7-8550 U central processing unit running at 1.80 GHz.

After each of the $t^{max} = 450$ timesteps within a single episode, the main policy and value networks were trained using the $M = 256$ batch of data sampled from the replay buffer. Once the main network training step was complete, the parameters of the target policy and value networks were updated as per Eqs. 3 and 4. Thus, the network parameters were trained over $450 \times 2000 = 900,000$ total steps. The author found that training the models after each timestep within an episode was much faster than training only once after every episode.

During training, every 25th episode was run with no added noise to the sampled actions from the policy network. This was done such that the performance of the trained network could be analyzed as a standard, deterministic policy with the trajectory of the chaser spacecraft animated and saved to the local machine. Animations were not made more regularly for two reasons: (1) generating animations increases the wall-clock time required for training and (2) running a deterministic (non-noisy) policy does not allow for environment exploration.

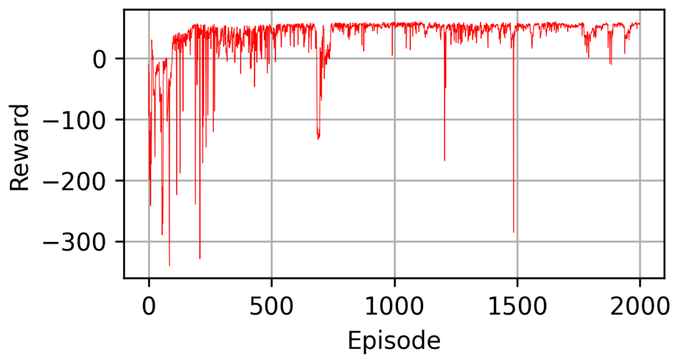
Reward and Loss Curves: The total reward obtained every episode was logged and plotted as shown in Fig. 7a. As expected, the reward at early training is very low since the agent has not found optimal actions to take at each state it encounters. By the end of training, the agent achieves a fairly consistent reward around 50.

Similarly, the mean loss of the main critic network (over the batch size) was monitored each timestep within an episode. The mean loss was averaged over each episode and is plotted in Fig. 7b. As expected, the loss decreases over the course of training. For the first 1000 episodes the loss decreases by 2.7 points, and for the last 1000 episodes the loss decreases by 0.1 points to reach a final value of around 1.1.

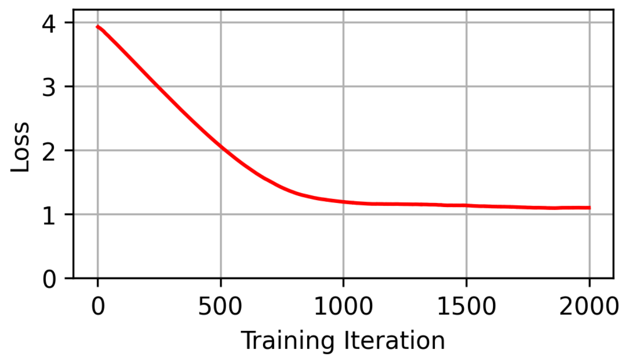
The reward and loss curves from network training can be compared to the results of Hovell and Ulrich [22], in which their models were trained for 47 hours on an Intel i7-8700 K central processing unit running at 3.70 GHz. Notably, Hovell and Ulrich [22] computed nearly 18,000 episodes and 780,000 training iterations to train their models. The reward and loss curves from Hovell and Ulrich [22] are plotted in Fig. 8.

Comparing the reward curve in Fig. 7a to the reward curve in Fig. 8a, it is clear that the results generated from this work have comparable performance once trained to Hovell and Ulrich [22] (i.e., generating rewards around a value of 50), however, the reward curve from this work is significantly noisier. The loss from Hovell and Ulrich [22] given in Fig. 8b plateaus around a value of 0.7, which is lower than the loss of 1.1 that was calculated in this work.

The differences in training performance for the models from this work and the models of Hovell and Ulrich [22] may be attributed to several factors: First, Hovell and Ulrich [22] use an $N = 5$ step return in the reward, which means the agent is looking at the impact of its current action further into the future (i.e., it is able to plan better). This work only considers an $N = 1$ step return. Second, Hovell and Ulrich [22] use $K = 10$ actors running in parallel to generate data for their replay buffer, meaning they sample much more data resulting in a higher probability of generating *better* samples. In this work, only a single actor is used. Third, and tying in with the previous point, Hovell and Ulrich [22] use a *prioritized* replay buffer, whereby the better-performing samples are drawn more often during training. In this work, only a simple replay buffer is used where the samples are drawn from a uniform

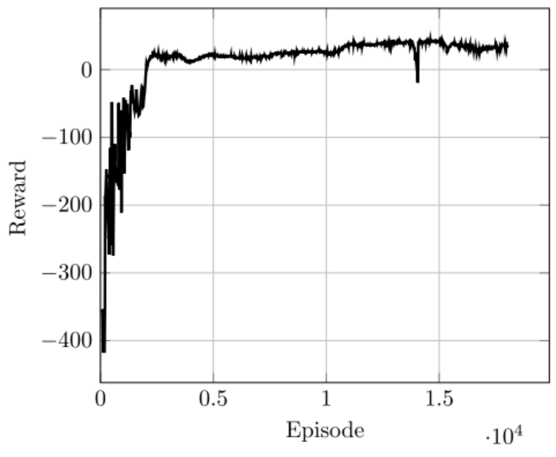


(a) Reward per episode

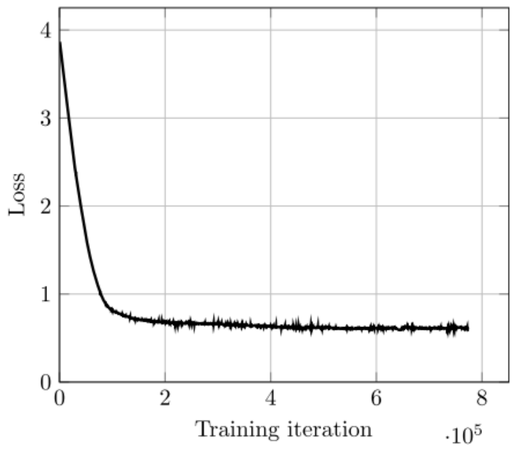


(b) Mean loss per episode

Fig. 7: The training performance using the code developed for this paper. The logged reward per episode is plotted in (a) and the average critic loss per episode is plotted in (b).



(a) Reward per episode



(b) Loss per training iteration

Fig. 8: The training performance from Hovell and Ulrich [22]. The logged reward per episode is plotted in (a) and the critic loss per training iteration is plotted in (b).

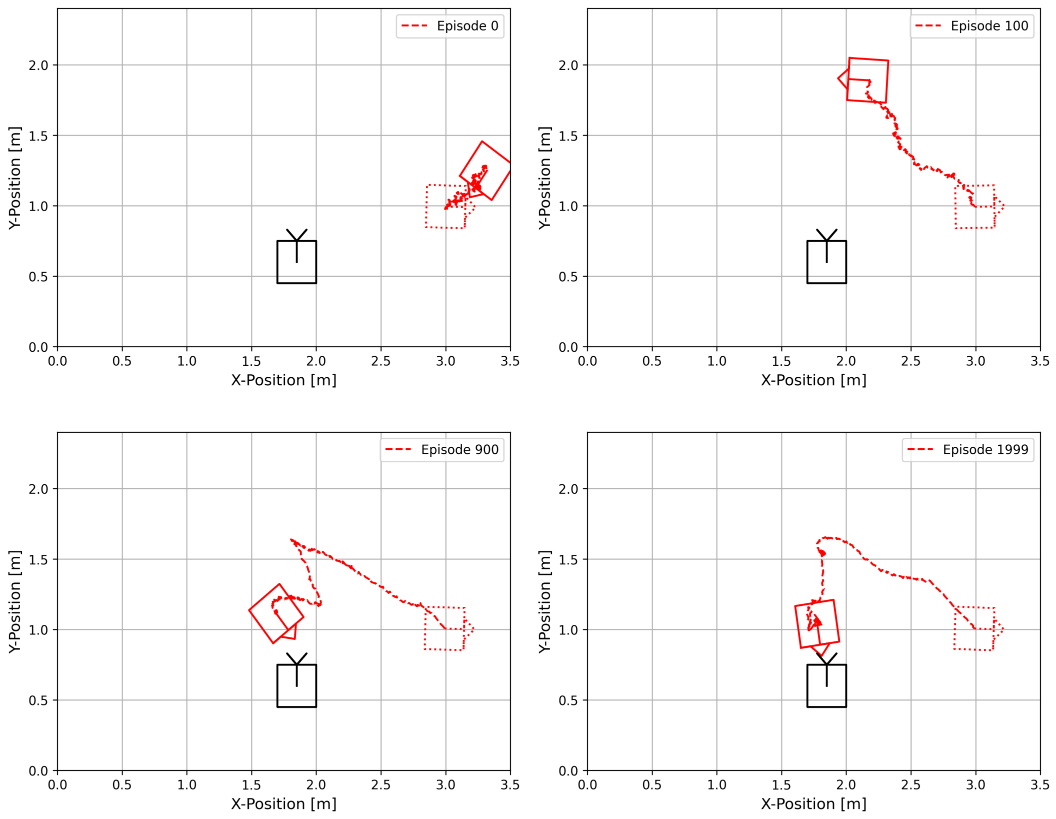


Fig. 9: Visualization of the chaser trajectories at various episodes during the training process, for the code developed for this paper.

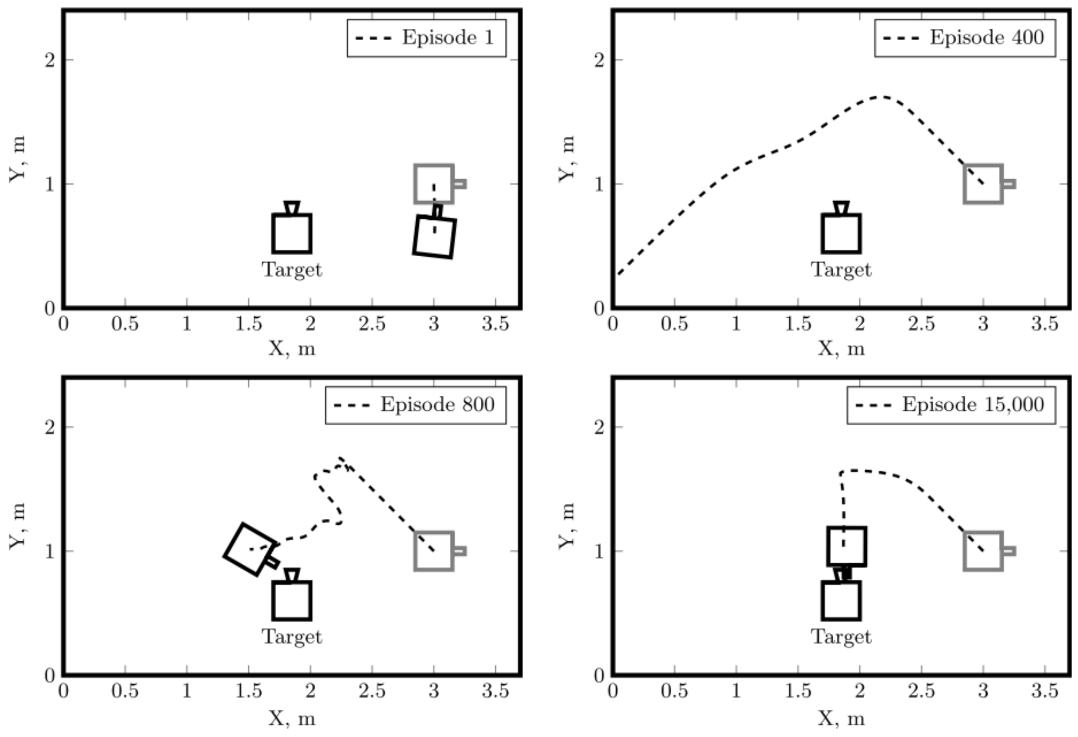


Fig. 10: Visualization of the chaser trajectories at various episodes during the training process, from Hovell and Ulrich [22].

distribution. The lack of a prioritized replay buffer means that training may take longer, and will not be done on the *best* data which limits the optimality of the trained network. Fourth, Hovell and Ulrich [22] have trained their network much longer on a better machine as compared to this work.

Trajectories During Training: The trajectories generated by the main policy network during the training process for various episodes are shown in Fig. 9. During these episodes, noise is added to the actions sampled by the policy network to generate the output velocity commands [i.e., Eq. (5)]. The chaser spacecraft is plotted in red (dotted for the initial state, solid for the final state) and the target is plotted in black. The trajectory taken by the chaser spacecraft over the duration of the episode is designated by the red dashed line. It is clear that through training the network, the chaser spacecraft slowly improves its performance in the assigned task of reducing its error from the holding and docking points.

In episode 0, the untrained network is simply outputting values associated with the original randomly initialized network parameters. After 100 episodes, the network has learned to direct its actions toward the holding point. After 900 episodes, the network has learned to move to the holding point and then switch to move toward the docking point. By episode 1999, the network has learned the basics of the assigned task. With more training episodes, the network would likely learn a more refined behaviour with a direct, smooth trajectory which would increase its reward, and would refine its ability to dock to the target.

The generated trajectories from this work in Fig. 9 can be compared to the results of Hovell and Ulrich [22] in Fig. 10 and are shown to be very similar. It is clear, however, in the trajectory of episode 15,000 in Fig. 10 that the network of Hovell and Ulrich [22] has learnt a smoother policy to accomplish the spacecraft pose tracking and docking task, and the chaser makes appropriate, head-on contact with the target. This smoother policy coincides with the fact that Hovell and Ulrich [22] achieve a lower loss function than the loss function achieved in this work, and therefore their trained network is expected to have better performance.

B. Results of Program 2

Two scenarios a) Chaser spacecraft with controller and b) Chaser spacecraft without controller were taken into account for Program 2 by L. Arora. The rewards averaged over 100 episodes were calculated and logged as shown in Fig. 11. The networks were trained for 3000 episodes (with controller) and 2300 episodes (without controller) taking around 18.5 hours and 14 hours, respectively of cell execution time on GPU provided by Google Colaboratory.

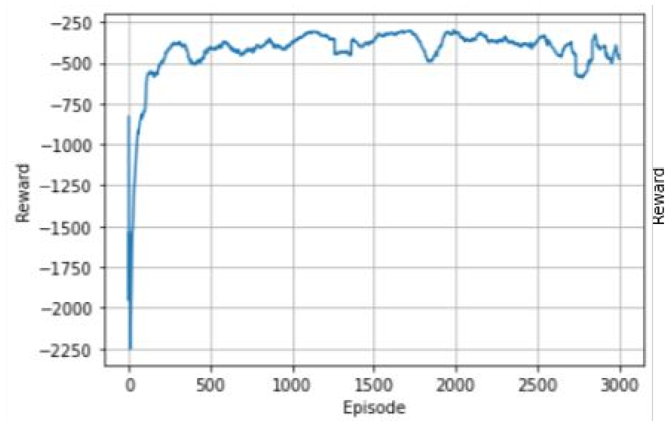
Here as well, the main policy and value network were trained with $M = 256$ batch of data sampled from the replay buffer. The total number of time steps came out to be $450 \times 3000 = 1,350,000$ by the network parameters with the PD controller with values as $K_P = 0$ and $K_D = 0.2$. For the scenario without controller, $450 \times 2300 = 1,035,000$ total time steps were considered. During the training process, noise is added to the actions sampled by the policy network to generate the output velocity commands. Initially, the reward obtained by the agent is very low since it still is finding the optimal policy (course of optimal actions) at each state.

In on-policy architecture, sequential samples (trajectory) are fed for each training batch. As the training progressed, the network rewards kept on increasing as expected, similar to the other curves. The DDPG algorithm in the PyTorch framework converged with higher but negative rewards. A qualitative evaluation of the learned models' behaviors after 3000 and 2300 training episodes reveals that, even though they have not solved the environment yet, they are all exceedingly good at stabilizing themselves after being initialized with some random value.

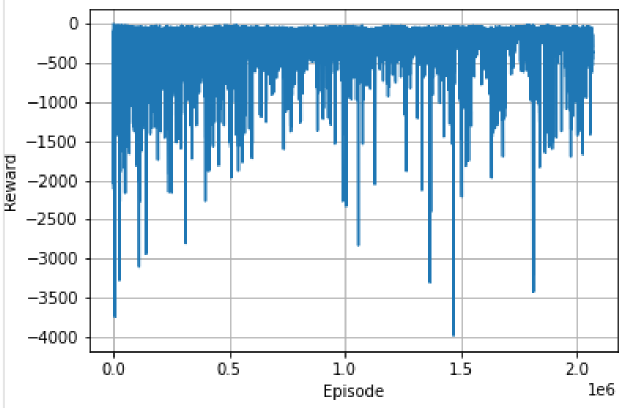
Plots with PyTorch: Comparing the reward curve in Fig. 11a to the reward curves in Fig. 8a and in Fig. 7a, there was discrepancy in the reward calculation as observed by the agent. This was to be expected since a customized environment was used with some different features, as mentioned earlier. Although, the plot follows the same trend as the other plots. Furthermore, while comparing the reward obtained without the controller in Fig. 11b, it was observed that here as well it follows the same trend but there is a lot of variability in the reward returns.

These discrepancies might be due to the fact that the training environments as well as some of the parameters taken by Hovell and Ulrich [22] and PyTorch are slightly different for the PyTorch framework. Moreover, the decrease in the training performance might be because of using DDPG as the base algorithm for this work, while Hovell and Ulrich uses D4PG, which is the extension of DDPG. Furthermore, Hovell and Ulrich [22] use various types of reward functions and Program 1 considers just one type of reward function, while different types of reward functions are used for PyTorch framework similar to the referenced paper. Furthermore, Program 2 uses MSE as the loss function for the value network whereas Hovell and Ulrich [22] and Program 1 take cross-entropy as the loss function. Program 2 was executed on publicly-available online GPU with less computation power and limited time whereas Hovell and Ulrich [22] have trained their network much longer on a better machine. Interestingly, the training performance could even be considered to have degraded when both noise techniques are incorporated. This is likely due to too much error in the model which ultimately degrades training accuracy.

The main objective of this part of the work is to show the efficacy of the DDPG algorithm as compared with D4PG algorithm using DDPG hyperparameters same as those used for D4PG, with the exception that for D4PG (1) hard target network updates are applied every 100 steps, and (2) exploration noise is sampled from a Gaussian distribution with fixed standard deviation (σ). Moreover, by comparing the plots from the referenced paper [22] and Program 1, Program 2 proves that reward shaping plays a vital role in the training of the networks as well.



(a) Reward with controller



(b) Reward without controller

Fig. 11: The logged averaged reward for 100 episodes with controller is plotted in (a) and the logged averaged reward for 100 episodes without controller is plotted in (b).

VII. Conclusion

This paper trained a policy network using the DDPG framework to output velocity commands for the task of spacecraft proximity operations, specifically, pose tracking and docking with a stationary target. The performance of the network was monitored during training through the episodic reward and the average critic network loss per episode. Comparisons with the reference paper that originally proposed this idea show that the network performance generated in this work is acceptable, but is not as optimal as demonstrated in literature. After 2000 training episodes using Program 1, the generated reward averages around 50 and the critic loss is about 1.1. The performance of the networks is also evaluated in the PyTorch environment. The rewards obtained in this framework were compared with the results obtained in the reference paper and the TensorFlow framework. Upon comparing, PyTorch proves to be an ideal framework for D4PG, on which the reference paper is based. Future work in improving this network training algorithm includes implementing an N-step return on the reward and a prioritized replay buffer to sample "better" experiences more frequently.

VIII. Milestone 4 Contributions

C. Bashnick completed the TensorFlow code implementation (Program 1) and the results written in Section VI-A with associated figures therein. She also added Fig. 4, wrote the Abstract and Conclusion, wrote the majority of Secs. I to IV, and edited Secs. I to V with a particular emphasis on updating Sec. V to reflect the final version of the TensorFlow code implementation (Program 1).

L. Arora completed the PyTorch framework code implementation (Program 2) and the results written in Section VI-B with associated figures therein. He helped in writing the Conclusion, wrote parts of the manuscript, created new table, and completed editing of the sections with a particular emphasis on writing Sections V-B2 and VI-B to reflect the final version of the PyTorch code implementation (Program 2). Moreover, the cost of GPU units from Google Colaboratory for Program 2 was borne by him.

Appendix A: Running Program 1

Supplementary material for Program 1 has been included with this report submission by C. Bashnick including: one iPython notebook (.ipynb), four pre-trained models (for the main and target policy and value networks; .h5), and one file containing the training reward and loss history for these pre-trained models (.csv).

The included iPython notebook file can be run as-is via, e.g., Jupyter Notebook, to test the provided pre-trained models on a machine with, in particular, Python version 3.7.15, TensorFlow version 2.1.0, Keras version 2.2.4-tf, and Numpy version 1.21.6. Differences from these software/library versions may cause compatibility issues. The policy network testing is achieved in the "Testing" code block (blocks 15 and 16) in which an animation and plot of the trajectory are generated and saved to the local machine.

To train a new model, the user can navigate to the "Training" code block (block 11) and update the parameter `trainModel` to True. In this case, the user must ensure that there is a folder called "ckpts" in the directory where the notebook file is located so that checkpoints of the model parameters may be regularly saved.

To test the newly trained network, the user must navigate to the 14th code block where models are loaded from files, and set the parameter `loadModelFromFile` to False.

Appendix B: Running Program 2

Program 2 can easily be run with the help of Google Colaboratory. The code is written in Python. All the basic modules are already installed there. If not, user just need to use `pip install <package name>` to install that latest version for that module on the user's Google Colab notebook. Furthermore, all the explanations for the code and lines are also included in the code file. Note that you need to install latest version of PyTorch for the notebook.

The included .ipynb file can be run as-is to test the models. Before running the code, it is suggested to mount your Google drive by running the first cell in the file. This way all the "checkpoints" or the saved models will be saved after every 25 episodes. Now, run all the cells before Main cell.

To use the controller, uncomment all the lines as mentioned in the Environment section of the file.

Supplementary material for Program 2 has been included with this report including: one iPython file named "Applied-AI-PyTorch-DDPG", Python extension file with the same name, JPEG files "Training-time-with-controller.jpg" and"Training-time-without-controller.jpg" for the execution time for training of the networks.

Note: This code was executed on the Pro subscription of Google Colaboratory. If you are running on a free version, training time will vary from what is in the pictures. Also, in order to run your code on a local machine, user can save all the cells in the notebook as different Python script files and execute the Main script file. Make sure to put the desired directory address to save all the checkpoints.

References

[1] D. Izzo, M. Märtens, and B. Pan, "A survey on artificial intelligence trends in spacecraft guidance dynamics and control," *Astrodynamics*, vol. 3, no. 4, pp. 287–299, 2019.

[2] M. Shirobokov, S. Trofimov, and M. Ovchinnikov, "Survey of machine learning techniques in spacecraft control design," *Acta Astronautica*, vol. 186, pp. 87–97, 2021.

[3] L. Cheng, Z. Wang, F. Jiang, and C. Zhou, "Real-time optimal control for spacecraft orbit transfer via multiscale deep neural networks," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 55, no. 5, pp. 2436–2450, 2018.

[4] S. Yin, J. Li, and L. Cheng, "Low-thrust spacecraft trajectory optimization via a dnn-based method," *Advances in Space Research*, vol. 66, no. 7, pp. 1635–1646, 2020.

[5] B. Sun and E.-J. van Kampen, "Reinforcement-learning-based adaptive optimal flight control with output feedback and input constraints," *Journal of Guidance, Control, and Dynamics*, vol. 44, no. 9, pp. 1685–1691, 2021.

[6] H. Yoo, B. Kim, J. W. Kim, and J. H. Lee, "Reinforcement learning based optimal control of batch processes using monte-carlo deep deterministic policy gradient with phase segmentation," *Computers & Chemical Engineering*, vol. 144, p. 107133, 2021.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[9] K. D. Julian and M. J. Kochenderfer, "Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 8, pp. 1768–1778, 2019.

[10] D. M. Chan and A.-a. Agha-mohammadi, "Autonomous imaging and mapping of small bodies using deep reinforcement learning," in *2019 IEEE Aerospace Conference*. Big Sky, MT: IEEE, March 2019, pp. 1–12.

[11] A. Scorsoglio, R. Furfaro, R. Linares, and B. Gaudet, "Image-based deep reinforcement learning for autonomous lunar landing," in *AIAA Scitech 2020 Forum*, Orlando, FL, January 2020, p. 1910.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[13] L. Federici, A. Scorsoglio, A. Zavoli, and R. Furfaro, "Autonomous guidance for cislunar orbit transfers via reinforcement learning," in *AAS/AIAA Astrodynamics Specialist Conference*, Big Sky, MT (Virtual), August 2021.

[14] B. Gaudet, R. Linares, and R. Furfaro, "Deep reinforcement learning for six degree-of-freedom planetary landing," *Advances in Space Research*, vol. 65, no. 7, pp. 1723–1741, 2020.

[15] J. Broida and R. Linares, "Spacecraft rendezvous guidance in cluttered environments via reinforcement learning," in *29th AAS/AIAA Space Flight Mechanics Meeting*. Maui, HI: American Astronautical Society Ka'anapali, Hawaii, January 2019, pp. 1–15.

[16] C. E. Oestreich, R. Linares, and R. Gondhalekar, "Autonomous six-degree-of-freedom spacecraft docking maneuvers via reinforcement learning," *arXiv preprint arXiv:2008.03215*, 2020.

[17] L. Federici, B. Benedikter, and A. Zavoli, "Deep learning techniques for autonomous spacecraft guidance during proximity operations," *Journal of Spacecraft and Rockets*, vol. 58, no. 6, pp. 1774–1785, 2021.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *31st International Conference on Machine Learning*. Lille, France: PMLR, 2015, pp. 1889–1897.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] M. A. Patterson and A. V. Rao, "GPOPS-II: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 1, pp. 1–37, 2014.

[21] K. Hovell and S. Ulrich, "On deep reinforcement learning for spacecraft guidance," in *AIAA Scitech 2020 Forum*, Orlando, FL, January 2020, p. 1600.

[22] ——, "Deep reinforcement learning for spacecraft proximity operations guidance," *Journal of Spacecraft and Rockets*, vol. 58, no. 2, pp. 254–264, 2021.

[23] ——, "Laboratory experimentation of spacecraft robotic capture using deep-reinforcement-learning–based guidance," *Journal of Guidance, Control, and Dynamics*, pp. 1–9, 2022.

[24] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients," *arXiv preprint arXiv:1804.08617*, 2018.

[25] W. H. Clohessy and R. S. Wiltshire, "Terminal Guidance System for Satellite Rendezvous," *Journal of the Aerospace Sciences*, vol. 27, no. 9, pp. 653–658, 1960.

[26] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.