Spacecraft Artificial Intelligence Agent

Berke Tezgocen

December 2024

1 Abstract

This Python code represents a sophisticated project that integrates deep learning and physics simulations to develop an intelligent control system for space missions. The project specifically includes the following core components:

Relativistic Space Physics Simulation: A physics engine designed to simulate planetary motions and spacecraft trajectories with high accuracy, considering the effects of general relativity. This simulation accounts for phenomena such as time dilation, light bending, and gravitational redshift.

Genetic Algorithm and Transformer-Based Optimization: An optimization framework combining genetic algorithms with Transformer-based deep learning models to optimize spacecraft trajectories and identify optimal control policies. This framework aims to produce solutions that perform well even in complex scenarios, adhering to physical laws.

Soft Actor-Critic (SAC) Algorithm: A control system developed using modifications of the Soft Actor-Critic (SAC) algorithm, a deep reinforcement learning method focused on reward maximization. These modifications include the integration of long-term memory and physics knowledge.

Prioritized Experience Replay (PER): A technique to accelerate and enhance the learning process by prioritizing and repeatedly utilizing significant experiences during training.

Simulation of Various Spacecraft Systems: Simulations of various risks and challenges, such as fuel consumption, engine failures, fuel leaks, and micro-asteroid collisions. These simulations aim to enable the control system to adapt to real-world conditions.

Visualization and User Interface: A graphical interface developed using the Pygame library, enabling real-time monitoring of the simulation, visualization of metrics, and user interaction. Additionally, Matplotlib animations are utilized to facilitate the recording and analysis of simulations.

This project aims to provide a comprehensive platform for developing and implementing autonomous control systems by addressing the complexities of space missions. It serves as an example of how deep learning and simulation technologies can be applied to space exploration.

2 Introduction

Space exploration and travel represent some of humanity's most ambitious and complex endeavors. These challenging journeys require not only advanced engineering capabilities but also the autonomous control of complex systems. Traditional approaches often rely on predefined trajectories and manual interventions. However, long-term space missions demand intelligent control systems capable of handling dynamic conditions, uncertainties, and potential failures. This necessity makes the integration of artificial intelligence and deep learning techniques into this domain inevitable.

This project aims to address the autonomous control of spacecraft by combining Deep Reinforcement Learning (DRL), Genetic Algorithms (GA), and high-fidelity physics simulations. The primary motivation is to develop intelligent control systems that can overcome the complex challenges encountered in long-distance space missions and deliver high performance. These systems must exhibit the following capabilities:

Adaptability: The ability to adapt to changing environmental conditions, unexpected events (e.g., engine failures, fuel leaks), and mission objectives. Optimization: The ability to optimize trajectories and propulsion strategies under various constraints, such as fuel consumption, travel time, and risk. Physical Realism: The ability to accurately simulate the fundamental physical principles of space travel (Newtonian physics, relativity). Learning Capability: The ability to continuously improve performance by learning from experience. To achieve these goals, the project adopts the following innovative approaches:

Relativity-Aware Physics Engine: A simulation engine that goes beyond Newtonian physics by incorporating the effects of general relativity. This enables more accurate trajectory predictions and control strategies, particularly in scenarios involving strong gravitational fields and high velocities. Genetic and Transformer-Based Trajectory Optimization: A hybrid optimization method that combines the exploratory capabilities of genetic algorithms with the representational power of Transformer networks to identify optimal trajectories. Physics-Enhanced SAC Algorithm: A modification of the Soft Actor-Critic (SAC) reinforcement learning algorithm that integrates physical laws and long-term memory mechanisms. Advanced Experience Replay: Techniques such as Prioritized Experience Replay (PER) are employed to enable more effective learning from significant experiences. User-Friendly Interface: A Pygame-based interface for real-time visualization and control of simulations. This project aims to demonstrate the potential of deep learning and simulation technologies in space exploration and autonomous systems, while also paving the way for new possibilities in future space missions. In the following sections, we will delve into the details of the code, examining how each component functions and interacts with others.

3 Motivation

The primary motivation behind this project is to provide innovative and autonomous solutions to the increasing complexities encountered in the field of space exploration and travel. Modern space missions demand longer durations, more distant destinations, and increasingly intricate maneuvers. These factors challenge the limits of human intervention and elevate the need for autonomous systems. The key motivations driving this project are as follows:

Increasing Complexity of Space Missions Space missions are becoming significantly more complex compared to those of the past. Interplanetary travel, orbital maneuvers, landings, and exploration activities require precise control and navigation. Furthermore, as mission durations and distances increase, response times become longer, and remote control becomes more challenging. This underscores the necessity for autonomous control systems capable of managing these tasks independently.

The Potential of Artificial Intelligence and Deep Learning Recent advancements in artificial intelligence (AI) and deep learning highlight the potential of these technologies in developing autonomous control systems for space missions. Deep Reinforcement Learning (DRL), in particular, has shown promise due to its ability to handle complex tasks and dynamic environments. This project aims to harness the potential of DRL in the context of space travel scenarios, opening new possibilities for intelligent mission control.

The Need for Physical Realism The success of space missions relies heavily on accurate physical modeling. Space travel requires not only classical Newtonian mechanics but also considerations of Einstein's General Theory of Relativity, especially in strong gravitational fields. This project seeks to move beyond simple physical approximations to simulate the motion of spacecraft and planetary bodies with greater accuracy. By achieving more realistic and reliable simulations, the project enables better training of control systems for real-world applications.

Dealing with Uncertainty and Margins of Error Space travel is fraught with uncertainties and unexpected events. Scenarios such as engine failures, fuel leaks, or micro-asteroid collisions test the adaptability of control systems. This project aims to develop robust systems capable of handling such uncertainties and making swift decisions in response to these challenges.

The Importance of Autonomous Systems The cost, time, and risks associated with space missions necessitate minimizing human intervention. Autonomous systems must possess the ability to complete tasks, make decisions, and resolve problems without human oversight. This project contributes to the development of such autonomous systems, enabling them to operate effectively in the demanding conditions of space missions.

Efficient Utilization of Space Resources Resources such as fuel, energy, and time are scarce and valuable in space missions. This project aims to develop control strategies that maximize efficiency, minimize waste, and optimize the use of these critical resources, ensuring mission success under constrained conditions.

Conclusion In summary, this project is an investment in the future of space exploration. By demonstrating how artificial intelligence and deep learning technologies can revolutionize autonomous space systems, it seeks to empower humanity to reach farther and delve deeper into the cosmos.

4 Architecture

This project is composed of various interconnected modules designed to simulate space travel and develop intelligent control systems. The modular structure enhances the flexibility, scalability, and maintainability of the project. Below are the core architectural components of the project and their interactions:

4.1 Physics Engine (RelativisticSpacePhysics)

Purpose: Simulates the motion of spacecraft and planets.

Key Features: Implements Newton's laws of motion. Accounts for general relativity effects, including time dilation, light bending, and gravitational redshift. Updates motion using numerical methods such as Runge-Kutta-Fehlberg (RKF45). Calculates planetary orbits and spacecraft trajectories.

Technologies: PyTorch (for heterogeneous data processing, automatic differentiation, and GPU acceleration).

4.2 Spacecraft Simulation (Spacecraft Class)

Purpose: Simulates the state and systems of the spacecraft.

Key Features:

Maintains essential properties such as mass, fuel, position, velocity, and acceleration. Simulates dynamic behaviors such as thrust, engine temperature, fuel consumption, damage, and repair systems. Includes functions for sensor readings, emergency protocols, and system health monitoring. Updates motion using the Leapfrog integration method.
Technologies: PyTorch (for heterogeneous data processing, automatic differentiation, and GPU acceleration).

4.3 Planet Simulation (Planet Class)

Purpose: Simulates the motion and properties of planets.
Key Features:
Stores essential properties such as mass, orbital radius, position, and velocity. Updates motion using methods like Runge-Kutta (RK45). Calculates Lagrange points.
Technologies: PyTorch (for heterogeneous data processing, automatic differentiation, and GPU acceleration).

4.4 Optimization Layer (GeneticTransformerOptimizer)

Purpose: Optimizes spacecraft trajectories and control parameters.
Key Features:
Combines genetic algorithms with Transformer-based deep learning models. Conducts population-based search. Applies crossover and mutation operators. Refines trajectories while adhering to physical laws. Utilizes adaptive learning rates and optimization techniques.
Technologies: PyTorch, Adam optimizer.

4.5 Intelligent Control Agents (SACAgent, LongTermSACAgent, GeneticSACAgent)

Purpose: Controls the spacecraft and accomplishes mission objectives.
Key Features:
Based on the Soft Actor-Critic (SAC) algorithm. Extended network architectures (e.g., multi-critic networks, value networks). Integrates physics knowledge (PINN - Physics Informed Neural Networks). Includes experience replay and prioritized experience replay (PER) mechanisms. Incorporates long-term memory using Transformer-based architecture (LongTermSACAgent). Evolves SAC models using genetic algorithms (GeneticSACAgent).
Technologies: PyTorch, Adam optimizer, torch.distributions.Normal.

4.6 Experience Memory (TransformerMemory)

Purpose: Stores past experiences and utilizes them for learning.
Key Features:
Transformer-based memory mechanism. Prioritizes experiences based on importance. Adds new experiences to memory and removes outdated ones as needed.
Technologies: PyTorch, Transformer.

4.7 Space Travel Environment (SpaceTravelEnv)

Purpose: Provides the simulation environment and trains the spacecraft control system.

Key Features:

Initializes and manages planets and the spacecraft. Executes simulation steps, computes rewards, and determines episode termination. Simulates various emergency scenarios.

Technologies: PyTorch.

4.8 Visualization and User Interface (SpaceSimulationGUI)

Purpose: Visualizes simulation results and enables user interaction with the simulation.

Key Features:

Real-time graphical interface developed using the Pygame library. Visualizes the positions of the spacecraft and planets. Displays simulation metrics and system statuses. Provides control options such as pause, zoom, and pan. Supports orbital analysis through Matplotlib animations.

Technologies: Pygame, Matplotlib.

4.9 Training Time Estimation (TrainingTimeEstimator)

Purpose: Estimates training duration and monitors the progress of training.

Key Features:

Predicts remaining and total training time. Displays GPU utilization.

Technologies: Python time library.

Data Flow: The simulation environment (SpaceTravelEnv) initializes a spacecraft, which receives state information from the environment.
The control agent (SACAgent) uses this information to select an action (e.g., thrust).
The selected action is applied to the physics engine, which calculates the new state of the spacecraft.
The new state is updated in the simulation environment, and a reward is calculated.
The experience is stored in the experience memory.
The agent samples from the experience memory to perform the learning process.
The visualization interface continuously displays the simulation and provides feedback to the user.
The optimization layer refines trajectories and control parameters.

5 Theoretical Background

Gravitational Force, Relativity, and Spacecraft Dynamics

5.1 Gravitational Force and Motion

5.2 Newton's Universal Law of Gravitation

The gravitational force between two masses is given by:

$$F = G \cdot \frac{m_1 \cdot m_2}{r^2}$$

Where:

- F: Gravitational force (N)
- G: Gravitational constant ($6.67430 \times 10^{-11}\,\mathrm{m^3/kg\,s^2}$)
- m_1, m_2: Masses of the interacting bodies (kg)
- r: Distance between the bodies (m)

5.3 Gravitational Force in Vector Form

The vector form of gravitational force is:

$$\vec{F} = -G \cdot \frac{m_1 \cdot m_2 \cdot \vec{r}}{\|\vec{r}\|^3}$$

Where:

- $\vec{r}$: The vector pointing from one mass to the other
- $\|\vec{r}\|$: Magnitude of $\vec{r}$

5.4 Gravitational Acceleration

The gravitational acceleration is:

$$\vec{a} = \frac{\vec{F}}{m}$$

Where:

- $\vec{a}$: Acceleration ($\mathrm{m/s^2}$)
- $\vec{F}$: Gravitational force (N)
- m: Mass of the object experiencing the force (kg)

5.5 Motion Equations (Update)

5.5.1 Position Update

The position of an object is updated as:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t) \cdot \Delta t + 0.5 \cdot \vec{a}(t) \cdot (\Delta t)^2$$

Where:

- $\vec{r}(t + \Delta t)$: New position (m)
- $\vec{r}(t)$: Current position (m)
- $\vec{v}(t)$: Current velocity (m/s)
- $\vec{a}(t)$: Current acceleration ($\mathrm{m/s^2}$)
- Δt: Time step (s)

5.5.2 Velocity Update

The velocity of an object is updated as:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t) \cdot \Delta t$$

Where:

- $\vec{v}(t + \Delta t)$: Updated velocity (m/s)
- $\vec{v}(t)$: Current velocity (m/s)
- $\vec{a}(t)$: Current acceleration (m/s^2)

5.6 Leapfrog Integration

5.6.1 Velocity Half-Step Update

$$\vec{v}(t + \frac{\Delta t}{2}) = \vec{v}(t) + 0.5 \cdot \Delta t \cdot \vec{a}(t)$$

5.6.2 Full-Step Position Update

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \Delta t \cdot \vec{v}(t + \frac{\Delta t}{2})$$

5.6.3 Acceleration Update at New Position

$$\vec{a}(t + \Delta t) = \frac{\vec{F}(t + \Delta t)}{m}$$

5.6.4 Full-Step Velocity Update

$$\vec{v}(t + \Delta t) = \vec{v}(t + \frac{\Delta t}{2}) + 0.5 \cdot \Delta t \cdot \vec{a}(t + \Delta t)$$

5.7 Runge-Kutta 4(5) Method (RK4/5)

The higher-order integration method is expressed as:

$$y_{n+1} = y_n + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$$

And the 5th-order result for error estimation:

$$y_5 = y + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$$

Where:

- y_n: Current state (e.g., position and velocity)
- y_{n+1}: Next state
- $k_1, k_2, k_3, k_4, k_5, k_6$: Intermediate Runge-Kutta coefficients

5.8 Relativity Effects

5.9 Lorentz Factor

The Lorentz factor is given by:

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Where:

- γ: Lorentz factor
- v: Velocity of the object (m/s)
- c: Speed of light (3×10^8 m/s)

5.10 Schwarzschild Metric

The Schwarzschild metric is:

$$ds^2 = -\left(1 - \frac{r_s}{r}\right) dt^2 + \frac{1}{1 - \frac{r_s}{r}} dr^2 + r^2 d\theta^2 + r^2 \sin^2\theta \, d\phi^2$$

Where:

- ds^2: Spacetime interval
- dt: Time interval
- $dr, d\theta, d\phi$: Radial and angular intervals
- $r_s = \frac{2GM}{c^2}$: Schwarzschild radius

5.11 Gravitational Redshift

The gravitational redshift is:

$$z = \frac{1}{\sqrt{1 - \frac{r_s}{r}}} - 1$$

Where:

- z: Gravitational redshift
- r_s: Schwarzschild radius
- r: Distance from the massive object

5.12 Spacecraft Control and Dynamics

5.13 Thrust Force

$$\vec{F}_{\text{thrust}} = \text{thrust\_magnitude} \cdot \text{max\_thrust} \cdot \text{thrust\_direction}$$

5.14 Fuel Consumption

$$\text{fuel\_loss} = \text{thrust\_magnitude} \cdot \text{fuel\_consumption\_rate} \cdot \Delta t$$

5.15 Total Force

$$\vec{F}_{\text{total}} = \vec{F}_{\text{thrust}} + \vec{F}_{\text{gravity}} + \vec{F}_{\text{correction}}$$

5.16 Delta-v

$$\Delta v = I_{\text{sp}} \cdot g_0 \cdot \ln\left(\frac{m_0}{m_f}\right)$$

Where:

- Δv: Velocity change (m/s)
- I_{sp}: Specific impulse (s)
- g_0: Gravitational acceleration ($9.81\,\text{m/s}^2$)
- m_0, m_f: Initial and final masses

5.17 Specific Angular Momentum

The specific angular momentum vector is defined as:

$$\vec{h} = \vec{r} \times \vec{v}$$

Where:

- $\vec{h}$: Specific angular momentum vector (m^2/s)
- $\vec{r}$: Position vector (m)
- $\vec{v}$: Velocity vector (m/s)

The specific angular momentum is an important parameter in orbital analysis and energy calculations.

5.18 Centripetal Acceleration

The centripetal acceleration required for an object to remain in a circular orbit is:

$$a_c = \frac{v^2}{r}$$

Where:

- a_c: Centripetal acceleration (m/s^2)
- v: Velocity (m/s)
- r: Orbital radius (m)

5.19 Inverse Square Law

The gravitational force follows the inverse square law:

$$F \propto \frac{1}{r^2}$$

Where:

- F: Gravitational force (N)
- r: Distance between two objects (m)

5.20 Doppler Shift

The Doppler shift describes the change in wavelength of light due to relative motion between the source and the observer. The shift is significant in astrophysical observations.

$$\Delta f = f_s \frac{v_r}{c}$$

or equivalently:

$$\Delta \lambda = \lambda_s \frac{v_r}{c}$$

Where:

- Δf: Change in observed frequency (Hz)
- f_s: Source frequency (Hz)
- $\Delta \lambda$: Change in observed wavelength (m)
- λ_s: Source wavelength (m)
- v_r: Relative velocity between the source and observer (m/s)
- c: Speed of light (3×10^8 m/s)

5.21 Kinetic Energy

The kinetic energy of an object is:

$$KE = \frac{1}{2}mv^2$$

Where:

- KE: Kinetic energy (J)
- m: Mass of the object (kg)
- v: Velocity (m/s)

5.22 Potential Energy

The gravitational potential energy is:

$$PE = -G\frac{Mm}{r}$$

Where:

- PE: Potential energy (J)
- G: Gravitational constant ($6.67430 \times 10^{-11}\,\mathrm{m}^3/\mathrm{kg\,s}^2$)
- M, m: Masses of the two objects (kg)
- r: Distance between the objects (m)

5.23 Total Energy

The total energy of the system is:

$$E = KE + PE$$

Where:

- E: Total energy (J)
- KE: Kinetic energy (J)
- PE: Potential energy (J)

5.24 Momentum

The momentum of an object is defined as:

$$\vec{p} = m\vec{v}$$

Where:

- $\vec{p}$: Momentum vector ($\mathrm{kg} \cdot \mathrm{m/s}$)
- m: Mass of the object (kg)
- $\vec{v}$: Velocity vector (m/s)

5.25 Adaptive Time Step

The adaptive time step used in simulations is calculated as:

$$\Delta t = \min\left(\max\left(\frac{\text{min\_period}}{50}, \text{min\_dt}\right), \text{max\_dt}\right)$$

Where:

- Δt: Calculated time step (s)
- min_period: Minimum orbital period of the planets (s)
- min_dt: Minimum time step (s)
- max_dt: Maximum time step (s)

When the spacecraft is near the target planet, smaller steps can be used:

$$\text{if any(distance\_to\_destination} < \text{AU)}: \Delta t = \min(\Delta t, 86400) \quad \text{(1 day)}$$

This ensures the simulation dynamically adjusts its speed and accuracy.

5.26 Soft Actor-Critic (SAC) Algorithm Action Selection

In the SAC algorithm, actions are sampled as:

$$a = \tanh(\mu(s) + \sigma(s) \cdot N(0, I))$$

Where:

- a: Selected action
- $\mu(s)$: Mean of the current state
- $\sigma(s)$: Standard deviation of the current state
- $N(0, I)$: Noise sampled from a standard normal distribution

5.27 Target Network Update

The target network parameters are updated as:

$$\theta_{\text{target}} = \tau\theta + (1 - \tau)\theta_{\text{target}}$$

Where:

- θ_{target}: Target network parameters
- θ: Main network parameters
- τ: Smoothing coefficient

This ensures the target network smoothly converges towards the main network.

5.28 Relativity Corrections

5.29 Relativistic Force Correction

The relativistic force correction is expressed as:

$$F_{\text{relativistic}} = F_{\text{classical}} \cdot \text{relativistic\_correction}$$

Where:

- $F_{\text{classical}}$: Classical Newtonian force
- relativistic_correction: Multiplicative relativistic effects

5.30 Relativistic Acceleration

The relativistic acceleration is given by:

$$a = \frac{F_{\text{relativistic}}}{\gamma \cdot m}$$

Where:

- a: Acceleration (m/s^2)
- $F_{\text{relativistic}}$: Relativistic corrected force (N)
- γ: Lorentz factor
- m: Mass (kg)

5.31 Orbital Parameters

5.32 Semi-Major Axis

The semi-major axis is calculated as:

$$a = -\frac{GM}{v^2 - \frac{2GM}{r}}$$

Where:

- a: Semi-major axis (m)
- G: Gravitational constant ($6.67430 \times 10^{-11}\,\text{m}^3/\text{kg}\,\text{s}^2$)
- M: Mass of the central body (kg)
- v: Velocity (m/s)
- r: Distance from the central body (m)

5.33 Eccentricity Vector

The eccentricity vector is:

$$\vec{e}_{\text{vec}} = \frac{\vec{v} \times \vec{h}}{GM} - \frac{\vec{r}}{r}$$

5.34 Eccentricity

The magnitude of the eccentricity vector gives the orbital eccentricity:

$$e = \|\vec{e}_{\text{vec}}\|$$

Where:

- e: Orbital eccentricity (dimensionless)

- $\vec{e}_{\text{vec}}$: Eccentricity vector
- GM: Gravitational parameter ($G \cdot M$)
- $\vec{r}$: Position vector (m)
- $\vec{v}$: Velocity vector (m/s)
- $\vec{h}$: Angular momentum vector (m^2/s)

5.34.1

6 Protocol Specification

6.0.1

This code establishes the foundational components required for a spacecraft simulation and an AI-based control system.

Required Tool:

Python libraries are integrated into the project for computation, plotting, animation, and AI modeling.

Preparing the GPU:

Configurations are set to leverage the computational power of the GPU, accelerating the simulations.

Simulation Settings:

Key parameters such as simulation runtime, time flow, and spacecraft power are defined. Additionally, potential failure or leakage rates are configured in this section.

Fundamental Physics Constants:

To ensure a realistic simulation, physical constants such as gravity, mass, and the speed of light are established.

In Summary:

This section prepares the groundwork for the simulation by defining the tools, GPU configurations, simulation parameters, and physical constants, ensuring the system is ready to operate.

code line (1-73)

RelativisticSpacePhysics Class

The `RelativisticSpacePhysics` class is an abstract data type designed to model general relativity and special relativity effects of space-time. This class contains the following basic components and functions:

Data Fields (Attributes):

- ***device:*** Specifies the device (GPU or CPU) on which the computations will be performed.
- ***c:*** Represents the speed of light constant (m/s).
- ***G:*** Represents the gravitation constant ($m^3/kg/s^2$).
- ***sunmass:*** Represents the mass of the sun (kg).
- ***rs:*** Represents the Schwarzschild radius (the event horizon radius of a black hole).

Initializer Method `init`:

Takes the device parameter and determines the device on which the class will run. It defines the speed of light (`c`), the gravitational constant (`G`), the solar mass (`sunmass`), and the Schwarzschild radius (`rs`). Initializes the basic physical constants of the simulation.

Methods

schwarzschild_metric(r): Calculates the Schwarzschild metric. This metric represents the curvature of space-time around a black hole and includes components such as time dilation (`gtt`), radial curvature (`grr`), and angular components (`gthth`, `gphph`).

- ***Input:*** r (distance from the Sun).
- ***Output:*** A beam containing the values `gtt`, `grr`, `gthth`, `gphph`.
- ***Function:*** Calculates the curvature of space-time around a black hole. Returns `None` in case of error.

calculate_relativistic_force(position, velocity, mass): Calculates the force including the effects of relativity. This calculation combines both special and general relativity effects. In addition to the Newtonian force, it applies the Lorentz factor, perihel shift, time dilation, and spatial curvature corrections.

- ***Input:*** `position` (position vector), `velocity` (velocity vector), `mass` (mass).
- ***Output:*** Relativity-corrected force vector.
- ***Function:*** Applies relativity corrections to the classical Newtonian force and limits the force to the speed of light limit. Returns the classical force in case of error.

update_planet_motion(planet, dt): Updates the motion of the planets with the Runge-Kutta-Fehlberg (RKF45) method. It takes into account relativity effects and special relativity corrections. It adaptively adjusts the step size according to the error tolerance and limits the speed so that it does not exceed the speed of light.

- ***Input:*** `planet` (planetary object), `dt` (time step).
- ***Output:*** Updated position and velocity vectors.
- ***Function:*** Simulates the motion of the planet with high accuracy. Returns the current position and velocity in case of error.

calculatelightbending(impact_parameter): Calculates the light bending. This is Einstein's general relativity prediction.

- ***Input:*** `impact_parameter` (impact parameter).
- ***Output:*** Light deflection angle.
- ***Function:*** Calculates how much light is deflected by the gravitational field. Returns `0.0` in case of error.

gravitationalredshift(r): Calculates the gravitational redshift. This shows the change in the wavelength of light in strong gravitational fields.

- ***Input:*** r (distance from the Sun).
- ***Output:*** Redshift value.
- ***Function:*** Calculates the change in the wavelength of light in a gravitational field. Returns `0.0` in case of error.

General Summary:

The `RelativisticSpacePhysics` class is a tool for modeling the relativistic effects of space-time. This class improves the accuracy of space simulations by combining both special and general relativistic effects. It includes fundamental relativity concepts such as the Schwarzschild metric, light divergence, and gravitational redshift. This class is especially necessary to more accurately model the motion of high-speed and high-mass objects. The class extends the fundamental laws of physics with relativistic effects, forming the basis for complex space simulations.

(76-239)

GeneticTransformerOptimizer Class

The `GeneticTransformerOptimizer` class is an abstract data type that combines genetic algorithm and Transformer architecture to optimize spacecraft control policies. This class contains the following basic components and functions:

Data Fields (Attributes):

- ***device:*** Specifies the device (GPU or CPU) on which the computations will be performed.
- ***population_size:*** Specifies the number of individuals in the genetic algorithm population.
- ***num_generations:*** Specifies the number of generations the genetic algorithm will run.
- ***mutation_rate:*** Specifies the mutation rate in the genetic algorithm.
- ***d_model:*** Specifies the input and output size (embedding size) of the Transformer model.
- ***n_head:*** Specifies the number of multi-head attention heads in the Transformer model.

- ***n_ layers:*** Specifies the number of layers in the Transformer encoder.
- ***d_ ff:*** Specifies the hidden layer size of the transformer feed-forward network.
- ***transformer:*** Transformer encoder model of the SAC (Soft Actor-Critic) network.
- ***optimizer:*** The optimization algorithm used to update the parameters of the transformer model (Adam).
- ***scaler:*** Automatic mixed precision (AMP) gradient scaler used to scale gradients during training.

Initializer Method (`init`):

- Takes the class properties (`population_size`, `num_generations`, `mutation_rate`, `d_model`, `n_head`, `n_layers`, `d_ff`) and initializes the required data fields.
- Creates an `nn.TransformerEncoder` model according to the specified parameters.
- Creates the `optim.Adam` optimization algorithm to optimize the parameters of the model and the `torch.amp.GradScaler` object to scale the gradients.

Methods:

create_ population(): Creates the initial population. This population consists of random tensors, each representing 360-degree orbit points.

- ***Output:*** A tensor (`[population_size, 360, 2]`) representing the population.
- ***Function:*** Creates the initial population of the genetic algorithm.

evaluate_ population_ batch(population_ batch, env): Evaluates a population batch in parallel. Runs each trajectory in a simulation environment and calculates the rewards obtained.

- ***Input:*** `population_batch` (batch of trajectories to evaluate), `env` (simulation environment).
- ***Output:*** A tensor containing the rewards obtained for each trajectory.
- ***Function:*** Measures how well each individual (trajectory) in the population performs.

crossover(parent1, parent2): Creates a new trajectory (child) by merging two trajectories with a one-point crossover.

- ***Input:*** `parent1`, `parent2` (parent trajectories).
- ***Output:*** Crossed child trajectory.
- ***Function:*** Combines trajectories to increase genetic diversity.

mutate(trajectory): Mutates a trajectory by adding Gaussian noise.

- ***Input:*** `trajectory` (trajectory to be mutated).
- ***Output:*** Mutated trajectory.
- ***Function:*** Makes small changes to trajectories to increase genetic diversity.

train(env): Runs the main training loop, which includes a combination of the genetic algorithm and reinforcement learning. In each generation, it evaluates the population, selects the best individuals, generates new generations by crossover and mutation, and updates the best trajectory.

- ***Input:*** `env` (simulation environment).
- ***Output:*** Best trajectory and best reward score.
- ***Function:*** Optimizes the parameters (trajectory) of the SAC agent using the genetic algorithm.

physics_loss(state, next_state): Calculates a physics-based loss function. This loss function includes energy and momentum conservation laws, relativity corrections, and encourages the agent to act in accordance with the laws of physics.

- ***Input:*** `state` (current state), `next_state` (next state).
- ***Output:*** Physics loss value (scalar tensor).
- ***Function:*** Forces the agent to exhibit physically realistic behavior.

forward(x): Performs forward propagation (forward pass) of the Transformer model. It runs optimized for batch size and uses memory optimization.

- ***Input:*** `x` (input tensor).
- ***Output:*** Output tensor, action, and log likelihood.
- ***Function:*** Predicts the action using the Transformer model.

_forward_single(x): Performs forward propagation for a single batch.

General Summary:

The `GeneticTransformerOptimizer` class implements a hybrid optimization method that combines genetic algorithms and Transformer-based reinforcement learning. This class is used to optimize spacecraft trajectories and combines the ability to learn both trajectory discovery and control tasks. The Transformer model evaluates each trajectory point, allowing it to learn better control policies even in more complex and dynamic situations. The class enhances the realism of the simulation by encouraging behavior in accordance with physical rules with a physics-based loss function. It makes the training process faster and more effective by using high-performance computing and optimization techniques.

(240-500)

LagrangePoints Class

The `LagrangePoints` class is an abstract data type designed to calculate and store Lagrange points in a planet-sun system. This class contains the following basic components and functions:

Data Fields (Attributes):

- ***mu:*** Reduced mass parameter representing the ratio of planetary mass to solar mass. This value is used to calculate Lagrange points.
- ***orbit_radius:*** Represents the orbit radius of the planet (m).
- ***l1_distance:*** Represents the distance from the Sun to the point L1 between the planet and the Sun (m).
- ***l1:*** Represents the position vector of point L1 relative to the Sun in the orbital plane of the planet (Tensor).
- ***l2_distance:*** Represents the distance from the Sun to the point L2 beyond the planet and the Sun (m).
- ***l2:*** Represents the position vector of point L2 relative to the Sun in the orbital plane of the planet (Tensor).
- ***influence_radius:*** Represents the radius of influence of the Lagrange points (m). This radius specifies the region around the Lagrange point that influences the motion of the spacecraft.

Initializer Method (`init`):

- ***planet_mass:*** Gets the mass of the planet (kg).
- ***planet_orbit_radius:*** Gets the planet's orbit radius (m).
- Calculates *mu* with the formula `planet_mass / (SUN_MASS + planet_mass)`.
- Calculates the approximate distance of point L1 to the Sun with the formula `orbit_radius * (1 - (mu/3)**(1/3))` and stores it as `l1_distance`.
- Initialize the position vector of point L1 to `[l1_distance, 0.0]` and store it as `l1`.
- Calculates the approximate distance of point L2 to the Sun with the formula `orbit_radius * (1 + (mu/3)**(1/3))` and stores it as `l2_distance`.
- Initialize the position vector of point `l2` to `[l2_distance, 0.0]` and store it as `l2`.
- Calculate the radius of influence of the Lagrange points as `0.15 * orbit_radius` and store it as `influence_radius`.
- Defines Lagrange points and influence radii.

General Overview:

The `LagrangePoints` class is a utility tool for modeling Lagrange points (L1 and L2) and influence zones in a planet-sun system. These points are places where spacecraft can stabilize or move with low energy. The class calculates the approximate locations of the Lagrange points and their radius of influence based on basic information such as the mass of the planet and the orbital radius. This information is useful in spacecraft trajectory design and simulations. The class helps improve the physical accuracy of simulations by identifying stable regions around the L1 and L2 points.

(500-512)

PLANETARY_DATA:

The `PLANETARY_DATA` dictionary contains the data of the planets used for the simulation. For each planet, there is information about its mass (kg), orbital radius (m) and Lagrange points (L1 and L2). These data represent the physical properties of the planets and the positions of the Lagrange points.

(513-522)

Planet Class

The `Planet` class is an abstract data type designed to model a planet. This class contains the following basic components and functions:

Data Fields (Attributes):

- ***name:*** Holds the name of the planet (string).
- ***device:*** Specifies the device (GPU or CPU) on which calculations will be performed.
- ***mass:*** A tensor representing the mass (kg) of the planet.
- ***orbit_radius:*** A tensor representing the planet's orbit radius (m).
- ***position:*** A tensor representing the position vector (x, y) of the planet in 2D space.
- ***velocity:*** A tensor representing the velocity vector (vx, vy) of the planet in 2D space.
- ***radius:*** A tensor representing the radius (m) of the planet.
- ***positions:*** A list storing the past positions (x, y) of the planet.
- ***l1_positions:*** A list storing the past positions of the planet's L1 Lagrange point.
- ***l2_positions:*** A list storing the past positions of the planet's L2 Lagrange point.
- ***lagrange:*** A `LagrangePoints` object that manages the planet's Lagrange points.

Initializer Method (`init`):

- Takes the parameters `name` (planet name), `mass` (planet mass), `orbit_radius` (planet orbit radius).
- Defines the basic properties of the planet (`name`, `mass`, `orbit radius`).
- Initially sets the position and velocity vectors to zero.
- Initially sets the radius of the planet to 0.
- Calculates Lagrange points and creates a `LagrangePoints` object.

Methods

gravitational_ acceleration(r): Calculates the gravitational acceleration relative to the position of the planet.

- ***Input:*** `r` (position vector from the planet).
- ***Output:*** Gravitational acceleration vector (tensor).
- ***Function:*** Calculates the acceleration of the planet according to Newton's law of gravity, returning zero acceleration at very close distances.

update_ position_ rk45(dt_ and_ state=None): Updates the position and velocity of the planet using the Runge-Kutta-Fehlberg 4(5) (RKF45) method.

- ***Input:*** `dt_and_state` (tuple with timestep and position/velocity information or just timestep).
- ***Output:*** Updated position and velocity.
- ***Function:*** Simulates the motion of the planet with high precision. Adjusts the time step adaptively according to the error tolerance.

reset(): Returns the planet to its initial position. Recalculates position and velocity using the orbit parameters.

- ***Function:*** Returns the planet to its initial position.

General Overview:

The `Planet` class is a base class for modeling planets. It manages the physical properties and motion of the planet. It accurately simulates the orbital motion of the planet using numerical integration methods such as RKF45. The class calculates the position, velocity and acceleration of the planet and manages Lagrange points. This class forms one of the basic building blocks required for the simulation of the solar system. It enables realistic modeling of planetary motion and forms the basis for spacecraft simulations.

(524-640)

Spacecraft Class

The **Spacecraft** class is a comprehensive abstract data type designed to model a spacecraft. This class manages various features, systems, and behaviors of the spacecraft.

Attributes:

- ***device:*** Specifies the device (GPU or CPU) for computation.
- ***mass:*** A tensor representing the total mass of the spacecraft (kg).
- ***dry_mass:*** A tensor representing the dry mass of the spacecraft (kg).
- ***velocity:*** A tensor representing the velocity vector (vx, vy) of the spacecraft in 2D space.
- ***acceleration:*** A tensor representing the acceleration vector (ax, ay) of the spacecraft in 2D space.
- ***position:*** A tensor representing the position vector (x, y) of the spacecraft in 2D space.
- ***thrust_history:*** A list storing the history of thrust magnitudes.
- ***speed_history:*** A list storing the history of speed magnitudes.
- ***acceleration_history:*** A list storing the history of acceleration magnitudes.
- ***position_history:*** A list storing the history of position vectors.
- ***fuel_consumption_history:*** A list storing the history of fuel consumption.
- ***orbit_parameters_history:*** A list storing the history of orbital parameters.
- ***main_engine_thrust:*** A tensor representing the thrust force (N) of the main engine.
- ***backup_engines:*** An integer representing the number of backup engines.
- ***current_engine:*** An integer representing the index of the currently active engine.
- ***engine_efficiency:*** A tensor representing the efficiency of the engine.
- ***engine_temperature:*** A tensor representing the temperature (K) of the engine.
- ***min_engine_temp:*** A tensor representing the minimum operating temperature (K) of the engine.
- ***max_engine_temp:*** A tensor representing the maximum operating temperature (K) of the engine.
- ***optimal_temp_range:*** A tuple representing the optimal temperature range for engine operation.
- ***heating_rate:*** A tensor representing the heating rate (K/s) of the engine.
- ***cooling_rate:*** A tensor representing the cooling rate (K/s) of the engine.

- ***fuel_consumption_rate:*** A tensor representing the fuel consumption rate (kg/s).
- ***has_fuel_leak:*** A boolean indicating whether there is a fuel leak.
- ***fuel_leak_rate:*** A tensor representing the fuel leak rate (kg/s).
- ***fuel_system_health:*** A tensor representing the health of the fuel system.
- ***hull_integrity:*** A tensor representing the hull integrity of the spacecraft.
- ***hull_damage:*** A tensor representing the damage to the spacecraft's hull.
- ***critical_damage_threshold:*** A tensor representing the critical damage threshold.
- ***repair_in_progress:*** A boolean indicating if repairs are underway.
- ***repair_time_remaining:*** A tensor representing the remaining repair time.
- ***repair_efficiency:*** A tensor representing the repair efficiency.
- ***auto_repair_systems:*** A boolean indicating whether automatic repair systems are enabled.
- ***navigation_accuracy:*** A tensor representing the accuracy of the navigation system.
- ***guidance_system_health:*** A tensor representing the health of the navigation system.
- ***trajectory_error:*** A tensor representing the error in trajectory tracking.
- ***power_level:*** A tensor representing the power level of the spacecraft.
- ***solar_panel_efficiency:*** A tensor representing the efficiency of the solar panels.
- ***battery_charge:*** A tensor representing the battery charge level.
- ***thermal_load:*** A tensor representing the thermal load.
- ***cooling_system_efficiency:*** A tensor representing the efficiency of the cooling system.
- ***radiator_performance:*** A tensor representing the performance of the radiators.
- ***life_support_health:*** A tensor representing the health of the life support systems.
- ***oxygen_level:*** A tensor representing the oxygen level.
- ***pressure_integrity:*** A tensor representing the pressure integrity.
- ***communication_strength:*** A tensor representing the communication strength.
- ***signal_quality:*** A tensor representing the signal quality.
- ***data_transmission_rate:*** A tensor representing the data transmission rate.
- ***delta_v_remaining:*** A tensor representing the remaining delta-v.
- ***mission_time:*** A tensor representing the mission duration.

- ***total_ distance_ traveled:*** A tensor representing the total distance traveled.
- ***system_ health:*** A dictionary containing the health of systems (propulsion, navigation, power, thermal, life_support, communication, structural).
- ***emergency_ systems_ active:*** A boolean indicating whether emergency systems are active.
- ***emergency_ power_ reserve:*** A tensor representing the emergency power reserve.
- ***emergency_ protocols:*** A dictionary indicating active emergency protocols (engine_shutdown, power_conservation, damage_control, life_support_backup).
- ***sensors:*** A dictionary containing sensor readings (radiation, temperature, pressure, acceleration, magnetic_field).
- ***limits:*** A dictionary containing performance limits (max_acceleration, max_speed, max_temp, min_temp, max_radiation, max_pressure).

Constructor (`__init__`):

- Takes ***mass*** as a parameter to set the initial mass of the spacecraft.
- Initializes all other attributes to their default values, defining the initial state of the spacecraft's systems, including propulsion, fuel, structural integrity, navigation, power, life support, communication, and emergency systems.

Methods:

apply_ action(action): Applies the specified action to the spacecraft. The action includes the thrust direction and magnitude. Calculates acceleration and initiates the spacecraft's motion.

update(dt): Updates the spacecraft's state over a given time step (`dt`). This includes updating position, velocity, fuel state, engine temperature, repair processes, and emergency systems.

- ***Input:*** `dt` (time step).
- ***Function:*** Updates the physical and system states of the spacecraft over time.

_ update_ history(): Records historical values such as thrust, velocity, acceleration, position, and fuel consumption.

get_ system_ status(): Returns a dictionary containing the status of the spacecraft's key systems (hull integrity, fuel, power, engine health, etc.).

get_engine_efficiency(): Calculates engine efficiency based on temperature.

- ***Output:*** Engine efficiency value.
- ***Function:*** Computes efficiency based on engine temperature.

apply_thrust(action): Applies thrust to the spacecraft. Adjusts the motion of the spacecraft using thrust direction, magnitude, and correction maneuvers. Updates fuel consumption and engine temperature.

- ***Input:*** `action` (thrust direction, magnitude, and correction maneuver).
- ***Function:*** Applies thrust to the spacecraft.

apply_force(force, thrust_direction=None, thrust_magnitude=0.0, correction=0.0): Applies a force to the spacecraft. Combines external and thrust forces to update the spacecraft's motion. Monitors engine temperature and fuel consumption.

- ***Input:*** `force` (force vector), `thrust_direction` (thrust direction vector), `thrust_magnitude` (thrust magnitude), `correction` (correction factor).
- ***Function:*** Applies the specified force and updates the spacecraft's motion.

apply_gravity_forces(planets): Applies gravitational forces from planets to the spacecraft.

- ***Input:*** `planets` (list of planets).
- ***Function:*** Calculates and applies gravitational forces from all planets.

get_emergency_status(): Returns a dictionary with the status of emergency systems (e.g., emergency active, engine shutdown, etc.).

get_sensor_readings(): Returns a dictionary of sensor readings (radiation, temperature, pressure, etc.).

get_performance_metrics(): Returns a dictionary containing performance metrics (delta-v, mission duration, distance traveled, current speed, current acceleration, etc.).

- ***Output:*** Dictionary of performance metrics.
- ***Function:*** Evaluates spacecraft performance.

update_emergency_systems(time_step): Updates emergency systems based on engine temperature, power level, life support systems, and structural integrity. Issues alerts or safely shuts down engines if necessary.

leapfrog_step(dt, force): Updates the motion of the spacecraft using the leapfrog integration method. Calculates acceleration, updates velocity by half a step, and updates position.

update_orbit_parameters(): Updates and records the spacecraft's orbital parameters (semi-major axis, eccentricity, inclination, angular momentum).

update_mass(fuel_consumed): Updates the spacecraft's mass based on fuel consumption.

- ***Input:*** `fuel_consumed` (amount of fuel consumed).
- ***Function:*** Adjusts mass based on fuel usage.

handle_motor_failure(): Manages motor failure, activates backup engines, or issues alerts in case of failure.

- ***Output:*** Returns `True` or `False` in case of failure.
- ***Function:*** Handles motor failure.

handle_fuel_leak(): Manages fuel leaks, determines the leakage rate, and issues alerts if a leak is detected.

- ***Output:*** Returns `True` or `False` if leakage occurs.
- ***Function:*** Manages fuel leaks.

handle_microasteroid_collision(): Manages micro-asteroid collisions, calculates hull damage, initiates repairs, and issues alerts.

- ***Output:*** Returns `True` or `False` if a collision occurs.
- ***Function:*** Handles micro-asteroid collisions.

calculate_orbital_elements(params): Calculates position and velocity based on the provided orbital parameters.

General Summary:

The **Spacecraft** class encompasses all the essential features and systems required for a complex spacecraft simulation. It models the spacecraft's physical properties, motion, system states, emergency scenarios, and sensor data, providing a comprehensive foundation for general spacecraft simulations and reinforcement learning-based control algorithms. By accurately simulating the dynamic behavior of the spacecraft, this class serves as a powerful tool for space missions and control system development.

(642-1305)

TransformerMemory Class

The `TransformerMemory` class is a Transformer-based abstract data type designed to provide a memory mechanism for a reinforcement learning agent. This class stores past experiences, calculates their importance, and recalls the most relevant experiences, helping the agent in long-term learning.

Data Fields (Attributes):

- ***state_ size:*** Specifies the size of the state vector.
- ***action_ size:*** Specifies the size of the action vector.
- ***memory_ size:*** Specifies the capacity of the memory, i.e. the maximum number of experiences to store.
- ***d_ model:*** Specifies the input and output size (embedding size) of the Transformer model.
- ***nhead:*** Specifies the number of multi-head attention heads in the Transformer model.
- ***num_ layers:*** Specifies the number of Transformer encoder layers.
- ***device:*** Specifies the device (GPU or CPU) on which the computations will be performed.
- ***memory:*** A memory matrix (Tensor) that stores experiences. Each experience contains state, action, reward and next state information.
- ***pos_ encoder:*** Sinusoidal position encoding parameter used to add position information to the input data of the Transformer model.
- ***input_ proj:*** Linear layer used to transform the input data into a format suitable for the Transformer model.
- ***transformer:*** Transformer encoder model.
- ***output_ proj:*** Linear layer used to convert Transformer output to action size.
- ***current_ idx:*** An integer following the index to which the next experience in memory will be added.
- ***importance_ net:*** A neural network that calculates the importance of the experience.

Initializer Method (`init`):

- Takes the parameters `state_size` (state size), `action_size` (action size), `memory_size` (memory size).
- Initializes necessary components such as memory, position encoding, input projection, Transformer encoder, output projection and importance network.
- Fills the memory with zeros and sets the memory index to 0.

Methods:

_ create_ position_ encoding(): Creates a sinusoidal position encoding. This encoding helps the Transformer model to understand the position information in the input data.

- ***Output:*** Position encoding tensor.
- ***Function:*** Adds position information to the input data of the Transformer model.

add_ experience(state, action, reward, next_ state): Adds a new experience to memory.

- ***Input:*** `state` (current state), `action` (action), `reward` (reward), `next_state` (next state).
- ***Function:*** Adds the new experience to memory and discards the least important experience when memory is full.

query_ memory(current_ state, k=10): Finds the k most similar experiences to the current state in memory. This method encodes the current state, compares it to the memory contents and returns the experiences with the highest similarity score.

- ***Input:*** `current_state` (current state), `k` (number of most similar experiences).
- ***Output:*** Output of the k most similar experiences.
- ***Function:*** Queries memory for the most similar experiences to the current state.

get_ memory_ influence(state): Calculates the influence of memory on the current state. This method calculates the norm of the memory's output and returns an influence value (between 0 and 1) via a sigmoid function.

- ***Input:*** `state` (current state).
- ***Output:*** The influence value of the memory (between 0 and 1).
- ***Function:*** Measures the effect of memory on the agent's behavior.

General Description:

The `TransformerMemory` class provides a memory mechanism for reinforcement learning agents. Using the Transformer architecture, it stores experiences in memory, determines the importance of experiences, and helps the agent learn better by recalling appropriate experiences. The class plays a critical role for long-term learning and contextual knowledge management, thus enabling the agent to accomplish more complex tasks. The influence of memory and the ability to query relevant experiences makes the agent's learning process more efficient and effective.

(1307-1451)

SACNetwork Class

The `SACNetwork` class is an abstract data type designed to define neural network architectures used in the Soft Actor-Critic (SAC) algorithm. This class contains the necessary layers and optimization mechanisms for the agent to learn policy (actor) and value functions (critic and value).

Attributes:

- ***device:*** Specifies the device (GPU or CPU) where computations will be performed.
- ***state_ size:*** Specifies the dimension of the state vector.
- ***action_ size:*** Specifies the dimension of the action vector.
- ***actor:*** Sequential neural network layers forming the actor network. This network takes state input and produces the mean and logarithm of standard deviation (`log_std`) of the action distribution.
- ***q1_ network:*** Sequential neural network layers forming the first Q-network (critic). This network takes state and action inputs to produce Q-value.
- ***q2_ network:*** Sequential neural network layers forming the second Q-network (critic). This network takes state and action inputs to produce Q-value.
- ***value_ network:*** Sequential neural network layers forming the value network. This network takes state input to estimate the value function.
- ***target_ value_ network:*** Target version of the value network. This network is used to make more stable updates during learning.
- ***mean_ head:*** Linear layer producing mean value from actor network output.
- ***log_ std_ head:*** Linear layer producing logarithmic standard deviation (`log_std`) from actor network output.
- ***actor_ optimizer:*** Optimization algorithm (Adam) used to optimize actor network parameters.
- ***q1_ optimizer:*** Optimization algorithm (Adam) used to optimize first Q-network parameters.
- ***q2_ optimizer:*** Optimization algorithm (Adam) used to optimize second Q-network parameters.
- ***value_ optimizer:*** Optimization algorithm (Adam) used to optimize value network parameters.
- ***log_ alpha:*** Learnable parameter used for entropy regulation.
- ***alpha_ optimizer:*** Optimization algorithm (Adam) used to optimize `log_alpha` parameter.
- ***target_ entropy:*** Target value for entropy adjustment.
- ***physics_ encoder:*** Neural network encoding physics-based features.
- ***conservation_ layer:*** Layer producing outputs representing conservation laws.
- ***relativistic_ layer:*** Layer producing outputs representing relativistic effects.

- ***scaler:*** Automatic mixed precision (AMP) gradient scaler used to scale gradients during training.
- ***initial_lr:*** Initial learning rate.
- ***min_lr:*** Minimum learning rate.
- ***lr_decay:*** Learning rate decay coefficient.
- ***max_grad_norm:*** Gradient clipping norm.
- ***buffer_size:*** Maximum capacity of experience replay buffer.
- ***batch_size:*** Batch size used for training.
- ***replay_buffer:*** Replay buffer where experiences are stored.
- ***per_alpha:*** Priority exponent used for prioritized experience replay.
- ***per_beta:*** Sampling correction factor used for prioritized experience replay.
- ***per_epsilon:*** Small positive value used for prioritized experience replay.

Initializer Method (`init`):

- Takes `state_size` and `action_size` parameters.
- Defines actor, Q-networks, value network, mean and `log_std` heads, physics encoder, and related layers.
- Defines learning rate adaptation, prioritized experience replay, and gradient clipping parameters.
- Initializes all model parameters with appropriate initial values (Xavier uniform).

Methods:

_init_weights(module): Initializes neural network layers with Xavier/Glorot initialization.

- ***Input:*** `module` (neural network module).
- ***Function:*** Ensures weights are initialized with appropriate values.

forward_critic(state, action): Performs forward propagation. Takes state and action inputs and returns two Q-values.

- ***Input:*** `state` (state tensor), `action` (action tensor).
- ***Output:*** Two Q-values.
- ***Function:*** Makes value estimation using Q networks.

forward_value(state): Performs forward propagation. Takes state input and returns value.

- ***Input:*** `state` (state tensor).
- ***Output:*** Value.
- ***Function:*** Makes value estimation using value networks.

forward(state): Performs forward propagation of actor network and returns action distribution parameters (mean and `log_std`). Provides dimension check, NaN check, and physics-interactive output.

- ***Input:*** `state` (state tensor).
- ***Output:*** Mean, `log_std` values.
- ***Function:*** Calculates action distribution based on state.

sample_action(state): Samples an action for a given state according to specified parameters (mean, `log_std`).

- ***Input:*** `state` (state tensor).
- ***Output:*** Sampled action and log probability.
- ***Function:*** Samples action from normal distribution for action selection.

physics_forward(state): Performs forward propagation of physics model. Takes state input and returns physics-based output.

- ***Input:*** `state` (state tensor).
- ***Output:*** Physics-based features.
- ***Function:*** Produces physics-based output.

update_learning_rate(episode): Adaptively updates learning rate.

- ***Function:*** Updates learning rate.

add_experience(state, action, reward, next_state, done): Adds new experience to replay buffer and calculates its priority.

- ***Input:*** `state` (current state), `action` (action), `reward` (reward), `next_state` (next state), `done` (episode end).
- ***Function:*** Adds new experience to priority replay buffer.

sample_ batch(): Samples a batch from replay buffer using prioritized sampling method.

- ***Output:*** Sampled batch.
- ***Function:*** Selects experience samples from replay buffer for training.

update(batch): Updates networks (critic, value, actor) using specified batch data. Calculates and optimizes value, Q, and policy losses, and updates target networks.

- ***Input:*** `batch` (sampled experiences).
- ***Function:*** Updates neural network parameters and synchronizes target network.

General Summary:

The `SACNetwork` class defines the neural networks that form the core components of the SAC algorithm. The class contains actor, critic, and value networks and provides optimization mechanisms and loss functions to train these networks. Advanced techniques such as memory, importance-based sampling, automatic learning rate adjustment are used to achieve a stable and efficient learning process. It enhances physical accuracy with physics-based networks and loss function, enabling the agent to exhibit more realistic behaviors. This class forms the foundation of reinforcement learning-based control algorithms.

(1453-1899)

SACAgent Class

The `SACAgent` class is an abstract data type that implements the Soft Actor-Critic (SAC) algorithm. This class represents an agent model that interacts with an environment, stores experiences, and learns a policy using the SAC algorithm.

Attributes:

- ***device:*** Specifies the device (GPU or CPU) where computations will be performed.
- ***state_ size:*** Specifies the dimension of the state vector.
- ***action_ size:*** Specifies the dimension of the action vector.
- ***sac:*** An instance of the neural network model (`SACNetwork`) implementing the SAC algorithm.
- ***gamma:*** Discount factor. Determines how much future rewards are discounted to present value.
- ***tau:*** Coefficient used for soft updates of target networks.
- ***batch_ size:*** Size of experience batch used in learning step.
- ***memory_ size:*** Maximum capacity of experience replay buffer.

- ***memory:*** List storing experiences.
- ***target_ entropy:*** Target entropy value.
- ***log_ alpha:*** Learnable parameter for entropy adjustment.
- ***alpha_ optimizer:*** Optimization algorithm (Adam) used to optimize `log_alpha` parameter.
- ***value_ optimizer:*** Optimization algorithm (Adam) used to optimize value network parameters.
- ***q1_ optimizer:*** Optimization algorithm (Adam) used to optimize first Q-network parameters.
- ***q2_ optimizer:*** Optimization algorithm (Adam) used to optimize second Q-network parameters.
- ***actor_ optimizer:*** Optimization algorithm (Adam) used to optimize actor network parameters.
- ***scaler:*** Automatic mixed precision (AMP) gradient scaler used to scale gradients during training.

Initializer Method (`init`):

- Takes `state_size` and `action_size` parameters.
- Initializes SAC networks (`SACNetwork`), hyperparameters (`gamma`, `tau`, `batch_size`, `memory_size`), and optimization tools.

Methods:

remember(state, action, reward, next_ state, done): Records experience (transition) to memory.

- ***Input:*** `state` (current state), `action` (action), `reward` (reward), `next_state` (next state), `done` (episode end).
- ***Function:*** Adds new experience to memory, removes oldest experience when memory is full.

act(state): Selects an action based on a given state. Samples action using SAC policy.

- ***Input:*** `state` (current state).
- ***Output:*** Sampled action (numpy array).
- ***Function:*** Selects action for a given state.

forward_value(states): Performs forward pass of value network.

- ***Input:*** `states` (state tensor).
- ***Output:*** Value (tensor).
- ***Function:*** Makes value estimation using value network.

forward_critic(states, actions, network=None): Performs forward pass of critic network.

- ***Input:*** `states` (state tensor), `actions` (action tensor), `network` (optional critic network).
- ***Output:*** Critic Q-value (tensor).
- ***Function:*** Makes Q-value estimation using critic network.

sample_action(states): Samples actions for states.

- ***Input:*** `states` (state tensor).
- ***Output:*** Action and log probability.
- ***Function:*** Samples action using actor network.

replay(): Trains SAC networks by taking a random batch from memory. Calculates value, Q, and policy losses and updates networks. Applies soft target network updates.

- ***Output:*** Dictionary containing loss values.
- ***Function:*** Trains agent's networks using experiences.

save_model(path): Saves agent's model parameters and optimizer states to given path.

- ***Input:*** `path` (path to save).
- ***Function:*** Saves model to disk.

load_model(path): Loads previously saved model parameters and optimizer states from given path.

- ***Input:*** `path` (path of model to load).
- ***Function:*** Restores saved model parameters.

General Summary:

The `SACAgent` class represents a reinforcement learning agent using the Soft Actor-Critic algorithm. This class includes functions for action selection, experience recording, learning from memory, and model saving/loading. Using the SAC algorithm, the agent learns an optimal policy (behavior strategy) by interacting with an environment and learning from its experiences. The class can be used in both training and testing phases and provides necessary tools for implementing reinforcement learning-based control algorithms. The `SACAgent` class can be used to manage complex space missions using the SAC algorithm and optimizes the learning process by interacting with dynamic environments.

(1901-2078)

LongTermSACNetwork Class

The `LongTermSACNetwork` class is an abstract data type derived from `SACNetwork` that integrates the Soft Actor-Critic (SAC) algorithm with a long-term memory mechanism. This class helps the agent make more effective decisions using both short-term experiences and long-term memory content.

Attributes:

- ***device:*** Specifies the device (GPU or CPU) for computations.
- ***state_size:*** Specifies state vector dimension.
- ***action_size:*** Specifies action vector dimension.
- ***memory:*** A `TransformerMemory` object managing the long-term memory mechanism.
- ***actor:*** Sequential neural network layers forming the actor network.
- ***mean_head:*** Linear layer producing mean value from actor network output.
- ***log_std_head:*** Linear layer producing logarithmic standard deviation from actor network output.
- ***memory_integration:*** Dictionary holding memory integration layers. These layers combine state and action information with memory outputs and contain gate mechanisms.
- ***state_gate:*** Used to create gate by combining current state and state from memory.
- ***action_gate:*** Used to create gate by combining current action and action from memory.

Initializer Method (`init`):

- Takes `state_size` and `action_size` parameters.
- Calls `SACNetwork`'s initializer and defines long-term memory (`TransformerMemory`) with memory integration layers. Updates actor, mean, and `log_std` layers.

Methods:

forward_actor(state):

- ***Input:*** `state` (state tensor)
- ***Output:*** Mean and `log_std` values
- ***Function:*** Calculates action distribution based on state

forward_with_memory(state):

- ***Input:*** `state` (state tensor)
- ***Output:*** Integrated action and `log_std` values
- ***Function:*** Produces action using long-term memory information for a given state

update_memory(state, action, reward):

- ***Input:*** `state` (current state), `action` (action), `reward` (reward)
- ***Function:*** Adds new experience to long-term memory

General Summary:

`LongTermSACNetwork` class presents a version of the SAC algorithm enriched with long-term memory. It inherits from `SACNetwork` and integrates with a long-term memory mechanism. This integration enables the agent to make smarter and more consistent decisions by considering past experiences and contextual information. The memory mechanism captures long-term dependencies and improves performance for complex tasks. The class supports optimal action selection using both short-term experience and long-term memory, making it particularly effective for agents operating in complex environments or long-term tasks.

(2080-2144)

LongTermSACAgent Class

A formal summary of the `LongTermSACAgent` class: The `LongTermSACAgent` class is a Soft Actor-Critic (SAC) agent derived from `SACAgent`, integrated with a long-term memory mechanism. This class presents a reinforcement learning agent model capable of making more effective decisions using both short-term experiences and long-term memory content.

Attributes:

- ***device:*** Specifies computation device (GPU or CPU)
- ***state_ size:*** Specifies state vector dimension
- ***action_ size:*** Specifies action vector dimension
- ***sac:*** Represents SAC network (`LongTermSACNetwork`) integrated with long-term memory
- ***gamma:*** Discount factor determining how future rewards are discounted to present value
- ***tau:*** Coefficient for soft updates of target networks
- ***batch_ size:*** Size of experience batch used in learning step
- ***memory_ size:*** Maximum capacity of experience replay buffer
- ***memory:*** List storing experiences

Initializer Method (`init`):

- Takes `state_size` and `action_size` parameters
- Calls `SACAgent`'s initializer and integrates `LongTermSACNetwork` with long-term memory features

Methods:

act(state):

- ***Input:*** `state` (current state)
- ***Output:*** Sampled action (numpy array)
- ***Function:*** Selects action using long-term memory via `LongTermSACNetwork`'s `forward_with_memory` method

remember(state, action, reward, next_ state, done):

- ***Input:*** `state`, `action`, `reward`, `next_state`, `done`
- ***Function:*** Records experience to both short-term (replay buffer) and long-term memory

forward_ value(states):

- ***Input:*** `states` (state tensor)
- ***Output:*** Value
- ***Function:*** Makes value estimation using value network

forward_ critic(states, actions, network=None):

- ***Input:*** `states`, `actions`, `network` (optional)
- ***Output:*** Two critic Q-values (tensor)
- ***Function:*** Makes Q-value estimation using critic network

sample_ action(states):

- ***Input:*** `states` (state tensor)
- ***Output:*** Action and log probability
- ***Function:*** Samples action using actor network

replay():

- ***Output:*** Dictionary of loss values
- ***Function:*** Trains agent's networks using experiences from memory

save_ model(path):

- ***Input:*** `path`
- ***Function:*** Saves model to disk

load_ model(path):

- ***Input:*** `path`
- ***Function:*** Loads saved model parameters

General Summary:

`LongTermSACAgent` enhances SAC algorithm capabilities by learning from both short-term and long-term experiences. It provides an agent model capable of making more rational decisions by combining historical and contextual information. This class performs better in complex and long-term tasks, particularly useful in simulations, robotic control systems, and AI applications requiring long-term learning. It improves the basic SAC agent's learning ability through long-term memory integration.

(2146-2187)

GeneticSACAgent Class

The `GeneticSACAgent` class is an abstract data type that uses the principles of genetic algorithm to evolve `SACAgent`s, a reinforcement learning agent. This class aims to improve the performance of SAC agents by managing operations such as creating a population, selection, crossover, mutation and evolution.

Data Fields (Attributes):

- ***device:*** Specifies the device (GPU or CPU) on which the computations will be performed.
- ***state_ size:*** Specifies the size of the state vector.
- ***action_ size:*** Specifies the size of the action vector.
- ***population_ size:*** Specifies the number of individuals (SAC agents) in the population of the genetic algorithm.
- ***batch_ size:*** Specifies the batch size to use when training SAC agents.
- ***gamma:*** Discount factor. It determines how much future rewards will be discounted to the current value.
- ***mutation_ rate:*** Specifies the mutation rate in the genetic algorithm.
- ***crossover_ rate:*** Specifies the crossover rate in the genetic algorithm.
- ***elite_ size:*** Specifies the number of top individuals to be passed on directly to the next generation.
- ***tournament_ size:*** Specifies the number of individuals to be used for tournament selection.
- ***memory:*** The replay buffer where experiences are stored.
- ***memory_ size:*** Maximum capacity of the experience replay buffer.
- ***population:*** A list that stores the SAC agents in the population.
- ***fitness_ scores:*** A tensor that stores the performance (reward) of each agent in the population.
- ***generation:*** An integer holding the current number of generations.
- ***actor:*** Sequential neural network layers that form the actor network shared for the entire population.
- ***value_ network:*** Sequential neural network layers that make up the shared value network for the entire population.
- ***q1_ network:*** Sequential neural network layers forming the first Q-network shared for the whole population.
- ***q2_ network:*** Sequential neural network layers forming the second Q-network shared for the whole population.
- ***q1_ optimizer:*** The optimization algorithm (Adam) used to optimize the parameters of the first Q-network.
- ***q2_ optimizer:*** Optimization algorithm used to optimize the parameters of the second Q-network (Adam).

- ***value_ optimizer:*** Optimization algorithm used to optimize the parameters of the Value network (Adam).
- ***mean_ head:*** Linear layer that generates the mean value from the output of the Actor network.
- ***log_ std_ head:*** Linear layer that generates the logarithm standard deviation (`log_std`) from the output of the Actor network.
- ***target_ entropy:*** Target entropy value.
- ***log_ alpha:*** A learnable parameter used for entropy tuning.
- ***alpha_ optimizer:*** Optimization algorithm (Adam) used to optimize the `log_alpha` parameter.
- ***log_ std_ min:*** Minimum limit for the `log_std` value.
- ***log_ std_ max:*** maximum limit for the `log_std` value.

Initializer Method (`init`):

- Takes the parameters `state_size` (state size), `action_size` (action size) and `population_size` (population size).
- Creates the population, resets the fitness scores, sets the initial generation to 0, defines the common actor model. And launches other related networks and optimization tools.
- Implements the process of resetting the weights

Methods

sample_ action(state): Samples action by state.

- ***Input:*** `state` (state tensor).
- ***Output:*** Sampled action and log probability.
- ***Function:*** Samples action using actor network based on state.

select_ parent(): Selects a parent by tournament selection.

- ***Output:*** Selected parent (SAC agent).
- ***Function:*** Selects a parent by tournament selection.

crossover(parent1, parent2): Creates a child agent by combining the parameters of two parent agents.

- ***Input:*** `parent1`, `parent2` (parent SAC agents).
- ***Output:*** New child agent (SAC agent).
- ***Function:*** Crosses parents to increase genetic diversity.

mutate(agent): Mutates an agent by adding Gaussian noise to its parameters.

- ***Input:*** `agent` (SAC agent to be mutated).
- ***Function:*** makes small changes to the agent's parameters.

evolve(): Evolves the current population. It selects the best individuals, creates a new population by crossover and mutation, and resets the current fitness scores.

- ***Function:*** Evolve the current population.

step_batch(actions): Steps through the environment in batch.

- ***Input:*** `actions`.
- ***Output:*** New states, rewards, chapter breaks and additional information.
- ***Function:*** Interacts with the entire population as a batch in the environment.

update_fitness(agent_idx, reward): Updates the fitness score of a given agent with the given reward.

- ***Input:*** `agent_idx` (agent's index), `reward` (reward).
- ***Function:*** Evaluates an agent based on its performance.

remember_batch(states, actions, rewards, next_states, dones): Adds batches of experiences to memory.

- ***Input:*** `states`, `actions`, `rewards`, `next_states`, `dones`, chapter breaks.
- ***Function:*** Adds new experiences to the replay buffer.

act_batch(states): Selects actions based on the state batch. Instances actions using the common actor network.

- ***Input:*** `states` (state batch).
- ***Output:*** Actions (numpy array).
- ***Function:*** Selects actions for all agents in parallel.

act(state): Selects an action for a single state. Instantiates the action using the shared actor network.

- ***Input:*** `state`.
- ***Output:*** Selected action.
- ***Function:*** Selects action for a single state.

forward_value(state): Performs forward propagation (forward pass) of the value network.

- ***Input:*** `state` (state tensor).
- ***Output:*** Value value.
- ***Function:*** Predicts the value using the Value mesh.

forward(state): Performs forward propagation of the Actor network and returns the parameters of the action distribution (mean and `log_std`).

- ***Input:*** `state` (state tensor).
- ***Output:*** Mean and `log_std` values.
- ***Function:*** Computes the action distribution by state.

forward_critic(states, actions, network=None): Performs forward propagation (forward pass) of the Critic network.

- ***Input:*** `states` (state tensor), `actions` (action tensor), `network` (optional critic network).
- ***Output:*** Two critic Q values.
- ***Function:*** Estimates the Q value using the critic network.

replay(): Trains SAC networks by taking a random batch from memory. Calculates value, Q and policy losses and updates the networks.

- ***Output:*** A dictionary containing the loss values.
- ***Function:*** Trains agents' nets using experience.

save_population(path): Saves the state of all agents in the population to the specified path.

- ***Input:*** `path` (path to the file to be saved).
- ***Function:*** Saves the model parameters of all agents in the population to disk.

General Description:

The `GeneticSACAgent` class provides an approach that combines genetic algorithms with reinforcement learning. This class aims to find a better control policy by managing multiple SAC agents in a population. The performance of SAC agents is improved by the evolutionary processes of the genetic algorithm (selection, crossover, mutation). Thus, better results are achieved by overcoming the limitations of individual SAC agents. This class is a powerful tool for improving and optimizing reinforcement learning algorithms and exploring a wider solution space. Effective in complex environments to achieve higher success rates.

(2188-2653)

LongTermGeneticSACAgent Class

The `LongTermGeneticSACAgent` class is an abstract data type derived from the `GeneticSACAgent` class that evolves SAC agents integrated with a long-term memory mechanism. This class aims to evolve agents that perform better on more complex tasks, using both genetic algorithm principles and long-term memory mechanisms.

Data Fields (Attributes):

- ***device:*** Specifies the device (GPU or CPU) on which the computations will be performed.
- ***state_size:*** Specifies the size of the state vector.
- ***action_size:*** Specifies the size of the action vector.
- ***population_size:*** Specifies the number of individuals (SAC agents) in the population of the genetic algorithm.
- ***batch_size:*** Specifies the batch size to use when training SAC agents.
- ***gamma:*** Discount factor. It determines how much future rewards will be discounted to the current value.
- ***mutation_rate:*** Specifies the mutation rate in the genetic algorithm.
- ***crossover_rate:*** Specifies the crossover rate in the genetic algorithm.
- ***elite_size:*** Specifies the number of top individuals to be passed on directly to the next generation.
- ***tournament_size:*** Specifies the number of individuals to be used for tournament selection.
- ***memory:*** The replay buffer where experiences are stored.
- ***memory_size:*** Maximum capacity of the experience replay buffer.
- ***population:*** A list that stores the `LongTermSACAgent`s in the population.
- ***fitness_scores:*** A tensor that stores the performance (reward) of each agent in the population.

- ***generation:*** An integer holding the current number of generations.
- ***actor:*** Sequential neural network layers that form the actor network shared for the entire population.
- ***value_ network:*** Sequential neural network layers that make up the shared value network for the entire population.
- ***q1_ network:*** Sequential neural network layers forming the first Q-network shared for the whole population.
- ***q2_ network:*** Sequential neural network layers forming the second Q-network shared for the whole population.
- ***q1_ optimizer:*** The optimization algorithm (Adam) used to optimize the parameters of the first Q-network.
- ***q2_ optimizer:*** Optimization algorithm used to optimize the parameters of the second Q-network (Adam).
- ***value_ optimizer:*** Optimization algorithm used to optimize the parameters of the Value network (Adam).
- ***mean_ head:*** Linear layer that generates the mean value from the output of the Actor network.
- ***log_ std_ head:*** Linear layer that generates the logarithm standard deviation (`log_std`) from the output of the Actor network.
- ***target_ entropy:*** Target entropy value.
- ***log_ alpha:*** A learnable parameter used for entropy tuning.
- ***alpha_ optimizer:*** Optimization algorithm (Adam) used to optimize the `log_alpha` parameter.
- ***log_ std_ min:*** Minimum limit for the `log_std` value.
- ***log_ std_ max:*** maximum limit for the `log_std` value.

Initializer Method (`init`):

- Takes the parameters `state_size` (state size), `action_size` (action size), and `population_size` (population size).
- Calls the initializer of the `GeneticSACAgent` class and creates the population using `LongTermSACAgent`s. Initializes other data fields and defines the corresponding parameters.

Methods

crossover(parent1, parent2): Creates a child agent by combining the parameters of two parent agents. It also includes the memory mechanism in the crossover.

- ***Input:*** `parent1`, `parent2` (parent `LongTermSACAgent`s).
- ***Output:*** New child agent (`LongTermSACAgent`).
- ***Function:*** Crosses parents to increase genetic diversity.

select_parent(): Selects a parent by tournament selection.

- ***Output:*** Selected parent (`LongTermSACAgent`).
- ***Function:*** Selects a parent by tournament selection.

mutate(agent): Mutates an agent by adding Gaussian noise to its parameters.

- ***Input:*** `agent` (`LongTermSACAgent` to be mutated).
- ***Function:*** makes small changes to the agent's parameters.

evolve(): Evolves the current population. It selects the best individuals, creates a new population by crossover and mutation, and resets the current fitness scores.

- ***Function:*** Evolve the current population.

step_batch(actions): Steps through the environment in batch.

- ***Input:*** `actions`.
- ***Output:*** New states, rewards, chapter breaks and additional information.
- ***Function:*** Interacts with the entire population as a batch in the environment.

update_fitness(agent_idx, reward): Updates the fitness score of a given agent with the given reward.

- ***Input:*** `agent_idx` (agent's index), `reward` (reward).
- ***Function:*** Evaluates an agent based on its performance.

remember_batch(states, actions, rewards, next_states, dones): Adds batches of experiences to memory.

- ***Input:*** `states`, `actions`, `rewards`, `next_states`, `dones`, chapter breaks.
- ***Function:*** Adds new experiences to the replay buffer.

act_ batch(states): Selects actions based on the state batch. Instances actions using the common actor network.

- ***Input:*** `states` (state batch).
- ***Output:*** Actions (numpy array).
- ***Function:*** Selects actions for all agents in parallel.

act(state): Selects an action for a single state. Instantiates the action using the shared actor network.

- ***Input:*** `state`.
- ***Output:*** Selected action.
- ***Function:*** Selects action for a single state.

forward_ value(state): Performs forward propagation (forward pass) of the value network.

- ***Input:*** `state` (state tensor).
- ***Output:*** Value value.
- ***Function:*** Predicts the value using the Value mesh.

forward(state): Performs forward propagation of the Actor network and returns the parameters of the action distribution (mean and `log_std`).

- ***Input:*** `state` (state tensor).
- ***Output:*** Mean and `log_std` values.
- ***Function:*** Computes the action distribution by state.

forward_ critic(states, actions, network=None): Performs forward propagation (forward pass) of the Critic network.

- ***Input:*** `states` (state tensor), `actions` (action tensor), `network` (optional critic network).
- ***Output:*** Two critic Q values.
- ***Function:*** Estimates the Q value using the critic network.

replay(): Trains SAC networks by taking a random batch from memory. Calculates value, Q and policy losses and updates the networks.

- ***Output:*** A dictionary containing the loss values.
- ***Function:*** Trains agents' nets using experience.

save_population(path): Saves the state of all agents in the population to the specified path.

- ***Input:*** `path` (path to the file to be saved).
- ***Function:*** Saves the model parameters of all agents in the population to disk.

General Description:

The `LongTermGeneticSACAgent` class makes reinforcement learning agent training more complex by combining genetic algorithms and long-term memory mechanisms. This class not only aims to find better solutions using the evolutionary processes of the genetic algorithm, but also provides more consistent and contextually meaningful learning with long-term memory. This approach aims to develop more successful agents, especially in complex environments and tasks that require long-term interactions. `LongTermGeneticSACAgent` improves the long-term learning capability of individual SAC agents, enabling them to exhibit more complex behaviors through a more efficient genetic algorithm.

(2690-2688)

SpaceTravelEnv Class

The `SpaceTravelEnv` class is an abstract data type designed to represent a space travel simulation environment. This class manages the planets, the spacecraft, the interactions of these entities, observes the state of the environment, implements transitions based on actions, calculates rewards, and controls chapter endings.

Data Fields (Attributes):

- ***device:*** Specifies the device (GPU or CPU) on which calculations will be performed.
- ***num_envs:*** Specifies the number of parallel simulations.
- ***time_step:*** Specifies the time step (in seconds) of the simulation.
- ***dt:*** A float representing the simulation time step (in seconds).
- ***G:*** A tensor representing the gravitational constant.
- ***sun_mass:*** A tensor representing the mass of the sun.
- ***orbital_parameters:*** A dictionary of planetary orbital parameters (semi-major axis, eccentricity, orbital period, etc.).
- ***physics_engine:*** A `RelativisticSpacePhysics` object used to calculate relativity effects.
- ***planets:*** A dictionary containing the planets in the simulation (`Planet` objects).
- ***destination:*** A `Planet` object representing the target planet.
- ***spacecraft:*** A list containing the spacecraft in the simulation (`Spacecraft` objects).

- ***_state_size:*** An integer holding the size of the environment's state vector.
- ***state_size:*** An integer holding the size of the environment's state vector.
- ***action_size:*** An integer holding the size of the action vector in the environment.
- ***time:*** A tensor representing the simulation time (in seconds).
- ***done:*** A boolean tensor representing whether the section is finished.
- ***fuel_history:*** A list that stores historical fuel levels.
- ***distance_history:*** A list storing past distances.
- ***speed_history:*** A list storing past speeds.
- ***time_history:*** A list storing past time steps.
- ***acceleration_history:*** A list storing past acceleration values.
- ***initial_distance:*** A tensor storing the initial distance.
- ***previous_distance:*** A tensor storing distances from previous steps.
- ***current_step:*** An integer tensor holding the simulation steps.
- ***max_steps:*** An integer holding the maximum number of steps in the simulation section.
- ***population:*** A list of spacecraft in the simulation environment.

Initializer Method (`init`):

- Takes the `destination_planet_name` (name of the destination planet) and the optional `num_envs` (number of parallel simulations) parameters.
- Initializes the physical properties of the simulation environment (gravitation, solar mass, etc.), creates the planets, defines the destination planet, creates the spacecraft and initializes it.
- Calculates state and action dimensions, resets simulation time and episode state (`done`).

Methods

_calculate_state_size(): Calculates the size of the state vector. It contains information such as spacecraft position, velocity, fuel, position and velocity of the target planet.

- ***Output:*** Size of the state vector (integer).
- ***Function:*** Calculates the size of the state vector.

get_ state(): Returns the current state. Creates a tensor containing all relevant information such as spacecraft, target planet and other metrics.

- ***output:*** A tensor containing the state of the simulation environment.
- ***Function:*** Represents the state of the simulation.

step_ batch(actions): Steps through the environment in batch. For each spacecraft it applies actions, observes the new state and calculates rewards.

- ***Input:*** `actions` (batch of action vectors).
- ***Output:*** New states, rewards, chapter breaks, additional information.
- ***Function:*** Updates the environment for all spacecraft.

get_ state_ tensor(): Gets the current state in tensor format.

- ***Output:*** Tensor representing the simulation state
- ***Function:*** Returns the position as a tensor.

update_ planet_ position(planet, dt): Updates the position of the specified planet according to Kepler's laws.

- ***Input:*** `planet` (planet name or `Planet` object), `dt` (time step).
- ***Function:*** Updates the position and velocity of the planet.

_ create_ planets(): Creates planets and sets their initial positions. Sets the initial positions for each planet using the orbit parameters.

- ***Output:*** Planets dictionary (`Planet` objects).
- ***Function:*** Creates the planets to be used in the simulation.

reset(): Returns the environment to the initial state. Sets the spacecraft position and velocity to initial values, resets the time and partition state, updates the position of the target planet and returns the initial state.

- ***Output:*** Initial state of the environment.
- ***Function:*** Returns the environment to its initial state.

reset_ batch(batch_ size): Resets the environment in batch.

- ***Input:*** `batch_size` (batch size to reset)
- ***Exit:*** Initial state of the media.
- ***Function:*** Returns all environments to initial state.

step(action): Takes a step in the environment. It applies the action to the spacecraft, updates the positions of the planets, gets the new state, calculates the reward and checks if the episode is over.

- ***Input:*** `action` (action vector).
- ***Output:*** New state, reward, end of episode information, additional information.
- ***Function:*** Simulates a step in the environment.

_check_emergencies(time_step): Checks for possible emergencies (engine failure, fuel leak, microasteroid collision, etc.) and takes necessary actions.

- ***Input:*** `time_step` (elapsed time).
- ***Output:*** Emergency information and penalty.
- ***Function:*** Simulates possible emergency situations.

_calculate_reward(spacecraft_index=None): Calculates a reward value based on the performance of the spacecraft in the simulation environment. This reward takes into account factors such as proximity to the target, speed, fuel economy, etc.

- ***Input:*** Optional `spacecraft_index` (spacecraft index).
- ***Output:*** Reward (float).
- ***Function:*** Calculates a reward to evaluate the agent's performance.

_check_done(spacecraft_index): Checks if the simulation part is finished. Checks for arrival at the destination, running out of fuel, collision, etc. to determine if the episode is finished.

- ***Introduction:*** Spacecraft index
- ***Exit:*** Whether the section ends or not (boolean).
- ***Function:*** Checks the end of the section.

adaptive_time_step(min_dt=(86400/2), max_dt=86400*15): Calculates an adaptive time step. The time step can vary according to the orbital periods of the planets, the speed of the spacecraft and the distance to the target.

_initialize_spacecraft(): Initializes the spacecraft and calculates the maximum fuel.

_calculate_minimum_fuel(): Calculates the minimum amount of fuel required for the spacecraft to reach the destination.

_update_history(): Saves metrics collected during the simulation such as fuel, distance, speed, time and acceleration.

_check_episode_end(): Checks the end of the simulation episode.

General Overview:

The `SpaceTravelEnv` class is a centralized class designed to manage space travel simulation. This class manages all simulation entities (planets and spacecraft), their interactions and the rules of the simulation. It also facilitates the training and testing of agents. It covers functions such as observation, interaction, reward calculation and episode management. It provides all the necessary infrastructure for reinforcement learning based space simulations and facilitates the training process of agents. This class is a powerful tool for managing complex space missions and learning the best control policies for these missions.

(2690-3446)

TrainingTimeEstimator Class

The `TrainingTimeEstimator` class is an abstract data type designed to estimate the duration and track the progress of a reinforcement learning training process. This class manages a timer that is initialized at the beginning of the training process and calculates the estimated time remaining, total time, elapsed time and completion percentage at the end of each episode. It also tracks GPU utilization.

Data Fields (Attributes):

- ***num_episodes:*** An integer representing the total number of episodes to run during the training process.
- ***start_time:*** A float representing the start time (in seconds) of the training process.
- ***completed_episodes:*** An integer representing the number of completed episodes.
- ***episode_times:*** A deque (double-ended queue) with a maximum size of 100 that stores the times (in seconds) of the last completed episodes.

Initializer Method (`init`):

- Takes the parameter `num_episodes` (total number of training episodes).
- Sets the start time (`start_time`) to `None`, sets the number of completed episodes (`completed_episodes`) to 0 and creates a deque to hold the episode times.

Methods

start(): Starts the training timer. Saves the start time (`start_time`) and resets the number of completed episodes (`completed_episodes`) to zero.

update(episode): Updates the training metrics at the end of each episode, calculating the estimated time remaining, total estimated time, elapsed time, completion percentage and GPU utilization.

- ***Input:*** `episode` (completed episode number).
- ***Output:*** A dictionary containing the calculated time metrics.
- ***Function:*** Tracks the progress of the training process.

Overview:

The `TrainingTimeEstimator` class is an important aid in the management of reinforcement learning training processes. This class helps to organize the training schedule and use resources efficiently by making predictions about how long the training process will take. It also provides metrics that can be used to monitor training progress and performance. This makes training processes more efficient and manageable. It provides the user with information such as how long it takes to train the model, how much is left, how much GPU it uses, and provides ease of monitoring and control throughout the training process.

(3449-3491)

train_ agent Function

The `train_agent` function is a procedure designed to train a reinforcement learning agent for a given target planet. This function sets up the simulation environment, trains the genetic SAC agent, tracks metrics during the training process and returns the training results.

Parameters

- ***destination_ planet:*** Represents the name (string) of the destination planet.
- ***num_ episodes:*** An integer representing the total number of episodes to run in the training process.

Functionality:

Environment and Agent Setup:

- Creates a simulation environment for the specified `destination_planet` using the `SpaceTravelEnv` class. This environment manages planets, spacecraft and physical interactions. The number of parallel simulations is specified as 48.
- Using the `GeneticSACAgent` class, creates a genetic SAC agent that matches the state and action dimensions of the environment. The population size is 48.
- Creates lists to store metrics to be used during training (total rewards, segment losses, best rewards, success rates, minimum distances, fuel consumption, number of steps).

Training Cycle:

- Creates a timer to estimate the training time using the `TrainingTimeEstimator` class.
- Creates a loop through as many training episodes as the specified `num_episodes`.
- Within each episode, resets the simulation environment for the entire population.
- Within each episode, it runs a loop until the specified maximum number of steps (`MAX_STEPS`) is reached for each population member.
- At each step, it takes the agent's actions, steps through the environment, stores experiences and trains the agent's networks.
- It tracks and updates the rewards achieved and other metrics.
- At the end of the episode, it checks the performance metrics and whether the simulations of all population members have ended.
- At the end of each episode, if there is a significant change in the metrics, it prints them.
- At the end of each episode, it evolves the agent's population using a genetic algorithm.

Metric Recording:

- At the end of each episode, it calculates metrics such as average reward, average number of steps, minimum distance, average fuel consumption, success rate.
- Saves these metrics in the relevant lists.
- Prints the training progress and saved metrics on the screen at certain intervals (5 episodes).
- Updates metrics for the GUI.

Error Management:

- Captures errors that occur during the training cycle and prints them on the screen.

Results:

- After the training cycle is complete, returns a tuple containing the trained agent, total rewards, episode losses, success rates, minimum distances and fuel consumption.

Outputs:

- ***trained_agent:*** A `GeneticSACAgent` object representing the trained agent.
- ***total_rewards:*** A list containing the average rewards calculated at the end of each episode.
- ***episode_losses:*** A list containing the average losses calculated at the end of each episode.
- ***success_rates:*** A list containing the success rates calculated at the end of each episode.

- ***min_distances:*** A list containing the minimum distances calculated at the end of each episode.
- ***fuel_consumptions:*** A list containing the average fuel consumptions calculated at the end of each section.

General Summary:

The `train_agent` function is designed to train a genetic SAC agent in a spacecraft simulation environment. This function manages a complex learning process, monitors the agent's performance, records training metrics and aims to obtain an optimized model as a result of training. By combining genetic algorithm and reinforcement learning techniques, it focuses on learning a more effective control policy in space travel missions. This function aims at automation of learning processes to achieve better results, especially in complex scenarios.

(3493-3620)

simulate_journey Function

The `simulate_journey` function is a procedure designed to run a space travel simulation using a trained reinforcement learning agent and record the results of this journey. This function can be used to test the behavior of the trained agent and to visualize the simulation results.

Parameters:

- ***env:*** An object of the `SpaceTravelEnv` class representing the simulation environment.
- ***agent:*** An object of the trained reinforcement learning agent (`SACAgent` or derivative) class.

Functionality:

Environment Reset:

- Initializes the simulation environment (`env`). This step involves placing the spacecraft in its initial position and reinitializing the orbits of the planets.

Journey Simulation:

- Creates an empty list to store the spacecraft position information.
- Creates an empty dictionary to hold simulation metrics (speed, acceleration, fuel level, distance to target, time).
- Creates a loop until the done state of the environment is correct (i.e. until the episode ends).
- At each step:
  - Allows the agent to choose an action based on the current state.

- It takes a step in the environment to check the new state, the reward and whether the episode is finished.
- Adds the obtained reward value to the previous reward value.
- Records the spacecraft's position, velocity, acceleration, fuel level, distance to the target and simulation time in the metrics dictionary.

Metric Record:

- Records the positions of the spacecraft.
- Records the spacecraft's velocity, acceleration, fuel level, distance to target and elapsed time in the metrics dictionary.

Results:

- At the end of the simulation, returns a tuple containing the simulation environment (`env`), the recorded spacecraft positions (numpy array), and the metrics.

Outputs:

- ***env:*** `SpaceTravelEnv` object representing the simulation environment (final state).
- ***spacecraft_positions:*** A numpy array containing the positions recorded by the spacecraft during the journey.
- ***metrics:*** A dictionary containing the velocity, acceleration, fuel, distance, and time recorded throughout the simulation.

General Summary:

The `simulate_journey` function is used to run a simulation of a space journey with a trained reinforcement learning agent and generate detailed records of the journey. This function allows to test the policy learned by the agent in a real simulation environment and makes the results of these tests available for visualization or analysis. The data from the simulation can be used to study the agent's behavior in more detail and evaluate the learning process. It facilitates the monitoring and analysis of the spacecraft's performance, changes during its journey and responses to different scenarios.

(3622-3658)

create_animation Function

The `create_animation` function is a procedure designed to visualize the results of a space travel simulation. This function uses the `matplotlib` library to create an animation that includes the motion of the spacecraft, the orbits of the planets, and performance metrics over time.

Parameters:

- ***env:*** An object of the `SpaceTravelEnv` class representing the simulation environment.
- ***spacecraft_positions:*** A numpy array containing the positions recorded by the spacecraft during the journey.
- ***metrics:*** A dictionary containing the velocity, acceleration, fuel, distance, and time recorded throughout the simulation.
- ***destination_planet_name:*** A string holding the name of the destination planet.

Functionality:

Visualization Settings:

- Implements a dark theme using the `matplotlib.pyplot` library.
- Creates a figure and two sub-axes for animation. The first axis will show the motion of the spacecraft and planetary orbits, the second axis will show the velocity over time.
- Assigns a title to the figure.
- Defines a scaling factor in astronomical units (AU).
- Calculates the maximum of the absolute values of the positions of the spacecraft to determine the plot boundaries.
- Sets the x and y axes, labels and grid settings of the master drawing.
- Draws the sun as a yellow circle.

Planetary Orbits and Lagrange Points:

- For each planet in the simulation:
  - Draws the circle representing the orbit.
  - Draws the positions of the planet's L1 and L2 Lagrange points.
  - Draws the circles representing the radii of influence of the L1 and L2 points.

Spacecraft Track and Position:

- Creates a line representing the motion of the spacecraft (trail) and a circle representing its instantaneous position (spacecraft).

Velocity Graph:

- Sets the x and y labels and grid settings for the velocity graph.
- Creates a speed graph line.

Animation Function:

- The animation function (`animate`) updates the spacecraft's position, track and velocity graph in each frame.

Creating Animation:

- Creates the animation using the `matplotlib.animation.FuncAnimation` function, determining the number of frames and interval.
- Closes the figure that created the animation.

Outputs:

- ***ani:*** A `matplotlib.animation.FuncAnimation` object representing the generated animation.

General Overview:

The `create_animation` function is a powerful tool designed to visually present the results of a space travel simulation. This function transforms complex simulation data into an understandable and engaging animation. By simultaneously visualizing the spacecraft trajectory, planetary positions, Lagrange points, velocity variations and elapsed time, it makes the results of the simulation easier to understand and analyze. This animation is a useful tool to evaluate the performance of the trained agent and share the results of the simulation.

(3660-3744)

plot_training_metrics Function

The `plot_training_metrics` function is a procedure designed to visualize the metrics recorded during the reinforcement learning training process. This function displays training rewards, losses and discovery rates (epsilon) in separate graphs using the `matplotlib` library.

Parameters:

- ***scores:*** A list containing the total rewards in the training process.
- ***losses:*** A list containing the loss values in the training process.
- ***epsilons:*** A list containing the discovery rates (epsilon values) in the training process.

Functionality:

Visualization Settings:

- Implements a dark theme using the `matplotlib.pyplot` library.
- Creates 3 subgraphs to show the metrics. The first graph shows training rewards, the second graph shows training losses and the third graph shows discovery rates.

Reward Graph:

- Plots the reward values for the first sub-chart.
- Adds title, x and y axis labels to the graph.
- Adds a grid to the graph (transparency set to 0.3).

Loss Graph:

- Plots the loss values for the second subgraph.
- Adds title, x and y axis labels to the chart.
- Adds a grid to the chart (transparency set to 0.3).

Discovery Rate Graph:

- Plots the discovery rate (epsilon) values for the third subgraph.
- Adds title, x and y axis labels to the chart.
- Adds a grid to the chart (transparency set to 0.3).

Arrangement:

- Automatically adjusts the layout of the graphs and edits the graphs for a better view.

Outputs:

- ***fig:*** A `matplotlib.figure.Figure` object representing the generated chart.

General Summary:

The `plot_training_metrics` function is a basic tool for evaluating the performance of reinforcement learning training. By visualizing the metrics recorded during the training process, this function makes it easy to understand how well the learning process is progressing and to evaluate the performance of the model. By aggregating important metrics such as reward, loss and discovery rates, it allows to analyze the performance of the model with an overview. Thus, it helps to make better decisions in the training process and allows to achieve optimal results more quickly.

(3746-3769)

SpaceSimulationGUI Class

The `SpaceSimulationGUI` class is an abstract data type designed to visualize a spacecraft simulation using the Pygame library. This class includes functionality to draw the simulation to the screen, handle user interactions, display metric information, and provide general simulation control.

Data Fields (Attributes):

- ***width:*** An integer representing the width (pixels) of the screen.
- ***height:*** An integer representing the height of the screen (pixels).
- ***screen:*** A `pygame.Surface` object representing the pygame screen object.
- ***BLACK:*** A tuple representing the color black.
- ***WHITE:*** A tuple representing the color white.
- ***YELLOW:*** A tuple representing the color yellow (used for the sun).
- ***RED:*** A tuple representing the color red (used for spacecraft).
- ***BLUE:*** A tuple representing the color blue (used for the target planet).
- ***GREEN:*** A tuple representing the color green (used for other planets).
- ***GRAY:*** A tuple representing the color gray (used for the orbit).
- ***sim_width:*** An integer representing the width (pixels) of the simulation area (80% of the screen).
- ***panel_width:*** An integer representing the width (pixels) of the side panel (20% of the screen).
- ***title_font:*** Pygame font object used for title text.
- ***font:*** Pygame font object used for normal text.
- ***small_font:*** Pygame font object used for small fonts.
- ***paused:*** A boolean indicating whether the simulation is paused.
- ***current_sim_index:*** An integer that stores the index (0-23) of the active simulation.
- ***show_all_sims:*** A boolean indicating whether all simulations are shown.
- ***zoom_level:*** A float representing the zoom level.
- ***pan_offset:*** A list (x, y) representing the pan offset.
- ***clock:*** Pygame clock object that controls the frame rate of the simulation.
- ***fps:*** An integer representing frames per second (FPS).
- ***surface_cache:*** A dictionary that stores the surface cache (for performance optimization).
- ***trail_points:*** A dictionary that stores the spacecraft's trail points for each simulation.
- ***max_trail_length:*** An integer storing the maximum length of the spacecraft tracks.
- ***buttons:*** A dictionary storing the buttons and their positions in the user interface.
- ***metric_history:*** A dictionary that stores historical metric values.

- ***scale:*** A float holding the scale factor used in space simulation.
- ***center_x:*** An integer holding the center x coordinate of the screen.
- ***center_y:*** An integer holding the center y coordinate of the screen.

Initializer Method (`init`):

- Takes the parameters `width` (screen width) and `height` (screen height).
- Initializes the Pygame screen, sets the title, defines colors, sets screen sections, loads fonts, sets simulation controls, initializes trackpoints, creates buttons, creates the metric history, calculates the scale factor and defines the screen center.

Methods

world_to_screen(pos): Converts a position in the simulation world to screen coordinates.

- ***Input:*** `pos` (world coordinates).
- ***Output:*** Screen coordinates (x, y).
- ***Function:*** Converts world coordinates to screen coordinates.

draw_side_panel(metrics): Draws the side panel and prints the metric indicators and control buttons on the screen.

- ***Input:*** `metrics` (dictionary of metric values).
- ***Function:*** Draws metric information and buttons on the screen.

draw_mini_graph(metric_key, x, y): Draws a mini-graph to visualize the metric history.

- ***Input:*** `metric_key` (key of the metric), `x` (x coordinate), `y` (y coordinate).
- ***Function:*** Draws a mini chart for a specific metric.

draw_buttons(): Draws the control buttons.

draw_simulation(env, metrics): Draws the simulation space, displaying the planets, spacecraft and their orbits.

- ***Input:*** `env` (simulation environment), `metrics` (dictionary of metric values).
- ***Function:*** Draws the simulation environment on the screen.

draw_spacecraft(spacecraft, index, alpha=255): Draws the spacecraft, track and status indicators on the screen.

- ***Input:*** `spacecraft` (spacecraft object), `index` (spacecraft index), `alpha` (transparency value).
- ***Function:*** Draws the spacecraft and its track on the screen.

handle_events(): Handles user input (mouse clicks, keyboard keys). Handles pausing the simulation, changing the simulation index, zoom and pan.

- ***Exit:*** Boolean indicating whether the simulation should continue or not.
- ***Function:*** Handles user input.

run_simulation(env, agent): Starts the simulation. Manages the interaction loop, updating the screen and other events.

- ***Input:*** `env` (simulation environment), `agent` (trained agent).
- ***Function:*** Starts and runs the simulation.

update(metrics): Writes metrics information to the screen.

- ***Input:*** `metrics` (dictionary containing metric values).
- ***Function:*** Writes the current metrics to the screen.

Overview:

The `SpaceSimulationGUI` class provides an interactive and user-friendly spacecraft simulation visualization using Pygame. This class helps the user to control the simulation, monitor results and visually understand the behavior of trained agents. It graphically presents the state of the simulation, the movement of spacecraft, the orbits of planets, and metric information. It also supports user interaction. It is a useful platform for testing and debugging the simulation and analyzing the results. The class serves as an important tool for both developers and users, making simulations more understandable and accessible.

(3771-4037)

run_gui_process Function

The `run_gui_process` function is a procedure designed to create a simple graphical user interface (GUI) that runs in a separate process (multiprocessing) using the Pygame library. This function takes metric information from the training process through a shared dictionary and prints them to the screen. Its purpose is to offer a lightweight interface to visualize and monitor the training process.

Parameters:

- ***shared_ data:*** A dictionary that holds the metric information from the training process and is shared with other processes. This dictionary is especially used to track the progress of the training process and to exchange information.

Functionality:

Pygame Initialization:

- Initializes the Pygame library.
- Creates a screen of the specified width and height (800x600).
- Creates a clock object to control the screen.
- Loads a font to write the text on the screen.

Main Loop:

- Starts a loop that will run continuously as long as the running switch in the `shared_data` dictionary is `True`.
- Handles Pygame events inside the loop. In particular, it stops the simulation when the user closes the window (`QUIT`).
- Clears the screen with black on each loop.
- If the shared data contains `current_metrics`, it prints the metric information to the screen. This is done by generating text for each metric and bliting it to the screen at specified locations.
- Updates the screen.
- Limits the running speed of the loop to 30 FPS.

Pygame Shutdown:

- Terminates the loop and closes the Pygame library when the running switch in the `shared_data` dictionary is `False`.

Outputs:

- This function does not return a direct value (`None`). Its function is to provide a continuously updated GUI using the specified data.

General Summary:

The `run_gui_process` function is used to visualize the progress of the training process with a simple GUI. By running in a separate process, this function allows to display real-time metrics without affecting the speed of the main training process. This makes it easy for the user to follow the training process and get feedback on performance. The GUI offers a simple and lightweight interface focused only on the display of training metrics. It is a simple visualization tool that works in a separate process, focused on tracking the educational process.

(4039-4065)

train_with_visualization Function

The `train_with_visualization` function is a procedure that aims to both monitor the process of training a reinforcement learning agent with visualization and record training metrics. This function sets up the `SpaceTravelEnv` environment, `SACAgent`, executes the training cycle, starts the GUI process and displays the results.

Functionality:

Shared Data Dictionary:

- Creates a dictionary called `shared_data` to enable communication between the training process and the GUI. This dictionary contains whether the simulation is running, pause status, current metrics and positions.

Starting the GUI Process:

- Starts the GUI in a separate process using the `run_gui_process` function. Thus, the main training process is not affected by the running of the GUI and they can run in parallel.

Environment and Agent Setup:

- Creates a "Neptune" targeted simulation environment using the `SpaceTravelEnv` class.
- Creates the SAC agent (`SACAgent`) in accordance with the state and action dimensions of the environment.

Training Cycle:

- Starts a loop for the total number of episodes (`NUM_EPISODES`).
- At the beginning of each episode, it resets the environment.
- Initiates a loop up to the maximum number of steps (`MAX_STEPS`) within each episode.
- Receives the action from the Agent.
- Takes a step in the environment and checks the reward, the new state and the end of the episode.

- Adds the experience to memory.
- Trains the agent if the experiences in memory have reached `batch_size`.
- Updates the current state with the new state.
- Updates the current metrics and positions in the `shared_data` dictionary to be displayed in the GUI.
- Stop the inner loop when the end of the chapter is reached.
- Saves the model's weights every 10 episodes.
- Prints the total reward and epsilon value at the end of each episode.

Error Management:

- Captures errors that occur during the training loop and prints them to the screen.

Result Management:

- Terminates the GUI after the training is finished.

Parameters:

- This function does not take any parameters.

Outputs:

- This function does not return a value (`None`). Its function is to visually track the training process and save the trained agent model to disk.

General Overview:

The `train_with_visualization` function aims to make the reinforcement learning training process both trackable and visually understandable. By separating the GUI and the main training process, this function allows both processes to run in parallel. While following the training process, information about the process can be obtained through the GUI at the same time. This approach facilitates the debugging process and helps us better understand the model's performance, especially when training complex reinforcement learning models.

(4067-4128)

draw_ simulation Function

The `draw_simulation` function is a procedure designed to draw a simulation of space travel on the screen using the Pygame library. This function visualizes the planets in the simulation, the spacecraft and an overview of the solar system in a simple way.

Parameters:

- ***env:*** An object of the `SpaceTravelEnv` class representing the simulation environment.
- ***screen:*** A `pygame.Surface` object representing the pygame screen object.

Functionality:

Color Definition:

- Defines the colors to be used (white, black, red, blue, yellow).

Screen Dimensions:

- Defines the width and height of the screen (`SCREEN_WIDTH`, `SCREEN_HEIGHT`).

Scaling Factor:

- Defines a scale factor to be used to properly display simulation data on the screen (`SCALE`).

Clear Screen:

- Fills the screen with black color.

Draw Sun:

- Represents the Sun by drawing a yellow circle in the center of the screen.

Drawing Planets:

- For each planet in the simulation environment:
  - Scales the planet's position relative to the screen and converts it to Pygame coordinates.
  - Represent the planets by drawing a red circle if it is the target planet and a blue circle for the other planets.

Draw Spacecraft:

- For each spacecraft in the simulation environment:
  - Scales the spacecraft position relative to the screen and converts it to Pygame coordinates.
  - Draws a white circle to represent the spacecraft.

Update Screen:

- Updates the screen after completing all drawings.

Outputs:

- This function does not return a value (`None`). The function's task is to visually represent the simulation on the screen.

General Overview:

The `draw_simulation` function is a tool for visualizing a space travel simulation in a simple way. This function draws the planets, the spacecraft and the sun on the screen using basic geometric shapes and colors. Its purpose is to make it easier for the user to follow the environment by visually presenting the basic elements of the simulation. The `draw_simulation` function is suitable for simple simulation follow-ups without the need for a graphical interface or for quick test visualizations.

(4130-4168)

main Function

The `main` function is the main execution procedure designed to manage the spacecraft simulation and the reinforcement learning training process. This function sets up the simulation environment, trains the agent, saves training metrics, runs a simulation and displays the training results and simulation on the screen.

Functionality:

Defining Constants:

- Defines the target planet (`DESTINATION_PLANET`) as "Neptune".
- Defines the population size (`POPULATION_SIZE`) as 48.
- Defines the number of training episodes (`NUM_EPISODES`) as 50000.
- Creates an empty list for the total reward record (`total_rewards`).

Simulation Environment and Agent Setup:

- Creates a simulation environment using the `SpaceTravelEnv` class with the specified target planet and population size.
- Creates a genetic SAC agent with the state and action dimensions of the environment using the `LongTermGeneticSACAgent` class.

Training Cycle:

- Starts a training loop and resets the simulation environment at the beginning of each episode.
- Select actions using the agent's `act_batch` function.
- Steps through the environment and adds experiences to memory using the agent's `remember_batch` function.
- If the memory is full, it trains the agent.
- Evolves the population
- Saves the average of all partition rewards in the `total_rewards` list.
- Calls the `train_agent` function to get the trained agent and metrics.
- Saves the model of the trained agent to disk.

Test Simulation:

- After training is complete, runs a test simulation using the trained agent and the simulation environment.

Visualization:

- Creates an object of the `SpaceSimulationGUI` class to visualize the simulation results and starts the simulation.

Metric Visualization:

- Graphs metrics recorded during training, such as average rewards, losses, success rates, minimum distances and fuel consumption.
- Saves the output of the plotted graphs as (training_metrics.png).

Error Management:

- Captures errors that occur in all processes and prints them to the screen.
- Clears CUDA memory in case of any error.

Result Management:

- After completing all processes, PyTorch clears the CUDA memory.
- It closes all matplotlib plots and terminates the program.

Parameters:

- This function takes no parameters.

Outputs:

- This function does not return any value (`None`). Its function is to manage the training and testing processes and present the results visually.

General Overview:

The `main` function is the main control center for running the spacecraft simulation, training the reinforcement learning agent, visualizing and saving the results. This function automates the entire process and allows it to complete the simulation and training without the need for user interaction. It saves the data generated during the training process, visualizes the final simulation and presents the training performance and results to the user. This function represents a complete execution flow of a complex simulation and reinforcement learning application.

(4170-4255)

Finally, the code is still under development and I cannot make a statement about the outputs because the project is not finished yet, changes and results will be shared as a separate report at the end of the project.

The End

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from collections import deque
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from matplotlib.patches import Circle
from concurrent.futures import ThreadPoolExecutor
from typing import List, Dict
import pygame
from typing import Dict, List , Any
import multiprocessing
import torch.cuda.amp
import traceback
import os
import math
import copy
import time
from datetime import timedelta

print("GPU Durumu:")
print(f"PyTorch: {'GPU' if torch.cuda.is_available() else 'CPU'}")
if torch.cuda.is_available():
    print(f"Kullanılan GPU: {torch.cuda.get_device_name(0)}")
    print(f"GPU Sayısı: {torch.cuda.device_count()}")
    print(f"Mevcut GPU İndeksi: {torch.cuda.current_device()}")
else:
    print("GPU bulunamadı, CPU kullanılacak")

# Görselleştirme modu kontrolü
VISUALIZATION_MODE = False  # True: Test/Geliştirme modu, False: Eğitim modu

# Sabitler
MAX_WORKERS = 32 if not VISUALIZATION_MODE else 1  # Eğitim modunda 24, görselleştirme modunda 1 işçi
THREAD_POOL = ThreadPoolExecutor(max_workers=MAX_WORKERS)

# CUDA yapılandırması ve optimizasyonları
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'  # CUDA hata ayıklama için
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'max_split_size_mb:512'  # Bellek parçalanmasını önle
device = torch.device("cuda")
torch.backends.cudnn.benchmark = True
torch.backends.cuda.matmul.allow_tf32 = True  # TF32'yi etkinleştir
torch.backends.cudnn.allow_tf32 = True        # cuDNN TF32'yi etkinleştir
torch.cuda.empty_cache()
```

```python

# Fiziksel sabitler
G = 6.67430e-11          # Gravitasyon sabiti (m³/kg/s²)
SUN_MASS = 1.989e30      # Güneş kütlesi (kg)
AU = 1.496e11            # Astronomik birim (m)
TOTAL_TIME = 15 * 365 * 24 * 3600  # Toplam simülasyon süresi (15 yıl)
TIME_STEP = 86400        # Zaman adımı (1 gün)
NUM_STEPS = int(TOTAL_TIME / TIME_STEP)
MAX_THRUST = 1e3         # Maksimum itki kuvveti (N)
ISP = 4000               # Özgül impuls (s)
g0 = 9.81                # Yerçekimi ivmesi (m/s²)
SOLAR_SYSTEM_BOUNDARY = 60 * AU  # 60 AU'luk görüş alanı
SPEED_OF_LIGHT = 3e8     # Işık hızı (m/s)
MAX_SPACECRAFT_SPEED = 150000  # Maksimum uzay aracı hızı (m/s)
OPTIMAL_VELOCITY = 40000  # Optimal hız (40 km/s)
MAX_ACCELERATION = 0.1   # Maksimum ivme (m/s²)
MIN_SAFE_DISTANCE_SUN = 696340000 * 2  # Güneş'e minimum güvenli mesafe (2 güneş yarıçapı)
SUCCESS_DISTANCE = 5e6   # Başarılı varış mesafesi (m)
MAX_EPISODE_TIME = 15* 365 * 24 * 3600  # Maksimum bölüm süresi (15 yıl)

# Acil durum olasılıkları
MOTOR_FAILURE_PROBABILITY = 0.00001      # Motor arıza olasılığı
FUEL_LEAK_PROBABILITY = 0.00002          # Yakıt sızıntısı olasılığı
MICROASTEROID_COLLISION_PROBABILITY = 0.0001  # Mikroasteroit çarpışma olasılığı
NUM_EPISODES = 50000     # Toplam eğitim bölümü sayısı
MAX_STEPS = 50000        # Bir bölümdeki maksimum adım sayısı

```

```python
# Görelilik fiziği hesaplamalarını yapan sınıf
class RelativisticSpacePhysics:
    def __init__(self, device='cuda' if torch.cuda.is_available() else 'cpu'):
        self.device = device
        self.c = 299792458.0  # Işık hızı (m/s)
        self.G = 6.67430e-11  # Gravitasyon sabiti (m³/kg/s²)
        self.sun_mass = 1.989e30  # Güneş kütlesi (kg)

        # Schwarzschild yarıçapı - Kara deliğin olay ufku yarıçapı
        self.rs = 2 * self.G * self.sun_mass / (self.c ** 2)

    def schwarzschild_metric(self, r):
        """Schwarzschild metriğini hesapla - Kara delik etrafındaki uzay-zaman eğriliğini tanımlar"""
        try:
            # Metrik bileşenleri
            g_tt = -(1 - self.rs / r)  # Zamansal bileşen - Zaman genişlemesi
            g_rr = 1 / (1 - self.rs / r)  # Radyal bileşen - Uzaysal eğrilik
            g_thth = r ** 2  # Açısal bileşen (theta) - Küresel koordinatlar
            g_phph = r ** 2 * torch.sin(torch.tensor(torch.pi/2, device=self.device)) ** 2  # Açısal bileşen (phi)

            return g_tt, g_rr, g_thth, g_phph
        except Exception as e:
            print(f"Metrik hesaplama hatası: {e}")
            return None

    def calculate_relativistic_force(self, position, velocity, mass):
        """Görelilik etkilerini içeren kuvveti hesapla - Özel ve genel görelilik etkilerini birleştirir"""
        try:
            r = torch.norm(position)  # Güneş'e olan uzaklık
            v = torch.norm(velocity)  # Hız büyüklüğü

            # Klasik Newton kuvveti
            F_classical = -self.G * self.sun_mass * mass * position / (r**3)

            # Görelilik düzeltmeleri
            gamma = 1.0 / torch.sqrt(1.0 - (v/self.c)**2)  # Lorentz faktörü

            # Schwarzschild metriği - Uzay-zaman eğriliği
            metric_result = self.schwarzschild_metric(r)
            if metric_result is None:
                return F_classical # Hata durumunda klasik kuvveti döndür

            g_tt, g_rr, _, _ = metric_result

            # Görelilik düzeltmesi faktörleri
            perihelion_shift = 1 + 3 * self.G * self.sun_mass / (r * self.c**2)  # Merkür'ün perihel kayması
            time_dilation = torch.sqrt(-g_tt)  # Zaman genişlemesi
```

```
            spatial_curvature = torch.sqrt(g_rr)  # Uzaysal eğrilik

            # Toplam görelilik etkisi
            relativistic_correction = gamma * perihelion_shift * time_dilation * spatial_curvature

            # Düzeltilmiş kuvvet
            F_relativistic = F_classical * relativistic_correction

            # Kuvvet sınırlaması - Işık hızı limiti
            max_force = mass * self.c**2 / r  # Maksimum izin verilen kuvvet
            F_magnitude = torch.norm(F_relativistic)
            if F_magnitude > max_force:
                F_relativistic = F_relativistic * (max_force / F_magnitude)

            return F_relativistic

        except Exception as e:
            print(f"Görelilik kuvveti hesaplama hatası: {e}")
            return -self.G * self.sun_mass * mass * position / (r**3)  # Hata durumunda klasik kuvveti döndür

    def update_planet_motion(self, planet, dt):
        """Gezegen hareketini Runge-Kutta-Fehlberg (RKF45) metodu ile güncelle"""
        try:
            # Görelilik etkili kuvvet
            force = self.calculate_relativistic_force(planet.position, planet.velocity, planet.mass)

            # Lorentz faktörü - Özel görelilik düzeltmesi
            v = torch.norm(planet.velocity)
            gamma = 1 / torch.sqrt(1 - (v/self.c)**2)

            # Görelilik etkili ivme (F = γma)
            self.acceleration = force / (gamma * planet.mass)

            # RKF45 katsayıları
            a2, a3, a4, a5, a6 = 1/4, 3/8, 12/13, 1, 1/2
            b21 = 1/4
            b31, b32 = 3/32, 9/32
            b41, b42, b43 = 1932/2197, -7200/2197, 7296/2197
            b51, b52, b53, b54 = 439/216, -8, 3680/513, -845/4104
            b61, b62, b63, b64, b65 = -8/27, 2, -3544/2565, 1859/4104, -11/40
```

```
        # RKF45 adımları
        def f(t, y):
            pos, vel = y[:2], y[2:]
            acc = self.calculate_relativistic_force(pos, vel, planet.mass) / (gamma * planet.mass)
            return torch.cat([vel, acc])

        y = torch.cat([planet.position, planet.velocity])
        t = torch.tensor(0.0, device=self.device)

        # k1 hesapla
        k1 = dt * f(t, y)

        # k2 hesapla
        k2 = dt * f(t + a2*dt, y + b21*k1)

        # k3 hesapla
        k3 = dt * f(t + a3*dt, y + b31*k1 + b32*k2)

        # k4 hesapla
        k4 = dt * f(t + a4*dt, y + b41*k1 + b42*k2 + b43*k3)

        # k5 hesapla
        k5 = dt * f(t + a5*dt, y + b51*k1 + b52*k2 + b53*k3 + b54*k4)

        # k6 hesapla
        k6 = dt * f(t + a6*dt, y + b61*k1 + b62*k2 + b63*k3 + b64*k4 + b65*k5)

        # 4. derece çözüm
        y4 = y + (25/216)*k1 + (1408/2565)*k3 + (2197/4104)*k4 - (1/5)*k5

        # 5. derece çözüm
        y5 = y + (16/135)*k1 + (6656/12825)*k3 + (28561/56430)*k4 - (9/50)*k5 + (2/55)*k6

        # Hata tahmini
        error = torch.norm(y5 - y4)

        # Optimal adım boyutu hesaplama
        tolerance = 1e-6
        optimal_dt = dt * (tolerance / (2 * error))**0.2

        # Eğer hata çok büyükse adım boyutunu küçült
        if error > tolerance:
            return self.update_planet_motion(planet, optimal_dt)
```

```
206
207             # Yeni pozisyon ve hızı ayarla
208             new_position = y5[:2]
209             new_velocity = y5[2:]
210
211             # Hız sınırlaması - Işık hızını geçemez
212             v_new = torch.norm(new_velocity)
213             if v_new >= self.c:
214                 new_velocity = new_velocity * (0.99 * self.c / v_new)
215
216             return new_position, new_velocity
217
218         except Exception as e:
219             print(f"Hareket güncelleme hatası: {e}")
220             return planet.position, planet.velocity
221
222     def calculate_light_bending(self, impact_parameter):
223         """Işık sapmasını hesapla - Einstein'ın genel görelilik öngörüsü"""
224         try:
225             deflection_angle = 4 * self.G * self.sun_mass / (impact_parameter * self.c**2)
226             return deflection_angle
227         except Exception as e:
228             print(f"Işık sapması hesaplama hatası: {e}")
229             return 0.0
230
231     def gravitational_redshift(self, r):
232         """Gravitasyonel kırmızıya kaymayı hesapla - Güçlü gravitasyon alanlarında ışığın dalga boyundaki değişim"""
233         try:
234             redshift = 1 / torch.sqrt(1 - self.rs/r) - 1
235             return redshift
236         except Exception as e:
237             print(f"Kırmızıya kayma hesaplama hatası: {e}")
238             return 0.0
```

```python
# Genetik algoritma ve Transformer tabanlı optimizasyon sınıfı
class GeneticTransformerOptimizer:
    def __init__(self,
                 population_size=48,      # Popülasyon büyüklüğü
                 num_generations=1000,     # Nesil sayısı
                 mutation_rate=0.1,        # Mutasyon oranı
                 d_model=64,               # Transformer model boyutu
                 n_head=4,                 # Attention head sayısı
                 n_layers=4,               # Transformer katman sayısı
                 d_ff=512,                 # Feed-forward boyutu
                 device='cuda'):

        self.device = device
        self.population_size = population_size
        self.num_generations = num_generations
        self.mutation_rate = mutation_rate

        # Transformer model parametreleri
        self.d_model = d_model
        self.n_head = n_head
        self.n_layers = n_layers
        self.d_ff = d_ff

        # Model oluşturma - SAC (Soft Actor-Critic) ağı
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(
                d_model=d_model,
                nhead=n_head,
                dim_feedforward=d_ff,
                dropout=0.1,
                batch_first=True
            ).to(device),
            num_layers=n_layers
        ).to(device)

        # Optimizer ve gradient scaler
        self.optimizer = optim.Adam(self.transformer.parameters(), lr=1e-4)
        self.scaler =torch.amp.GradScaler('cuda')

    def create_population(self):
        """Başlangıç popülasyonunu oluştur - 360 derecelik yörünge noktaları"""
        return torch.randn(self.population_size, 360, 2, device=self.device)

    def evaluate_population_batch(self, population_batch, env):
        """GPU üzerinde toplu değerlendirme - Paralel simülasyon"""
```

```python
    def evaluate_population_batch(self, population_batch, env):
        """GPU üzerinde toplu değerlendirme - Paralel simülasyon"""
        batch_size = len(population_batch)
        rewards = torch.zeros(batch_size, device=self.device)

        states = torch.stack([env.reset() for _ in range(batch_size)])
        dones = torch.zeros(batch_size, dtype=torch.bool, device=self.device)

        for _ in range(MAX_STEPS):
            with torch.no_grad():
                # Transformer girişi için boyut düzeltmesi
                states_input = states.unsqueeze(1).float()  # [batch_size, 1, state_dim]
                actions = self.transformer(states_input).squeeze(1)  # [batch_size, action_dim]

            # Çevreye gönderilecek aksiyonların tipini kontrol et
            actions = actions.detach()  # Gradyanları kopar

            # Batch işlemi için tip kontrolü
            next_states, step_rewards, step_dones, _ = env.step_batch(actions)

            # Ödül ve done maskelerinin tiplerini kontrol et
            step_rewards = step_rewards.to(self.device)
            step_dones = step_dones.to(self.device)

            # Ödül ve done güncellemeleri
            rewards += step_rewards * (~dones)
            dones |= step_dones

            if dones.all():
                break

            states = next_states.clone()  # Güvenli kopya

        return rewards

    def crossover(self, parent1, parent2):
        """İki yörüngeyi çaprazla - Tek noktalı çaprazlama"""
        crossover_point = torch.randint(0, parent1.size(0), (1,))
        child = torch.cat([
            parent1[:crossover_point],
            parent2[crossover_point:]
        ])
        return child
```

```python
    def mutate(self, trajectory):
        """Yörüngede mutasyon uygula - Gaussian gürültü ekleme"""
        mutation_mask = torch.rand_like(trajectory) < self.mutation_rate
        trajectory += mutation_mask * torch.randn_like(trajectory) * 0.1
        return trajectory

    def train(self, env):
        """Ana eğitim döngüsü - Genetik algoritma ve DQN kombinasyonu"""
        try:
            best_fitness = float('-inf')
            best_trajectory = None

            for generation in range(self.num_generations):
                # Yeni popülasyon oluştur
                population = self.create_population()
                fitness_scores = self.evaluate_population_batch(population, env)

                # En iyileri seç
                elite_indices = torch.topk(fitness_scores, k=self.population_size//4).indices
                elite = population[elite_indices]

                # Yeni nesil oluştur
                new_population = [elite]
                while len(new_population) < self.population_size:
                    parent1, parent2 = random.sample(elite.tolist(), 2)
                    child = self.crossover(parent1, parent2)
                    child = self.mutate(child)
                    new_population.append(child)

                population = torch.stack(new_population)

                # En iyi sonucu güncelle
                max_fitness, max_idx = torch.max(fitness_scores, dim=0)
                if max_fitness > best_fitness:
                    best_fitness = max_fitness
                    best_trajectory = population[max_idx].clone()

                if generation % 10 == 0:
                    print(f"Nesil {generation}: En iyi fitness = {best_fitness:.2f}")

        except Exception as e:
            print(f"Eğitim hatası: {e}")
            traceback.print_exc()

        return best_trajectory, best_fitness
```

```python

    def physics_loss(self, state, next_state):
        """PINN fizik kaybını hesapla - Fizik yasalarına uygunluk kontrolü"""
        # Fizik kodlaması
        physics_encoding = self.current_model['physics_encoder'](state)
        next_physics_encoding = self.current_model['physics_encoder'](next_state)

        # Korunum yasaları kaybı
        conservation_pred = self.current_model['conservation_layer'](physics_encoding)
        next_conservation_pred = self.current_model['conservation_layer'](next_physics_encoding)

        # Enerji korunumu
        energy_loss = torch.mean((conservation_pred[:, 0] - next_conservation_pred[:, 0]) ** 2)

        # Momentum korunumu
        momentum_loss = torch.mean((conservation_pred[:, 1:] - next_conservation_pred[:, 1:]) ** 2)

        # Görelilik düzeltmeleri
        rel_pred = self.current_model['relativistic_layer'](physics_encoding)
        velocity = torch.norm(state[:, 3:5], dim=1)
        gamma = 1.0 / torch.sqrt(1.0 - (velocity/3e8)**2)
        rel_loss = torch.mean((rel_pred - gamma) ** 2)

        return energy_loss + momentum_loss + rel_loss

    @torch.amp.autocast('cuda')
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """İleri yayılım - Batch işleme ve bellek optimizasyonu"""
        batch_size = x.size(0)
        max_batch = 32  # Maksimum batch boyutu

        if batch_size <= max_batch:
            return self._forward_single(x)

        outputs = []
        for i in range(0, batch_size, max_batch):
            batch = x[i:i + max_batch]
            with torch.no_grad():
                output = self._forward_single(batch)
            outputs.append(output)

        return torch.cat(outputs, dim=0)

```

```python

    def _forward_single(self, x: torch.Tensor) -> torch.Tensor:
        """
        Tek batch için ileri yayılım işlemi
        """
        try:
            # Giriş boyutu düzeltme ve veri tipi dönüşümü
            if len(x.shape) == 2:
                x = x.unsqueeze(1)  # [batch_size, 1, features]
            x = x.to(dtype=torch.float16)  # FP16'ya dönüştür

            # Giriş projeksiyonu
            x = self.current_model['input_proj'](x)  # [batch_size, seq_len, d_model]

            # Positional encoding ekle
            seq_len = x.size(1)
            pos_enc = self.current_model['pos_encoder'](
                torch.zeros(x.size(0), seq_len, dtype=torch.long, device=x.device))
            pos_enc = x + pos_enc

            # Transformer katmanı
            # Attention mask oluştur (False = herhangi bir padding yok)
            mask = torch.zeros((x.size(0), x.size(1)), device=x.device).bool()
            x = self.current_model['transformer'](x, src_key_padding_mask=mask)  # [batch_size, seq_len, d_model]

            # Global pooling - sequence boyutunu ortadan kaldır
            x = x.mean(dim=1)  # [batch_size, d_model]

            # Physics-Informed katmanlar
            with torch.amp.autocast('cuda'):
                # Fizik kodlaması
                physics_features = self.current_model['physics_encoder'](x)

                # Korunum yasaları
                conservation_pred = self.current_model['conservation_layer'](physics_features)
                conservation_factor = torch.sigmoid(conservation_pred[:, 0]).unsqueeze(1)

                # Görelilik etkileri
                relativistic_pred = self.current_model['relativistic_layer'](physics_features)
                gamma_factor = torch.sigmoid(relativistic_pred[:, 0]).unsqueeze(1)

                # SAC mimarisi
                actor_output = self.current_model['actor'](x)
                mean, log_std = torch.chunk(actor_output, 2, dim=-1)
                log_std = torch.clamp(log_std, min=-20, max=2)
```

```
454
455                 # SAC mimarisi
456                 actor_output = self.current_model['actor'](x)
457                 mean, log_std = torch.chunk(actor_output, 2, dim=-1)
458                 log_std = torch.clamp(log_std, min=-20, max=2)
459                 std = log_std.exp()
460
461                 # Gaussian dağılımdan örnekleme
462                 normal = torch.distributions.Normal(mean, std)
463                 action = normal.rsample()
464                 action = torch.tanh(action)
465
466                 # Log olasılığını hesapla
467                 log_prob = normal.log_prob(action)
468                 log_prob = log_prob - torch.log(1 - action.pow(2) + 1e-6)
469                 log_prob = log_prob.sum(1, keepdim=True)
470
471             # Fizik tabanlı düzeltmeler
472             physics_correction = (1.0 + 0.1 * conservation_factor) * (1.0 + 0.05 * gamma_factor)
473             q_values = q_values * physics_correction
474
475             # Çıktı normalizasyonu ve sınırlama
476             q_values = torch.clamp(q_values, min=-100.0, max=100.0)
477
478             # Bellek optimizasyonu
479             del physics_features, conservation_pred, relativistic_pred
480             del value_stream, advantage_stream, advantage_mean
481             torch.cuda.empty_cache()
482
483             return q_values,action, log_prob
484
485         except Exception as e:
486             print(f"Forward pass hatası: {str(e)}")
487             print("Hata detayları:")
488             print(f"Giriş boyutu: {x.shape}")
489             print(f"Giriş tipi: {x.dtype}")
490             print(f"Cihaz: {x.device}")
491             traceback.print_exc()
492             raise e
```

```
484
485         except Exception as e:
486             print(f"Forward pass hatası: {str(e)}")
487             print("Hata detayları:")
488             print(f"Giriş boyutu: {x.shape}")
489             print(f"Giriş tipi: {x.dtype}")
490             print(f"Cihaz: {x.device}")
491             traceback.print_exc()
492             raise e
493
494         finally:
495             # Ek bellek temizliği
496             if hasattr(self, '_temp_storage'):
497                 del self._temp_storage
498             torch.cuda.empty_cache()
499
```

```python
class LagrangePoints:
    def __init__(self, planet_mass, planet_orbit_radius):
        self.mu = planet_mass / (SUN_MASS + planet_mass)
        self.orbit_radius = planet_orbit_radius

        # L1 point
        self.l1_distance = self.orbit_radius * (1 - (self.mu/3)**(1/3))
        self.l1 = torch.tensor([self.l1_distance, 0.0], device=device)
        self.l2_distance = self.orbit_radius * (1 + (self.mu/3)**(1/3))
        self.l2 = torch.tensor([self.l2_distance, 0.0], device=device)
        self.influence_radius = 0.15 * self.orbit_radius

# Planetary data
PLANETARY_DATA = {
    'Mercury': {'mass': 3.3011e23, 'orbit_radius': 0.387 * AU, 'lagrange_points': None},
    'Venus': {'mass': 4.8675e24, 'orbit_radius': 0.723 * AU, 'lagrange_points': None},
    'Earth': {'mass': 5.97237e24, 'orbit_radius': 1.0 * AU, 'lagrange_points': None},
    'Mars': {'mass': 6.4171e23, 'orbit_radius': 1.52 * AU, 'lagrange_points': None},
    'Jupiter': {'mass': 1.8982e27, 'orbit_radius': 5.20 * AU, 'lagrange_points': None},
    'Saturn': {'mass': 5.6834e26, 'orbit_radius': 9.54 * AU, 'lagrange_points': None},
    'Uranus': {'mass': 8.6810e25, 'orbit_radius': 19.19 * AU, 'lagrange_points': None},
    'Neptune': {'mass': 1.02413e26, 'orbit_radius': 30.07 * AU, 'lagrange_points': None}
}

```

```python
class Planet:
    def __init__(self, name: str, mass: float, orbit_radius: float):
        self.name = name
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.mass = torch.tensor(mass, device=self.device)
        self.orbit_radius = torch.tensor(orbit_radius, device=self.device)
        self.position = torch.zeros(2, device=self.device)
        self.velocity = torch.zeros(2, device=self.device)
        self.radius = torch.tensor(0.0, device=self.device)
        self.positions = []
        self.l1_positions = []
        self.l2_positions = []

        # Lagrange noktalarını hesapla
        self.lagrange = LagrangePoints(mass, orbit_radius)

    def gravitational_acceleration(self, r: torch.Tensor) -> torch.Tensor:
        r_mag = torch.norm(r)
        if r_mag < 1e-10:
            return torch.zeros_like(r)
        return -G * SUN_MASS * r / (r_mag**3)

    def update_position_rk45(self, dt_and_state=None):
        def derivatives(t_and_state: tuple) -> torch.Tensor:
            t, state = t_and_state
            r, v = state[:2], state[2:]
            a = self.gravitational_acceleration(r)
            return torch.cat([v, a])

        # Eğer dt_and_state tuple olarak geldiyse, ayrıştır
        if isinstance(dt_and_state, tuple):
            dt, (pos, vel) = dt_and_state
            y = torch.cat([pos, vel])
        else:
            dt = dt_and_state  # Normal dt değeri
            y = torch.cat([self.position, self.velocity])

        # Zaman noktalarını önceden hesapla
        t0 = torch.zeros(1, device=self.device)
        t1 = torch.tensor([dt/4], device=self.device)
        t2 = torch.tensor([3*dt/8], device=self.device)
        t3 = torch.tensor([12*dt/13], device=self.device)
        t4 = torch.tensor([dt], device=self.device)
        t5 = torch.tensor([dt/2], device=self.device)

```

```
        # RK45 katsayıları
        k1 = dt * derivatives((t0, y))
        k2 = dt * derivatives((t1, y + k1/4))
        k3 = dt * derivatives((t2, y + 3*k1/32 + 9*k2/32))
        k4 = dt * derivatives((t3, y + 1932*k1/2197 - 7200*k2/2197 + 7296*k3/2197))
        k5 = dt * derivatives((t4, y + 439*k1/216 - 8*k2 + 3680*k3/513 - 845*k4/4104))
        k6 = dt * derivatives((t5, y - 8*k1/27 + 2*k2 - 3544*k3/2565 + 1859*k4/4104 - 11*k5/40))


        y4 = y + 25*k1/216 + 1408*k3/2565 + 2197*k4/4104 - k5/5
        y5 = y + 16*k1/135 + 6656*k3/12825 + 28561*k4/56430 - 9*k5/50 + 2*k6/55

        # Hata tahmini
        error = torch.norm(y5 - y4)

        # Optimal adım boyutu hesaplama
        tolerance = 1e-6
        optimal_dt = dt * (tolerance / (2 * error))**0.2

        # Eğer hata çok büyükse adım boyutunu küçült ve tekrar dene
        if error > tolerance:
            return self.update_position_rk45((optimal_dt, (y5[:2], y5[2:])))

        # Pozisyon ve hızı güncelle
        self.position = y5[:2]
        self.velocity = y5[2:]

        # Lagrange noktalarını güncelle
        angle = torch.atan2(self.position[1], self.position[0])
        rotation_matrix = torch.tensor([
            [torch.cos(angle), -torch.sin(angle)],
            [torch.sin(angle), torch.cos(angle)]
        ], device=self.device)

        l1_pos = torch.matmul(rotation_matrix, self.lagrange.l1)
        l2_pos = torch.matmul(rotation_matrix, self.lagrange.l2)

        self.l1_positions.append(l1_pos)
        self.l2_positions.append(l2_pos)
        self.positions.append(self.position.clone())
```

```python

    def reset(self):
        """Gezegeni başlangıç konumuna döndür"""
        if self.name in self.orbital_parameters:
            params = self.orbital_parameters[self.name]

            # Başlangıç açısını tensor olarak bir kez oluştur
            initial_angle = params['initial_angle'].to(self.device)

            # Trigonometrik hesaplamaları bir kez yap
            cos_angle = torch.cos(initial_angle)
            sin_angle = torch.sin(initial_angle)

            # Yarıçapı hesapla
            r = params['semi_major_axis'] * (1 - params['eccentricity']**2) / \
                (1 + params['eccentricity'] * cos_angle)

            # Pozisyon vektörünü oluştur
            self.position = torch.stack([
                r * cos_angle,
                r * sin_angle
            ]).to(self.device)

            # Hız büyüklüğünü hesapla
            velocity_magnitude = torch.sqrt(G * SUN_MASS *
                                            (2/r - 1/params['semi_major_axis']))

            # Hız vektörünü oluştur
            self.velocity = torch.stack([
                -velocity_magnitude * sin_angle,
                velocity_magnitude * cos_angle
            ]).to(self.device)
```

```python

class Spacecraft:
    def __init__(self, mass: float):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        # Temel özellikler
        self.mass = torch.tensor(mass, device=device, dtype=torch.float32)
        self.dry_mass = torch.tensor(mass, device=device, dtype=torch.float32)
        self.velocity = torch.zeros(2, device=device, dtype=torch.float32)
        self.acceleration = torch.zeros(2, device=device, dtype=torch.float32)
        # Başlangıç hızını ayrıca ayarlayın
        self.velocity[1] = 1500.0  # Y ekseni hızını 1500 olarak ayarla
        # Durum vektörleri - pozisyon, hız, ivme
        self.position = torch.zeros(2, device=self.device)
        self.velocity = torch.zeros(2, device=self.device)
        self.acceleration = torch.zeros(2, device=self.device)

        # Geçmiş kayıtları
        self.thrust_history: List[float] = []
        self.speed_history: List[float] = []
        self.acceleration_history: List[float] = []
        self.position_history: List[torch.Tensor] = []
        self.fuel_consumption_history: List[float] = []
        self.orbit_parameters_history: List[Dict[str, float]] = []

        # Motor sistemleri
        self.main_engine_thrust = torch.tensor(50000.0, device=device, dtype=torch.float32)  # Newton
        self.backup_engines = 3
        self.current_engine = 0
        self.engine_efficiency = torch.tensor(0.95, device=device, dtype=torch.float32)
        self.engine_temperature = torch.tensor(300.0, device=device, dtype=torch.float32)  # Kelvin
        self.min_engine_temp = torch.tensor(200.0, device=device, dtype=torch.float32)  # Kelvin
        self.max_engine_temp = torch.tensor(1500.0, device=device, dtype=torch.float32)  # Kelvin
        self.optimal_temp_range = (800.0, 1200.0)  # Optimal çalışma sıcaklığı aralığı
        self.heating_rate = torch.tensor(2.0, device=device, dtype=torch.float32)  # Kelvin/s
        self.cooling_rate = torch.tensor(1.0, device=device, dtype=torch.float32)  # Kelvin/s

        # Yakıt sistemi
        self.fuel_consumption_rate = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.has_fuel_leak = False
        self.fuel_leak_rate = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.fuel_system_health = torch.tensor(1.0, device=device, dtype=torch.float32)
```

```python
        # Yapısal bütünlük
        self.hull_integrity = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.hull_damage = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.critical_damage_threshold = torch.tensor(0.8, device=device, dtype=torch.float32)

        # Onarım sistemleri
        self.repair_in_progress = False
        self.repair_time_remaining = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.repair_efficiency = torch.tensor(0.8, device=device, dtype=torch.float32)
        self.auto_repair_systems = True

        # Navigasyon sistemleri
        self.navigation_accuracy = torch.tensor(0.99, device=device, dtype=torch.float32)
        self.guidance_system_health = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.trajectory_error = torch.tensor(0.0, device=device, dtype=torch.float32)

        # Güç sistemleri
        self.power_level = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.solar_panel_efficiency = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.battery_charge = torch.tensor(1.0, device=device, dtype=torch.float32)

        # Isı yönetimi
        self.thermal_load = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.cooling_system_efficiency = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.radiator_performance = torch.tensor(1.0, device=device, dtype=torch.float32)

        # Hayat destek sistemleri
        self.life_support_health = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.oxygen_level = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.pressure_integrity = torch.tensor(1.0, device=device, dtype=torch.float32)

        # İletişim sistemleri
        self.communication_strength = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.signal_quality = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.data_transmission_rate = torch.tensor(1.0, device=device, dtype=torch.float32)

        # Performans metrikleri
        self.delta_v_remaining = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.mission_time = torch.tensor(0.0, device=device, dtype=torch.float32)
        self.total_distance_traveled = torch.tensor(0.0, device=device, dtype=torch.float32)

```

```python
        # Sistem durumu izleme
        self.system_health = {
            'propulsion': torch.tensor(1.0, device=device, dtype=torch.float32),
            'navigation': torch.tensor(1.0, device=device, dtype=torch.float32),
            'power': torch.tensor(1.0, device=device, dtype=torch.float32),
            'thermal': torch.tensor(1.0, device=device, dtype=torch.float32),
            'life_support': torch.tensor(1.0, device=device, dtype=torch.float32),
            'communication': torch.tensor(1.0, device=device, dtype=torch.float32),
            'structural': torch.tensor(1.0, device=device, dtype=torch.float32)
        }

        # Acil durum sistemleri
        self.emergency_systems_active = False
        self.emergency_power_reserve = torch.tensor(1.0, device=device, dtype=torch.float32)
        self.emergency_protocols = {
            'engine_shutdown': False,
            'power_conservation': False,
            'damage_control': False,
            'life_support_backup': False
        }

        # Sensör sistemleri
        self.sensors = {
            'radiation': torch.tensor(0.0, device=device, dtype=torch.float32),
            'temperature': torch.tensor(300.0, device=device, dtype=torch.float32),
            'pressure': torch.tensor(1.0, device=device, dtype=torch.float32),
            'acceleration': torch.tensor(0.0, device=device, dtype=torch.float32),
            'magnetic_field': torch.tensor(0.0, device=device, dtype=torch.float32)
        }

        # Performans sınırları
        self.limits = {
            'max_acceleration': torch.tensor(MAX_ACCELERATION, device=device, dtype=torch.float32),
            'max_speed': torch.tensor(MAX_SPACECRAFT_SPEED, device=device, dtype=torch.float32),
            'max_temp': torch.tensor(2000.0, device=device, dtype=torch.float32),
            'min_temp': torch.tensor(100.0, device=device, dtype=torch.float32),
            'max_radiation': torch.tensor(1000.0, device=device, dtype=torch.float32),
            'max_pressure': torch.tensor(2.0, device=device, dtype=torch.float32)
        }

```

```python
    def apply_action(self, action):
        """Uzay aracına eylemi uygula"""
        try:
            # Eylemi tensor'a çevir
            if isinstance(action, np.ndarray):
                action = torch.from_numpy(action).float().to(self.device)

            # İtki yönü ve büyüklüğünü ayır
            thrust_direction = action[:2]  # x, y yönü
            thrust_magnitude = torch.abs(action[2])  # İtki büyüklüğü

            # İtki yönünü normalize et
            thrust_direction = thrust_direction / (torch.norm(thrust_direction) + 1e-8)

            # İtki kuvvetini hesapla (Newton)
            MAX_THRUST = 1000.0  # Newton
            thrust_force = thrust_magnitude * MAX_THRUST * thrust_direction

            # Kuvveti uygula
            self.acceleration = thrust_force / self.mass

        except Exception as e:
            print(f"Action uygulama hatası: {e}")

    def update(self, dt: float):
        """Uzay aracının durumunu güncelle"""
        try:
            # Tensor boyutlarını kontrol et
            assert self.position.size() == (2,), f"Position size error: {self.position.size()}"
            assert self.velocity.size() == (2,), f"Velocity size error: {self.velocity.size()}"
            assert self.acceleration.size() == (2,), f"Acceleration size error: {self.acceleration.size()}"

            # dt'yi tensor'a çevir
            dt = torch.tensor(dt, device=self.device)

            # Vektör boyutlarını kontrol et ve düzelt
            if self.velocity.shape != (2,):
                self.velocity = self.velocity[:2]
            if self.acceleration.shape != (2,):
                self.acceleration = self.acceleration[:2]

            # Pozisyonu güncelle
            self.position = self.position + self.velocity * dt + 0.5 * self.acceleration * dt * dt

```

```
        # Hız sınırlaması
        speed = torch.norm(self.velocity)
        if speed > MAX_SPACECRAFT_SPEED:
            self.velocity *= (MAX_SPACECRAFT_SPEED / speed)

        # Yakıt sızıntısı varsa yakıt kaybını hesapla
        if self.has_fuel_leak:
            fuel_loss = self.fuel_leak_rate * dt
            self.fuel_mass = max(0.0, self.fuel_mass - fuel_loss)
            self.mass = self.dry_mass + self.fuel_mass

        # Motor sıcaklığını güncelle
        if torch.norm(self.acceleration) > 0:
            # Motor çalışıyorsa ısınma
            self.engine_temperature = torch.clamp(
                self.engine_temperature + self.heating_rate * dt,
                min=self.min_engine_temp,
                max=self.max_engine_temp
            )

            # Aşırı ısınma durumunda motor hasarı
            if self.engine_temperature > self.max_engine_temp * 0.9:
                damage_factor = (self.engine_temperature - self.max_engine_temp * 0.9) / (self.max_engine_temp * 0.1)
                self.hull_damage += damage_factor * 0.01 * dt
        else:
            # Motor çalışmıyorsa soğuma
            self.engine_temperature = torch.clamp(
                self.engine_temperature - self.cooling_rate * dt,
                min=self.min_engine_temp,
                max=self.max_engine_temp
            )

        # Onarım işlemlerini güncelle
        if self.repair_in_progress:
            self.repair_time_remaining = max(0.0, self.repair_time_remaining - dt)
            if self.repair_time_remaining == 0:
                self.hull_damage *= (1 - self.repair_efficiency)
                self.repair_in_progress = False
```

```
            # Sistem durumlarını güncelle
            self.update_emergency_systems(dt)

            # Geçmiş kayıtlarını güncelle
            self._update_history()

        except Exception as e:
            print(f"Update hatası: {e}")
            print(f"Position size: {self.position.size()}")
            print(f"Velocity size: {self.velocity.size()}")
            print(f"Acceleration size: {self.acceleration.size()}")
            raise e

    def _update_history(self):
        """Geçmiş kayıtlarını güncelle"""
        self.thrust_history.append(float(torch.norm(self.acceleration)))
        self.speed_history.append(float(torch.norm(self.velocity)))
        self.acceleration_history.append(float(torch.norm(self.acceleration)))
        self.position_history.append(self.position.clone())
        self.fuel_consumption_history.append(float(self.fuel_mass))

    def get_system_status(self) -> Dict[str, float]:
        """Sistemlerin genel durumunu döndürür."""
        return {
            'hull_integrity': float(self.hull_integrity),
            'fuel_remaining': float(self.fuel_mass),
            'power_level': float(self.power_level),
            'engine_health': float(self.system_health['propulsion']),
            'navigation_accuracy': float(self.navigation_accuracy),
            'life_support': float(self.life_support_health),
            'communication': float(self.communication_strength),
            'thermal_load': float(self.thermal_load)
        }

    def get_engine_efficiency(self) -> float:
        """Motor verimliliğini hesapla"""
        if self.engine_temperature < self.min_engine_temp:
            return 0.0
        elif self.engine_temperature > self.max_engine_temp:
            return 0.0
        elif self.optimal_temp_range[0] <= self.engine_temperature <= self.optimal_temp_range[1]:
            return 1.0
        else:
```

```python
        # Optimal aralık dışındaysa azalan verimlilik
        if self.engine_temperature < self.optimal_temp_range[0]:
            t_diff = (self.engine_temperature - self.min_engine_temp) / (self.optimal_temp_range[0] - self.min_engine_temp)
        else:
            t_diff = (self.max_engine_temp - self.engine_temperature) / (self.max_engine_temp - self.optimal_temp_range[1])
        return max(0.0, min(1.0, t_diff))

    def apply_thrust(self, action: torch.Tensor) -> None:
        """Uzay aracına itki uygula"""
        # Action vektörünü parçalara ayır
        thrust_direction = action[:2]  # İlk 2 eleman yön vektörü
        thrust_magnitude = action[2]   # 3. eleman itki büyüklüğü
        correction = action[3]         # 4. eleman düzeltme manevrası

        # Yön vektörünü normalize et
        thrust_direction = thrust_direction / (torch.norm(thrust_direction) + 1e-8)

        # İtki kuvvetini hesapla (Newton)
        max_thrust = 5000.0  # Maximum thrust in Newtons
        thrust_force = thrust_magnitude * max_thrust * thrust_direction

        # İvmeyi hesapla (F = ma)
        self.acceleration = thrust_force / self.mass

        # Düzeltme manevrasını uygula
        correction_force = torch.tensor([
            -self.velocity[1],
            self.velocity[0]
        ], device=self.device) * correction

        # Toplam ivmeyi güncelle
        self.acceleration += correction_force / self.mass

        # Motor sıcaklığını güncelle
        self.engine_temperature += self.heating_rate * thrust_magnitude
        self.engine_temperature = max(self.min_engine_temp,
                                      min(self.max_engine_temp,
                                          self.engine_temperature))

        # Yakıt tüketimi
        fuel_consumption = thrust_magnitude * 0.1  # kg/s
        self.fuel_mass = max(0.0, self.fuel_mass - fuel_consumption)
```

```python
    def apply_force(self, force: torch.Tensor, thrust_direction: torch.Tensor = None,
                    thrust_magnitude: float = 0.0, correction: float = 0.0):
        """Uzay aracına kuvvet uygula"""
        try:
            total_force = force.clone()

            # İtki kuvveti varsa ekle
            if thrust_direction is not None and thrust_magnitude > 0:
                # Yön vektörnü normalize et
                thrust_direction = thrust_direction / (torch.norm(thrust_direction) + 1e-8)

                # İtki kuvvetini hesapla
                max_thrust = 5000.0  # Maximum thrust in Newtons
                thrust_force = thrust_magnitude * max_thrust * thrust_direction
                total_force += thrust_force

                # Düzeltme manevrasını uygula
                if correction != 0:
                    correction_force = torch.tensor([
                        -self.velocity[1],
                        self.velocity[0]
                    ], device=self.device) * correction
                    total_force += correction_force

                # Motor sıcaklığını güncelle
                self.engine_temperature += self.heating_rate * thrust_magnitude
                self.engine_temperature = torch.clamp(
                    self.engine_temperature,
                    min=self.min_engine_temp,
                    max=self.max_engine_temp
                )

                # Yakıt tüketimi
                fuel_consumption = thrust_magnitude * 0.1  # kg/s
                self.fuel_mass = max(0.0, self.fuel_mass - fuel_consumption)
                self.mass = self.dry_mass + self.fuel_mass

            # İvmeyi hesapla (F = ma)
            self.acceleration = total_force / self.mass

        except Exception as e:
            print(f"Kuvvet uygulama hatası: {e}")
            traceback.print_exc()
```

```python
    def apply_gravity_forces(self, planets: List[Planet]):
        total_force = torch.zeros(2, device=self.device)
        for planet in planets:
            r = planet.position - self.position
            r_mag = torch.norm(r)
            force = G * planet.mass * self.mass * r / (r_mag ** 3)
            total_force += force
        self.acceleration = total_force / self.mass

    def update(self, dt: float):
        """Uzay aracının durumunu güncelle"""
        try:
            dt = torch.tensor(dt, device=self.device)

            # Acceleration tensor'unu düzelt (2,4) -> (2,)
            if self.acceleration.dim() > 1:
                self.acceleration = self.acceleration[:, 0]  # İlk sütunu al

            # Boyut kontrolü
            self.acceleration = self.acceleration[:2]  # Sadece x,y bileşenlerini al
            self.velocity = self.velocity[:2]
            self.position = self.position[:2]

            # Güncelleme
            delta_pos = (self.velocity * dt) + (0.5 * self.acceleration * dt * dt)
            self.position = self.position + delta_pos
            self.velocity = self.velocity + (self.acceleration * dt)

        except Exception as e:
            print(f"Update hatası detayı: {e}")
            print(f"Position boyutu: {self.position.size()}")
            print(f"Velocity boyutu: {self.velocity.size()}")
            print(f"Acceleration boyutu: {self.acceleration.size()}")
            print(f"dt değeri: {dt}")

    def get_emergency_status(self) -> Dict[str, bool]:
        """Acil durum sistemlerinin durumunu döndürür."""
        return {
            'emergency_active': self.emergency_systems_active,
            'engine_shutdown': self.emergency_protocols['engine_shutdown'],
            'power_conservation': self.emergency_protocols['power_conservation'],
            'damage_control': self.emergency_protocols['damage_control'],
```

```
            'damage_control': self.emergency_protocols['damage_control'],
            'life_support_backup': self.emergency_protocols['life_support_backup']
        }

    def get_sensor_readings(self) -> Dict[str, float]:
        """Sensör okumalarını döndürür."""
        return {key: float(value) for key, value in self.sensors.items()}

    def get_performance_metrics(self) -> Dict[str, float]:
        """Performans metriklerini döndürür."""
        return {
            'delta_v_remaining': float(self.delta_v_remaining),
            'mission_time': float(self.mission_time),
            'distance_traveled': float(self.total_distance_traveled),
            'current_speed': float(torch.norm(self.velocity)),
            'current_acceleration': float(torch.norm(self.acceleration))
        }

    def update_emergency_systems(self, time_step: float):
        # Print mesajları için zaman kontrolü
        should_print = not hasattr(self, '_last_warning_time') or \
                       time_step - self._last_warning_time > 300  # 5 dakikada bir uyarı

        # Geliştirilmiş motor sıcaklık yönetimi
        if torch.norm(self.acceleration) > 0:
            heat_generation = time_step * 0.05 * torch.norm(self.acceleration)
            cooling_power = time_step * 0.1 * self.cooling_system_efficiency

            if self.engine_temperature > 1000:
                self.cooling_system_efficiency *= 0.999

            net_temperature_change = heat_generation - cooling_power
            old_temp = float(self.engine_temperature)

            # Sıcaklık değişimini sınırla ve uygula
            self.engine_temperature = torch.clamp(
                self.engine_temperature + net_temperature_change,
                min=self.min_engine_temp,
                max=self.max_engine_temp
            )

            if self.engine_temperature > self.max_engine_temp * 0.9:
                self.system_health['propulsion'] *= 0.995
```

```python
            # Acil durum soğutması
            if self.engine_temperature > self.max_engine_temp * 0.95:
                emergency_cooling = (self.engine_temperature - self.max_engine_temp * 0.95) * 0.1
                self.engine_temperature = torch.clamp(
                    self.engine_temperature - emergency_cooling,
                    min=self.min_engine_temp,
                    max=self.max_engine_temp
                )
                self.power_level *= 0.99

            # Sadece önemli değişikliklerde ve belirli aralıklarla print
            if should_print and (
                self.system_health['propulsion'] < 0.5 or
                abs(float(self.engine_temperature - old_temp)) > 500
            ):
                print(f"\nSistem Durumu:")
                print(f"- Motor Sıcaklığı: {float(self.engine_temperature):.1f}K")
                print(f"- Motor Sağlığı: {float(self.system_health['propulsion'])*100:.1f}%")
                print(f"- Soğutma Verimi: {float(self.cooling_system_efficiency)*100:.1f}%")
                self._last_warning_time = time_step
        else:
            # Motor çalışmıyorken soğuma
            cooling_rate = time_step * 0.2 * self.cooling_system_efficiency
            self.engine_temperature = torch.clamp(
                self.engine_temperature - cooling_rate,
                min=self.min_engine_temp,
                max=self.max_engine_temp
            )

        # Hayat destek sistemleri güncellemesi
        if self.power_level < 0.5:
            self.life_support_health *= 0.99
            self.oxygen_level *=0.99
            if should_print and self.life_support_health < 0.8:
                print("\nKritik Sistem Uyarısı:")
                print(f"- Hayat Destek Sistemi: {float(self.life_support_health)*100:.1f}%")
                print(f"- Oksijen Seviyesi: {float(self.oxygen_level)*100:.1f}%")
                self._last_warning_time = time_step

        # Acil durum protokolleri kontrolü
        critical_systems = {
            'propulsion': self.system_health['propulsion'] < 0.5,
            'power': self.power_level < 0.3,
            'life_support': self.life_support_health < 0.7,
```

```python
            'life_support': self.life_support_health < 0.7,
            'structural': self.hull_integrity < 0.6
        }

        if any(critical_systems.values()) and should_print:
            print("\nACİL DURUM RAPORU:")
            for system, is_critical in critical_systems.items():
                if is_critical:
                    print(f"- {system.upper()} sistemi kritik seviyede!")
            self._last_warning_time = time_step

        # Motor güvenlik protokolü
        if self.engine_temperature > self.max_engine_temp * 0.95 and should_print:
            if not hasattr(self, '_engine_shutdown_announced'):
                print("\nACİL DURUM: Motor güvenlik protokolü devrede!")
                self._engine_shutdown_announced = True
            self.emergency_protocols['engine_shutdown'] = True
            self.acceleration *= 0.1

    def leapfrog_step(self, dt: float, force: torch.Tensor):
        """
        Leapfrog integrasyon yöntemi ile uzay aracının hareketini günceller.
        """
        # İvmeyi güncelle
        self.acceleration = force / self.mass

        # Yarım adım hız güncellemesi
        half_velocity = self.velocity + 0.5 * dt * self.acceleration

        # Pozisyon güncellemesi
        self.position += dt * half_velocity

        # İvmeyi tekrar hesapla (yeni pozisyonda)
        self.acceleration = force / self.mass

        # Hızın kalan yarısını güncelle
        self.velocity = half_velocity + 0.5 * dt * self.acceleration

        # Hız sınırlaması
        current_acceleration = torch.norm(self.acceleration)
        max_safe_acceleration = self.limits['max_acceleration'] * self.system_health['propulsion']

```

```python
        # Hız sınırlaması
        current_acceleration = torch.norm(self.acceleration)
        max_safe_acceleration = self.limits['max_acceleration'] * self.system_health['propulsion']

        if current_acceleration > max_safe_acceleration:
            old_acceleration = float(current_acceleration)
            self.acceleration *= (max_safe_acceleration / current_acceleration)

            # Sadece büyük değişikliklerde ve aralıklı olarak print
            if (not hasattr(self, '_last_acceleration_warning') or
                dt - self._last_acceleration_warning > 300) and \
                abs(old_acceleration - float(max_safe_acceleration)) > 10:

                print(f"\nİvme Kontrolü: {old_acceleration:.2f} -> {float(max_safe_acceleration):.2f} m/s²")
                self._last_acceleration_warning = dt

            # İvme sınırlaması
            current_acceleration = torch.norm(self.acceleration)
            if current_acceleration > self.limits['max_acceleration']:
                self.acceleration *= (self.limits['max_acceleration'] / current_acceleration)

            # Geçmiş kayıtlarını güncelle
            self.position_history.append(self.position.clone())
            self.speed_history.append(float(torch.norm(self.velocity)))
            self.acceleration_history.append(float(current_acceleration))

            # Yörünge parametrelerini güncelle
            self.update_orbit_parameters()

    def update_orbit_parameters(self):
        """Yörünge parametrelerini günceller."""
        try:
            r = self.position
            v = self.velocity

            # 2D uzayda çalışıyoruz, z bileşeni 0 olan 3D vektörler oluştur
            r_3d = torch.cat([r, torch.tensor([0.0], device=device, dtype=torch.float32)])
            v_3d = torch.cat([v, torch.tensor([0.0], device=device, dtype=torch.float32)])

            # Spesifik açısal momentum (3D cross product için)
            h = torch.cross(r_3d.unsqueeze(0), v_3d.unsqueeze(0))[0]

            # Yarı-major eksen
            r_mag = torch.norm(r)
```

```
            v_mag = torch.norm(v)
            a = -G * SUN_MASS / (v_mag**2 - 2 * G * SUN_MASS / r_mag)

            # Eksantrisite vektörü
            e_vec = torch.cross(v_3d.unsqueeze(0), h.unsqueeze(0))[0] / (G * SUN_MASS) - r_3d / r_mag
            e = torch.norm(e_vec)

            # İnklinasyon (2D'de her zaman 0)
            i = torch.tensor(0.0, device=device, dtype=torch.float32)

            # Yörünge parametrelerini kaydet
            orbit_params = {
                'semi_major_axis': float(a),
                'eccentricity': float(e),
                'inclination': float(i),
                'angular_momentum': float(torch.norm(h))
            }

            self.orbit_parameters_history.append(orbit_params)

        except Exception as e:
            print(f"Yörünge parametreleri hesaplanırken hata: {e}")
            # Hata durumunda varsayılan değerler
            self.orbit_parameters_history.append({
                'semi_major_axis': 0.0,
                'eccentricity': 0.0,
                'inclination': 0.0,
                'angular_momentum': 0.0
            })

    def update_mass(self, fuel_consumed: float):
        """Yakıt tüketimini güncelleyerek kütle değişimini hesaplar."""
        fuel_consumed = torch.tensor(fuel_consumed, device=device, dtype=torch.float32)
        self.fuel_mass = torch.maximum(
            torch.tensor(0.0, device=device, dtype=torch.float32),
            self.fuel_mass - fuel_consumed
        )
        self.mass = self.dry_mass + self.fuel_mass
        self.fuel_consumption_history.append(float(fuel_consumed))

    def handle_motor_failure(self) -> bool:
        """Motor arızasını yönetir."""
        if self.backup_engines > 0:
            self.backup_engines -= 1
```

```
            self.current_engine += 1
            print(f"Motor arızası! Yedek motor #{self.current_engine} devrede.")
            print(f"Kalan yedek motor: {self.backup_engines}")
            return True
        else:
            print("Kritik: Tüm motorlar arızalı!")
            return False

    def handle_fuel_leak(self) -> bool:
        """Yakıt sızıntısını yönetir."""
        if not self.has_fuel_leak:
            self.has_fuel_leak = True
            self.fuel_leak_rate = torch.tensor(
                random.uniform(0.0001, 0.001),
                device=device,
                dtype=torch.float32
            )
            print(f"Yakıt sızıntısı tespit edildi! Sızıntı oranı: {self.fuel_leak_rate*100:.4f}%/s")
            return True
        else:
            print("Kritik: İkinci yakıt sızıntısı! Sistem başarısız!")
            return False

    def handle_microasteroid_collision(self) -> bool:
        """Mikroasteroit çarpışmasını yönetir."""
        damage = torch.tensor(
            random.uniform(0.01, 0.05),
            device=device,
            dtype=torch.float32
        )
        self.hull_damage += damage
        print(f"Mikro asteroit çarpışması! Hasar: {damage*100:.1f}%, Toplam hasar: {self.hull_damage*100:.1f}%")

        if self.hull_damage >= self.critical_damage_threshold:
            print("Kritik: Gövde bütünlğü kritik seviyede!")
            return False

        if not self.repair_in_progress and self.auto_repair_systems:
            self.repair_in_progress = True
            self.repair_time_remaining = torch.tensor(3600.0, device=device, dtype=torch.float32)

        return True
```

```python

    def calculate_orbital_elements(self, params):
        initial_angle = params['true_anomaly']

        # Açıyı tensor olarak bir kez oluştur
        angle_tensor = torch.tensor(initial_angle, device=self.device)
        cos_angle = torch.cos(angle_tensor)
        sin_angle = torch.sin(angle_tensor)

        # Yörünge yarıçapı
        r = params['semi_major_axis'] * \
            (1 - params['eccentricity']**2) / \
            (1 + params['eccentricity'] * cos_angle)

        # Pozisyon vektörü
        position = torch.tensor([
            r * cos_angle,
            r * sin_angle
        ], device=self.device)

        # Hız büyüklüğü
        velocity_magnitude = torch.sqrt(G * SUN_MASS * (2/r - 1/params['semi_major_axis']))

        # Hız vektörü
        velocity = torch.tensor([
            -velocity_magnitude * sin_angle,
            velocity_magnitude * cos_angle
        ], device=self.device)

        return position, velocity
```

```python

class TransformerMemory(nn.Module):
    def __init__(self, state_size = 41, action_size = 4, memory_size = 1000):
        super().__init__()
        self.state_size = state_size
        self.action_size = action_size
        self.memory_size = memory_size
        self.d_model = 256  # Transformer model boyutu
        self.nhead = 8      # Multi-head attention başlık sayısı
        self.num_layers = 4  # Transformer katman sayısı
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        # Bellek matrisi [state, action, reward, next_state]
        self.memory = torch.zeros(
            memory_size,
            2*state_size + action_size + 1,  # [state + next_state + action + reward]
            device=self.device
        )

        # Pozisyon kodlaması
        self.pos_encoder = nn.Parameter(
            self._create_position_encoding(),
            requires_grad=False
        )

        # Giriş projeksiyonu
        self.input_proj = nn.Sequential(
            nn.Linear(2*state_size + action_size + 1, self.d_model),
            nn.LayerNorm(self.d_model),
            nn.ReLU()
        )

        # Transformer kodlayıcı
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=self.d_model,
            nhead=self.nhead,
            dim_feedforward=1024,
            dropout=0.1,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(
            encoder_layer,
            num_layers=self.num_layers
        )

```

```python

        # Çıkış projeksiyonu
        self.output_proj = nn.Sequential(
            nn.Linear(self.d_model, self.d_model),
            nn.ReLU(),
            nn.Linear(self.d_model, action_size)
        )

        # Bellek indeksi
        self.current_idx = 0

        # Önem ağırlıkları
        self.importance_net = nn.Sequential(
            nn.Linear(2*state_size + action_size + 1, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )

        self.to(self.device)

    def _create_position_encoding(self):
        """Sinüzoidal pozisyon kodlaması oluştur"""
        position = torch.arange(self.memory_size).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, self.d_model, 2) * (-math.log(10000.0) / self.d_model))
        pe = torch.zeros(self.memory_size, self.d_model)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        return pe

    def add_experience(self, state, action, reward, next_state):
        """Yeni deneyimi belleğe ekle"""
        # Deneyimi birleştir
        experience = torch.cat([
            state.flatten(),
            action.flatten(),
            reward.view(-1),
            next_state.flatten()
        ])

        # Önem değerini hesapla
        importance = self.importance_net(experience.unsqueeze(0)).item()
```

```
        importance = self.importance_net(experience.unsqueeze(0)).item()

        # Eğer bellek doluysa ve yeni deneyim önemliyse
        if self.current_idx >= self.memory_size:
            # En düşük önemli deneyimi bul
            importances = torch.tensor([
                self.importance_net(self.memory[i].unsqueeze(0)).item()
                for i in range(self.memory_size)
            ])
            min_idx = torch.argmin(importances)

            # Yeni deneyim daha önemliyse değiştir
            if importance > importances[min_idx]:
                self.memory[min_idx] = experience
        else:
            # Bellek dolmamışsa direkt ekle
            self.memory[self.current_idx] = experience
            self.current_idx = (self.current_idx + 1) % self.memory_size

    def query_memory(self, current_state: torch.Tensor, k: int = 10) -> torch.Tensor:
        """Mevcut duruma en benzer k deneyimi getir"""
        with torch.no_grad():
            # Mevcut durumu kodla
            state_encoding = self.input_proj(current_state.unsqueeze(0))

            # Belleği kodla
            memory_encoded = self.input_proj(self.memory) + self.pos_encoder

            # Attention skorlarını hesapla
            attention_scores = torch.matmul(
                state_encoding, memory_encoded.transpose(-2, -1)
            ) / math.sqrt(self.d_model)
            attention_weights = F.softmax(attention_scores, dim=-1)

            # En benzer k deneyimi seç
            _, indices = torch.topk(attention_weights.squeeze(), k)
            selected_memories = self.memory[indices]

            # Transformer ile işle
            encoded = self.transformer(
                self.input_proj(selected_memories) + self.pos_encoder[:k]
            )

```

```
1434
1435            # Çıktıyı hesapla
1436            output = self.output_proj(encoded.mean(dim=0))
1437
1438            return output
1439
1440    def get_memory_influence(self, state: torch.Tensor) -> torch.Tensor:
1441        """Belleğin mevcut duruma etkisini hesapla"""
1442        with torch.no_grad():
1443            # En benzer deneyimleri al
1444            memory_output = self.query_memory(state)
1445
1446            # Bellek etkisini hesapla (0-1 arası)
1447            memory_influence = torch.sigmoid(
1448                torch.norm(memory_output) / math.sqrt(self.action_size)  # Kapanış parantezi eklendi
1449            )
1450
1451            return memory_influence
```

```python
class SACNetwork(nn.Module):
    def __init__(self, state_size: int, action_size: int):
        super().__init__()
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.state_size = state_size  # Yeni eklenen satır
        self.action_size = action_size  # Yeni eklenen satır

        # Actor ağı eklendi
        self.actor = nn.Sequential(
            nn.Linear(state_size, 2048),
            nn.LayerNorm(2048),
            nn.ReLU(),
            nn.Linear(2048, 1024),
            nn.LayerNorm(1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.ReLU()
        ).to(self.device)

        # Genişletilmiş Critic ağları
        self.q1_network = nn.Sequential(
            nn.Linear(state_size + action_size, 2048),
            nn.LayerNorm(2048),
            nn.ReLU(),
            nn.Linear(2048, 1024),
            nn.LayerNorm(1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)

        self.q2_network = nn.Sequential(
            nn.Linear(state_size + action_size, 2048),
            nn.LayerNorm(2048),
            nn.ReLU(),
            nn.Linear(2048, 1024),
            nn.LayerNorm(1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
```

```
            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)

        # Genişletilmiş Value ağı
        self.value_network = nn.Sequential(
            nn.Linear(state_size, 2048),
            nn.LayerNorm(2048),
            nn.ReLU(),
            nn.Linear(2048, 1024),
            nn.LayerNorm(1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)

        # Target value ağı
        self.target_value_network = copy.deepcopy(self.value_network)

        # Mean ve log_std başlıkları
        self.mean_head = nn.Sequential(
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, action_size)
        ).to(self.device)

        self.log_std_head = nn.Sequential(
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, action_size)
        ).to(self.device)
```

```python
        # Optimizerlar
        self.actor_optimizer = optim.Adam(
            list(self.actor.parameters()) +
            list(self.mean_head.parameters()) +
            list(self.log_std_head.parameters()),
            lr=3e-4
        )
        self.q1_optimizer = optim.Adam(self.q1_network.parameters(), lr=3e-4)
        self.q2_optimizer = optim.Adam(self.q2_network.parameters(), lr=3e-4)
        self.value_optimizer = optim.Adam(self.value_network.parameters(), lr=3e-4)

        # Entropy ayarı için alpha parametresi
        self.log_alpha = nn.Parameter(torch.zeros(1, requires_grad=True, device=self.device))
        self.alpha_optimizer = optim.Adam([self.log_alpha], lr=3e-4)
        self.target_entropy = -action_size

        # Physics encoder eklendi
        self.physics_encoder = nn.Sequential(
            nn.Linear(state_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU()
        ).to(self.device)

        # Physics forward metodu için conservation layer eklendi
        self.conservation_layer = nn.Sequential(
            nn.Linear(256, 128),
            nn.LayerNorm(128),
            nn.ReLU(),
            nn.Linear(128, 4)  # [energy, momentum_x, momentum_y, momentum_z]
        ).to(self.device)

        # Relativistic layer eklendi
        self.relativistic_layer = nn.Sequential(
            nn.Linear(256, 128),
            nn.LayerNorm(128),
            nn.ReLU(),
            nn.Linear(128, 1)  # gamma factor
        ).to(self.device)
```

```
        # Scaler eklendi
        self.scaler = torch.amp.GradScaler('cuda')

        # Ağırlık başlatma
        self.apply(self._init_weights)

        # Adaptif öğrenme oranı için parametreler
        self.initial_lr = 3e-4
        self.min_lr = 1e-5
        self.lr_decay = 0.995

        # Sıcaklık parametresi için adaptif ayarlama
        self.target_entropy = -action_size  # Hedef entropi değeri
        self.log_alpha = nn.Parameter(torch.zeros(1, requires_grad=True, device=self.device))
        self.alpha_optimizer = optim.Adam([self.log_alpha], lr=3e-4)

        # Gradient clipping değeri
        self.max_grad_norm = 1.0

        # Optimizerları güncelle - adaptif öğrenme oranı ile
        self.actor_optimizer = optim.Adam(
            list(self.actor.parameters()) +
            list(self.mean_head.parameters()) +
            list(self.log_std_head.parameters()),
            lr=self.initial_lr
        )

        self.q1_optimizer = optim.Adam(self.q1_network.parameters(), lr=self.initial_lr)
        self.q2_optimizer = optim.Adam(self.q2_network.parameters(), lr=self.initial_lr)
        self.value_optimizer = optim.Adam(self.value_network.parameters(), lr=self.initial_lr)

        # Gelişmiş deneyim replay buffer
        self.buffer_size = 1_000_000
        self.batch_size = 256
        self.replay_buffer = {
            'state': [],
            'action': [],
            'reward': [],
            'next_state': [],
            'done': [],
            'priority': []  # Öncelikli deneyim replay için
        }
```

```
1625        self.per_alpha = 0.6  # Öncelik üssü
1626        self.per_beta = 0.4   # Önemli örnekleme düzeltme faktörü
1627        self.per_epsilon = 1e-6  # Küçük pozitif değer
1628
1629    def _init_weights(self, module):
1630        """Ağırlıkları Xavier/Glorot başlatması ile başlat"""
1631        if isinstance(module, (nn.Linear, nn.Conv2d)):
1632            nn.init.xavier_uniform_(module.weight)
1633            if module.bias is not None:
1634                module.bias.data.fill_(0.01)
1635
1636    def forward_critic(self, state, action):
1637        """Genişletilmiş critic ileri yayılımı"""
1638        x = torch.cat([state, action], dim=-1)
1639        q1 = self.q1_network(x)
1640        q2 = self.q2_network(x)
1641        return q1, q2
1642
1643    def forward_value(self, state):
1644        """Genişletilmiş value ileri yayılımı"""
1645        return self.value_network(state)
1646
1647    def forward(self, state):
1648        """Genişletilmiş ileri yayılım"""
1649        try:
1650            # Giriş boyutunu kontrol et ve düzelt
1651            if isinstance(state, np.ndarray):
1652                state = torch.FloatTensor(state).to(self.device)
1653
1654            # Batch boyutu ve özellik sayısını ayır
1655            batch_size = state.size(0)
1656
1657            # State'i doğru boyuta getir
1658            if state.size(-1) != self.state_size:
1659                # Eksik özellikleri sıfırlarla doldur
1660                padded_state = torch.zeros(batch_size, self.state_size, device=self.device)
1661                padded_state[:, :state.size(-1)] = state
1662                state = padded_state
1663
1664            # Actor özellikleri
1665            with torch.amp.autocast('cuda'):
1666                actor_features = self.actor(state)
1667                actor_features = torch.clamp(actor_features, -10.0, 10.0)
1668
```

```
                # Mean ve log_std hesaplama
                mean = self.mean_head(actor_features)
                log_std = self.log_std_head(actor_features)

                # Sınırlama ve stabilizasyon
                mean = torch.clamp(mean, -1.0, 1.0)
                log_std = torch.clamp(log_std, -5.0, 2.0)

                # Fizik çıktısı
                with torch.no_grad():
                    physics_output = self.physics_forward(state)
                    physics_output = torch.clamp(physics_output, -1.0, 1.0)

                # Adaptif ağırlıklandırma
                alpha = torch.sigmoid(self.log_alpha)
                combined_mean = alpha * mean + (1 - alpha) * physics_output

                # Son sınırlama
                combined_mean = torch.clamp(combined_mean, -1.0, 1.0)

                return combined_mean, log_std

        except Exception as e:
            print(f"Forward pass hatası: {e}")
            print(f"Giriş boyutu: {state.shape}")
            print(f"Beklenen boyut: {self.state_size}")
            return (torch.zeros((state.size(0), self.action_size), device=self.device),
                    torch.full((state.size(0), self.action_size), -5.0, device=self.device))

    def sample_action(self, state):
        """Geliştirilmiş aksiyon örnekleme"""
        try:
            mean, log_std = self.forward(state)

            # NaN kontrolü
            mean = torch.nan_to_num(mean, nan=0.0)
            log_std = torch.nan_to_num(log_std, nan=-20.0)

            std = torch.exp(log_std) + 1e-6  # Sayısal stabilite için epsilon ekle
```

```
            # Normal dağılım örneklemesi
            normal = torch.distributions.Normal(mean, std)
            x_t = normal.rsample()

            # Tanh sınırlama
            action = torch.tanh(x_t)

            # Log olasılık hesaplama
            log_prob = normal.log_prob(x_t)
            log_prob = torch.nan_to_num(log_prob, nan=0.0)  # NaN kontrolü

            # Tanh düzeltmesi
            log_prob -= torch.log(1 - action.pow(2) + 1e-6)
            log_prob = log_prob.sum(-1, keepdim=True)

            # Son NaN kontrolü
            action = torch.nan_to_num(action, nan=0.0)
            log_prob = torch.nan_to_num(log_prob, nan=0.0)

            return action, log_prob

        except Exception as e:
            print(f"Sample action hatası: {e}")
            return (torch.zeros_like(mean),
                    torch.zeros((mean.size(0), 1), device=self.device))

    def physics_forward(self, state):
        """Fizik tabanlı forward pass"""
        physics_features = self.physics_encoder(state)
        conservation_pred = self.conservation_layer(physics_features)
        relativistic_pred = self.relativistic_layer(physics_features)

        return conservation_pred[:, :self.action_size]  # Sadece eylem boyutuna kadar al

    def update_learning_rate(self, episode: int):
        """Öğrenme oranını adaptif olarak güncelle"""
        for optimizer in [self.actor_optimizer, self.q1_optimizer,
                          self.q2_optimizer, self.value_optimizer]:
            current_lr = optimizer.param_groups[0]['lr']
            new_lr = max(self.min_lr, current_lr * self.lr_decay)
            for param_group in optimizer.param_groups:
                param_group['lr'] = new_lr

```

```
            # Normal dağılım örneklemesi
            normal = torch.distributions.Normal(mean, std)
            x_t = normal.rsample()

            # Tanh sınırlama
            action = torch.tanh(x_t)

            # Log olasılık hesaplama
            log_prob = normal.log_prob(x_t)
            log_prob = torch.nan_to_num(log_prob, nan=0.0)  # NaN kontrolü

            # Tanh düzeltmesi
            log_prob -= torch.log(1 - action.pow(2) + 1e-6)
            log_prob = log_prob.sum(-1, keepdim=True)

            # Son NaN kontrolü
            action = torch.nan_to_num(action, nan=0.0)
            log_prob = torch.nan_to_num(log_prob, nan=0.0)

            return action, log_prob

        except Exception as e:
            print(f"Sample action hatası: {e}")
            return (torch.zeros_like(mean),
                    torch.zeros((mean.size(0), 1), device=self.device))

    def physics_forward(self, state):
        """Fizik tabanlı forward pass"""
        physics_features = self.physics_encoder(state)
        conservation_pred = self.conservation_layer(physics_features)
        relativistic_pred = self.relativistic_layer(physics_features)

        return conservation_pred[:, :self.action_size]  # Sadece eylem boyutuna kadar al

    def update_learning_rate(self, episode: int):
        """Öğrenme oranını adaptif olarak güncelle"""
        for optimizer in [self.actor_optimizer, self.q1_optimizer,
                          self.q2_optimizer, self.value_optimizer]:
            current_lr = optimizer.param_groups[0]['lr']
            new_lr = max(self.min_lr, current_lr * self.lr_decay)
            for param_group in optimizer.param_groups:
                param_group['lr'] = new_lr

```

```python
    def add_experience(self, state, action, reward, next_state, done):
        """Gelişmiş deneyim ekleme"""
        # TD hatası ile öncelik hesapla
        with torch.no_grad():
            current_q1, current_q2 = self.forward_critic(state, action)
            next_value = self.forward_value(next_state)
            target_q = reward + (1 - done) * self.gamma * next_value
            td_error = abs(target_q - torch.min(current_q1, current_q2)).item()

        # Öncelikli deneyim replay için öncelik hesapla
        priority = (td_error + self.per_epsilon) ** self.per_alpha

        # Buffer'a ekle
        if len(self.replay_buffer['state']) >= self.buffer_size:
            # En düşük öncelikli deneyimi çıkar
            min_priority_idx = np.argmin(self.replay_buffer['priority'])
            for key in self.replay_buffer:
                self.replay_buffer[key].pop(min_priority_idx)

        # Yeni deneyimi ekle
        self.replay_buffer['state'].append(state)
        self.replay_buffer['action'].append(action)
        self.replay_buffer['reward'].append(reward)
        self.replay_buffer['next_state'].append(next_state)
        self.replay_buffer['done'].append(done)
        self.replay_buffer['priority'].append(priority)

    def sample_batch(self):
        """Öncelikli örnekleme ile batch seç"""
        buffer_size = len(self.replay_buffer['state'])
        if buffer_size < self.batch_size:
            return None

        # Önceliklere göre örnekleme olasılıkları
        probs = np.array(self.replay_buffer['priority'])
        probs = probs / np.sum(probs)

        # Öncelikli örnekleme
        indices = np.random.choice(
            buffer_size,
            self.batch_size,
            p=probs,
            replace=False
        )
```

```
        # Importance sampling ağırlıkları
        weights = (buffer_size * probs[indices]) ** (-self.per_beta)
        weights = weights / weights.max()

        # Batch'i hazırla
        batch = {
            'state': torch.stack([self.replay_buffer['state'][i] for i in indices]),
            'action': torch.stack([self.replay_buffer['action'][i] for i in indices]),
            'reward': torch.tensor([self.replay_buffer['reward'][i] for i in indices], device=self.device),
            'next_state': torch.stack([self.replay_buffer['next_state'][i] for i in indices]),
            'done': torch.tensor([self.replay_buffer['done'][i] for i in indices], device=self.device),
            'weights': torch.FloatTensor(weights).to(self.device),
            'indices': indices
        }

        return batch

    def update(self, batch):
        """Geliştirilmiş güncelleme fonksiyonu"""
        # Batch verilerini al
        states = batch['state']
        actions = batch['action']
        rewards = batch['reward']
        next_states = batch['next_state']
        dones = batch['done']
        weights = batch['weights']

        # Value loss
        with torch.amp.autocast('cuda'):
            current_value = self.forward_value(states)
            next_value = self.forward_value(next_states)
            next_value = next_value * (1 - dones)
            target_value = rewards + self.gamma * next_value
            value_loss = (weights * F.mse_loss(current_value, target_value.detach(), reduction='none')).mean()

        # Optimize value network
        self.value_optimizer.zero_grad()
        self.scaler.scale(value_loss).backward()
        torch.nn.utils.clip_grad_norm_(self.value_network.parameters(), self.max_grad_norm)
        self.scaler.step(self.value_optimizer)

        # Q losses
        with torch.amp.autocast('cuda'):
```

```python
        # Q losses
        with torch.amp.autocast('cuda'):
            current_q1, current_q2 = self.forward_critic(states, actions)
            q_loss = (weights * (F.mse_loss(current_q1, target_value.detach(), reduction='none') +
                                 F.mse_loss(current_q2, target_value.detach(), reduction='none'))).mean()

        # Optimize Q networks
        self.q1_optimizer.zero_grad()
        self.q2_optimizer.zero_grad()
        self.scaler.scale(q_loss).backward()
        torch.nn.utils.clip_grad_norm_(self.q1_network.parameters(), self.max_grad_norm)
        torch.nn.utils.clip_grad_norm_(self.q2_network.parameters(), self.max_grad_norm)
        self.scaler.step(self.q1_optimizer)
        self.scaler.step(self.q2_optimizer)

        # Policy loss
        with torch.amp.autocast('cuda'):
            new_actions, log_probs = self.sample_action(states)
            alpha = self.log_alpha.exp()
            q1_new = self.forward_critic(states, new_actions, self.q1_network)
            q2_new = self.forward_critic(states, new_actions, self.q2_network)
            q_new = torch.min(q1_new, q2_new)
            policy_loss = (weights * (alpha * log_probs - q_new)).mean()

        # Optimize policy network
        self.actor_optimizer.zero_grad()
        self.scaler.scale(policy_loss).backward()
        torch.nn.utils.clip_grad_norm_(self.actor.parameters(), self.max_grad_norm)
        self.scaler.step(self.actor_optimizer)

        # Alpha loss - adaptif sıcaklık parametresi
        with torch.amp.autocast('cuda'):
            alpha_loss = -(self.log_alpha * (log_probs + self.target_entropy).detach()).mean()

        # Optimize alpha
        self.alpha_optimizer.zero_grad()
        self.scaler.scale(alpha_loss).backward()
        self.scaler.step(self.alpha_optimizer)

        # Scaler güncelle
        self.scaler.update()

        # Öncelikleri güncelle
        td_errors = abs(target_value - torch.min(current_q1, current_q2)).detach().cpu().numpy()
```

```
        new_priorities = (td_errors + self.per_epsilon) ** self.per_alpha
        for idx, priority in zip(batch['indices'], new_priorities):
            self.replay_buffer['priority'][idx] = priority

        return {
            'value_loss': value_loss.item(),
            'q_loss': q_loss.item(),
            'policy_loss': policy_loss.item(),
            'alpha_loss': alpha_loss.item(),
            'alpha': alpha.item()
        }

```

```python
class SACAgent:
    def __init__(self, state_size: int, action_size: int):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.state_size = state_size
        self.action_size = action_size

        # SAC ağları
        self.sac = SACNetwork(state_size, action_size).to(self.device)

        # Hiperparametreler
        self.gamma = 0.99
        self.tau = 0.005
        self.batch_size = 64
        self.memory_size = 100000
        self.memory = []

        self.target_entropy = -action_size  # Hedef entropi değeri
        self.log_alpha = nn.Parameter(torch.zeros(1, requires_grad=True, device=self.device))
        self.alpha_optimizer = optim.Adam([self.log_alpha], lr=3e-4)

       # Boyut düzeltmeleri için value ve critic ağlarının çıktı boyutlarını ayarla
        self.sac.value_network[-1] = nn.Linear(256, 1)  # Son katmanı 1 boyutlu çıktı verecek şekilde değiştir
        self.sac.q1_network[-1] = nn.Linear(256, 1)     # Q ağları için de aynı şekilde
        self.sac.q2_network[-1] = nn.Linear(256, 1)
        self.q1_network = self.sac.q1_network
        self.q2_network = self.sac.q2_network

        # Optimizer'lara referanslar ekle
        self.value_optimizer = self.sac.value_optimizer
        self.q1_optimizer = self.sac.q1_optimizer
        self.q2_optimizer = self.sac.q2_optimizer
        self.actor_optimizer = self.sac.actor_optimizer
        self.alpha_optimizer = self.sac.alpha_optimizer

        # Scaler'ı da ekle
        self.scaler = torch.amp.GradScaler('cuda')

    def remember(self, state, action, reward, next_state, done):
        if len(self.memory) >= self.memory_size:
            self.memory.pop(0)
        self.memory.append((state, action, reward, next_state, done))
```

```python
    def act(self, state):
        """Duruma göre eylem seç"""
        with torch.no_grad():  # Gradient hesaplamayı devre dışı bırak
            if isinstance(state, np.ndarray):
                state = torch.FloatTensor(state).to(self.device)

            # Batch boyutu kontrolü
            if state.dim() == 1:
                state = state.unsqueeze(0)

            # Belleği temizle
            if torch.cuda.is_available():
                torch.cuda.empty_cache()

            action, _ = self.sac.sample_action(state)
            return action.squeeze(0).cpu().numpy()

    def forward_value(self, states):
        """Value network için forward pass"""
        return self.sac.forward_value(states)

    # Yeni metot ekle
    def forward_critic(self, states, actions, network=None):
        """Critic network için forward pass"""
        if network is None:
            return self.sac.forward_critic(states, actions)
        return network(torch.cat([states, actions], dim=-1))
    def sample_action(self, states):
        """Durumlar için eylem örnekle"""
        return self.sac.sample_action(states)

    def replay(self):
        """Deneyim replay ile öğrenme"""
        if len(self.memory) < self.batch_size:
            return {'value_loss': 0, 'q_loss': 0, 'policy_loss': 0, 'alpha_loss': 0}

        # Batch örnekleme
        batch = random.sample(self.memory, self.batch_size)

        # Batch verilerini tensor'lara dönüştür ve GPU'ya taşı
        states = torch.stack([torch.as_tensor(x[0], device=self.device) for x in batch])
        actions = torch.stack([torch.as_tensor(x[1], device=self.device) for x in batch])
        rewards = torch.tensor([x[2] for x in batch], dtype=torch.float32, device=self.device)
```

```
        next_states = torch.stack([torch.as_tensor(x[3], device=self.device) for x in batch])
        dones = torch.tensor([x[4] for x in batch], dtype=torch.float32, device=self.device)

        # Modeli GPU'ya taşı
        self.sac = self.sac.to(self.device)
            # Value loss
        with torch.amp.autocast('cuda'):
            current_value = self.forward_value(states).view(-1, 1)
            next_value = self.forward_value(next_states).view(-1, 1)
            next_value = next_value * (1 - dones).view(-1, 1)
            target_value = rewards.view(-1, 1) + self.gamma * next_value
            value_loss = F.mse_loss(current_value, target_value.detach())

            # Q losses
            current_q1, current_q2 = self.forward_critic(states, actions)
            q_loss = F.mse_loss(current_q1, target_value.detach()) + \
                     F.mse_loss(current_q2, target_value.detach())

            # Policy loss
            new_actions, log_probs = self.sample_action(states)
            alpha = self.log_alpha.exp()
            q1_new = self.forward_critic(states, new_actions, self.q1_network).view(-1, 1)
            q2_new = self.forward_critic(states, new_actions, self.q2_network).view(-1, 1)
            q_new = torch.min(q1_new, q2_new)
            policy_loss = (alpha * log_probs - q_new).mean()

            # Alpha loss
            alpha_loss = -(self.log_alpha * (log_probs + self.target_entropy).detach()).mean()
            # Value network optimization
        self.value_optimizer.zero_grad()
        self.scaler.scale(value_loss).backward(retain_graph=True)
        # Infinity kontrolü ekle
        self.scaler.unscale_(self.value_optimizer)
        torch.nn.utils.clip_grad_norm_(self.sac.value_network.parameters(), 1.0)
        self.scaler.step(self.value_optimizer)
            # Q networks optimization
        self.q1_optimizer.zero_grad()
        self.q2_optimizer.zero_grad()
        self.scaler.scale(q_loss).backward(retain_graph=True)
        # Infinity kontrolü ekle
        self.scaler.unscale_(self.q1_optimizer)
        self.scaler.unscale_(self.q2_optimizer)
```

```
        torch.nn.utils.clip_grad_norm_(self.sac.q1_network.parameters(), 1.0)
        torch.nn.utils.clip_grad_norm_(self.sac.q2_network.parameters(), 1.0)
        self.scaler.step(self.q1_optimizer)
        self.scaler.step(self.q2_optimizer)
            # Policy network optimization
        self.actor_optimizer.zero_grad()
        self.scaler.scale(policy_loss).backward(retain_graph=True)
        # Infinity kontrolü ekle
        self.scaler.unscale_(self.actor_optimizer)
        torch.nn.utils.clip_grad_norm_(self.sac.actor.parameters(), 1.0)
        self.scaler.step(self.actor_optimizer)
            # Alpha optimization
        self.alpha_optimizer.zero_grad()
        self.scaler.scale(alpha_loss).backward()
        # Infinity kontrolü ekle
        self.scaler.unscale_(self.alpha_optimizer)
        torch.nn.utils.clip_grad_norm_([self.log_alpha], 1.0)
        self.scaler.step(self.alpha_optimizer)
            # Scaler güncelle
        self.scaler.update()
            # Soft update target network
        for target_param, param in zip(self.sac.target_value_network.parameters(),
                                       self.sac.value_network.parameters()):
            target_param.data.copy_(
                target_param.data * (1.0 - self.tau) + param.data * self.tau
            )
            return {
            'value_loss': value_loss.item(),
            'q_loss': q_loss.item(),
            'policy_loss': policy_loss.item(),
            'alpha_loss': alpha_loss.item()
        }

    def save_model(self, path):
        torch.save({
            'sac_state_dict': self.sac.state_dict(),
            'value_optimizer_state_dict': self.sac.value_optimizer.state_dict(),
            'q1_optimizer_state_dict': self.sac.q1_optimizer.state_dict(),
            'q2_optimizer_state_dict': self.sac.q2_optimizer.state_dict(),
            'policy_optimizer_state_dict': self.sac.policy_optimizer.state_dict(),
```

```
            'q2_optimizer_state_dict': self.sac.q2_optimizer.state_dict(),
            'policy_optimizer_state_dict': self.sac.policy_optimizer.state_dict(),
            'alpha_optimizer_state_dict': self.sac.alpha_optimizer.state_dict()
        }, path)

    def load_model(self, path):
        checkpoint = torch.load(path)
        self.sac.load_state_dict(checkpoint['sac_state_dict'])
        self.sac.value_optimizer.load_state_dict(checkpoint['value_optimizer_state_dict'])
        self.sac.q1_optimizer.load_state_dict(checkpoint['q1_optimizer_state_dict'])
        self.sac.q2_optimizer.load_state_dict(checkpoint['q2_optimizer_state_dict'])
        self.sac.policy_optimizer.load_state_dict(checkpoint['policy_optimizer_state_dict'])
        self.sac.alpha_optimizer.load_state_dict(checkpoint['alpha_optimizer_state_dict'])
```

```python
class LongTermSACNetwork(SACNetwork):
    def __init__(self, state_size: int, action_size: int):
        super().__init__(state_size, action_size)

        # Uzun vadeli bellek mekanizması
        self.memory = TransformerMemory(state_size, action_size)
        self.actor = nn.Sequential(
            nn.Linear(state_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU()
        ).to(self.device)
        # Mean ve log_std başlıkları
        self.mean_head = nn.Linear(256, action_size).to(self.device)
        self.log_std_head = nn.Linear(256, action_size).to(self.device)
        # Bellek entegrasyonu için ek katmanlar
        self.memory_integration = nn.ModuleDict({
            'state_gate': nn.Sequential(
                nn.Linear(state_size * 2, state_size),
                nn.Sigmoid()
            ),
            'action_gate': nn.Sequential(
                nn.Linear(action_size * 2, action_size),
                nn.Sigmoid()
            )
        })
    def forward_actor(self, state):
        """Actor network için forward pass"""
        features = self.actor(state)
        mean = self.mean_head(features)
        log_std = self.log_std_head(features)

        # Çıktıları sınırla
        mean = torch.clamp(mean, -1.0, 1.0)
        log_std = torch.clamp(log_std, -5.0, 2.0)

        return mean, log_std

    def forward_with_memory(self, state):
        # Normal forward pass
        mean, log_std = self.forward_actor(state)

        # Bellekten benzer deneyimleri sorgula
        memory_output = self.memory.query_memory(state)
```

```
        memory_state = memory_output[:self.state_size]
        memory_action = memory_output[self.state_size:self.state_size + self.action_size]

        # Bellek entegrasyonu
        state_gate = self.memory_integration['state_gate'](
            torch.cat([state, memory_state], dim=-1)
        )
        action_gate = self.memory_integration['action_gate'](
            torch.cat([mean, memory_action], dim=-1)
        )

        # Gated entegrasyon
        integrated_mean = state_gate * mean + (1 - state_gate) * memory_action

        return integrated_mean, log_std

    def update_memory(self, state, action, reward):
        """Belleği güncelle"""
        self.memory.add_experience(state, action, reward)
```

```python
class LongTermSACAgent(SACAgent):
    def __init__(self, state_size: int, action_size: int):
        super().__init__(state_size, action_size)
        self.sac = LongTermSACNetwork(state_size, action_size).to(self.device)

    def act(self, state):
        """Tek bir durum için eylem seç"""
        try:
            # State'i tensor'a çevir
            if isinstance(state, np.ndarray):
                state = torch.FloatTensor(state).to(self.device)
            if not isinstance(state, torch.Tensor):
                raise ValueError(f"Unexpected state type: {type(state)}")

            with torch.no_grad():
                # Boyut kontrolü ve düzeltmesi
                if state.dim() == 1:
                    state = state.unsqueeze(0)

                # Forward pass ve eylem örnekleme
                action, _ = self.sac.forward_with_memory(state)

                # NaN kontrolü
                if torch.isnan(action).any():
                    print("Uyarı: NaN eylem üretildi")
                    action = torch.zeros_like(action)

                return action.cpu().numpy()[0]

        except Exception as e:
            print(f"Act metodu hatası: {e}")
            return np.random.uniform(-1, 1, self.action_size)

    def remember(self, state, action, reward, next_state, done):
        super().remember(state, action, reward, next_state, done)
        # Belleği güncelle
        self.sac.update_memory(
            torch.FloatTensor(state).to(self.device),
            torch.FloatTensor(action).to(self.device),
            torch.FloatTensor([reward]).to(self.device)
        )

```

```python
class GeneticSACAgent:
    def __init__(self, state_size = 41, action_size = 4, population_size = 48):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.state_size = state_size
        self.action_size = action_size
        self.population_size = population_size
        self.batch_size = 256
        self.gamma = 0.99  # İndirim faktörü
        # Genetik algoritma parametreleri
        self.mutation_rate = 0.1
        self.crossover_rate = 0.8
        self.elite_size = max(2, population_size // 8)
        self.tournament_size = 3
        self.memory = []
        self.memory_size = 100000
        # Popülasyon oluştur
        self.population = [SACAgent(state_size, action_size) for _ in range(population_size)]
        self.fitness_scores = torch.zeros(population_size, device=self.device)
        self.generation = 0
        self.actor = None  # Önce mevcut actor'ü temizle
        if hasattr(self, 'actor'):
            del self.actor
        torch.cuda.empty_cache()  # GPU belleğini temizle
        self.actor = nn.Sequential(
            nn.Linear(state_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, action_size)  # 256 -> 4
        ).to(self.device)

        # Ağırlıkları sıfırla
        for layer in self.actor.modules():
            if isinstance(layer, nn.Linear):
                torch.nn.init.xavier_uniform_(layer.weight)
                torch.nn.init.zeros_(layer.bias)
        self.value_network = nn.Sequential(
            nn.Linear(state_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
```

```
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)
        self.q1_network = nn.Sequential(
            nn.Linear(state_size + action_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)

        self.q2_network = nn.Sequential(
            nn.Linear(state_size + action_size, 512),
            nn.LayerNorm(512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Linear(256, 1)
        ).to(self.device)
        # Q optimizerları ekle
        self.q1_optimizer = optim.Adam(self.q1_network.parameters(), lr=3e-4)
        self.q2_optimizer = optim.Adam(self.q2_network.parameters(), lr=3e-4)
        self.mean_head = nn.Linear(256, action_size).to(self.device)
        self.log_std_head = nn.Linear(256, action_size).to(self.device)
        self.value_optimizer = optim.Adam(self.value_network.parameters(), lr=3e-4)
        self.log_std_min = -20
        self.log_std_max = 2
        # Target entropy için negatif action_size
        self.target_entropy = -action_size
        # Target entropy ve log_alpha ekle
        self.target_entropy = -action_size  # Hedef entropi değeri
        self.log_alpha = nn.Parameter(torch.zeros(1, requires_grad=True, device=self.device))
        self.alpha_optimizer = optim.Adam([self.log_alpha], lr=3e-4)
        self.mean_head = nn.Linear(256, action_size).to(self.device)
        print(f"Actor first layer weight shape: {self.actor[0].weight.shape}")
    def sample_action(self, state):
        """Duruma göre eylem örnekle"""
        try:
```

```
        """Duruma göre eylem örnekle"""
        try:
            mean, log_std = self.forward(state)

            # log_std'yi sınırla
            log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)

            # Normal dağılımdan örnekle
            std = log_std.exp()
            normal = torch.randn_like(mean)

            # Reparametrization trick
            action = mean + std * normal

            # tanh ile sınırla
            action = torch.tanh(action)

            # Log probability hesapla
            log_prob = (-0.5 * ((normal ** 2) + 2 * log_std + np.log(2 * np.pi))).sum(dim=-1)

            # tanh düzeltmesi
            log_prob = log_prob - torch.log(1 - action.pow(2) + 1e-6).sum(dim=-1)

            return action, log_prob.unsqueeze(-1)

        except Exception as e:
            print(f"Sample action hatası: {e}")
            return (torch.zeros_like(mean),
                    torch.zeros((state.size(0), 1), device=self.device))

    def select_parent(self) -> SACAgent:
        tournament = random.sample(list(enumerate(self.population)), self.tournament_size)
        tournament_fitness = [self.fitness_scores[idx] for idx, _ in tournament]
        winner_idx = tournament[torch.argmax(torch.tensor(tournament_fitness))][0]
        return self.population[winner_idx]

    def crossover(self, parent1: SACAgent, parent2: SACAgent) -> SACAgent:
        if random.random() > self.crossover_rate:
            return copy.deepcopy(parent1)

        child = SACAgent(self.state_size, self.action_size)

        for (name1, param1), (name2, param2) in zip(
            parent1.sac.named_parameters(),
            parent2.sac.named_parameters()
```

```
        ):
            if random.random() < 0.5:
                dict(child.sac.named_parameters())[name1].data.copy_(param1.data)
            else:
                dict(child.sac.named_parameters())[name1].data.copy_(param2.data)

        return child

    def mutate(self, agent: SACAgent):
        for param in agent.sac.parameters():
            if random.random() < self.mutation_rate:
                noise = torch.randn_like(param) * 0.1
                param.data += noise
                param.data.clamp_(-1, 1)

    def evolve(self):
        elite_indices = torch.argsort(self.fitness_scores, descending=True)[:self.elite_size]
        new_population = [copy.deepcopy(self.population[idx]) for idx in elite_indices]

        while len(new_population) < self.population_size:
            parent1 = self.select_parent()
            parent2 = self.select_parent()

            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)

        self.population = new_population
        self.generation += 1
        self.fitness_scores = torch.zeros(self.population_size, device=self.device)

    def step_batch(self, actions):
        """Batch halinde adım at"""
        try:
            next_states = []
            rewards = []
            dones = []
            infos = []

            # Her uzay aracı için paralel simülasyon
            for i in range(self.num_envs):
                # NumPy dizisini tensor'a dönüştür
                if isinstance(actions, np.ndarray):
                    action = torch.from_numpy(actions[i]).float().to(self.device)
```

```
                if isinstance(actions, np.ndarray):
                    action = torch.from_numpy(actions[i]).float().to(self.device)
                else:
                    action = actions[i]

                # Her bir ajan için adım at
                with torch.no_grad():
                    if action.dim() == 1:
                        action = action.unsqueeze(0)

                    # Eylemi uygula ve sonuçları al
                    next_state = self.get_state()
                    reward = self._calculate_reward(i)
                    done = self._check_episode_end(i)
                    info = {}

                    next_states.append(next_state[i])
                    rewards.append(reward)
                    dones.append(done)
                    infos.append(info)

            # Sonuçları tensor'a dönüştür
            next_states = torch.stack(next_states).to(self.device)
            rewards = torch.tensor(rewards, device=self.device)
            dones = torch.tensor(dones, device=self.device)

            return next_states, rewards, dones, infos

        except Exception as e:
            print(f"Step batch hatası: {e}")
            return (torch.zeros((self.num_envs, self.state_size), device=self.device),
                    torch.zeros(self.num_envs, device=self.device),
                    torch.ones(self.num_envs, dtype=torch.bool, device=self.device),
                    [{} for _ in range(self.num_envs)])

    def update_fitness(self, agent_idx: int, reward: float):
        self.fitness_scores[agent_idx] += reward

    def remember_batch(self, states, actions, rewards, next_states, dones):
        """Batch halinde deneyimleri belleğe ekle"""
        try:
            # Batch boyutunu al
            batch_size = len(states)

```

```
            # Her bir deneyimi belleğe ekle
            for i in range(batch_size):
                if len(self.memory) >= self.memory_size:
                    self.memory.pop(0)

                self.memory.append((
                    states[i],
                    actions[i],
                    rewards[i],
                    next_states[i],
                    dones[i]
                ))

        except Exception as e:
            print(f"Remember batch hatası: {e}")

    def act_batch(self, states):
        """Batch halinde eylem seç"""
        try:
            if isinstance(states, np.ndarray):
                states = torch.FloatTensor(states).to(self.device)

            # Batch boyutu kontrolü
            if states.dim() == 1:
                states = states.unsqueeze(0)

            with torch.no_grad():
                actions = self.actor(states)
                return actions.cpu().numpy()

        except Exception as e:
            print(f"Act batch hatası: {e}")
            return np.zeros((states.shape[0], self.action_size))

    def act(self, state):
        """Tek bir durum için eylem seç"""
        try:
            if isinstance(state, np.ndarray):
                state = torch.FloatTensor(state).to(self.device)

            if state.dim() == 1:
                state = state.unsqueeze(0)
```

```
                state = state.unsqueeze(0)

            with torch.no_grad():
                action = self.actor(state)
                return action.squeeze(0).cpu().numpy()

        except Exception as e:
            print(f"Act metodu hatası: {e}")
            return np.zeros(self.action_size)

    def forward_value(self, state):
        """Value network için forward pass"""
        return self.value_network(state)

    def forward(self, state):
        """Genişletilmiş ileri yayılım"""
        try:
            # Giriş boyutunu kontrol et ve düzelt
            if isinstance(state, np.ndarray):
                state = torch.FloatTensor(state).to(self.device)

            # Batch boyutu kontrolü
            if state.dim() == 1:
                state = state.unsqueeze(0)

            # Actor özellikleri
            actor_features = self.actor[:-1](state)  # Son katmanı hariç tut
            # Mean ve log_std başlıklarını uygula
            mean = self.mean_head(actor_features)
            log_std = self.log_std_head(actor_features)

            # Çıktıları sınırla ve boyutları kontrol et
            mean = torch.clamp(mean, -1.0, 1.0)
            log_std = torch.clamp(log_std, -5.0, 2.0)

            # Boyut kontrolü ekle
            if mean.dim() == 1:
                mean = mean.unsqueeze(0)
            if log_std.dim() == 1:
                log_std = log_std.unsqueeze(0)

            return mean, log_std
```

```
        except Exception as e:
            print(f"Forward pass hatası: {e}")
            return (torch.zeros((1, self.action_size), device=self.device),
                    torch.full((1, self.action_size), -5.0, device=self.device))

    def forward_critic(self, states, actions, network=None):
        """Critic networks için forward pass"""
        try:
            # Boyut kontrolü ve düzeltmesi
            if states.dim() == 1:
                states = states.unsqueeze(0)
            if actions.dim() == 1:
                actions = actions.unsqueeze(0)

            # Batch boyutlarını eşitle
            if states.size(0) != actions.size(0):
                if states.size(0) == 1:
                    states = states.expand(actions.size(0), -1)
                elif actions.size(0) == 1:
                    actions = actions.expand(states.size(0), -1)

            # State ve action'ları birleştir
            x = torch.cat([states, actions], dim=-1)

            if network is None:
                # Her iki Q değerini hesapla
                q1 = self.q1_network(x)
                q2 = self.q2_network(x)
                return q1, q2
            else:
                # Belirtilen network için Q değerini hesapla
                return network(x)

        except Exception as e:
            print(f"Forward critic hatası: {e}")
            if network is None:
                return (torch.zeros((states.size(0), 1), device=self.device),
                    torch.zeros((states.size(0), 1), device=self.device))
            else:
                return torch.zeros((states.size(0), 1), device=self.device)

```

```python
    def replay(self):
        """Deneyim replay ile öğrenme"""
        if len(self.memory) < self.batch_size:
            return {
                'value_loss': 0.0,
                'q_loss': 0.0,
                'policy_loss': 0.0,
                'alpha_loss': 0.0
            }

        try:
            # Batch örnekleme
            batch = random.sample(self.memory, self.batch_size)

            # Batch verilerini tensor'lara dönüştür ve boyutları düzelt
            states = []
            for x in batch:
                state = x[0]
                if isinstance(state, np.ndarray):
                    state = torch.FloatTensor(state)
                if state.dim() == 1:
                    state = state.unsqueeze(0)
                elif state.size(0) == 48:
                    state = state[0].unsqueeze(0)
                states.append(state.to(self.device))
            states = torch.cat(states, dim=0)  # stack yerine cat kullan

            # Benzer şekilde actions için de boyut düzeltmesi yap
            actions = []
            for x in batch:
                action = x[1]
                if isinstance(action, np.ndarray):
                    action = torch.FloatTensor(action)
                if action.dim() == 1:
                    action = action.unsqueeze(0)
                elif action.size(0) == 48:
                    action = action[0].unsqueeze(0)
                actions.append(action.to(self.device))
            actions = torch.cat(actions, dim=0)

            # Diğer verileri tensor'a dönüştür
            rewards = torch.tensor([x[2] for x in batch], dtype=torch.float32, device=self.device)

```

```
        # Next states için de aynı boyut düzeltmelerini yap
        next_states = []
        for x in batch:
            next_state = x[3]
            if isinstance(next_state, np.ndarray):
                next_state = torch.FloatTensor(next_state)
            if next_state.dim() == 1:
                next_state = next_state.unsqueeze(0)
            elif next_state.size(0) == 48:
                next_state = next_state[0].unsqueeze(0)
            next_states.append(next_state.to(self.device))
        next_states = torch.cat(next_states, dim=0)

        dones = torch.tensor([x[4] for x in batch], dtype=torch.float32, device=self.device)

        with torch.amp.autocast(device_type='cuda' if torch.cuda.is_available() else 'cpu'):
            current_value = self.forward_value(states).view(-1, 1)  # Boyutu [batch_size, 1] yap
            next_value = self.forward_value(next_states).view(-1, 1)
            next_value = next_value * (1 - dones).view(-1, 1)
            target_value = rewards.view(-1, 1) + self.gamma * next_value
            value_loss = F.mse_loss(current_value, target_value.detach())

            # Q losses
            current_q1, current_q2 = self.forward_critic(states, actions)
            q_loss = F.mse_loss(current_q1, target_value.detach()) + \
                     F.mse_loss(current_q2, target_value.detach())

            # Policy loss
            new_actions, log_probs = self.sample_action(states)
            alpha = self.log_alpha.exp()
            q1_new = self.forward_critic(states, new_actions, self.q1_network)
            q2_new = self.forward_critic(states, new_actions, self.q2_network)
            q_new = torch.min(q1_new, q2_new)
            policy_loss = (alpha * log_probs - q_new).mean()
            # Alpha hesaplama
            alpha = self.log_alpha.exp()

            # Policy loss
            new_actions, log_probs = self.sample_action(states)
            q1_new = self.forward_critic(states, new_actions, self.q1_network)
            q2_new = self.forward_critic(states, new_actions, self.q2_network)
            q_new = torch.min(q1_new, q2_new)
            policy_loss = (alpha * log_probs - q_new).mean()
```

```
            # Alpha loss
            alpha_loss = -(self.log_alpha * (log_probs + self.target_entropy).detach()).mean()

            # Alpha optimization
            self.alpha_optimizer.zero_grad()
            alpha_loss.backward(retain_graph=True)
            self.alpha_optimizer.step()

        return {
            'value_loss': float(value_loss.detach().cpu().item()),
            'q_loss': float(q_loss.detach().cpu().item()),
            'policy_loss': float(policy_loss.detach().cpu().item()),
            'alpha_loss': float(alpha_loss.detach().cpu().item())
        }

    except Exception as e:
        print(f"Replay hatası detayı: {e}")
        traceback.print_exc()
        return {
            'value_loss': 0.0,
            'q_loss': 0.0,
            'policy_loss': 0.0,
            'alpha_loss': 0.0
        }

def save_population(self, path: str):
    """Tüm popülasyonu kaydet"""
    try:
        population_states = []
        for i, agent in enumerate(self.population):
            if hasattr(agent, 'sac'):
                population_states.append({
                    'agent_idx': i,
                    'sac_state_dict': agent.sac.state_dict(),
                    'fitness': float(self.fitness_scores[i])
                })
        torch.save(population_states, path)
    except Exception as e:
        print(f"Popülasyon kaydetme hatası: {e}")
```

```python
class LongTermGeneticSACAgent(GeneticSACAgent):
    def __init__(self, state_size: int, action_size: int, population_size: int = 48):
        super().__init__(state_size, action_size, population_size)
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.state_size = state_size
        self.action_size = action_size
        self.population_size = population_size

        # Popülasyonu LongTermSACAgent'lardan oluştur
        self.population = [
            LongTermSACAgent(state_size, action_size) for _ in range(population_size)
        ]

        # Diğer parametreler aynı
        self.elite_size = 12
        self.mutation_rate = 0.1
        self.crossover_rate = 0.8
        self.fitness_scores = torch.zeros(population_size, device=self.device)

    def crossover(self, parent1: LongTermSACAgent, parent2: LongTermSACAgent) -> LongTermSACAgent:
        """Bellek mekanizmasını da içeren çaprazlama"""
        child = super().crossover(parent1, parent2)

        # Bellek entegrasyonu için ek parametreleri çaprazla
        for (name1, param1), (name2, param2) in zip(
            parent1.sac.memory_integration.named_parameters(),
            parent2.sac.memory_integration.named_parameters()
        ):
            if random.random() < self.crossover_rate:
                mask = torch.rand_like(param1) < 0.5
                new_param = torch.where(mask, param1, param2)
                dict(child.sac.memory_integration.named_parameters())[name1].data.copy_(new_param)

        return child
```

```python
class SpaceTravelEnv:
    def __init__(self, destination_planet_name: str, num_envs: int = 1):
        """
        Uzay yolculuğu simülasyon ortamını başlat

        Args:
            destination_planet_name: Hedef gezegenin adı
            num_envs: Paralel simülasyon sayısı (varsayılan: 1)
        """
        # Önce cihaz ve fizik sabitlerini tanımla
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.num_envs = num_envs
        self.time_step = 86400.0
        # Zaman adımını tanımla - 1 gün (saniye cinsinden)
        self.dt = self.time_step

        # Fizik sabitleri - En başta tanımlanmalı
        self.G = torch.tensor(G, device=self.device, dtype=torch.float32)  # Gravitasyon sabiti
        self.sun_mass = torch.tensor(SUN_MASS, device=self.device, dtype=torch.float32)  # Güneş kütlesi

        # Orbital parametreleri tanımla
        self.orbital_parameters = {
            'Mercury': {
                'semi_major_axis': torch.tensor(0.387 * AU, device=self.device),
                'eccentricity': torch.tensor(0.206, device=self.device),
                'orbital_period': torch.tensor(87.97 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(0.0, device=self.device)
            },
            'Venus': {
                'semi_major_axis': torch.tensor(0.723 * AU, device=self.device),
                'eccentricity': torch.tensor(0.007, device=self.device),
                'orbital_period': torch.tensor(224.7 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(torch.pi / 4, device=self.device)
            },
            'Earth': {
                'semi_major_axis': torch.tensor(1.0 * AU, device=self.device),
                'eccentricity': torch.tensor(0.017, device=self.device),
                'orbital_period': torch.tensor(365.25 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(torch.pi / 2, device=self.device)
            },
            'Mars': {
                'semi_major_axis': torch.tensor(1.524 * AU, device=self.device),
                'eccentricity': torch.tensor(0.093, device=self.device),
                'orbital_period': torch.tensor(687.0 *24 *3600, device=self.device),
                'initial_angle': torch.tensor(3 * torch.pi / 4, device=self.device)
```

```
            'Jupiter': {
                'semi_major_axis': torch.tensor(5.203 * AU, device=self.device),
                'eccentricity': torch.tensor(0.048, device=self.device),
                'orbital_period': torch.tensor(4331.0 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(torch.pi, device=self.device)
            },
            'Saturn': {
                'semi_major_axis': torch.tensor(9.537 * AU, device=self.device),
                'eccentricity': torch.tensor(0.054, device=self.device),
                'orbital_period': torch.tensor(10747.0 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(5 * torch.pi / 4, device=self.device)
            },
            'Uranus': {
                'semi_major_axis': torch.tensor(19.191 * AU, device=self.device),
                'eccentricity': torch.tensor(0.047, device=self.device),
                'orbital_period': torch.tensor(30589.0 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(3 * torch.pi / 2, device=self.device)
            },
            'Neptune': {
                'semi_major_axis': torch.tensor(30.069 * AU, device=self.device),
                'eccentricity': torch.tensor(0.009, device=self.device),
                'orbital_period': torch.tensor(59800.0 * 24 * 3600, device=self.device),
                'initial_angle': torch.tensor(7 * torch.pi / 4, device=self.device)
            }
        }

        # Fizik motoru
        self.physics_engine = RelativisticSpacePhysics(device=self.device)

        # Gezegenleri oluştur
        self.planets = self._create_planets()
        self.destination = self.planets[destination_planet_name]

        # Uzay araçlarını oluştur
        self.spacecraft = [Spacecraft(1000.0) for _ in range(num_envs)]

        # Başlangıç durumunu ayarla
        self._initialize_spacecraft()

        # Durum ve eylem boyutlarını hesapla
        self._state_size = self._calculate_state_size()
        self.state_size = self._state_size  # Dışarıdan erişim için
        self.action_size = 4  # [thrust_x, thrust_y, thrust_magnitude, correction]

```

```python
        # Simülasyon zamanı ve durum
        self.time = torch.zeros(num_envs, device=self.device)
        self.done = torch.zeros(num_envs, dtype=torch.bool, device=self.device)

        # Geçmiş kayıtları
        self.fuel_history = []
        self.distance_history = []
        self.speed_history = []
        self.time_history = []
        self.acceleration_history = []

        # İlk mesafeyi kaydet
        self.initial_distance = torch.norm(
            self.spacecraft[0].position - self.destination.position
        )
        self.previous_distance = torch.full(
            (num_envs,), float('inf'), device=self.device
        )

        # Adım sayacını ekle
        self.current_step = torch.zeros(num_envs, device=self.device)
        self.max_steps = 1000  # Maksimum adım sayısı
        # Popülasyon zelliğini ekle
        self.population = self.spacecraft  # spacecraft listesini population olarak kullan

    def _calculate_state_size(self) -> int:
        """Durum vektörünün boyutunu hesapla"""
        # Basitleştirilmiş state boyutu
        spacecraft_state = {
            'position': 2,          # x, y pozisyonu
            'velocity': 2,          # x, y hızı
            'fuel': 1,             # yakıt miktarı
        }

        # Hedef gezegen durumu
        target_state = {
            'position': 2,          # x, y pozisyonu
            'velocity': 2,          # x, y hızı
            'distance': 1,          # uzay aracına olan mesafe
        }

        # Genel metrikler
        general_metrics = {
            'mission_time': 1,      # görev süresi
        }
```

```python
        # Toplam boyutu hesapla
        total_size = (sum(spacecraft_state.values()) +
                      sum(target_state.values()) +
                      sum(general_metrics.values()))

        return total_size

    def get_state(self):
        """Güncel durumu döndür - optimize edilmiş versiyon"""
        try:
            # Uzay araçlarının durumlarını tek seferde hesapla
            positions = torch.stack([s.position for s in self.spacecraft])
            velocities = torch.stack([s.velocity for s in self.spacecraft])
            fuel_masses = torch.tensor([[s.fuel_mass] for s in self.spacecraft], device=self.device)

            # Uzay aracı durumlarını birleştir
            spacecraft_states = torch.cat([positions, velocities, fuel_masses], dim=1)

            # Hedef gezegen durumunu tekrarla
            target_pos_vel = torch.cat([
                self.destination.position,
                self.destination.velocity
            ]).repeat(self.num_envs, 1)

            # Zamanı ekle
            time_tensor = self.time.view(-1, 1)

            # Tüm durumları birleştir
            states = torch.cat([
                spacecraft_states,  # [batch_size, 5]
                target_pos_vel,     # [batch_size, 4]
                time_tensor         # [batch_size, 1]
            ], dim=1)

            return states

        except Exception as e:
            print(f"State hesaplama hatası: {e}")
            return torch.zeros((self.num_envs, self._state_size), device=self.device)

    def step_batch(self, actions):
        """Batch halinde adım at"""
        try:
```

```
            rewards = []
            dones = []
            infos = []

            # Her uzay aracı için simülasyon
            for i in range(len(actions)):
                # Aksiyonu tensor'a çevir ve doğru cihaza taşı
                if isinstance(actions[i], np.ndarray):
                    action = torch.from_numpy(actions[i]).float().to(self.device)
                else:
                    action = actions[i].to(self.device)

                # Uzay aracını güncelle
                self.spacecraft[i].apply_action(action)
                self.spacecraft[i].update(self.dt)  # time_step yerine dt kullan

                # Yeni durumu al
                next_state = self.get_state()[i]

                # Ödülü hesapla
                reward = float(self._calculate_reward(i))  # float'a çevir

                # Episode'un bitip bitmediğini kontrol et
                done = bool(self._check_done(i))  # bool'a çevir

                # Sonuçları listeye ekle
                next_states.append(next_state)
                rewards.append(reward)
                dones.append(done)
                infos.append({})

            # Sonuçları tensor'a çevir
            next_states = torch.stack(next_states)
            rewards = torch.tensor(rewards, device=self.device)
            dones = torch.tensor(dones, device=self.device)

            return next_states, rewards, dones, infos

        except Exception as e:
            print(f"Step batch hatası: {e}")
            traceback.print_exc()
            return (torch.zeros((len(actions), self.state_size), device=self.device),
                    torch.zeros(len(actions), device=self.device),
                    torch.ones(len(actions), dtype=torch.bool, device=self.device),
```

```
                    torch.ones(len(actions), dtype=torch.bool, device=self.device),
                    [{} for _ in range(len(actions))])

    def get_state_tensor(self):
        """Mevcut durumu tensor olarak al"""
        # Uzay aracı durumu
        spacecraft_state = torch.cat([
            self.spacecraft.position,
            self.spacecraft.velocity,
            torch.tensor([self.spacecraft.fuel_mass], device=self.device)
        ]).unsqueeze(0)  # [1, 5] boyutuna dönüştür

        # Gezegen durumları
        planet_states = []
        for planet in self.planets:
            planet_state = torch.cat([
                planet.position,
                planet.velocity
            ]).unsqueeze(0)  # [1, 4] boyutuna dönüştür
            planet_states.append(planet_state)

        # Tüm gezegen durumlarını birleştir
        all_planet_states = torch.cat(planet_states, dim=0) if planet_states else torch.zeros((0, 4), device=self.device)

        # Tüm durumları birleştir
        state = torch.cat([spacecraft_state, all_planet_states.reshape(-1).unsqueeze(0)], dim=1)
        print(f"Hesaplanan state boyutu: {state}")
        return state.squeeze(0)  # Final boyut: [state_size]

    def update_planet_position(self, planet, dt):
        """Gezegen pozisyonunu Kepler yasalarına göre güncelle"""
        if isinstance(planet, str):
            planet = self.planets[planet]
            # dt'yi kullanarak pozisyon güncelleme işlemleri
            current_time = self.time + dt
        params = self.orbital_parameters[planet.name]

        # Mevcut zaman açısını hesapla
        current_time = self.time % params['orbital_period']
        mean_motion = 2 * np.pi / params['orbital_period']

        # Batch boyutunu kontrol et ve mean_anomaly'yi uygun şekilde hesapla
        if isinstance(self.time, torch.Tensor) and self.time.dim() > 0:
```

```
        # Batch boyutunu kontrol et ve mean_anomaly'yi uygun şekilde hesapla
        if isinstance(self.time, torch.Tensor) and self.time.dim() > 0:
            # Batch durumu
            mean_anomaly = mean_motion * current_time + params['initial_angle']
            mean_anomaly = mean_anomaly.to(device=self.device, dtype=torch.float32)
        else:
            # Tekil durum
            mean_anomaly = torch.tensor(mean_motion * current_time + params['initialangle'],
                                        device=self.device, dtype=torch.float32)

        # Eksantrik anomaliyi hesapla (Newton-Raphson metodu)
        e = torch.tensor(params['eccentricity'], device=self.device, dtype=torch.float32)
        E = mean_anomaly.clone()
        for _ in range(5):  # 5 iterasyon genellikle yeterli
            E = E - (E - e * torch.sin(E) - mean_anomaly) / (1 - e * torch.cos(E))

        # Gerçek anomaliyi hesapla
        true_anomaly = 2 * torch.atan(torch.sqrt((1 + e)/(1 - e)) * torch.tan(E/2))

        # Yarı-major eksen
        a = torch.tensor(params['semi_major_axis'], device=self.device, dtype=torch.float32)

        # Radyal mesafe
        r = a * (1 - e * torch.cos(E))

        # Pozisyon ve hız hesaplamaları için boyut kontrolü
        if r.dim() > 0:
            # Batch durumu - pozisyonları güncelle
            planet.position = torch.stack([
                r * torch.cos(true_anomaly),
                r * torch.sin(true_anomaly)
            ], dim=-1)
        else:
            # Tekil durum
            planet.position[0] = r * torch.cos(true_anomaly)
            planet.position[1] = r * torch.sin(true_anomaly)

        # Yörüngesel hız
        p = a * (1 - e*e)  # yarı-latus rectum
        velocity_magnitude = torch.sqrt(self.G * self.sun_mass * (2/r - 1/a))

        # Hız vektörü - boyut kontrolü ile
        if velocity_magnitude.dim() > 0:
            # Batch durumu
```

```
        if velocity_magnitude.dim() > 0:
            # Batch durumu
            planet.velocity = torch.stack([
                -velocity_magnitude * torch.sin(true_anomaly),
                velocity_magnitude * torch.cos(true_anomaly) * (1 + e * torch.cos(true_anomaly))
            ], dim=-1)
        else:
            # Tekil durum
            planet.velocity[0] = -velocity_magnitude * torch.sin(true_anomaly)
            planet.velocity[1] = velocity_magnitude * torch.cos(true_anomaly) * (1 + e * torch.cos(true_anomaly))

        # Hızı normalize et
        planet.velocity *= torch.sqrt(self.G * self.sun_mass / r)

        return planet.position, planet.velocity

    def _create_planets(self) -> Dict[str, Planet]:
        """Gezegenleri oluştur ve başlangıç konumlarını ayarla"""
        planets = {}
        for name, data in PLANETARY_DATA.items():
            planet = Planet(name, data['mass'], data['orbit_radius'])
            if name in self.orbital_parameters:
                params = self.orbital_parameters[name]

                # Başlangıç açısını tensor olarak bir kez oluştur
                initial_angle = params['initial_angle'].clone().detach()

                # Trigonometrik hesaplamaları bir kez yap
                cos_angle = torch.cos(initial_angle)
                sin_angle = torch.sin(initial_angle)

                # Yarıçapı hesapla
                r = params['semi_major_axis'] * (1 - params['eccentricity']**2) / \
                    (1 + params['eccentricity'] * cos_angle)

                # Pozisyon vektörünü oluştur
                planet.position = torch.stack([
                    r * cos_angle,
                    r * sin_angle
                ]).to(self.device)

                # Hız büyüklüğünü hesapla
                velocity_magnitude = torch.sqrt(self.G * self.sun_mass *
                                                (2/r - 1/params['semi_major_axis']))
```

```
3032
3033                # Hız vektörünü oluştur
3034                planet.velocity = torch.stack([
3035                    -velocity_magnitude * sin_angle,
3036                    velocity_magnitude * cos_angle
3037                ]).to(self.device)
3038
3039            planets[name] = planet
3040
3041        return planets
3042
3043    def get_state(self):
3044        """Güncel durumu döndür - optimize edilmiş versiyon"""
3045        try:
3046            # Tüm durumları tek seferde hesapla
3047            spacecraft_states = torch.stack([
3048                torch.cat([
3049                    s.position,
3050                    s.velocity,
3051                    torch.tensor([s.fuel_mass], device=self.device)
3052                ]) for s in self.spacecraft
3053            ])
3054
3055            # Hedef gezegen durumu - tek seferde
3056            target_state = torch.cat([
3057                self.destination.position,
3058                self.destination.velocity
3059            ]).repeat(self.num_envs, 1)
3060
3061            # Zaman ve diğer metrikler - tek seferde
3062            other_metrics = torch.stack([
3063                self.time,
3064                torch.zeros_like(self.time)  # veya başka bir metrik
3065            ], dim=1)
3066
3067            # Tüm durumları birleştir
3068            states = torch.cat([
3069                spacecraft_states,
3070                target_state,
3071                other_metrics
3072            ], dim=1)
3073
```

```
3075
3076            except Exception as e:
3077                print(f"State hesaplama hatası: {e}")
3078                return torch.zeros((self.num_envs, self._state_size), device=self.device)
3079
3080        def _check_collision(self, env_idx: int) -> bool:
3081            """Çarpışma kontrolü - optimize edilmiş versiyon"""
3082            try:
3083                # Uzay aracının pozisyonunu al
3084                spacecraft_pos = self.spacecraft[env_idx].position
3085
3086                # Güneş'le çarpışma kontrolü
3087                sun_distance = torch.norm(spacecraft_pos)
3088                if sun_distance < MIN_SAFE_DISTANCE_SUN:
3089                    return True
3090
3091                # Tüm gezegenlerin pozisyonlarını tek seferde al
3092                planet_positions = torch.stack([p.position for p in self.planets.values()])
3093                planet_radii = torch.tensor([p.radius for p in self.planets.values()], device=self.device)
3094
3095                # Mesafeleri tek seferde hesapla
3096                distances = torch.norm(planet_positions - spacecraft_pos.unsqueeze(0), dim=1)
3097
3098                # Çarpışma kontrolü
3099                return bool(torch.any(distances <= planet_radii * 1.5))  # 1.5 güvenlik faktörü
3100
3101            except Exception as e:
3102                print(f"Çarpışma kontrolü hatası: {e}")
3103                return False
3104
3105        def reset(self):
3106            try:
3107                # CPU'da başlangıç değerlerini ayarla
3108                earth = self.planets['Earth']
3109                earth_pos = earth.position.clone().cpu()
3110                earth_vel = earth.velocity.clone().cpu()
3111
3112                for s in self.spacecraft:
3113                    # CPU'da değerleri ayarla
3114                    s.position = earth_pos.clone()
3115                    s.velocity = earth_vel.clone()
3116                    s.fuel_mass = torch.tensor(1000.0, device='cpu')
3117                    s.hull_integrity = torch.tensor(1.0, device='cpu')
3118                    s.engine_temperature = torch.tensor(293.15, device='cpu')
```

```
                s.has_fuel_leak = False
                s.backup_engines = 3

                # GPU'ya taşı
                s.position = s.position.to(self.device)
                s.velocity = s.velocity.to(self.device)
                s.fuel_mass = s.fuel_mass.to(self.device)
                s.hull_integrity = s.hull_integrity.to(self.device)
                s.engine_temperature = s.engine_temperature.to(self.device)

            # Initial distance'ı ayarla
            self.initial_distance = torch.norm(
                self.spacecraft[0].position - self.destination.position
            )

            # Previous distance'ı başlat
            self.previous_distance = torch.full(
                (self.num_envs,), float('inf'), device=self.device
            )

            # Zamanı sıfırla
            self.time = torch.zeros(self.num_envs, device=self.device)
            self.done = torch.zeros(self.num_envs, dtype=torch.bool, device=self.device)

            # State'i döndür
            state = self.get_state()
            return state.cpu().numpy()

        except Exception as e:
            print(f"Reset hatası: {str(e)}")
            torch.cuda.empty_cache()
            return np.zeros(self.state_size)

    def reset_batch(self, batch_size):
        """Batch halinde çevre sıfırlama"""
        try:
            states = []
            for _ in range(batch_size):
                # Her bir ortam için reset çağır
                state = self.reset()

                # numpy array'i tensor'a çevir
```

```
            # numpy array'i tensor'a çevir
            if isinstance(state, np.ndarray):
                state = torch.from_numpy(state).float()

            # GPU'ya taşı
            state = state.to(self.device)
            states.append(state)

        # Tüm state'leri birleştir
        return torch.stack(states)

    except Exception as e:
        print(f"Reset batch hatası: {e}")
        # Hata durumunda sıfır tensor'u dön
        return torch.zeros((batch_size, self.state_size), device=self.device)

def step(self, action):
    try:
        # NumPy dizisini tensor'a dönüştür
        if isinstance(action, np.ndarray):
            action = torch.from_numpy(action).float().to(self.device)
        elif not isinstance(action, torch.Tensor):
            action = torch.tensor(action, dtype=torch.float32, device=self.device)

        # Eylemi normalize et
        normalized_action = torch.clamp(action, -1, 1)

        # İtki kuvvetini hesapla
        thrust_direction = normalized_action[:2] if normalized_action.dim() == 1 else normalized_action[:, :2]
        thrust_magnitude = torch.abs(normalized_action[2]) if normalized_action.dim() == 1 else torch.abs(normalized_action[:, 2])

        # Yön vektörünü normalize et
        thrust_direction = thrust_direction / (torch.norm(thrust_direction, dim=-1, keepdim=True) + 1e-8)

        # Tüm uzay araçları için döngü
        for i in range(self.num_envs):
            # İtki kuvvetini hesapla
            thrust_force = thrust_magnitude[i] * MAX_THRUST * thrust_direction[i] if thrust_direction.dim() > 1 else thrust_magnitude * MAX_THRUST * thrust_direction

            # Toplam kuvveti hesapla
            total_force = torch.zeros_like(thrust_force)
```

```python
                # Gezegenlerin çekim kuvvetlerini ekle
                for _, planet in self.planets.items():
                    r = planet.position - self.spacecraft[i].position
                    r_mag = torch.norm(r)
                    if r_mag > 0:
                        force = G * planet.mass * self.spacecraft[i].mass * r / (r_mag ** 3)
                        if force.shape != thrust_force.shape:
                            force = force[:2] if force.dim() == 1 else force[:, :2]
                        total_force = total_force + force

                # İtki kuvvetini ekle
                total_force = total_force + thrust_force

                # Uzay aracını güncelle
                self.spacecraft[i].apply_force(total_force)
                self.spacecraft[i].update(self.dt)

            # Gezegenleri güncelle
            for _, planet in self.planets.items():
                planet.update_position_rk45(self.dt)

            # Yeni durumu al
            next_state = self.get_state()

            # Ödülü hesapla
            reward = self._calculate_reward()

            # Bölümün bitip bitmediğini kontrol et
            done = self._check_episode_end()

            return next_state, reward, done, {}

        except Exception as e:
            print(f"Simülasyon hatası: {e}")
            traceback.print_exc()
            return (torch.zeros((self.num_envs, self._state_size), device=self.device),
                    torch.tensor(0.0, device=self.device),
                    torch.tensor(True, device=self.device),
                    {})

    def _check_emergencies(self, time_step: float) -> Dict[str, bool | float]:
        """Acil durumları kontrol et"""
        result = {'critical_failure': False, 'penalty': 0.0}
```

```
3245
3246        # Motor arızası kontrolü
3247        if random.random() < MOTOR_FAILURE_PROBABILITY * (time_step / 86400):
3248            if not self.spacecraft.handle_motor_failure():
3249                result['critical_failure'] = True
3250                result['penalty'] = -10000
3251                return result
3252
3253        # Yakıt sızıntısı kontrolü
3254        if random.random() < FUEL_LEAK_PROBABILITY * (time_step / 86400):
3255            if not self.spacecraft.handle_fuel_leak():
3256                result['critical_failure'] = True
3257                result['penalty'] = -5000
3258                return result
3259
3260        # Mikroasteroit çarpışması kontrolü
3261        if random.random() < MICROASTEROID_COLLISION_PROBABILITY * (time_step / 86400):
3262            if not self.spacecraft.handle_microasteroid_collision():
3263                result['critical_failure'] = True
3264                result['penalty'] = -8000
3265                return result
3266
3267        return result
3268
3269    def _calculate_reward(self, spacecraft_index=None):
3270        """Ödül hesaplama"""
3271        try:
3272            # Uzay aracını belirle
3273            if spacecraft_index is None:
3274                spacecraft = self.spacecraft[0]  # Tek uzay aracı durumu
3275            else:
3276                spacecraft = self.spacecraft[spacecraft_index]
3277                # Mesafe bazlı ödül
3278            current_distance = torch.norm(spacecraft.position - self.destination.position)
3279            distance_reward = -current_distance / AU  # AU'ya göre normalize et
3280
3281            # Hız bazlı ödül
3282            velocity = torch.norm(spacecraft.velocity)
3283            optimal_velocity = 30000  # Optimal hız (m/s)
3284            velocity_reward = -abs(velocity - optimal_velocity) / optimal_velocity
3285
3286            # Yakıt kullanımı bazlı ödül
3287            fuel_efficiency = spacecraft.fuel_mass / self.max_fuel
```

```
            # Yakıt kullanımı bazlı ödül
            fuel_efficiency = spacecraft.fuel_mass / self.max_fuel
            fuel_reward = fuel_efficiency * 0.5  # Yakıt tasarrufu için ödül

            # Çarpışma cezası
            collision_penalty = 0.0
            if self._check_collision(spacecraft_index if spacecraft_index is not None else 0):
                collision_penalty = -100.0

            # Başarı ödülü
            success_reward = 0.0
            if current_distance < SUCCESS_DISTANCE:
                success_reward = 100.0

            # Tüm ödülleri birleştir
            reward_dict = {
                'distance_reward': float(distance_reward),
                'velocity_reward': float(velocity_reward),
                'fuel_reward': float(fuel_reward),
                'collision_penalty': float(collision_penalty),
                'success_reward': float(success_reward)
            }

            # Toplam ödülü hesapla
            total_reward = sum(reward_dict.values())

            # Ödülü sınırla
            total_reward = max(min(total_reward, 100.0), -100.0)

            return total_reward
        except Exception as e:
            print(f"Ödül hesaplama hatası: {e}")
            traceback.print_exc()
            return 0.0  # Hata durumunda varsayılan değer

    def _check_done(self, spacecraft_index):
        """Episode'un bitip bitmediğini kontrol et"""
        try:
            spacecraft = self.spacecraft[spacecraft_index]

            # Hedef gezegene ulaşma kontrolü
            distance_to_target = torch.norm(spacecraft.position - self.destination.position)
            if float(distance_to_target) < SUCCESS_DISTANCE:  # tensor'ı float'a çevir
                return True
```

```
3330
3331            # Yakıt bitti mi?
3332            if float(spacecraft.fuel_mass) <= 0:  # tensor'ı float'a çevir
3333                return True
3334
3335            # Çarpışma kontrolü
3336            if self._check_collision(spacecraft_index):
3337                return True
3338
3339            # Maksimum adım sayısı kontrolü
3340            if float(self.current_step[spacecraft_index]) >= self.max_steps:  # tensor'ı float'a çevir
3341                return True
3342
3343            return False
3344
3345        except Exception as e:
3346            print(f"Done kontrolü hatası: {e}")
3347            return True  # Hata durumunda episode'u bitir
3348
3349    def adaptive_time_step(self, min_dt=(86400/2), max_dt=86400*15):  # min 1 gün, max 15 gün
3350        """Adaptif zaman adımı hesaplama"""
3351        min_period = float('inf')
3352        for planet in self.planets.values():
3353            if planet.name in self.orbital_parameters:
3354                period = self.orbital_parameters[planet.name]['orbital_period']
3355                min_period = min(min_period, period)
3356
3357        # Daha büyük zaman adımları kullan
3358        dt = min(max(min_period / 50, min_dt), max_dt)
3359
3360        # Hedef gezegene yakınken daha küçük adımlar kullan
3361        distance_to_destination = torch.norm(
3362            torch.stack([s.position for s in self.spacecraft]) -
3363            self.destination.position, dim=1
3364        )
3365        if torch.any(distance_to_destination < AU):
3366            dt = min(dt, 86400)  # 1 gün
3367
3368        return dt
3369
3370    def _initialize_spacecraft(self):
3371        """Uzay aracını başlangıç konumuna yerleştir"""
3372        earth = self.planets['Earth']
3373        for s in self.spacecraft:
```

```
            s.velocity = earth.velocity.clone()
        self.max_fuel = self._calculate_minimum_fuel()
        for s in self.spacecraft:
            s.fuel_mass = self.max_fuel

    def _calculate_minimum_fuel(self) -> float:
        """Minimum yakıt miktarını hesapla"""
        r1 = torch.norm(self.planets['Earth'].position)
        r2 = torch.norm(self.destination.position)
        mu = self.G * self.sun_mass

        # Hohmann transfer için delta-v hesabı
        delta_v1 = torch.sqrt(mu / r1) * (torch.sqrt(2 * r2 / (r1 + r2)) - 1)

        delta_v2 = torch.sqrt(mu / r2) * (1 - torch.sqrt(2 * r1 / (r1 + r2)))

        delta_v = delta_v1 + delta_v2

        m0 = self.spacecraft[0].dry_mass * 40
        mf = m0 / torch.exp(delta_v / (ISP * g0))
        return float(torch.max(torch.tensor(0.0), m0 - mf))

    def _update_history(self):
        """Geçmiş kayıtlarını güncelle"""
        self.fuel_history.append(torch.tensor([s.fuel_mass for s in self.spacecraft], device=self.device))
        self.distance_history.append(torch.norm(
            torch.stack([s.position for s in self.spacecraft]) -
            self.planets['Earth'].position, dim=1
        ))
        self.speed_history.append(torch.norm(torch.stack([s.velocity for s in self.spacecraft]), dim=1))
        self.time_history.append(self.time.clone())
        self.acceleration_history.append(
            torch.norm(torch.stack([s.acceleration for s in self.spacecraft]), dim=1))

    def _check_episode_end(self):
        """Bölümün bitip bitmediğini kontrol et"""
        # Her simülasyon için kontrol yap
        dones = torch.zeros(self.num_envs, dtype=torch.bool, device=self.device)

        for i in range(self.num_envs):
            # Güneş'e çarpma kontrolü
            sun_distance = torch.norm(self.spacecraft[i].position)
            if sun_distance < MIN_SAFE_DISTANCE_SUN * 0.5:
                dones[i] = True
```

```
3420            # Hedef gezegene varış kontrolü
3421            distance_to_destination = torch.norm(
3422                self.spacecraft[i].position - self.destination.position
3423            )
3424            if distance_to_destination < SUCCESS_DISTANCE:
3425                dones[i] = True
3426
3427            # Yakıt bitti mi kontrolü
3428            if self.spacecraft[i].fuel_mass <= 0:
3429                dones[i] = True
3430
3431            # Zaman aşımı kontrolü
3432            if self.time[i] >= MAX_EPISODE_TIME:
3433                dones[i] = True
3434
3435            # Kritik hasar kontrolü
3436            if self.spacecraft[i].hull_damage >= self.spacecraft[i].critical_damage_threshold:
3437                dones[i] = True
3438
3439        return dones
3440
```

```python
class TrainingTimeEstimator:
    def __init__(self, num_episodes: int):
        self.num_episodes = num_episodes
        self.start_time = None
        self.completed_episodes = 0
        self.episode_times = deque(maxlen=100)  # Son 100 bölümün sürelerini tut

    def start(self):
        """Eğitim zamanlayıcısını başlat"""
        self.start_time = time.time()
        self.completed_episodes = 0

    def update(self, episode: int) -> dict:
        """Her bölüm sonunda süre tahminini güncelle"""
        current_time = time.time()
        if self.start_time is None:
            self.start(episode)
            return {}

        self.completed_episodes = episode + 1
        episode_time = (current_time - self.start_time) / self.completed_episodes
        self.episode_times.append(episode_time)

        # Ortalama bölüm süresini hesapla
        avg_episode_time = sum(self.episode_times) / len(self.episode_times)

        # Kalan süreyi hesapla
        remaining_episodes = self.num_episodes - self.completed_episodes
        estimated_remaining_time = remaining_episodes * avg_episode_time

        # Toplam süreyi hesapla
        total_time = (current_time - self.start_time) + estimated_remaining_time

        # GPU kullanım bilgisini al
        gpu_usage = torch.cuda.memory_allocated() / torch.cuda.max_memory_allocated() if torch.cuda.is_available() else 0

        return {
            'elapsed_time': str(timedelta(seconds=int(current_time - self.start_time))),
            'estimated_remaining': str(timedelta(seconds=int(estimated_remaining_time))),
            'total_estimated_time': str(timedelta(seconds=int(total_time))),
            'progress_percentage': (self.completed_episodes / self.num_episodes) * 100,
            'gpu_usage': f"{gpu_usage * 100:.1f}%"
        }
```

```python
def train_agent(destination_planet: str, num_episodes: int):
    try:
        # Shared data sözlüğünü başlangıçta tanımla
        shared_data = {
            'running': True,
            'current_metrics': {},
            'current_positions': {}
        }

        # Ortam ve genetik SAC ajanını oluştur
        env = SpaceTravelEnv(destination_planet, num_envs=48)
        agent = GeneticSACAgent(env.state_size, env.action_size, population_size=48)
        # Metrikleri kaydet bölümüne ekleyin:
        best_rewards.append(max(episode_rewards) if episode_rewards else 0.0)
        # Eğitim metrikleri
        total_rewards = []
        episode_losses = []
        best_rewards = []
        success_rates = []
        min_distances = []
        fuel_consumptions = []
        step_counts = []

        # Zaman tahmincisini başlat
        time_estimator = TrainingTimeEstimator(num_episodes)
        time_estimator.start()

        for episode in range(num_episodes):
            states = env.reset_batch(agent.population_size)  # Tüm popülasyonu sıfırla
            episode_rewards = []
            episode_steps = []
            episode_distances = []
            episode_fuels = []

            for step in range(MAX_STEPS):
                try:
                    # Batch halinde eylemler al
                    actions = agent.act_batch(states)

                    # Tüm popülasyon için adım at
                    next_states, rewards, dones, _ = env.step_batch(actions)

                    # Deneyimleri kaydet ve öğren
                    agent.remember_batch(states, actions, rewards, next_states, dones)
                    loss = agent.replay()
```

```
                # Metrikleri kaydet
                episode_rewards.extend(rewards.cpu().numpy())
                episode_steps.append(step + 1)

                # Mesafe ve yakıt tüketimini kaydet
                for i in range(len(env.spacecraft)):
                    distance = torch.norm(env.spacecraft[i].position - env.destination.position)
                    episode_distances.append(float(distance) / AU)
                    episode_fuels.append(float(env.spacecraft[i].fuel_mass))

                states = next_states

                # Tüm popülasyon üyeleri bittiyse döngüden çık
                if all(dones):
                    break

            except Exception as e:
                print(f"Step hatası: {e}")
                continue

        try:
            # Bölüm metriklerini hesapla
            avg_reward = np.mean(episode_rewards)
            avg_steps = np.mean(episode_steps)
            min_distance = min(episode_distances)
            avg_fuel = np.mean(episode_fuels)
            success_rate = sum(1 for r in episode_rewards if r > 0) / len(episode_rewards) * 100

            # Metrikleri kaydet
            total_rewards.append(avg_reward)
            if loss is not None and isinstance(loss, dict):
                loss = loss.get('value_loss', 0.0)  # Doğrudan loss'u güncelle
            else:
                loss = float(loss) if loss is not None else 0.0

            episode_losses.append(loss)
            step_counts.append(avg_steps)
            min_distances.append(min_distance)
            fuel_consumptions.append(avg_fuel)
            success_rates.append(success_rate)

```

```
            fuel_consumptions.append(avg_fuel)
            success_rates.append(success_rate)

            # Her 1 bölümde bir rapor yazdır
            if episode % 1 == 0:
                time_metrics = time_estimator.update(episode)
                print("\nEpoch {}/{}".format(episode, num_episodes))
                print("=" * 50)
                print(f"Geçen Süre: {time_metrics['elapsed_time']}")
                print(f"Tahmini Kalan Süre: {time_metrics['estimated_remaining']}")
                print(f"Toplam Tahmini Süre: {time_metrics['total_estimated_time']}")
                print(f"İlerleme: %{time_metrics['progress_percentage']:.1f}")
                print(f"GPU Kullanımı: {time_metrics['gpu_usage']}")
                print(f"Ortalama Ödül: {avg_reward:.2f}")
                print(f"Ortalama Kayıp: {loss:.4f}" if loss is not None else "Kayıp: N/A")
                print(f"Ortalama Adım Sayısı: {avg_steps:.1f}")
                print(f"Başarı Oranı: {success_rate:.1f}%")
                print(f"Minimum Mesafe: {min_distance:.2f} AU")
                print(f"Yakıt Tüketimi: {avg_fuel:.1f} kg")
                print("=" * 50)

                # GUI için metrikleri güncelle
                shared_data['current_metrics'].update({
                    'episode': episode,
                    'reward': avg_reward,
                    'loss': loss if loss is not None else 0.0,
                    'success_rate': success_rate,
                    'min_distance': min_distance,
                    'fuel': avg_fuel
                })

            # Evrim uygula
            agent.evolve()

        except Exception as e:
            print(f"Metrik hesaplama hatası: {e}")
            continue

    return agent, total_rewards, episode_losses, success_rates, min_distances, fuel_consumptions
```

```
3610        except Exception as e:
3611            print(f"Eğitim hatası: {e}")
3612            traceback.print_exc()
3613            return None, [], [], [], [], []
```

```python
def simulate_journey(env: SpaceTravelEnv, agent: SACAgent):
    state = env.reset()
    spacecraft_positions = [env.spacecraft.position.cpu().numpy()]

    # Simlasyon metrikleri
    metrics = {
        'velocities': [float(torch.norm(env.spacecraft.velocity))],
        'accelerations': [0],
        'fuel_levels': [float(env.spacecraft.fuel_mass)],
        'distances_to_target': [float(torch.norm(
            env.spacecraft.position - env.destination.position
        ))],
        'time': [0]
    }

    while not env.done:
        # GPU üzerinde action hesaplama
        with torch.no_grad():
            action = agent.act(state)

        # Simülasyon adımı
        next_state, reward, done, _ = env.step(action)
        state = next_state
        total_reward += reward
        if done:  # Eğer bölüm bittiyse döngüden çık
            break
        # Pozisyon ve metrik kayıtları
        spacecraft_positions.append(env.spacecraft.position.cpu().numpy())
        metrics['velocities'].append(float(torch.norm(env.spacecraft.velocity)))
        metrics['accelerations'].append(float(torch.norm(env.spacecraft.acceleration)))
        metrics['fuel_levels'].append(float(env.spacecraft.fuel_mass))
        metrics['distances_to_target'].append(float(torch.norm(
            env.spacecraft.position - env.destination.position
        )))
        metrics['time'].append(env.time)
```

```python

    return env, np.array(spacecraft_positions), metrics

def create_animation(env: SpaceTravelEnv, spacecraft_positions: np.ndarray,
                     metrics: dict, destination_planet_name: str):
    plt.style.use('dark_background')
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))
    fig.suptitle(f'Uzay Yolculuğu Simülasyonu - Hedef: {destination_planet_name}',
                 color='white', size=12)

    # Scaling factor for AU conversion
    scale_factor = 1 / AU

    # Calculate maximum distance for plot limits
    max_distance = max(
        np.max(np.abs(spacecraft_positions[:, 0])),
        np.max(np.abs(spacecraft_positions[:, 1])) * scale_factor  # Add closing parenthesis here
    )

    # Main plot settings
    ax1.set_xlim(-max_distance * 1.1, max_distance * 1.1)
    ax1.set_ylim(-max_distance * 1.1, max_distance * 1.1)
    ax1.set_xlabel('X Position (AU)')
    ax1.set_ylabel('Y Position (AU)')
    ax1.grid(True, alpha=0.3)

    # Create sun
    sun = plt.Circle((0, 0), 0.1, color='yellow')
    ax1.add_artist(sun)

    # Planet orbits and Lagrange points
    for name, planet in env.planets.items():
        # Orbit
        orbit = plt.Circle((0, 0), planet.orbit_radius * scale_factor,
                           fill=False, linestyle='--', alpha=0.3)
        ax1.add_artist(orbit)

        # Convert Lagrange points to numpy
        l1_points = torch.stack(planet.l1_positions).cpu().numpy() * scale_factor
        l2_points = torch.stack(planet.l2_positions).cpu().numpy() * scale_factor

```

```
        # Plot Lagrange points
        ax1.plot(l1_points[:, 0], l1_points[:, 1], 'c.', alpha=0.3, label=f'{name} L1')
        ax1.plot(l2_points[:, 0], l2_points[:, 1], 'm.', alpha=0.3, label=f'{name} L2')

        # Influence radius circles
        influence_radius = planet.lagrange.influence_radius * scale_factor
        l1_circle = plt.Circle((l1_points[-1, 0], l1_points[-1, 1]),
                               influence_radius, color='c', fill=False, alpha=0.2)
        l2_circle = plt.Circle((l2_points[-1, 0], l2_points[-1, 1]),
                               influence_radius, color='m', fill=False, alpha=0.2)
        ax1.add_artist(l1_circle)
        ax1.add_artist(l2_circle)

    # Spacecraft trail
    trail, = ax1.plot([], [], 'r-', alpha=0.5)
    spacecraft, = ax1.plot([], [], 'ro')

    # Metrics plot
    ax2.set_xlabel('Time (days)')
    ax2.set_ylabel('Speed (km/s)')
    ax2.grid(True, alpha=0.3)
    speed_line, = ax2.plot([], [])

    ax1.legend(loc='upper right', fontsize='small')

    def animate(frame):
        # Update spacecraft position
        pos = spacecraft_positions[frame] * scale_factor
        spacecraft.set_data(pos[0], pos[1])

        # Update trail
        trail.set_data(spacecraft_positions[:frame+1, 0] * scale_factor,
                       spacecraft_positions[:frame+1, 1] * scale_factor)

        # Update speed plot
        times = np.array(metrics['time'][:frame+1]) / (24*3600)  # Convert to days
        speeds = np.array(metrics['velocities'][:frame+1]) / 1000  # Convert to km/s
        speed_line.set_data(times, speeds)
        ax2.relim()
        ax2.autoscale_view()
```

```
3733
3734        ani = animation.FuncAnimation(fig, animate, frames=len(spacecraft_positions),
3735                                      interval=50, blit=True)
3736        plt.close()
3737        return ani
3738
3739    def plot_training_metrics(scores: List[float], losses: List[float], epsilons: List[float]):
3740        plt.style.use('dark_background')
3741        fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 15))
3742
3743        ax1.plot(scores)
3744        ax1.set_title('Training Rewards')
3745        ax1.set_xlabel('Episode')
3746        ax1.set_ylabel('Total Reward')
3747        ax1.grid(True, alpha=0.3)
3748
3749        ax2.plot(losses)
3750        ax2.set_title('Training Loss')
3751        ax2.set_xlabel('Episode')
3752        ax2.set_ylabel('Loss')
3753        ax2.grid(True, alpha=0.3)
3754
3755        ax3.plot(epsilons)
3756        ax3.set_title('Exploration Rate')
3757        ax3.set_xlabel('Episode')
3758        ax3.set_ylabel('Epsilon')
3759        ax3.grid(True, alpha=0.3)
3760
3761        plt.tight_layout()
3762        return fig
3763
```

```python
class SpaceSimulationGUI:
    def __init__(self, width=1920, height=1080):
        self.width = width
        self.height = height
        self.screen = pygame.display.set_mode((width, height))
        pygame.display.set_caption("Uzay Aracı Simülasyonu")

        # Renkler
        self.BLACK = (0, 0, 0)
        self.WHITE = (255, 255, 255)
        self.YELLOW = (255, 255, 0)  # Güneş rengi
        self.RED = (255, 0, 0)       # Uzay aracı rengi
        self.BLUE = (0, 0, 255)      # Hedef gezegen rengi
        self.GREEN = (0, 255, 0)     # Diğer gezegen rengi
        self.GRAY = (128, 128, 128)  # Yörünge rengi

        # Ekran bölümleri
        self.sim_width = int(width * 0.8)  # Simülasyon alanı - %80
        self.panel_width = width - self.sim_width  # Yan panel - %20

        # Font ve UI elemanları
        self.title_font = pygame.font.Font(None, 48)  # Başlık fontu
        self.font = pygame.font.Font(None, 36)        # Normal font
        self.small_font = pygame.font.Font(None, 24)  # Küçük font

        # Simülasyon kontrolleri
        self.paused = False  # Duraklatma durumu
        self.current_sim_index = 0  # Aktif simülasyon indeksi
        self.show_all_sims = False  # Tüm simülasyonları göster
        self.zoom_level = 1.0  # Yakınlaştırma seviyesi
        self.pan_offset = [0, 0]  # Kaydırma konumu

        # Performans optimizasyonu
        self.clock = pygame.time.Clock()
        self.fps = 60  # Saniyedeki kare sayısı
        self.surface_cache = {}  # Yüzey önbelleği
        self.trail_points = {i: [] for i in range(24)}  # Her simülasyon için iz noktaları
        self.max_trail_length = 1000  # Maksimum iz uzunluğu
```

```
        # Düğmeler
        self.buttons = {
            'next_sim': pygame.Rect(width - 150, height - 100, 100, 40),  # Sonraki simülasyon
            'prev_sim': pygame.Rect(width - 270, height - 100, 100, 40),  # Önceki simülasyon
            'toggle_all': pygame.Rect(width - 390, height - 100, 100, 40),  # Tümünü göster/gizle
            'reset_view': pygame.Rect(width - 150, height - 160, 100, 40)  # Görünümü sıfırla
        }

        # Metrik geçmişi
        self.metric_history = {
            'hz': deque(maxlen=100),  # Son 100 hız değeri
            'yakıt': deque(maxlen=100),  # Son 100 yakıt değeri
            'mesafe': deque(maxlen=100),  # Son 100 mesafe değeri
            'ödül': deque(maxlen=100)  # Son 100 ödül değeri
        }

        # Ölçekleme faktörü - 60 AU'luk görüş alanı
        self.scale = width / (60 * AU)

        # Başlangıç pozisyonları
        self.center_x = width // 2  # Merkez X koordinatı
        self.center_y = height // 2  # Merkez Y koordinatı

    def world_to_screen(self, pos):
        """Dünya koordinatlarını ekran koordinatlarına dönüştür"""
        x = (pos[0] * self.scale * self.zoom_level + self.pan_offset[0]) + self.sim_width/2
        y = (pos[1] * self.scale * self.zoom_level + self.pan_offset[1]) + self.height/2
        return int(x), int(y)

    def draw_side_panel(self, metrics):
        """Yan paneli çiz - Metrik göstergeleri"""
        # Panel arkaplanı
        panel_rect = pygame.Rect(self.sim_width, 0, self.panel_width, self.height)
        pygame.draw.rect(self.screen, self.BLACK, panel_rect)
        pygame.draw.line(self.screen, self.WHITE, (self.sim_width, 0), (self.sim_width, self.height))

        # Panel başlığı
        title = self.title_font.render("Metrikler", True, self.WHITE)
        self.screen.blit(title, (self.sim_width + 20, 20))
```

```
3842
3843        # Metrikleri göster
3844        y = 100
3845        for key, value in metrics.items():
3846            # Metrik geçmişini güncelle
3847            self.metric_history[key].append(float(value))
3848
3849            # Metrik değerini göster
3850            text = self.font.render(f"{key}: {value}", True, self.WHITE)
3851            self.screen.blit(text, (self.sim_width + 20, y))
3852
3853            # Mini grafik çiz
3854            self.draw_mini_graph(key, self.sim_width + 20, y + 30)
3855            y += 100
3856
3857        # Kontrol düğmeleri
3858        self.draw_buttons()
3859
3860        # Simülasyon bilgisi
3861        sim_info = self.font.render(f"Simülasyon {self.current_sim_index + 1}/24", True, self.WHITE)
3862        self.screen.blit(sim_info, (self.sim_width + 20, self.height - 200))
3863
3864    def draw_mini_graph(self, metric_key, x, y):
3865        """Mini grafik çiz - Metrik geçmişi görselleştirmesi"""
3866        if len(self.metric_history[metric_key]) < 2:
3867            return
3868
3869        values = list(self.metric_history[metric_key])
3870        max_val = max(values)
3871        min_val = min(values)
3872        range_val = max_val - min_val or 1
3873
3874        points = []
3875        for i, val in enumerate(values):
3876            px = x + (i * 200 / len(values))
3877            py = y + 50 - ((val - min_val) * 50 / range_val)
3878            points.append((int(px), int(py)))
3879
3880        if len(points) > 1:
3881            pygame.draw.lines(self.screen, self.GREEN, False, points, 2)
3882
```

```python

    def draw_buttons(self):
        """Kontrol düğmelerini çiz"""
        for name, rect in self.buttons.items():
            color = self.GRAY if name == 'toggle_all' and self.show_all_sims else self.BLUE
            pygame.draw.rect(self.screen, color, rect)

            text = None
            if name == 'next_sim':
                text = self.font.render("→", True, self.WHITE)  # Sonraki
            elif name == 'prev_sim':
                text = self.font.render("←", True, self.WHITE)  # Önceki
            elif name == 'toggle_all':
                text = self.small_font.render("Tümü", True, self.WHITE)  # Tümünü göster/gizle
            elif name == 'reset_view':
                text = self.small_font.render("Sıfırla", True, self.WHITE)  # Görünümü sıfırla

            if text:
                text_rect = text.get_rect(center=rect.center)
                self.screen.blit(text, text_rect)

    def draw_simulation(self, env, metrics):
        """Simülasyonu çiz - Ana görselleştirme fonksiyonu"""
        # Ekranı temizle
        self.screen.fill(self.BLACK)

        # Güneş'i çiz
        sun_pos = (self.center_x + self.pan_offset[0],
                   self.center_y + self.pan_offset[1])
        pygame.draw.circle(self.screen, self.YELLOW, sun_pos,
                           int(20 * self.zoom_level))

        # Gezegenleri ve yörüngelerini çiz
        for name, planet in env.planets.items():
            # Yörünge
            orbit_radius = int(planet.orbit_radius * self.scale * self.zoom_level)
            pygame.draw.circle(self.screen, self.GRAY, sun_pos, orbit_radius, 1)

            # Gezegen pozisyonu
            pos = planet.position.cpu().numpy()
            screen_x = self.center_x + int(pos[0] * self.scale * self.zoom_level) + self.pan_offset[0]
            screen_y = self.center_y + int(pos[1] * self.scale * self.zoom_level) + self.pan_offset[1]

```

```
3929
3930            # Gezegen adını yaz
3931            text = self.small_font.render(name, True, self.WHITE)
3932            self.screen.blit(text, (screen_x + 10, screen_y - 10))
3933
3934        # Uzay araçlarını çiz
3935        if self.show_all_sims:
3936            for i, spacecraft in enumerate(env.spacecraft):
3937                self.draw_spacecraft(spacecraft, i, alpha=128)  # Yarı saydam
3938        else:
3939            self.draw_spacecraft(env.spacecraft[self.current_sim_index],
3940                                 self.current_sim_index)
3941
3942        # Yan paneli çiz
3943        self.draw_side_panel(metrics)
3944
3945        # Ekranı güncelle
3946        pygame.display.flip()
3947
3948    def draw_spacecraft(self, spacecraft, index, alpha=255):
3949        """Uzay aracını çiz - İz ve durum göstergeleri"""
3950        pos = spacecraft.position.cpu().numpy()
3951        screen_x = self.center_x + int(pos[0] * self.scale * self.zoom_level) + self.pan_offset[0]
3952        screen_y = self.center_y + int(pos[1] * self.scale * self.zoom_level) + self.pan_offset[1]
3953
3954        # İz noktalarını güncelle
3955        self.trail_points[index].append((screen_x, screen_y))
3956        if len(self.trail_points[index]) > self.max_trail_length:
3957            self.trail_points[index].pop(0)
3958
3959        # İzi çiz
3960        if len(self.trail_points[index]) > 1:
3961            pygame.draw.lines(self.screen, (*self.RED[:3], alpha), False,
3962                              self.trail_points[index], int(2 * self.zoom_level))
3963
3964        # Uzay aracını çiz
3965        pygame.draw.circle(self.screen, (*self.RED[:3], alpha),
3966                           (screen_x, screen_y), int(5 * self.zoom_level))
```

```python
    def handle_events(self):
        """Kullanıcı girdilerini işle - Kontrol ve etkileşim"""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return False

            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    self.paused = not self.paused  # Duraklat/Devam et
                elif event.key == pygame.K_r:
                    self.zoom_level = 1.0  # Yaknlaştırmayı sıfırla
                    self.pan_offset = [0, 0]  # Kaydırmayı sıfırla

            elif event.type == pygame.MOUSEBUTTONDOWN:
                pass

    def run_simulation(self, env, agent):
        """Ana simülasyon döngüs"""
        running = True

        while running:
            running = self.handle_events()

            if not self.paused:
                # Simülasyon adım
                state = env.get_state()
                with torch.no_grad():
                    action = agent.act(state)
                next_state, reward, done, info = env.step(action)
                state = next_state
                # Metrikleri güncelle
                metrics = {
                    'hız': f"{float(torch.norm(env.spacecraft[self.current_sim_index].velocity))/1000:.1f} km/s",
                    'yakıt': f"{float(env.spacecraft[self.current_sim_index].fuel_mass):.0f} kg",
                    'mesafe': f"{float(torch.norm(env.spacecraft[self.current_sim_index].position - env.destination.position))/AU:.2f} AU",
                    'ödül': f"{float(reward[self.current_sim_index]):.1f}",
                    'info': info  # info'yu ekle
                }

```

```
4006
4007                # Ekranı güncelle
4008                self.screen.fill(self.BLACK)
4009                self.draw_simulation(env, metrics)
4010                self.draw_side_panel(metrics)
4011                pygame.display.flip()
4012
4013                if done.any():
4014                    env.reset()
4015
4016            self.clock.tick(self.fps)
4017
4018        pygame.quit()
4019
4020    def update(self, metrics):
4021        self.screen.fill(self.BLACK)
4022        # Metrik bilgilerini ekrana yaz
4023        font = pygame.font.Font(None, 36)
4024        y = 10
4025        for key, value in metrics.items():
4026            text = font.render(f"{key}: {value:.2f}", True, self.WHITE)
4027            self.screen.blit(text, (10, y))
4028            y += 40
4029        pygame.display.flip()
4030
```

```python
def run_gui_process(shared_data: Dict[str, Any]):
    """GUI process'ini çalıştıran fonksiyon"""
    pygame.init()
    screen = pygame.display.set_mode((800, 600))
    clock = pygame.time.Clock()
    font = pygame.font.SysFont(None, 24)

    while shared_data['running']:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                shared_data['running'] = False

        # Ekranı temizle
        screen.fill((0, 0, 0))

        # Mevcut metrikleri göster
        if 'current_metrics' in shared_data:
            y = 10
            for key, value in shared_data['current_metrics'].items():
                text = font.render(f"{key}: {value}", True, (255, 255, 255))
                screen.blit(text, (10, y))
                y += 30

        pygame.display.flip()
        clock.tick(30)

    pygame.quit()
```

```python
def train_with_visualization():
    """Eğitim sürecini görselleştirmeyle birlikte yürüten ana fonksiyon"""
    # Shared data dictionary'sini başlangıçta tanımla
    shared_data = {
        'running': True,
        'paused': False,
        'current_metrics': {},
        'current_positions': {}
    }

    # GUI process'ini başlat
    gui_process = multiprocessing.Process(target=run_gui_process, args=(shared_data,))
    gui_process.start()

    try:
        env = SpaceTravelEnv("Neptune")
        agent = SACAgent(env.state_size, env.action_size)

        # Eğitim döngüsü
        for episode in range(NUM_EPISODES):
            state = env.reset()
            total_reward = 0

            for step in range(MAX_STEPS):
                action = agent.act(state)
                next_state, reward, done, _ = env.step(action)

                agent.remember(state, action, reward, next_state, done)

                if len(agent.memory) > agent.batch_size:
                    agent.replay()

                state = next_state
                total_reward += reward

                # GUI için metrikleri güncelle
                shared_data['current_metrics'] = {
                    'episode': episode,
                    'step': step,
                    'reward': total_reward,
                    'epsilon': agent.epsilon
                }
```

```
                shared_data['current_positions'] = {
                    'spacecraft': env.spacecraft[0].position.tolist(),
                    'destination': env.destination.position.tolist()
                }

                if done:
                    break

            # Her 10 bölümde bir hedef modeli güncelle
            if episode % 10 == 0:
                agent.save_model(f"sac_model_episode_{episode}.pth")

            print(f"Bölüm: {episode}, Toplam Ödül: {total_reward}, Epsilon: {agent.epsilon}")

    except Exception as e:
        print(f"Eğitim hatası: {e}")
    finally:
        shared_data['running'] = False
        if gui_process.is_alive():
            gui_process.join()

```

```python
def draw_simulation(env, screen):
    """Simülasyonu ekrana çiz"""
    # Renk tanımlamaları
    WHITE = (255, 255, 255)
    BLACK = (0, 0, 0)
    RED = (255, 0, 0)
    BLUE = (0, 0, 255)
    YELLOW = (255, 255, 0)

    # Ekran boyutları
    SCREEN_WIDTH = 800
    SCREEN_HEIGHT = 600

    # Ölçekleme faktörü
    SCALE = 100

    screen.fill(BLACK)

    # Güneş'i çiz
    pygame.draw.circle(screen, YELLOW,
                       (SCREEN_WIDTH//2, SCREEN_HEIGHT//2), 20)

    # Gezegenleri çiz
    for name, planet in env.planets.items():
        pos = planet.position.cpu().numpy()
        screen_x = SCREEN_WIDTH//2 + int(pos[0] / AU * SCALE)
        screen_y = SCREEN_HEIGHT//2 + int(pos[1] / AU * SCALE)

        color = BLUE if name != env.destination.name else RED
        pygame.draw.circle(screen, color, (screen_x, screen_y), 5)

    # Uzay aracını çiz
    for spacecraft in env.spacecraft:
        pos = spacecraft.position.cpu().numpy()
        screen_x = SCREEN_WIDTH//2 + int(pos[0] / AU * SCALE)
        screen_y = SCREEN_HEIGHT//2 + int(pos[1] / AU * SCALE)
        pygame.draw.circle(screen, WHITE, (screen_x, screen_y), 3)

    pygame.display.flip()

def main():
```

```python

def main():
    try:
        DESTINATION_PLANET = 'Neptune'
        POPULATION_SIZE = 48
        NUM_EPISODES = 50000
        total_rewards = []

        print("\nSimülasyon başlatılıyor...")
        env = SpaceTravelEnv(DESTINATION_PLANET, num_envs=POPULATION_SIZE)
        print("Ortam başarıyla oluşturuldu")

        agent = LongTermGeneticSACAgent(env.state_size, env.action_size)
        print("Ajan başarıyla oluşturuldu")

        print("\nEğitim başlıyor...")
        for episode in range(NUM_EPISODES):
            print(f"\nBölüm {episode + 1}/{NUM_EPISODES} başlıyor...")
            state = env.reset()
            episode_reward = torch.zeros(POPULATION_SIZE, device=agent.device)
            done = torch.zeros(POPULATION_SIZE, dtype=torch.bool, device=agent.device)  # done'ı tensor olarak başlat

            while not done.all():  # şimdi .all() çalışacak
                action = agent.act_batch(state)
                next_state, reward, done, _ = env.step_batch(action)
                agent.remember_batch(state, action, reward, next_state, done)
                state = next_state
                episode_reward += reward

                # Bellek yeterince dolunca eğitim yap
                if len(agent.memory) >= agent.batch_size:
                    agent.replay()

            # Episode sonunda popülasyonu evrimleştir
            agent.evolve()
            total_rewards.append(episode_reward.mean().item())

        # Eğitimi başlat
        trained_agent, rewards, losses, success_rates, distances, fuels = train_agent(DESTINATION_PLANET, NUM_EPISODES)
        trained_agent.save_population('trained_model.pth')
        while not done.all():  # done bir tensor olduğu için .all() kullan
            action = agent.act_batch(state)
            next_state, reward, done, _ = env.step_batch(action)
```

```
4206            gui = SpaceSimulationGUI()
4207            gui.run_simulation(env, trained_agent)
4208
4209            # Eğitim metriklerini görselleştir
4210            plt.figure(figsize=(15, 10))
4211
4212            plt.subplot(231)
4213            plt.plot(rewards)
4214            plt.title('Ortalama Ödüller')
4215            plt.xlabel('Episode')
4216
4217            plt.subplot(232)
4218            plt.plot(losses)
4219            plt.title('Eğitim Kaybı')
4220            plt.xlabel('Episode')
4221
4222            plt.subplot(233)
4223            plt.plot(success_rates)
4224            plt.title('Başarı Oranı (%)')
4225            plt.xlabel('Episode')
4226
4227            plt.subplot(234)
4228            plt.plot(distances)
4229            plt.title('Minimum Mesafe (AU)')
4230            plt.xlabel('Episode')
4231
4232            plt.subplot(235)
4233            plt.plot(fuels)
4234            plt.title('Yakıt Tüketimi (kg)')
4235            plt.xlabel('Episode')
4236
4237            plt.tight_layout()
4238            plt.savefig('training_metrics.png')
4239            print("\nMetrikler 'training_metrics.png' dosyasına kaydedildi.")
4240
4241        except Exception as e:
4242            print(f"\nHata oluştu: {e}")
4243            traceback.print_exc()
4244        finally:
4245            torch.cuda.empty_cache()
4246            plt.close('all')
4247            print("\nProgram sonlandırıldı.")
```

```
4244        finally:
4245            torch.cuda.empty_cache()
4246            plt.close('all')
4247            print("\nProgram sonlandırıldı.")
4248
4249    if __name__ == "__main__":
4250        main()
```