**Apollo AI: AI-Driven Lunar Landing – Phase 3 Update**

**Team Members**

  A person with a beard

AI-generated content may be incorrect.

From left to right: Tamar Boone (MS student in Robotics Engineering), Sidhant Karamchandani (MS student in Robotics Engineering), Palanivel Sathiya Moorthy (MS student in Robotics Engineering)

**Abstract**

The Apollo AI project presents a comprehensive investigation into AI-driven autonomous landing systems for lunar exploration modules, addressing the complex challenges posed by the dynamic and uncertain lunar environment. Leveraging the LunarLander-v3 simulation from Gymnasium, this study evaluates and compares three state-of-the-art (SOTA) methods: Monte Carlo Tree Search (MCTS), Deep Reinforcement Learning (DRL), and Neural Network-Based Model Predictive Control (NN-MPC). MCTS explores randomized landing scenarios to identify robust strategies, DRL trains an agent through trial-and-error to optimize throttle and orientation adjustments, and NN-MPC combines neural networks with control theory for real-time trajectory optimization. Key milestones include implementing these methods using libraries such as PyTorch, Gymnasium, and NumPy, and evaluating their performance based on landing accuracy, fuel efficiency, and stability.

Preliminary results highlight the adaptability of DRL, which successfully solved the environment by achieving an average score of 200+ after 691 episodes, demonstrating its ability to learn precise control policies despite early exploration challenges. In contrast, MCTS struggled with computational inefficiency due to the environment's stochastic initialization and the difficulty of replicating simulations accurately, while NN-MPC faced limitations in dynamics modeling, as its learned neural network failed to predict the environment's nonlinear physics reliably. These findings underscore the trade-offs between model-free (DRL) and model-based (MCTS, NN-MPC) approaches in complex, sparse-reward tasks.

The project aligns with real-world space exploration goals, such as NASA’s Artemis program, by advancing autonomous landing technologies critical for future lunar and Martian missions. Future work will focus on hybrid methods, improved dynamics modeling, and real-world robustness enhancements to bridge the gap between simulation and deployment.

**Keywords**: Autonomous lunar landing, reinforcement learning, Monte Carlo simulations, model predictive control, Gymnasium, AI-driven space exploration.

**Motivations & Domain**

The LunarLander-v3 environment simulates the complex task of autonomously landing a spacecraft on the moon, a challenge that falls under the domain of autonomous control systems and reinforcement learning (RL). The goal is to control the lander’s thrusters to manage its position, velocity, and orientation while minimizing fuel consumption and avoiding crashes, all within a dynamic and uncertain environment. This problem is inspired by real-world challenges in space exploration, where factors like low gravity, lack of atmosphere, and the need for precision make lunar landings particularly difficult. Historically, lunar landings have relied on human-operated systems or pre-programmed algorithms, but advancements in AI and RL offer the potential for more efficient and adaptive autonomous systems. Autonomous landing technologies are critical for future missions to the moon, Mars, and beyond, as they reduce reliance on human intervention and enable more ambitious exploration. For example, NASA’s Artemis program aims to return humans to the moon by 2025 using autonomous systems (National Aeronautics and Space Administration, 2021), while failures like the 2019 crash of the Israeli Beresheet lander, which resulted in a $100 million loss, highlight the need for reliable autonomous solutions (National Aeronautics and Space Administration, n.d.). NASA’s Autonomous Landing Hazard Avoidance Technology (ALHAT), which successfully demonstrated real−time hazard detection and trajectory adjustment during field tests (National Aeronautics and Space Administration, n.d.), and SpaceX’s over 200 successful autonomous Falcon 9 landings as of 2023 (SpaceX, 2023), demonstrate the feasibility and importance of these systems. The global space economy, valued at 464 billion in 2022 and projected to reach $1 trillion by 2040 (Morgan Stanley, 2020), underscores the growing significance of autonomous technologies, especially given that lunar missions have historically faced challenges, with many attempts ending in failure (Choi, 2021).

Given the current US administration’s goals (mainly Elon Musk) to get humans to Mars, NASA’s Artemis program is necessary to achieve this goal in a practical way. If the Artemis program is successful, the Moon can be used as an intermediate stop on the way to Mars by creating a lunar base. The latest mission to Mars took about 6 – 7 months one way and was completed by the Mars rover Perserverance. Currently, the longest person to stay in space in a continuous trip was Valeri Polyakov, a Russian Astronaut that spent 437 continuous days aboard the Mir space station from 1994–1995 (*Times*, 2022, p. 35). This means that a trip to Mars is very much possible. However, since no one has experience living on another planet, the trip to the Moon can be used to simulate how someone might live on Mars in terms of extracting resources from the planet, building livable structures, and more.

**SOTA Methods**

For this project, we will experiment with and evaluate existing state of the art (SOTA) methods to achieve a safe and effective lunar landing. Specifically, the three SOTA methods that we will explore are Monte Carlo Simulations, Deep Reinforcement Learning, and Neural Network-Based Model Predictive Control. More details on each below.

**Monte Carlo Simulations**

Monte Carlo methods involve running thousands of randomized simulations to explore the possible outcomes and optimize decision-making. In our case, we will randomize many terrain profiles and initial conditions, let the lunar module choose the best next step at each moment, and aggregate the data at the end for the most optimal path to the landing zone. The MCS method is very valuable in our case (Lugo et al., 2022) as it allows for a rapid analysis of many different scenarios, and it allows us to visualize a comprehensive view of possible outcomes in a field where historical data is lacking or hard to parse/translate.

**Deep Reinforcement Learning**

DRL trains an agent (in our case, the lunar lander) to learn an optimal policy through trial and error in a simulated environment. We will need to define a reward function in which we reward safe landings: low velocity, stable orientation, proximity to the target. The neural network will be trained on inputs such as terrain obstacles, altitude, and velocity, to actions such as thrust and orientation changes. We have a few algorithms within DRL to choose from, but we will need to do more research to decide which one(s) to use. DRL is a good choice for our use case (Awasthi, 2025; Mali et al., 2023) as it is very adaptable to complex, dynamic environments. In real life, no two lunar landings will ever be the same.

**Neural Network-Based Model Predictive Control**

NN-MPC is a hybrid approach that combines control theory with the flexibility of neural networks for non-linear dynamics. While traditional models rely on physics equations, in this case these are replaced by neural networks trained to predict the system's state transitions based on current state and actions. Again, we will simulate lunar landing scenarios and collect data, training the neural network on inputs (current state and action) and outputs (next state). Again, this method is a good choice for our use case (Wang et al., 2024; Silvestrini et al., 2020), as it can handle complex scenarios, and it is known to be both adaptable and computationally efficient.

**Proposed High-level Solution and Process:**

For the *LunarLander-v3* environment in Gymnasium, we propose a structured approach to evaluate and compare three state-of-the-art (SOTA) methods: Monte Carlo Simulations (MCS), Deep Reinforcement Learning (DRL), and Neural Network-Based Model Predictive Control (NN-MPC). Each method addresses the challenge of achieving a safe and fuel-efficient lunar landing, but they differ in their underlying mechanisms and suitability for the environment’s dynamics. Libraries that will be used are the following:

* Gymnasium: Environment simulation.
* PyTorch/TensorFlow: Needed neural networks for DRL and NN-MPC.
* Numpy: Numerical operations (array manipulations, math functions)
* Collections (dequeue): Efficient queue for storing recent scores
* Random: Random action selection
* Glob: Find saved video files
* Io: Read video files for display
* Base64: Encode videos for HTML embedding
* Imageio: Save rendered frames as MP4
* Concurrent.futures: Parallelize MCTS simulations for speed.

Monte Carlo Simulations (MCS) will be employed to explore the environment’s stochastic nature by running thousands of randomized landing scenarios. Given *LunarLander-v3*’s continuous action space (throttle for main and side engines), we will use the Cross-Entropy Method (CEM) to iteratively refine action sequences. At each step, CEM samples candidate actions, evaluates their outcomes using the environment’s physics, and selects the best-performing sequences for the next iteration. This approach is particularly useful for identifying robust strategies in the absence of a pre-trained model, though it may struggle with real-time optimization due to computational costs.

Deep Reinforcement Learning (DRL) will leverage the *LunarLander-v3*’s reward function, which penalizes high velocities, misalignment, and crashes while rewarding proximity to the landing pad. We will implement Proximal Policy Optimization (PPO), a stable DRL algorithm, to train a neural network policy. The policy will map the lander’s state (8D vector: position, velocity, angle, etc.) to optimal throttle adjustments. Training will involve episodic rollouts with a customized reward function, such as reward = -distance\_to\_target - 0.1\*velocity - 0.5\*angle, to incentivize smooth landings. While DRL excels in adaptability, its reliance on extensive trial-and-error training may limit initial performance.

Neural Network-Based MPC (NN-MPC) combines the predictive power of neural networks with the precision of control theory. A dynamics model will be trained offline using data from random or MCS-generated trajectories, learning to predict the next state (e.g., lander’s position and velocity) given the current state and action. During deployment, MPC will optimize actions over a receding horizon using CEM or gradient-based methods, minimizing a cost function (e.g., distance to target + fuel usage). NN-MPC is well-suited to *LunarLander-v3*’s nonlinear dynamics, offering a balance between computational efficiency and adaptability, though its performance hinges on the accuracy of the learned model.

Implementation Details:

* Environment Setup: We will use Gymnasium’s *LunarLander-v3* with a modified reward function to emphasize safety.
* Data Collection: For NN-MPC, we will generate training data by simulating random or MCS-guided trajectories.
* Tooling: PyTorch will train the DRL policy and dynamics model, while Ray will parallelize MCS rollouts.
* Evaluation: Success will be measured by landing accuracy (distance to target), fuel efficiency (total thrust), and stability (angle at touchdown).

Preliminary Insights:

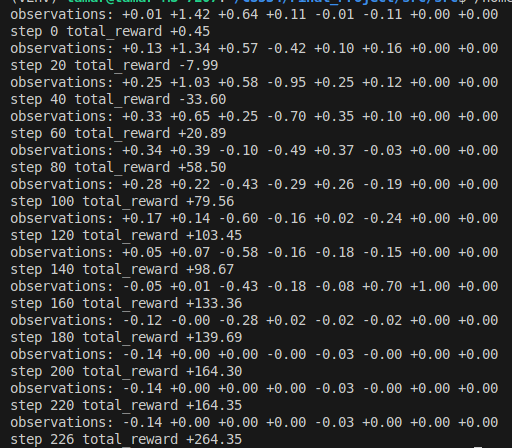
* MCS provides a baseline but may lack precision due to its reliance on random sampling.
* DRL is expected to achieve high success rates but requires significant training time.
* NN-MPC could offer the best trade-off, leveraging learned dynamics for real-time control.

**Evaluation**

To evaluate the performance and effectiveness of each of our techniques and algorithms above, we first need to define the success criteria in this context. The environment that we’re using is being provided by the Gymnasium environment provided by <https://gymnasium.farama.org>, which is an open-source Reinforcement Learning (RL) benchmarking platform, developed as part of the Farama Foundation. This environment was chosen to save time on having the team build an environment from scratch and it has clearly defined objectives to solve for a solution to the environment. The environment has the following conditions:

1. The landing pad is always at coordinates (0,0).
2. The lander starts at the top center of the viewing screen (directly above the landing pad) with a random initial force applied to its center of mass.
3. The observation state is an 8-dimensional vector: the coordinates of the lander in x & y, its linear velocities in x & y, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not.
4. There are four discrete actions available:
   1. 0: do nothing
   2. 1: fire left orientation engine
   3. 2: fire main engine
   4. 3: fire right orientation engine
5. After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode. For each step, the reward:
   1. is increased/decreased the closer/further the lander is to the landing pad.
   2. is increased/decreased the slower/faster the lander is moving.
   3. is decreased the more the lander is tilted (angle not horizontal).
   4. is increased by 10 points for each leg that is in contact with the ground.
   5. is decreased by 0.03 points each frame a side engine is firing.
   6. is decreased by 0.3 points each frame the main engine is firing.
   7. The episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively.
   8. An episode is considered a solution if it scores at least 200 points.
6. The episode finishes if:
   1. The lander crashes (the lander body gets in contact with the moon);
   2. The lander gets outside of the viewport (x coordinate is greater than 1);
   3. The lander is not awake. From the [Box2D docs](https://box2d.org/documentation/md__d_1__git_hub_box2d_docs_dynamics.html#autotoc_md61), a body which is not awake is a body which doesn’t move and doesn’t collide with any other body

Our algorithm will consider the environment solved when an average score of 200 is reached within a 100-episode block. This means that an average score will be taken every 100 episodes, and when the average score reaches and average score of 200 within a 100-episode segment, the algorithm will consider the last 100 episodes the solution to the environment, and present that solution as a video of the Lunar-Lander attempting to land within the landing pad. If it lands within the landing pad without issues, then we’ll know if the algorithm works or not. An example of a solution to the environment is provided by the Gymnasium library in the site package (gymnasium/envs/box2d/lunar\_lander.py). This solution solves the environment using a heuristics approach and is what we’ll use as a comparison for our algorithm. An example run of the Gymnasium provided code is shown below:



Picture 1: Example run of the heuristic solution provided by lunar\_lander.py

**Final Experimental Results and Discussions**

*MCTS*

The Monte Carlo Tree Search (MCTS) approach for the LunarLander-v3 environment operates by building a search tree through repeated simulations to determine optimal actions at each state. Beginning with the current environment state as the root node, the algorithm expands the tree by generating child nodes for each possible action (firing main engine, left/right thrusters, or doing nothing). For each action, it performs N=100 randomized simulations (rollouts) that continue until episode termination (either successful landing or crash), accumulating rewards along each trajectory. These rollouts enable MCTS to estimate the expected value of each action by averaging the total rewards across all simulations for that branch. The action with the highest average reward is selected for execution in the actual environment, and the process repeats for the next state. To mitigate computational costs, parallelization via multithreading was implemented, with each thread running independent environment copies to accelerate simulations.

How the MCTS algorithm was constructed is explained below.

1. The current state of the environment is considered the root node. From this, the root node is expanded, and four child nodes are created (1 child node for each of the four possible actions).
2. For each of the four children, N simulations are run on the environment, until either the Lander crashes or lands successfully (in other words the episode completes)
   1. N = 100 was chosen because it was sufficient to find successful simulations, and higher values of N would lower the performance in terms of time.
3. After the N simulations, the total reward is averaged across the N simulations for each child node, and this score is backpropagated to be associated with the corresponding original child node.
4. After each child node has a score, the action corresponding to the child with the highest score is chosen as the next action.

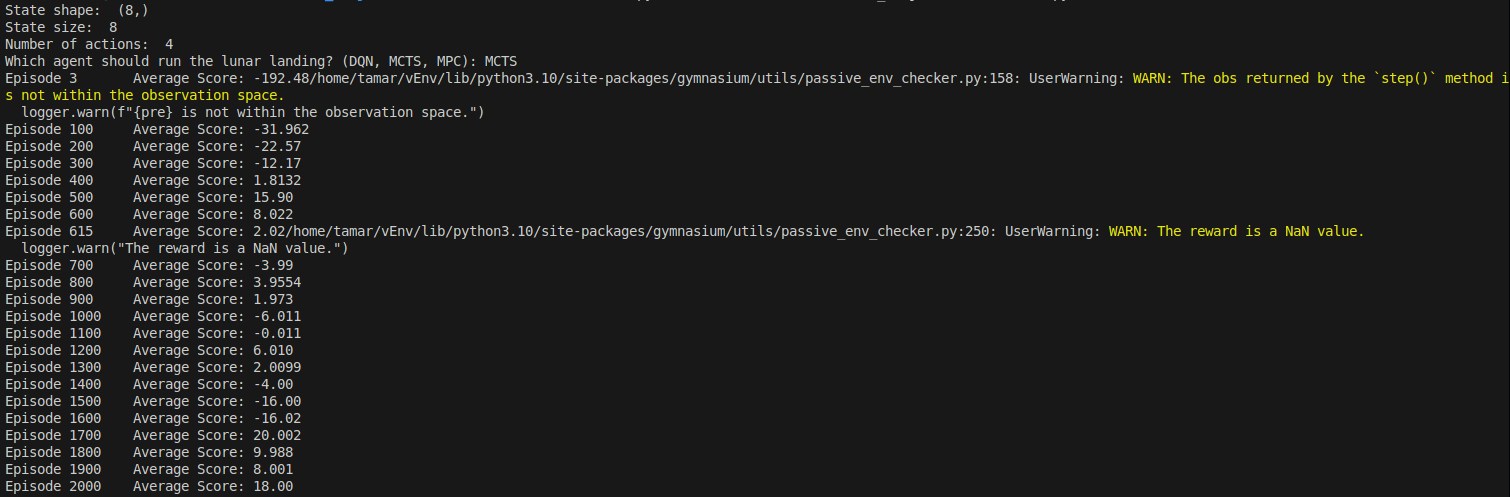
*Issues with MCTS*

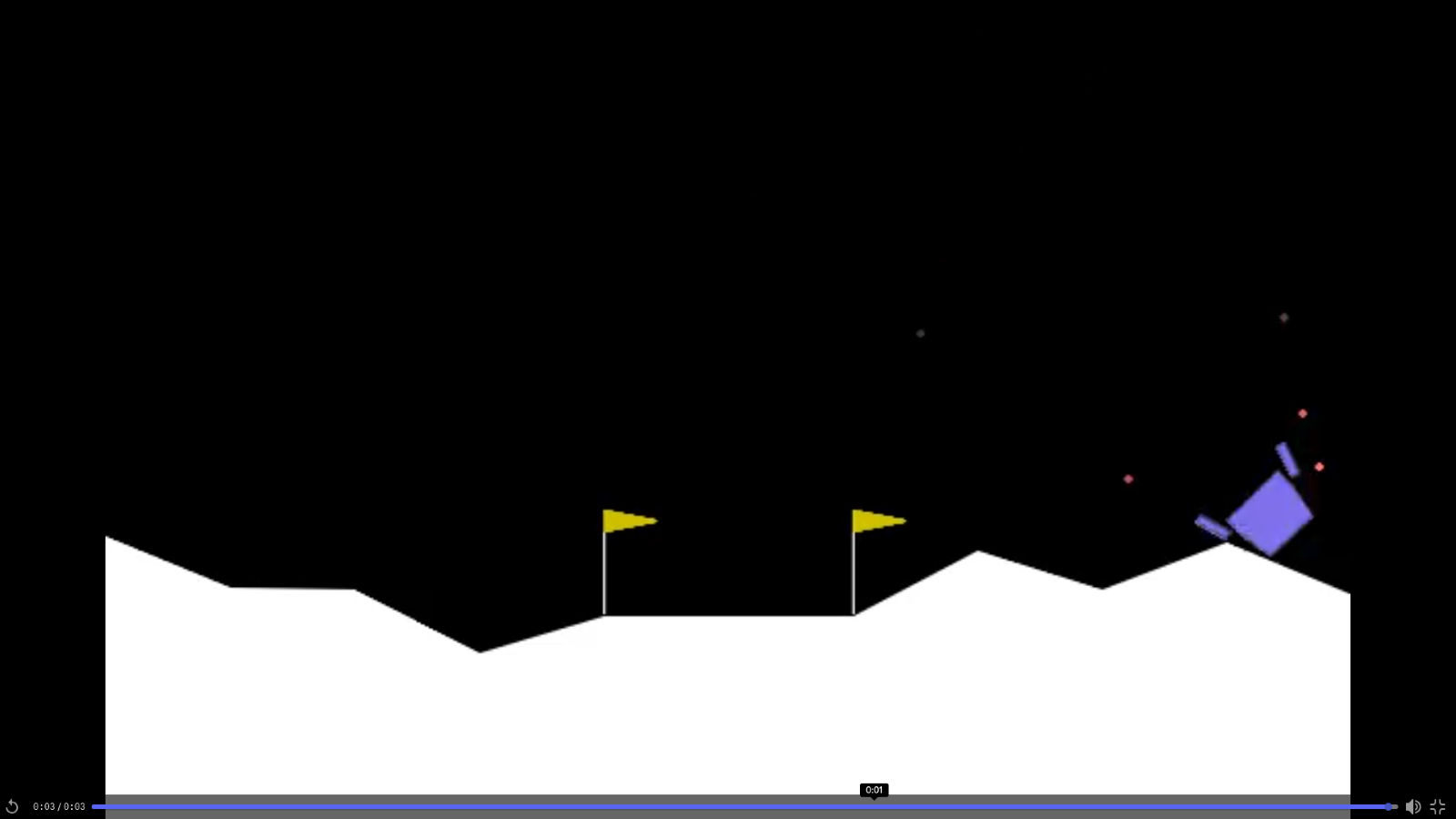
Because the MCTS requires full simulation runs, the construction of the Lunar-Lander\_v3 doesn’t align with MCTS strengths. The Lunar-Lander\_v3 environment uses 2000 max episodes, each consisting of 1000 max timesteps. This means that a MCTS would have to run a full 2000-episode simulation to find a reliable solution vs stopping once it finds a solution within a 100-episode block per our criteria for a viable solution. We moved forward with 2000 episodes and 1000 timesteps anyway. However, this was not time efficient at all. To improve the time efficiency, multi-threading was implemented. Each simulation was executed in parallel across different CPU cores, and a brand new environment copy was created for each thread. This enabled us to execute roughly 100 episodes per minute, which is acceptable considering how many simulations we're running and how computationally expensive modifying the environment is. In addition, the library that we are using for this project is not friendly at all to creating copies and simulations. The gymnasium Python library wraps C++ objects in swig containers. Therefore, copying the outer environment doesn’t help as the inner objects point to their original addresses in memory. To work around this, we tried two things:

1. Create a brand new environment and copy every possible attribute we can access from the original environment to the new environment.
   1. This didn’t fully work as it wasn’t as simple as just copying over the (x, y) position of the lander because the environment’s step() function involved many more internal variables, functions, and physics. However, this method did have some success.
2. Create a brand new environment, keep track of all the actions applied to the original environment, and apply that exact same sequence of actions to the copy environment to create a replica at each time step.
   * 1. This method did not work well because each environment is initialized with some randomness (a random force applied to the lander’s center of gravity).

Although the MCTS was not able to come up with a solution to solve the environment, the lander was able to improve its score over time as the initial score ranged around –190 to -30 and increased to around 20. We are confident that this approach would work if replicating the environment was easier.

Here’s a run of the MCTS simulation:

Figure 2: Example run of the MCTS solution

Figure 3: Example run of the MCTS solution result

*NN-MPC*

The Neural Network Model Predictive Control (NN-MPC) agent represents an experimental hybrid approach that merges neural network-based dynamics modeling with control-theoretic optimization for autonomous lunar landing. At the heart of this system lies a carefully designed DynamicsNetwork - a neural network featuring two hidden layers with 256 neurons each, utilizing ReLU activation and LayerNorm for feature normalization. This network takes as input the current 8-dimensional state vector (containing position, velocity, angle, and other parameters) along with a one-hot encoded action tensor (meaning a tensor where the action index is a 1 and all other action indexes are a 0), and outputs both a predicted next state (expressed as a delta from the current state) and an estimated reward. The agent operates in two distinct modes: during action selection (act()), it employs the Cross-Entropy Method (CEM) to evaluate multiple potential action sequences across a dynamically expanding planning horizon that grows from 1 to 10 steps as training progresses. This CEM process involves simulating numerous action sequences through the learned dynamics model, predicting their outcomes, and selecting the most promising initial action for actual execution in the environment. During learning phases (learn()), the system switches to training mode, drawing from a replay buffer of past experiences to compute and minimize a weighted loss function that prioritizes reward prediction accuracy (weighted 5 times more heavily) over state prediction precision, using the Adam optimizer with a carefully tuned learning rate of 1e-4 to ensure stable training without overshooting optimal parameters.

The final architecture and training approach emerged from extensive experimentation with various configurations. Initial tests with three hidden layers and dropout rates of 10-20% were abandoned when they degraded performance, likely due to the relatively small network size and the environment's deterministic nature. Similarly, alternative activation functions (Leaky ReLU, GELU, SiLU) failed to outperform standard ReLU, while different optimizers revealed Adam as the clear winner over AdamW, SGD, and RMSprop. The dynamic adjustment of the planning horizon proved particularly crucial, allowing the agent to begin with cautious, short-term planning before gradually incorporating longer-term strategies as its predictive capabilities improved. This carefully optimized combination of architectural choices and training strategies enables the NN-MPC agent to simultaneously learn the environment's dynamics while optimizing its control policies, though challenges remain in achieving consistently successful landings due to the inherent difficulties in precisely modeling the lunar lander's complex physics.

How the NN-MPC algorithm was constructed is explained below.

1. Dynamics Network was built with:
   1. Input: 8D state vector + one-hot encoded action (4D)
   2. Architecture: 2 hidden layers (256 neurons each) with ReLU and LayerNorm
   3. Output: Predicted state delta (8D) + reward (1D)
2. Planning System:
   1. Uses Cross-Entropy Method (CEM) for action selection
   2. Samples 10 action sequences per step
   3. Planning horizon grows from 1→10 steps during training
   4. Executes first action of best predicted sequence
3. Training Process:
   1. Online learning (Continuously Updates its neural network *during* environment interaction) via experience replay (using pre-collected datasets)
   2. Weighted loss function (5:1 reward:state prediction)
   3. Adam optimizer (learning rate=1e-4)
   4. No dropout (harmful for this regression task)

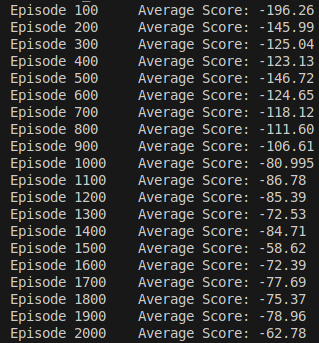
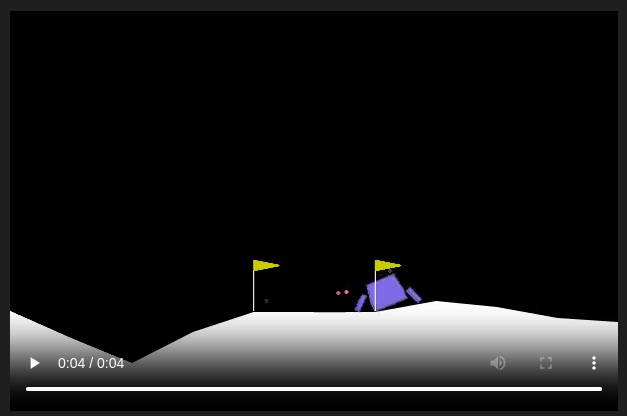


Figure 4: Example run of the NN-MPC solution

Figure 5: Example run of the NN-MPC solution result

*Issues with NN-MPC*

Despite its theoretical advantages, the NN-MPC agent struggles to solve the LunarLander-v3 environment, as evidenced by its persistently negative average scores. One major issue is the inaccuracy of the learned dynamics model. The environment's highly nonlinear and discontinuous dynamics, such as ground contact and thruster forces, are difficult for the neural network to predict precisely, leading to compounding errors over the planning horizon. Additionally, the sparse reward structure of LunarLander-v3—where the agent only receives a large positive reward upon successful landing—makes it challenging for the dynamics model to learn meaningful reward predictions. The agent's reliance on random action sampling (HORIZON\_SAMPLES = 10) further limits its ability to discover high-reward trajectories, as it lacks systematic exploration mechanisms like DQN's epsilon-greedy policy. Computational constraints, such as the initially short planning horizon and low sample count, exacerbate these issues, preventing the agent from effectively strategizing long-term maneuvers like controlled deceleration before landing.

*DRL*

The Deep Q-Network (DQN) agent successfully solves the LunarLander-v3 environment by leveraging deep reinforcement learning (DRL) to learn an optimal policy through trial-and-error interactions with the simulated lunar landing scenario. At its core, DQN combines Q-learning, a classic reinforcement learning algorithm, with deep neural networks to approximate the optimal action-value function (Q-function), which estimates the expected cumulative reward for taking a specific action in a given state. The agent observes the environment's state (including position, velocity, orientation, and contact flags) and selects actions (fire main engine, left/right orientation thrusters, or do nothing) based on an epsilon-greedy policy. This policy balances exploration (random actions) and exploitation (actions with the highest predicted Q-values) by gradually decaying the exploration rate (epsilon) from 1.0 to 0.01 over training. The neural network (a three-layer fully connected model) maps states to Q-values for each action, and the agent updates its predictions using temporal difference learning, where the target Q-values are computed using a separate target network to stabilize training. Experience replay is employed to break correlations in the data by storing transitions in a buffer and sampling mini-batches for learning.

How the DRL algorithm was constructed is explained below.

1. Q-Network (DQN Agent):
   1. A neural network that estimates Q-values (expected future rewards) for each action given a state.
   2. Input: Current state (8D vector).
   3. Output: Q-values for each of the 4 discrete actions.
   4. Architecture: Three fully connected layers (64 neurons each) with ReLU activation.
2. Learning Mechanism:
   1. Experience Replay: Stores past transitions in a buffer and samples mini-batches to break correlations.
   2. Target Network: A separate, slowly updated network to stabilize Q-value targets.
   3. Loss Function: Mean Squared Error (MSE) between predicted and target Q-values.
3. Exploration vs. Exploitation:
   1. Uses epsilon-greedy policy: starts with high exploration (ε = 1.0), decays to ε = 0.01.
   2. Initially takes random actions, gradually shifts to maximizing Q-values.

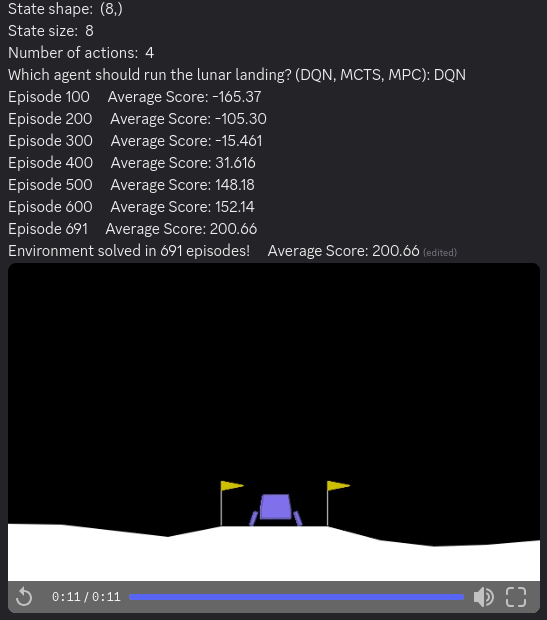


Figure 6: Example run of the DQN/DRL solution result

*Issues with DRL*

The reward function in LunarLander-v3 naturally incentivizes safe landings by providing positive rewards for low velocity, proper orientation near the landing pad, and successful touchdowns, while penalizing crashes and excessive fuel use. The DQN agent's success in solving the environment (reaching an average score of 200+ by episode 691) demonstrates its ability to learn this complex task, but the training curve reveals key challenges. Early episodes show poor performance (average score -165 at episode 100) as the agent explores randomly, followed by gradual improvement as it begins to correlate states with valuable actions. The sharp improvement between episodes 300-500 suggests the agent discovered critical strategies like orientation control and descent throttling, while the final convergence reflects mastery of fine-grained adjustments needed for precise landings. However, DQN still faces limitations: the sample inefficiency of learning from scratch (requiring nearly 700 episodes), sensitivity to hyperparameters (learning rate, discount factor), and potential instability from the moving target problem in Q-learning. Additionally, while DQN adapts well to the simulated environment's variability, real-world lunar landings would require further robustness against sensor noise, hardware delays, and truly novel scenarios not encountered during training, suggesting areas for improvement through techniques like prioritized experience replay or hybrid model-based augmentation.

**Lessons Learned**

Through this project, we have learned about OpenAI libraries such as Gymnasium and how they work to provide users with tools to develop various AI algorithms to solve different scenarios. We also made heavy use of Pytorch and Tensorflow for the neural networks.

Regarding individual algorithms, we also learned about Deep Q-Learning, Monte Carlo Tree Search, and Model Predictive Control; and when and how to apply them depending on the situation. For example, MCTS is best used for scenarios where a competent dynamics model is unavailable and must rely on environmental sampling, particularly in discrete action spaces. In addition, learning how to use multithreading in Python to process multiple batches of data in parallel to handle computationally expensive processes was a significant breakthrough. But MCTS was very limited in the sense that the environment module was impossible to fully replicate. Model Predictive Control works decently well by creating a physical representation of the environment in its neural network, but the dynamics representation is imperfect. Lastly, Deep Q-Learning is best for solving LunarLander overall because it efficiently handles its discrete action space (outputs) and high-dimensional state space (inputs) using neural networks, to come up with an optimized policy at every state.

Plus, as a side note, we learned that it is much easier to visualize the results from a DNN as opposed to a simulation-based algorithm like MCTS. Visualizing MCTS requires piecing together fragments of previous steps and simulations, while visualizing a DDN-based algorithm simply needs the agent to act out its learned policy on a brand-new environment.

This will allow us to accelerate simulations and optimize hyperparameter tuning efficiently in the other methods used for AI development of a solution to the Lunar-Lander\_v3.

**Conclusions and Future Work**

By the end of this project, our team successfully implemented and evaluated three state-of-the-art AI strategies for autonomous lunar landing: Monte Carlo Simulations (MCS), Deep Reinforcement Learning (DRL), and Neural Network-Based Model Predictive Control (NN-MPC). In alignment with the task table and the timeline laid out in Phases 1 and 2, we completed the setup of the LunarLander-v3 environment, implemented the baseline heuristic controller for comparison, and fully integrated the MCTS-based sampling system. We also trained a DRL agent using Deep Q-Network (DQN), which successfully solved the environment by reaching an average score of over 200 by episode 691. In parallel, we designed and tested an NN-MPC system by collecting trajectory data, training a neural dynamics model, and applying a receding-horizon planning strategy. All models were benchmarked using consistent metrics such as landing accuracy, fuel efficiency, and stability, and our findings were documented through code, plots, and this final report.

However, not all intended tasks were fully completed. Although MCTS improved landing scores modestly (from around –190 to +20), it did not reach the success threshold due to limitations in handling the environment's stochastic initialization. Similarly, while NN-MPC was functional, the learned dynamics model struggled to accurately predict the nonlinear behavior of the lander, limiting its effectiveness. Additionally, we were unable to prototype the planned hybrid controller that combines MCTS for sampling, DRL for policy learning, and NN-MPC for trajectory optimization. Other unaccomplished tasks include embedding the models into real-time, flight-grade hardware and conducting hardware-in-the-loop (HIL) testing—both essential for operational deployment but outside the project’s time constraints.

Looking forward, several directions can help advance Apollo AI into a more practical and usable system. First, developing a hybrid AI architecture that merges the strengths of MCTS (broad scenario coverage), DRL (adaptability), and NN-MPC (predictive control) could produce a more robust and efficient controller. Second, improving the fidelity of the learned dynamics model through techniques like domain randomization, adversarial noise injection, or transfer learning could enhance NN-MPC's real-world reliability. Finally, porting the AI models to embedded hardware and validating them via HIL simulation would be critical steps toward creating a deployable system for future lunar or planetary landing missions. Together, these improvements would enable Apollo AI to support advanced space exploration initiatives like NASA’s Artemis program and beyond.

**Supporting Literature**

1. Awasthi, A. (2025, February 4). Evaluating reinforcement learning algorithms for LunarLander-v2. *Research Square*. Retrieved from <https://www.researchsquare.com/article/rs-5939959/v1>
2. Choi, C. Q. (2021, November 10). The moon: A history of exploration. *Space.com*. Retrieved from <https://www.space.com/55-earths-moon-formation-composition-and-orbit.html>
3. Frank, J. D. (2020, November 2). Advanced AI techniques for lunar exploration. *AIAA*. Retrieved from <https://arc.aiaa.org/doi/abs/10.2514/6.2020-4164>
4. Lugo, R. A., Cianciolo, A. D., Dutta, S., Williams, R. A., Green, J. S., Chen, P. T., D’Souza, S., & Pensado, A. R. (2022). Precision landing performance of a human-scale lunar lander using a generalized simulation framework. *NASA Technical Reports Server (NTRS)*. <https://ntrs.nasa.gov/api/citations/20210024283/downloads/nav_moon_scitech2022_v7.pdf>
5. Mali, R., Kande, N., Mandwade, S., Nagre, V., & Sable, N. P. (2023, August 19). Continuous-action policy-gradient actor-critic algorithms for lunar lander. *IEEE*. Retrieved from <https://ieeexplore.ieee.org/abstract/document/10392057>
6. Morgan Stanley. (2020, July 24). *Space: Investing in the final frontier*. Morgan Stanley. Retrieved from <https://www.morganstanley.com/ideas/investing-in-space>
7. National Aeronautics and Space Administration. (n.d.). *Autonomous Landing Hazard Avoidance Technology (ALHAT)*. NASA Space Technology Mission Directorate. Retrieved October 2023, from <https://www.nasa.gov/space-technology-mission-directorate/tdm/autonomous-landing-hazard-avoidance-technology-alhat/>
8. National Aeronautics and Space Administration. (2021, November 9). *Artemis program: Returning humans to the moon*. NASA. Retrieved from <https://www.nasa.gov/specials/artemis/>
9. National Aeronautics and Space Administration. (n.d.). *Beresheet mission*. NASA Science. Retrieved October 2023, from <https://science.nasa.gov/mission/beresheet/>
10. National Aeronautics and Space Administration. (n.d.). *Mars 2020 mission*. Jet Propulsion Laboratory. Retrieved March 2025, from <https://www.jpl.nasa.gov/news/press_kits/mars_2020/landing/mission/>
11. Silvestrini, S., Piccinin, M., Zanotti, G., Brandonisio, A., Bloise, I., Feruglio, L., Lunghi, P., Lavagna, M., & Varile, M. (2022, April). Optical navigation using CNNs for lunar landing. *ScienceDirect*. Retrieved from <https://www.sciencedirect.com/science/article/abs/pii/S1270963822001778>
12. SpaceX. (2023). *Falcon 9*. Retrieved from <https://www.spacex.com/vehicles/falcon-9/>
13. Valeri Polyakov; Cosmonaut who spent a record 437 consecutive days in space and returned looking “big and strong enough to wrestle a bear.” (2022). *Times (London, England : 1788)*, 35-. Retrieved from <https://go-gale-com.ezpv7-web-p-u01.wpi.edu/ps/i.do?p=GIC&u=mlin_c_worpoly&id=GALE%7CA719967362&v=2.1&it=r&sid=summon&ugroup=outside&aty=ip>
14. Vodopija, A., Cork, J. N., & Filipič, B. (2024, July 14). Lunar lander landing site selection benchmark reexamined. *ACM Digital Library*. Retrieved from <https://dl.acm.org/doi/abs/10.1145/3638529.3654229>
15. Wang, K., Chen, Z., Lu, F., & Li, J. (2024, February 20). Neural network-based optimal guidance for lunar landing. arXiv. Retrieved from <https://arxiv.org/abs/2402.12920>