Machine Learning I - Tutorial 1: The Lunar Lander

Feature extraction and manual programming of an agent.

Nicolas Carbone (100591409@alumnos.uc3m.es)
Alex Kotkin (100591107@alumnos.uc3m.es)

February 5, 2026

1 Exercises

1.1 Question 1 - What information is shown in the interface? And in the terminal?

The graphical interface displays a limited amount of information. It displays the environment, including the terrain the lander is descending towards, and the two flags on the ground denoting the desired landing zone. Additionally, it displays information about the lander itself, specifically the positioning of the lander's legs and body, as well as outputs from the side and main rockets whenever they are fired. The terminal displays a much richer set of information. When the LunarLander.py file is initially run, it displays startup information, including the environment name, the value set for the gravity constant, the size of the action space, and a description of the observation space. Additionally at startup, it displays whether the game is running in keyboard or agent mode, and if keyboard mode was selected, it gives a quick summary of the controls available. Throughout the duration of each episode (at each tick), the terminal displays information about the game state. This includes all of the data available in the observation array as well as the current score of the episode and the last action that the lander took. This cycle repeats until the end of the episode, which happens when the lander crashes or lands safely. At this point the terminal states the episode has ended, as well as which of the two outcomes occurred, displays the final score for the episode, and states that a new episode is starting. Finally, once the user quits, it displays the total number of episodes that were completed during the session.

1.2 Question 2 - What happens if you change the value of the GRAVITY variable?

As the value for the gravity variable becomes more negative (stronger gravity) the lander descends more quickly and is unable to rotate as quickly as it is with weaker gravity.

1.3 Question 3 - In your opinion, which data from the observation could be useful to decide what action the lander should take on each tick? Explain your reasoning.

Our implementation of the manual landing function used all eight of the pieces of data found in each observation. The explanations for how and why each piece of data was used follow:

- **X Position:** The X Position of the robot allows us to detect where the robot is in space, how different that position is from the desired position (within the landing zone), as well as how the robot should be moving to achieve that desired position based on the difference between the desired and actual position. Physically, this represents the lander's horizontal distance from the landing pad (which is centered at X = 0), additionally negative values are to the left and positive values are to the right of the landing pad. In the control function, we use this difference between desired and actual position to define a target X Velocity. For example, if the X Position is 0.5, then the function calculates that we should be moving at a velocity of -.25, to eventually center the lander. Additionally, this calculation takes into account the fact that if X Position = 0, then our target X velocity should be 0. This simplified approach works because the position of the landing zone is constant. Ultimately, the X Position observation is used to create a "gentle pull" back to the center throughout the landing process.
- **Y Position:** While this observation is less useful than the X Position, it still allows us to understand how far we are from the desired landing position vertically. This allows us to cut the engines at the very end of the descent, which improves our score slightly as firing the main engine to slow the robot's descent all the way to landing can be expensive. However, we did have to tune the height at which the main engine cuts out, as cutting the main engine too high can cause the lander to crash right at the end.
- **X Velocity:** Physically, this observation represents how fast the lander is moving left (negative values) or right (positive values). In our implementation, the X Velocity reading is compared to the target velocity calculated with the X Position to create an estimate for our X Velocity error (i.e., how far off is our actual X Velocity from the optimal X Velocity that will center the lander). This velocity error is then used as the target angle, which allows the lander to tilt to generate thrust in the direction that will center our robot.
- **Y Velocity:** This reading represents how fast the lander is rising (positive values) or falling (negative values). In our implementation, this reading is used as a threshold for firing the main engine. If the Y Velocity is too negative (falling too quickly), then we fire the main engine no matter what in order to reduce our falling speed. This is essential for slowing the lander to a point where we don't crash every time we make contact with the ground.
- **Angle:** This value represents the current tilt of the lander in radians. Our code uses this value to calculate the angle error, a value that represents the difference between our actual tilt and the desired tilt (calculated with X Velocity). This allows us to decide which of the two side engines to fire in order to properly orient the lander to ensure a centered landing.

- **Angular Velocity:** This value represents how quickly the lander is currently rotating, and in which direction. We use this value as a damper when calculating the angle_todo value, which we then compare to our angle tolerance constant to decide whether we should fire our side engines to correct orientation or not. Without using the angular velocity to dampen this value, we ran into problems where the lander would try to correct its orientation too aggressively, overshoot, and oscillate out of control.
- **Left/Right Leg Contact:** These two observations are useful as they allow us to detect when the robot has touched down, at which point we no longer need to fire any engines, as the main objective of landing safely has been completed. One optimization we could have implemented if we had more time is to use this in conjunction with the X Position observation to see if we are truly in the landing zone before deciding to cut the engines. However, figuring out how to adjust the position safely once on the ground was a little too tricky given the time constraints.

1.4 Question 4 - print_line_data(game) method explanation

Implemented in LunarLander.py. The print_line_data(game) method is primarily responsible for data formatting. It extracts the raw observation values from the GameState object (passed in as the game argument in the main game loop) and uses an f-string to interpolate the variable values directly into a single string suitable for a CSV file (commas between all values, in matching order with the headers of the file). Additionally, it appends the \n character to the end of the string to ensure that the next input to the file begins on a new line. This implementation is then used in the main function to write to the CSV file on every tick. Before the game loop begins, we check if the CSV file exists; if it does, we continue; but if it doesn't, we create the correct file and write the header line to the file. In the main game loop, once each new observation comes in, we use the print_line_data method to create the formatted string and then open the CSV file (which is guaranteed to exist) in append mode (so that we don't override previous data) and write the string returned from print_line_data. This isn't the most efficient implementation, as opening and closing the file on each tick is expensive; it doesn't affect the functionality for our simple lunar lander implementation and it does ensure that if the program is terminated early, we don't lose the data we have collected so far.

1.5 Question 5 - move_tutorial_1(game) method explanation

Implemented in LunarLander.py. This function implements a Rule-Based agent, instead of using any AI techniques, a series of if-then physics based rules are used to control the lander behavior. The overall behavior of the function can be split into two sections, first we calculate where the lander should be relative to it's actual position, then based on that difference, we decide how to fire the engines in order to achieve our desired position. The first step is controlling the horizontal position. This step was the highest priority because the initial X/Y velocities of the lander are randomized, and we were running into failures when the lander was initialized moving very quickly horizontally. The goal of this step is to move the lander closer to X = 0 (the landing pad). This is done by deciding how much the robot should tilt to get to the center. The function calculates a v_x_error (also our target angle for orientation), using the current distance from the center and the current speed. The reasoning behind this is that if the lander is to the right of the landing pad, we wish to fly left, but in order to fly left, we must tilt left. This is then clamped by a MAX_TILT value of 0.2 which we added to prevent the lander from over correcting and subsequently spinning out of control. We use the v_x_error as our target angle, and the next step is figuring out how to achieve this angle using the controls available. We calculate the angle error as the difference between our target angle and the actual angle of the lander, then calculate what our needed correction is in the angle_todo variable. This variable is the weighted difference of the angle error and the angular velocity of the lander, where the angular velocity is a dampening factor to prevent over correcting and losing control. This leads us to the first action we might take, as if the angle_todo is beyond our tolerance, we fire one of the two side engines based on which direction we want to rotate. This is our highest priority action, as it ensures that our lander is both centering itself by adjusting its orientation inwards (which allows us to move towards the center by firing the main engine) as well as preventing itself from overrotating (at which point firing the main engine would cause a loss of control). Once our angle has been corrected, we move onto controlling our vertical speed. If we are falling too quickly we fire the main engine to brake, slowing our fall, and since the previous step oriented us slightly inward, this action also brings us closer to the center of the landing pad. However, if we are low enough, this action is ignored, as firing our main engine all the way to the landing pad is inefficient from a score perspective. Our final control step is an engine cutoff check, if either of our legs is making contact with the ground, we do nothing. This prevents the lander from destabilizing itself after successfully touching down.

1.6 Question 6 - The agent you just programmed does not use any machine learning technique. What advantages do you think machine learning may have to control the Lunar Lander?

In the Lunar Landing environment, using machine learning is advantageous because the agent does not need every rule of the physical system to be hard-coded by hand. Instead of manually defining how to react to gravity, thrust, rotation, and random initial conditions, a learning-based agent can discover effective strategies through experience and repeated interaction with the environment. Over time, the agent can learn patterns such as the most cost-effective trajectory to achieve a safe landing, how to adjust its direction based on a random starting position, and which engine to fire depending on its height and velocity. It can also learn how much thrust and rotational control to apply to avoid over-rotation while still landing accurately. This approach is often more scalable and robust than hard-coding rules, especially in environments with random elements, meaning that factors such as the lander's initial position, orientation, and velocity vary between episodes. Due to this variability, a hard-coded policy may work well in some cases but fail in others that were not explicitly anticipated by the developer. A machine learning agent, however, can learn patterns that perform well on average across many different situations, generalize to unseen states, and ultimately achieve higher and more consistent scores in complex and unpredictable environments.

2 Conclusions

In this tutorial, we explored the Lunar Lander environment by analyzing the information provided through both the graphical interface and the terminal, and by manually programming a rule-based agent to control the lander. By examining the observation space in detail, we identified how each component of the state, such as position, velocity, orientation, and leg contact, can be used to inform control decisions. This understanding allowed us to design a physics-based controller capable of stabilizing the lander, centering it over the landing pad, and reducing its descent speed to achieve safe landings.

While the rule-based approach performed well in many scenarios, it required careful tuning of constraints and thresholds and was sensitive to changes in initial conditions and environment patterns. These limitations highlight the difficulty of manually designing robust control policies in environments with randomness and continuous state spaces. In contrast, machine learning approaches allow an agent to learn effective control strategies directly through interactions with the environment, rather than relying on hand-crafted rules.

Overall, this tutorial provided valuable insight into both manual agent design and the motivation for using machine learning techniques in control problems. The rule-based agent serves as a strong baseline and a useful point of comparison for future learning-based approaches, such as reinforcement learning, which could further improve robustness, adaptability, and performance in the Lunar Lander environment.