VRIJE
UNIVERSITEIT
BRUSSEL

REPORT FOR REINFORCEMENT LEARNING PROJECT

Fundamental Version

Mirbagher Hosseini

June 14, 2024

Sciences and Bio-Engineering Sciences

1 Introduction

In this report, I will dive deep into the fundamental version of the Reinforcement Learning course project. The primary objective of this project is to implement a reinforcement learning (RL) agent capable of performing the LunarLanderContinuous task using OpenAI Gym. This involves leveraging various optimization techniques to enhance the agent's learning process. The project is divided into several tasks aimed at exploring different optimization approaches, specifically Policy Gradient Optimization and Gradient-Free Optimization methods.

In the following sections of this report, I will first investigate the environment used for our simulations. Then, in the "Project Description" section, I will examine all the files developed for this project. Initially, I will explore parametric policies and explain the Policy Gradient method. Following this, I will explain Gradient-Free Optimizations, starting with Zeroth-order Optimization and then a simple version of Population Methods. Next, the file related to the evaluation of Gradient-Free Optimization will be discussed. Subsequently, the files related to plotting, and executing the whole simulation will be explained respectively. In the final section, I will delve into the results obtained from the runs and compare different methods.

2 Environment

2.1 LunarLanderContinuous-v2

LunarLanderContinuous-v2 is an environment in the OpenAI Gym that presents a continuous control challenge where the goal is to land a spacecraft safely on the moon's surface. The environment is designed to simulate the dynamics and physics of a lunar landing scenario, providing a rich platform for testing reinforcement learning algorithms.

2.1.1 Environment Details

Observation Space The observation space is an 8-dimensional vector representing the state of the lander. These 8 scalars provide a full observation of the lander's state:

- x - Horizontal position of the lander.
- y - Vertical position of the lander.
- vx - Horizontal velocity of the lander.
- vy - Vertical velocity of the lander.
- θ - Angle of the lander.
- $vtheta$ - Angular velocity of the lander.
- left_leg_contact - Boolean indicating whether the left leg is in contact with the ground.
- right_leg_contact - Boolean indicating whether the right leg is in contact with the ground.

Action Space The action space consists of two continuous values:

- main engine thrust - A real value in the range $[-1.0, 1.0]$, controlling the vertical thrust.
- directional engine thrust - A real value in the range $[-1.0, 1.0]$, controlling the horizontal thrust and orientation.

These continuous control inputs allow for fine-grained control over the lander's movements, making the task challenging and requiring sophisticated policy networks for optimal performance.

Reward Structure

- The primary objective is to achieve a total reward of at least 200 points.
- Rewards for moving from the top of the screen to the landing pad with zero speed range from 100 to 140 points.
- If the lander moves away from the landing pad, it loses reward.
- Episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 points.
- Each leg ground contact grants +10 points.
- Firing the main engine costs -0.3 points per frame.
- Firing a side engine costs -0.03 points per frame.

2.1.2 Goal

The ultimate goal is to land the spacecraft gently on the designated landing pad marked in the game without crashing.

3 Project Description

In this section, I introduce and delve into the detailed implementation of this project. The project consists of the following files:

- ***policy_NN.py*** - Implements the Neural Network Policy for LunarLanderContinuous-v2.
- ***policy_gradient_optimization.py*** - Implements policy gradient descent for parametric policy optimization.
- ***zeroth_order_optimization.py*** - Implements Zeroth-order Optimization as a gradient-free method.
- ***population_method.py*** - Implements a basic Population Method for gradient-free optimization.
- ***policy_evaluation_utils.py*** - Provides utilities for evaluating gradient-free optimizations.
- ***plotting.py*** - Contains functions for plotting the results.
- ***main.py*** - The main script to run the entire codebase.

In the subsequent sections, I will comprehensively explain each file, diving deep into the details of the code and its implementation.

3.1 Implementation of the Neural Network Policy in `policy_NN.py` for LunarLanderContinuous-v2

In this section, I describe the implementation of the neural network for the LunarLanderContinuous-v2 task, which is developed in the `policy_NN.py` file. The network is designed to map the 8-dimensional state vector of the lunar lander to a 2-dimensional action vector, representing the main and directional engine thrusts. The implementation uses PyTorch, a powerful deep learning framework, and follows a straightforward feedforward architecture.

The network consists of an input layer, a hidden layer, and an output layer. The input layer accepts an 8-dimensional state vector, which includes the lander's horizontal and vertical positions, velocities, angle, angular velocity, and contact indicators for the lander's legs. This input is processed by the hidden layer, which contains 128 neurons and uses the ReLU activation function to introduce non-linearity into the model, allowing it to learn complex representations of the state space. The output layer, which has 2 neurons, applies the Tanh activation function to produce action values bounded between -1 and 1, ensuring that the control inputs for the lander's engines are within the required range.

This neural network policy is implemented in the `PolicyNetwork` class. The `forward` method defines the forward pass of the network, where the input state is sequentially passed through the hidden layer and the output layer. By using this structure, the network can effectively learn a mapping from states to actions, enabling the agent to make informed decisions to control the lunar lander. This implementation is a critical component of this project, providing the foundation for further optimization using policy gradient or gradient-free methods as outlined in the project requirements.

3.2 Implementation of parametric policy (Gradient Descent Policy) in `policy_gradient_optimization.py`

In the `policy_gradient_optimization.py` file, I implement a function to optimize a policy using the policy gradient method [Ko] for the `LunarLanderContinuous-v2` environment. This optimization directly adjusts the policy to maximize the cumulative reward the agent receives. The function, is responsible for training the policy by iteratively improving its parameters based on the rewards obtained from the environment.

After importing the related libraries and setting up the optimizer, which is the Adam optimizer and use to update the policy network's parameters efficiently and ensuring that the policy network is moved to the chosen device, a log file is opened in write mode to record the total reward obtained in each episode.

The core of the function is an episode loop that runs for a specified number of episodes. Each episode represents a single run of the agent in the environment, starting with an initial state provided by resetting the environment. The `done` flag is initialized to `False` to indicate that the episode is active, and various lists are prepared to store log probabilities and rewards.

Within each episode, a state tensor is created from the current state and passed through the policy network to obtain the mean of the action distribution. A normal distribution is then created using this mean and a standard deviation set to one. This dynamic setting ensures that the action distribution is flexible and adaptable to different situations.

The action is sampled from this distribution, converted to a numpy array, and clipped to ensure it falls within the valid action space defined by the environment. This action is executed in the environment, which returns the next state, reward, and status flags indicating whether the episode has terminated or been truncated. The log probability of the taken action is computed and stored for later use in calculating the policy loss.

Rewards obtained from each action are accumulated to track the total reward for the episode. The episode loop continues until the `done` flag is set, indicating the end of the episode either through termination or truncation.

After an episode ends, I calculate the discounted rewards for each time step. This involves summing the future rewards, each discounted by a factor of gamma ($\gamma = 0.99$), to give more importance to immediate rewards compared to distant ones. The discounted rewards are computed as:

$$G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

The discounted rewards are then normalized to have zero mean and unit variance, which helps stabilize the training process by ensuring consistent reward scales.

The policy loss is computed by multiplying the negative log probabilities of the actions by the corresponding discounted rewards (advantages). This loss is then summed to get the total policy loss for the episode. The formula for the policy loss is:

$$L = -\sum_{t=0}^{T} \log \pi(a_t|s_t;\theta) G_t$$

where $\pi(a_t|s_t;\theta)$ is the policy, a_t is the action taken, s_t is the state, and θ are the policy parameters. Gradients of this loss with respect to the policy network parameters are computed using backpropagation, and the optimizer updates the policy network parameters to minimize the policy loss.

Finally, the total reward for the episode is written to the log file, providing a record of performance over time. This iterative process of sampling actions, accumulating rewards, computing losses, and updating the policy continues across multiple episodes, gradually improving the policy's performance in the environment.

3.3 Implementation of Gradient-Free Optimization (Zeroth-order Optimization) in `zeroth_order_optimization.py`

In thhe `zeroth_order_optimization.py` file, I implement Zeroth Order Optimization as a type of Gradient-Free optimization methods which are able to optimize a function without computing its gradient.

1. **Initialization**:
    - I begin by importing essential libraries such as `torch` for tensor operations and `numpy` for random number generation.
    - The `evaluate_policy` function from `policy_evaluation_utils` is imported to assess the performance of our policy given the current parameters. The details of the `evaluate_policy` function from `policy_evaluation_utils.py` are explained in Section 3.5.
2. **Device Setup**:
    - Then I check for the availability of a CUDA-compatible GPU and sets the device accordingly, optimizing computations for either GPU or CPU based on availability.
3. **Opening Log File**:

- A log file is opened to record the returns (rewards) obtained in each episode. This helps in tracking the performance of the policy over time. Through this file I can then plot the outputs.

4. **Main Loop**:

   - The main optimization loop runs for a specified number of episodes. In each episode:
     (a) **Clone Policy Parameters**: Current parameters of the policy are cloned to create a baseline (θ). This ensures the original parameters remain unchanged during perturbations.
     (b) **Generate Perturbations**: A perturbation vector is created with the same shape as the policy parameters, consisting of random values generated using `torch.randn_like`.
     (c) **Perturbed Parameters**: Two sets of perturbed parameters are created: θ_+ (by adding the perturbation vector to the original parameters) and θ_- (by subtracting the perturbation vector from the original parameters).
     (d) **Evaluate Perturbations**: The policy is evaluated using both sets of perturbed parameters. This involves running the policy in the environment and computing the total reward (or score) for each set.
     (e) **Compute Gradient**: A gradient estimate is approximated based on the difference in scores obtained from the two sets of perturbed parameters. This gradient is calculated as the average difference between the positive and negative perturbations, scaled by the perturbation vector.
     (f) **Update Parameters**: The original policy parameters are updated by moving them in the direction of the computed gradient. The update step is scaled by a predefined learning rate to ensure controlled adjustments.

5. **Logging**:

   - After each episode, the maximum return obtained from the two evaluations is logged to a file.

This method allows the policy to iteratively improve without requiring explicit gradient information, making it robust and applicable in scenarios where calculating gradients is not feasible. The zeroth-order optimization thus provides a practical approach to policy optimization by leveraging random perturbations and performance evaluations. Zeroth-order optimization relies on a learning rate, meaning that after evaluating the perturbations, a direction is determined to adjust θ. This direction is then followed using a small learning rate, which can still encounter issues with local minima.

3.4 Implementation of Gradient-Free Optimization (Basic Population Method) in `population_method.py`

The `population_method.py` file implements a basic version of a population-based optimization method. This method optimizes a policy by evaluating multiple perturbations of the policy parameters and selecting the best-performing perturbation. This approach could avoid the issues of local optima and does not require gradient information.

1. **Initialization**:

- Essential libraries such as `torch` for tensor operations and `numpy` for random number generation are imported.
- The `evaluate_policy` function from `policy_evaluation_utils` is imported to assess the performance of the policy given different perturbations of the parameters. The details of the `evaluate_policy` function from `policy_evaluation_utils.py` are explained in Section 3.5.

2. **Device Setup**:
   - The code checks for the availability of a CUDA-compatible GPU and sets the device accordingly to optimize computations for either GPU or CPU based on availability.
3. **Opening Log File**:
   - A log file is opened to record the returns (rewards) obtained in each episode, facilitating the tracking of policy performance over time.
4. **Main Loop**:
   - The main optimization loop runs for a specified number of episodes. In each episode:
     (a) **Initialization**: A list to store scores and perturbed parameters for the population is created.
     (b) **Generate Perturbations**: Multiple sets of perturbed parameters are generated by adding random noise to the current policy parameters. The number of perturbations is determined by the population size.
     (c) **Evaluate Perturbations**: Each set of perturbed parameters is evaluated by running the policy in the environment and computing the total reward (or score) for each set.
     (d) **Select Best Perturbation**: The perturbation that results in the highest score is identified. This set of parameters is considered the best among the population.
     (e) **Update Parameters**: The original policy parameters are updated to the best-performing set of parameters from the population. This ensures that the policy gradually evolves towards better-performing configurations.
5. **Logging**:
   - After each episode, the highest return obtained from the population evaluations is logged to a file. This logging is essential for analyzing the improvement in the policy's performance over time.

The population-based method leverages the power of evaluating multiple policy perturbations in parallel, thereby increasing the likelihood of escaping local optima and finding better policy configurations. By selecting the best-performing perturbation and updating the policy accordingly, this method provides a robust approach to optimizing the policy without relying on gradient information. The iterative nature of the method ensures continuous improvement in the policy's performance across episodes.

3.5 Evaluation of Gradient-Free Optimizations in `policy_evaluation_utils.py`

In the `policy_evaluation_utils.py` file, I develop a function to evaluate the performance of Gradient-Free Optimizations as explained in previous sections. This evaluation function plays an essential role in the optimization processes by providing a measure of how well the policy performs after various perturbations or updates. The primary function in this file, `evaluate_policy`, is responsible for assessing the effectiveness of the policy by running it in the environment and calculating the total reward it accumulates.

To ensure that the evaluation of perturbed policy parameters does not permanently alter the original policy, I first create a backup of the current parameters. This backup is critical as it allows the function to restore the policy to its original state after the evaluation. Once the current parameters are backed up, I replace them with the provided perturbed parameters. This replacement allows the policy to be evaluated with the new set of parameters, simulating how it would perform if these were the actual parameters.

The evaluation process involves interacting with the environment in a controlled manner. The environment is reset to obtain an initial state, providing a consistent starting point for the evaluation. The function then enters a simulation loop, running step-by-step through the environment. During each step, the current state is converted to a tensor format compatible with the policy network. The policy network uses this state to generate an action, which is then executed in the environment. The environment responds with a new state, a reward, and a done flag indicating whether the episode has ended. The reward from each step is accumulated to calculate the total reward for the episode.

To ensure that the evaluation process is efficient and does not run indefinitely, I enforce a maximum number of steps per episode. This step limitation prevents infinite loops and ensures that the evaluation completes within a reasonable timeframe. By simulating the policy's actions in the environment and accumulating rewards, I provide a quantitative measure of the policy's performance.

After completing the evaluation, I restore the original policy parameters from the backup. This restoration is essential as it ensures that the evaluation does not leave any permanent changes to the policy parameters, allowing the optimization process to proceed with the original policy intact.

The final step of the evaluation process is to calculate and return the total reward accumulated during the episode. This total reward serves as the performance score for the evaluated policy, providing a basis for comparing the effectiveness of different sets of policy parameters. This score is crucial for making informed decisions during the policy optimization process, as it directly reflects how well the policy performs in the environment.

3.6 Plotting the Results in `plotting.py`

This file contains three main parts:

1. Functions to Read and Plot the Performance (Cumulative Reward) of each Method:

   This section contains functions to read and plot the results based on the number of episodes and Cumulative rewards. The `plot_multiple_files` function is designed to visually compare the performance of different policy optimization methods by plotting their cumulative rewards over multiple episodes. Each policy is assigned a specific color, which will be explained in detail in the results section. The function reads the log files through the `read_list_from_file` function. The scores are processed to calculate cumulative, which smooth out the performance trends. This approach highlights how each method performs

over episodes, offering a straightforward way to see which methods are improving and which are not. These functions are used in the `main.py` and `read_plot.py` files.

2. Functions to read and plot the running time of each method:

   This section contains functions to read and plot log files related to the running time of each method. These functions help in analyzing and comparing the computational efficiency of the different policy optimization methods.

3. Reading and Plotting the performance (Cumulative Reward) for each method SEPARATELY:

   This section contains a function to plot the results based on the number of episodes and Cumulative rewards for each method separately in different figures.

3.7 Full Code Execution: `main.py`

In this file, I will run the entire code. To do so, I define the environment and then run each policy with different hyperparameters. The hyperparameters and the output of each run will be comprehensively described in the Results section. Here I Also log the running time of each method and save them. After running, the results will be plotted and saved through different functions.

4 Results

In this section, I comprehensively explain the experiments, comparisons, and results obtained. I compare all methods from two perspectives:

1. The performance of each optimization method during episodes in achieving high rewards and learning to successfully land the lunar lander on the landing pad.

2. The running time required for each optimization method.

To compare the methods, I evaluate each method with different hyperparameter values. I compare the Zeroth-Order Optimization with different learning rates (0.001, 0.005, 0.01). Additionally, I run the Population Method with three different sample sizes (10, 20, 30). Finally, I conduct experiments for the gradient descent optimization method with three learning rates (0.001, 0.005, 0.01).

As shown in the figures, I present the results of each algorithm with a specific color scheme. Zeroth-Order Optimization is represented with shades of blue, the basic Population Method with shades of red, and the gradient descent optimization method with shades of green. Within each group, darker colors indicate higher values for hyperparameters.

The computational resources used for this project include an Intel(R) Core(TM) i7-4510U CPU operating at a base frequency of 2.00 GHz, with a boost frequency up to 2.60 GHz, 16.0 GB of RAM, and an NVIDIA GeForce 840M GPU alongside the integrated Intel(R) HD Graphics Family.

4.1 Comparative Performance Analysis

In this section, I compare the results from three perspectives:

1. Comparison of each method with different hyperparameters.

2. Comparison of all methods together.

4.1.1 Comparison of Each Method with Different Hyperparameters

Zeroth-Order Optimization Figure 1 shows the results for Zeroth-Order Optimization with different learning rates. The general conclusion from this plot is that higher learning rates (LR 0.01) lead to better and faster optimization results compared to lower learning rates (LR 0.001 and LR 0.005). Below is a detailed explanation of this figure.

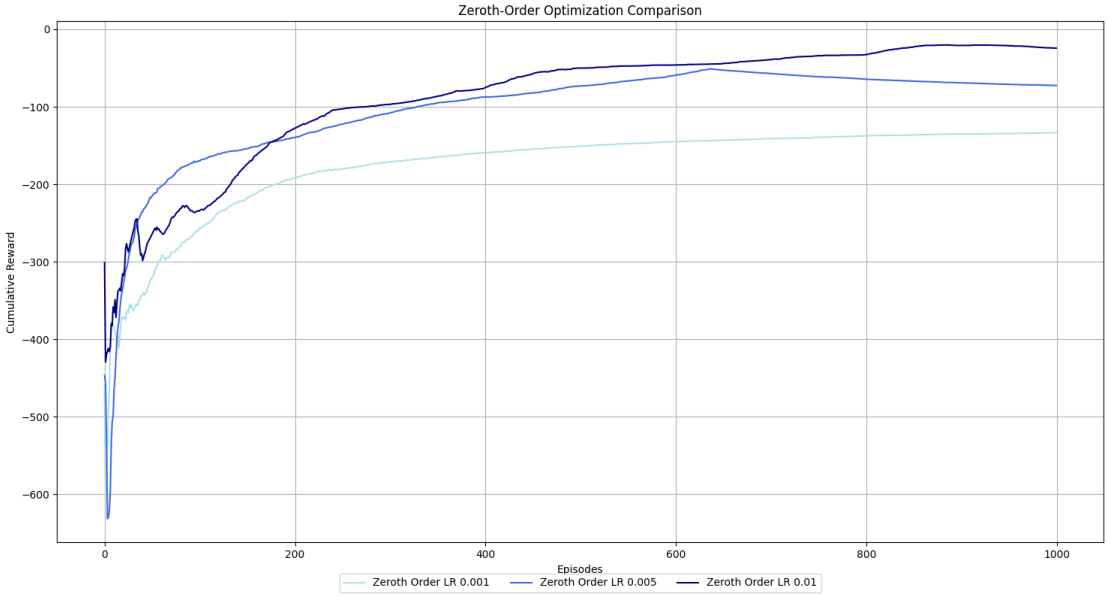

Figure 1: Comparison of Zeroth-Order Optimization based on Different Learning Rates

Observations:

- **Light Blue Curve (Zeroth Order LR 0.001):** This curve shows the performance with a learning rate of 0.001. It starts with lower rewards and exhibits gradual improvement over time but converges to a lower reward compared to the other learning rates.
- **Blue Curve (Zeroth Order LR 0.005):** This curve represents a learning rate of 0.005. It shows a faster improvement compared to the light blue curve and achieves a higher average reward.
- **Dark Blue Curve (Zeroth Order LR 0.01):** This curve corresponds to the highest learning rate of 0.01. It demonstrates the quickest improvement and achieves the highest average reward among the three learning rates.

Comparison:

- The dark blue curve (LR 0.01) has the best performance, reaching the highest average reward, indicating that a higher learning rate is beneficial in this scenario.
- The blue curve (LR 0.005) performs better than the light blue curve but not as well as the dark blue curve, suggesting a moderate improvement.
- The light blue curve (LR 0.001) has the slowest improvement and the lowest final average reward, indicating that the lowest learning rate is less effective.

Population Method Figure 2 presents the results for the basic population method with different sample sizes. The general conclusion from this plot is that larger population sizes (Population Size 30) lead to better optimization results compared to smaller population sizes (Population Size 10 and Population Size 20). Below is a detailed explanation of this figure.

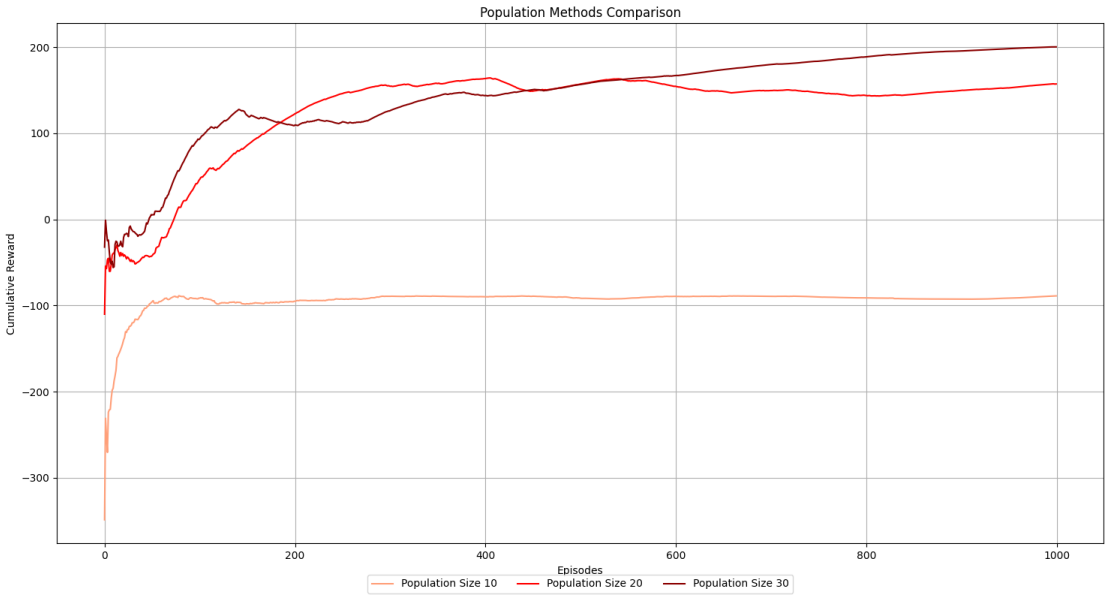

Figure 2: Comparison of Basic Population Method based on Different Population Sizes

Observations:

- **Orange Curve (Population Size 10):** This curve shows the performance with a population size of 10. It starts with lower rewards and exhibits initial improvement but then levels off, showing little improvement and converging to a lower average reward compared to the other population sizes.
- **Red Curve (Population Size 20):** This curve represents a population size of 20. It shows a rapid initial improvement and continues to increase steadily, achieving a higher average reward than the population size of 10 but lower than the population size of 30.
- **Dark Red Curve (Population Size 30):** This curve corresponds to the highest population size of 30. It demonstrates the quickest and most significant improvement, achieving the highest average reward among the three population sizes.

Comparison:

- The dark red curve (Population Size 30) has the best performance, reaching the highest average reward, indicating that a larger population size is beneficial in this scenario.
- The red curve (Population Size 20) performs better than the orange curve but not as well as the dark red curve, suggesting a moderate improvement.
- The orange curve (Population Size 10) has the slowest improvement and the lowest final average reward, indicating that the smallest population size is less effective.

Gradient Descent Optimization Figure 3 shows the results for the gradient descent optimization method with different learning rates. The general conclusion from this plot is that the value for the learning rate should be chosen carefully. The lower learning rate initially shows better results compared to other learning rates, but it drops suddenly, possibly indicating it fell into a local minimum. Although higher learning rates do not work well in this scenario, this conclusion cannot be generalized. It is important to choose the learning rate carefully by experimenting with different values or using other methods, such as Learning Rate Schedules.

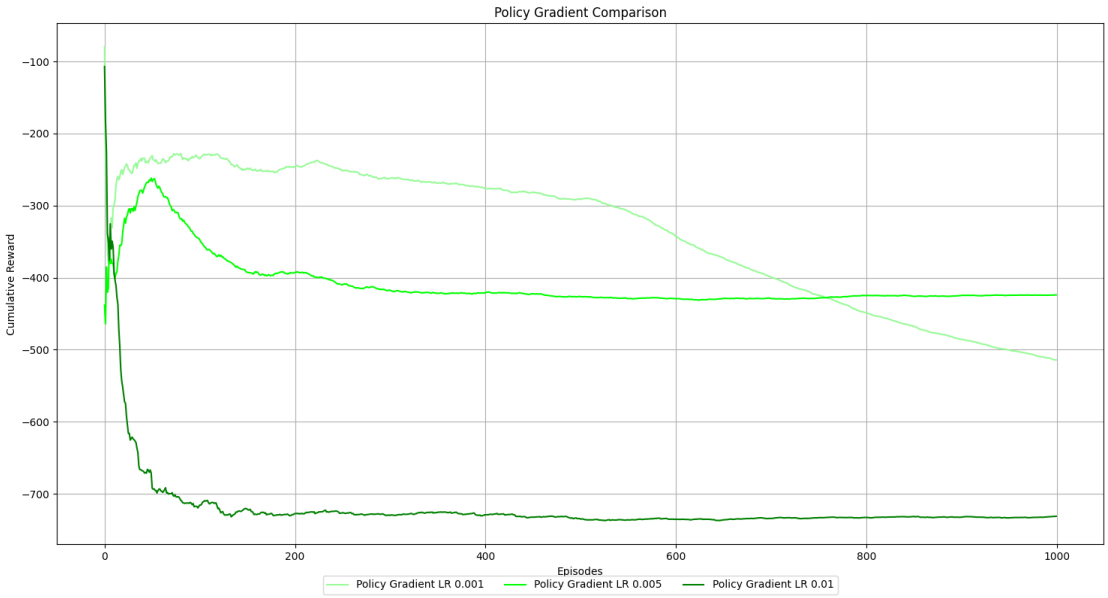

Figure 3: Comparison of Policy Gradient based on Different Learning Rates

4.1.2 Comparison of All Methods Together

Figure 4 shows the results of all methods together. At first glance, it is evident that the Population Methods specially with larger population sizes clearly outperform both Zeroth-Order Optimization and Policy Gradient Methods. They achieve higher and more stable average rewards. Zeroth-Order Optimization shows decent performance but is less stable and effective compared to population methods. Policy Gradient Methods perform the worst, struggling with stability and achieving lower average rewards throughout the episodes.

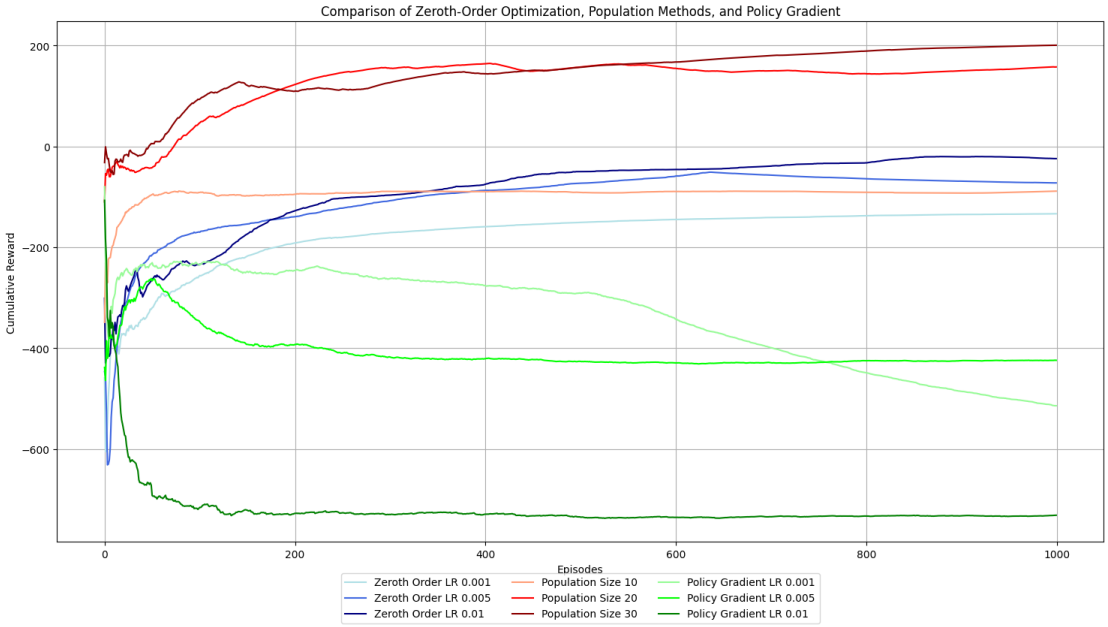

Figure 4: Comparison of Performance All Optimization Methods

4.2 Comparative Running Time Analysis

Figure 5 shows the running time of each method. It is evident that Population Methods take longer to train over the same number of episodes on the same computer. Although Policy Gradient Methods take shorter time compared to Population Methods, they still require more time compared to Zeroth-Order Optimization methods. Regarding running time, Zeroth-Order Optimization performs better than the other methods.

4.2.1 Zeroth-Order Optimization:

- **Zeroth Order LR 0.001:** 56.69 seconds
- **Zeroth Order LR 0.005:** 78.05 seconds
- **Zeroth Order LR 0.01:** 168.17 seconds

These methods have relatively low running times, with the time increasing as the learning rate increases.

4.2.2 Population Methods:

- **Population Size 10:** 252.17 seconds
- **Population Size 20:** 1675.69 seconds
- **Population Size 30:** 2424.99 seconds

These methods have the highest running times, with the time increasing significantly as the population size increases. Population Size 30 takes the longest time, followed by Population Size 20 and Population Size 10.

4.2.3 Policy Gradient Methods:

- **Policy Gradient LR 0.001:** 218.74 seconds
- **Policy Gradient LR 0.005:** 108.26 seconds
- **Policy Gradient LR 0.01:** 131.16 seconds

These methods have moderate running times, with LR 0.005 having the lowest running time among the policy gradient methods.

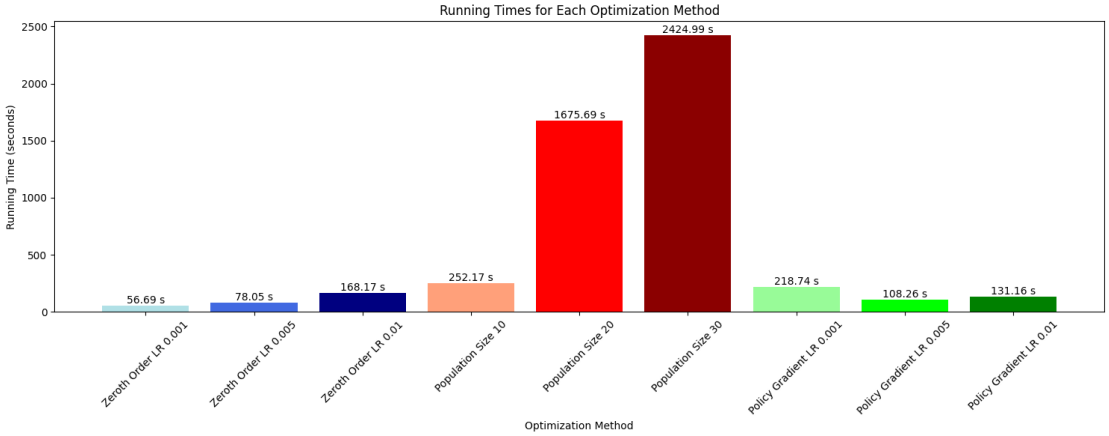

Figure 5: Comparison of Running time of All Optimization Methods

5 Conclusion

In this project, I delved deep into the fundamental concepts of Reinforcement Learning, aiming to develop and compare different optimization methods (gradient policies and gradient-free policies). I not only compared the methods together but also investigated each method with different hyperparameters. The final conclusions are as follows:

- **Population Methods:** While they offer the best performance in terms of optimization, they are the most computationally expensive, especially with larger population sizes.
- **Zeroth-Order Optimization:** These methods provide a good balance between running time and performance, being the most computationally efficient.
- **Policy Gradient Methods:** These methods have moderate running times but show the least effective optimization performance, suggesting they might not be the best choice if both running time and performance are critical factors.

References

[Ko] Dae-hwa Ko Min-soo. *Implementation Policy Gradient*. URL: https://wikidocs.net/175871.