VRIJE
UNIVERSITEIT
BRUSSEL

FUNDAMENTAL PROJECT

Reinforcement Learning

Ardavan Khalij

May 30, 2023

Sciences and Bio-Engineering Sciences

1 Introduction

In this report, I will explain my project and the process of writing and implementing the fundamental project of the course, Reinforcement Learning (RL). However, before that, I will do a brief literature review about similar methods and solutions that have been done before.

What is Reinforcement Learning? A branch of machine learning known as reinforcement learning (RL) studies how intelligent agents should behave in a given environment to maximize the concept of cumulative reward. Along with supervised and unsupervised learning, reinforcement learning is one of the three fundamental machine learning paradigms.

In contrast to supervised learning, reinforcement learning does not need the presentation of labeled input/output pairings or the explicit correction of sub-optimal actions. Instead, the emphasis is on striking a balance between utilization of existing knowledge and exploration of unexplored terrain (Kaelbling et al., 1996).

Critical components of RL include:

1. Agent
2. Environment
3. State
4. Action
5. Reward
6. Policy
7. Value Function
8. Model

2 Literature Review

I will check the literature in this section with the related topics mentioned in the project description. I also briefly go through them.

2.1 Gradient Optimization

As mentioned in the project description, "the objective of Reinforcement Learning is to compute a policy that maximizes the expected discounted return it obtains: policy maximizes the expected (over episodes) sum (over time-steps) of (LearningRate)t rt, with rt the reward obtained at time t." Some approaches has been mentioned in the class and one of them is Policy Gradient. Policy Gradient (Peters and Bagnell, 2010) behave with policy in the same way as parameters and because of that can be handled by Neural Networks. A policy gradient technique is a reinforcement learning strategy that uses a specific variation of gradient descent to directly optimize a parameterized control policy. In contrast to conventional value function approximation approaches, which derive policies from a value function, these techniques fall within the category of policy search strategies that aim to maximize the expected return of a policy within a specified policy class (Peters and Bagnell, 2010).

As mentioned in the project description, Policy Gradient's gradient-following behavior is a concern. A gradient only knows the local effects of the parameters on the policy. Gradient descent encounters issues like becoming stranded in local optima. It can be challenging to decide on the learning rate. As a result, gradient-free methods—methods that don't employ a gradient—can be intriguing (Peters and Bagnell, 2010).

Couple of methods has been mentioned by Peters and Bagnell (2010) of different methods of Policy Gradient.

- **Finite Difference Gradients**: Finite Difference Gradients are one of the most straightforward policy gradient approaches. This approach estimates the gradient by perturbing the policy parameters, similar to what we did in this project but using a gradient. (Fragkiadaki, 2019).

- **Likelihood-Ratio Gradients**: Policy for Likelihood Ratio In Reinforcement Learning, the gradient of the anticipated reward concerning the policy parameters is calculated using gradients. It has sense in domains with ongoing action and state spaces where a significant change in the policy could harm the environment. You can find details about it in the paper by Sutton et al. (2000). You can also find some more details in (Peters and Bagnell, 2010; Fragkiadaki, 2019).

2.2 Gradient-Free Optimization

A function can be optimized using gradient-free optimization techniques without computing its gradient. This means that they enable improving a policy in Reinforcement Learning without calculating the gradient of its parameters. Gradient-free methods consider the parameters and perturb them randomly to get completely different parameters with a better result.

There has been done some work in order to improve these kinds of algorithms. Most of the works focused on improving how the noises are produced (Ebrahimi et al., 2017; Zhai et al., 2021). The approach by Zhai et al. (2021) is exciting. The authors tried to use RL in the process of sampling for the noises as well. They took the Zeroth-Order algorithm (ZO) and tried their idea. Based on their evaluations, they succeeded in making the ZO converge faster. They called this approach ZO-RL.

Population methods are more straightforward and more random. They guarantee the perfect strategy if the algorithm runs forever to avoid getting stuck in a local maximum. A simple one is mentioned in the project description. They are also other algorithms with the same principle. One of them is PBT (Jaderberg et al., 2017).

An asynchronous optimization approach for training neural networks is called population-based training (PBT). It entails training a population of models with various hyperparameters, then choosing which models to copy and which to alter using a selection process. The population's worst-performing models are then replaced with the hyperparameters of the best-performing models through mutation. Until convergence or a predetermined stopping criterion is reached, this process keeps going. PBT enables more effective exploration of the hyperparameter space, which improves performance over training with a single fixed set. So PBT has an approach that combines the best of both worlds (random and Bayesian algorithms).

Another example of these kinds of algorithms is PB2. The new algorithm Population-Based Bandits (PB2), first by Parker-Holder et al. (2021), offers theoretic assurances for effective hyperparameter optimization. The bandit-based exploration technique used in PB2 expands on the PBT strategy by balancing exploration and exploitation. PB2 chooses which hyperparameters to investigate using a multi-armed bandit method, and it modifies the distribution of hyperparameters based on the effectiveness of each configuration. This method makes PB2 more reliable and theoretically grounded than PBT by enabling it to effectively explore the hyperparameter space without relying on custom meta-hyperparameters

3 Project Description

In this section, I will explain my project in detail and separate this part into some subsections. Each of these subsections focuses on one of the files that are in my code. I separated each critical part of the implementation into a separate Python file. There are six files in my implementation, and We will look at each in a separate subsection. These files are separated in a way that they are also separated in terms of concept, so explaining them separately is possible and appropriate.

3.1 Saving File

This is the simplest file in these six files. This file only consists of two functions, save_list_to_file and read_list_from_file. save_list_to_file is a function that gets a string and a list of floats and saves it in a text file. The name of that file is the string input. read_list_from_file gets a string as input and as the file name and returns a list of floats saved in that file.

3.2 Plotting File

This file is also a straightforward part of the implementation. This file contains one function called plot_lists. This function has seven inputs:

1. **list1**: This is the list of rewards over the episodes of the first algorithm.
2. **list2**: This is the list of rewards over the episodes of the second algorithm.
3. **title**: This is the plot's title.
4. **x_label**: This is the label of the x axis of the plot.
5. **y_label**: This is the label of the y-axis of the plot.
6. **legend_labels**: This is the label of each algorithm for the legend of the paper.
7. **each**: This is an integer, and if it is 10, each point in the plot is the mean of rewards of 10 episodes.

3.3 Policy Neural Network File

In this file, I implemented the neural network for the policy. The code of this neural network is illustrated in Figure 1 but before that a breif explenation about the environment.

3.3.1 Lunarlandercontinues

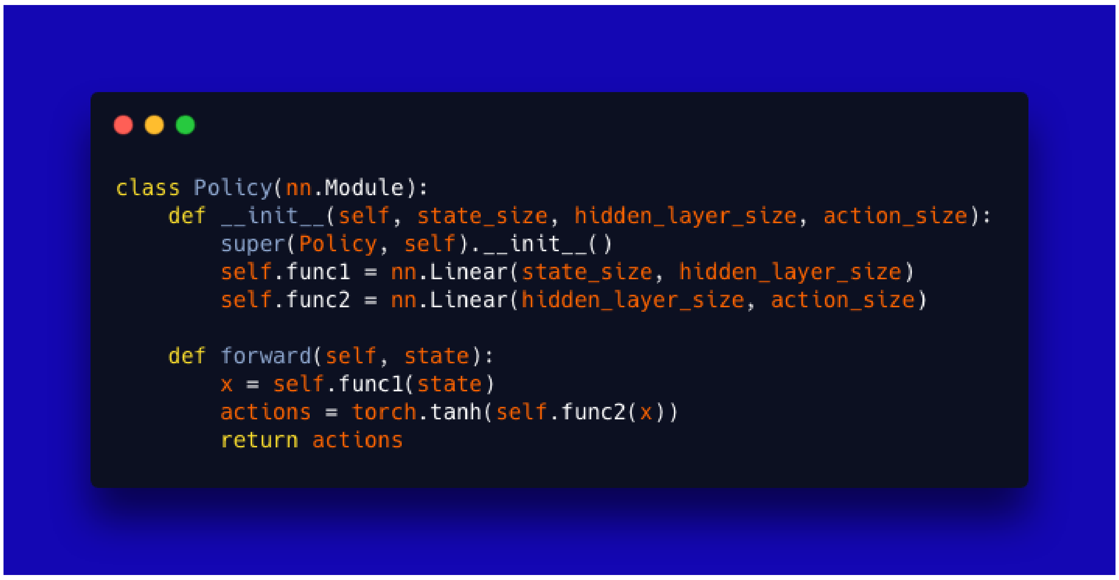
Lunar Lander Continues is one of the popular reinforcement learning environments. It is about learning an agent that is a spacecraft to land on the moon. The OpenAI Gym framework is the solution for access to this environment. The spacecraft has two main control inputs or actions.

The state space of the spacecraft consists of position, velocity, angle, and angular velocity, a boolean that shows whether the legs are in contact with the surface or not, and the current fuel level. The actions space consists of two float numbers, throttle and orientation control. These numbers can be between -1 and 1.

3.3.2 Neural Network

As illustrated in Figure 1, the number of neurons in the hidden layer and the number of inputs and outputs are definable, so it can be anything the user wants. These numbers had been mentioned in the project description that will be mentioned in the section about the main file.

As suggested in the project description, this NN has only one hidden layer, and all the layers are linear. In the end, $Tanh$ has been used for the output to have outputs scaled between -1 and 1. This has been used because of the environment that we are using. The input of the NN is a space with eight values, and the output is an action with two values.



```python
class Policy(nn.Module):
    def __init__(self, state_size, hidden_layer_size, action_size):
        super(Policy, self).__init__()
        self.func1 = nn.Linear(state_size, hidden_layer_size)
        self.func2 = nn.Linear(hidden_layer_size, action_size)

    def forward(self, state):
        x = self.func1(state)
        actions = torch.tanh(self.func2(x))
        return actions
```

Figure 1: Neural network for the policy

NNs have some parameters. These parameters help the NN to predict the required result. In this case, these parameters can play the policy role, and based on them, required actions can be predicted. This NN should be trained on some data to set its parameters to the optimal values.

The main problem in RL is that there is no date for training, so setting these parameters should happen based on the reward that they can produce, and that is what happens in this project and just like methods like PHC or PHC WoLF (Bowling and Veloso, 2001). So two algorithms have been implemented for setting these parameters and will be explained in sections 3.4 and 3.5.

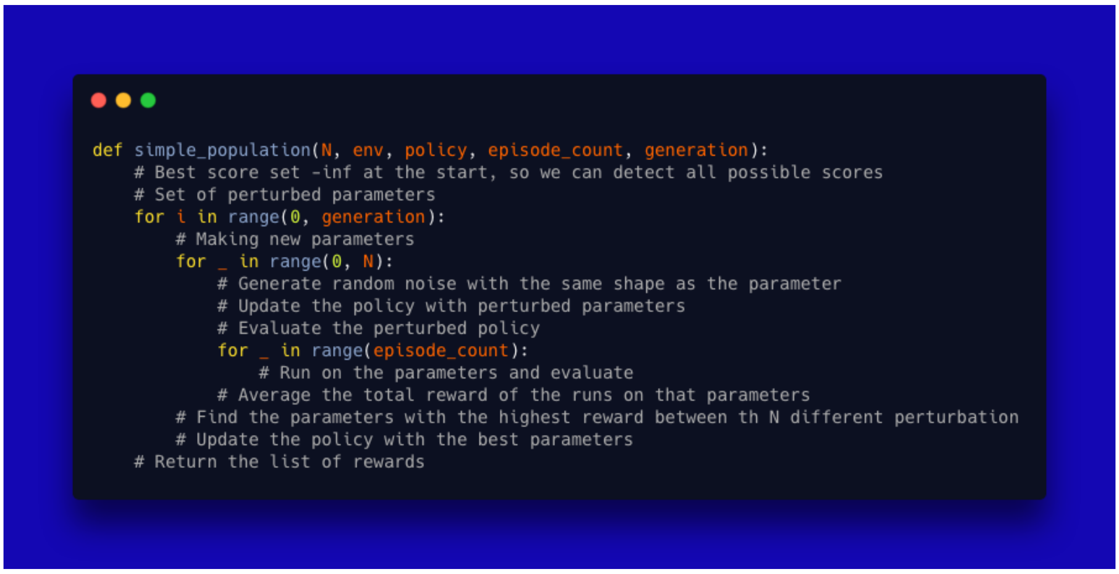
3.4 Simple Population File

This method is one of the population methods that work as described in Figure 2. This figure contains the main structure of the code, and for the details, just the comments.

This algorithm is implemented inside the function simple_population. This function has some inputs:

1. **N**: N is an integer that defines the number of perturbations.
2. **env**: Is the environment. In this case, it is the Lunarlandercontinues.
3. **policy**: Is the policy and its parameters.
4. **episode_count**: This is the number of times the user needs to evaluate each perturbation.
5. **generation**: This is the number of episodes that the user wants to train and update the parameters of the policy.

The steps used in this method are the same as described in the project description. So at first, we produce N perturbations of parameters, then evaluate each perturbation in the environment by performing one or more episodes, and then average the scores. After that, we select the



```python
def simple_population(N, env, policy, episode_count, generation):
    # Best score set -inf at the start, so we can detect all possible scores
    # Set of perturbed parameters
    for i in range(0, generation):
        # Making new parameters
        for _ in range(0, N):
            # Generate random noise with the same shape as the parameter
            # Update the policy with perturbed parameters
            # Evaluate the perturbed policy
            for _ in range(episode_count):
                # Run on the parameters and evaluate
            # Average the total reward of the runs on that parameters
        # Find the parameters with the highest reward between th N different perturbation
        # Update the policy with the best parameters
    # Return the list of rewards
```

Figure 2: Code structure and the descriptions of simple population method

parameters with the largest score and then update the policy for the next generation, then do all of these again.

3.5 Zeroth Order File

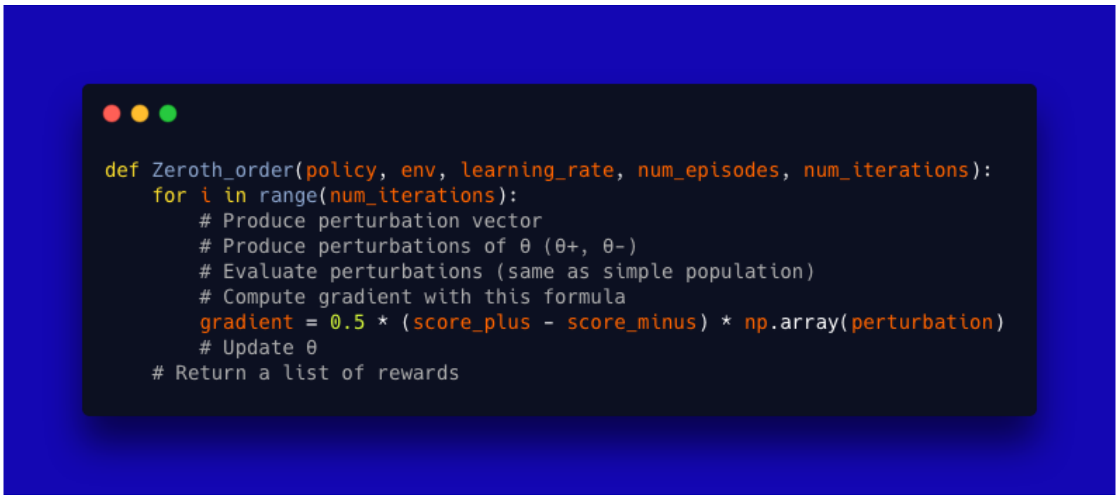
This method is one of the gradient free methods that work as described in Figure 3. This figure contains the main structure of the code, and for the details, just the comments.

This algorithm is implemented inside the function Zeroth_order. This function has some inputs:

1. **learning_rate**: This is the size of the step that the parameters change.
2. **env**: Is the environment. In this case, it is the Lunarlandercontinues.
3. **policy**: Is the policy and its parameters.
4. **episode_count**: This is the number of times the user needs to evaluate each perturbation.
5. **num_iterations**: This is the number of episodes that the user wants to train and update the parameters of the policy.

The steps used in this method are the same as described in the project description. First, we made a loop and put the steps inside it. The first step is to produce a perturbation vector with the same shape as the parameters vector (just like the simple population method); then, we make the perturbations of the parameters and call them theta+ and theta-. The next step is to evaluate each perturbation and calculate a gradient with the formula mentioned in the project description. In the end, we add the gradient vector to the parameters, and we also use the learning rate and multiply the gradient vector values by that number to change the size of the steps. We repeat all these steps.

In this file, there are some other functions as well that are mainly for making the coding cleaner:



```python
def Zeroth_order(policy, env, learning_rate, num_episodes, num_iterations):
    for i in range(num_iterations):
        # Produce perturbation vector
        # Produce perturbations of θ (θ+, θ-)
        # Evaluate perturbations (same as simple population)
        # Compute gradient with this formula
        gradient = 0.5 * (score_plus - score_minus) * np.array(perturbation)
        # Update θ
    # Return a list of rewards
```

Figure 3: Code structure and the descriptions of zeroth order method

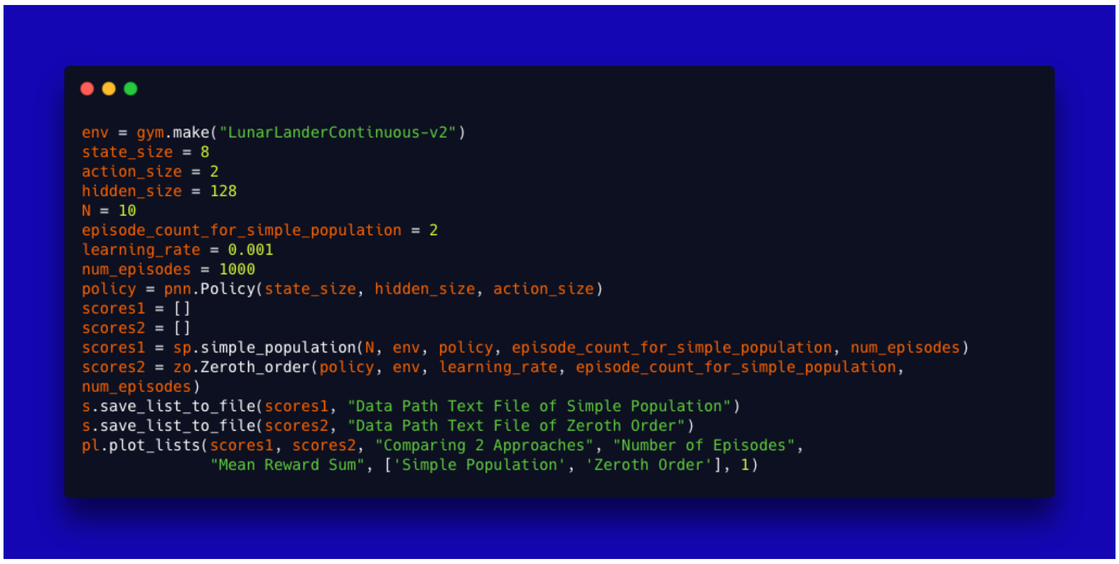
- **get_parameters**: This function gets the policy and returns a list of tensors containing the NN parameters.
- **produce_perturbation**: This function gets the parameters list and returns a perturbation with the same shape.
- **add_list**: This function gets two lists of tensors and does the sum operation on them. I had to write a function for this because it was not a vector but a list of vectors.
- **diff_list**: This function gets two lists of tensors and does the diff operation on them. I had to write a function for this because it was not a vector but a list of vectors.
- **set_parameters**: This function gets the new parameters and the policy and then updates the policy parameters with the new parameters.
- **evaluate_policy**: This function gets the policy, number of episodes for evaluating, env, and the parameters and evaluates the parameters.

3.6 Main File

In this file, we just used all of the other files. We give suitable values to the different variables. As mentioned before, the state size is 8, and the action size is 2. The hidden layer's size is also set to 128 based on the project description's advice. N is the same N for the simple population, and episode_count_for_simple_population is the same as the number of evaluations of each perturbation. The rest of the variables and their values are illustrated clearly in Figure 4. After assigning the values, both algorithms will run for 1000 episodes, and their results will be saved to a text file. The plotting happens.

4 Results & Conclusion

As mentioned in Section 3, both algorithms have been running for 1000 episodes and evaluated each step 2 times. The learning rate for this result is 0.001. Each value for plotting is set to be



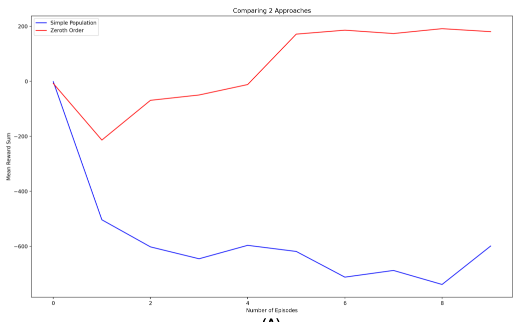
```
env = gym.make("LunarLanderContinuous-v2")
state_size = 8
action_size = 2
hidden_size = 128
N = 10
episode_count_for_simple_population = 2
learning_rate = 0.001
num_episodes = 1000
policy = pnn.Policy(state_size, hidden_size, action_size)
scores1 = []
scores2 = []
scores1 = sp.simple_population(N, env, policy, episode_count_for_simple_population, num_episodes)
scores2 = zo.Zeroth_order(policy, env, learning_rate, episode_count_for_simple_population,
num_episodes)
s.save_list_to_file(scores1, "Data Path Text File of Simple Population")
s.save_list_to_file(scores2, "Data Path Text File of Zeroth Order")
pl.plot_lists(scores1, scores2, "Comparing 2 Approaches", "Number of Episodes",
              "Mean Reward Sum", ['Simple Population', 'Zeroth Order'], 1)
```

Figure 4: Code structure of main

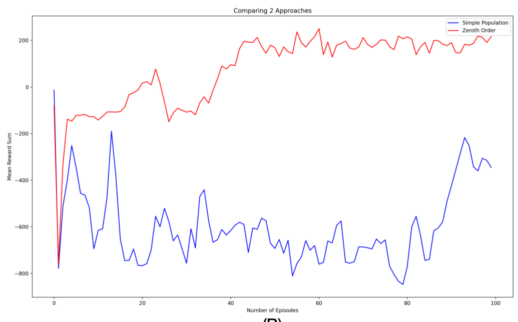
1, 10, and 100, and all of these three are available in this section to show the progress of the algorithms perfectly. The results are illustrated in Figure 5, and for higher quality images, you can check the plots in the code folder and this article's appendix.

As illustrated in Figure 5, the zeroth order is better. It converges to a high value. On the other hand, the simple population method is not converging. Of course, the zeroth order algorithm can get stuck in the local maximum.

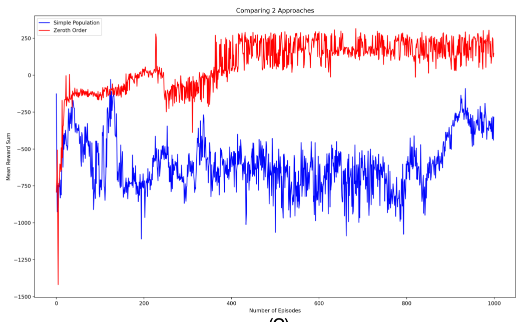
In conclusion, the zeroth order algorithm is less time complex and a better one that converges to a maximum value; however, there is a chance that that maximum would be a local maximum, not a global maximum. On the other hand simple population algorithm is the slower method that does not converge but also does not get stuck in a local maximum because it has a stronger random factor than the other one.



(A)



(B)



(C)

Figure 5: Comparison of simple population and zeroth order method. (A) is for $each = 100$, (B) is for $each = 10$, and (C) is for $each = 1$

5 Appendix

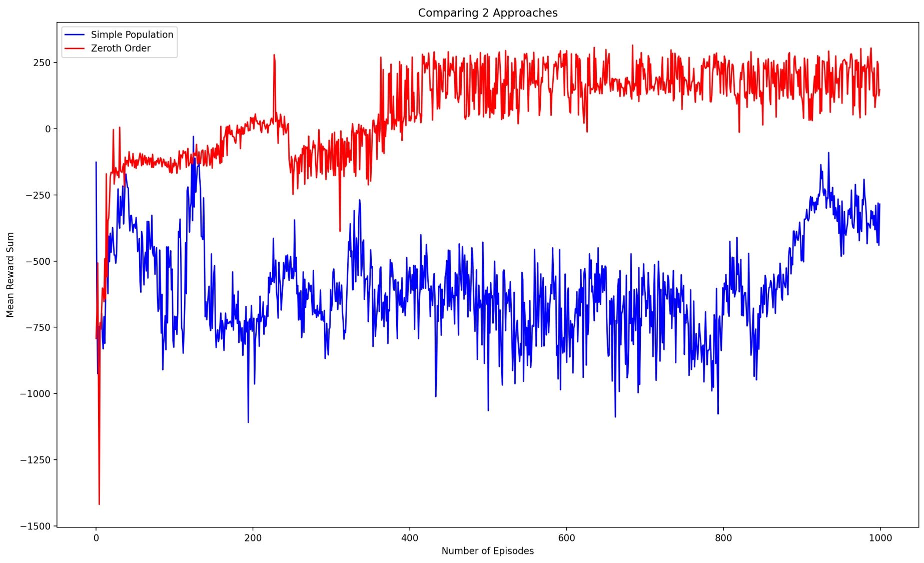


Figure 6: Comparison of simple population and zeroth order method for $each = 1$

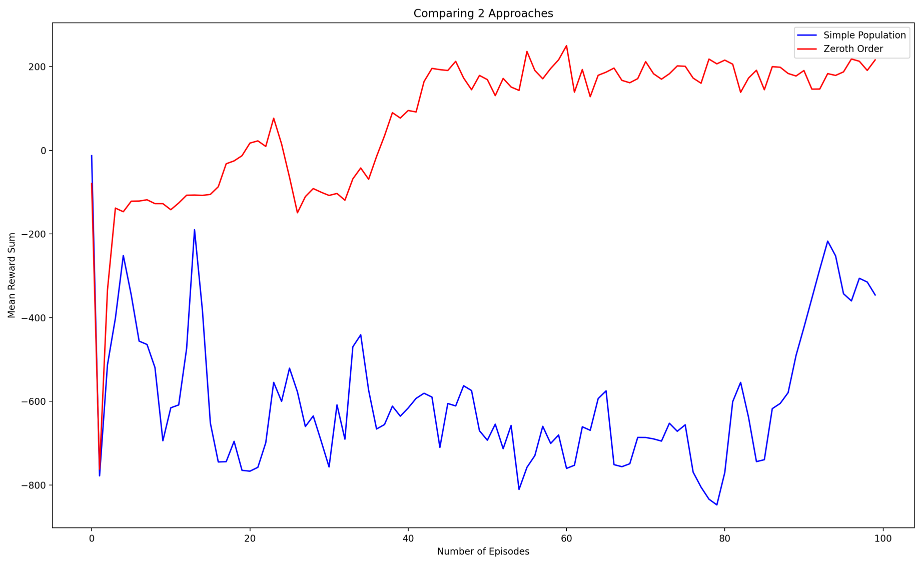


Figure 7: Comparison of simple population and zeroth order method for $each = 10$

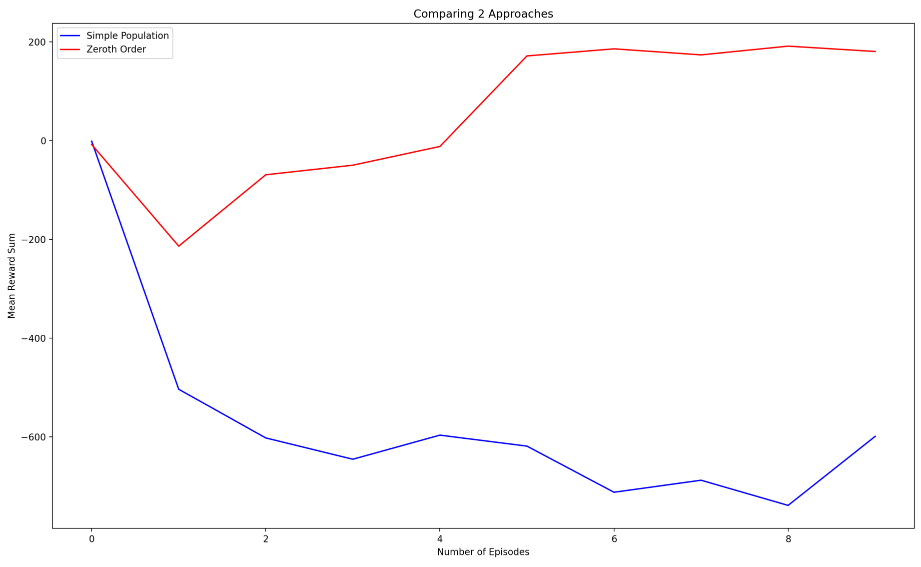


Figure 8: Comparison of simple population and zeroth order method for $each = 100$

References

M. Bowling and M. Veloso. Rational and convergent learning in stochastic games. *IJCAI International Joint Conference on Artificial Intelligence*, 08 2001.

S. Ebrahimi, A. Rohrbach, and T. Darrell. Gradient-free policy architecture search and adaptation. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 505–514. PMLR, 13–15 Nov 2017. URL `https://proceedings.mlr.press/v78/ebrahimi17a.html`.

K. Fragkiadaki. Policy gradients. `https://www.andrew.cmu.edu/course/10-403/slides/S19_lecture11_PG.pdf`, 2019. CMU 10-403.

M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu. Population based training of neural networks, 2017.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey, 1996.

J. Parker-Holder, V. Nguyen, and S. Roberts. Provably efficient online hyperparameter optimization with population-based bandits, 2021.

J. Peters and J. A. Bagnell. *Policy Gradient Methods*, pages 774–776. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_640. URL `https://doi.org/10.1007/978-0-387-30164-8_640`.

R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063. MIT Press, 2000.

Z. Zhai, B. Gu, and H. Huang. Learning sampling policy for faster derivative free optimization. *CoRR*, abs/2104.04405, 2021. URL `https://arxiv.org/abs/2104.04405`.