VRIJE
UNIVERSITEIT
BRUSSEL

PROJECT - FUNDAMENTAL

Reinforcement Learning Seminar

Lars Bonnefoy

August 16, 2025

Sciences and Bio-Engineering Sciences

1 Project Overview

The aim of this project is to solve the LunarLander-V3 (Klimov 2025) problem using **Gradient Free** Reinforcement Learning (RL) methods.

1.1 Environment: LunarLander-V3

The aim of this task is to safely land between the two colored poles (See Fig. 1). Main and lateral throttle activity are represented by red dots flowing out of the lander. Landing spot and throttle activity are considered when computing the rewards (Section 1.1.3 for a full overview of rewards). Two distinct versions of this problem exist: the discrete version, which limits throttle control to binary states (active or inactive), and the continuous version, which enables fine-grained control through exact throttle level selection.

We will work on the continuous formulation. As noted by Lillicrap et al. 2019 this poses some challenges, as standard Deep Q-Networks (DQNs) are not suitable for continuous action spaces. DQN can handle discrete environments by learning Q-values for each possible action and state pair and select the best action according to that value. However, continuous action spaces have infinitely many possible values, requiring long iterative optimization at each decision step to find the best action. Continuous spaces necessitate alternative approaches such as actor-critic methods like Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC).

This work, however, uses a gradient-free approach where we rely on parameter perturbation without explicit gradient computation. Its benefits will be highlighted in a later section.

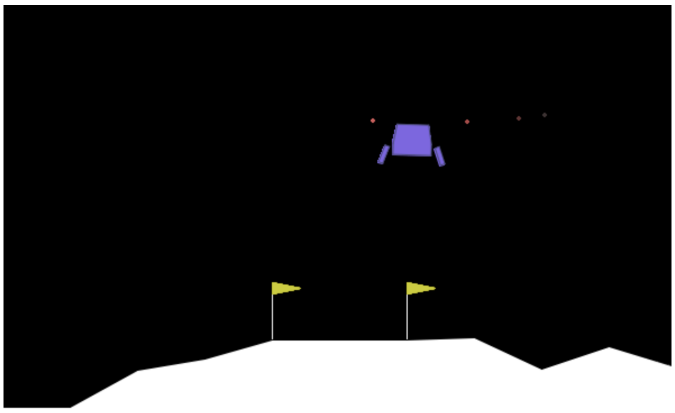

Figure 1: Lunar Lander environment from Gymnasium (Klimov 2025).

1.1.1 Observation space

The observation space is comprised of eight variables that are:

- Coordinate of the x position.
- Coordinate of the y position.
- Linear velocity in the x direction.

- Linear velocity in the y direction.
- The angle with respect to the horizontal plane.
- The angular velocity.
- Boolean representing contact of the *left* leg with the ground.
- Boolean representing contact of the *right* leg with the ground.

1.1.2 Action space

In the continuous version of our environment, the action space consists of two inputs $a_1, a_2 \in \mathbb{R}$:

1. Throttle of the main engine. The main engine is turned off if main < 0 and it scales affinely from 50% to 100% when $0 \leq \text{main} \leq 1$.
2. Throttle of the lateral boosters. If $-0.5 < \text{lateral} < 0.5$ the lateral boosters will be off. If lateral $\in [0.5; 1]$ the right boosters will fire, with the throttle scaling affinely from 50% to 100%. Conversely, if lateral $\in [-1; -0.5]$ the left booster will fire.

1.1.3 Rewards

The rewards are shaped to take into account different requirements. We want our shuttle to land safely between the two flags as fast as possible, without using too much fuel. This is why the reward is increased or decreased at each step depending on:

- The distance to the landing pad. Closer is better.
- The speed the lander is moving. Higher is better.
- The angle of the lander. Horizontal is better.
- Each leg touching the ground.
- The firing of the side engines. Less is better.
- The firing of the main engine. Less is better.

Each episode receives an additional reward (-100, +100) if the lander crashes or lands safely. An episode is considered a solution if it scores at least 200 points, which means that safe landing alone is not sufficient, and some other reward sources have to be taken into account.

2 Gradient Free Methods

Gradient-Free optimization methods in RL find the best parameters for a policy without computing its gradient. We can find a good approximation π^*_θ of the optimal policy π by perturbing the parameters θ by some random value and evaluating this perturbed policy in the environment. In case this perturbation provides an improvement over the current policy it is taken into account to adjust the parameters θ.

Gradient-Free optimization techniques provide several advantages:

- **Exploration**: Parameter perturbation introduces stochasticity, enabling model exploration, with the perturbation's standard deviation affecting exploration strength.

- **Non-differentiable Functions**: Can optimize functions which are discontinuous or non-differentiable.
- **Robustness to local optima**: Gradient-Free population methods are insensitive to local optima.
- **Parallelization**: Multiple perturbations can be computed and evaluated in parallel.

2.1 Policy

We implemented a parametric policy θ by using a Pytorch Module. It is made up of a simple Feed-Forward Network with 8 input units, one for each observable state, 128 hidden neurons and an output layer with 2 output units representing actions a_1 and a_2. We use one ReLU activation layer after our hidden units. Finally, the output is passed through a hyperbolic tangent function to constrain our output values to the domain $a_1, a_2 \in [-1; 1]$.

2.2 Zeroth-order Optimization

Zeroth-order optimization produces two perturbations of θ with opposite signs. The policy π_θ is then updated by moving θ in the direction of the perturbation that yielded the best reward. The evaluation of the perturbed policy can be averaged over m episodes to reduce the dependence on environmental stochasticity. The full algorithm is as follows:

Algorithm 1 Zeroth-order Optimization

```
1:  **Input:** Initial policy parameters $\theta$, learning rate $\alpha$, number of episodes $E$, standard deviation $\sigma$
2:  **for** $i = 1$ to $E$ **do**
3:      Sample a perturbation vector $\delta \sim \mathcal{N}(0, \sigma^2)$ of same shape as $\theta$
4:      Compute perturbations: $\theta^+ = \delta$, $\theta^- = -\delta$
5:      Inject $\theta + \theta^+$ into policy $\pi$, run $m$ episodes and compute average return $R^+$
6:      Inject $\theta + \theta^-$ into policy $\pi$, run $m$ episodes and compute average return $R^-$
7:      Estimate gradient: $g = \frac{1}{2}(R^+ - R^-) \cdot \theta^+$
8:      Update parameters: $\theta \leftarrow \theta + \alpha \cdot g$
9:  **end for**
10: **Output:** Optimized policy parameters $\theta$
```

2.2.1 Hyperparameters

Three key hyperparameters influence training to find the optimal policy. They are:

1. **Width of the hidden layer**: This influences the number of total trainable parameters θ. Too few parameters will lead to under-parameterization, which will make it difficult to learn a new policy. An excessive number of parameters can result in over-parameterization, potentially giving rise to issues such as overfitting, longer training times, and difficulties to find local optima.

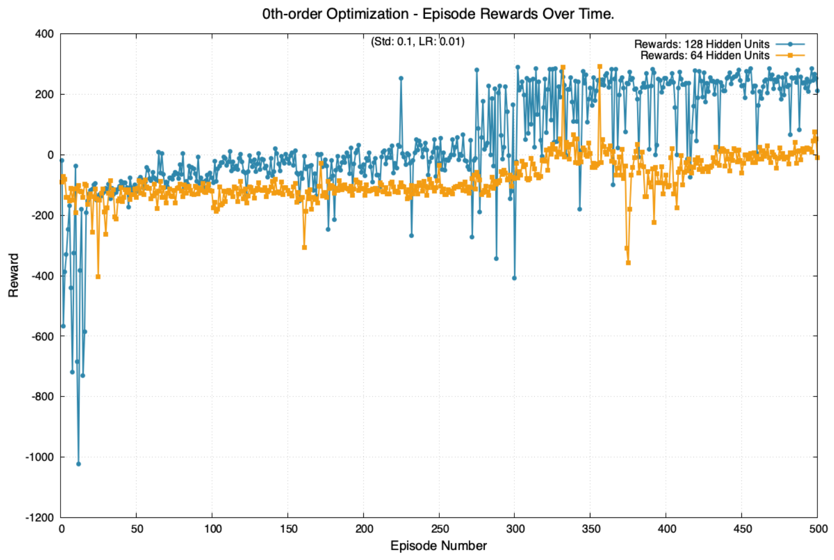

Figure 2: Rewards with respect to the number of units in the hidden layer.

By comparing 64 and 128 hidden units (Fig. 2) with identical standard deviations and learning rates, we can see that perturbations on a bigger network produces higher variance with respect to the computed rewards per episode. It, however, learns a policy which leads to success in most cases as they score above 200 while our smaller network struggles to produce successful episodes.

2. **Standard deviation of perturbations** (σ): This effectively influences the exploration versus exploitation trade-off. A high standard deviation will produce bigger perturbations, which could lead our policy to discover higher rewarding states. It also makes it easier to avoid poor local optima. However this can also backfire as our network can escape a good optimum and find itself with a worse policy.

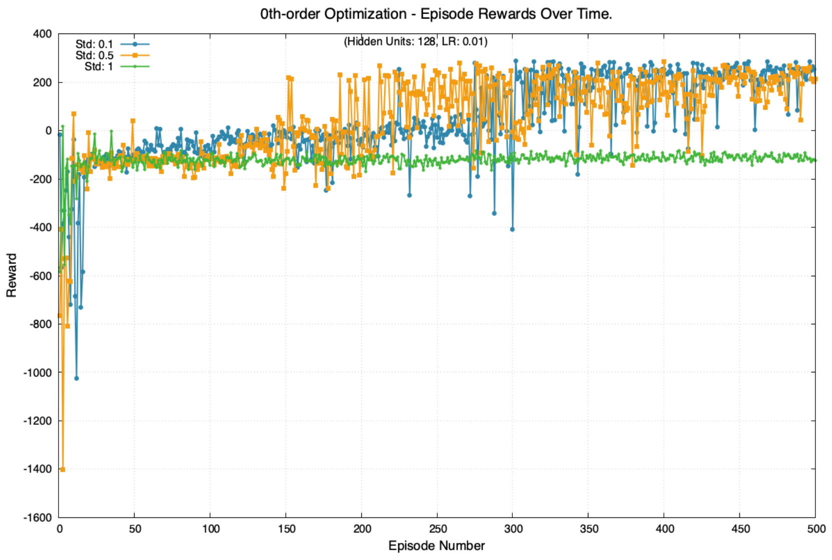

Figure 3: Rewards with respect to the standard deviation of our perturbation generating process.

As expected, changing the standard deviation of our perturbation vector changes the way our model learns (Fig. 3). A high standard deviation ($\sigma = 1$) is never able to learn the right policy as perturbations are too high to converge to some local optimum. When comparing $\sigma = 0.1$ and $\sigma = 0.5$ we can see that a higher value (orange) finds a policy resulting in successful episodes (reward > 200) earlier (around episode 200). The lower σ value (blue) is slower to find a correct policy resulting in successes. Additionally a smaller σ results in lower variation around the current parameters θ which in turn tend to produce lower variations in the rewards meaning that the training algorithm is less likely to revert to a suboptimal policy once a good one is found.

3. **Learning rate (LR)**: This parameter changes how much the optimization algorithm moves the weights in a given direction. With a higher learning rate our network tends to take bigger steps toward an optimum but could jump around that local optimum while a lower learning rate makes convergence slower but could approach a local optimum closer. The learning rate and perturbation standard deviation have an opposite relationship in hyperparameter tuning: we can either use high standard deviation with a low learning rate to explore broadly while taking conservative update steps, or use low standard deviation with a high learning rate to make bigger updates based on precise gradient estimates. This can be seen in Fig. 4 where we compared σ and a learning rate which differed by one order of magnitude. Both sets of hyperparameters approached local optima but a higher σ seems to make our policy return to less desirable states more often.

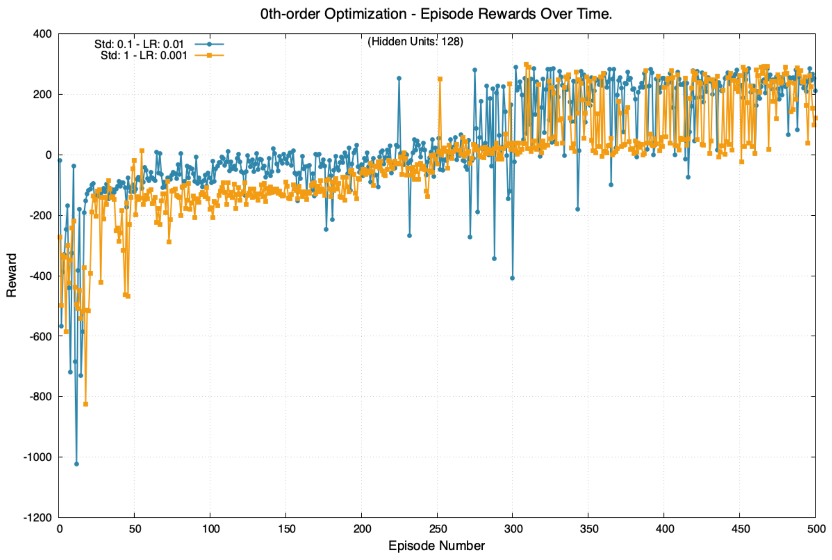

Figure 4: Rewards with respect to opposite learning rates and σ.

2.2.2 Improving Convergence

As shown by our previous set of results on hyperparameter tuning, one of the main issues of our current approach is to stabilize our parameters θ once a good local optimum is found. This happens because our model never reduces its tendency to explore (i.e., it always keeps the same σ for its noise-generating process). Therefore, we improve convergence to a good policy and avoid exiting a good local optimum by gradually reducing the magnitude of σ in a process similar to simulated annealing. Simulated annealing is an optimization technique which enables leeway in the beginning of the learning process to avoid local minimus. It authorizes some suboptimal moves at the start but gradually reduces the probability that bad moves are selected the further the algorithm is trained (Russell et al. 2020).

Our technique consists in applying a rolling average over the past l evaluation rewards. When this average reaches some predefined target, we multiply σ by some decay $d \in [0; 1]$. As long as the target value is met, σ continues to decrease. Conversely, if the target is not reached we increase σ by dividing by d. The size l of the rolling average window changes the sensitivity of our decay applications, while d controls the strength of our decay.

In Fig. 5 we compared a training run with a constant $\sigma = 0.5$ and our weight decaying method. We used a window size $l = 3$, a decay $d = 0.95$ and set as a target a reward of 100 (other parameters are hiddenUnits = 128 and lr = 0.01). The decay could not decrease our initial σ by a factor of more than 20 ($\sigma_{min} = 0.025$) to avoid complete collapse of exploration. By averaging the rewards of the last 100 episodes we can see that our weight decay method does stabilize learning as we obtain a higher average (272 for an adaptive σ and 172 for constant σ).

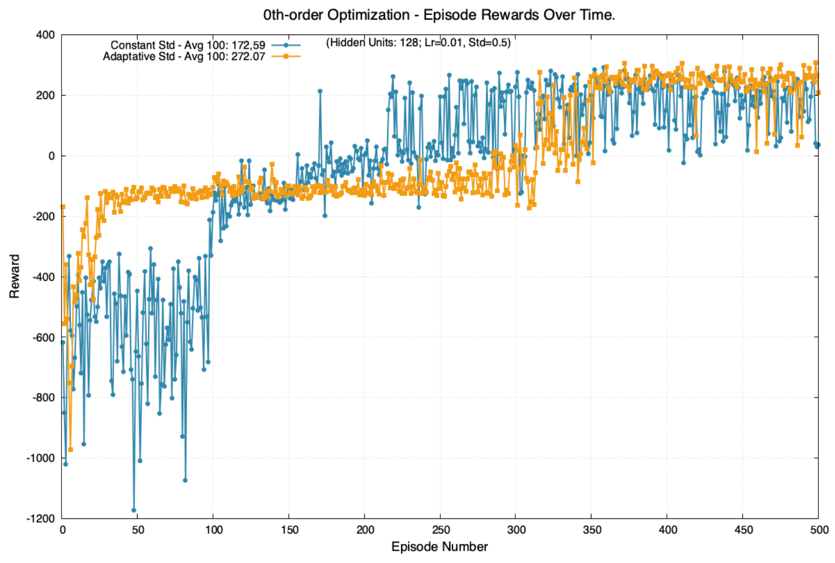

Figure 5: Rewards with respect to exploration, 0th order optimization

2.3 Population Methods

Population methods generate a number of random perturbations n which are evaluated. The perturbation that yields the highest reward is then retained. Given a sufficiently large number n, our algorithm will try a sufficiently large number of combinations to achieve convergence. The full algorithm is as follows:

Algorithm 2 Population Methods

```
1:  **Input:** Initial policy parameters $\theta$, number of perturbations $n$, number of training iterations
    $e$, standard deviation $\sigma$
2:  **for** $i = 1$ to $e$ **do**
3:      **for** $j = 1$ to $n$ **do**
4:          Sample perturbation $\delta_j \sim \mathcal{N}(0, \sigma^2)$
5:          Generate perturbed parameters: $\theta_j = \delta_j$
6:          Inject $\theta + \theta_j$ into policy $\pi$, run $m$ episodes and compute average return $R_j$
7:      **end for**
8:      Identify $j^* = \arg\max_j R_j$
9:      Update parameters: $\theta \leftarrow \theta + \theta_{j^*}$
10: **end for**
11: **Output:** Optimized policy parameters $\theta$
```

2.3.1 Hyperparameters

Population methods have two similar hyperparameters to 0th-order optimization, namely the width of the hidden layer and the standard deviation of the perturbation. Their effects have been discussed in section 2.2.1. The parameter replacing the learning rate is n the number of perturbation generated from which the best is selected. A higher n increases the probability of finding weights which improve the policy at the cost of training speed. A solution to this is to use parallel evaluation of the n policies.

2.3.2 Improving Convergence

As for 0th-order optimization, we applied a decreasing σ method in order to stabilize learning once good rewards are reached. From Fig. 6 we can see that our adaptive σ has a higher average reward over the last 100 episodes (225.32), but takes more time to reach successful episodes than a constant σ. This can however be related to the stochastic nature of exploration rather than our adaptive σ.

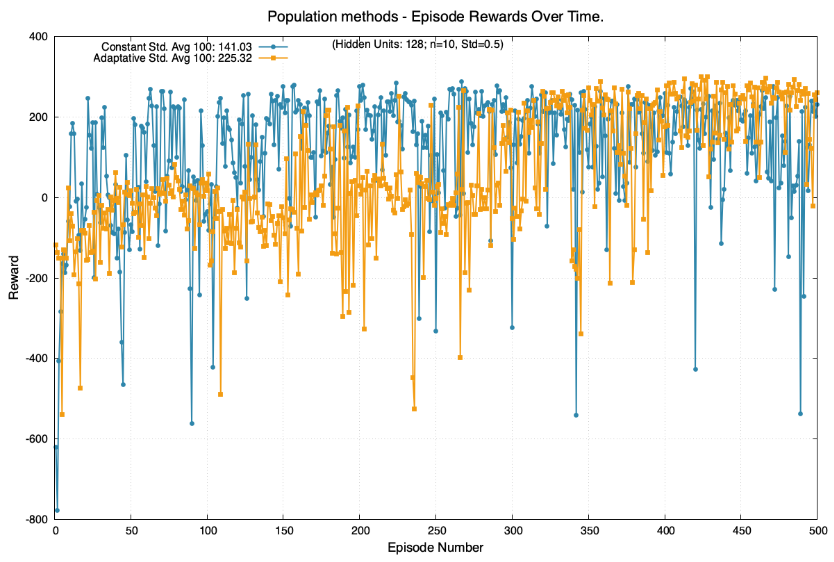

Figure 6: Rewards with respect to exploration, population methods

A key advantage of population methods over zeroth-order optimization is faster convergence to successful policies, requiring fewer episodes to achieve good performance. However, this comparison must be contextualized: population-based methods evaluate more candidates per episode ($n = 10$ in our case versus $n = 2$ for zeroth-order), effectively exploring a broader parameter space within each episode.

2.3.3 Parallelization

As mentioned earlier, one of the advantages of population methods is that they are suitable for parallelization. We can run our evaluation for the n perturbations in parallel (and additionally evaluate a perturbed policy m times and take the average reward over those m episodes). Both of those numbers influence the number of computed evaluation episodes per training episode of our policy as we need to evaluate n perturbed policies m times. The total number of episodes executed to find our optimal policy is therefore $m * n * e$.

However, our parallel implementation did not improve training times, largely due to the relatively small scale of our network. This makes for rapid policy evaluation, but still incurs the additional overhead attributed to parallelization processes, especially in copying the policy in each n perturbation to avoid race conditions. Parallelization became more effective with higher m and n

count but at that stage the evaluation times were too long and good results were obtained with simpler serial implementations.

3 Comparison with baseline

In order to compare the effectiveness of our gradient-free methods we compare them to a simple baseline model implementing policy π_{random} where actions are taken at random and the model does not have any learnable parameters. Zeroth-order and population were trained for 500 episodes and their performance compared for 100 episodes with π_{random}.

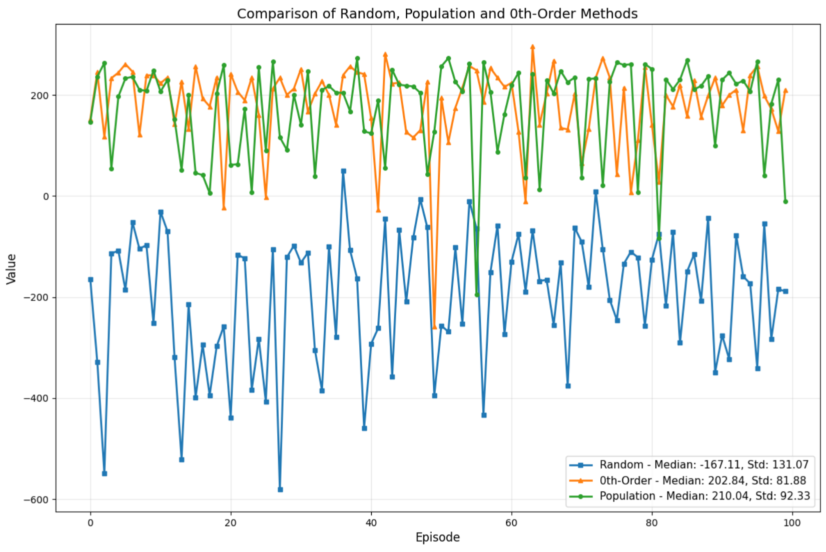

Figure 7: Zeroth-order vs. Population vs. Baseline

We can see from Fig. 7 that both our gradient free methods produce better results than our random baseline, with population methods providing a better median but with higher standard deviation.

4 Conclusion

In this work we have highlighted that gradient-free methods can be suitable to learn simple continuous policies. There are however several limitations. These experiments were not repeated extensively and would benefit from multiple trials to achieve statistically significant results and comparison with state of the art methods such as DDPG is also missing. Additionally, we could improve noise generation by using auto-correlated noise (Wawrzyński 2015) instead of random noise. Finally it would be interesting to compare energy usage from gradient free methods with respect to convential gradient methods.

References

Klimov, O. (2025). *Lunar Lander*. Version v1.2.0. URL: `https://gymnasium.farama.org/environments/box2d/lunar_lander/` (visited on 07/25/2025).

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2019). *Continuous control with deep reinforcement learning.* arXiv: `1509.02971 [cs.LG]`. URL: `https://arxiv.org/abs/1509.02971`.

Russell, S. and P. Norvig (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Section 22.7: Unsupervised Learning and Transfer Learning, Subsection on Autoencoders. Pearson. Chap. 22: Deep Learning. ISBN: 9780134610993. URL: `http://aima.cs.berkeley.edu/`.

Wawrzyński, P. (Apr. 2015). "Control Policy with Autocorrelated Noise in Reinforcement Learning for Robotics". In: *International Journal of Machine Learning and Computing* 5, pp. 91–95. DOI: `10.7763/IJMLC.2015.V5.489`.