Solving the moon landing environment with a stochastically trained neural network

Carlos Lafuente

University of Zaragoza

Darek Petersen

University of Hamburg

Juan Rubio

University of Zaragoza

Maciej Złotorowicz

Akademia Górniczo-Hutnicza

16.10.2024

Abstract

This paper presents an innovative approach to solving the moon landing problem within the OpenAI Gymnasium framework by employing a neural network optimized through Natural Evolution Strategies (NES). Our methodology focuses on training an agent to safely and efficiently land a lunar module by minimizing risks during descent and achieving high precision in landing. The neural network leverages an 8-dimensional observation space simulating real-world conditions, including position, velocity, and contact states.

To enhance the robustness of optimization, we incorporate techniques, such as natural gradients, importance mixing, adaptive randomness injection, and reward shaping, ensuring efficient exploration of the policy space and avoiding local minima. Results demonstrate that our approach is able to find the solution for this environment.

Contents

List of Figures

1 Motivation

The motivation behind this research stems from the increasing interest in autonomous systems capable of performing complex tasks in unpredictable environments, such as space exploration. The moon landing scenario serves as an ideal test bed for developing and refining algorithms that can handle high-stakes decision-making under uncertainty. By employing a neural network trained through Natural Evolution Strategies, we aim to push the boundaries of current methodologies and improve landing safety and efficiency.

2 Theoretical Background

2.1 Introduction

Natural Evolution Strategies (NES) is a novel approach to optimizing unknown fitness functions within a black-box optimization framework. It allows us to search for good or near-optimal solutions to numerous difficult real-world problems. [4]
The key characteristics of NES include:

- It maintains a multinormal distribution and iteratively updates over the candidate solution space.
- Updates are performed using the **Natural Gradient** to improve expected fitness.
- Incorporates innovations such as optimal fitness baselines and **importance mixing** to reduce fitness evaluations.

NES aims to achieve robust performance, minimize costly fitness evaluations, and maintain scalability in high-dimensional problems. By calculating the exact inverse of the Fisher Information Matrix (FIM), NES avoids the approximations typical of previous methods, ensuring stability and convergence. [4]

2.2 Search Gradients

The goal of NES is to maximize an unknown fitness function $f : \mathbb{R}^d \to \mathbb{R}$ while minimizing the number of evaluations. The algorithm proceeds iteratively:

1. A batch of samples $z_1, \ldots, z_n$ is generated from the search distribution $p(z|\theta)$.
2. Fitness evaluations $f(z_1), \ldots, f(z_n)$ are used to adjust the parameters θ.
3. The expected fitness $J(\theta) = \int f(z)p(z|\theta)\, dz$ is maximized by following the estimated gradient direction, obtained using the log-likelihood trick and Monte Carlo estimation.

The core idea of our approach is to find, at each iteration, a small adjustment $\delta\theta$, such that the expected fitness $J(\theta + \delta\theta)$ is increased. The most straightforward approach is to set $\delta\theta \propto \nabla_\theta J(\theta)$, where $\nabla_\theta J(\theta)$ is the gradient on $J(\theta)$. Using the 'log-likelihood trick', the gradient can be written as

$$\nabla_\theta J(\theta) = \nabla_\theta \int f(z)p(z|\theta)\,dz = \int f(z)\nabla_\theta p(z|\theta)\,dz$$
$$= \int f(z)\frac{p(z|\theta)}{p(z|\theta)}\nabla_\theta p(z|\theta)\,dz = \int p(z|\theta)\cdot(f(z)\nabla_\theta \ln p(z|\theta))\;dz.$$

The last term can be approximated using Monte Carlo:

$$\nabla^s_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n} f(z_i)\nabla_\theta \ln p(z_i|\theta),$$

where $\nabla^s_\theta J(\theta)$ denotes the estimated search gradient. [4, p. 2]

In our algorithm, we assume that the search distribution $p(z|\theta)$ is Gaussian, parameterized by $\theta = \langle x, A\rangle$, where x is the mean and A is the Cholesky decomposition of the covariance matrix C, such that $C = A^\top A$ with A being an upper triangular matrix.
We use A instead of C because A explicitly reveals the $d(d+1)/2$ independent parameters that determine the covariance matrix C and as the diagonal elements of A are the square roots of the eigenvalues of C we are ensuring that our matrix is always positive semidefinite, as the variance of any Gaussian distribution is defined as positive.

The parameters of the search distribution θ are represented as a combination of the mean vector x and the Cholesky factors of the covariance matrix A. Specifically, θ is structured as:

$$\theta = \left[(\theta^0)^\top,\quad (\theta^1)^\top,\quad \ldots,\quad (\theta^d)^\top\right]^\top,$$

where $\theta^0 = x$ and $\theta^k = [a_{k,k}, \ldots, a_{k,d}]^\top$ for $1 \le k \le d$.

To calculate the natural gradient, we compute $g(z|\theta) = \nabla_\theta \ln p(z|\theta)$, which involves:

$$g(z|\theta) = \nabla_\theta \left(\frac{d}{2}\ln 2\pi - \frac{1}{2}\ln|A|^2 - \frac{1}{2}(A^{-T}(z-x))^\top(A^{-T}(z-x))\right).$$

The gradient w.r.t. x is:

$$\nabla_x \ln p(z|\theta) = C^{-1}(z-x),$$

and the gradient w.r.t. $a_{i,j}$ $(i \le j)$ is:

$$\frac{\partial}{\partial a_{i,j}} \ln p(z|\theta) = r_{i,j} - \delta(i,j)a^{-1}_{i,i},$$

where $r_{i,j}$ is the (i,j)-th element of $R = A^{-T}(z-x)(z-x)^\top C^{-1}$ and $\delta(i,j)$ is the Kronecker Delta function.
From $g(z|\theta)$, the estimated search gradient $\nabla^s_\theta J(\theta)$ is $\nabla^s_\theta J(\theta) = \frac{1}{n} Gf$,

where $G = [g(z_1|\theta), \ldots, g(z_n|\theta)]$, and $f = [f(z_1), \ldots, f(z_n)]^\top$.
The parameter update is then given by:

$$\delta\theta = \eta \nabla^s_\theta J(\theta),$$

where η is an empirically tuned step size.

2.3 Using the Natural Gradient

Traditional gradient methods often struggle to converge efficiently in fitness landscapes characterized by ridges and plateaus.
An important feature of the natural gradient is its invariance to the parameterization of the search distribution. This makes it particularly effective in handling ill-conditioned fitness landscapes, providing isotropic convergence properties. These properties help prevent premature convergence on plateaus and overly aggressive updates on steep ridges.

In this context, we focus on a specific instance of the natural gradient $\tilde{\nabla}_\theta J$. The natural gradient satisfies the relationship:

$$\mathbf{F}\tilde{\nabla}_\theta J = \nabla_\theta J,$$

where $\mathbf{F}$ is the Fisher Information Matrix. This adjustment ensures convergence properties are robust and independent of the parameter space scaling. [4, p. 2]

The Fisher Information Matrix (FIM) is defined as:

$$\mathbf{F} = \mathbb{E}\Big[\nabla_\theta \ln p(z \mid \theta) \nabla_\theta \ln p(z \mid \theta)^\top\Big].$$

If $\mathbf{F}$ is invertible, the natural gradient can be computed directly as:

$$\delta\theta = \eta \mathbf{F}^{-1} \nabla^s_\theta J,$$

where $\nabla^s_\theta J$ is the estimated search gradient. This adjustment ensures efficient updates that adapt to the geometry of the parameter space.

However, the empirical FIM has several disadvantages:

1. It is not guaranteed to be invertible.

2. Approximations may require computationally expensive batch sizes.
3. Direct inversion of $\mathbf{F}$ is inefficient for large-dimensional problems, with complexity $O(d^3)$.

To address these issues, NES avoids direct computation of the empirical FIM by leveraging an analytical form. For Gaussian search distributions with the typical scalar product $\theta = \langle x, A \rangle$, the exact FIM can be expressed as:

$$\mathbf{F} = \begin{bmatrix} \mathbf{C}^- & 0 \\ 0 & \mathbf{F}_A \end{bmatrix},$$

where $\mathbf{C}^-$ corresponds to the upper-left block and captures positional contributions and $\mathbf{F}_A$ accounts for contributions from the covariance parameter A.

The matrix elements are computed analytically, avoiding empirical estimation. For example, the (m, n)-th element is:

$$(\mathbf{F})_{m,n} = \frac{\partial \mathbf{x}^\top}{\partial \theta_m} \mathbf{C}^- \frac{\partial \mathbf{x}}{\partial \theta_n} + \frac{1}{2} \mathbf{tr}\left(\mathbf{C}^- \frac{\partial \mathbf{C}}{\partial \theta_m} \mathbf{C}^- \frac{\partial \mathbf{C}}{\partial \theta_n} \right)$$

This approach ensures that the FIM is always invertible, computationally efficient, and precise for NES applications. [4, p. 3]

The lower-right submatrix $\mathbf{D}_k$ of $\mathbf{C}^-$ has dimension $d + 1 - k$, with specific cases like:

$$\mathbf{D}_1 = \mathbf{C}^-, \quad \mathbf{D}_d = (\mathbf{C}^-)_{d,d}.$$

We prove that the Fisher Information Matrix (FIM) is invertible, provided that $\mathbf{C}^-$ is invertible. Specifically:

- For $1 \leq k \leq d$, the submatrix $\mathbf{D}_k$ on the main diagonal is positive definite due to $\mathbf{C}^-$'s properties.
- Adding a_k^{-2} to the diagonal of $\mathbf{D}_k$ ensures that eigenvalues remain positive.

Additionally, $\mathbf{F}_0 = \mathbf{C}^-$ is invertible, which implies that the entire FIM, $\mathbf{F}$, is also invertible.

The block-diagonal structure of $\mathbf{F}$ partitions parameters θ into $d + 1$ orthogonal groups $\theta^0, \ldots, \theta^k$. This structure enables independent modification of each parameter group without affecting others.

Computing the inverse of $\mathbf{F}$, which has size $d^2 \times d^2$, directly would involve complexity $O(d^6)$, making it impractical for most applications. Instead:

1. By inverting each block $\mathbf{F}_k$ separately, the complexity is reduced to $O(d^4)$.
2. Using an iterative method, the time complexity is further reduced to $O(d^3)$, with space complexity scaling as $O(d^2)$, which is linear in the number of parameters.

This approach significantly reduces computational and memory demands, enabling practical application of the FIM inversion in NES.

2.4 Optimal Fitness Baselines

The concept of *fitness baselines*, introduced by Wierstra et al. (2008b), serves as an efficient variance reduction method for estimating the update $\delta\theta$. However, earlier baselines, such as those proposed by Peters (2007), are suboptimal with respect to the variance of $\delta\theta$. This suboptimality arises because the associated Fisher Information Matrix (FIM) may not be invertible under those approaches.

To address this, we turn to the exact FIM, which is invertible and can be computed efficiently. Leveraging this property, we derive an optimal baseline to minimize the variance of $\delta\theta$. This variance is given by:

$$\mathrm{Var}(\delta\theta) = \eta^2 \mathbb{E}\Big[\Big(F^{-1}\nabla^s_\theta J - \mathbb{E}[F^{-1}\nabla^s_\theta J]\Big)^\top \Big(F^{-1}\nabla^s_\theta J - \mathbb{E}[F^{-1}\nabla^s_\theta J]\Big)\Big],$$

[4, p. 4] where $\nabla^s_\theta J$ denotes the *estimated search gradient*, expressed as:

$$\nabla^s_\theta J = \frac{1}{n}\sum_{i=1}^{n}\Big[f(z_i) - b\Big]\nabla_\theta \ln p(z_i \mid \theta),$$

[4, p. 4]
Here, b represents the *fitness baseline*, which is a scalar value. Importantly, the addition of b does not influence the expectation of the search gradient, as demonstrated by:

$$\mathbb{E}[\nabla^s_\theta J] = \nabla_\theta \int \Big(f(z) - b\Big)p(z \mid \theta)dz = \nabla_\theta \int f(z)p(z \mid \theta)dz.$$

Thus, while b affects the variance of $\nabla^s_\theta J$, it leaves the expected gradient unchanged. This property is exploited to minimize the variance of $\delta\theta$, leading to more efficient and reliable updates in optimization algorithms.

The variance of $\delta\theta$ depends on the choice of the baseline b. Formally, it is expressed as:

$$\mathrm{Var}(\delta\theta) \propto b^2\mathbb{E}[F^{-1}G]^\top(F^{-1}G) - 2b\mathbb{E}[F^{-1}G]^\top\mathbb{E}[F^{-1}G] + \text{const.}$$

The optimal value of b is derived by minimizing this expression:

$$b^* = \frac{\mathbb{E}[F^{-1}G]^\top\mathbb{E}[F^{-1}G]}{\mathbb{E}[F^{-1}G]^\top(F^{-1}G)}.$$

To approximate b from data, assuming the samples are i.i.d., the following formulation is used:

$$b \approx \frac{\sum_{i=1}^{n} f(z_i)\Big(F^{-1}g(z_i)\Big)^{\top}\Big(F^{-1}g(z_i)\Big)}{\sum_{i=1}^{n}\Big(F^{-1}g(z_i)\Big)^{\top}\Big(F^{-1}g(z_i)\Big)}.$$

[4, p. 4]
The primary disadvantage of a parameter-specific baseline, where b is computed individually for each parameter θ_j, lies in cases of highly correlated parameters. In such situations, the estimates become unreliable due to the variance associated with each specific b_j.

To mitigate this, we propose the concept of the *block fitness baseline*, where a single baseline is used for a group of parameters θ_b. The block formulation is given by:

$$b_b^* = \frac{\mathbb{E}[h_b^{\top} G_b]\mathbb{E}[h_b^{\top} G_b]}{\mathbb{E}[h_b^{\top} G_b G_b^{\top} h_b]},$$

where h_b is the b-th row vector of F^{-1}. This approach leverages the intuition that if the parameters (θ_m, θ_n) are orthogonal and weakly correlated, then a shared baseline can be employed effectively.

2.5 Importance Mixing

In each batch, we evaluate n new samples generated from the search distribution $p(z \mid \theta)$. However, since small updates ensure that the KL divergence between consecutive search distributions is generally small, most new samples fall in regions of high density under the previous distribution $p(z \mid \theta)$. This leads to redundant fitness evaluations in the same areas.

To address this, we introduce a two-step process called *importance mixing*, which aims to reuse fitness evaluations from the previous batch while ensuring that the updated batch conforms to the new search distribution.

2.5.1 Two-Step Process

1. **Rejection Sampling:** Each sample z is accepted with probability:

$$\min\left\{1, \frac{(1-\alpha)p(z \mid \theta)}{p(z \mid \theta')}\right\}.$$

2. **Reverse Rejection Sampling:** New samples from $p(z \mid \theta')$ are accepted with probability:

$$\max\left\{\alpha, 1 - \frac{p(z \mid \theta)}{p(z \mid \theta')}\right\}.$$

2.5.2 Batch Construction

The n_a samples from the previous batch, together with $n - n_a$ newly accepted samples, constitute the new batch. This ensures statistical correctness with:

$$p(z \mid \theta') = (1 - \alpha)\frac{p(z \mid \theta)}{p(z \mid \theta')} + \alpha.$$

2.5.3 Advantages

1. Fewer fitness evaluations are required in each batch, reducing computational costs.
2. More effective reuse of previous evaluations results in more reliable and accurate gradient updates.

2.6 The Algorithm

Integrating all the algorithmic elements introduced above, Natural Evolution Strategies (with block fitness baselines) can be summarized as follows [4, p.5]:

1. Initialize $A \leftarrow I$
2. **Repeat**
   (a) Compute A^-, and $C^- = A^- A^{-\top}$
   (b) Update batch using importance mixing
   (c) Evaluate $f(z_i)$ for new z_i
   (d) Compute the gradient $\mathbf{G}$
   (e) **For** $k = d$ to 0
      i. Compute the exact FIM inverse $\mathbf{F}_k^-$
      ii. Compute the baseline b^k
      iii. $\delta\theta^k \leftarrow \mathbf{F}_k^- \mathbf{G}^k \left(f - b^k\right)$
   (f) **End For**
   (g) $\theta \leftarrow \theta + \eta\delta\theta$
3. **Until** stopping criteria is met

2.7 Main Advantages of NES

Natural Evolution Strategies (NES) constitute a competitive, theoretically well-founded, and relatively simple method for stochastic search. Unlike previous natural gradient meth-

ods, NES *quickly* calculates the inverse of the *exact* Fisher information matrix. This enhances robustness and accuracy, even in high-dimensional search spaces.

3 Formal problem description

The OpenAI Gymnasium offers environments with problems to solve by training neural networks. [1] We choose the lunar lander environment. [2]
All information describing the environment is taken either from the Farama-Foundation website or the Gymnasium GitHub repository [2, 3].

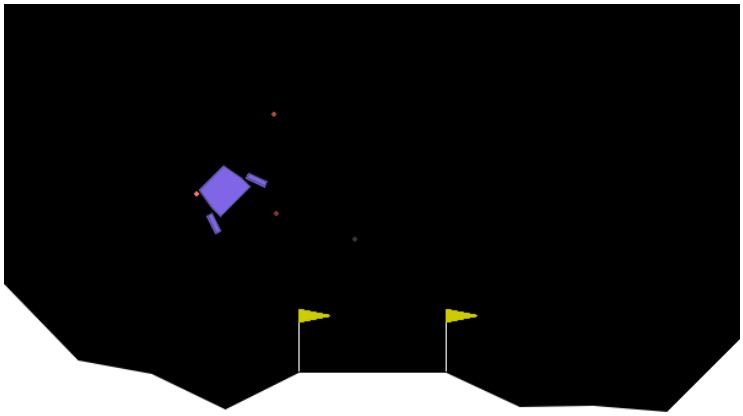


Figure 1: Picture of the lunar lander environment from the OpenAI Gymnasium [2]

This environment consists of a lunar lander which is placed on a constant position $\vec{x}_0 = const$ in the viewport above a landing pad with a constant position located in a random lunar area and is assigned a random initial force $\vec{F}_0$ applied to its center of mass, see equation (1).

$$\vec{F}_0 \in UNIFORM([-1000, +1000] \times [-1000, +1000]) \tag{1}$$

The surface of the moon is also randomly generated. It is created by dividing the surface into chunks. Each chunk is assigned a random value sampled uniformly between 0 and half of the maximum possible elevation $H/2$. [3]

At the center of the environment, a landing pad is created. The segments surrounding the landing pad are manually adjusted to ensure a stable surface. [2]

The moon exerts a gravitational force $g \in [-12, 0]$ on the lander. The standard value set by the simulation environment is $g = -10$. We will be working with this value for g. [2]

3.1 Observation Space

The observation space is an 8-dimensional state vector. The exact domain of each vector element is given by:

$$\begin{aligned} min(\vec{x}) &= (-2.5, -2.5, -10, -10, -6.2831855, -10, 0, 0)^T \\ max(\vec{x}) &= (2.5, 2.5, 10, 10, 6.2831855, 10, 1, 1)^T \end{aligned} \tag{2}$$

This 8-dimensional vector represents the coordinates of the lander in x & y, its linear velocities in x & y, its angle, its angular velocity, and two boolean that represent whether each leg is in contact with the ground. [2]

3.2 Action Space

The simulation has 2 different modes. It can either work with discrete actions or continuous action space. [2]

The discrete action space is a 1-dimensional vector $\vec{o} \in [0, 3] \subset N$. There are four discrete actions available from which the neural network can choose one:

$$\begin{aligned} &0 : do\ nothing \\ &1 : fire\ left\ orientation\ engine \\ &2 : fire\ main\ engine \\ &3 : fire\ right\ orientation \end{aligned} \tag{3}$$

The continuous action space is a 2-dimensional vector $\vec{o}$. The first value of the vector is controlling the main engine, where the value 0 means the main engine is turned off and 1 means the main engine is on full throttle. The second value represents the orientation engines. This value can be negative, as it controls both orientation engines. [2]

For this project, we used the discrete action space.

3.3 Termination Conditions

According to the documentation [2], the simulation stops if one of the following conditions is fulfilled:

1. the lander crashes (the lander body gets in contact with the moon)
2. the lander moves outside the viewport
3. the lander is not awake. (fancy talk for it landed)

3.3.1 Reward Function

To explain the reward function, it is easiest to quote to documentation [2]:

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 is ...

- increased/decreased the closer/further the lander is to the landing pad.
- increased/decreased the closer/further the lander is to the landing pad.
- increased/decreased the slower/faster the lander is moving.
- decreased the more the lander is tilted (angle not horizontal).
- increased by 10 points for each leg that is in contact with the ground.
- decreased by 0.03 points each frame a side engine is firing.
- decreased by 0.3 points each frame the main engine is firing.

The episode receives an additional reward of -100 or +100 points for crashing or landing safely, respectively.
An episode is considered a solution if it scores at least 200 points.

3.4 Environment Arguments

There are several arguments to pay attention to. Some of the ones that could be important are the gravity, the turbulence, the wind, or the wind power.

All of them can be implemented in the project in different ways, for example, the formula below could create the wind which will affect the trajectory. [2]

$$\tanh\left(\sin\left(2k\left(x+C\right)\right)\ +\ \sin\left(\pi\cdot k\left(x+C\right)\right)\right)$$

4 Software Agent Architecture

In each episode of the training process, the agents' average fitness in the problem environment is estimated, and some performance indicators are logged for visualization of performance development. This simulation runner itself is a paralleled loop running all the agents in batches according to the hyperparameters in section 5.3. In every step of each of these simulations, the environment provides an observation vector. According to his vector, our randomized neural networks create an action vector. The action vector in return is passed to the environment, return the next state vector and a reward for this time step (see subsection 3.3.1). This loop continues until the simulation ended, resulting in a cumulative total reward for that run of the simulation. For each randomized network, this is done several times to calculate the average fitness.

The agents and their averaged total fitnesses are passed to the stochastic optimizer, where the next reference model is generated through NES 2. This process repeats itself over and over again. For a visualization of this process, see figure 2.

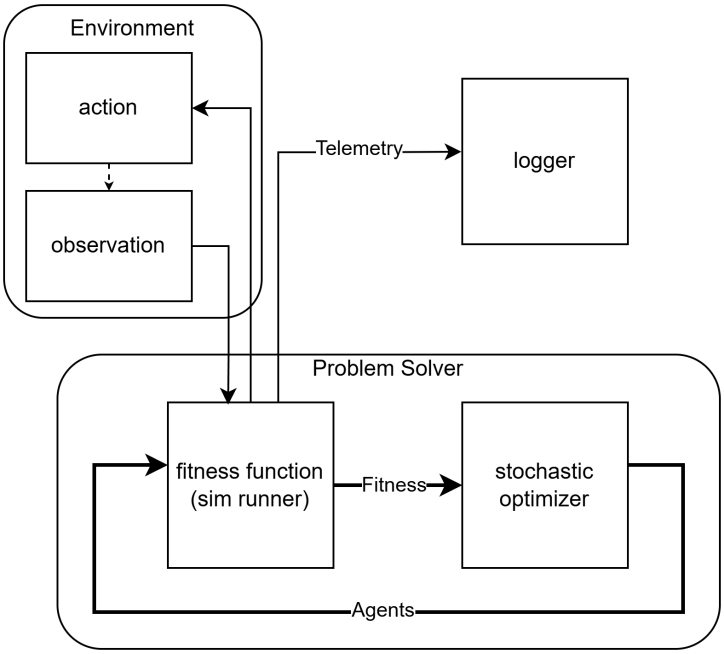


Figure 2: Visualization of program architecture.

5 Implementation and Training

NES algorithm tends to converge to a local minimum [4], which is why we have implemented a set of tools to enhance the exploration of the policy space. Including:

- Randomness injections to the policy
- Variable hyperparameters
- Custom reward shaping to guide the agent to the desired behavior
- l2 regularization

5.1 Exploration vs. Exploitation

We tried to mitigate the typical exploration versus exploitation problem by implementing randomness injections into the policies. For a description of the hyperparameters, view table 5.3.

sigma jumping After each generation, *params.sigma* was multiplied by a factor to decrease it. After a set number of generations, *params.sigma* was reset to its original value.

step randomness To introduce further randomness, we introduced code that after *params.set_randomness_...* added some Gaussian noise with a sigma of *params.sigma_randomness_...* to the reference model.

5.2 Natural Gradient

Natural Evolution Strategies (NES), as already described in section 2, is, in a nutshell, a novel approach to optimizing unknown fitness functions within a black-box optimization framework. It allows us to search for good or near-optimal solutions to numerous difficult real-world problems.

However, as you saw we changed our original search distribution from a Gaussian distribution to a Uniform one. One of the main reasons and the main advantage of this change is the decentralization around the mean.
That feature of the Uniform distribution allows us to explore a broader solution space and avoid biases toward the center. Which proved to be advantageous, allowing the algorithm to escape local minima and better handle the stochastic nature of the environment.

By facilitating a more extensive search, the uniform distribution aligns more effectively with the objectives of robust optimization in high-dimensional spaces.

5.3 Hyperparameters

To allow adjustment of training parameters, the following hyperparameters were introduced:

params.hidden_layers Specifies the architecture of the policy network. NES algorithm can handle various network architectures, but we recommend simple architectures with few neurons in each layer (for example, `[12, 4]` or `[4]`).

params.model_penalty L2 regularization penalty for the policy network.

params.batch_size Defines the number of episodes in each batch. Reduces the amount of instances of the environment. Recommended values are `2-10`.

params.repetitions Number of fitness evaluations per episode, providing robustness in estimating the policy's quality. Higher values improve the quality of the policy but increase computational cost. Recommended values are `10-200`.

params.max_steps Max number of steps in each episode.

params.episodes Stopping criterion for the training process.

params.step_randomness_to_w_small and params.step_randomness_to_w_big Frequency of introducing small and large random perturbations to the policy.

params.sigma_random_small and params.sigma_random_big The magnitude of small and large random adjustments, respectively. These values control how much randomness is injected into the policy at each step. Recommended values are `0.001-0.01` for small and `0.05-0.15` for large perturbations. Magnitude is dependent on frequency.

params.learning_rate Step size used to update the distribution parameters in NES. Higher values speed up training but risk overshooting optimal solutions. Recommended values are `0.01-0.5`.

params.sigma The standard deviation of the noise distribution used for sampling perturbations. Higher values increase exploration but may slow down convergence. Recommended values are `0.5-5.0`.

params.npop The population size, representing the number of candidate policies evaluated per generation. Higher values improve robustness but require more computational resources. Recommended values are `10-100`.

Some parameters (e.g., params.hidden_layers) are set for a specific neural network and can not be changed during training.

Mean fitness refers to the average fitness of a sampled population during training, incorporating regularization. Typically, mean fitness values are lower due to the randomness in the model's parameters. In contrast, reference fitness represents the average fitness of the original model, excluding regularization.

5.4 Custom Reward Shaping

Even with the randomness injections and when adjusting the hyperparameters during training, the model still was not performing optimally. Therefore, we added custom reward shaping. This allows us to steer the model toward the desired behavior. As the model approached the optimal target solution, we decreased the amount of reward shaping until it no longer had any influence on the fitness function.

An example of this was the model's inability to perform a proper landing, even though it managed to ascend with ease and was even able to counteract strong random initial forces and wind. In the end, it would always get stuck in the very end as it was not turning off the engine to stay still but rather continued to slightly turn on the thrusters over and over. We assume that this happened because it was triggering some reward bonus.

In order to counteract this behavior, we introduced custom reward shaping and gave rewards that encouraged standing still and close to the floor (figure 1. At episode, 20000 – visible in the very right of figure 3 – the mean fitness of the model finally rapidly increased. We assume that this is because the model was eventually able to perform a proper landing and therefore received the landing reward for the first time.

On Code snippet 1 shows our approach to guide the agent into behavior that allows him to land.

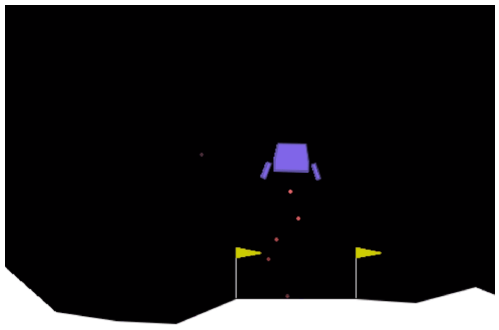
We rewarded the agent for ...

- Stabilizing rotations (keeping it close to 0)
- reducing x velocity and keeping y velocity slightly negative
- penalty for hovering (getting stuck just above the ground)
- maneuvers that will stabilize the angle and position
- penalty for going too far off-center
- huge reward for touching the ground

We gradually decreased the weights of these rewards as training progressed, starting with values close to 1 and reducing them to almost 0.

Additionally, during training, we began with reduced gravity (-5) and no wind (0), gradually increasing these values as the training advanced to the nominal values (-10 and 16).

By playing with the hyperparameters and custom reward shaping, we tried to find the best possible set of neural network parameters.
As the model repeatedly got stuck in local minima, we also adjusted the parameters during the training process. In figure 3 the different colors each represent a training session with different hyperparameters and sometimes reward shaping.



Animation 1: NES stuck in local minimum, link: github/Animation-Stuck.gif

5.5 Training Process

The training process was rough, with the model frequently getting stuck in local solutions and occasionally escaping them randomly. A key factor in training the model was the use of custom rewards mentioned earlier. Although these rewards were removed later, their side effect was that the model struggled to fulfill the end condition (turning the engines

off for a few frames after landing). However, it eventually managed to learn this behavior in the later stages of training, after reward shaping was removed. The average fitness is spiking during the run due to the changing values of hyperparameters (mainly decreasing and increasing sigma). Higher values of sigma introduce more randomness into the runs and decrease the average performance of the model.

Training was divided into 4 parts

1. Section

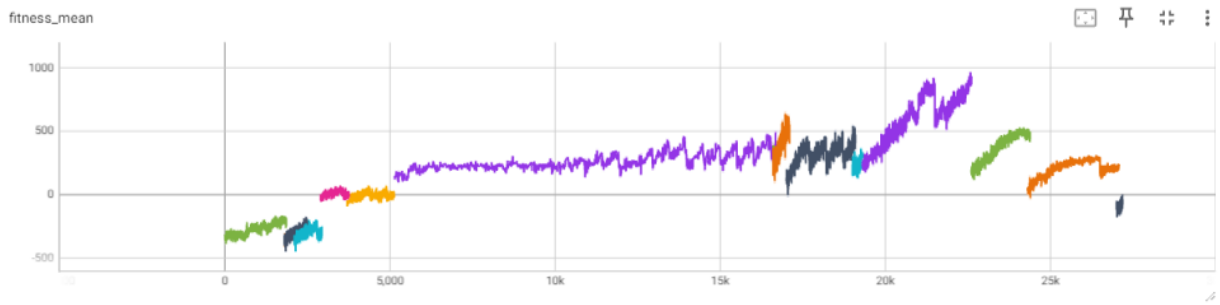


Figure 3: Average fitness over the whole training process of the trained neural network model.

Because the reward function was being modified, Figure 3 and 4 do not accurately represent the model's performance, particularly during sudden jumps. The most illustrative example of the fitness function is Figure 5, which is recalculated using the original reward function for each saved checkpoint.

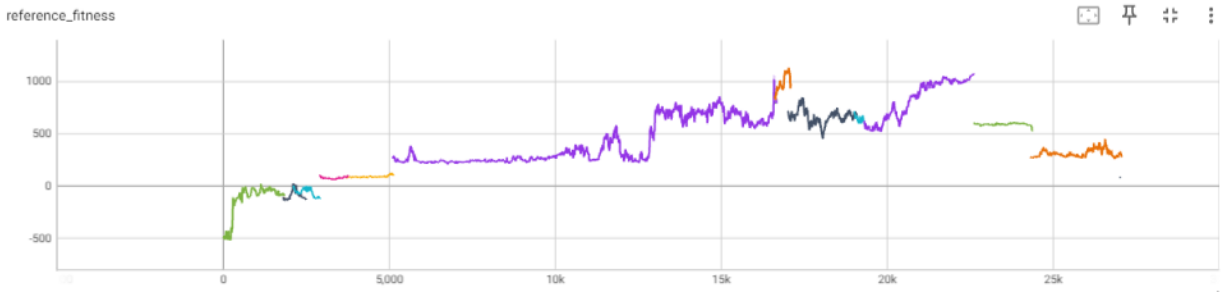


Figure 4: Reference fitness over the whole training process of the trained neural network model.

Final fitness shows that the performance of the model almost reaches the target score for the solution (200). The x-axis represents the model checkpoint index.

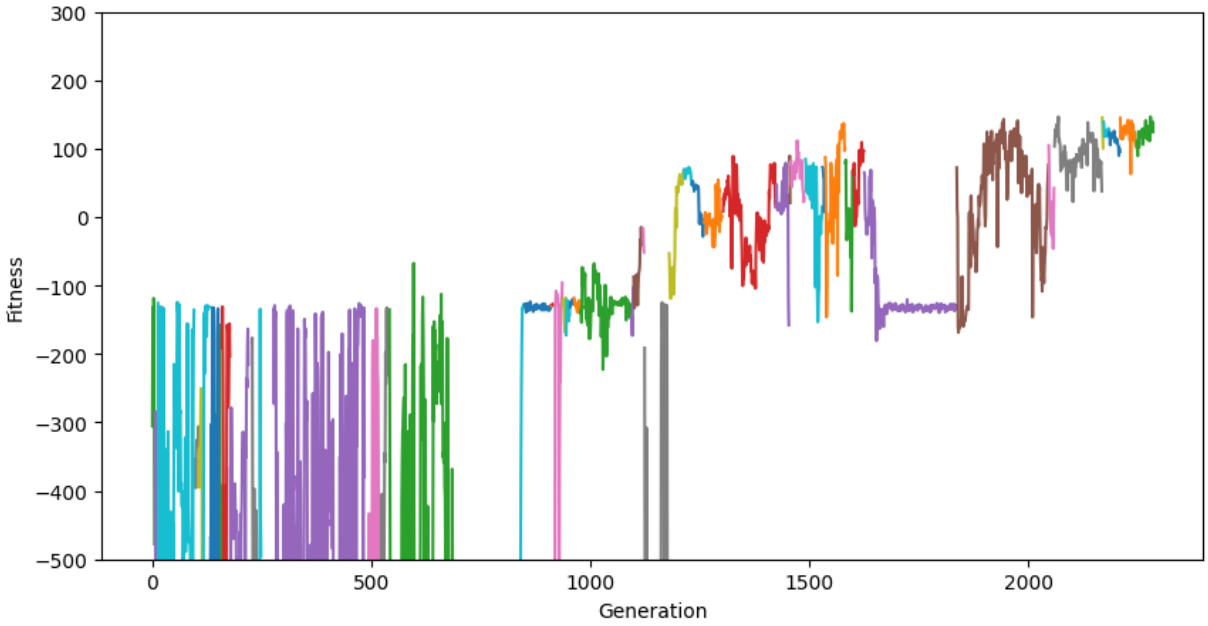


Figure 5: Values of original fitness function

We also kept other telemetry during training, including all values of the fitness function across every attempt of the agent. With this, we can visualize this as a histogram (Figure 6).

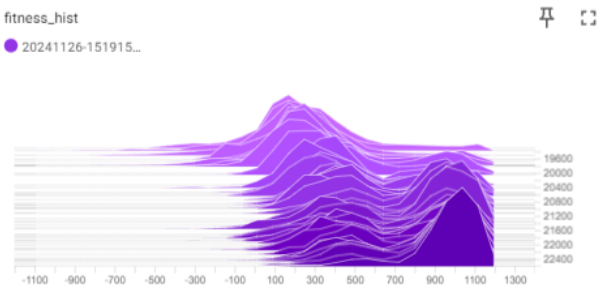


Figure 6: Histogram of fitness values for each agent in episode

We also tracked the values of the L2 penalty applied to the model. Initially, the algorithm quickly reduced the magnitude of the parameters to minimize the L2 penalty. However, later in training, as the model learned how to solve the problem, it accepted slightly higher L2 penalties in exchange for achieving greater rewards (Figure 3)

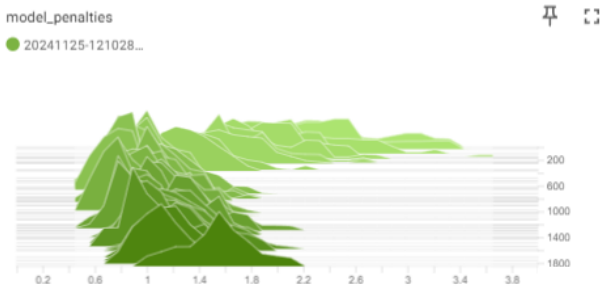


Figure 7: Values of l2 regularization applied in training process

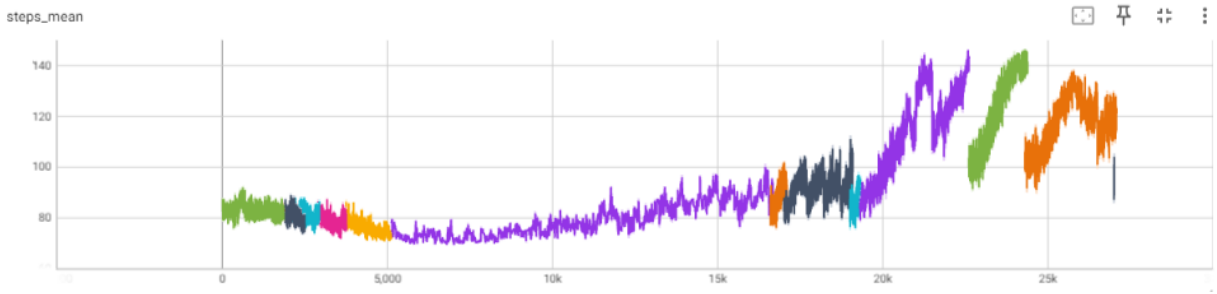


Figure 8: Mean amount of steps per run

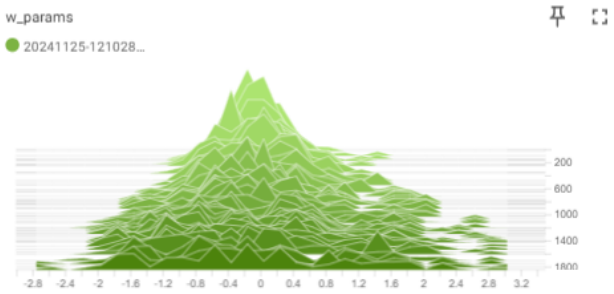


Figure 9: Histogram of parameters of the model across episodes

6 Results

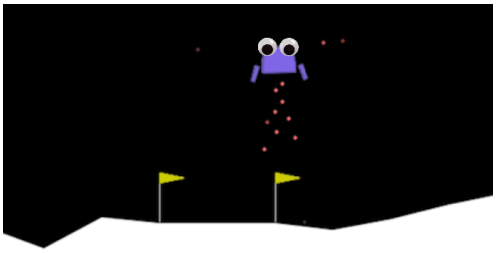
Our final model A [1] can perform a lunar landing easily. It can handle random initial forces of up to 1.5 times the normal environment value and has no problem dealing with the simulated wind, as shown in figure 3 and 4. With more than 60000 *eps* the model's training took quite long, requiring an enormous amount of fitness function computations.

[1] available in the GitHub repository under `https://github.com/Lord225/stochastic-gym-solver`

The final model is the result of 3.5 days (1344 CPU hours) of training on a 16-core machine and the code was well optimized for parallel computations, though, as the NES algorithm does not seem to be behaving as well as we were hoping for (fig. 3). Additionally, we had to change hyperparameters and add reward shaping during training, as the NES algorithm kept getting stuck in local minima. A decision tree of the final model behavior shows in which situation the environment chose which actions (see figure 10).

6.1 Model Behavior

You can see the model performance in the video linked under animation 2.



Animation 2: Summary of training process, link:
https://www.youtube.com/watch?v=NRX6X9aTKds

7 General Conclusions

In this study, we successfully applied Natural Evolution Strategies (NES) to optimize a neural network for the moon landing environment in OpenAI Gymnasium. By leveraging a stochastic optimization approach, we achieved a robust solution that allows for smooth and safe landings of the virtual moon lander. The integration of advanced techniques such as the natural gradient and optimal fitness baselines significantly enhanced the performance and efficiency of our algorithm. Our findings demonstrate that NES not only reduces the number of fitness evaluations needed but also maintains stability in high-dimensional parameter spaces. Future work can explore extending this methodology to other complex environments, further validating the efficacy of NES in various applications. We performed the attempt to explain the behavior of the agent using decision tree, it was drawn on figure 10.

8 Possible farther research

One aspect we did not consider is the influence of reward function as a stochastic problem itself. As discussed in 3, the Gymnasium environment contains some randomness. To get a mean reward function, that properly describes a model's behavior on average, we ran every

model *params.repetitions* times and average the rewards received (see subsection 5.3). It would be interesting to compare different ways of computing the reward function.
Some ideas are ...

- Generalize to other environments - This project can be easily applied to solve other environments found in gym.
- Find a better solution to the problem.

Appendices

A Final Model Parameters

The final model comprises a small network with 2 hidden layers: 16 and 4 neurons. It was trained for approximately 35k episodes that consists of 1000-2000 environment runs. Around 35 millions fitness function evaluations. Training was incremental, and we were changing hiperparmeters and optimization goal to speed up the process. The trained model, all code, and additional content can be found in the GitHub-Repository.

B Models' Decision Tree

Figure 10: Decision tree that reproduces 70% of agent behavior

```python
# Penalize inaction unless vertical velocity is negligible
if decision == 0:
    if abs(v_y) > 0.20:
        reward -= 1
    else:
        reward += 0.5

# Reward stabilizing rotation towards zero angle
reward += (1.0 - abs(angle))*0.1

# Reward reducing velocities (horizontal and vertical)
reward += max(0, 1.0 - abs(v_x)) * 0.1
reward += max(0, 1.0 - abs(v_y+0.5)) * 0.8

# penaly for hovering
if abs(v_y) < 0.1:
    reward -= 0.5

# Reward main engine steering for stabilizing both angle and position
if (angle > 0 and x > 0 and decision == 2) or \
(angle < 0 and x < 0 and decision == 2):
    reward += 0.1

# Heavlily penalize going out of 2.0 rage off the center
if abs(x) > 1.25:
    reward -= 3

if abs(x) > 2.0:
    reward -= 3

# Heavily reward achieving stability across all key metrics
reward += max(0, 1.0 - abs(x)) * 3
reward += max(0, 1.0 - abs(v_y))*0.05
reward += max(0, 1.0 - abs(v_x))*0.05
reward += max(0, 1.0 - abs(ang_vel))*0.1

if leg1 == 1 or leg2 == 1:
    reward += 25
```

Code Snippet 1: Custom Reward shaping

References

[1] OpenAI. *OpenAI Gymnasium*. URL: https://gymnasium.farama.org/ (visited on 10/26/2024).

[2] OpenAI. *OpenAI Gymnasium Lunar Lander Docs*. URL: https://gymnasium.farama.org/environments/box2d/lunar_lander/ (visited on 10/16/2024).

[3] Mark Towers et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. URL: https://github.com/Farama-Foundation/Gymnasium (visited on 01/09/2025).

[4] Sun Yi et al. "Stochastic search using the natural gradient". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 1161–1168. ISBN: 9781605585161. DOI: 10.1145/1553374.1553522. URL: https://doi.org/10.1145/1553374.1553522.