DSA4212:
Natural Evolution Strategies

Group Composition:

1. Yap Vit Chun, A0225837X, e0596609@u.nus.edu

November 8, 2024

1 Introduction

In DSA4212, we have focused on optimization problems involving smooth and differentiable functions. While some of these problems, like the Rosenbrock function, are ill-conditioned with narrow valleys, we have learned techniques like the momentum method and second-order methods to handle such challenges. These methods fundamentally rely on computing the gradients of the objective functions, such as the Gradient Descent algorithm.

However, some optimization problems involve objective functions that are difficult to define, let alone the derivatives. These functions may be non-smooth or non-differentiable, making gradient computation challenging or impossible. Such challenges motivate the field of black-box optimization, which requires only evaluations of the objective function without additional information, like gradients. Evolutionary strategies (ES) are a family of algorithms well-suited for black-box optimization, offering a robust approach to tackle complex, non-differentiable problems.

2 Evolution Strategies (ES)

Evolution Strategies, as the name implies, aim to ”evolve” solutions over multiple iterations through the following steps:

1. Generate some candidate solutions and evaluate their performance or fitness.
2. Retain the best-performing candidate solutions while discarding the others.
3. Apply slight modifications (mutations) to the surviving solutions to generate new candidates for the next generation.

This simple yet effective iterative process has been shown to yield good to excellent results for many complex optimization problems.

A straight-forward example of this evolutionary process is using a Gaussian distribution. Initially, solutions are generated in the parameter space based on a Gaussian distribution with specific initialized parameters (μ and Σ). Each solution is then evaluated for fitness, and the distribution parameters (μ and Σ) are updated based on the best-performing solutions. This process is

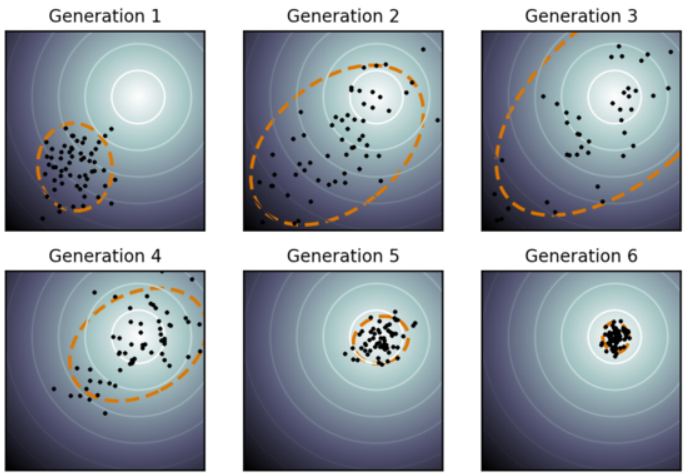


Figure 1: An illustration of how evolution strategies (specifically a variant of ES called CMA-ES) work. Towards the end, the samples are concentrated at the global minimum. (Image source: Wikipedia contributors, 2024

repeated until satisfactory solutions are found. Figure 1 illustrates this process.

While the basic framework of ES has proven effective in optimizing complex problems by evolving candidate solutions through mutation and selection, there are limitations that arise when tackling high-dimensional problems or when the solution space has intricate dependencies between variables. In such cases, traditional ES methods may struggle to efficiently explore the solution space, giving rise to the Natural Evolution Strategies (NES).

3 Natural Evolution Strategies (NES)

The Natural Evolution Strategies (NES) is an extension of the standard ES framework that improves upon it by adapting the search process based on the distribution of solutions rather than just modifying individual candidates (Wierstra et al., 2014). NES uses a probabilistic model to represent the entire population of solutions, which is iteratively adjusted based on the observed fitness values. Mathematically, if we let θ represent the parameter of density $\pi(\mathbf{z}|\theta)$ and $f(\mathbf{z})$ represent the fitness function, the expected fitness $J(\theta)$ under the search distribution is

$$J(\theta) = \mathbb{E}_\theta[f(\mathbf{z})] = \int f(\mathbf{z})\,\pi(\mathbf{z}|\theta)\,d\mathbf{z}.$$

Using the log-likelihood trick presented in the paper, the derivative of $J(\theta)$ can be rewritten as

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\theta\left[f(\mathbf{z})\nabla_\theta \log \pi(\mathbf{z}|\theta)\right] \\ &\approx \frac{1}{\lambda}\sum_{k=1}^{\lambda} f(\mathbf{z}_k)\nabla_\theta \log \pi(\mathbf{z}_k|\theta)\end{aligned}$$

where λ represents the population size. Then, the update of the parameter is the standard gradient ascent:

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$$

However, the approach described above (referred to by Wierstra et al. (2014) as plain search gradient updates) has some limitations. Mainly, as the update progress, they can become increasingly unstable since the "gradient controls both position and variance of a distribution over the same search space dimension".

The updates are then improved by using the natural gradient, which addresses some of the instability issues mentioned earlier. Instead of relying on the plain gradient, the natural gradient takes into account the geometry of the problem by using the Kullback-Leibler (KL) divergence as a measure of the difference between two probability distributions. This allows the optimization process to be more stable and efficient by removing the dependence on the specific parameterization of the distribution.

Based on previous work cited in Wierstra et al. (2014), the natural gradient turns out to be

$$\nabla_\theta J(\theta) = \mathbf{F}^{-1} \nabla_\theta J(\theta)$$

and thus the update rule becomes

$$\theta \leftarrow \theta + \eta \mathbf{F}^{-1} \nabla_\theta J(\theta)$$

where $\mathbf{F}$ is the Fisher information matrix given by

$$\mathbf{F} = \mathbb{E}\left[\nabla_\theta \log \pi(\mathbf{z}|\theta) \nabla_\theta \log \pi(\mathbf{z}|\theta)^T\right]$$

As such, we have the pseudocode of the natural evolution strategies as described in Algorithm 3 (Wierstra et al., 2014)

Algorithm 3: Canonical Natural Evolution Strategies

input: f, θ_{init}
repeat
  for $k = 1 \dots \lambda$ **do**
    draw sample $\mathbf{z}_k \sim \pi(\cdot|\theta)$
    evaluate the fitness $f(\mathbf{z}_k)$
    calculate log-derivatives $\nabla_\theta \log \pi(\mathbf{z}_k|\theta)$
  end
  $\nabla_\theta J \leftarrow \frac{1}{\lambda} \sum_{k=1}^{\lambda} \nabla_\theta \log \pi(\mathbf{z}_k|\theta) \cdot f(\mathbf{z}_k)$
  $\mathbf{F} \leftarrow \frac{1}{\lambda} \sum_{k=1}^{\lambda} \nabla_\theta \log \pi\left(\mathbf{z}_k|\theta\right) \nabla_\theta \log \pi\left(\mathbf{z}_k|\theta\right)^\top$
  $\theta \leftarrow \theta + \eta \cdot \mathbf{F}^{-1} \nabla_\theta J$
until *stopping criterion is met*

With the algorithm provided by the authors (Wierstra et al., 2014), we will now try to implement the natural evolution strategies to solve two optimization problems. We did not solve the third problem as required in this assignment as we realized we did not have enough space for the report. There was also a time constraint to finish this assignment.

4 Implementation of NES

4.1 Problem 1: Rosenbrock function

As encountered many times in this course, the Rosenbrock function is given by (setting $a = 1$ and $b = 100$)

$$R(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

This function has long, elongated and narrow valleys around the global minimum, resulting in zig-zag or unstable updates if we use Gradient Descent. We have chosen this simple and classic function to optimize first to test our understanding of the algorithm. For the NES, we have used the Gaussian distribution to generate our populations. For the fitness function, we use a rank-based utility function (smaller rosenbrock value ranks higher) as presented in Wierstra et al. (2014)

However, implementing NES turned out to be more challenging than anticipated, despite the seemingly straightforward nature of the algorithm. Namely, Algorithm 3 outlined above requires the computation of the Fisher Information Matrix and its inversion, which became a significant bottleneck during our implementation. Not only is the inversion of the matrix computationally expensive, it is also very unstable.

For instance, we encountered issues related to the stability of the covariance matrix. Since the covariance update in NES is based on the empirical gradient (i.e., using the observed gradient estimate), this leads to instability in the covariance matrix update process. Over time, this instability can cause the covariance matrix to lose its positive-semidefiniteness, which is crucial for ensuring that the algorithm maintains a valid probability distribution. Without positive-semidefiniteness, the covariance matrix cannot represent a valid Gaussian distribution, leading to errors and unreliable updates. Despite trying to add in some small epsilon to the matrix during the update, the problem seemed to persist.

Furthermore, we also observed that after several iterations, both the entries of the mean vector and the covariance matrix began to explode. We suspect that this phenomenon was primarily due to the near-singularity nature of the matrix.

Upon reading more literature (Akimoto et al., 2010; Ye & Zhang, 2023, Sun et al., 2012), it turns out that the exact Fisher information matrix of a multinormal distribution can be obtained

theoretically. Skipping the mathematical details, we have

$$\mathbf{F} = \begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & \frac{1}{2}\Sigma^{-1} \otimes \Sigma^{-1} \end{bmatrix}$$

where $\otimes$ denotes the Kronecker product. Therefore, the inverse of the Fisher information matrix is just

$$\mathbf{F}^{-1} = \begin{bmatrix} \Sigma & 0 \\ 0 & 2\Sigma \otimes \Sigma \end{bmatrix}$$

Given the easily derived $\nabla_\theta \log \pi(\mathbf{z}|\theta)$ with respect to both μ and Σ, multiplying the exact inverse Fisher information matrix with the derivatives give us

$$\mathbf{F}^{-1} \nabla_\theta \log \pi(\mathbf{z}|\theta) = \begin{bmatrix} \mathbf{z} - \mu \\ (\mathbf{z} - \mu)(\mathbf{z} - \mu)^T - \Sigma \end{bmatrix}$$

With the above result, the update of μ and Σ now becomes trivial

$$\mu \leftarrow \mu + \eta \frac{1}{\lambda} \sum_{k=1}^{\lambda} f(\mathbf{z_k})(\mathbf{z_k} - \mu)$$

$$\Sigma \leftarrow \Sigma + \eta \frac{1}{\lambda} \sum_{k=1}^{\lambda} f(\mathbf{z_k}) \left((\mathbf{z_k} - \mu)(\mathbf{z_k} - \mu)^T - \Sigma \right)$$

The above update rule will be used throughout our implementation of NES. In fact, the above update rule yields us stable updates that leads to convergence eventually. Figure 2 illustrates the convergence of the algorithm with different initialized parameters.

To compare the NES with the basic framework of ES, we have also implemented a simple (100 + 1) ES algorithm to optimize the Rosenbrock function. A (100 + 1) ES algorithm is one where during each iteration, 100 candidate solutions are generated and the best solution is selected to generate subsequent candidates. For simplicity, we do not implement the update of covariance but just the mean. Figure 3 shows the trajectory of the mean parameter, which illustrates unstable updates near the global minima (back and forth updates).

Lastly, from this simple example, we can also explore how the learning rates and population size for each iteration affect the convergence. Figure 4(a) shows the convergence for different learning rates and Figure 4(b) shows the convergence for different population sizes.

When the learning rate increases, the algorithm converges to the global minimum more quickly but at the cost of increased instability. If the learning rate is set too high (e.g., LR = 3), the algorithm may diverge entirely. In contrast, the choice of population size appears to have a less significant impact on convergence. As shown in Figure 4(b), once the population size is increased to 50, the differences in convergence speed become minimal. However, larger population sizes require more computational resources, which can slow down convergence. Therefore, it is generally better to choose a population size that strikes a balance—not too small to hinder progress, but not too large to cause excessive computational delays.

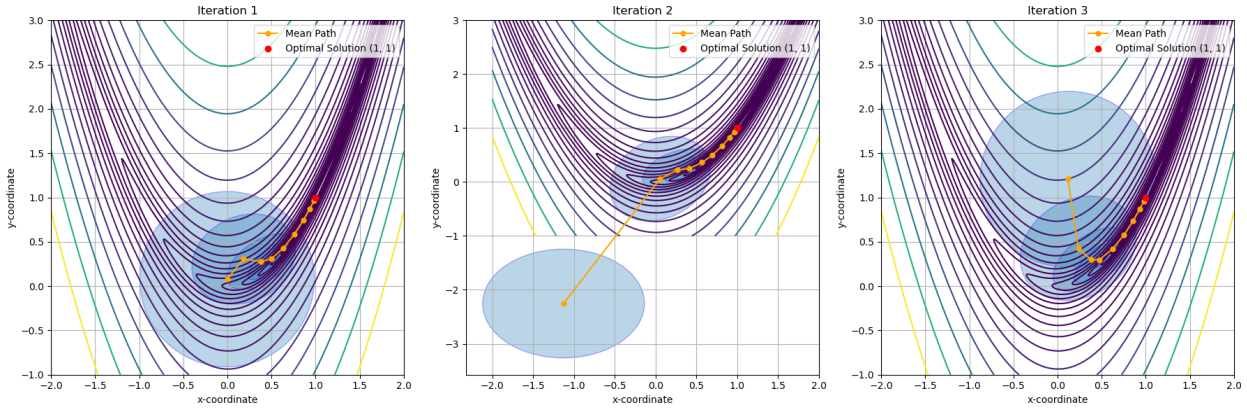


Figure 2: An illustration of the convergence of NES algorithm even with three different initial parameters (μ). We can see that as the solution navigates through the valley, the covariance also decreases and the samples are concentrated at the global minima eventually

4.2 Problem 2: Alternative to reinforcement learning

This optimization problem is motivated from the paper published by OpenAI in 2017, in which the researchers demonstrate that a variant of NES performs exceptionally well as an alternative to traditional Reinforcement Learning (RL) techniques, such as Q-learning and Policy Gradients (Salimans et al., 2017). Notably, the proposed algorithm scales efficiently with the increasing number of available CPUs through parallelization, significantly accelerating the training process.

Briefly speaking, RL trains an agent by defining a policy function, typically a deep neural network, that determines how the agent should act based on the current state of the environment (e.g., a game). According to OpenAI (2017) these policy functions have about 1 million parameters, so the task of RL is to find the optimized parameters such that the policy plays well.

In RL, the agent interacts with the environment through exploration and updates the policy using backpropagation. However, a key challenge in RL is that, unlike supervised learning, the exact gradient of the loss function is not directly available. Instead, the gradient of the expected reward must be estimated through sampling, which can lead to issues such as exploding gradients. Additionally, the need to backpropagate gradients and synchronize large models across multiple workers or GPUs can create significant bottlenecks.

In contrast, evolution strategies simplify this process by eliminating the need for backpropagation. With ES, only a feedforward pass is required to evaluate rewards, making it much easier to parallelize. This allows multiple workers to operate simultaneously, with minimal computational overhead, as long as there is access to CPU resources.

In this assignment, we have tried to implement NES to learn the InvertedPendulum problem in MuJoCo environment. The algorithm outlined by the paper neglects the update of the covariance

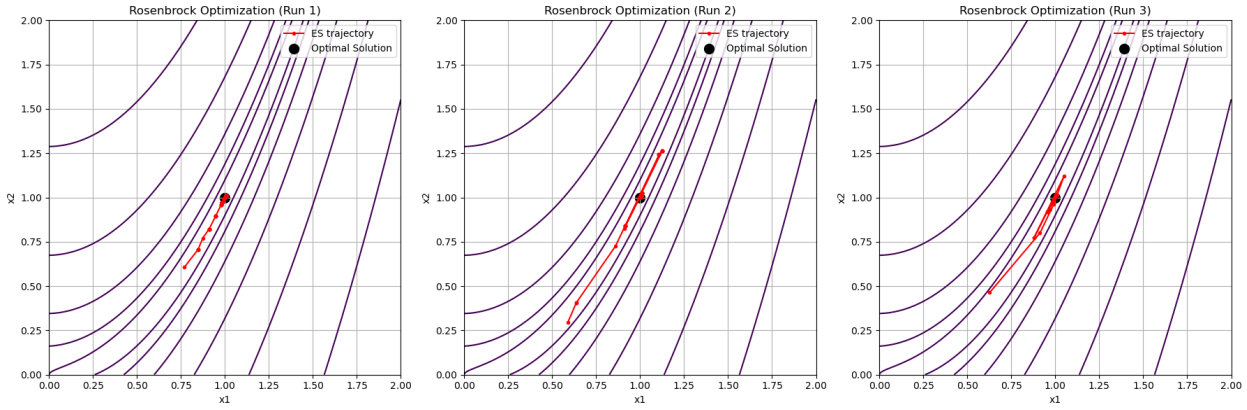
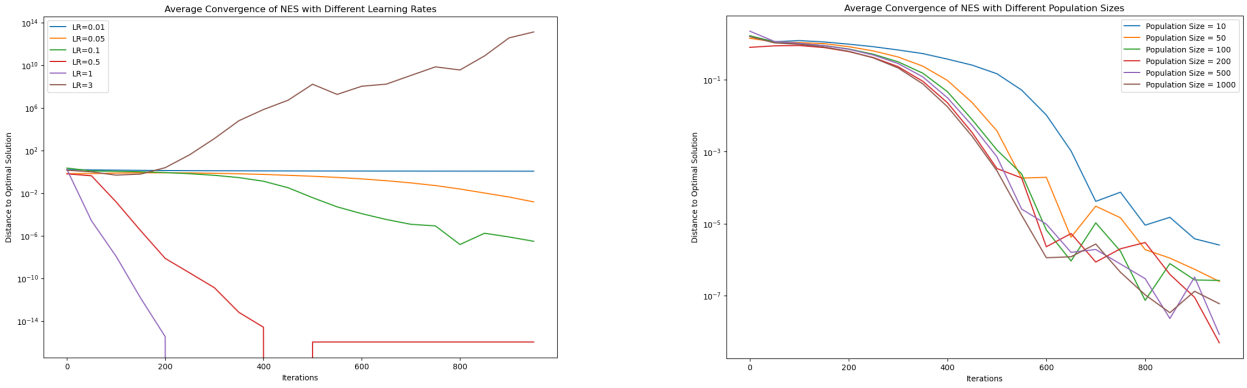


Figure 3: An illustration of the convergence of (100 + 1) ES algorithm. Despite eventual convergence, we can see that the updates are less stable as compared to NES, with some zig-zag movements near the global minima solution



(a) Convergence of NES with different learning rates

(b) Convergence of NES with different population size

Figure 4: Hyperparameter tuning for NES

due to the large number of parameters, focusing only on the updates of mean for each parameter in the neural network. Therefore, the update is simply

$$\mu \leftarrow \mu + \eta \frac{1}{\lambda} \sum_{k=1}^{\lambda} f(\mu + \sigma \epsilon_k) \epsilon_k$$

where $\epsilon \sim \mathbb{N}(0, I)$. Note that the above update rule is the same as what we have above upon some rearrangement. Essentially, the algorithm injects noise into the policy parameters and evaluates the outcome by running an episode. In contrast, RL introduces noise into the action space and employs backpropagation to adjust the parameters. In simple terms, ES performs a "guess and check" on the parameters, while RL does so on the actions. (OpenAI. 2017)

Learning Rate (LR)	Population Size	Time to 1000 (s)	Best Reward
0.01	10	110.10	669
0.01	50	86.27	1000
0.01	100	125.42	1000
0.01	200	118.01	1000
0.05	10	30.73	1000
0.05	50	20.40	1000
0.05	100	23.89	1000
0.05	200	12.55	1000
0.1	10	5.14	1000
0.1	50	7.99	1000
0.1	100	45.65	1000
0.1	200	530.52	102
0.2	10	10.59	1000
0.2	50	95.97	1000
0.2	100	236.06	63
0.2	200	474.04	56

Table 1: Summary of Hyperparameter Optimization Results

With the above update rule, it's clear how easily the training process can be parallelized, although we did not implement this in this assignment. Note that a significant portion of the computation comes from the term $\sum_{k=1}^{\lambda} f(\mu + \sigma\epsilon_k)\epsilon_k$, which can be computationally expensive, especially with deep neural networks. However, each $f(\mu + \sigma\epsilon_k)\epsilon_k$ can be computed independently! This allows for generating λ child parameters, evaluating them in parallel, collecting the results, and then performing the update.

In our implementation, for simplicity, we used a neural network with two hidden layers, each containing 64 nodes, built with PyTorch. The main challenge in this implementation was ensuring the correct dimension matching, which required some time to debug. Despite this, the algorithm was able to find a policy that maximized the reward (reaching 1000 in the InvertedPendulum environment) within just 10-20 seconds, demonstrating impressive speed. Additionally, we fine-tuned the learning rate and population size to optimize the convergence rate, further enhancing the algorithm's performance. Table 1 summarizes the result. As observed, with learning rate set to 0.1 and population size set to 10, we managed to finish training the agent in just 5.13 seconds. This result thus shows the importance of hyperparameter tuning.

Ideally, we would also like to compare the performance with RL technique such as Q-learning, however due to time constraint, we did not have time to learn the technique and implement it to compare.

References

[ANOK10] Youhei Akimoto, Yuichi Nagata, Isao Ono, and Shigenobu Kobayashi. *Bidirectional Relation between CMA Evolution Strategies and Natural Evolution Strategies*. 1 2010.

[con24] Wikipedia contributors. CMA-ES, 9 2024.

[Ope17] OpenAI. Evolution strategies as a scalable alternative to reinforcement learning, 3 2017.

[SHCS17] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution Strategies as a scalable alternative to reinforcement learning. *arXiv (Cornell University)*, 1 2017.

[SWSS12] Yi Sun, Daan Wierstra, Tom Schaul, and Juergen Schmidhuber. Efficient natural evolution strategies. *arXiv (Cornell University)*, 1 2012.

[WSP+14] Daan Wierstra, Tom Schaul, Jan Peter, Jurgen Schmidhuber, and Yi Sun. Natural Evolution Strategies. *Journal of Machine Learning Research*, 15:949–980, 2014.

[YZ23] Haishan Ye and Tong Zhang. Mirror natural evolution strategies. *arXiv (Cornell University)*, 1 2023.