PART II. OPTIMIZATION: NUMERICAL APPROACHES
(LECTURE 3)

Shpilev Petr Valerievich
Faculty of Mathematics and Mechanics, SPbU

September, 2025

Санкт-Петербургский государственный университет

Comments

In this lecture, we focus on stochastic, population-based methods for global optimization. We begin with the cross-entropy method, discussing its core idea, implementation, and search process, as well as limitations arising from Gaussian approximations. We then introduce natural evolution strategies (NES), deriving the log-likelihood gradient for multivariate Gaussians and presenting the algorithm's mechanics. Building on this, we study covariance matrix adaptation evolution strategies (CMA-ES), examining mean updates, step-size control, covariance adaptation, and practical behavior, with comparisons to the cross-entropy approach. The second part of the lecture introduces broader population-based methods, starting from initialization strategies—including heavy-tailed distributions—and then moving to genetic algorithms. We cover chromosome representations, initialization, selection mechanisms, and tournament selection, emphasizing how evolutionary operators drive exploration and exploitation in complex search spaces.

Cross-Entropy Method: Basic Idea

Main Principle: The algorithm maintains and iteratively updates a probability distribution $P^{(k)}$ to focus search around the optimum.

Iteration structure:

- Sample m candidate solutions $x^{(1)}, \ldots, x^{(m)}$ from $P^{(k)}$
- Evaluate objective function $f(x^{(i)})$ for each candidate
- Select the top m_{elite} candidates forming the elite set
- Fit a new distribution $P^{(k+1)}$ to the elite set (typically via MLE)

Typical choice: Multivariate Gaussian distribution $\mathcal{N}(\mu^{(k)}, \Sigma^{(k)})$

- Mean: $\mu^{(k+1)} = \frac{1}{m_{elite}} \sum_{i=1}^{m_{elite}} x^{(i)}$
- Covariance: $\Sigma^{(k+1)} = \frac{1}{m_{elite}} \sum_{i=1}^{m_{elite}} (x^{(i)} - \mu^{(k+1)})(x^{(i)} - \mu^{(k+1)})^T$

Goal: As $k \to \infty$, the distribution $P^{(k)}$ converges to a Dirac mass at the global minimizer of f.

Comments

The cross-entropy method is a general stochastic optimization strategy. At each iteration, we generate a set of candidate solutions from a probability distribution over the search space. The specific choice of distribution may vary — common options include multivariate normal, Bernoulli, or others — depending on the problem structure.

After evaluating all samples using the objective function, we select the best-performing fraction — known as the elite set. The goal is then to update the parameters of the distribution so that future samples are more likely to resemble these elite ones.

A common approach is to use maximum likelihood estimation to fit the new distribution to the elite samples. If we assume a multivariate normal distribution, the maximum likelihood estimates for the mean and covariance are simply the sample mean and sample covariance of the elite points.

This gives us the update rules shown on the slide:

The new mean μ_k is the average of the elite samples.

The new covariance Σ_k is their empirical covariance matrix.

These updates allow the algorithm to progressively focus the sampling distribution on promising regions, while still maintaining some randomness to explore the space.

Cross-Entropy Method: Implementation and Example

Algorithm 13: Cross-Entropy Method.

```julia
using Distributions
function cross_entropy_method(f, P, k_max, m=100, m_elite=10)
    for k in 1:k_max
        samples = rand(P, m)  # Generate m samples from current distribution
        order = sortperm([f(samples[:, i]) for i in 1:m])  # Sort by f-value
        P = fit(typeof(P), samples[:, order[1:m_elite]]) #Update distribution
    end
    return P
end
```

```julia
import LinearAlgebra: norm
seed!(0)                                # Set seed for reproducibility
f = x -> norm(x)                        # Objective: Euclidean norm
μ, Σ = [0.5, 1.5],[1.0 0.2; 0.2 2.0] # Initial Gaussian parameters
P = MvNormal(μ, Σ)                      # Initial sampling distribution
k_max = 10
P = cross_entropy_method(f, P, k_max)
```

After k = 10: $\mu \approx [-6.14 \cdot 10^{-7}, -1.37 \cdot 10^{-6}]$

Comments

Here we see a Julia implementation of the Cross-Entropy Method.

The function `cross_entropy_method` takes a target function f, an initial sampling distribution P, and the number of iterations k_{max}. On each iteration, it generates m random samples from the distribution P. These samples are evaluated using the function f, and the best m_{elite} samples — those with the lowest values — are selected.

Then comes the key step: we update the distribution by fitting it to the elite samples. This is done by the function fit. This function comes from the Distributions library. It constructs a new distribution of the same type as P — for example, a multivariate normal — and adjusts its parameters based on the sample of elite points. By default, it uses maximum likelihood estimation. For a multivariate normal, this means computing the sample mean and the sample covariance matrix.

In the example in the second half of the slide, we minimize the Euclidean norm, which reaches its minimum at the origin. The starting distribution is a two-dimensional normal centered at a point away from zero. After running the algorithm for ten iterations, the mean of the distribution moves closer to the origin.

Also note: this code uses standard Julia syntax — indexing starts at one.

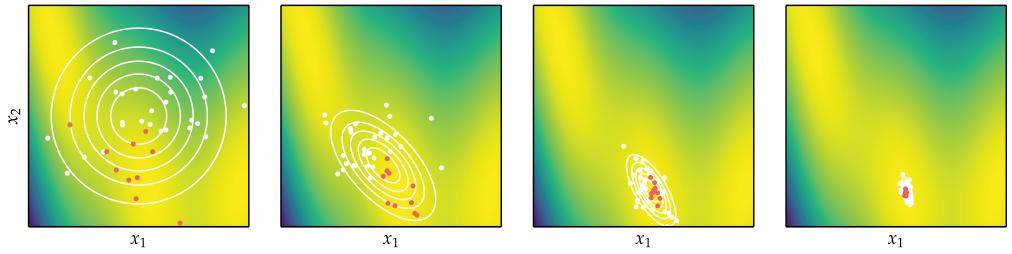


Figure: The cross-entropy method with m = 40 applied to the Branin function using a multivariate Gaussian proposal distribution. The 10 elite samples in each iteration are in red.

The Branin function is a two-dimensional function,

$$f(x) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1 - t)\cos(x_1) + s$$

with recommended values $a = 1$, $b = \frac{5.1}{4\pi^2}$, $c = \frac{5}{\pi}$, $r = 6$, $s = 10$, and $t = \frac{1}{8\pi}$.

Comments

This figure illustrates the cross-entropy method applied to the two-dimensional Branin function, a classic benchmark in global optimization. The method begins by sampling from an initial multivariate normal distribution. Each iteration generates 40 candidate points — shown in black — and evaluates the objective function at those points.

From these, the 10 best candidates, called elite samples, are selected — highlighted in red. These elite points guide the update of the sampling distribution: we estimate a new mean and covariance based on them using maximum likelihood. As a result, the distribution gradually shifts and contracts toward regions of lower function values.

In this example, we can clearly see the progression over iterations: the cloud of points narrows and centers around the lower valleys of the Branin function, where its global minima lie. Multiple elite clusters may appear early on due to the multimodal nature of the function, but over time the distribution concentrates near a single mode — one of the global optima.

This visual makes it clear how the cross-entropy method combines global exploration with adaptive focusing based on past performance — an elegant balance between randomness and learning.

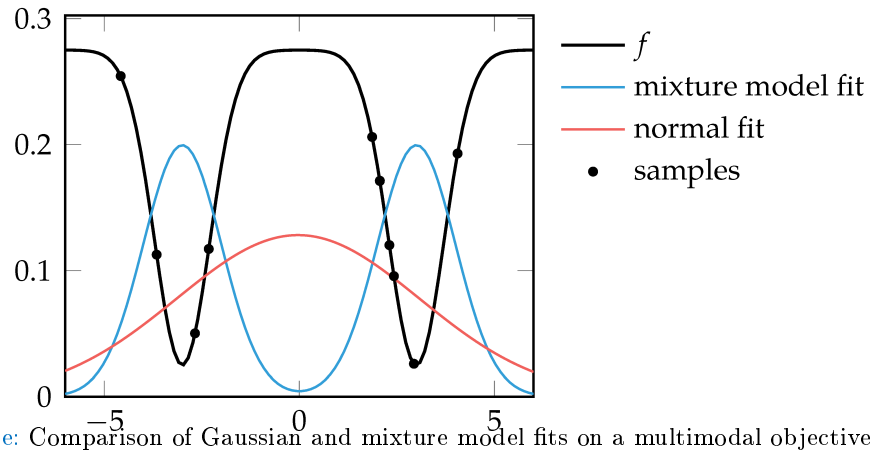


Figure: Comparison of Gaussian and mixture model fits on a multimodal objective function.

Example: A normal distribution fails to represent multiple local minima, while mixture models can simultaneously focus on several modes.

Comments

This figure illustrates an important limitation of the cross-entropy method: the success of the optimization depends heavily on the chosen distribution family. When using a simple multivariate normal distribution, it may fail to adequately represent multimodal objective functions — functions with multiple local minima. The figure shows that the normal distribution centers its density between the minima, thus missing both. In contrast, a mixture model — a mixture of Gaussians — can split its mass and focus on multiple promising regions at once. This makes it better suited for problems with multiple optima. Therefore, it is crucial to choose a distribution family that is flexible enough to model the structure of the objective function when applying the cross-entropy method.

Goal: Minimize the expected objective value under a parameterized distribution $p(x \mid \theta)$:

$$\min_{\theta} \mathbb{E}_{x \sim p(\cdot \mid \theta)}[f(x)]$$

Gradient Estimation via Log-Likelihood Trick:

$$\nabla_\theta \mathbb{E}_{x \sim p}[f(x)] = \mathbb{E}_{x \sim p}\left[f(x) \nabla_\theta \log p(x \mid \theta)\right] = \int \frac{p(x|\theta)}{p(x|\theta)} \nabla_\theta p(x|\theta) f(x) dx =$$

$$\int p(x|\theta) \nabla_\theta \log p(x|\theta) f(x) dx \approx \frac{1}{m} \sum_{i=1}^{m} f(x^{(i)}) \nabla_\theta \log p(x^{(i)} \mid \theta)$$

Key Characteristics:

- No gradient of $f(x)$ is required.
- Uses samples to estimate gradient.
- Black-box optimization friendly.

Requirements:

- $p(x \mid \theta)$ differentiable w.r.t. θ.
- $\nabla_\theta \log p(x \mid \theta)$ tractable.
- Efficient sampling from $p(x \mid \theta)$.

Comments

The key idea behind the Natural Evolution Strategies method is to replace the task of minimizing the objective function with the task of minimizing its expected value under a chosen probability distribution. That is, instead of looking directly for the best point, we learn a distribution that is more and more concentrated around good solutions.

This approach is especially useful when the objective function is non-differentiable, noisy, or black-box. Since we work with the expected value, we can still use gradient descent — but not in the space of inputs. Instead, we perform gradient descent in the space of parameters of the distribution.

The way we compute this gradient is based on a simple identity from probability theory. It allows us to rewrite the gradient of the expectation in a special form — as another expectation, which does not require us to differentiate the objective function at all. What we need instead is the gradient of the log-probability density of the distribution we use, like the normal distribution. This is always known analytically.

Practically, at each iteration we sample multiple candidate solutions from the current distribution. We evaluate the objective function at each of these samples, and then we form a gradient estimate — a direction in the space of distribution parameters that leads toward lower expected objective values.

After this, we perform a standard gradient descent update on the distribution parameters. As a result, the distribution gets gradually "pulled" toward areas in the search space where the objective function tends to be smaller.

This is what makes the method powerful: we never need to compute gradients of the function we are optimizing. Instead, we only need to evaluate it at different points. This makes the method ideal for problems where traditional gradient-based techniques are inapplicable.

Log Likelihood Gradient for Multivariate Gaussian

Distribution: Multivariate normal $\mathrm{p}(\mathrm{x} \mid \mu, \Sigma)$

$$\mathrm{p}(\mathrm{x} \mid \mu, \Sigma) = (2\pi)^{-\mathrm{d}/2} |\Sigma|^{-1/2} \exp\left(-\frac{1}{2}(\mathrm{x}-\mu)^{\mathrm{T}} \Sigma^{-1} (\mathrm{x}-\mu)\right)$$

Log Likelihood:

$$\log \mathrm{p}(\mathrm{x} \mid \mu, \Sigma) = -\frac{\mathrm{d}}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}(\mathrm{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathrm{x}-\mu)$$

Gradient w.r.t. mean μ:

$$\nabla_{\mu} \log \mathrm{p}(\mathrm{x} \mid \mu, \Sigma) = \Sigma^{-1}(\mathrm{x}-\mu)$$

Gradient w.r.t. covariance Σ:

$$\nabla_{\Sigma} \log \mathrm{p}(\mathrm{x} \mid \mu, \Sigma) = \frac{1}{2}\Sigma^{-1}(\mathrm{x}-\mu)(\mathrm{x}-\mu)^{\mathrm{T}}\Sigma^{-1} - \frac{1}{2}\Sigma^{-1}$$

Note on updating Σ: Directly updating Σ via its gradient can break positive definiteness. A common solution is to reparameterize $\Sigma = \mathrm{A}^{\top}\mathrm{A}$, and update A instead. The corresponding gradient is:

$$\nabla_{\mathrm{A}} \log \mathrm{p}(\mathrm{x} \mid \mu, \mathrm{A}) = \mathrm{A}\left[\nabla_{\Sigma} \log \mathrm{p} + \nabla_{\Sigma} \log \mathrm{p}^{\top}\right]$$

Comments

This slide presents the exact expressions for the gradient of the log-likelihood of a multivariate normal distribution — with respect to the mean vector and the covariance matrix. These gradients are essential for implementing the Natural Evolution Strategies algorithm.

The gradient with respect to the mean is relatively straightforward: it tells us how to shift the mean vector to increase the likelihood of a given sample under the current distribution. In NES, this is used to compute a descent direction for updating the mean. Since this gradient is a vector, it can be directly incorporated into a gradient descent step.

The situation with the covariance matrix is more delicate. The gradient here is a matrix that points in the direction in which the shape of the distribution should change to improve its performance. However, the covariance matrix must always remain symmetric and positive definite. A naive gradient step — where we subtract the scaled gradient from the matrix — can easily violate these properties.

To avoid this issue, NES implementations do not usually update the covariance matrix directly. Instead, they represent it in a factorized form — for example, as the product of a matrix with its transpose. Then the update is performed on the factor, not the covariance itself. This guarantees that the resulting covariance matrix remains valid.

Alternatively, some methods, like CMA-ES, use specific update rules that are designed to preserve positive definiteness. In simpler NES variants, the covariance matrix may even be fixed, or restricted to be diagonal.

In summary, while these gradients provide the theoretical foundation for NES updates, practical implementations must handle the covariance matrix with care to ensure that the distribution remains well-defined.

Algorithm 14: NES with reparameterized covariance.

```
using Distributions, LinearAlgebra
function natural_evolution_strategies(f, μ, A, k_max; m=100, α=0.01)
    for k in 1:k_max
        Σ = A' * A                          # Current covariance matrix
        P = MvNormal(μ, Symmetric(Σ))
        samples = [rand(P) for i in 1:m]
        g_μ = sum(f(x) * (Σ \ (x - μ)) for x in samples) / m
        μ -= α * g_μ                        # Update mean
        g_A = zeros(size(A))  # Estimate gradient for A
        for x in samples
            dx = x - μ
            g_Σ = 0.5 * (Σ \ dx * dx' * Σ \ I - Σ \ I)
            g_A += f(x) * A * (g_Σ + g_Σ')  # Symmetric part
        end
        A -= α * g_A/m                      # Update covariance factor
    end
    return μ, A
end  # In Julia  Σ \ (x - μ)  ≡  inv(Σ) * (x - μ)
```


Comments

This slide presents a compact implementation of the Natural Evolution Strategies algorithm for optimizing black-box functions using a multivariate Gaussian distribution. The algorithm maintains two key parameters: the mean vector, which controls where the samples are centered, and a matrix that defines the covariance structure. The covariance matrix itself is never updated directly. Instead, we work with a matrix factor whose product with its own transpose gives us a valid covariance at each iteration. This ensures the covariance stays symmetric and positive definite.

In each iteration, we first construct the current distribution using the mean and the derived covariance matrix. Then we sample a batch of points from this distribution. For each sampled point, we evaluate the objective function and compute the gradient of the log-density with respect to the parameters. These gradients are weighted by the function value at each sample — this focuses the update on the regions where the objective performs better.

The mean is updated using a standard gradient descent step. For the covariance, we don't update the full matrix directly. Instead, we estimate the gradient with respect to the covariance, and then use the chain rule to backpropagate that gradient to the matrix factor. This trick guarantees that the updated matrix continues to define a valid distribution.

Note that all matrix operations use linear system solvers instead of explicitly inverting the covariance. This is both more numerically stable and computationally efficient. The result is a robust and fully differentiable algorithm that scales well and remains valid throughout the optimization process.

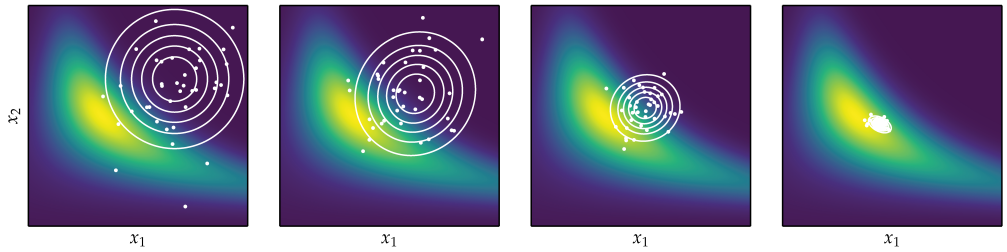


Figure: A few iterations of the natural evolution strategies with multivariate Gaussian distributions applied to Wheeler's Ridge function.

Wheeler's ridge function is a two-dimensional function with a single global minimum in a deep curved peak

$$f(x) = -\exp\left(-(x_1 x_2 - a)^2 - (x_2 - a)^2\right)$$

with a typically equal to 1.5, for which the global optimum of -1 is at $\left[1, \frac{3}{2}\right]$.

Comments

This slide shows a visualization of Natural Evolution Strategies applied to a two-dimensional test function known as Wheeler's Ridge. The surface has a narrow, curved valley with a single global minimum located near the point one comma one point five. The function drops off steeply in some directions and more gently in others, which makes it a good example for testing the behavior of adaptive algorithms.

We can see several iterations of the algorithm, each represented by an ellipse that illustrates the shape of the current sampling distribution. The center of the ellipse corresponds to the mean of the Gaussian, while its shape and orientation reflect the covariance. Initially, the distribution is wide and roughly circular, allowing for broad exploration. As the iterations proceed, the ellipse narrows and elongates along the valley, adapting to the structure of the objective function.

This behavior demonstrates one of the strengths of Natural Evolution Strategies: its ability to reshape the sampling distribution over time to better follow the landscape of the function being optimized. Instead of relying on point-wise updates, the algorithm gradually learns where promising regions are located and focuses its search accordingly.

By the final iteration, the distribution is tightly focused around the optimum, and the algorithm effectively converges. This makes NES particularly well-suited for problems where gradient information is unavailable or unreliable, but the overall shape of the function can still be exploited through sampling.

Covariance Matrix Adaptation (CMA-ES)

Key idea: Iteratively adapt the parameters of a multivariate normal distribution to minimize the objective function.

Maintains:

- Mean vector μ
- Covariance matrix Σ
- Global step size scalar σ

At each iteration:

1. m points sampled from $\mathcal{N}(\mu, \sigma^2\Sigma)$
2. Points sorted by objective values $\mathrm{f}(\mathrm{x}^{(1)}) \leq \cdots \leq \mathrm{f}(\mathrm{x}^{(\mathrm{m})})$
3. New mean: weighted average of top $\mathrm{m}_{\mathrm{elite}}$ samples

- Covariance and step size are updated adaptively
- CMA-ES emphasizes adaptation based on ranking, not function values

Sampling rule: $\mathrm{x}^{(\mathrm{i})} \sim \mathcal{N}(\mu^{(\mathrm{k})}, (\sigma^{(\mathrm{k})})^2\Sigma^{(\mathrm{k})})$

Comments

Now, we consider Covariance Matrix Adaptation (CMA-ES), a powerful black-box optimization algorithm. CMA-ES iteratively adapts a multivariate Gaussian distribution to efficiently search for a function's global minimum. The key insight is to iteratively modify not only the center of the distribution, but also its shape and overall scale.

The distribution is defined by three parameters: the mean vector, which guides the location of sampling; the global step size, which controls how far we explore around the mean; and the covariance matrix, which determines the shape and orientation of the search region.

At each iteration, a batch of points is sampled from a Gaussian distribution. These samples are evaluated using the objective function and then sorted. The top-performing points — called the elite samples — are used to compute a new mean, which steers the distribution toward better regions of the search space.

In addition to updating the mean, the algorithm adapts the covariance matrix, which allows it to effectively learn the directions of sensitivity in the objective function landscape and adjust its search accordingly. The global step size controls the overall spread of the samples, and is adjusted separately using a cumulation process.

Compared to NES, CMA-ES has a key advantage: it does not require estimating gradients. It uses only the ranking of sampled points. This simplifies implementation and reduces sensitivity to noisy evaluations. However, CMA-ES comes with a cost: it typically requires more samples per iteration than NES, especially in high-dimensional spaces.

CMA-ES: Mean Update and Weighting Scheme

New mean: weighted combination of top m_{elite} samples

$$\mu^{(k+1)} = \sum_{i=1}^{m_{elite}} w_i x^{(i)}$$

$$\sum_{i=1}^{m_{elite}} w_i = 1, \quad w_1 \geq w_2 \geq \cdots \geq w_{m_{elite}} \geq 0$$

Weighting recommendation:

$$w_i' = \ln\left(\frac{m+1}{2}\right) - \ln(i), \quad i = 1, \ldots, m$$

Normalize positive and negative weights separately.

- Weights emphasize better samples (higher rank).
- This generalizes the cross-entropy method (CEM), which uses uniform weights.
- Non-elite weights may be negative so all the weights approximately sum to 0:

$$\sum_{i=1}^{m} w_i \approx 0$$

Comments

Once the candidate points are sampled and evaluated, the next step in CMA-ES is to form a new mean vector. This is done using a weighted average of the top-ranked samples, where the weights are non-negative, sorted in descending order, and sum to one.

The intuition is simple: samples with better fitness values should have a stronger influence on the new direction of search. The recommended weights are derived from a logarithmic formula that favors the top individuals and decreases smoothly with rank.

Interestingly, CMA-ES allows the use of negative weights for non-elite samples. These are included in the full vector of weights $w_1, \ldots, w_m$ and their sum is approximately zero, helping to maintain diversity. This makes CMA-ES more flexible than the simpler cross-entropy method, which only uses uniform weights over elites.

This weighted recombination is one of the core innovations in CMA-ES that allows it to quickly steer the search distribution toward promising regions.

Cumulative evolution path:

$$p_\sigma^{(1)} = 0, \quad p_\sigma^{(k+1)} \leftarrow (1 - c_\sigma) p_\sigma^{(k)} + \sqrt{c_\sigma (2 - c_\sigma) \mu_{\text{eff}}} \cdot (\Sigma^{(k)})^{-1/2} \delta_w$$

$$\delta_w = \sum_{i=1}^{m_{\text{elite}}} w_i \cdot \frac{x^{(i)} - \mu^{(k)}}{\sigma^{(k)}}, \quad \mu_{\text{eff}} = \frac{1}{\sum_{i=1}^{m_{\text{elite}}} w_i^2}$$

Step size update:

$$\sigma^{(k+1)} \leftarrow \sigma^{(k)} \cdot \exp\left(\frac{c_\sigma}{d_\sigma} \left(\frac{\|p_\sigma^{(k+1)}\|}{\mathbb{E}\|\mathcal{N}(0, I)\|} - 1 \right) \right)$$

$$\mathbb{E}\|\mathcal{N}(0, I)\| = \sqrt{2} \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} \approx \sqrt{n} \left(1 - \frac{1}{4n} + \frac{1}{21n^2} \right)$$

- μ_{eff} controls the effective population size
- If steps (i.e. $\|p_\sigma^{(k+1)}\|$) are consistently large, σ increases; if small, σ decreases.

Comments

In the CMA-ES algorithm, the global step size — denoted by σ — determines the overall scale of the sampling distribution. To adapt this step size dynamically, CMA-ES introduces an auxiliary vector called p_σ, which tracks the direction and consistency of recent steps.

This vector is computed using what is known as an exponential moving average. At each iteration, we first calculate the weighted average of the best sampled points, measured relative to the previous mean and normalized by dividing by the current step size sigma. This gives a direction vector called δ_w, which reflects the typical movement of elite samples.

Then, the new value of p_σ is obtained by combining the previous value of p_σ with the current δ_w vector, scaled appropriately. The update rule uses a decay parameter, which controls how much influence recent steps have compared to older ones.

Importantly, before combining the vectors, the δ_w is transformed by applying the inverse square root of the covariance matrix. This transformation corresponds to a standardization of the Gaussian process — that is, it removes the current scale and orientation of the distribution. As a result, the behavior of the step size adaptation becomes independent of how stretched or rotated the sampling distribution is.

The key idea is that if the search steps behave like random noise, the norm of p_σ — that is, its Euclidean length — will remain close to the expected value for a standard normal vector. In an n-dimensional space, this expected length is approximately equal to the square root of n, corrected with small terms for better accuracy. But if the steps start showing consistent movement in one direction, the norm of p_σ grows beyond that baseline. This signals that the distribution is progressing steadily, and the step size sigma is increased to speed up the search. Conversely, if the steps are short and inconsistent, the norm of p_σ becomes small, and sigma is reduced to prevent excessive spread.

To implement this, the new sigma is computed by multiplying the previous sigma by the exponential of a factor that compares the norm of p_σ to its expected value. This smooth, multiplicative update ensures stability, continuity, and responsiveness to the quality of the search trajectory.

Constants for step size control:

$$c_\sigma = \frac{\mu_{\text{eff}} + 2}{n + \mu_{\text{eff}} + 5}, \quad d_\sigma = 1 + 2\max\left(0, \sqrt{\frac{\mu_{\text{eff}} - 1}{n+1}} - 1\right) + c_\sigma$$

Cumulative path for covariance matrix:

$$p_\Sigma^{(k+1)} \leftarrow (1 - c_\Sigma) p_\Sigma^{(k)} + h_\sigma \sqrt{c_\Sigma (2 - c_\Sigma) \mu_{\text{eff}}} \cdot \delta_w$$

$$h_\sigma = \begin{cases} 1 & \text{if } \frac{\|p_\sigma\|}{\sqrt{1-(1-c_\sigma)^{2(k+1)}}} < \left(1.4 + \frac{2}{n+1}\right) \mathbb{E}\|\mathcal{N}(0, I)\| \\ 0 & \text{otherwise} \end{cases}$$

- p_Σ accumulates directional information across steps
- h_σ prevents over-aggressive updates when step sizes are small
- the constant c_Σ will be defined further
- These updates shape the covariance matrix adaptively based on path history

Comments

This slide presents the constants used in CMA-ES for controlling the step size: c_σ and d_σ. The value of c_σ determines how fast the evolution path decays, while d_σ serves as a damping parameter that controls the adaptation rate of the global step size.

We also introduce the evolution path for the covariance matrix, denoted p_Σ. This path accumulates directional changes in the search trajectory, helping to guide the shape of the covariance matrix in future iterations. However, to avoid unstable updates when the step size is too small, an indicator function h_σ is used. If the norm of the step-size path exceeds a certain threshold, h_σ becomes zero and effectively stalls the adaptation of p_Σ.

Together, these mechanisms make the algorithm more robust and responsive to the search landscape.

CMA-ES: Covariance Matrix Update

Adjusted weights:

$$w_i^\circ = \begin{cases} w_i & \text{if } w_i \geq 0 \\ \frac{n w_i}{\|\Sigma^{-1/2}\delta^{(i)}\|^2} & \text{if } w_i < 0 \end{cases}$$

Covariance matrix update:

$$\Sigma^{(k+1)} \leftarrow \underbrace{(1 + c_1 c_\Sigma (1 - h_\sigma)(2 - c_\Sigma) - c_1 - c_\mu)}_{\text{(A) scaled previous}} \cdot \Sigma^{(k)} + \underbrace{c_1 p_\Sigma p_\Sigma^\top}_{\text{(B) rank-one update}} + \underbrace{c_\mu \sum_{i=1}^{\mu} w_i^\circ \, \delta^{(i)} (\delta^{(i)})^\top}_{\text{(C) rank-}\mu\text{ update}}$$

Parameter recommendations:

$$c_\Sigma = \frac{4 + \mu_{\text{eff}}/n}{n + 4 + 2\mu_{\text{eff}}/n}, \; c_1 = \frac{2}{(n+1.3)^2 + \mu_{\text{eff}}}, \; c_\mu = \min\left(1 - c_1, \; \frac{2(\mu_{\text{eff}} - 2 + 1/\mu_{\text{eff}})}{(n+2)^2 + \mu_{\text{eff}}}\right)$$

- Rank-one update exploits correlations in search direction
- Rank-μ update uses elite samples to reshape distribution
- Constants balance long-term directionality and population variance

Comments

This slide shows how the covariance matrix is updated in CMA-ES to reflect new information from the sampled solutions.

The update consists of three parts. The first part is the previous covariance matrix multiplied by a scaling factor. This factor may reduce the influence of the past depending on how stable the recent steps were. If the steps were stable, the term might even vanish entirely — that's controlled by the variable h_σ. This allows the algorithm to shift focus when confident in a new direction.

The second term is the so-called rank-one update. It uses the evolution path vector, which accumulates recent successful directions. Adding its outer product stretches the distribution along promising paths. This helps the algorithm exploit correlations in the search space.

The third term is the rank-mu update. It uses the best sampled steps from the current generation. Each step is scaled and weighted. Positive weights are used as is. Negative weights, however, are downscaled using the squared Mahalanobis distance of the corresponding step. This prevents unstable directions from having too much influence.

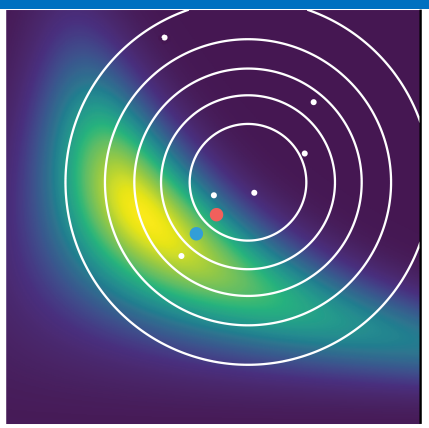
The three parts are all symmetric and carefully balanced using the constants at the bottom of the slide. These constants depend on the effective sample size, which reflects how many weights meaningfully contribute.

Even in the case where the coefficient in front of the old covariance matrix becomes zero, the result remains valid. The new matrix may have reduced rank for one iteration, but this is corrected over time as new samples are added. In practice, the algorithm includes safeguards to keep the matrix positive definite.

So, this update mechanism allows CMA-ES to gradually learn both the scale and shape of the underlying objective function.

CMA-ES vs Cross-Entropy: Mean Update

Figure: CMA-ES (blue) vs CEM (red) using 3 elite samples.



Key difference: CMA-ES assigns decreasing weights to all samples (some may be negative), while CEM uses equal weights on elite samples only.

- CMA-ES mean shifts more aggressively toward better samples
- CEM update is less sensitive to sample quality
- Negative weights in CMA-ES push away from bad solutions

Comments

This figure illustrates how CMA-ES and the Cross-Entropy Method update the mean after sampling. Both methods start from the same distribution and use the same six samples — shown as white dots — to perform the update.

The white ellipses are contour lines of the original Gaussian distribution. They visualize the shape and spread of the probability density. These ellipses are the same for both methods and represent just one distribution — no update to the covariance has happened yet.

The Cross-Entropy Method selects the three best samples and averages their coordinates using equal weights. This results in a modest shift of the mean, shown as the red point.

In contrast, CMA-ES uses all six samples. The weights decrease with rank: the best sample receives the largest positive weight, while the worst ones may even receive negative weights. As a result, the best points pull the mean toward better regions, and the worst points push it away from poor areas.

This leads to a more decisive shift in the mean, shown by the blue point. CMA-ES therefore responds more aggressively to sample quality and adapts more effectively to the structure of the problem.

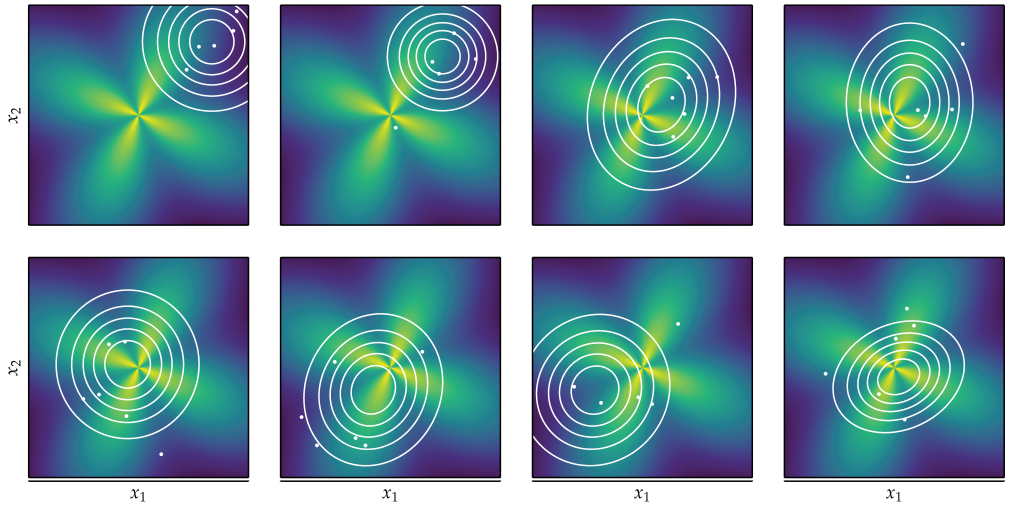


Figure: CMA-ES progress on a challenging non-separable function.

Comments

The figures on this and the next slide illustrate the full power of CMA-ES in action. The algorithm is applied to a difficult non-separable objective function with narrow, curved valleys. At each iteration, the algorithm updates both the mean vector and the covariance matrix.

You can see how the search distribution gradually adapts to the shape of the function. Initially, the covariance is spherical, but over time, it becomes increasingly elongated along the valley, allowing the algorithm to follow the function's structure. The mean shifts toward better solutions, guided by the weighted combination of elite samples. The step-size is also dynamic — it increases when progress is consistent and contracts near convergence.

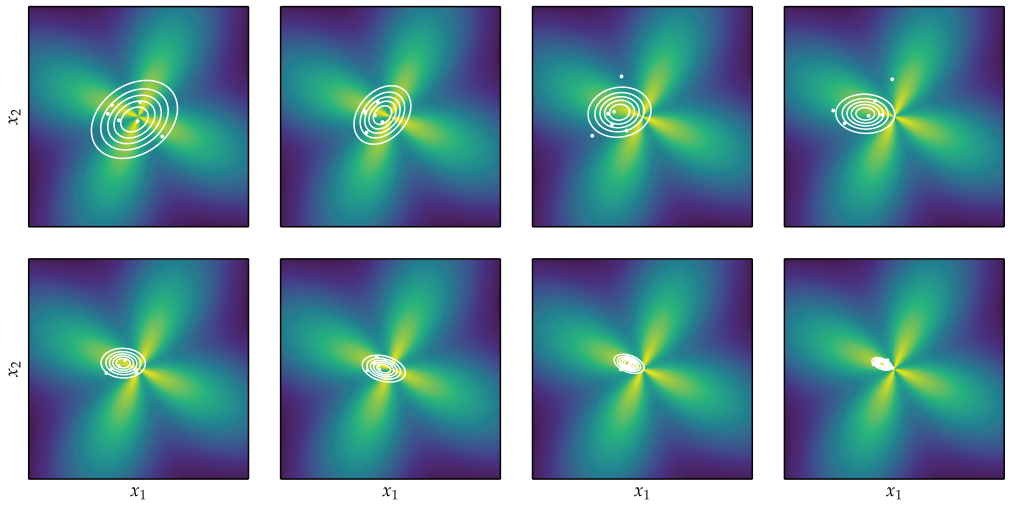


Figure: CMA-ES progress on a challenging non-separable function.

Comments

In the end, the search distribution contracts tightly around the optimal region, demonstrating that CMA-ES can discover and exploit the underlying geometry of the problem — even in complex, high-dimensional settings.

Algorithm 15: Covariance Matrix Adaptation Evolution Strategy (CMA-ES).

```
function covariance_matrix_adaptation(f, x, k_max;
    σ = 1.0,
    m = 4 + floor(Int, 3*log(length(x))),     # population size
    m_elite = div(m,2))                       # elite count

    μ, n = copy(x), length(x)                 # mean vector and dim
    ws = log((m+1)/2) .- log.(1:m)            # log-based weights
    ws[1:m_elite] ./= sum(ws[1:m_elite])      # normalize positive weights
    μ_eff = 1 / sum(ws[1:m_elite].^2)    # variance effective selection mass

    cσ = (μ_eff + 2)/(n + μ_eff + 5)          # cumulation constant
    dσ = 1 + 2max(0, sqrt((μ_eff-1)/(n+1))-1) + cσ # damping
    cΣ = (4 + μ_eff/n)/(n + 4 + 2μ_eff/n)     # learning rate for Σ
    c1 = 2/((n+1.3)^2 + μ_eff)                # learning rate rank-one
    cμ = min(1-c1, 2*(μ_eff-2+1/μ_eff)/((n+2)^2 + μ_eff)) # rank-μ rate
    ws[m_elite+1:end] .*= -(1 + c1/cμ)/sum(ws[m_elite+1:end]) # adjust neg.
```


Comments

This part of the code initializes the CMA-ES algorithm. We copy the starting point into the variable mu, which represents the center of the distribution. We also determine the dimension of the search space.

Then, we compute the weights. These are log-based and give higher importance to top-ranked samples. The expression "log of m plus one divided by two minus log of one to m" is applied elementwise using Julia's dot notation. In this language, a dot before a function or operator means it works elementwise on arrays.

Next, the positive weights — that is, those for the top-performing individuals — are normalized by dividing each of them by their total sum. This uses dot-equals syntax, which again applies the operation component-wise.

After that, we calculate the variance effective selection mass, denoted μ_{eff}. This value reflects how concentrated or spread out the weights are. If only one individual has a large weight, μ_{eff} is close to one. If weights are nearly equal, it grows toward the number of elites. This parameter controls how strongly the selection influences the distribution.

Finally, we set key hyperparameters — including learning rates for the step-size and the covariance matrix — and also rescale the negative weights. These are assigned negative values and adjusted so that their overall influence remains balanced. Again, dot syntax ensures the operations apply elementwise.

```
    E = n^0.5 * (1 - 1/(4n) + 1/(21*n^2))      # expected norm
    pσ, pΣ, Σ = zeros(n), zeros(n), Matrix(1.0I, n, n)  # state vectors

    for k in 1:k_max
        P = MvNormal(μ, σ^2 * Σ)               # sampling distribution
        xs = [rand(P) for i in 1:m]            # candidate solutions
        ys = [f(x) for x in xs]                # evaluate objective
        is = sortperm(ys)                      # sort by fitness

        δs = [(x - μ)/σ for x in xs]           # standardized steps
        δw = sum(ws[i] * δs[is[i]] for i in 1:m_elite)  # weighted sum
        μ += σ * δw                            # update mean

        C = Σ^(-0.5)                           # inverse sqrt of covariance
        pσ = (1 - cσ)*pσ + sqrt(cσ*(2 - cσ)*μ_eff) * C * δw
        σ *= exp(cσ/dσ * (norm(pσ)/E - 1))     # step-size update
```


Comments

Here begins the main loop of CMA-ES. At each step, we generate a new population by sampling candidate solutions from a multivariate Gaussian distribution. This distribution is centered at the current mean, scaled by the current step-size, and shaped by the current covariance matrix.

Each candidate is evaluated using the objective function. Then, we sort the candidates in ascending order of their objective values — the best ones come first.

For each candidate, we compute its normalized displacement from the current mean.Then, we update the mean by taking a weighted average of the displacements from the top-performing candidates.The better the candidate, the higher its weight in this average.

Next, we adapt the step-size. We update an auxiliary path called the evolution path.If this path becomes longer than expected, it means the search is moving in a consistent direction, so we increase the step-size. If the path is shorter, the search is more erratic, and the step-size is reduced.

This helps CMA-ES adjust its exploration dynamically — speeding up when things are going well, and slowing down when it needs caution.

```
        hσ = Int(norm(pσ)/sqrt(1 - (1 - cσ)^(2k)) <
                 (1.4 + 2/(n + 1)) * E)       # heuristic test
        pΣ = (1 - cΣ)*pΣ + hσ*sqrt(cΣ*(2 - cΣ)*μ_eff) * δw

        w0 = [ws[i] ≥ 0 ? ws[i] :             # adjust weights
              n * ws[i] / norm(C * δs[is[i]])^2
              for i in 1:m]

        Σ = (1 - c1 - cμ) * Σ +               # decay old covariance
            c1 * (pΣ * pΣ' + (1 - hσ) * cΣ * (2 - cΣ) * Σ) +
            cμ * sum(w0[i] * δs[is[i]] * δs[is[i]]'
                     for i in 1:m)            # weighted outer products

        Σ = triu(Σ) + triu(Σ, 1)'             # enforce symmetry
    end
    return μ                                  # return best mean
end
```


Comments

Now we adapt the covariance matrix — the key step that gives CMA-ES its power.

First, we compute a boolean flag h_σ to decide whether the evolution path indicates stable progress. This flag is set to 1 if the length of the path is below a threshold; otherwise, it is 0. This distinction helps the algorithm handle noisy or erratic behavior.

We then update a second evolution path, used specifically for the covariance matrix.This path accumulates information about the directions in which successful steps are taken.

Next comes the main update of the covariance matrix. This update has three parts:

First, a scaled version of the previous matrix — this keeps past information.

Second, a rank-one correction that adds directional structure based on the evolution path.

Third, a rank-μ update using all samples, where each sample contributes a weighted outer product of its displacement vector.

Positive weights reinforce directions from good samples. Negative weights penalize directions from poor ones — they "push back" to prevent degeneration. Importantly, the update formula ensures the covariance matrix remains symmetric and positive definite. At the end of each iteration, we enforce symmetry numerically to avoid round-off errors.

The algorithm runs for a fixed number of iterations k_{max} and returns the final mean μ as the best estimate of the optimum. In practice, one would often include stopping criteria — for example, if the step-size becomes too small, if the mean stops changing, or if the population loses diversity. Such criteria are not shown here but are crucial in real-world applications.

Population Methods: Motivation and Key Ideas

Earlier we discussed methods that generate solutions by sampling from an adaptively updated probability distribution.

Key Idea: In contrast, population-based methods maintain and evolve a set of candidate solutions—a *population*—directly across iterations.

Key Characteristics:

- Maintain a diverse population of candidates.
- Share information among individuals.
- Naturally explore the design space in parallel.
- Stochastic nature enables global search.

Typical structure: At each iteration:

- Generate new individuals based on current population.
- Evaluate objective function at each point.
- Use selection, recombination, and mutation to form the next population.

Advantages over single-point methods:

- Reduced risk of getting trapped in local minima.
- Better exploration-exploitation balance.
- Easy to parallelize for high performance.

Comments

So far, we've examined methods like the cross-entropy method, natural evolution strategies, and CMA-ES. These approaches generate new candidate solutions by sampling from a probability distribution whose parameters are updated over time based on observed performance.

We now turn to a fundamentally different class of methods, known as population methods.

In contrast to methods that adapt a distribution, population methods explicitly maintain a set of candidate solutions — called a population — and evolve this set directly from iteration to iteration. This leads to greater transparency and flexibility in how the search proceeds.

Each individual in the population represents a design point. The population is updated as a whole, using mechanisms inspired by biological evolution: selection, recombination, and mutation. These operations define how individuals are chosen and modified to produce the next generation.

A key advantage of this approach is that it maintains diversity. Different individuals can explore different regions of the search space simultaneously, reducing the risk of getting trapped in local minima. It also allows for a more flexible exploration-exploitation balance and naturally fits parallel computing environments.

This combination of diversity, structure, and adaptability makes population methods especially well suited for global optimization in complex, high-dimensional spaces.

Population Initialization

Goal: Create an initial population that broadly covers the design space.

Uniform Sampling:

- Search region defined by bounds $a = (a_1, \ldots, a_d)$ and $b = (b_1, \ldots, b_d)$
- Each coordinate sampled independently from $U(a_i, b_i)$

```
function rand_population_uniform(m, a, b)
    d = length(a)
    return [a + rand(d).* (b - a) for i in 1:m]
end
```

Normal Sampling:

- Population sampled from multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$
- Covariance matrix Σ is typically diagonal

```
using Distributions
function rand_population_normal(m, μ, Σ)
    D = MvNormal(μ, Σ)
    return [rand(D) for i in 1:m]
end
```


Comments

To apply a population-based method, we first need to generate an initial population — a set of candidate solutions that will evolve over time. The quality of this initial population can significantly affect both convergence speed and the ability to find a global optimum.

The most straightforward approach is uniform sampling. Here, we assume that the design space is bounded and represented as a hyperrectangle, with lower and upper bounds a and b. Each coordinate of each individual is sampled independently from a uniform distribution between the corresponding a_i and b_i. This provides broad, unbiased coverage of the space and is a good default when no prior information is available.

An alternative is normal sampling, where individuals are drawn from a multivariate normal distribution with a given mean vector μ and covariance matrix Σ. Typically, Σ is chosen to be diagonal, so that the sampling is still axis-aligned but concentrated around μ. This is useful when we have some prior belief about where good solutions might be located.

In both cases, we generate m individuals, each represented as a vector in R^d. The code snippets show simple implementations in Julia — one using uniform sampling, the other using the MvNormal distribution from the Distributions package.

Choosing between uniform and normal sampling depends on the problem. If the region of interest is known, normal sampling can accelerate convergence. Otherwise, uniform sampling provides a safe, general-purpose starting point.

Cauchy Initialization:

- Population initialized from Cauchy distribution: $\mathcal{C}(\mathrm{x} \mid \mu, \gamma)$.
- Heavy-tailed: encourages wide exploration.
- Often used for global optimization to cover distant regions early.

Why not Gaussian?

- Gaussian: limited to narrow vicinity of the mean.
- Less likely to sample distant regions initially.
- Poor initial diversity can hinder global search.

Cauchy Sampling:

```
using Distributions
function rand_population_cauchy(m, μ, σ)
    n = length(μ)
    return [[rand(Cauchy(μ[j],σ[j])) for j in 1:n] for i in 1:m]
end
```

Comments

In global optimization, the way we initialize the population strongly affects the overall success of the search. A common mistake is to use Gaussian sampling by default. While convenient, the Gaussian distribution is sharply peaked — most samples cluster close to the mean. This leads to a narrow initial coverage of the search space and reduces the chance of discovering promising regions early on.

To address this, Cauchy distributions are often used for population initialization. The Cauchy distribution has heavy tails, meaning it produces extreme values more frequently. As a result, the initial population is spread out over a much wider region, increasing the likelihood of capturing diverse and distant solutions. This improves robustness and makes the optimization process less sensitive to the initial choice of mean. It's especially important in high-dimensional case or when the objective function has multiple local minima.

The Julia function shown here demonstrates how to sample a population from independent Cauchy distributions. The input μ is a vector specifying the mean for each coordinate, and σ controls the scale (spread). The outer loop for i in 1 to m constructs a population of m individuals. The inner loop for j in 1 to n generates each individual by drawing a value from a univariate Cauchy distribution centered at μ. This double loop structure allows us to create a population of m vectors, each of dimension n, where coordinates are sampled independently but follow the same distribution pattern.

This initialization step is particularly useful in the early exploration phase. Once enough information is gathered, later iterations may switch to tighter distributions, such as Gaussians, to refine the search more locally.

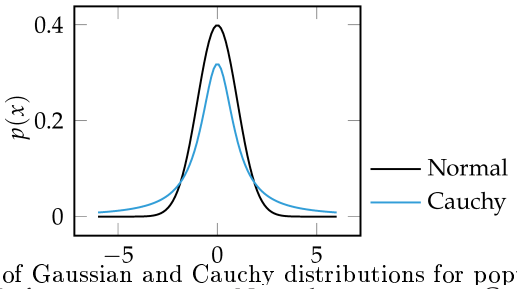


Figure: Comparison of Gaussian and Cauchy distributions for population initialization.

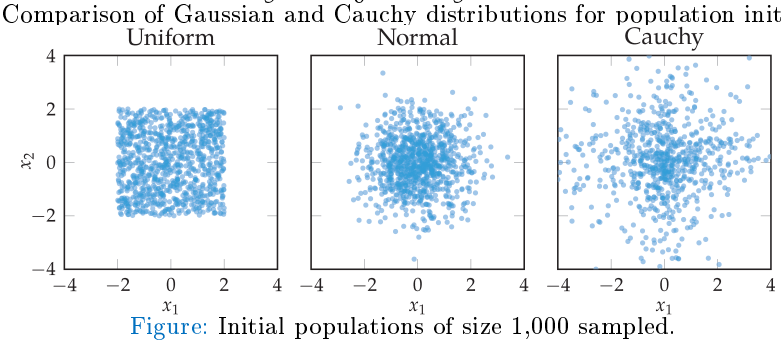


Figure: Initial populations of size 1,000 sampled.

Comments

To understand the impact of initialization distributions in population-based optimization, it's helpful to compare their shapes and sampling outcomes. The top figure shows the probability densities of the standard normal and Cauchy distributions. Although both are centered at zero and controlled by a scale parameter (standard deviation for the normal and γ for the Cauchy), they behave very differently. The normal distribution decays quickly as we move away from the mean, concentrating most of its mass in a narrow region. In contrast, the Cauchy distribution has much heavier tails — meaning it places significantly more probability mass far from the center. This makes it much more likely to generate extreme values.

The bottom figure illustrates this difference in practice. It shows three populations, each of 1,000 points in two dimensions, initialized using different distributions: uniform, normal, and Cauchy. The uniform distribution samples from a fixed box around the origin, giving a flat and bounded spread. The normal distribution, centered at the origin with identity covariance, produces a tight cloud of samples. Meanwhile, the Cauchy distribution creates a much more dispersed population — several samples fall far from the center, covering the space more broadly.

This comparison highlights the practical consequences of the initialization choice. Uniform sampling ensures bounded exploration, normal sampling provides tight concentration, while Cauchy sampling prioritizes broad coverage at the cost of some stability. For problems with multiple basins or deceptive landscapes, the wider initial spread of the Cauchy distribution can provide a strong exploratory advantage — helping to avoid early commitment to poor regions of the space.

Key idea: evolve a population through selection, crossover, and mutation.

Algorithm 16: Genetic Algorithm.

```
function genetic_algorithm(f, population, k_max, S, C, M)
    for k in 1:k_max
        parents = select(S, f.(population))
        children = [crossover(C, population[p[1]], population[p[2]])
                    for p in parents]
        population.= mutate.(Ref(M), children)
    end
    # In Julia argmin returns the index of the minimum element in an array
    return population[argmin(f.(population))]
end
```

- **S** – selection method
- **C** – crossover operator
- **M** – mutation operator
- Population evolves over `k_max` generations

Comments

Now that we have an initial population, the next step is to define how it evolves. Algorithm 16 presents a generic structure shared by many population-based optimization methods. The main idea is simple: we iteratively update the population, hoping that better solutions will emerge over time.

At each generation, we perform three key operations: selection, recombination, and mutation. These are abstract steps here — we'll examine specific implementations shortly. The purpose of selection is to identify promising individuals from the current population. These are used to generate new candidates during recombination, where their components are combined or blended in various ways. Mutation then introduces small random changes to increase variability and allow the algorithm to explore new regions.

This loop repeats for a fixed number of iterations or until some stopping criterion is satisfied.

Let's also look at how the algorithm is implemented in code. The function takes an objective f, an initial population, and the number of generations k_{max}, along with three operators: selection S, crossover C, and mutation M. The algorithm starts with a given population — a collection of candidate vectors. At each generation, we apply selection to choose the most promising individuals based on their function values. These selected candidates are paired and passed to the crossover function, which generates children — new candidate vectors. After that, the mutation function applies small random changes to each child to maintain diversity. These new vectors become the next generation. After repeating this process for k_{max} iterations, we return the best individual found so far. The line population evaluates the function on each vector and returns the one with the smallest value — that is, the current best solution.

This simple structure underlies many successful algorithms — including genetic algorithms, evolution strategies, and differential evolution — with differences mainly in how the three core steps are implemented.

Chromosome Representations

Binary Chromosomes:

- Chromosome = binary string, e.g., `10101001`.
- Easy to implement crossover and mutation.
- Can be generated via `bitrand(d)`.
- Not always straightforward to decode into valid design points.

Real-Valued Chromosomes:

- Chromosome = vector in $\mathbb{R}^d$, e.g., $[0.3, -1.1, 2.7]$.
- Directly corresponds to design point.
- Easier to interpret and use in continuous domains.

Figure: A chromosome represented as a binary string

Comments

Now that we've seen how population-based algorithms evolve a set of candidate solutions, it's time to look more closely at how each individual solution — or "individual" — is actually represented. In the terminology of genetic algorithms, each solution is called a chromosome. This term comes from the biological metaphor and reflects the idea that each solution carries genetic material that can be recombined and mutated to produce offspring.

There are two main ways to represent chromosomes in genetic algorithms: binary strings and real-valued vectors.

Binary string chromosomes are inspired by the structure of DNA and consist of sequences of zeroes and ones. These are easy to manipulate — for example, crossover can be performed by swapping segments of bits, and mutation can flip individual bits. However, decoding a binary string to a meaningful point in the design space isn't always simple. Sometimes, the resulting binary string may not correspond to a valid or useful solution.

To avoid these issues, many applications prefer real-valued chromosomes, especially when dealing with continuous optimization problems. A real-valued chromosome is simply a vector of real numbers and directly corresponds to a candidate solution in the design space. This makes the algorithm more intuitive and avoids additional encoding and decoding steps.

We'll see that the rest of the algorithm — selection, crossover, mutation — works with either form. But choosing the right representation often depends on the nature of the problem being solved.

Why initialization matters:

- Genetic algorithms start with a population of m candidate solutions.
- Initial diversity helps avoid premature convergence to local minima.
- A poor initialization can limit exploration and degrade performance.

Sampling strategies:

- Binary chromosomes: Each bit sampled independently: $P(1) = 0.5$.
- Real-valued chromosomes:
  - Uniform sampling from box $\mathcal{X} = [a_1, b_1] \times \cdots \times [a_n, b_n]$.
  - Normal sampling: $\mathcal{N}(\mu, \sigma^2 I)$ centered near a promising region.
  - Cauchy sampling: broad, heavy-tailed distribution for global exploration.

Good practice:

- Use uniform sampling as default when no prior knowledge is available.
- For known structure: tailor distribution to problem geometry.
- Ensure each individual is a valid chromosome (e.g., respects bounds or constraints).

Comments

Initialization is the first step in running any genetic algorithm, and its impact is often underestimated. Since we begin with a population of candidate solutions, their initial quality and diversity directly affect how well and how fast the algorithm explores the search space.

The idea is to spread out the individuals — the chromosomes — across the space so that we sample broadly rather than starting in just one region. If the initial population lacks diversity, the algorithm may converge prematurely to a suboptimal solution.

For binary chromosomes, the simplest approach is to assign each bit a 50/50 chance of being 0 or 1. For real-valued chromosomes, we have several choices. Uniform sampling across the feasible box is common and safe when we know little about the problem. Normal sampling lets us concentrate around a mean — helpful if we suspect where good solutions lie. Cauchy sampling, which we've seen earlier, allows for very wide exploration due to its heavy tails.

It's also important to ensure that all generated chromosomes are valid — that they satisfy any given constraints, variable bounds, or domain-specific rules.

Overall, the goal at this stage is to create a diverse and representative starting population. A good initialization lays the foundation for a successful evolutionary search.

Genetic Algorithms: Selection

Goal of selection:

- Choose individuals from current population to serve as parents.
- Favor individuals with better fitness (lower objective value).
- Preserve diversity by maintaining selection pressure.

Common selection schemes:

- **Tournament selection:** randomly select k candidates, pick the best.
- **Roulette wheel:** selection probabilities inversely proportional to the objective values.
- **Truncation selection:** take the top-performing fraction of the population — say, the best 20% — and use only them as parents.

Selection pressure:

- High pressure: fast convergence, risk of premature stagnation.
- Low pressure: more exploration, slower progress.
- Tradeoff must be tuned based on problem characteristics.

Comments

In genetic algorithms, the population is updated at each iteration by generating a new set of individuals. The first step is selection: choosing which individuals become parents.

Selection favors better individuals — those with lower objective values — but still allows others to be picked to maintain diversity. This helps balance exploitation and exploration.

There are several common selection methods.

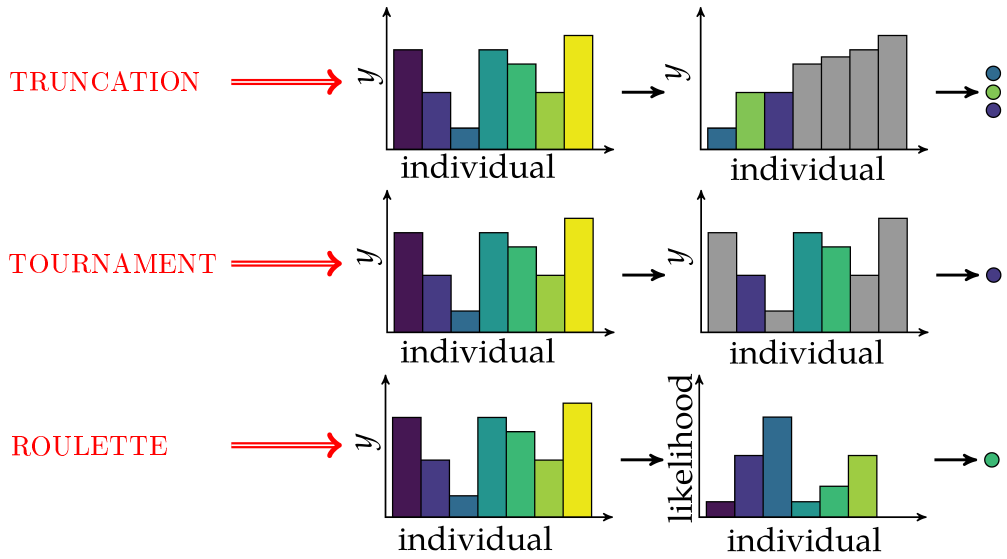
Tournament selection: We randomly divide the population into small groups — say, 2 or 3 individuals per group — and choose the best one in each group to become a parent. This is repeated many times until we have enough parents. Larger group sizes increase the chance of selecting better individuals, which creates stronger selection pressure.

Roulette wheel selection: Each individual gets a probability of being selected that depends on its performance — better individuals get higher chances. It's called "roulette wheel" because we imagine assigning each candidate a slice of a wheel proportional to its fitness. We spin the wheel repeatedly to pick parents.

Truncation selection: We simply take the top-performing fraction of the population — say, the best 20

The term selection pressure refers to how strongly the algorithm favors better individuals. High pressure means only the best get selected, but it can lead to premature convergence. Low pressure maintains diversity but may slow progress.

In all cases, the goal of selection is to create a pool of parents whose "genetic material" — that is, their vector coordinates — can be recombined and slightly modified to generate offspring. Ideally, the offspring will inherit the good properties of their parents and lead to improved solutions over time.



Comments

This slide illustrates how three common selection methods work on a small population of size m=7. Each vertical bar represents an individual, with bar height showing its objective value — the lower, the better. Colors identify individuals across methods.

In truncation selection, we rank the entire population and keep the top k=3 individuals as parents. This is a deterministic approach: we always select a certain percentage of the best performers. In this example, exactly three individuals are selected — they directly form the parent set.

In tournament selection, we repeatedly sample k=3 individuals randomly from the population. In each tournament, the best individual among the three is selected as a parent. Here, only one individual is selected per tournament. To obtain multiple parents (say, r parents), we repeat the tournament r times with fresh samples. This method introduces stochasticity and gives weaker individuals a chance to participate.

In roulette wheel selection, all individuals are assigned selection probabilities proportional to their fitness values. A random draw is made according to these probabilities. Again, a single parent is selected per draw. To get r parents, we perform r independent draws from the probability distribution shown on the right. In this example, the worst-performing individual has zero probability and cannot be selected.

In summary, truncation gives k=3 parents at once, while tournament and roulette selection give one parent per sampling trial, which can be repeated to collect as many parents as needed.

Selection Strategies in Genetic Algorithms

```julia
abstract type SelectionMethod end

# Select random pairs from top-k best individuals
struct TruncationSelection <: SelectionMethod
    k  # number of top individuals to keep
end
function select(t::TruncationSelection, y)
    p = sortperm(y)  # ranks by fitness (lower is better)
    return [p[rand(1:t.k, 2)] for i in y]
end

# Select parents with probability inversely proportional to fitness
struct RouletteWheelSelection <: SelectionMethod end
function select(::RouletteWheelSelection, y)
    y = maximum(y) .- y  # better scores get higher weights
    cat = Categorical(normalize(y, 1))
    return [rand(cat, 2) for i in y]
end
```


Comments

This slide illustrates how to implement two parent selection strategies in Julia. Both rely on the common abstract type `SelectionMethod`, which serves as a base type for all such strategies. This setup allows us to later write a generic select function that works with any method. The less colon symbol (<:) means that a specific strategy, such as `TruncationSelection`, is a subtype of `SelectionMethod`.

The function `select(t::TruncationSelection, y)` applies truncation selection. Here, t is the parameter object that holds the selection setting — in this case, the number k of top individuals to choose from. The input y is a vector of objective function values for the current population — one value per individual. The population is sorted by these values (from best to worst), and for each offspring, two parents are chosen randomly from the top k.

In contrast, the `RouletteWheelSelection` strategy doesn't need any parameters, so we omit the variable name and write `select(::RouletteWheelSelection, y)`. The double colon (::) still checks that the correct type is passed, but the unnamed argument indicates we don't need to access it. This method assigns higher selection probability to individuals with lower objective values. Since we are minimizing, we invert y by subtracting each value from the maximum. After normalization, we sample pairs from the resulting probability distribution.

Both versions return a list of index pairs that specify which individuals will act as parents in the next generation.

Tournament Selection Strategy

```julia
# Define tournament selection as a subtype of SelectionMethod
struct TournamentSelection <: SelectionMethod
    k  # size of the tournament
end

# Select best individual from k randomly chosen candidates
function select(t::TournamentSelection, y)
    getparent() = begin
        p = randperm(length(y))            # shuffle population
        p[argmin(y[p[1:t.k]])]             # pick best of k
    end
    return [[getparent(), getparent()] for i in y]
end
```

Key Idea: At each selection step, we randomly sample k individuals and select the one with the best fitness. This is repeated to generate each parent.

Comments

This implementation shows how to realize tournament selection in Julia. The type `TournamentSelection` is defined as a subtype of the abstract `SelectionMethod`, and contains a single field k, which determines how many individuals compete in each tournament.

The function `select(t::TournamentSelection, y)` takes two arguments: t, the selection strategy, and y, the vector of objective function values — one per individual in the population. Since we are minimizing, lower values of y indicate better individuals.

To select one parent, we randomly permute the indices of the population and take the best individual among the first k. This process is encapsulated in the helper function `getparent()`. We call it twice for each offspring to generate a pair of parents. Repeating this across the population gives us a list of parent pairs.

This method introduces randomness while still favoring better candidates. The larger k is, the stronger the selection pressure — meaning the method is more likely to choose the best individuals.