Lecture 14: Optimizers and Numerical Optimization

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

May 8, 2023

1 Introduction

Optimization problems arise in many fields of science and engineering. If one can build a numerical model that relates the input(s) and output(s) of this problem, then numerical optimization, which is a branch of applied mathematics, can help with finding the best solution to this very problem among a set of possible solutions. Briefly, in numerical optimization, the main aim is finding the values of one or more variables that minimize or maximize a given objective function (also known as cost function), $f(x)$ subject to certain constraints, i.e., $x \in X$, where a design point "x" can be represented as a vector of values corresponding to different design variables.

There are several numerical optimization algorithms that use different mathematical techniques to search for the best solution. These algorithms may be based on iterative methods, gradient descent, or other search strategies, and may involve the use of derivatives or other forms of mathematical analysis to guide the search process.

Optimization is a central component of machine learning, as it allows us to find the best set of *model parameters* that minimize the error between the predicted and actual output of the model. Optimization techniques such as gradient descent, stochastic gradient descent, and variants like Adam and RMSprop are commonly used in machine learning to update the model parameters iteratively until the cost function is minimized to an acceptable level. These methods allow us to optimize models with large numbers of parameters efficiently and effectively. The performance of a machine learning model depends critically on the quality of the optimization process. Poorly optimized models may overfit to the training data, leading to poor generalization performance on new data. On the other hand, well-optimized models can capture the underlying patterns in the data and generalize well to new, unseen data.

2 Gradient Based Numerical Optimization Methods

In gradient based numerical optimization, we pay a lot of attention to derivatives due to their important roles in finding the minimum or maximum value of a function. The first derivative of a function gives us information about the slope or gradient of the function at a particular point. In optimization, we use this information to find the direction in which the function is decreasing or increasing the most rapidly, which can be used to guide the search for the minimum or maximum of the function. Specifically, we can use gradient descent, a popular optimization algorithm, which takes steps proportional to the negative gradient of the function in order to find the minimum.

The second derivative of a function tells us about the curvature of the function at a particular point. If the second derivative is positive, the function is said to be convex and has a single minimum. Conversely, if the second derivative is negative, the function is said to be concave and has a single maximum. In optimization, we can use this information to determine whether a candidate's minimum or maximum is indeed the global minimum or maximum.

Univariate Functions: A univariate function is a mathematical function that takes a single variable as input and returns a single output. In other words, it is a function of only one variable. Univariate functions are commonly used in mathematics, physics, engineering, and other fields to model relationships between a single independent variable and a single dependent variable.

Mathematically, a univariate function can be written as $f(x) = y$, where x is the input variable or independent variable, y is the output variable or dependent variable, and f is the function that maps the input variable to the output variable.

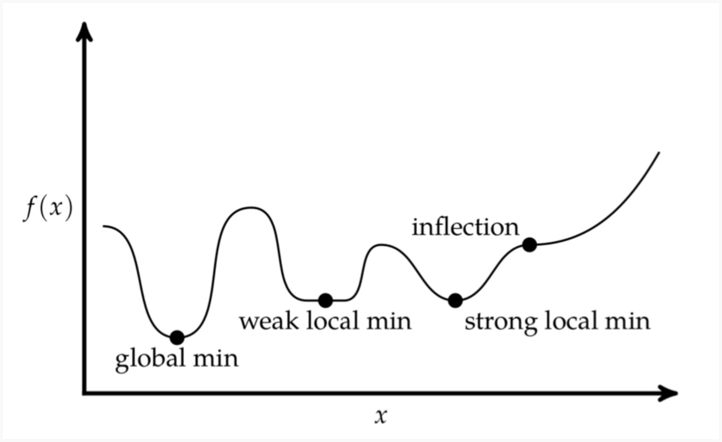


Figure 1: Critical points of a univariate function f that changes with x.

As shown in Fig. 1, there are different types of critical points and inflection points that can occur in a univariate function.

- *Global minimum* is the lowest point in the entire function's domain. A global minimum occurs when the function is lower than or equal to all other values in the entire domain of the function.
- *Weak local minimum* is another critical point where the function has the lowest value in the local region around that point, but there may be other points with lower values outside of that region. A weak local minimum occurs when the function is lower than or equal to all other nearby points in a small region around that point, but there are points with lower values outside of that region.
- *Strong local minimum* is another critical point where the function has the lowest value in the local region around that point, and there are no other points with lower values anywhere in the entire domain of the function. A strong local minimum occurs when the function is lower than all other nearby points in a small region around that point, and there are no points with lower values outside of that region.
- *Inflection point* is a point on the curve where the function changes from being concave upward to concave downward, or vice versa, i.e., the second derivative of the function changes sign at that point.

Multivariate Functions: A multivariate function takes multiple variables as inputs and returns a single output. In other words, it is a function of more than one variable. Multivariate functions are commonly used in mathematics, physics, engineering, and other fields to model complex relationships between variables.

Mathematically, a multivariate function can be written as:

$$f(x_1, x_2, \cdots, x_n) = y \tag{1}$$

where $x_1, x_2, \cdots, x_n$ are the input variables or independent variables, y is the output variable or dependent variable, and f is the function that maps the input variables to the output variable.

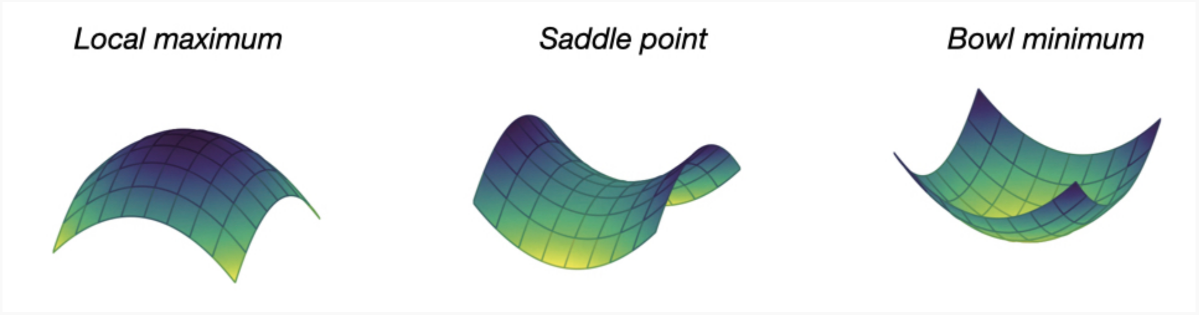


Figure 2: Multivariate functions and their local maximum (peak), saddle point (a plateau in the function's graph with a downward slope in some directions and upward slope in others), and bowl minimum (valley).

As shown in Fig. 2, there are different types of critical points that can occur in a multivariate function.

At a *local maximum*, the function has the highest value in the local region around that point. Mathematically, at a critical point, the gradient of the function is zero and a local maximum occurs when the Hessian matrix of second partial derivatives is negative definite at that point.

At a *saddle point*, the function has a flat region with both positive and negative slopes in different directions. Mathematically, a saddle point occurs when the Hessian matrix of second partial derivatives has both positive and negative eigenvalues at that point.

At a *bowl minimum*, the function has the lowest value in the local region around that point. This is the opposite scenario of having a local maximum, hence it requires the Hessian matrix of second partial derivatives to be positive definite at that point.

Some of the gradient based optimization algorithms (e.g., Newton-Raphson for finding the roots of a function) use both the first and second derivatives to find the minimum or maximum of a function. As previously mentioned, gradient descent is a popular optimization algorithm that is widely used in numerical optimization. There are several variants of gradient descent, including those with fixed learning rates (Steepest Descent, Conjugate Gradient) and those with varying learning rates (Momentum, Adam, RMSprop, etc.). Before we delve into these variants, let's first try to understand how these algorithms work in general.

3 A General Approach to Optimization

Local descent, also known as local optimization, is a technique used in numerical optimization to find the local minimum of a function by iteratively updating the current estimate of the minimum in the direction of *steepest descent*.

The basic idea behind local descent is to start with an initial guess of the minimum and then update it iteratively in the direction of the negative gradient of the function. The negative gradient points in the direction of steepest descent, which is the direction that the function decreases the most rapidly. By following the negative gradient direction, the algorithm moves towards a local minimum of the function. Various techniques, such as line search and trust region methods, can be used to determine the step size.

The local model may be obtained, for example, from a first- or second-order Taylor approximation. Here's a sample algorithm:

1. Check whether $\mathbf{x}^{(k)}$ satisfies the termination conditions, i.e. $f(\mathbf{x}^{(k)}) \overset{?}{<} \epsilon$, where ϵ is the desired threshold. If it does, terminate; otherwise proceed to the next step.
2. Determine the descent direction $\mathbf{d}^{(k)}$ using local information such as the gradient (or Hessian for multivariate optimization).
3. Determine the step size or learning rate $\alpha^{(k)}$. Note that if $\alpha^{(k)} == h$, then this is same as the Taylor approximation. Here, the idea is choose something smaller or larger to control the learning speed!.
4. Compute the next design point according to: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}$. Go back to step-1.

Note that this approach is valid for both single-variable and multi-variable optimization problems.

Local descent is particularly useful when the function has a well-defined local minimum, but the global minimum is unknown or hard to find. However, the algorithm can get trapped in a local minimum and fail to find the global minimum if the function is not convex. Also note that we just assumed that $\mathbf{d}$ is a valid descent direction, but we didn't say anything about how we can find a valid descent direction. There are several ways of finding a valid descent direction. Now let's learn about them.

4 Gradient Based Optimization Algorithms with Fixed Learning Rate

4.1 Steepest Descent

The basic idea of steepest descent is to start with an initial guess for the input variables, and then repeatedly update the values of the variables by taking steps in the direction of the negative gradient of the function with respect to the variables. The negative gradient points in the direction of steepest descent, which is the direction in which the function decreases most rapidly. By taking small steps in this direction, we can iteratively approach the minimum of the function.

Mathematically, the update rule for gradient descent is given by:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)}) \tag{2}$$

where $\mathbf{x}^{(k)}$ is the current value of the input variables, α is the step size (also called the learning rate), and $\nabla f(\mathbf{x}^{(k)})$ is the gradient of the function f with respect to the input variables evaluated at $\mathbf{x}^{(k)}$.

The choice of the learning rate is important, as it determines the size of the steps taken at each iteration. If the learning rate is too large, the algorithm may overshoot the minimum and diverge, while if it is too small, the algorithm may converge very slowly. Finding an appropriate learning rate is often done through trial and error, or by using more sophisticated techniques such as adaptive learning rates.

4.2 Conjugate Gradient

The conjugate gradient method can be used to solve linear systems of equations, as well as to perform numerical optimization. In numerical optimization, we use it to find the minimum or maximum of a quadratic function of the form:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{3}$$

where $\mathbf{x}$ is a vector of input variables, $\mathbf{A}$ is a symmetric positive-definite matrix, $\mathbf{b}$ is a vector, and c is a constant. Note that if $\mathbf{x}$ and $\mathbf{b}$ are $N \times 1$ and $\mathbf{A}$ is $N \times N$, then $f(\mathbf{x})$ is a scalar, i.e. 1×1!

Minimizing the quadratic function is equivalent to solving the linear equation $\mathbf{Ax} = \mathbf{b}$ and in order to achieve this, a powerful method is to find a sequence of N *conjugate directions* satisfying

$$(\mathbf{d}^{(i)})^T \mathbf{A}\mathbf{d}^{(j)} = 0 \quad (i \neq j), \tag{4}$$

and this is exactly the basic idea of the conjugate gradient method: iteratively searching for the minimum of the function along a set of conjugate directions. The algorithm starts with an initial guess for the minimum, and at each iteration, a new direction is chosen that is conjugate to the previous directions, meaning that it is orthogonal to all previous directions with respect to the matrix A. So, let's provide more content.

Assume $\mathbf{g}^{(k)}$ represents the gradient of f, i.e., $\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$. We start with the direction of steepest descent

$$\mathbf{d}^{(1)} = -\mathbf{g}^{(1)} \tag{5}$$

We then use line search to find the next design point. For quadratic functions $f = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c$, the step factor α can be computed as

$$\begin{aligned}\frac{\partial f(\mathbf{x}+\alpha d)}{\partial \alpha} &= \frac{\partial}{\partial \alpha}\left[\frac{1}{2}(\mathbf{x}+\alpha\mathbf{d})^T\mathbf{A}(\mathbf{x}+\alpha\mathbf{d}) + \mathbf{b}^T(\mathbf{x}+\alpha\mathbf{d}) + c\right] \\ &= \mathbf{d}^T\mathbf{A}(\mathbf{x}+\alpha\mathbf{d}) + \mathbf{d}^T\mathbf{b} \\ &= \mathbf{d}^T(\mathbf{Ax}+\mathbf{b}) + \alpha\mathbf{d}^T\mathbf{A}\mathbf{d}\end{aligned} \tag{6}$$

Let the gradient be zero,

$$\alpha = -\frac{\mathbf{d}^T(\mathbf{Ax}+\mathbf{b})}{\mathbf{d}^T\mathbf{A}\mathbf{d}} \tag{7}$$

Then the update is

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \alpha\mathbf{d}^{(1)} \tag{8}$$

For the next step

$$\mathbf{d}^{(k+1)}) = -\mathbf{g}^{(k+1)} + \beta^{(k)}\mathbf{d}^{(k)} \tag{9}$$

where $\beta^{(k)}$ is a series of scalar parameters. Larger values of β indicate that the previous descent direction contributes strongly.

We solve β, from the followings

$$(\mathbf{d}^{(k+1))^T}\mathbf{A}\mathbf{d}^{(k)} = 0 \tag{10}$$

$$(-\mathbf{g}^{(k+1)} + \beta^{(k)}\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)} = 0 \tag{11}$$

$$-\mathbf{g}^{(k+1)}\mathbf{A}\mathbf{d}^{(k)} + \beta^{(k)}(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)} = 0 \tag{12}$$

$$\beta^{(k)} = \frac{(\mathbf{g}^{(k+1)})^T\mathbf{A}\mathbf{d}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} \tag{13}$$

The conjugate method is exact for quadratic functions. But it can be applied to non quadractic functions as well when the quadratic function is a good approximation.

Unfortunately, we don't know the value of $\mathbf{A}$ that best approximate f around $\mathbf{x}^{(k)}$. So we need choose a way to compute β. In literature, there are two commonly used formulas.
Fletcher-Reeves:

$$\beta^{(k)} = \frac{(\mathbf{g}^{(k)})^T\mathbf{g}^{(k)}}{(\mathbf{g}^{(k-1)})^T\mathbf{g}^{(k-1)}} \tag{14}$$

Polak-Ribiere Formula:

$$\beta^{(k)} = \frac{(\mathbf{g}^{(k)})^T(\mathbf{g}^{(k)} - \mathbf{g}^{(k-1)})}{(\mathbf{g}^{(k-1)})^T\mathbf{g}^{(k-1)}} \tag{15}$$

The search is then performed along this new direction until a new minimum is found, and the process is repeated until convergence. The conjugate gradient method is indeed a popular and powerful optimization algorithm for quadratic functions, as it can converge to the minimum in a relatively small number of iterations, even for large systems of equations. However, it may not perform as well for non-quadratic functions, where other optimization algorithms such as gradient descent or Newton's method may be more effective.

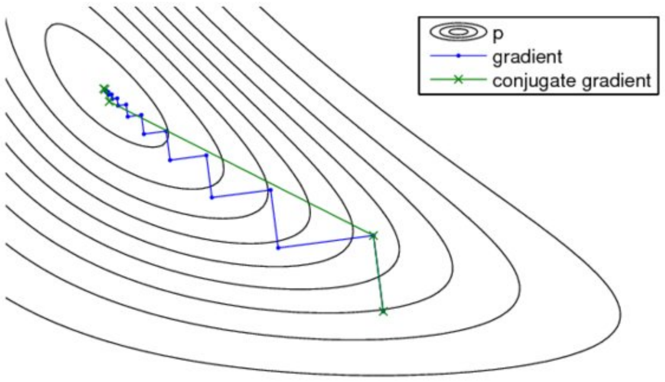


Figure 3: Convergence of steepest descent vs. conjugate gradient.

Steepest descent and conjugate gradient methods, which usually work with the line search methods. However, these standard versions might take a long time to traverse a nearly flat surface. To address this issue, several methods have been proposed as follows.

5 Gradient Based Optimization Algorithms with Varying Learning Rate

5.1 Momentum

The basic idea behind the momentum algorithm is to use a *"momentum"* term to keep track of the direction of the gradient descent updates and use this information to accelerate convergence and avoid oscillations.

In standard gradient descent, the update at each iteration is based only on the gradient of the objective function at the current point. In contrast, the momentum algorithm adds a fraction of the previous update direction to the current update direction, so that the algorithm gains momentum and continues moving in the same direction. This fraction is called the *"momentum coefficient"* and is typically a value between 0 and 1.

Specifically, the update equation for the momentum algorithm is:

$$\mathbf{v}^{(k)} = \beta\mathbf{v}^{(k-1)} + (1-\beta)\nabla f(\mathbf{x}^{(k)}) \tag{16}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\mathbf{v}^{(k)} \tag{17}$$

where $\mathbf{v}^{(k)}$ is the momentum at step k, β is the momentum coefficient, α is the learning rate, and $\nabla f(\mathbf{x}^{(k)})$ is the gradient of the objective function with respect to the parameters at step k.

Intuitively, the momentum term can be thought of as a "rolling average" of the past gradients, which smooths out the updates and helps to avoid oscillations in the optimization process. In addition, the momentum term can help the algorithm to escape from local minima by providing the necessary momentum to jump over small barriers in the objective function. Momentum can improve the convergence speed and stability of gradient descent, especially when the objective function is non-convex or has a high degree of curvature, as depicted in Fig. 4.

Note that when β=0, the momentum algorithm is same as the gradient descent algorithm.

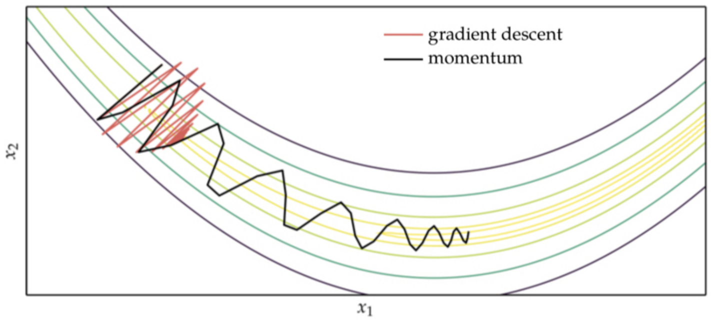


Figure 4: Convergence of gradient descent vs. momentum.

5.2 Nesterov Momentum

One issue of momentum is that the steps do not slow down enough at the bottom of a valley and it tends to overshoot the valley. Nesterov Momentum remedies the issue as follows.

The Nesterov momentum algorithm, also known as Nesterov accelerated gradient (NAG), is a variant of the momentum algorithm that was proposed by Yurii Nesterov in 1983. The Nesterov momentum algorithm is a widely used in machine learning, as it is known to converge faster than standard momentum.

The basic idea behind the Nesterov momentum algorithm is to take into account the momentum when computing the gradient at the next iteration. This is done by using an estimate of the future position of the parameters based on the current momentum direction. This helps to reduce the oscillations that can occur with standard momentum and improve the convergence speed.

Mathematically, the update rule for the Nesterov momentum algorithm is given by:

$$\mathbf{v}^{(k)} = \beta \mathbf{v}^{(k-1)} + (1-\beta)\nabla f(\mathbf{x}^{(k)} - \beta \mathbf{v}^{(k-1)}) \tag{18}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \mathbf{v}^{(k)} \tag{19}$$

where $\nabla f(\mathbf{x}^{(k)} - \beta \mathbf{v}^{(k-1)})$ is the gradient of the objective function computed at the future position of the parameters.

In essence, the Nesterov momentum algorithm evaluates the gradient at a point that is a step ahead in the direction of the momentum before updating the parameters. As illustrated in Fig. 5, this results in a more accurate estimate of the gradient and allows the algorithm to take larger steps towards the minimum of the objective function.

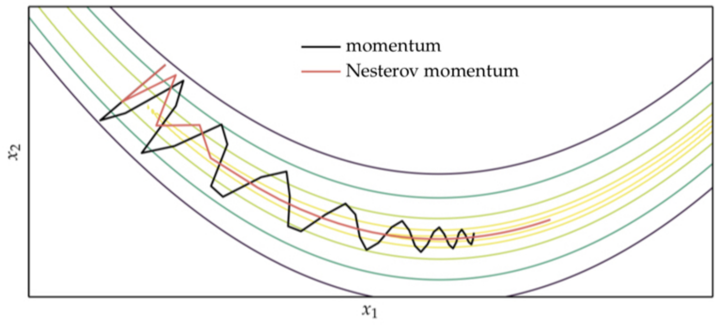


Figure 5: Convergence of momentum vs Nesterov momentum.

5.3 Adagrad

Momentum and Nesterov Momentum update all componens of **x** with the same learning rate. The adaptive subgradient method (Adagrad), adapts a learning rate for each one in **x**. In other words, Adagrad decreases the learning rate for parameters that have received large updates, and increase the learning rate for parameters that have received small updates. This helps to prevent the learning rate from being too large and causing the algorithm to overshoot the minimum of the objective function.

$$\mathbf{x}_i^{(k+1)} = \mathbf{x}_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} \mathbf{x}^{(k)}$$
$$s_i^{(k)} = \sum_{j=1}^{k} \left(g_i^{(j)} \right)^2$$

where ϵ is a small value on the order of 10^{-8}, to prevent the case of division by zero. Adagrad is far less sensitive to the learning rate α.

5.4 RMSprop and Adadelta

In Adagrad, the learning rate may monotonically decrease. To prevent this, *RMSprop* (root mean squared propagation) maintains a decaying average of squared gradients.

$$\mathbf{s}^{(k+1)} = \gamma \mathbf{s}^{(k)} + (1-\gamma)(\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}) \tag{20}$$

where γ is between 0 and 1, and usually is 0.9.

$$\begin{aligned} \mathbf{x}_i^{(k+1)} &= \mathbf{x}_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} \mathbf{g}_i^{(k)} \\ &= \mathbf{x}_i^{(k)} - \frac{\alpha}{\epsilon + \mathrm{RMS}(\mathbf{g}_i)} \mathbf{g}_i^{(k)} \end{aligned} \tag{21}$$

While in *Adadelta*, an exponentially decaying average is used,

$$\mathbf{x}_i^{(k+1)} = \mathbf{x}_i^{(k)} - \frac{\mathrm{RMS}(\nabla_i)}{\epsilon + \mathrm{RMS}(\mathbf{g}_i)} \mathbf{g}_i^{(k)} \tag{22}$$

5.5 Adam

The adaptive moment estimation (Adam) is another popular optimization algorithm that combines the benefits of momentum and adaptive learning rate methods. Adam is so far the most widely used optimization method in neural network training. It computes adaptive learning rates for each parameter based on both the first and second moments of the gradients, and uses a moving average of the parameter updates to update the model parameters.

It stores both an exponentially decaying squared gradient like RMSProp and Adadelta, but also an exponentially decaying gradient like momentum.

$$\begin{aligned}
\mathbf{v}^{(k+1)} &= \gamma_v \mathbf{v}^{(k)} + (1-\gamma_v)\mathbf{g}^{(k)} \\
\mathbf{s}^{(k+1)} &= \gamma_s \mathbf{s}^{(k)} + (1-\gamma_s)\left(\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\right) \\
\hat{\mathbf{v}}^{(k+1)} &= \mathbf{v}^{(k+1)}/(1-\gamma_v^{(k)}) \\
\hat{\mathbf{s}}^{(k+1)} &= \mathbf{s}^{(\mathbf{k+1})}/(1-\gamma_s^{(k)}) \\
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha \hat{\mathbf{v}}^{(k+1)}/\left(\epsilon + \sqrt{\hat{\mathbf{s}}^{(k+1)}}\right)
\end{aligned} \tag{23}$$

To sum up, varying learning rate optimization algorithms can converge faster than basic gradient descent, as they use information from previous iterations to update the parameters in a more informed way. In practical settings, the gradients computed from the objective function can be noisy or incomplete. Accelerated descent methods can mitigate the effect of noisy gradients by using adaptive learning rates, momentum terms, or other techniques to update the parameters in a more stable and robust way. Accelerated descent methods can be used with different objective functions and architectures, and they can be tuned for different settings and scenarios. For example, Adam and Adagrad can adaptively adjust the learning rate for each parameter based on its gradient history, which can improve performance on complex optimization landscapes. Last but not least, these accelerated descent methods can help to improve generalization by avoiding overfitting, which occurs when the model memorizes the training data and performs poorly on new data.

6 Direct Methods in Optimization

Unlike gradient-based methods, direct methods do not require derivatives of the objective function, making them suitable for problems where derivatives are unavailable or too costly to compute. Let's briefly discuss some of the most common direct methods, such as cyclic coordinate search, Powell's method, Nelder-Mead simplex method, simulated annealing, cross-entropy method, and covariance matrix adaptation.

6.1 Cyclic Coordinate Search (CCS)

CCS algorithm iteratively searches along one coordinate direction while holding the other coordinates fixed. The algorithm starts with an initial guess of the minimum and iteratively updates the coordinate direction to search along until a stopping criterion is met.

The search starts from an initial $\mathbf{x}^{(1)}$ and optimize the first input.

$$\mathbf{x}^{(2)} = \arg\min_{\mathbf{x}_1} f(\mathbf{x}_1, \mathbf{x}_2^{(1)}, \mathbf{x}_3^{(1)}, \cdots, \mathbf{x}_n^{(1)}) \tag{24}$$

Then, it moves to the next coordinate,

$$\mathbf{x}^{(3)} = \arg\min_{\mathbf{x}_2} f(\mathbf{x}_1^{(2)}, \mathbf{x}_2, \mathbf{x}_3^{(2)}, \cdots, \mathbf{x}_n^{(2)}) \tag{25}$$

This process is equivalent to doing a sequence of line searches along the set of n basis vectors. It is terminated after no significant improvement is made.

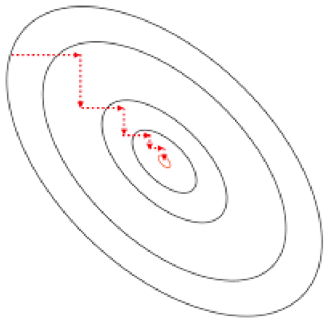


Figure 6: Convergence of the CCS method.

Similar to the momentum method in the gradient descent, the cyclic method can be augmented with an acceleration step to help traverse diagonal valleys. For each full cycle starting with $\mathbf{x}^{(1)}$ from (1) to (k), an additional line search is conducted along with the direction of $\mathbf{x}^{(k+1)} - \mathbf{x}^{(1)}$.

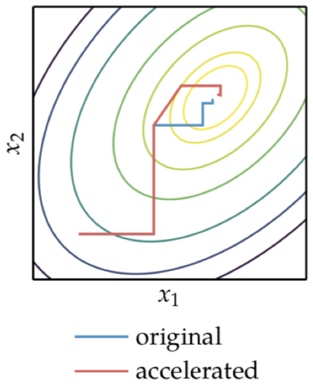


Figure 7: Convergence of the original vs. accelerated CCS methods.

The algorithm is simple to implement and can handle non-linear, non-convex functions. However, the algorithm can suffer from slow convergence and may converge to a local minimum.

6.2 Powell's Method

Powell's method is a direct search method that uses a set of basis vectors to iteratively search for the minimum. The algorithm starts with an initial set of basis vectors (a list of search directions $\mathbf{u}_1, \mathbf{u}_2, \cdots \mathbf{u}_n$) and iteratively updates them based on the function evaluations. The algorithm terminates when a stopping criterion is met.

Starting at $\mathbf{x}^{(1)}$, Powell's method conduct a line search for each direction, updating the design point each time. Then shift each $\mathbf{u}$ by one index and drop the first list, $\mathbf{u}_1$. The last direction is replaced with the direction of $\mathbf{x}_n - \mathbf{x}_1$.

$$\begin{aligned}\mathbf{x}^{(k+1)} &\leftarrow \text{line search}(f, \mathbf{x}^{(k)}, \mathbf{u}^{(k)}) \text{ for all } i \in 1, \cdots, n \\ \mathbf{u}_{i+1}^{(k+1)} &\leftarrow \mathbf{u}_i^{(k)} \text{ for all } i \in 1, \cdots, n-1 \\ \mathbf{u}_n^{(k+1)} &\leftarrow \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\end{aligned} \tag{26}$$

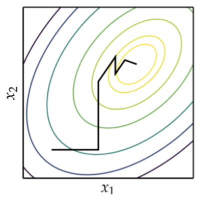


Figure 8: Convergence of the Powell's method.

Powell showed that for quadratic functions, after k full iterations the last k direction will be mutually conjugate. It is recommended to reset every n or $n+1$ iterations. Note that the Powell's method can suffer from slow convergence and may converge to a local minimum.

6.3 Nelder-Mead Simplex Method

The Nelder-Mead simplex method is a direct search method that constructs a simplex, which is a geometrical shape in high-dimensional space. The simplex is updated by reflecting, expanding, contracting, or shrinking its vertices. The algorithm terminates when a stopping criterion is met.

The Nelder-Mead simplex method uses a simplex to traverse the space in search of a minimum. A simplex is a $n+1$-vertices polyhedron in n-dimensional space.

- $\mathbf{x}_h$, pt of highest f,
- $\mathbf{x}_s$, pt of 2nd highest f,
- $\mathbf{x}_l$, pt of lowest f,
- $\bar{x}$, mean pt excluding $\mathbf{x}_h$.
- Reflection. $\mathbf{x}_r = \bar{x} + (\bar{x} - \mathbf{x}_h)$,
- Expansion. $\mathbf{x}_e = \bar{x} + 2(\mathbf{x}_r - \bar{x})$,
- Contraction. $\mathbf{x}_c = \bar{x} + 0.5(\mathbf{x}_h - \bar{x})$,
- Shrinkage, halving the distance to $\mathbf{x}_l$.

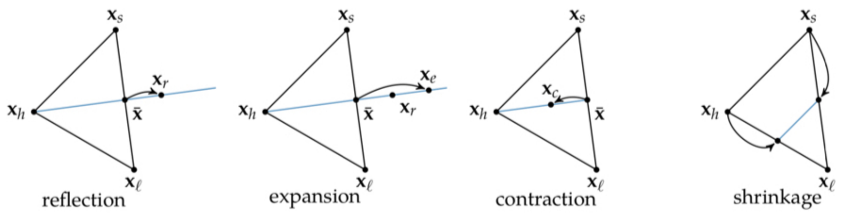
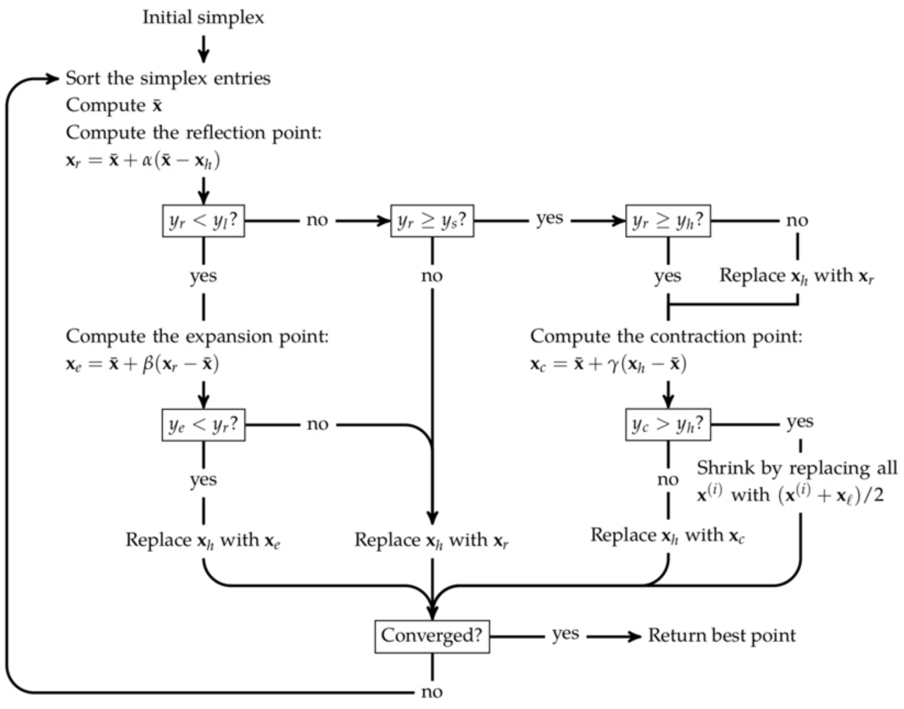


Figure 9: Four mechanisms of the Nelder-Mead simplex algorithm.



Initial simplex
Sort the simplex entries
Compute $\bar{\mathbf{x}}$
Compute the reflection point:
$\mathbf{x}_r = \bar{\mathbf{x}} + \alpha(\bar{\mathbf{x}} - \mathbf{x}_h)$
$y_r < y_l$?
no
$y_r \geq y_s$?
yes
$y_r \geq y_h$?
no
yes
no
yes
Replace $\mathbf{x}_h$ with $\mathbf{x}_r$
Compute the expansion point:
$\mathbf{x}_e = \bar{\mathbf{x}} + \beta(\mathbf{x}_r - \bar{\mathbf{x}})$
Compute the contraction point:
$\mathbf{x}_c = \bar{\mathbf{x}} + \gamma(\mathbf{x}_h - \bar{\mathbf{x}})$
$y_e < y_r$?
no
$y_c > y_h$?
yes
yes
no
Shrink by replacing all $\mathbf{x}^{(i)}$ with $(\mathbf{x}^{(i)} + \mathbf{x}_\ell)/2$
Replace $\mathbf{x}_h$ with $\mathbf{x}_e$
Replace $\mathbf{x}_h$ with $\mathbf{x}_r$
Replace $\mathbf{x}_h$ with $\mathbf{x}_c$
Converged?
yes
Return best point
no

The Nelder-Mead simplex method is a simple and robust optimization method that can handle non-linear, non-convex functions. However, the algorithm can suffer from slow convergence and may converge to a local minimum.

6.4 Simulated Annealing*

Simulated annealing is a stochastic optimization method[1] that simulates the annealing process in metallurgy. The algorithm starts with an initial guess of the minimum and iteratively updates the solution by randomly perturbing it and accepting the perturbation based on a probability function. The algorithm decreases the temperature parameter over time, which controls the acceptance probability, until a stopping criterion is met.

Temperature is used to control the degree of stochasticity during the randomized search.

- t starts high, allowing the process to freely move with the hope of finding a good region with the best local minimum.
- t is then slowly brought down, reducing the stochasticity and forcing the search to converge to a minimum.

Simulated annealing is often used on functions with many local minima due to its ability to escape local minima. At every iteration, a candidate transition from $\mathbf{x}$ to $\mathbf{x}'$ is sampled from a transition distribution T and is accepted with *probability*

$$\begin{cases} 1 & \text{if } \Delta y \leq 0 \\ \min(\exp(-\Delta y/t), 1) & \text{if } \Delta y > 0 \end{cases} \tag{27}$$

where $\Delta y = f(\mathbf{x}) - f(\mathbf{x}')$

[1]Stochastic optimization methods are marked with a *.

Simulated annealing is a robust optimization method that can handle non-linear, non-convex functions and can escape local minima. However, the algorithm can suffer from slow convergence and may require a large number of function evaluations.

6.5 Cross-Entropy Method (CEM)*

The cross-entropy method is another stochastic optimization method that uses a probabilistic model to iteratively generate candidate solutions. The algorithm updates the model parameters based on the function evaluations and generates new candidate solutions using the updated model. The algorithm terminates when a stopping criterion is met.

It requires choosing a family of distributions parameterized by θ, such as multivariate normal distributions with *a mean vector and a covariance matrix*. The algorithm also requires us to specify the number of elite samples, m_{elite} , to use when fitting the parameters for the next iteration.

$$\begin{aligned} \boldsymbol{\mu}^{(k+1)} &= \frac{1}{m_{\text{elite}}} \sum_{i=1}^{m_{\text{elite}}} \mathbf{x}^{(k)} \\ \Sigma^{(k+1)} &= \frac{1}{m_{\text{elite}}} \sum_{i=1}^{m_{\text{elite}}} (\mathbf{x}^{(k)} - \boldsymbol{\mu}^{(k+1)})(\mathbf{x}^{(k)} - \boldsymbol{\mu}^{(k+1)})^T \end{aligned} \tag{28}$$

This probability distribution, often called a *proposal distribution*, is used to propose new samples for the next iteration. At each iteration, we sample from the proposal distribution and then update the proposal distribution to fit a collection of the best samples.

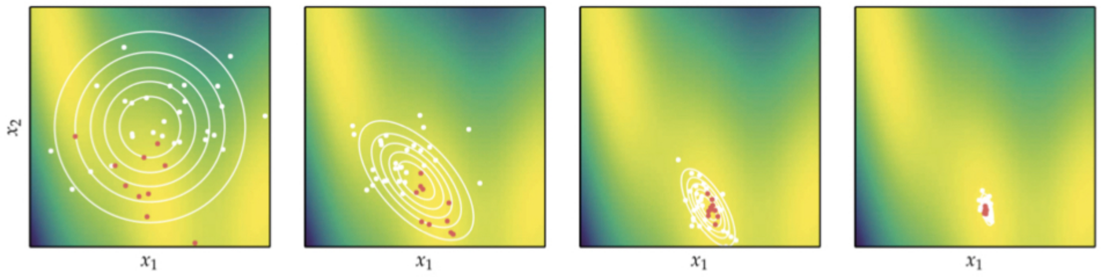


Figure 10: Convergence of a cross entropy optimization.

The cross-entropy method is a robust optimization method that can handle non-linear, non-convex functions and can escape local minima. However, the algorithm can suffer from slow convergence and may require a large number of function evaluations.

6.6 Covariance Matrix Adaptation*

Covariance matrix adaptation (CMA) is another stochastic optimization method that uses a Gaussian distribution to generate candidate solutions. The algorithm updates the mean and covariance matrix of the Gaussian distribution based on the function evaluations and generates new candidate solutions using the updated distribution. The algorithm terminates when a stopping criterion is met.

Covariance matrix adaptation maintains a mean vector $\boldsymbol{\mu}$, a covariance matrix $\boldsymbol{\Sigma}$, and an additional step-size scalar δ. The covariance matrix only increases or decreases in a single direction with every iteration, whereas the step-size scalar is adapted to control the overall spread of the distribution. At every iteration, m designs are sampled from the multivariate Gaussian

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \Sigma) \tag{29}$$

The designs are then sorted according to their objective function values such that $f(x^1) \leq f(x^2) \leq \cdots \leq f(x^m)$. A new mean vector $\boldsymbol{\mu}^{(k+1)}$ is formed using a weighted average of the sampled designs:

$$\boldsymbol{\mu}^{(k+1)} \leftarrow \sum_{i=1}^{m} w_i \mathbf{x}^{(k)}$$
$$\sum_{i}^{m} w_i = 1 \quad w_1 > w_2 > \cdots > w_m > 0$$

The recommended weighting is obtained by

$$w'_i = \ln \frac{m+1}{2} - \ln i \text{ for } i \in \{1, \cdots, m\} \tag{30}$$

to obtain $\mathbf{w} = \mathbf{w}' / \sum_i w'_i$.

The step size is updated using a cumulative $\mathbf{p}_\sigma$ that tracks steps over time

$$\begin{aligned}
\mathbf{p}_\sigma^1 &= 0 \\
\mathbf{p}_\sigma^{(k+1)} &\leftarrow (1 - c_\sigma)\mathbf{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_{\text{eff}}}(\Sigma^{(k)})^{-1/2}\sigma_w \\
\mu_{\text{eff}} &= \frac{1}{\sum_i w_i^2} \\
\sigma_w &= \sum_{i=1}^{m_{\text{elite}}} w_i \sigma^i \text{ for } \sigma^i = \frac{\mathbf{x}^{(k)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}
\end{aligned} \tag{31}$$

The new step size is

$$\sigma^{(k+1)} \leftarrow \sigma^{(k)} \exp\left(\frac{c_\sigma}{\mathbf{d}_\sigma}\left[\frac{||\mathbf{p}_\sigma||}{\mathbb{E}||\mathcal{N}(0, \mathbf{I})||} - 1\right]\right) \tag{32}$$

where $\mathbb{E}$ is the expected length of a vector drawn from Gaussian distribution.

$$\mathbb{E}||\mathcal{N}(0, \mathbf{I})|| = \sqrt{2}\frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} \approx \sqrt{n}\left(1 - \frac{1}{4n} + \frac{1}{21n^2}\right) \tag{33}$$

$$\begin{aligned}
c_\sigma &= (\mu_{\text{eff}} + 2)/(n + \mu_{\text{eff}} + 5) \\
\mathbf{d}_\sigma &= 1 + 2\max(0, \sqrt{\mu_{\text{eff}} - 1)/(k+1)} - 1) + c_\sigma
\end{aligned} \tag{34}$$

The covariance matrix is updated as follows

$$\begin{aligned}
\mathbf{p}_\Sigma^1 &= 0 \\
\mathbf{p}_\Sigma^{(k+1)} &\leftarrow (1 - c\Sigma)\mathbf{p}_{\boldsymbol{\Sigma}}^{(\mathbf{k})} + h_\sigma \sqrt{c\Sigma(2 - c_\Sigma)\mu_{\text{eff}}}\,\text{œ}_w
\end{aligned} \tag{35}$$

where

$$h_\sigma = \begin{cases} 1 & if \frac{||\mathbf{p}_\Sigma||}{(1 - c_\sigma^{2k+1})} < (1.4 + \frac{2}{n+1})\mathbb{E}||\mathcal{N}(0, \mathbf{I})|| \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

The update requires the adjusted weights $\mathbf{w}$:

$$w_i^0 = \begin{cases} w_i & \text{if } w_i \geq 0 \\ \frac{n w_i}{||\Sigma^{-1/2}\delta^i||^2} & \text{otherwise} \end{cases} \tag{37}$$

The covariance update is then

$$\Sigma^{(k+1)} \leftarrow [1 + c_1 c_\sigma (1 - h_\sigma)(2 - c_\sigma) - c_1 - c_\mu]\Sigma^{(k)} + c_1 \mathbf{p}_\Sigma \mathbf{p}_\Sigma^T + c_\mu \sum_{i=1}^{\mu} w_i^0 \delta^i (\delta^i)^T \tag{38}$$

The constants have the following recommended values

$$\begin{aligned} c_\Sigma &= \frac{4 + \mu_{\text{eff}}/n}{n + 4 + 2\mu_{\text{eff}}/n} \\ c_1 &= \frac{2}{(n+1.3)^2 + \mu_{\text{eff}}} \\ c_\mu &= \min\left(1 - c_1, 2\frac{\mu_{\text{eff}} - 2 + 1/\mu_{\text{eff}}}{(n+2)^2 + \mu_{\text{eff}}}\right) \end{aligned} \tag{39}$$

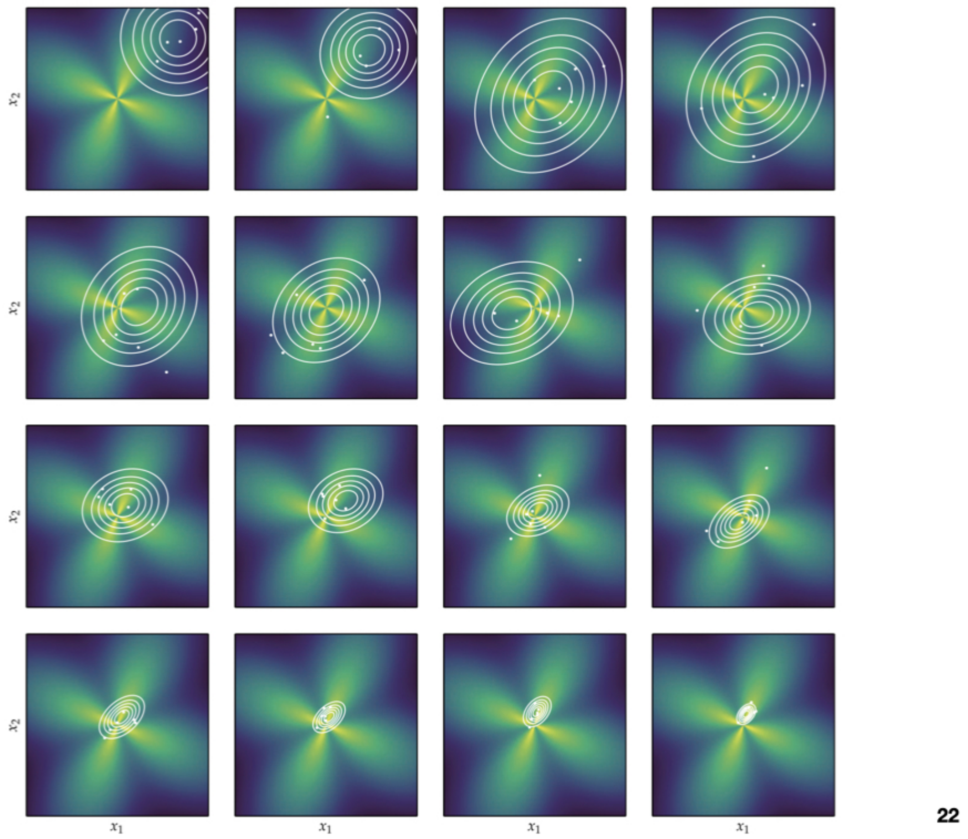


Figure 11: Convergence of an CMA optimization.

CMA is a robust optimization method that can handle non-linear, non-convex functions and can escape local minima. CMA is particularly well-suited for high-dimensional optimization problems. However, the algorithm can suffer from slow convergence and may require a large number of function evaluations.

Summary:

- Stochastic methods employ random numbers during the optimization process
- Simulated annealing guesses a temperature that controls random exploration and which is reduced over time to converge on a local minimum.
- The cross-entropy method and evolution strategies maintain proposal distributions from which they sample in order to inform updates.
- Covariance matrix adaptation is a robust and sample-efficient optimizer that maintains a multivariate Gaussian proposal distribution with a full covariance matrix.

7 Population Based Numerical Optimization Methods

Population-based numerical optimization algorithms are a class of optimization algorithms that mimic the behavior of populations of organisms to find the optimal solution. Most population methods are stochastic in nature, and it is generally easy to parallelize the computation.

These methods typically have the following steps

- Initialization
- Encoding
- Mutation
- Crossover
- Selection

Population methods begin with an initial population, just as descent methods require an initial design point. The initial population should be spread over the design space to increase the chances that the samples are close to the best regions. The following strategies can be applied

- Uniform distribution in a bounded region
- Multivariate normal distribution centered over a region of interest.
- The Cauchy distribution has an unbounded variance and can cover a much broader space.

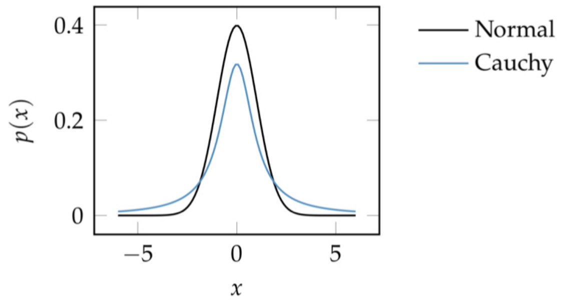


Figure 12: Normal vs. Cauchy distribution.

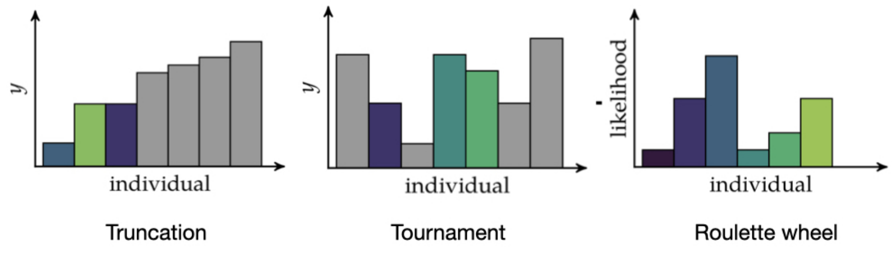
Two of the most well-known population-based numerical optimization algorithms are the genetic algorithm (GA) and particle swarm optimization (PSO).

7.1 Genetic Algorithm

The genetic algorithm is a population-based optimization algorithm inspired by the process of natural selection. The algorithm starts with an initial population of candidate solutions, which are represented as chromosomes. Each chromosome is evaluated based on the objective function, and the algorithm selects the best solutions to reproduce. The reproduction process involves recombining the genetic material of the selected chromosomes and mutating them to create new offspring. The algorithm then evaluates the offspring and selects the best solutions to form the next generation of the population.

Selection is the process of choosing chromosomes to use as parents for the next generation. For a population with m chromosomes, a selection method will produce a list of m parental pairs for the m children of the next generation. The selected pairs may contain duplicates.

- Truncation, random one from the best k truncation
- Tournament, the fittest out of k randomly chosen
- Roulette wheel, chosen with a probability proportional to the fitness



The genetic algorithm is a powerful optimization algorithm that can handle non-linear, non-convex functions and can escape local minima. The algorithm is robust to noise and can handle constraints on the variables. However, the genetic algorithm can suffer from premature convergence if the population size is too small or the selection criteria are too strict. Additionally, the algorithm can require a large number of function evaluations to converge to the optimal solution.

7.2 Particle Swarm Optimization

Particle swarm optimization is a population-based optimization algorithm inspired by the behavior of flocks of birds or schools of fish. The algorithm starts with an initial population of particles, which represent candidate solutions. Each particle has a position and a velocity, and the algorithm updates the particles' positions and velocities based on their own and their neighbors' best positions. The algorithm evaluates each particle based on the objective function, and the particles adjust their positions and velocities to move towards the optimal solution.

PSO introduces momentum to accelerate convergence toward minima. Each individual (particle), in the population keeps track of its current position, velocity, and the best position it has seen so far. Momentum allows an individual to accumulate speed in a favorable direction, independent of local perturbations.

$$\begin{aligned} \mathbf{x}^{(k)} &\leftarrow \mathbf{x}^{(k)} + \mathbf{v}^i \\ \mathbf{v}^i &\leftarrow w\mathbf{v}^i + c_1 r_1 (\mathbf{x}^i_{lbest} - x^i) + c_2 r_2 (\mathbf{x}_{gbest} - x^i) \end{aligned} \tag{40}$$

where

- $\mathbf{x}_{lbest}$: the current local best locations for the given population
- $\mathbf{x}_{gbest}$: the global best locations
- w, c_1, c_2: empirical parameters
- r_1, r_2: random numbers drawn from $U(0,1)$

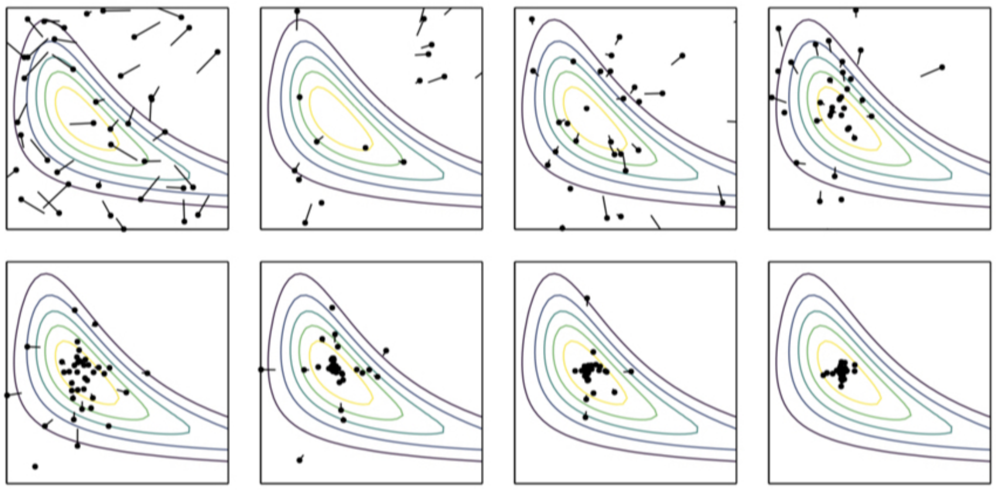


Figure 13: Convergence of a PSO implementation.

The particle swarm optimization algorithm is a powerful optimization algorithm that can handle non-linear, non-convex functions and can escape local minima. The algorithm is robust to noise and can handle constraints on the variables. Additionally, the algorithm can converge quickly and requires fewer function evaluations than the genetic algorithm. However, the particle swarm optimization algorithm can suffer from premature convergence if the swarm size is too small or the parameters are not tuned properly.

Population-based numerical optimization algorithms, such as the genetic algorithm and particle swarm optimization, are powerful tools for finding the optimal solution of complex optimization problems. These algorithms can handle non-linear, non-convex functions, escape local minima, and are robust to noise and constraints. However, these algorithms can suffer from premature convergence if not tuned properly, and may require a large number of function evaluations to converge to the optimal solution.

8 What to do when the function evaluation is time-consuming and computationally expensive?

Optimizing expensive functions is a challenging task, especially when the function evaluation is time-consuming and computationally expensive. In such cases, traditional optimization algorithms may not be efficient since they require numerous function evaluations to converge. Therefore, special optimization methods are required that can optimize the expensive function with the minimum number of function evaluations. Before we discuss the basics of two popular optimization methods that work well for expensive function optimization, let's emphasize the difference between the stochastic and probabilistic optimization models.

Stochastic optimization models such as CMA, Stimulated Annealing, and CEM assume that there is some underlying random process that generates the uncertain variables in the optimization problem. The goal is to find an optimal decision that is robust to the variability in the uncertain variables. Stochastic optimization models typically involve optimization problems with random or uncertain parameters, and they use statistical methods to represent and analyze the uncertainty. Stochastic optimization models provide a range of possible outcomes and their probabilities, rather than a single deterministic solution.

On the other hand, probabilistic optimization models explicitly represent uncertainty using probability distributions. The goal is to find the optimal decision that maximizes the expected objective function value, taking into account the probability distribution of the uncertain variables. Probabilistic optimization models are often used in situations where the probability distributions of the uncertain variables are known, or can be estimated from data. Bayesian optimization and decision analysis are common techniques used in probabilistic optimization.

8.1 Bayesian Optimization (BO)

The BO algorithm works by iteratively selecting the next point to evaluate based on the information gathered from previous function evaluations. The approach is based on the Bayesian framework, where the prior distribution is updated with the data from the function evaluations to obtain the posterior distribution.

The algorithm models the unknown function using a Gaussian Process (GP), which is a popular non-parametric probabilistic model used for regression tasks. The GP models the function as a distribution over functions and provides a measure of uncertainty at each point in the solution space. The GP is defined by a mean function and a covariance function. The mean function provides the expected value of the function at a given point, and the covariance function captures the correlation between the function values at different points.

To determine the next point to evaluate, the algorithm uses an acquisition function, which is a heuristic that balances the trade-off between exploration and exploitation. The acquisition function is defined using the posterior distribution and provides a measure of the expected improvement over the current best solution. There are several popular acquisition functions, such as Expected Improvement (EI), Probability of Improvement (PI), and Upper Confidence Bound (UCB).

The algorithm iteratively evaluates the function at the point with the highest acquisition function value and updates the posterior distribution using the data from the function evaluation. The algorithm continues until a stopping criterion is met, such as a maximum number of function evaluations or a convergence criterion.

One of the major advantages of Bayesian optimization is its ability to efficiently optimize expensive functions with a small number of function evaluations. The algorithm can adaptively search the solution space and efficiently explore the regions of high uncertainty. Moreover, the algorithm can handle noisy and non-smooth functions since the GP can capture the uncertainty and correlation between function evaluations.

BO has been successfully applied to various real-world problems, such as chemical synthesis, circuit design, robotics, and hyperparameter tuning in machine learning. The algorithm has also been extended to handle complex optimization problems, such as multi-objective optimization and constrained optimization.

8.2 Surrogate-based Optimization (SBO)

The surrogate model is trained using a set of low-fidelity or cheap surrogate models to provide a fast approximation of the expensive objective function. The surrogate model is then used to guide the optimization search in the high-fidelity or expensive domain.

The surrogate model can be any machine learning model, such as a Gaussian process (GP), neural network, or decision tree. The choice of the surrogate model depends on the problem characteristics and the available data. GP models are popular in SBO due to their ability to capture the uncertainty and correlation between function evaluations.

The SBO algorithm proceeds in two phases: the surrogate model training phase and the optimization phase. In the surrogate model training phase, a set of low-fidelity or cheap surrogate models are trained using the available data. The surrogate models are then used to guide the optimization search in the high-fidelity or expensive domain. The optimization phase proceeds by iteratively selecting the next point to evaluate based on the surrogate model predictions and the acquisition function. The acquisition function balances the trade-off between exploration and exploitation and guides the search towards the regions of high uncertainty or promising solutions.

The optimization phase continues until a stopping criterion is met, such as a maximum number of function evaluations or a convergence criterion. At each iteration, the surrogate model is updated with the data from the new function evaluations to improve the accuracy of the surrogate model.

In addition to its capability of handling expensive objective functions, SBO also can handle noisy and non-smooth objective functions since the surrogate model can capture the uncertainty and correlation between function evaluations. Last but not least, it can handle high-dimensional optimization problems by reducing the dimensionality of the search space using the surrogate model.