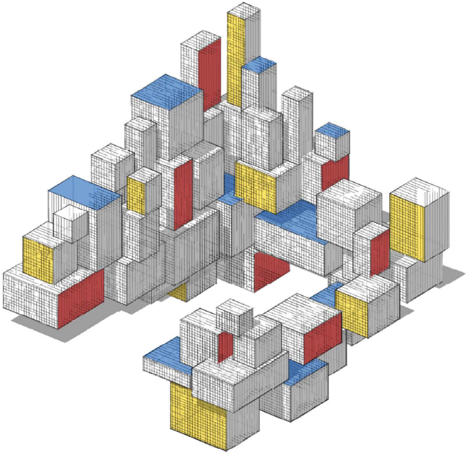
Cross-Entropy Method



Purpose

In this lecture we discuss the Cross-Entropy method, which can be used for:

- adaptive importance sampling estimation
- solving deterministic optimization problems
- solving noisy optimization problems

Cross-Entropy Method

The cross-entropy (CE) method is a simple Monte Carlo algorithm that can be used for both optimization and estimation.

The basic idea for minimizing a function S on a set $\mathcal{X}$:

- Define a parametric family of pdfs $\{f(\cdot \,|\, \boldsymbol{v}), \boldsymbol{v} \in \mathcal{V}\}$ on $\mathcal{X}$.
- Iteratively update the parameter $\boldsymbol{v}$ so that $f(\cdot \,|\, \boldsymbol{v})$ places more mass on states $\boldsymbol{x}$ that have smaller S values than previously.

In particular, the CE algorithm has two basic phases:

1. *Sampling*: Samples $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ are drawn independently according to $f(\cdot \,|\, \boldsymbol{v})$. The objective function S is evaluated at these points.
2. *Updating*: A new parameter $\boldsymbol{v}'$ is selected on the basis of those $\boldsymbol{X}_i$ for which $S(\boldsymbol{X}_i) \leqslant \gamma$ for some level γ. These $\{\boldsymbol{X}_i\}$ form the elite sample set, $\mathcal{E}$.

Cross-Entropy Method

At each iteration the level parameter γ is chosen as the worst of the $N^{\text{elite}} := \lceil \varrho N \rceil$ elite samples, where $\varrho \in (0,1)$ is the rarity parameter — typically, $\varrho = 0.1$ or $\varrho = 0.01$.

The parameter $\boldsymbol{v}$ is updated as a smoothed average $\alpha \boldsymbol{v}' + (1-\alpha)\boldsymbol{v}$, where $\alpha \in (0,1)$ is the smoothing parameter and

$$\boldsymbol{v}' := \underset{\boldsymbol{v} \in \mathcal{V}}{\operatorname{argmax}} \sum_{\boldsymbol{X} \in \mathcal{E}} \ln f(\boldsymbol{X} \,|\, \boldsymbol{v}).$$

The updating rule above is the result of minimizing the Cross-Entropy distance (Kullback–Leibler divergence) between the conditional density of $\boldsymbol{X} \sim f(\boldsymbol{x} \,|\, \boldsymbol{v})$ given $S(\boldsymbol{X}) \leqslant \gamma$, and $f(\boldsymbol{x}; \boldsymbol{v})$.

Algorithm 1: Cross-Entropy Method for Minimization

```
input: Function $S$, initial sampling parameter $\boldsymbol{v}_0$, sample size $N$, rarity parameter
       $\varrho$, smoothing parameter $\alpha$.
output: Approximate minimum of $S$ and optimal sampling parameter $\boldsymbol{v}$.
1  Initialize $\boldsymbol{v}_0$, set $N^{\text{elite}} \leftarrow \lceil \varrho N \rceil$ and $t \leftarrow 0$.
2  while a stopping criterion is not met do
3      $t \leftarrow t + 1$
4      Simulate an iid sample $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ from the pdf $f(\cdot \mid \boldsymbol{v}_{t-1})$.
5      Evaluate the performances $S(\boldsymbol{X}_1), \ldots, S(\boldsymbol{X}_N)$ and sort them from smallest
         to largest: $S_{(1)}, \ldots, S_{(N)}$.
6      Let $\gamma_t$ be the sample $\varrho$-quantile of the performances:
           $\gamma_t \leftarrow S_{(N^{\text{elite}})}$.                                    (1)
7      Determine the set of elite samples $\mathcal{E}_t = \{\boldsymbol{X}_i : S(\boldsymbol{X}_i) \leqslant \gamma_t\}$.
8      Let $\boldsymbol{v}'_t$ be the MLE of the elite samples:
           $\boldsymbol{v}'_t \leftarrow \underset{\boldsymbol{v}}{\operatorname{argmax}} \sum_{\boldsymbol{X} \in \mathcal{E}_t} \ln f(\boldsymbol{X} \mid \boldsymbol{v})$.        (2)
9      Update the sampling parameter as
           $\boldsymbol{v}_t \leftarrow \alpha \boldsymbol{v}'_t + (1 - \alpha)\boldsymbol{v}_{t-1}$.                  (3)
10 return $\gamma_t$, $\boldsymbol{v}_t$
```

Cross-Entropy Method

Note that (2) yields the maximum likelihood estimator (MLE) of $\boldsymbol{v}$ based on the elite samples. Explicit solutions can be found for specific families of distributions.

When $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}^2))$, the mean vector $\boldsymbol{\mu}$ and the vector of variances $\boldsymbol{\sigma}^2$ are simply updated via the sample mean and sample variance of the elite samples. This is known as normal updating.

The CE algorithm produces a sequence of pairs $(\gamma_1, \boldsymbol{v}_1), (\gamma_2, \boldsymbol{v}_2), \ldots,$ such that

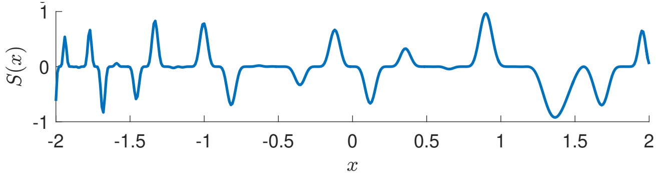
- γ_t converges (approximately) to the minimal function value, and
- converges $f(\cdot \,|\, \boldsymbol{v}_t)$ to a degenerate pdf that (approximately) concentrates all its mass at a minimizer of S.

A possible stopping condition is to stop when the sampling distribution $f(\cdot \,|\, \boldsymbol{v}_t)$ is sufficiently close to a degenerate distribution. For normal updating this means that the standard deviation is sufficiently small.

Example: Minimizing the Wiggly Function

We use a CE algorithm to minimize the following "wiggly" function:

$$S(x) = \begin{cases} -\mathrm{e}^{-x^2/100} \sin(13x - x^4)^5 \sin(1 - 3x^2)^2, & \text{if } -2 \leqslant x \leqslant 2, \\ \infty, & \text{otherwise.} \end{cases}$$



We take the family of normal distributions $\{\mathcal{N}(\mu, \sigma^2)\}$ for the sampling step (Step 4 of Algorithm 1), starting with $\mu = 0$ and $\sigma = 3$.

Minimizing the Wiggly Function

The choice of the initial parameter is quite arbitrary, as long as σ is large enough to sample a wide range of points: We take $N = 100$ samples at each iteration, set $\varrho = 0.1$, and keep the $N^{\text{elite}} = 10 = \lceil N\varrho \rceil$ smallest ones as the elite samples.

The parameters μ and σ are then updated via the sample mean and sample standard deviation of the elite samples.

In this case we do not use any smoothing ($\alpha = 1$).

In the following Python code the 100×2 matrix `Sx` stores the x-values in the first column and the function values in the second column. The rows of this matrix are sorted in ascending order according to the function values, giving the matrix `sortSx`.

The first $N^{\text{elite}} = 10$ rows of this sorted matrix correspond to the elite samples and their function values.

CEmethod.py

```python
from simann import wiggly
import numpy as np
np.set_printoptions(precision=3)
mu, sigma = 0, 3
N, Nel = 100, 10
eps = 10**-5
S = wiggly
while sigma > eps:
    X = np.random.randn(N,1)*sigma + np.array(np.ones((N,1)))*mu
    Sx = np.hstack((X, S(X)))
    sortSx = Sx[Sx[:,1].argsort(),]
    Elite = sortSx[0:Nel,:-1]
    mu = np.mean(Elite, axis=0)
    sigma = np.std(Elite, axis=0)
    print('S(mu)= {}, mu: {}, sigma: {}\n'.format(S(mu), mu, sigma))
```

```
S(mu)= [0.071], mu: [0.414], sigma: [0.922]
S(mu)= [0.063], mu: [0.81], sigma: [0.831]
S(mu)= [-0.033], mu: [1.212], sigma: [0.69]
S(mu)= [-0.588], mu: [1.447], sigma: [0.117]
S(mu)= [-0.958], mu: [1.366], sigma: [0.007]
S(mu)= [-0.958], mu: [1.366], sigma: [0.]
S(mu)= [-0.958], mu: [1.366], sigma: [3.535e-05]
S(mu)= [-0.958], mu: [1.366], sigma: [2.023e-06]
```

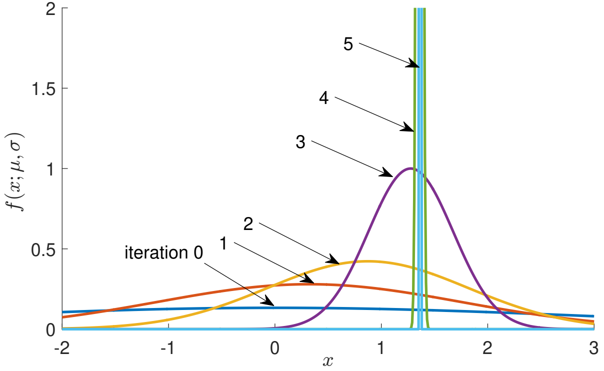


Figure: The normal pdfs of the first five sampling distributions, truncated to the interval $[-2, 3]$. The initial sampling distribution is $\mathcal{N}(0, 3^2)$.

CE Method fo Noisy Optimization

The CE method can also be used to optimize a noisy function $S(\boldsymbol{x}) = \mathbb{E}\widetilde{S}(\boldsymbol{x}, \boldsymbol{\xi})$.

The only change required in Algorithm 1 is that every function value $S(\boldsymbol{x})$ be replaced by its estimate $\widehat{S}(\boldsymbol{x})$.

Depending on the level of noise in the function, the sample size N might have to be increased considerably.

Example: CE Method for Noisy Optimization

Suppose there is a “black box” that contains an unknown binary sequence of n bits.

If one feeds the black box any input vector, it will:

- first scramble the input by independently flipping the bits (changing 0 to 1 and 1 to 0) with a probability θ and then
- return the number of bits that do not match the true (unknown) binary sequence.

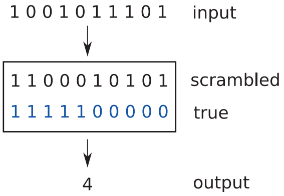


Figure: A noisy optimization function as a black box.

Example: CE Method for Noisy Optimization

Let $S(\boldsymbol{x})$ the true number of matching digits of a binary input vector $\boldsymbol{x}$.

The black box thus returns a noisy estimate $\widehat{S}(\boldsymbol{x})$.

The objective is to estimate the binary sequence inside the black box, by feeding it with many input vectors and observing their output; i.e., to minimize $S(\boldsymbol{x})$ using $\widehat{S}(\boldsymbol{x})$ as a proxy.

Since there are 2^n possible input vectors, it is infeasible to try all possible vectors $\boldsymbol{x}$ even for moderate n.

The following Python program implements the noisy function $\widehat{S}(\boldsymbol{x})$ for $n = 100$. Each input bit is flipped with a rather high probability $\theta = 0.4$.

The "true" vector has 1s at positions $1, \ldots, 50$ and 0s at $51, \ldots, 100$.

Snoisy.py

```python
import numpy as np

def Snoisy(X):   #takes a matrix
    n = X.shape[1]
    N = X.shape[0]
    # true binary vector
    xorg = np.hstack((np.ones((1,n//2)), np.zeros((1,n//2))))
    theta = 0.4 # probability to flip the input
    # storing the number of bits unequal to the true vector
    s = np.zeros(N)
    for i in range(0,N):
       # determine which bits to flip
       flip = (np.random.uniform(size=(n)) < theta).astype(int)
       ind = flip>0
       X[i][ind] = 1-X[i][ind]
       s[i] = (X[i] != xorg).sum()
    return s
```

Example: CE Method for Noisy Optimization

The CE code to optimize $S(\boldsymbol{x})$ is quite similar to the non-noisy optimization CE code.

Instead of sampling iid random variables $X_1, \ldots, X_N$ from a normal distribution, we now sample iid binary vectors $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ from a $\mathsf{Ber}(\boldsymbol{p})$ distribution.

More precisely, given a row vector of probabilities $\boldsymbol{p} = [p_1, \ldots, p_n]$, we independently simulate the components $X_1, \ldots, X_n$ of each binary vector $\boldsymbol{X}$ according to $X_i \sim \mathsf{Ber}(p_i)$, $i = 1, \ldots, n$.

After each iteration, the vector $\boldsymbol{p}$ is updated as the (vector) mean of the elite samples.

Example: CE Method for Noisy Optimization

The sample size is $N = 1000$ and the number of elite samples is 100.

The components of the initial sampling vector $\boldsymbol{p}$ are all equal to $1/2$; that is, the $\boldsymbol{X}$ are initially uniformly sampled from the set of all binary vectors of length $n = 100$.

At each subsequent iteration the parameter vector is updated via the mean of the elite samples and evolves towards a degenerate vector $\boldsymbol{p}^*$ with only 1s and 0s.

Sampling from such a $\mathsf{Ber}(\boldsymbol{p}^*)$ distribution gives an outcome $\boldsymbol{x}^* = \boldsymbol{p}^*$, which can be taken as an estimate for the minimizer of S; that is, the true binary vector hidden in the black box.

The algorithm stops when $\boldsymbol{p}$ has degenerated sufficiently.

CEnoisy.py

```python
from Snoisy import Snoisy
import numpy as np
n = 100
rho = 0.1
N = 1000; Nel = int(N*rho); eps = 0.01
p = 0.5*np.ones(n)
i = 0
pstart = p
ps = np.zeros((1000,n))
ps[0] = pstart
pdist = np.zeros((1,1000))
while np.max(np.minimum(p,1-p)) > eps:
    i += 1
    X = (np.random.uniform(size=(N,n)) < p).astype(int)
    X_tmp = np.array(X, copy=True)
    SX = Snoisy(X_tmp)
    ids = np.argsort(SX,axis=0)
    Elite = X[ids[0:Nel],:]
    p = np.mean(Elite,axis=0)
    ps[i] = p
print(p)
```

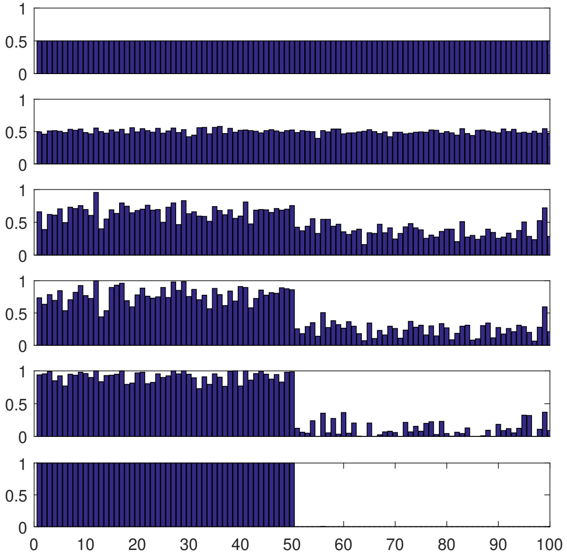


Figure: Evolution of the vector of probabilities $\boldsymbol{p} = [p_1, \ldots, p_n]$ towards the degenerate solution.