Motoare de inferenţă

Cursul 11

Programare şi modelare probabilistă - anul III

Facultatea de Informatică, UAIC

e-mail: adrian.zalinescu@uaic.ro

web: https://sites.google.com/view/fiicoursepmp/home

8 Ianuarie 2024

Până acum, ne-am concentrat atenţia asupra construcţiei de modele, interpretarea rezultatelor şi critica modelelor.

- Ne-am bazat astfel doar pe *butonul de inferenţă*, interpretat în PyMC de funcţia `pm.sample`, pentru a calcula distribuţia a posteriori.
- În acest curs ne vom concentra pe unele detalii privind motorul de inferenţă din spatele acestei funcţii.
- Scopul programării probabiliste este de a face eşantionarea din distribuţia a posteriori, nu de a înţelege cum se face aceasta.
- Totuşi, acest lucru este important pentru a înţelege procesul de inferenţă şi ar putea să ne ajute atunci când aceste metode eşuează, pentru a vedea ce se poate face.

Plan

1. Motoare de inferenţă

2. Metode non-Markoviene
   - Grid computing
   - Metoda pătratică
   - Metode variaţionale

3. Metode Markoviene
   - Metropolis-Hastings
   - Hamiltonian Monte Carlo
   - Monte Carlo secvenţial

Motoare de inferenţă

Deşi conceptual sunt simple, metodele Bayesiene se pot dovedi dificile din punct de vedere matematic sau numeric.

- Motivul este că verosimilitatea marginală, numitorul din formula lui Bayes

$$p(\theta \mid y) = \frac{p(y \mid \theta) \cdot p(\theta)}{p(y)},$$

  ia de obicei forma unei integrale costisitoare din punct de vedere computaţional,

$$p(y) = \int p(y \mid \theta) \cdot p(\theta) d\theta.$$

- Din acest motiv, distribuţia a posteriori este de obicei estimată numeric folosind algoritmi din familia *Markov Chain Monte Carlo* (*MCMC*) sau, mai recent, *algoritmi variaţionali*.
- Existenţa acestor metode, numite motoare de inferenţă, au motivat dezvoltarea limbajelor de programare probabilistă precum PyMC.

- Scopul limbajelor de programare probabilistă este de asepara procesul construcţiei modelelor de procesul de inferenţă pentru a facilita paşii iterativi ai *construcţiei modelului*, *evaluării* şi *modificării/extinderii* acestuia.
- Prin tratarea procesului de inferenţă ca o cutie neagră (black box), utilizatorii limbajelor de programare probabilistă sunt liberi să se concentreze pe problemele lor specifice, lăsându-le pe acestea să gestioneze detaliile computaţionale.
- Dar înainte de apariţia acestora, cei care proiectau modelele probabiliste îşi scriau singuri metodele de inferenţă, în general croite după modelul considerat sau simplificau modelele pentru a le face potrivite pentru anumite aproximări matematice.
- Acest lucru prezintă avantajul eleganţei şi chiar unei maniere mai eficiente de a calcula distribuţia a posteriori, dar poate fi supusă erorilor şi este consumatoare de timp.

Grid computing

Acesta este o metodă de *forţă brută*:
Să presupunem că vrem să calculăm distribuţia a posteriori pentru un model cu un singur parametru. Atunci aproximarea de tip *grid* se face în modul următor:

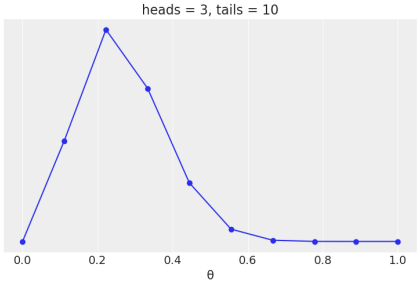
- Se defineşte un interval rezonabil pentru paramtrul θ (distribuţia a priori ne poate da un indiciu).
- Se partiţionează intervalul (în general în sub-intervale de lungime egală).
- Pentru fiecare punct θ din *diviziune* (*grid*), se multiplică verosimilitatea cu probabilitatea a priori: $p(y \mid \theta) \cdot p(\theta)$.

Codul de mai jos implementează această metodă pentru a calcula distribuţia a posteriori pentru modelul aruncării cu banul:

```
def posterior_grid(grid_points=50, heads=6, tails=9):
    """
    A grid implementation for the coin-flipping problem
    """
    grid = np.linspace(0, 1, grid_points)
    prior = np.repeat(1/grid_points, grid_points)  # uniform prior
    likelihood = stats.binom.pmf(heads, heads+tails, grid)
    posterior = likelihood * prior
    posterior /= posterior.sum()
    return grid, posterior
```

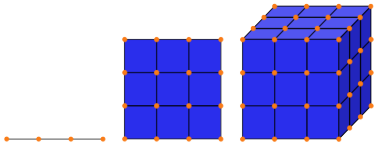
Să presupunem că am aruncat de 13 ori o monedă şi am observat trei steme:

```
data = np.repeat([0, 1], (10, 3))
points = 10
h = data.sum()
t = len(data) - h
grid, posterior = posterior_grid(points, h, t)
plt.plot(grid, posterior, 'o-')
plt.title(f'heads = {h}, tails = {t}')
plt.yticks([])
plt.xlabel('θ');
```



Se poate observa că un număr mai mare de puncte (echivalent: o diviziune mai fină) duce la o aproximare mai bună:

- distribuţia a posteriori este obţinută ca limită a aproximării atunci când numărul de puncte creşte către $+\infty$.
- Pe de altă parte, acest proces creşte costul computaţional.
- Cel mai mare dezavantaj al acestei metode este că se comportă din ce în ce mai rău pe măsură ce numărul de parametri creşte (absenţa *scalabilităţii*).



Pe lângă faptul că numărul de puncte din diviziune creşte, se manifestă un alt fenomen:

- regiunea din spaţiul parametrilor unde este concentrată distribuţia a posteriori este din ce în ce mai mică faţă de volumul eşantionat.
- Acest fenomen întâlnit în statistică şi machine learning se numeşte *blestemul dimensionalităţii* sau *concentrarea măsurilor*:
- De exemplu, într-un hypercub, marea parte a volumului este lângă frontiera acestuia, nu în centru.
- Din punctul nostru de vedere, problema este că mare parte a timpului este petrecută alegând valori care au o contribuţie aproape nulă pentru distribuţia a posteriori, risipind în acest fel resurse valoroase.

Metoda pătratică

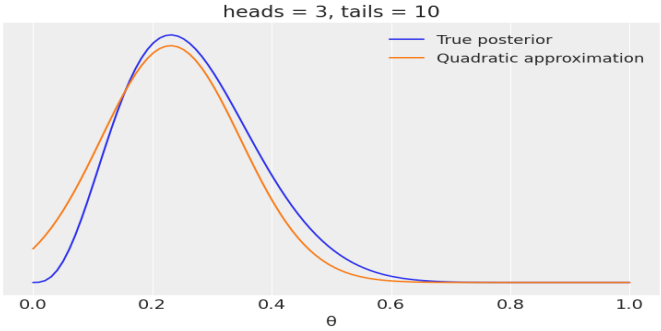
Cunoscută şi ca *metoda Laplace* sau *aproximarea normală*, *metoda pătratică* constă în aproximarea distribuţiei a posteriori, pe care o notăm aici $p(x)$, cu o distribuţie Gaussiană, $q(x)$.
Metoda are doi paşi:

- Găsim *modul* distribuţiei a posteriori, adică $\mu = \text{argmax}\, p(x)$. Acesta va fi media lui $q(x)$.
- Calculăm matricea Hessiană a lui $p(x)$ în $x = \mu$. Vom calcula deviaţia standard a lui $q(x)$ ca inversa rădăcinii pătrate a Hessianului (ce dă curbura lui $p(x)$ în μ).

```
with pm.Model() as normal_aproximation:
    p = pm.Beta('p', 1., 1.)
    w = pm.Binomial('w',n=1, p=p, observed=data)
    mean_q = pm.find_MAP()
    std_q = ((1/pm.find_hessian(mean_q, vars=[p]))**0.5)[0]
mean_q['p'], std_q
```

Să vedem cum arată aproximarea pătratică pentru modelul beta-binomial:

```
# analytical calculation
x = np.linspace(0, 1, 100)
plt.plot(x, stats.beta.pdf(x , h+1, t+1),
         label='True posterior')
# quadratic approximation
plt.plot(x, stats.norm.pdf(x, mean_q['p'], std_q),label='Quadratic approximation')
plt.legend(loc=0, fontsize=13)
plt.title(f'heads = {h}, tails = {t}')
plt.xlabel('θ', fontsize=14)
plt.yticks([])
```



Observaţii:

- Putem aplica metoda Laplace variabilelor nemărginite, adică variabilelor care au suportul (trăiesc) în $\mathbb{R}^N$, din cauză că
- distribuţia Gaussiană este o distribuţie nemărginită, aşa că
- dacă încercăm să modelăm cu ajutorul ei o distribuţie mărginită, vom ajunge să estimăm o densitate pozitivă acolo unde ea ar trebui să fie nulă.
- Putem totuşi aplica această metodă dacă transformăm mai intâi variabila mărginită într-una nemărginită.
- De exemplu, folosim de obicei o distribuţie HalfNormal pentru a modela deviaţia standard din cauză că aceasta ia valori pozitive; putem să o facem nemărginită dacă extragem logaritmul din aceasta.

Metode variaţionale

Acestea sunt alternative la metodele Markoviene care ar putea fi o alegere mai bună în cazurile în care:

- avem seturi de date foarte mari (big data) şi/sau
- pentru distribuţii a posteriori foarte costisitoare din punct de vedere computaţional.

Ideea este de a aproxima distribuţia a posteriori cu distribuţii mai simple (cum am făcut cu metoda Laplace, dar într-un mod mai elaborat).

- Vom găsi această distribuţie mai simplă prin rezolvarea unei probleme de optimizare, găsind *cea mai apropiată* distribuţie de cea a posteriori.
- Trebuie deci să definim ce înseamnă *apropiere*.

O modalitate de a măsura această apropiere între distribuţii este prin folosirea *divergenţei Kullback-Leibler* (*KL*):

$$\mathrm{D}_{KL}\big(q(\theta) \parallel p(\theta|y)\big) := \int q(\theta) \log \frac{q(\theta)}{p(\theta|y)} d\theta,$$

unde q este distribuţia mai simplă folosită pentru aproximarea celei a posteriori, $p(\theta|y)$. Cum $p(\theta|y)$ este necunoscută, căutăm moduri alternative de a găsi soluţii ale problemei variaţionale

$$\operatorname{argmin}_{q(\theta)} \mathrm{D}_{KL}\big(q(\theta) \parallel p(\theta|y)\big).$$

Avem:

$$\begin{aligned} \mathrm{D}_{KL}(q(\theta) \parallel p(\theta|y)) &= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} p(y) d\theta \\ &= \int q(\theta) \left[\log \frac{q(\theta)}{p(\theta, y)} + \log p(y) \right] d\theta \\ &= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} d\theta + \int q(\theta) \log p(y) d\theta \\ &= \int q(\theta) \log \frac{q(\theta)}{p(\theta, y)} d\theta + \log p(y), \end{aligned}$$

deoarece $\int q(\theta) d\theta = 1$.

Deci

$$\mathrm{D}_{KL}\big(q(\theta) \parallel p(\theta|y)\big) = -\underbrace{\int q(\theta)\log\frac{p(\theta,y)}{q(\theta)}d\theta}_{\text{evidence lower bound } (ELBO)} + \log p(y)$$

- Cum D_{KL} este pozitivă, avem $\log p(y) \geq ELBO$, adică distribuţia marginală (evidenţa) este întotdeauna mai mare decât $ELBO$, de unde numele.
- Deoarece $\log p(y)$ este constantă, problema revine la a maximiza $ELBO$.
- $q(\theta)$ va fi aleasă dintr-o familie de distribuţii cât mai simple:
  - o soluţie este de a alege $q(\theta)$ ca fiind descrisă de distribuţii 1-dimensionale independente:
  $$q(\theta) = \prod_j q_j(\theta_j),$$
  - iar q_j alese din *familia exponenţială* de distribuţii, ce cuprinde distribuţiile normală, exponenţială, Beta, Dirichlet, Gamma, Poisson , categorială (deci şi Bernoulli).

- Din păcate, metoda de mai sus vine cu an algoritm specific pentru fiecare model, deci nu avem o metodă universală de inferenţă, (pentru toate modelele).
- Din fericire, au fost propuse soluţii pentru automatizarea metodelor variaţionale.
- O metodă recent propusă este *Automatic Differentiation Variational Inference* (*ADVI*), în `http://arxiv.org/abs/1603.00788`, ai cărei paşi sunt:
  - transformarea tuturor distribuţiilor mărginite în distribuţii nemărginite (aşa cum a fost discutat pentru metoda Laplace);
  - aproximarea parametrilor nemărginiţi cu o distribuţie Gaussiană (acei q_j); este de remarcat faptul că o distribuţie Gaussiană pe spaţiul transformat al parametrilor este non-Gaussiană pe spaţiul original.
  - folosirea diferenţierei automate pentru a maximiza *ELBO*.
- Documentaţia PyMC oferă multe exemple privind folosirea inferenţei variaţionale în PyMC.

Metode Markoviene

Constituie o familie, cunoscute drept metode *MCMC* (*Markov Chain Monte Carlo*), ce dau posibilitatea de a eşantiona din distribuţia a posteriori, atât timp cât putem calcula punctual verosimilitatea şi distribuţia a priori.

- Metodele MCMC sunt capabile de a lua mai multe eşantioane din regiunile de probabilitate mai mare decât cele de probabilitate mai mică, în felul acesta depăşind în performanţă grid computing.
- La nivel fundamental, esenţial în statistică este calculul mediilor de tipul următor:

$$\mathbb{E}[f] = \int p(\theta) f(\theta) d\theta.$$

- Cu MCMC, aproximăm cantitatea de mai sus cu

$$\mathbb{E}_\pi[f] = \frac{1}{N}\sum_{n=1}^{N} f(\theta_n),$$

unde $\theta_1, \ldots, \theta_N$ reprezintă un eşantion finit conform distribuţiei p, deoarece

$$\lim_{N\to\infty} \mathbb{E}_\pi[f] = \mathbb{E}[f],$$

conform LNM (legea numerelor mari).

- A fi siguri că un eşantion particular aproximează bine cantitatea căutată nu este o problemă uşoară.
- În practică, se fac teste empirice pentru a fi ziguri că avem o bună aproximare MCMC.

În continuare vom aminti câteva aspecte ale celor două părţi componente ale MCMC: *Monte Carlo* şi *lanţuri Markov*.

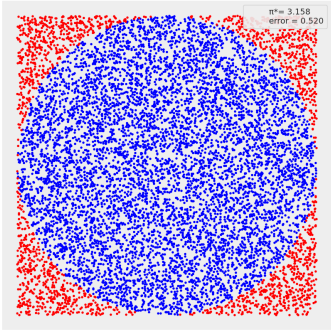
Monte Carlo

- Metodele *Monte Carlo* reprezintă o clasă largă de algoritmi care folosesc eşantioane aleatoare pentru a calcula sau simula un anumit proces.
- Unul dintre dezvoltatorii acestei metode, Stanislaw Ulam, a avut ideea că pentru multe probleme ce sunt dificil de rezolvat sau de formulat de o manieră precisă, ele pot fi studiate prin prelevarea de eşantioane din acestea.
- De exemplu, calculul probabilităţii de a obţine o anumită configuraţie la jocul de *Solitaire*.
- Una din primele aplicaţii a fost rezolvarea unei probleme de fizică nucleară.
- În zilele noastre, chiar şi PC-urile sunt suficient de puternice pentru a rezolva multe probleme interesante folosind metoda Monte Carlo.

Un exemplu clasic este determinarea valorii lui π, folosind următoarea procedură:

- Aruncaţi N puncte la întâmplare în interiorul unui pătrat de latură $2R$;
- Desenaţi un cerc de rază R înscris în cerc şi număraţi câte puncte N_{int} sunt în interiorul acelui cerc.
- Estimaţi π ca raportul $4N_{int}/N$.

```
N = 10000
x, y = np.random.uniform(-1, 1, size=(2, N))
inside = (x**2 + y**2) <= 1
pi = inside.sum()*4/N
error = abs((pi - np.pi) / pi) * 100
outside = np.invert(inside)
plt.figure(figsize=(8, 8))
plt.plot(x[inside], y[inside], 'b.')
plt.plot(x[outside], y[outside], 'r.')
plt.plot(0, 0, label=f'π*= {pi:4.3f}\n
        error = {error:4.3f}', alpha=0)
plt.axis('square')
plt.xticks([])
plt.yticks([])
plt.legend(loc=1, frameon=True, framealpha=0.9)
```



Lanţuri Markov

- Un *lanţ Markov* este un obiect matematic care consistă într-un şir de stări şi un set de probabilităţi de tranziţie care descriu mişcarea între stări.
- Având la dispoziţie un astfel de lanţ, putem efectua o *plimbare aleatoare* (*random walk*) alegând un punct de plecare şi apoi trecând prin diverse stări conform probabilităţilor de tranziţie.
- Dacă am putea găsi un lanţ Markov cu probabilităţile de tranziţie proporţionale cu distribuţia din care dorim să eşantionăm (în cazul nostru, distribuţia a posteriori), eşantionarea devine doar o plimbare aleatoare printre valorile căutate (în cazul nostru, prin spaţiul de parametri).
- O cerinţă des întâlnită este ca lanţul Markov să fie *reversibil*, adică probabilitatea de a trece din starea i în starea j să fie aceeaşi ca cea de a trece din starea j în starea i.

Metropolis-Hastings

Este probabil cea mai populară metodă de tip MCMC.

- Pentru unele distribuţii, ca cea Gaussiană, există algoritmi eficienţi de eşantionare, dar pentru altele nu este cazul.
- *Metropolis-Hastings* ne permite să eşantionăm din orice distribuţie de probabilitate, $p(x)$, atât timp cât putem calcula măcar o valoare proporţională cu acesta, ignorând astfel factorul de normalizare, care de obicei este greu de estimat (în cazul nostru, acesta este verosimilitatea marginală).

Pentru a înţelege acest concept, vom folosi următoarea analogie:

- Să presupunem că suntem interesaţi în a găsi volumul unui lac şi de asemenea punctul în care lacul are cea mai mare adâncime.
- De asemenea, lacul este mâlos şi foarte întins, aşa că o aproximare de tip grid nu e o idee prea bună.
- Strategia de eşantionare necesită doar o barcă şi un băţ foarte lung:

Paşii sunt următorii:

1. Alegem un loc aleatoriu din lac şi mutăm barca acolo.
2. Folosim băţul pentru a măsura adâncimea lacului.
3. Mutăm barca într-un alt loc şi măsurăm din nou.
4. Comparăm cele două măsurători astfel:
   - Dacă locul nou este mai adânc decât primul, notăm adâncimea noului loc şi repetăm pasul 2.
   - Dacă este mai puţin adânc, avem două opţiuni: să acceptăm sau să respingem. Acceptare înseamnă să notăm adâncimea noului loc şi să repetăm pasul 2. Respingere înseamnă să mergem înapoi în locul anterior şi să notăm (din nou!) adâncimea acestuia.

Regula pentru a decide dacă să acceptăm sau să respingem este cunoscută drept *criteriul Metropolis-Hastings*:

- în principiu, aceasta stipulează că vom accepta noul loc cu o probabilitate care este proporţională cu raportul între adâncimea noului loc şi a celui vechi.
- Folosind această procedură, vom găsi nu numai volumul total şi adâncimea maximă a lacului, dar şi forma fundului acestuia.

Evident, fundul lacului reprezintă densitatea distribuţiei pe care o căutăm (cu semn schimbat), adâncimea lacului este modul distribuţiei, iar volumul lacului este factorul de normalizare.
Astfel, algoritmul Metropolis-Hastings are următorii paşi:

1. Alegem o valoare iniţială pentru parametru, x_i (la întâmplare sau conform intuiţiei).
2. Alegem un nou parametru, x_{i+1}, prin simularea dintr-o distribuţie uşor de eşantionat, precum distribuţia Gaussiană sau uniformă, $q(x_{i+1}|x_i)$.
3. Calculăm probabilitatea de a accepta noul parametru prin utilizarea criteriului Metropolis-Hastings:

$$p_a(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})q(x_i|x_{i+1})}{p(x_i)q(x_{i+1}|x_i)}\right).$$

4. Dacă probabilitatea de la pasul 3 este mai mare decât valoarea rezultată dintr-o distribuţie uniformă pe $[0, 1]$, atunci acceptăm noua stare; dacă nu, rămânem în starea anterioară.
5. Iterăm pasul 2 până obţinem un eşantion suficient de mare.

Observaţii:

- Dacă distribuţia $q(x_{i+1}|x_i)$ este simetrică, obţinem criteriul *Metropolis*:

$$p_a(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})}{p(x_i)}\right).$$

- Paşii 3 şi 4 asigură că eşantionarea va fi făcută predominant în zonele cu valori ale lui p mai mari (mai adânci ale lacului).
- Distribuţia ţintă (distribuţia a posteriori în cazul nostru) este aproximată de lista parametrilor din eşantion. În caz de acceptare, adăugăm listei noua valoare x_{i+1}; dacă respingem, adăugăm listei valoarea lui x_i (chiar dacă valoarea se repetă).
- În final, distribuţia este doar un `array` de valori, simplu de manipulat.

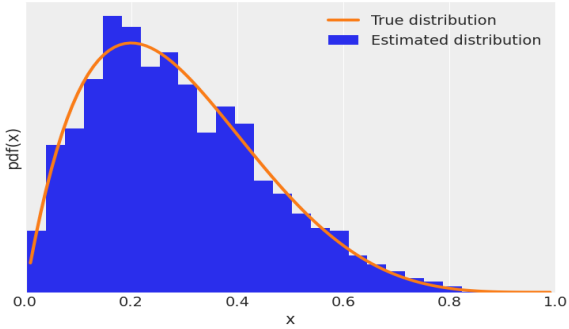
Exemplu de implementare a algoritmului Metropolis

- pentru o distribuţie oarecare:

```
def metropolis(func, draws=10000):
    """A very simple Metropolis implementation"""
    trace = np.zeros(draws)
    old_x = 0.5  # func.mean()
    old_prob = func.pdf(old_x)
    delta = np.random.normal(0, 0.5, draws)
    for i in range(draws):
        new_x = old_x + delta[i]
        new_prob = func.pdf(new_x)
        acceptance = new_prob / old_prob
        if acceptance >= np.random.random():
            trace[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            trace[i] = old_x
    return trace
```

- pentru cazul particular al unei distribuţii Beta:

```
func = stats.beta(2, 5)
trace = metropolis(func=func)
x = np.linspace(0.01, .99, 100)
y = func.pdf(x)
plt.xlim(0, 1)
plt.plot(x, y, 'C1-', lw=3, label='True distribution')
plt.hist(trace[trace > 0], bins=25, density=True, label='Estimated distribution')
plt.xlabel('x')
plt.ylabel('pdf(x)')
plt.yticks([])
plt.legend()
```



- Eficienţa algoritmului depinde de distribuţia q, numită *distribuţia de propunere*:
- dacă starea propusă este foarte departe de cea actuală, atunci probabilitatea de respingere este foarte mare;
- dacă starea propusă este foarte aproape, explorarea spaţiului de parametri se face foarte încet.
- În general, distribuţia de propunere este o distribuţie Gaussiană multivariată, a cărei matrice de covarianţă este determinată în faza de *tuning* (*calibrare*).
- PyMC calibrează covarianţa folosind următoarea regulă generală:
  - rata de acceptare ideală este între 50% pentru o distribuţie Gaussiană 1-dimensională şi 23% pentru o distribuţie Gaussiană n-dimensională.

Hamiltonian Monte Carlo

- Metodele MCMC vin cu garanţia că dacă eşantionul generat este suficient de mare, vom găsi o aproximare precisă a distribuţiei căutate.
- În practică, s-ar putea ca acest lucru să ia mai mult timp decât avem la dispoziţie.
- Din această cauză, au fost propuse alternative la Metropolis-Hastings.
- Multe din aceste alternative, ca de altfel şi Metropolis-Hastings, au fost concepute pentru a rezolva probleme din *mecanica statistică*, o ramură a fizicii care studiază proprietăţile sistemelor atomice şi moleculare.
- O astfel de modificare este *Hamiltonian Monte Carlo* (*HMC*): în termeni simpli, un *Hamiltonian* descrie energia totală a unui sistem fizic.

Metoda HMC este practic aceeaşi ca Metropolis-Hasting, cu *excepţia* faptului că propunerea unei noi poziţii *nu este aleatoare*:

- în loc de a muta barca în mod aleatoriu, folosim curbura fundului lacului:
- imprimăm unei bile perfecte (fără frecare) o mişcare pe fundul lacului pornind de la poziţia curentă pentru un scurt moment până o oprim;
- acceptăm sau respingem noua poziţie folosind criteriul Metropolis (unde nu apărea q).
- repetăm procedura *de multe ori*.

Metoda aceasta evită unul din principalele dezavantaje ale algoritmului Metropolis-Hastings: o explorare eficientă a spaţiului de parametri presupune respingerea multor paşi propuşi.

Pe de altă parte, acest lucru vine cu un preţ: trebuie estimat *gradientul* funcţiei ţintă (bila se va mişca în direcţia acestuia, cu o viteză proporţională cu magnitudinea acestuia). Astfel:

- fiecare pas al HMC este mai costisitor decât unul al Metropolis-Hastings, dar
- probabilitatea de aceptare a acestuia este mai mare decât pentru Metropolis-Hastings.

Pentru a înclina balanţa în favoarea HMC, trebuie din nou *calibraţi* (*tuned*) unii parametri ai HMC:

- procedura cere mai multă experienţă decât pentru Metropolis-Hastings (ceea ce face HMC mai puţin popular), dar
- PyMC vine cu un *sampler* relativ nou, cunoscut ca *No-U-Turn Sampler* (*NUTS*).
- Dezavantajul acestuia este că merge doar cu distribuţii continue (nu putem calcula gradienţi pentru distribuţii discrete).

Animaţii pentru vizualizarea a diverşi algoritmi MCMC:
`https://chi-feng.github.io/mcmc-demo/`

Monte Carlo secvenţial

Una dintre problemele cu care se confruntă Metropolis-Hasting şi NUTS (şi alte variante HMC) este că dacă distribuţia a posteriori are vârfuri multiple (separate de regiuni de foarte mică probabilitate), aceste metode pot să rămână blocate în zone uni-modale şi să le rateze pe altele.

- Multe soluţii dezvoltate pentru a trece de aceste probleme legate de minime locale multiple sunt legate de ideea de a introduce *temperatura* sistemului.
- Această idee vine, din nou, din mecanica statistică:
  - dacă temperatura sistemului este 0°K (minim absolut), acesta este blocat (îngheţat) într-o singură stare;
  - dacă temperatura sistemului este infinită, atunci stările sistemului sunt echiprobabile;
  - noi suntem interesaţi de stări intermediare, între aceste extreme.

- Pentru modelele Bayesiene, această idee se adaptează printr-o variantă a formulei lui Bayes:
$$p(\theta|y)_\beta = p(y|\theta)^\beta p(\theta),$$
unde diferenţa este legată de specificarea parametrului β, cunoscut drept *temperatura inversă*:
  - dacă $\beta = 0$, atunci *posteriorul temperat* $p(\theta|y)_\beta$ este distribuţia a priori;
  - dacă $\beta = 1$, atunci $p(\theta|y)_\beta$ este distribuţia a posteriori (până la factorul de normalizare).
- Cum eşantionarea din distribuţia a priori este în general mai uşoară decât cea din distribuţia a posteriori, vom începe eşantionarea din distribuţia mai simplă (cu $\beta = 0$) şi transformarea înceată în distribuţia complexă pe care o avem drept ţintă.

O metodă care exploatează această idee este *Monte Carlo secvenţial* (*SMC*). În PyMC ea este implementată în felul următor:

1. Iniţializăm $\beta = 0$.
2. Generăm un eşantion de N valori, S_β, din distribuţia temperată.
3. Creştem *puţin* pe β.
4. Calculăm un set de N ponderi W, conform noii distribuţii temperate.
5. Obţinem S_W prin reeşantionarea S_β conform W.
6. Pornim N lanţuri Metropolis, pornind pe fiecare valoare din S_W.
7. Ne întoarcem la pasul 3 până când $\beta \geq 1$.

Pasul de reeşantionare funcţionează prin ştergerea valorilor de probabilitate mică şi înlocuirea acestora cu valori de probabilitate mare.

Eficienţa acestei metode depinde mult de valorile intermediare ale lui β, cunoscute drept *schema de răcire*.

- Cu cât este mai mică diferenţa între două valori succesive ale lui β, cu atât cele două distribuţii temperate vor fi mai apropiate, şi astfel tranziţia de la o etapă la alta va fi mai uşoară.
- Pe de altă parte, dacă paşii sunt prea mici, vom irosi multe resurse computaţionale, fără a îmbunătăţi semnificativ acurateţea rezultatelor.
- Din fericire, SMC poate calcula automat valorile intermediare ale lui β; schema de răcire va fi adaptată în funcţie de dificultatea problemei: distribuţiile mai complicate vor necesita mai multe etape.

- În figura de mai jos, primul grafic arată un eşantion de 5 valori (în portocaliu) la o anumită etapă.
- Al doilea grafic arată cum aceste valori sunt re-ponderate conform distribuţiei a posteriori ponderate (în albastru este densitatea acesteia).
- Al treilea grafic ne arată rezultatul unui anumit număr de paşi Metropolis, iniţializaţi din eşantionul re-ponderat din al doilea grafic. Remarcăm cum 2 valori (de pondere mică) nu sunt folosite pentru noile lanţuri Markov.