#187 メンバー / TakumiOtagaki / 研究進捗 /

WIP jax-capr

Updated at 2025-11-04 11:38

はじめに

実装力がないと言われてクッソーと思ったので Max Ward らに任せるのではなく、彼らの `jax-rnafold` を利用して jax-capr を実装してやろうじゃないか。と考えています。

jax-rnafold の dp table

実装で利用されているのは

- $\xi(i)$: 実質 $Z(i, n)$
- $P(i, j)$: 実質 $Z^b(i, j)$
- $M(p, i, j)$: Z^m と Z^{m1} を併せ持っている（ここがちょっとややこしい）
  - i, j がマルチループの中にあって、その中に 少なくとも p 個の塩基対が存在するような全ての部分構造 $\sigma[i; j]$ のボルツマンファクターの和
  - i, j による塩基対は含めない。

また、outside algorithm が未実装であるという課題があるが、これは落ち着いてやればできそう。
forward の漸化式から半自動的に outside を導出できる.

forward 漸化式 in jax-rnafold

$$
\begin{aligned}
\xi(i+1) &= \xi(i) \cdot \exp(-\beta \cdot 0) \\
&\quad + \sum_{\substack{j \\ j<i}} \xi(j) P(i, j) \\
P(i, j) &= B(f_1(i, j)) + \sum_{i<h<l<j} P(h, l) B(f_2(h, l)) \\
&\quad + M(2, i+1, j-1) B(M_c) B(M_p) \\
M(p, i, j) &= M(p, i+1, j) B(M_u) \\
&\quad + \sum_{i<k<j} P(i, k) B(M_p) M(\max(0, p-1), k+1, j)
\end{aligned}
$$

ここで $\bar{\xi}$ のようにバーをつけることで backward を表すことにすると

$$
\begin{aligned}
\bar{\xi}(i) &= \bar{\xi}(i-1) + \sum_{\substack{j \\ j<i-1}} \bar{\xi}(j) P(i-1, j) \\
\bar{P}(h,l) &= \bar{\xi}(h)\xi(l+1) \\
&\quad + \sum_{\substack{i,j \\ i<h,l<j}} \bar{P}(i,j) \begin{pmatrix} B(f_2(i,j,h,l)) \\ +B(M_c + M_p) \begin{Bmatrix} +M(1,i+1,h-1) \cdot B(M_u(j-l-1) \\ +M(1,l+1,j-1) \cdot B(M_u(h-i-1) \\ +M(1,i+1,h-1) \cdot M(1,l+1,j-1) \end{Bmatrix} \end{pmatrix} \\
&= \bar{\xi}(h)\xi(l+1) \\
&\quad + \sum_{\substack{i,j \\ i<h,l<j}} \bar{P}(i,j) \begin{pmatrix} B(f_2(i,j,h,l)) \\ +B(M_c + M_p) \begin{bmatrix} M(1,i+1,h-1) \cdot \{B(M_u(j-l-1)) + M(1,l+1,j-1)\} \\ +M(1,l+1,j-1) \cdot B(M_u(h-i-1)) \end{bmatrix} \end{pmatrix}
\end{aligned}
$$

のようにしてかける。
また、懸念として $B(f_2)$ の計算で internal outer mismatch を考慮する際に、少し散らかった式が出てくるだろうということが挙げられる。これは実装の際にもう少し詳しくみてみることにする。

以下のようにして計算量を下げることができる。

$$
\begin{aligned}
\bar{P}^m(i,l) &:= \sum_{\substack{j \\ l<j}} \bar{P}(i,j) M(1,l+1,j-1) \\
\bar{P}^{m1}(i,l) &:= \sum_{\substack{j \\ l<j}} \bar{P}(i,j) B(M_u(j-l-1))
\end{aligned}
$$

これらはいずれも より長い領域の計算が終わっていればすぐに計算できるし、$O(n^3)$ で全て計算することができる。

これらを用いることで、元々の $\bar{P}(h,l)$ は以下のように書き直すことができる。すなわち、

$$
\begin{aligned}
\bar{P}(h,l) &= \bar{\xi}(h)\xi(l+1) \\
&\quad + \sum_{\substack{i,j \\ i<h,l<j}} \bar{P}(i,j) \begin{pmatrix} B(f_2(i,j,h,l)) \\ +B(M_c + M_p) \begin{bmatrix} M(1,i+1,h-1) \cdot \{B(M_u(j-l-1)) + M(1,l+1,j-1)\} \\ +M(1,l+1,j-1) \cdot B(M_u(h-i-1)) \end{bmatrix} \end{pmatrix} \\
&= \bar{\xi}(h)\xi(l+1) + \sum_{\substack{i,j \\ i<h,l<j}} B(f_2(i,j,h,l)) \\
&\quad + B(M_c + M_p) \sum_{\substack{i \\ i<h}} \sum_{\substack{j \\ l<j}} \begin{bmatrix} \bar{P}(i,j) M(1,i+1,h-1) B(M_u(j-l-1)) \\ +\bar{P}(i,j) M(1,i+1,h-1) M(1,l+1,j-1) \\ +\bar{P}(i,j) M(1,l+1,j-1) B(M_u(h-i-1)) \end{bmatrix} \\
&= \bar{\xi}(h)\xi(l+1) + \sum_{\substack{i,j \\ i<h,l<j}} B(f_2(i,j,h,l)) \\
&\quad + B(M_c + M_p) \sum_{\substack{i \\ i<h}} \begin{bmatrix} M(1,i+1,h-1) \bar{P}^{m1}(i,l) \\ +M(1,i+1,h-1) \bar{P}^{m}(i,l) \\ +B(M_u(h-i-1)) \bar{P}^{m}(i,l) \end{bmatrix} \\
&= \bar{\xi}(h)\xi(l+1) + \sum_{\substack{i,j \\ i<h,l<j \\ h-i-1+j-l-1 \leq 30}} B(f_2(i,j,h,l)) \\
&\quad + B(M_c + M_p) \sum_{\substack{i \\ i<h}} \begin{bmatrix} M(1,i+1,h-1) \bar{P}^{m1}(i,l) \\ +\{M(1,i+1,h-1) + B(M_u(h-i-1))\} \bar{P}^{m}(i,l) \end{bmatrix}
\end{aligned}
$$

となって、これは $O(n^3)$ で全て計算することができる。
この方向性で実装を進めれば、待望の outside partition function $P(i,j)$ を手に入れることができる...

これをベクトル化するためには 同じ配列長さ d となる任意の h, l つまり

$$\forall (h,l),(h',l'), \text{s.t. } l-h-1 = l'-h'-1 (=d), (h,l) \neq (h',l')$$

に対して $\bar{P}(h,l)$ と $\bar{P}(h',l')$ の計算は互いに依存しないため、それぞれを vectorization することができる。
さらに、各 $\bar{P}(h,l)$ の計算の中で登場する $\sum_i$ に関しても vectorization することで効率的に和を計算することができる。
これらのベクトル化をかませば、対角線のサイズ $O(n)$ だけの計算時間で計算を終わらせることができる（メモリが無限にある限り）。

続いて、M についても outside algorithm を走らせる必要があって、

$$\begin{aligned}
M(2,i,j) &= M(2,i+1,j)B(M_u) + \sum_{i<k<j} P(i,k)B(M_p)M(1,k+1,j) \\
M(1,i,j) &= M(1,i+1,j)B(M_u) + \sum_{i<k<j} P(i,k)B(M_p)M(0,k+1,j) \\
M(0,i,j) &= M(0,i+1,j)B(M_u) + \sum_{i<k<j} P(i,k)B(M_p)M(0,k+1,j)
\end{aligned}$$

であって $M(2,i,j)$ は P の漸化式でも出てくることから、以下のように書き下すことができる。

$$\begin{aligned}
\bar{M}(2,h,l) &= \bar{M}(2,h-1,l)B(M_u) + \bar{P}(h-1,l+1)B(M_c+M_p) \\
\bar{M}(1,h,l) &= \bar{M}(1,h-1,l)B(M_u) + \sum_{i<h-1} P(i,h-1)B(M_p)\bar{M}(2,i,l) \\
\bar{M}(0,h,l) &= \bar{M}(0,h-1,l)B(M_u) + \sum_{i<h-1} P(i,h-1)B(M_p)\left(\bar{M}(0,i,l) + \bar{M}(1,i,l)\right)
\end{aligned}$$

のように書き下すことができる。
これにて、ξ, P, M それぞれの outside algorithm が仕上がった。あとは実装。
ここまで丁寧な情報整理ができていればすぐに実装できる気がする。

残った課題

Q. general internal loop に加算される outer mismatch (OMM) の table について backward を計算する必要はあるのか？
A. おそらく yes. OMM によって計算された項を P(i, j) が受け取ることになっている。さらに、OMM の計算にも P が出てくるから。だけど OMM は P としか関係を持っていないシンプルな構造なので、工夫次第で実装はシンプルにできる。けどそれは自動微分の話であって、構造を合理的に頭で考えて構成した場合は不要にも思えてきて... みたいな悩み。ご飯食べてから詰めることにする。

1102 update

内側アルゴリズムと外側アルゴリズムを整理する。

内側アルゴリズム:

$$\xi(i+1) = \xi(i)B(0) + \sum_{\substack{j \\ j<i}} \xi(j)P(i,j)$$

$$P(i,j) = B(f_1(i,j)) + \sum_{\substack{h,l \\ i<h<l<j}} P(h,l)B(f_2(i,j,h,l,OMM)$$

$$M(p,i,j) = M(p,i+1,j)B(M_u) + \sum_{\substack{k \\ i<l<j}} P(i,k)B(M_p)M(\max(0,p-1),k+1,j)$$

$$OMM(i,j) = \text{energy\_internalloop\_mismatch}(x_i, x_j, x_{i-1}, x_{j+1})$$

$$B(f_2(i,j,h,l,OMM)) = B(f_{2,\text{without\_il\_mismatch\_energy}}(i,j,h,l)) \times OMM(h,l) \times \text{is\_internal}(i,j,h,l)$$

外側アルゴリズム:

$$\bar{\xi}(i) = \bar{\xi}(i-1) + \sum_{\substack{j \\ j<i-1}} \bar{\xi}(j)P(j,i-1)$$

$$\bar{P}^m(i,l) := \sum_{\substack{j \\ l<j}} \bar{P}(i,j)M(1,l+1,j-1)$$

$$\bar{P}^{m1}(i,l) := \sum_{\substack{j \\ l<j}} \bar{P}(i,j)B(M_u(j-l-1))$$

$$\bar{P}(h,l) := \bar{\xi}(h)\xi(l+1) + \sum_{\substack{i,j \\ i<h,l<j \\ h-i-1+j-l-1 \le 30}} B(f_2(i,j,h,l,OMM))\bar{P}(i,j) + B(M_c+M_p)\sum_{\substack{i \\ i<h}} \begin{bmatrix} M(1,i+1,h-1)\bar{P}^{m1}(i,l) \\ +\{M(1,i+1,h-1)+B(M_u(h-i-1))\}\bar{P}^m(i,l) \end{bmatrix}$$

$$\bar{M}(2,h,l) := \bar{M}(2,h-1,l)B(M_u) + \bar{P}(h-1,l+1)B(M_c+M_p)$$

$$\bar{M}(1,h,l) := \bar{M}(1,h-1,l)B(M_u) + \sum_{i<h-1} P(i,h-1)B(M_p)\bar{M}(2,i,l)$$

$$\bar{M}(0,h,l) := \bar{M}(0,h-1,l)B(M_u) + \sum_{i<h-1} P(i,h-1)B(M_p)\left(\bar{M}(0,i,l)+\bar{M}(1,i,l)\right)$$

1104 update

とりあえず python pseudocode を作成した。

py

```
beta = 1 / 0.6 # 1 / kT
Lmax = 30
import numpy as np

def B(x):
    return np.exp(- beta * x)

def f_2(i, j, h, l, OMM):
    return 1.0 # Placeholder for the actual function

def outside(n, xi, P, OMM, M, en_Mu, en_Mp, en_Mc, Lmax):
    # init ...
    bar_xi = [0.0]*(n+1); bar_xi[n] = 1.0
    bar_P  = np.zeros((n+1, n+1))
    bar_M = np.zeros((3, n+1, n+1))  # bar_M[0]: M0, bar_M[1]: M1, bar_M[2]: M2

    # 集約テーブルの初期化
    bar_Pm  = np.zeros((n+1, n+1))
    bar_Pm1 = np.zeros((n+1, n+1))

    # (A) xi recursion
    for i in range(1, n + 1): # i = 1, 2, ..., n (0-origin)
        bar_xi[i] += xi[i-1] + sum([
            bar_xi[j] * P[j, i-1] for j in range(0, i-1) #j = 0, 1, ..., i - 2
        ])

    # (B) span-descending pass for bar_P and M_bar
    for d in range(n - 1, 0, -1):
        for h in range(1, n - d + 1):
            l = h + d
            # ------- P --------
            # bar_P(h, l)
            bar_P[h, l] += bar_xi[h] * xi[l+1] \
                + sum([
                    B(f_2(i, j, h, l, OMM)) * bar_P[i, j]
                    for i in range(1, h) for j in range(l + 1, n + 1)
                ])
            # bar_P^m(h, l)
            bar_Pm[h, l] += sum([
                bar_P[h, j] * M[1, l + 1, j - 1]
                for j in range (l + 1, n + 1)
            ])
            # bar_P^{m+1}(h, l)
            bar_Pm1[h, l] += sum([
                bar_P[h, j] * (B(en_Mu) ** (j - l - 1))
                for j in range (l + 1, n + 1)
            ])
            # ------- M0, M1, M2 -------
            # bar_M2(h, l)
            bar_M[2, h, l] += bar_M[2, h - 1, l] * B(en_Mu) + bar_P[h - 1, l + 1] * B(en_Mc +
en_Mp)
            # bar_M1(h, l)
            bar_M[1, h, l] += bar_M[1, h - 1, l] * B(en_Mu) + sum([
                P[i, h - 1] * B(en_Mp) * bar_M[2, i, l]
                for i in range(1, h - 1)
            ])
```

```
            # bar_M0(h, l)
            bar_M[0, h, l] += bar_M[0, h - 1, l] * B(en_Mu) + sum([
                P[i, h - 1] * B(en_Mp) * ( bar_M[1, i, l] + bar_M[0, i, l])
                for i in range(1, h - 1)
            ])

    Z = xi[n]
    post = [[0.0]*(n+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(i+1, n+1):
            post[i][j] = (P[i][j] * bar_P[i][j]) / Z
    return post, bar_xi, bar_P, bar_M
```

気づいたこととしては

- $\bar{\xi}$ は ξ の計算を左からやるか右からやるかみたいな差しかない（i, j じゃなくて i だけの左右片方しか動かないから）ため、両者の計算には共通して $P(i, j)$ が利用される。forward が終わっていると P は計算済みとなっているため、そのままほぼ独立に $\bar{\xi}$ を計算することができる。
- dp をこの順番にした理由は
  - まず $\bar{\xi}$ が計算済みになるため、ξ から繋がる table を優先的に計算する必要があるが、それは $\bar{P}$ だけ。だからまずは $\bar{P}$
    - $\bar{P}^m, \bar{P}^{m1}$ などは 0 で初期化しておく
  - 続いて $\bar{P}$ に依存する $\bar{P}^m, \bar{P}^{m1}$ を計算する。
  - それから $\bar{M}$ を計算する。
- また chatGPT に添削させると、$\bar{\xi}$ の更新式が間違っていると指摘される。これはどうやら自動微分に則って更新式を構築すると summation の範囲が i < j となるみたいなことを言っていて、けど二次構造を意識して更新式を作ると絶対に j < i - 1 となるのが正しいみたいな、奇妙なことが起こる。これについては考察をしなくてはならないし、McCaskill の自動微分が、outside に一致するかどうかみたいな話に関わってきそうなので、feature work として検討する必要がありそう。ただ、一つは、上述の通り、xi は左右の片方 (i) しか情報を持っていなくてそれがなんか悪さをしてるのかなーとか（そんなわけない）

1104 update [Scaling]

以下のような方針で scaling を行う。

scaling factor $s()$

$$s(i) := \exp(-ki)$$

ただし、k は定数.

具体的には以下のようにする↓

$$
\begin{aligned}
\xi'(i) &:= s(n-i+1)\xi(i) \\
P'(i,j) &:= s(j-i+1)P(i,j) \\
M'(i,j) &:= s(j-i+1)M(p,i,j) \\
\bar{\xi}'(i) &:= s(i-1)\bar{\xi}(i) \\
\bar{P}'(i,j) &:= s(i-1+n-j)\bar{P}(i,j) \\
\bar{M}'(p,i,j) &:= s(i-1+n-j)\bar{M}(p,i,j)
\end{aligned}
$$

実際に、上述の漸化式に対して $\bar{P}(i,j) = \bar{P}'(i,j)/s(i-1+n-j)$ のような代入を行うことで、scaled table だけを用いて漸化式をかき直すことができる。

書き直したものは以下になる↓

scaled 外側アルゴリズム:

$$
\begin{aligned}
\bar{\xi}'(i) &= s(1)\bar{\xi}'(i-1) + \sum_{\substack{j \\ j<i-1}} \bar{\xi}'(j)P'(j,i-1) \\
\bar{P}'(h,l) &:= \bar{\xi}'(h)\xi'(l+1) + \sum_{\substack{i,j \\ i<h,l<j \\ h-i-1+j-l-1\leq 30}} B(f_2(i,j,h,l,OMM))\bar{P}'(i,j)s(h-i+j-l) \\
&\quad + B(M_c+M_p)\sum_{\substack{i \\ i<h}} \begin{bmatrix} s(1)M'(1,i+1,h-1)\bar{P}'_{m1}(i,l) \\ +\{s(1)M'(1,i+1,h-1) + s(h-i)B(M_u(h-i-1))\}\bar{P}'_m(i,l) \end{bmatrix} \\
\bar{P}'_m(i,l) &:= \sum_{\substack{j \\ l<j}} s(1)\bar{P}'(i,j)M'(1,l+1,j-1) \\
\bar{P}'_{m1}(i,l) &:= \sum_{\substack{j \\ l<j}} s(j-l)\bar{P}'(i,j)B(M_u(j-l-1)) \\
\bar{M}'(2,h,l) &:= s(1)\bar{M}'(2,h-1,l)B(M_u) + s(2)\bar{P}(h-1,l+1)B(M_c+M_p) \\
\bar{M}'(1,h,l) &:= s(1)\bar{M}'(1,h-1,l)B(M_u) + \sum_{i<h-1} P'(i,h-1)B(M_p)\bar{M}'(2,i,l) \\
\bar{M}'(0,h,l) &:= s(1)\bar{M}'(0,h-1,l)B(M_u) + \sum_{i<h-1} P'(i,h-1)B(M_p)\left(\bar{M}'(0,i,l) + \bar{M}'(1,i,l)\right)
\end{aligned}
$$

このように、各更新式で $s(1)$ などが登場する。これらが scaling つき outside algorithm になっている。
forward でも同様の scaling がなされるべきなのだが、これは実は jax-rnafold ではすでに 実装されていて、default で同様の scaled forward algorithm が計算される。

というわけで現時点で outside algorithm の書き下しが完了した。
おそらく次は 要件定義＆実装、と進めるのが良い。

ベクトル化ロードマップ

同じ対角線 d にいる異なる (h,l) についての outside table の計算は互いに独立だから、ベクトル化の技術を使って gpu 上で高速に計算することができる。

イメージは以下になる↓

py

```
beta = 1 / 0.6 # 1 / kT
Lmax = 30
import numpy as np

def B(x):
    return np.exp(- beta * x)

def f_2(i, j, h, l, OMM):
    return 1.0 # Placeholder for the actual function

def fill_barP(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_Pm, bar_Pm1, bar_M, em_Mu, en_Mp, en_M
c, Lmax):
    # 対角線 d の全てを埋める
    return

def fill_barMm(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_M, em_Mu, en_Mp, en_Mc, Lmax):
    # 対角線 d の全てを埋める
    return

def outside(n, xi, P, OMM, M, en_Mu, en_Mp, en_Mc, Lmax):
    # init ...
    bar_xi = [0.0]*(n+1); bar_xi[n] = 1.0
    bar_P  = np.zeros((n+1, n+1))
    bar_M = np.zeros((3, n+1, n+1))  # bar_M[0]: M0, bar_M[1]: M1, bar_M[2]: M2

    # 集約テーブルの初期化
    bar_Pm  = np.zeros((n+1, n+1))
    bar_Pm1 = np.zeros((n+1, n+1))

    # (A) xi recursion
    for i in range(1, n + 1): # i = 1, 2, ..., n (0-origin)
        bar_xi[i] += xi[i-1] + sum([
            bar_xi[j] * P[j, i-1] for j in range(0, i-1) #j = 0, 1, ..., i - 2
        ]) # summation はベクトルを作って合計取る

    # (B) span-descending pass for bar_P and M_bar
    for d in range(n - 1, 0, -1): # jax.lax.scan などを使う
       # 計算グラフが大きくなりすぎるため、勾配チェックポインティングを使う
       fill_barP(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_Pm, bar_Pm1, bar_M, em_Mu, en_Mp, en
_Mc, Lmax)
       fill_barPm(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_Pm, bar_Pm1, bar_M, em_Mu, en_Mp, e
n_Mc, Lmax)
       fill_barPm1(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_Pm, bar_Pm1, bar_M, em_Mu, en_Mp,
en_Mc, Lmax)
       fill_barM(n, d, xi, P, OMM, M, bar_xi, bar_P, bar_Pm, bar_Pm1, bar_M, em_Mu, en_Mp, en
_Mc, Lmax)



    Z = xi[n]
    post = [[0.0]*(n+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(i+1, n+1):
            post[i][j] = (P[i][j] * bar_P[i][j]) / Z
    return post, bar_xi, bar_P, bar_M
```

実際は `get_outside_partitionfn` という、関数 `outside_partition_fn` を出力する関数を作成し、出力された関数に対して jax.jit をかませる。未来は勾配法の枠組みにおける、エネルギーパラメータの最適化や構造プロファイルに基づいた逆foldingに取り組む予定であるため、just in time compile によって繰り返し計算を最適化することは非常に役にたつ。
この辺の技術については jax-rnafold の実装を見ると極めて明瞭に書かれているため、そちらを参考にするのが良い。