Optimalisatie van het FlowSOM algoritme voor grote datasets

Ruben Vandamme

3e bachelor informatica

FlowSOM is een algoritme voor het clusteren en visualiseren van hoog-dimensionale cytometrische data, zoals die uit flowcytometrie. Het gebruikt Self-Organizing Maps (SOM) om de data te structureren en een minimal spanning tree (MST) om de relaties tussen clusters te visualiseren, waardoor patronen in complexe biologische datasets duidelijker zichtbaar worden. Naarmate deze datasets steeds groter worden, groeit de behoefte aan schaalbare en efficiënte implementaties [1]. In deze paper worden twee van zulke geoptimaliseerde implementaties gepresenteerd.

Inhoudsopgave

1 Introductie

Deze paper beschrijft twee alternatieve implementaties van de Python FlowSOM-bibliotheek van Saeyslab [2], die de prestaties verbeteren terwijl de nauwkeurigheid behouden blijft. De eerste versie optimaliseert het online SOM-algoritme [3] door kleine verbeteringen in een JIT-gecompileerde omgeving toe te passen. De tweede versie implementeert het BatchSOM-algoritme [4] met behulp van de XPySOM-bibliotheek [5]. Onderzoek toont aan dat deze bibliotheek veelbelovend is, doordat de fundamentele beperkingen van het online algoritme gedeeltelijk konden worden overwonnen met het parallelle BatchSOM, wat leidde tot een versnelling tot wel 125x. De codesegmenten zijn terug te vinden in 1 en 2.

2 Methoden

Als eerste stap werd onderzocht waar FlowSOM de meeste tijd besteedt. Uit een cProfile-analyse bleek dat de bottleneck zich bevindt in de SOM-functie in _som.py. De bespreking die volgt zal zich daarom richten op de optimalisatie van deze SOM-functie.

2.1 Online SOM

Online SOM is een conceptueel eenvoudige methode voor het implementeren van een SOM [3]. Tijdens het leerproces worden de gewichten van de codevectoren iteratief aangepast om clusters te vormen die de dataset representeren. Op deze manier wordt hoog-dimensionale data geprojecteerd op een tweedimensionaal grid. De besproken verbeteringen zijn ook toepasbaar in andere vergelijkbare algoritmen.

2.1.1 Geslaagde verbeteringen

- Python zelf is zeer flexibel, maar kan bij numerieke berekeningen traag zijn vanwege zijn interpretatieve aard. Numba is een krachtige tool voor het versnellen van deze numerieke berekeningen door gebruik te maken van Just-In-Time (JIT) compilatie. JIT compilatie wordt reeds gebruikt in de originele implementatie en is de voornaamste bron van optimalisatie. Er zijn echter enkele technieken die de prestaties van JIT verder kunnen verbeteren:
  - De optie **fastmath=True** schakelt wiskundige benaderingen in die de snelheid verhogen, maar de nauwkeurigheid iets verminderen. Dit is vooral nuttig bij het berekenen van de Euclidische afstand, een veelvoorkomende berekening om de dichtstbijzijnde codevector ten opzichte van een gekozen datapunt te bepalen. Hoewel het belangrijk is om een goede codevector te kiezen, zijn kleine afwijkingen geen groot probleem.
  - De code is al parallel geoptimaliseerd door de optie parallel=True. De toevoeging van **nogil=True** bouwt hierop voort door de Python Global Interpreter Lock (GIL) op te heffen tijdens de uitvoering. Hierdoor kunnen meerdere threads effectief onafhankelijk van elkaar draaien, zonder dat ze door de GIL worden gehinderd. Dit resulteert vooral bij numerieke bewerkingen, zoals in de SOM functie, in prestatiewinst.

- Voor het aanmaken van een lege array is **numpy.empty** sneller dan numpy.zeros [6]. Dit komt omdat numpy.empty alleen geheugen toewijst zonder de elementen te initialiseren. De array zal direct gevuld zijn met willekeurige waarden die al in het toegewezen geheugen aanwezig waren. Bij numpy.zeros worden daarentegen alle elementen expliciet op nul gezet, wat extra rekentijd kost. Voor het aanmaken van de xdists-array waar de waarden toch direct worden overschreven, gebruiken we dus beter numpy.empty (lijn 15).

- Bij de berekening van de **Euclidische afstand** is de vierkantswortel de laatste stap om de werkelijke afstand tussen twee punten te verkrijgen (lijn 7). Wanneer we echter alleen de relatieve afstanden tussen verschillende punten willen vergelijken, is het **nemen van de vierkantswortel overbodig**. Dit komt omdat de vierkantswortel een monotone functie is: als de kwadratische afstanden tussen punten a, b, x en y voldoen aan de ongelijkheid $d_{ab}^2 < d_{xy}^2$, dan geldt ook $d_{ab} < d_{xy}$ na het nemen van de vierkantswortel. Het vergelijken van de kwadraten van de afstanden behoudt dus dezelfde rangorde als de werkelijke Euclidische afstanden.

- De learning rate wordt berekend aan de hand van

$$alpha = alphas[0] - (alphas[0] - alphas[1]) * k/niter \qquad (lijn...).$$

  Dit is echter een relatief zware berekening om te doen in een lus. Een betere manier is om gebruik te maken van het feit dat de learning rate met een constante waarde verminderd tussen een vast maximum en minimum. We kunnen deze constante reeds vooraf berekenen (**precomputing**) en vervolgens het maximum in de lus met deze constante laten verminderen; een simpele aftrekking.

- Stel $rlen$ het aantal iteraties en n het aantal datapunten. Voor elke iteratie wordt er gekeken naar n random datapunten. Dit wordt samengebracht in één lus. Elke iteratie gebeurt echter ook een controle of het SOM nog wel veranderd, en zo niet, kan de training vroeger stoppen. Er is dus een probleem met het gebruik van maar één lus: om te weten wanneer een nieuwe iteratie begint, wordt elke keer '$if\ k\ \%\ n == 0$' uitgevoerd. Dit is niet logisch en bovendien ook trager. **Splits de lus dus in de twee lussen** die het eigenlijk is (lijn 25-26).

2.1.2 Mislukte pogingen

- De originele implementatie maakt uitbundig gebruik van for-loops. We kunnen deze loops vervangen door **vector-operaties**. De derde for-loop (lijn 39) past de gewichten van de codevectoren aan op basis van hun nabijheid tot de dichtstbijzijnde codevector. We kunnen een mask nemen van de codevectoren die zich moeten aanpassen en deze gebruiken om een loop te vectoriseren. Helaas bleek dit trager dan voordien. De reden is dat de SOM-functie JIT gecompiled wordt door numba, en numba erg goed overweg kan met loops. Gebruiken we geen JIT, dan is de vector-operatie wel sneller.

- Er was een idee voorgesteld om een KDtree te gebruiken om de de dichtste codevector te vinden nabij een datapunt [7]. Een KDtree partitioneert langs de dimensies en organiseert de datapunten hiërarchisch. Het opzoeken van de dichtstbijzijnde codevector kan hierdoor in tijdscomplexiteit O(log n). Het opstellen van de KDtree gebeurt in O(n log n). Het

probleem is nu dat de codevectoren in elke iteratie geüpdatet moeten worden en dus de KDtree opnieuw moet opgesteld worden. De versnelling is dus eigenlijk een vertraging aangezien de originele tijdscomplexiteit voor het opzoeken O(n) is.

2.2 Batch SOM

BatchSOM is een variant van het traditionele online SOM die de codevectoren niet na elke individuele toewijzing van een datapunt, maar na het verwerken van een hele batch van datapunten bijwerkt [4]. In deze aanpak worden de afstanden tussen alle datapunten in de batch en de codevectoren parallel berekend. Vervolgens wordt voor elke codevector de gemiddelde verandering berekend op basis van alle datapunten in de batch. Deze gemiddelde veranderingen worden in één enkele update-stap toegepast, wat de stabiliteit van de training vergroot en de prestaties versnelt. Het parallel berekenen van afstanden en updates maakt BatchSOM erg efficiënt voor grote datasets.

Ik heb ervoor gekozen om XPySom te integreren in het FlowSOM-project [5]. XPySom is een BatchSOM implementatie gebaseerd op MiniSom en is ontworpen voor parallelisatie en hoge snelheidsprestaties op één enkele node. Dit wordt bereikt door een efficiënt ontwerp van de gegevensverwerkingsoperaties, waarbij de interactie met Python tot een minimum wordt beperkt. Het is 2 tot 3 grootteordes sneller dan andere populaire bibliotheken zoals Somoclu en TFSOM, terwijl het een vergelijkbare nauwkeurigheid behoudt. Standaard maakt XPySOM gebruik van Numpy, maar GPU-versnelling is eenvoudig mogelijk met CuPy. Bovendien gebruikt XPySom vergelijkbare parameters als de online SOM (lijn 2-10).

3 Resultaten

In de eerste grafiek (1) zijn de drie implementaties te zien: de originele FlowSOM (blauw), de geoptimaliseerde FlowSOM (rood) en de batch FlowSOM (groen) implementatie. De batch FlowSOM maakt géén gebruik van GPU-acceleratie. De x-as geeft steeds grotere datasets weer. Deze werden gesampled uit de 100 eerste files van de FR-FCM-ZYX9 dataset. De groottes zijn gelijk gekozen aan de benchmarks uit de originele FlowSOM paper. Op de y-as staat de uitvoeringstijd in seconden. De geoptimaliseerde versie is 5-15% sneller dan de originele versie. De eerder besproken verbeteringen hebben dus weinig tijdswinst opgeleverd. De batch versie daarentegen doet het beter. Het is 2 tot 3 keer sneller dan de originele versie. De uitvoering vond plaats op een lokale computer, waarvan de specificaties zijn weergegeven in tabel 3.

Een tweede grafiek (2) illustreert het verschil in uitvoeringstijd tussen het gebruik van Numpy (blauw) en CuPy (rood) voor batch FlowSOM. De x-as toont opnieuw de steeds grotere datasets, terwijl de y-as de uitvoeringstijd in seconden weergeeft. Hier is het verschil wel direct duidelijk. Er is een versnelling van ongeveer 50x. Dit betekent dat de batch FlowSOM met CuPy ruwweg een versnelling heeft van 125x ten opzichte van de originele online FlowSOM implementatie. Hiermee presteert het beter dan de andere besproken versies. Aangezien mijn computer geen compatibele GPU had, werd één node van de joltik cluster van de UGent HPC gebruikt. Specificaties hiervan staan vermeld in tabel 3.

Uiteindelijk toont tabel 1 ook aan dat de nauwkeurigheid voor de alternatieve implementaties

behouden blijft: de v_measure blijft ongeveer gelijk voor enkele gelabelde referentie-datatsets uit FR-FCM-ZZPH [8]. Ook het maximale geheugengebruik blijft zo goed als gelijk (tabel 2).

4 Conclusie

Er werden twee alternatieve implementaties gegeven van de FlowSOM bibliotheek door Saeyslab, waarbij de uitvoeringstijd geoptimaliseerd werd. Een analyse van het FlowSOM algoritme toonde aan dat het merendeel van de tijd gespendeerd werd aan het opbouwen van een SOM. De batch SOM implementatie aan de hand van XPySOM en CuPy gaf de grootste versnelling: ongeveer 125x. Ondanks de snelheidswinst, had ik echter meer verwacht. De XPySOM paper beloofde namelijk een versnelling van 2 à 3 grootteordes ten opzichte van andere batch implementaties. Deze batch implementaties zijn vermoedelijk veel sneller dan hun online versies, waardoor een nog grotere prestatieverbetering logisch zou zijn. Het suggereert dat er mogelijk andere factoren of beperkingen zijn die de verwachte versnelling beïnvloeden. Verder onderzoek naar deze factoren is dus nodig.

Referenties

[1] K. Quintelier, A. Couckuyt, A. Emmaneel, J. Aerts, Y. Saeys en S. Van Gassen, "Analyzing high-dimensional cytometry data using FlowSOM," *Nat Protoc*, jrg. 16, nr. 8, p. 3775–3801, aug 2021, Epub 2021 Jun 25. DOI: 10.1038/s41596-021-00550-0.

[2] A. Couckuyt, B. Rombaut, Y. Saeys en S. Van Gassen, "Efficient cytometry analysis with FlowSOM in Python boosts interoperability with other single-cell tools," *Bioinformatics*, jrg. 40, nr. 4, btae179, apr 2024, ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae179. eprint: https://academic.oup.com/bioinformatics/article-pdf/40/4/btae179/57341836/btae179.pdf. adres: https://doi.org/10.1093/bioinformatics/btae179.

[3] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, jrg. 78, nr. 9, p. 1464–1480, 1990. DOI: 10.1109/5.58325.

[4] T. Kohonen, E. Oja, O. Simula, A. Visa en J. Kangas, "Engineering applications of the self-organizing map," *Proceedings of the IEEE*, jrg. 84, nr. 10, p. 1358–1384, 1996. DOI: 10.1109/5.537105.

[5] R. Mancini, A. Ritacco, G. Lanciano en T. Cucinotta, "XPySom: High-Performance Self-Organizing Maps," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, p. 209–216. DOI: 10.1109/SBAC-PAD49847.2020.00037.

[6] J. Brownlee, *Benchmark Fastest Way To Create NumPy Array*, https://superfastpython.com/benchmark-fastest-way-to-create-numpy-array/, Accessed: 26 August 2024, nov 2023.

[7] M. Skrodzki, "The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time," mrt 2019.

[8] L. M. Weber en M. D. Robinson, "Comparison of clustering methods for high-dimensional single-cell flow and mass cytometry data," *Cytometry A*, jrg. 89, nr. 12, p. 1084–1096, dec 2016, Epub 2016 Dec 19. DOI: 10.1002/cyto.a.23030.

```python
@jit(nopython=True, parallel=False, fastmath=True, nogil=True)
def eucl(p1, p2) -> float:
    distance = 0.0
    for j in range(len(p1)):
        diff = p1[j] - p2[j]
        distance += diff * diff
    return distance

@jit(nopython=True, parallel=True, nogil=True)
def OPTIMIZED_SOM(
    data, codes, nhbrdist, alphas, radii, ncodes, rlen=10, distf=eucl, seed=None
) -> np.ndarray:
    if seed is not None:
        np.random.seed(seed)
    xdists = np.empty(ncodes)
    n = data.shape[0]
    px = data.shape[1]
    niter = rlen * n
    threshold = radii[0]
    thresholdStep = (radii[0] - radii[1]) / niter
    change = 1.0
    alpha = alphas[0]
    alpha_sub = (alphas[0] - alphas[1]) / niter

    for _ in range(rlen):
        for _ in range(n):
            i = np.random.randint(n)

            nearest = 0
            for cd in range(ncodes):
                xdists[cd] = distf(data[i, :], codes[cd, :])
                if xdists[cd] < xdists[nearest]:
                    nearest = cd

            if threshold < 1.0:
                threshold = 0.5
            alpha -= alpha_sub

            for cd in range(ncodes):
                if nhbrdist[cd, nearest] > threshold:
                    continue

                for j in range(px):
                    tmp = data[i, j] - codes[cd, j]
                    change += abs(tmp)
                    codes[cd, j] += tmp * alpha

            threshold -= thresholdStep

        if change < 1:
            break
        change = 0.0
    return codes
```

Listing 1: Geoptimaliseerde SOM implementatie

```python
def BATCH_SOM(
    data: np.ndarray,
    xdim: int = 10,
    ydim: int = 10,
    topology: str = "rectangular",
    nhbrdist: str = "gaussian",
    alphas: tuple[float, float] = (0.05, 0.01),
    rlen: int = 10,
    distf: str = "euclidean",
    seed: Optional[int] = None,
) -> np.ndarray:
    som = XPySom(
        n_parallel=0,
        x=xdim,
        y=ydim,
        input_len=data.shape[1],
        learning_rate=alphas[0],
        learning_rateN=alphas[1],
        neighborhood_function=nhbrdist,
        topology=topology,
        activation_distance=distf,
        compact_support=True,
        random_seed=seed,
    )
    som.train(data=data, num_epochs=rlen)
    nodes = som._weights
    return np.reshape(nodes, (nodes.shape[0]*nodes.shape[1], nodes.shape
    [2]))
```

Listing 2: BatchSOM implementatie

dataset	originele SOM	geoptimaliseerde SOM	BatchSOM
FlowCAP_ND	0.44	0.42	0.40
Levine_13dim	0.86	0.86	0.84
Levine_32dim	0.84	0.80	0.86
Samusik_01	0.89	0.92	0.89
Samusik_all	0.83	0.82	0.83

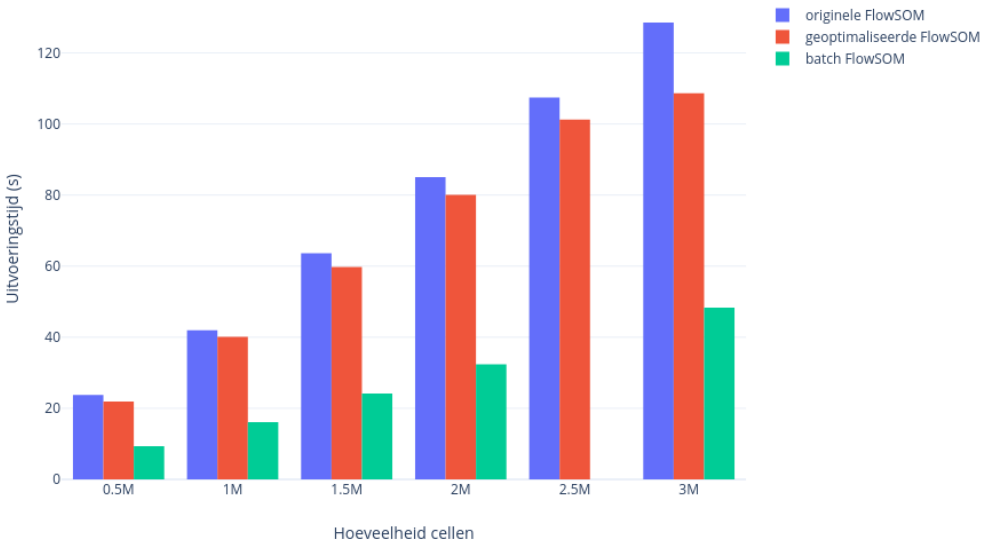
Tabel 1: Vergelijking van de v_measure scores van enkele referentie-datasets voor de verschillende implementaties.

SOM	geheugen (MiB)
origineel	220.7
geoptimaliseerd	221.3
batch	221.9

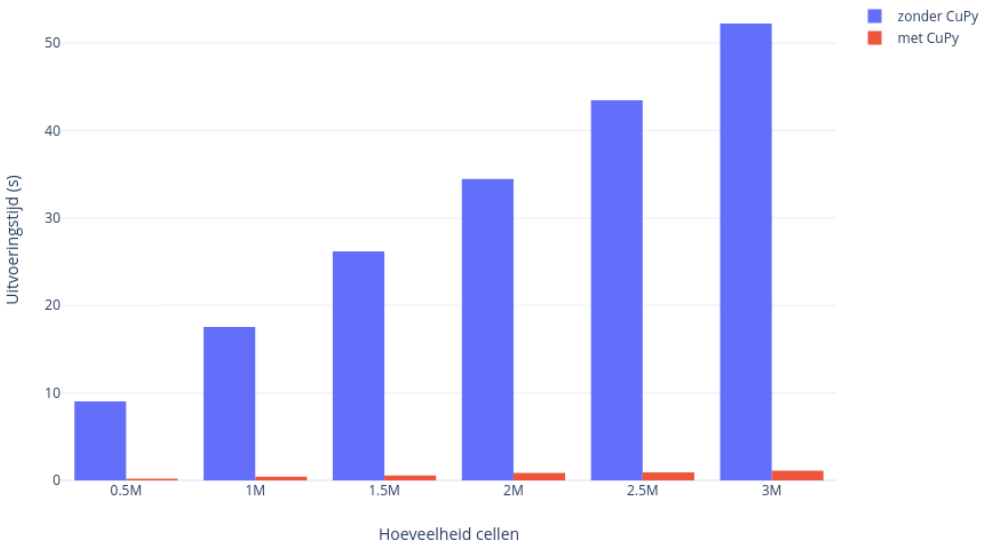
Tabel 2: Maximale geheugenverbruik voor de verschillende implementaties bij het verwerken van een dataset met 100000 cellen.

— computer specificaties —	
OS	Microsoft Windows 11 Home 10.0.22631
Processor	AMD Ryzen 7 7735HS, 3.2 GHz, 8 Cores
RAM	16 gb, DDR5
Python versie	Python 3.12.3
— HPC specificaties —	
Cluster name	joltik
OS	RHEL 8
Processor	Intel Xeon Gold 6242, 2.8GHz, 16 Cores
GPUs	4x NVIDIA Volta V100 GPUs (32GB GPU memory)
geheugen	256 GiB
Python versie	Python 3.11.3

Tabel 3: Hard- en software specificaties



Figuur 1: Vergelijking van de uitvoeringstijd van de verschillende implementaties bij steeds grotere datasets.



Figuur 2: Vergelijking van de uitvoeringstijd van de BatchSOM implementatie, zonder en met gebruik van CuPy.