Day 2

UMAP vs t-SNE: what's new?

Step	UMAP (essence)	t-SNE (essence)	Shared idea
1 Neighbourhood	*k*-NN with NN-Descent (fast, approx.)	Exact or approx. pairwise distances	Build local neighbour graph
2 High-dim weights	Adaptive exponential kernel $w_{ij} = \exp[-\max(0, d_{ij} - \varrho_i) / \sigma_i]$ Choose σ_i so $\Sigma_j w_{ij} = \log_2 k$. *$\varrho_i$ = distance from point i to its closest neighbour*	Gaussian kernel with σ_i set so entropy of $P_{j\|i}$ equals log(perplexity)	Density-adaptive similarities
3 Symmetrisation	$w_{ij} + w_{ji} - w_{ij} w_{ji}$	$(P_{j\|i} + P_{i\|j}) / (2 N)$	Undirected affinity matrix
4 Low-dim model & loss	Heavy-tailed $1 / (1 + a d^{2b})$; cross-entropy + negative sampling	Student-t $1 / (1 + d^2)$; KL divergence	Match two similarity dists. with heavy tails
5 Optimisation	SGD + negative sampling; **O(N k)** per epoch	GD + Barnes-Hut / FFT repulsion; **O(N log N)** per iter	Iterative neighbour pull + cheap repulsion
6 Init / tuning	Spectral embedding; *min_dist* tunes cluster tightness	PCA + early exaggeration	Quick linear init → nonlinear refine

Python coding style - PEP8

Write readable code, avoid bugs, and make collaboration easier (you will review each other's code)

Readings:

PEP8 - Official style guide for python code: https://peps.python.org/pep-0008/
Google python style guide: https://google.github.io/styleguide/pyguide.html

Give your variables meaningful and understandable names

How to start coding first:

Use a formatter (autopep8, yapf, black)

```
pip install black
black mycode.py
```

Use a linter (e.g. pylint)

Python coding style - PEP20

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Implementation-level efficiency of python scientific software

Both Algorithm and Implementation level difference can leads to >1000x performance difference. So you have to pay attention to both.

Matrix multiplication of two 4096x4096 matrices

Pure Python

55432s (extrapolated)

Numpy Default BLAS/ATLAS (multi-core)

2.79s

Numpy Intel MKL (multi-core)

265 ms.

CUDA (CUBLAS) with PyTorch

21.7ms

Can you do it get faster than CUBLAS?

Use fast linear algebra routine whenever possible

What if no existing implementation available?

(Ranked from easy to hard, lower to higher performance improvement potential if used correctly)

Numba JIT

Write Cython code which compiles to C

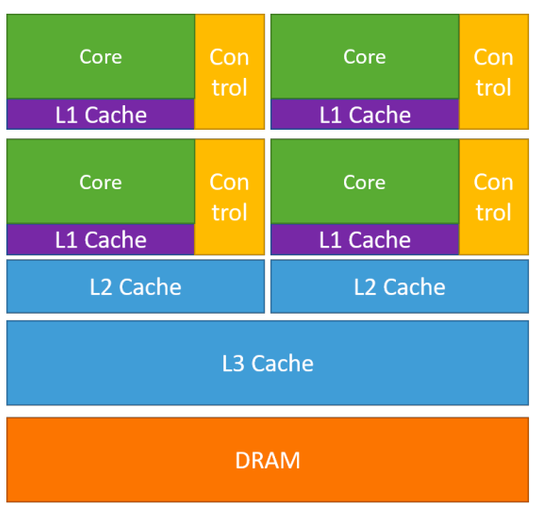
Implement with C/C++ and create pybind11 binding

Low-level optimizations for speed

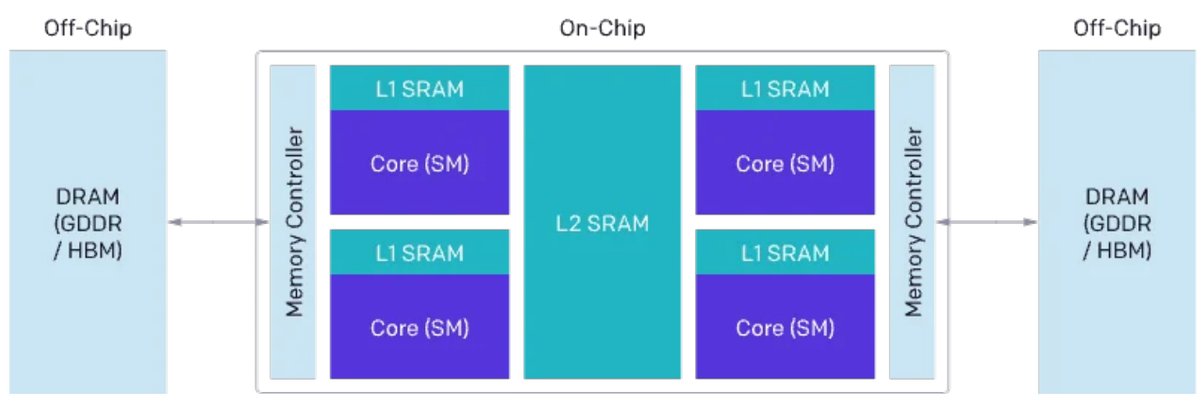
Built-in parallelism in modern CPU: Single Instruction Multiple Data (SIMD)

Microarchitecture	ISA	throughput (per clock)	SIMD width	max flops/cycle
Nehalem	SSE	1 add + 1 mul	4	8
Sandy Bridge /Ivy Bridge	AVX	1 add + 1 mul	8	16
Haswell	AVX2	2 fmadds	8	32
Knights Corner	AVX-512F	1 fmadd(∗)	16	32
Knights Landing	AVX-512F	2 fmadds	16	32

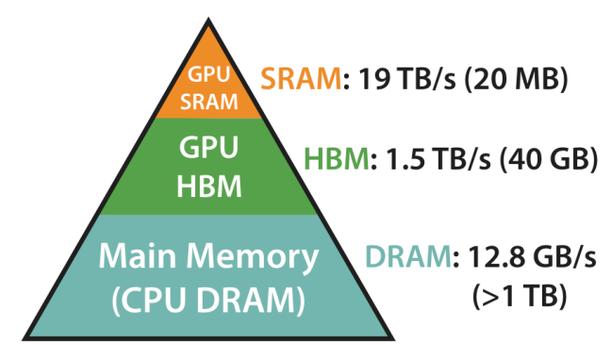
Cache efficiency: communication bandwidth



CPU



GPU



Memory Hierarchy with Bandwidth & Memory Size

The order of accessing data matters (improving spatially locality reduces cache miss)

Communication cost for multicore, multi-CPU, or multi-node computation

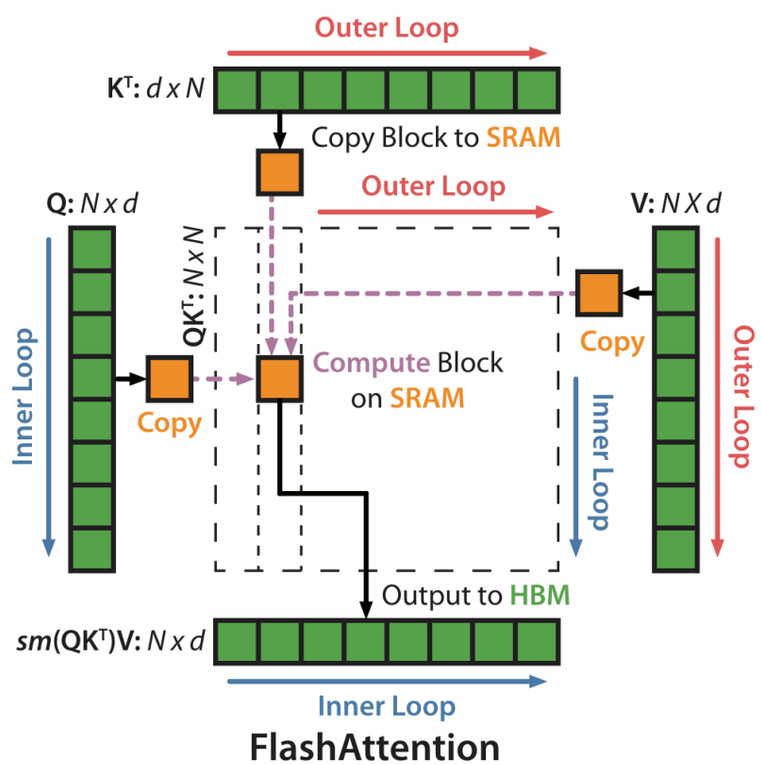
https://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/parallel_distributed/2016/slides/pdf/simd.pdf
https://teivah.medium.com/go-and-cpu-caches-af5d32cc5592
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
https://arxiv.org/pdf/2205.14135

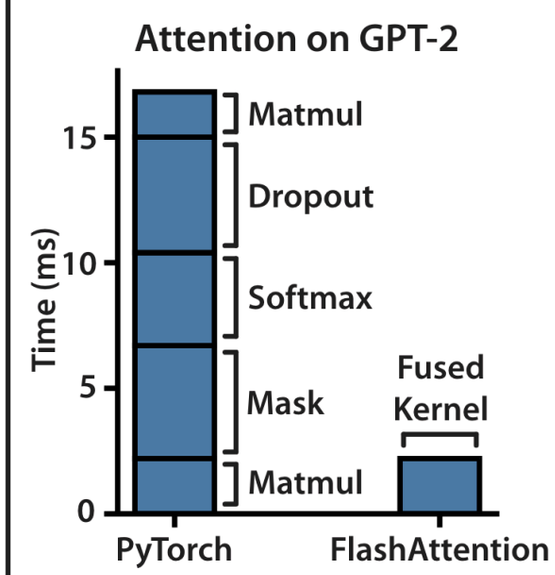
Higher-level GPU-programming than CUDA

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

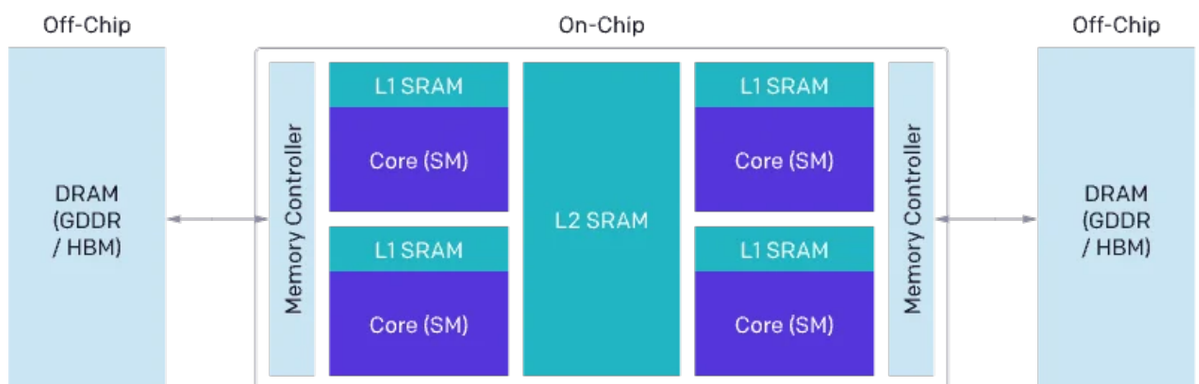
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.





Higher-level GPU-programming than CUDA

Triton: CUDA programming at Stream Multiprocessor / Block-level



	CUDA	TRITON
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Within SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual

```python
@triton.jit
def softmax(Y, stride_ym, stride_yn, X, stride_xm,
stride_xn, M, N):
    # row index
    m = tl.program_id(0)
    # col indices
    # this specific kernel only works for matrices that
    # have less than BLOCK_SIZE columns
    BLOCK_SIZE = 1024
    n = tl.arange(0, BLOCK_SIZE)
    # the memory address of all the elements
    # that we want to load can be computed as follows
    X = X + m * stride_xm + n * stride_xn
    # load input data; pad out-of-bounds elements with 0
    x = tl.load(X, mask=n < N, other=-float('inf'))
    # compute numerically-stable softmax
    z = x - tl.max(x, axis=0)
    num = tl.exp(z)
    denom = tl.sum(num, axis=0)
    y = num / denom
    # write back to Y
    Y = Y + m * stride_ym + n * stride_yn
    tl.store(Y, y, mask=n < N)
```

Higher-level GPU-programming than CUDA

```python
@triton.jit
def _attn_fwd_inner(acc, l_i, m_i, q,  #
                    K_block_ptr, V_block_ptr,  #
                    start_m, qk_scale,  #
                    BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr, BLOCK_N: tl.constexpr,  #
                    STAGE: tl.constexpr, offs_m: tl.constexpr, offs_n: tl.constexpr,  #
                    N_CTX: tl.constexpr, fp8_v: tl.constexpr):
    # range of values handled by this stage
    if STAGE == 1:
        lo, hi = 0, start_m * BLOCK_M
    elif STAGE == 2:
        lo, hi = start_m * BLOCK_M, (start_m + 1) * BLOCK_M
        lo = tl.multiple_of(lo, BLOCK_M)
    # causal = False
    else:
        lo, hi = 0, N_CTX
    K_block_ptr = tl.advance(K_block_ptr, (0, lo))
    V_block_ptr = tl.advance(V_block_ptr, (lo, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        start_n = tl.multiple_of(start_n, BLOCK_N)
        # -- compute qk ----
        k = tl.load(K_block_ptr)
        qk = tl.dot(q, k)
        if STAGE == 2:
            mask = offs_m[:, None] >= (start_n + offs_n[None, :])
            qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            qk -= m_ij[:, None]
        else:
            m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
            qk = qk * qk_scale - m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)
        # -- update m_i and l_i
        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        acc = acc * alpha[:, None]
        # update acc
        v = tl.load(V_block_ptr)
        if fp8_v:
            p = p.to(tl.float8e5)
        else:
            p = p.to(tl.float16)
        acc = tl.dot(p, v, acc)
        # update m_i and l_i
        m_i = m_ij
        V_block_ptr = tl.advance(V_block_ptr, (BLOCK_N, 0))
        K_block_ptr = tl.advance(K_block_ptr, (0, BLOCK_N))
    return acc, l_i, m_i
```

https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html

Fast algorithm design!

Fast algorithms:

Barnes-Hut t-SNE

Fast interpolation-based t-SNE

Randomized SVD

Basic ideas:

Nearest neighbors

Random projection

FFT

Coarse graining

Tree

Computational complexity of common operations

Vector element wise product

Vector dot product (vector -> scalar)

Matrix multiplication (two n x n matrices):

In practice, O(n^3) and driven by implementation efficiency

year	algorithm	order of growth
?	brute force	N^{3}
1969	Strassen	$N^{2.808}$
1978	Pan	$N^{2.796}$
1979	Bini	$N^{2.780}$
1981	Schönhage	$N^{2.522}$
1982	Romani	$N^{2.517}$
1982	Coppersmith-Winograd	$N^{2.496}$
1986	Strassen	$N^{2.479}$
1989	Coppersmith-Winograd	$N^{2.376}$
2010	Strother	$N^{2.3737}$
2011	Williams	$N^{2.3727}$
?	?	$N^{2+\varepsilon}$

number of floating-point operations to multiply two N-by-N matrices

Computational complexity of common operations

Vector element wise product

Vector dot product (vector -> scalar)

Matrix multiplication (two n x n matrices):

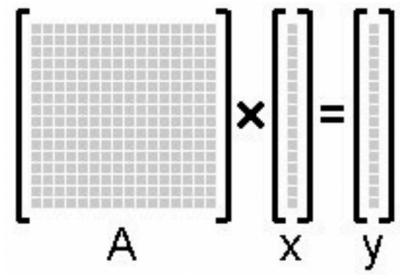
Matrix-vector multiplication

year	algorithm	order of growth
?	brute force	N^3
1969	Strassen	$N^{2.808}$
1978	Pan	$N^{2.796}$
1979	Bini	$N^{2.780}$
1981	Schönhage	$N^{2.522}$
1982	Romani	$N^{2.517}$
1982	Coppersmith-Winograd	$N^{2.496}$
1986	Strassen	$N^{2.479}$
1989	Coppersmith-Winograd	$N^{2.376}$
2010	Strother	$N^{2.3737}$
2011	Williams	$N^{2.3727}$
?	?	$N^{2+\varepsilon}$

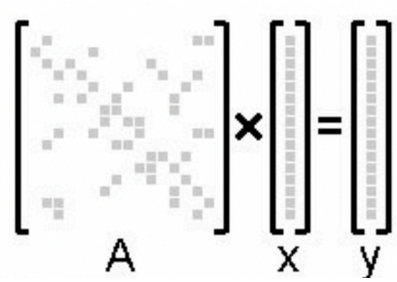
number of floating-point operations to multiply two N-by-N matrices

Structure in the data can be exploited for faster computation

Dense Matrix-vector Multiplication : O(n^2)



Sparse Matrix-vector Multiplication: O(m)

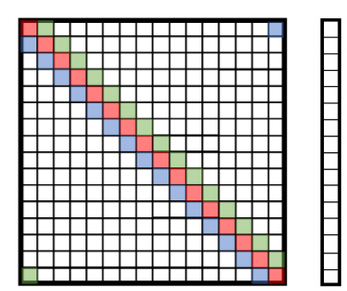


(Structured sparse patterns can be exploited, e.g. block diagonal, banded)

Circulant Matrix-vector Multiplication: O(n log n)

$$\begin{bmatrix} a_0 & a_{-1} & a_{-2} & a_{-3} \\ a_1 & a_0 & a_{-1} & a_{-2} \\ a_2 & a_1 & a_0 & a_{-1} \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \quad \begin{bmatrix} 7 & 11 & 5 & 6 \\ 3 & 7 & 11 & 5 \\ 8 & 3 & 7 & 11 \\ 1 & 8 & 3 & 7 \end{bmatrix}$$

Convolution can be viewed as circulant matrix vector multiplication



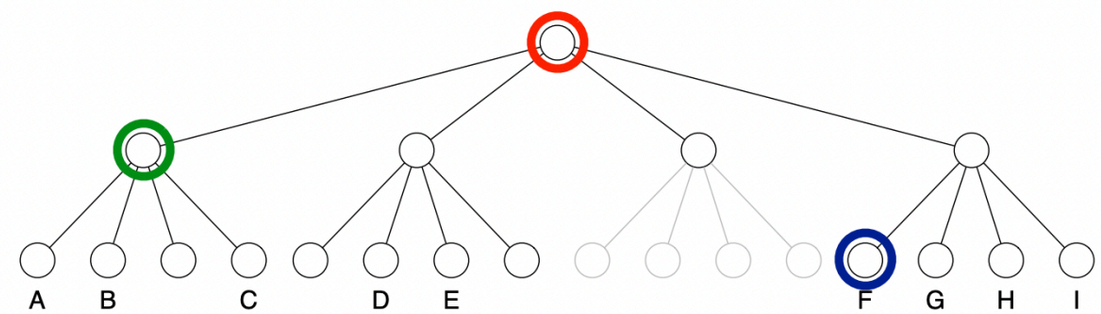
Barnes-Hut t-SNE

O(N log N).

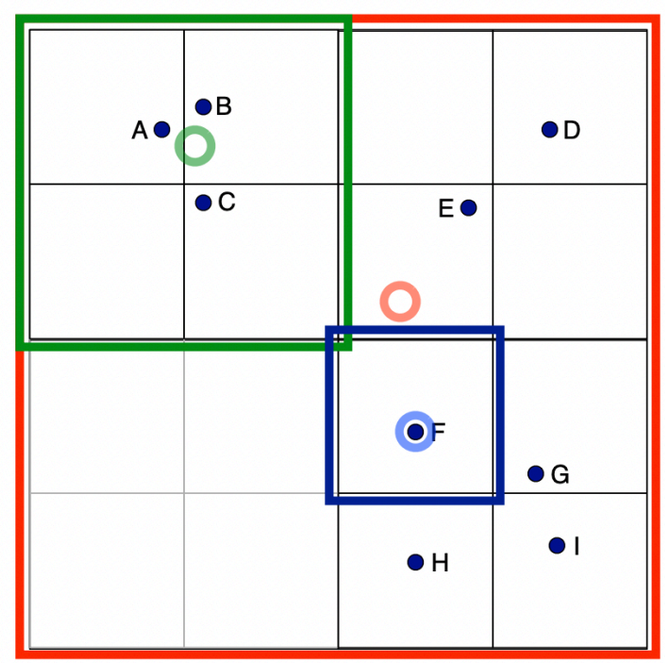
The objective can be viewed as pairwise attractive and repulsive forces between all cells

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4(F_{attr} + F_{rep}) = 4\left(\sum_{j \neq i} p_{ij} q_{ij} Z(\mathbf{y}_i - \mathbf{y}_j) - \sum_{j \neq i} q_{ij}^2 Z(\mathbf{y}_i - \mathbf{y}_j)\right)$$

Borrow ideas from N-body simulation!



If points are sufficiently far away,
it is enough to approximate with average of a cell



Accelerating t-SNE using Tree-Based Algorithms, 2014

Fast interpolation-based t-SNE

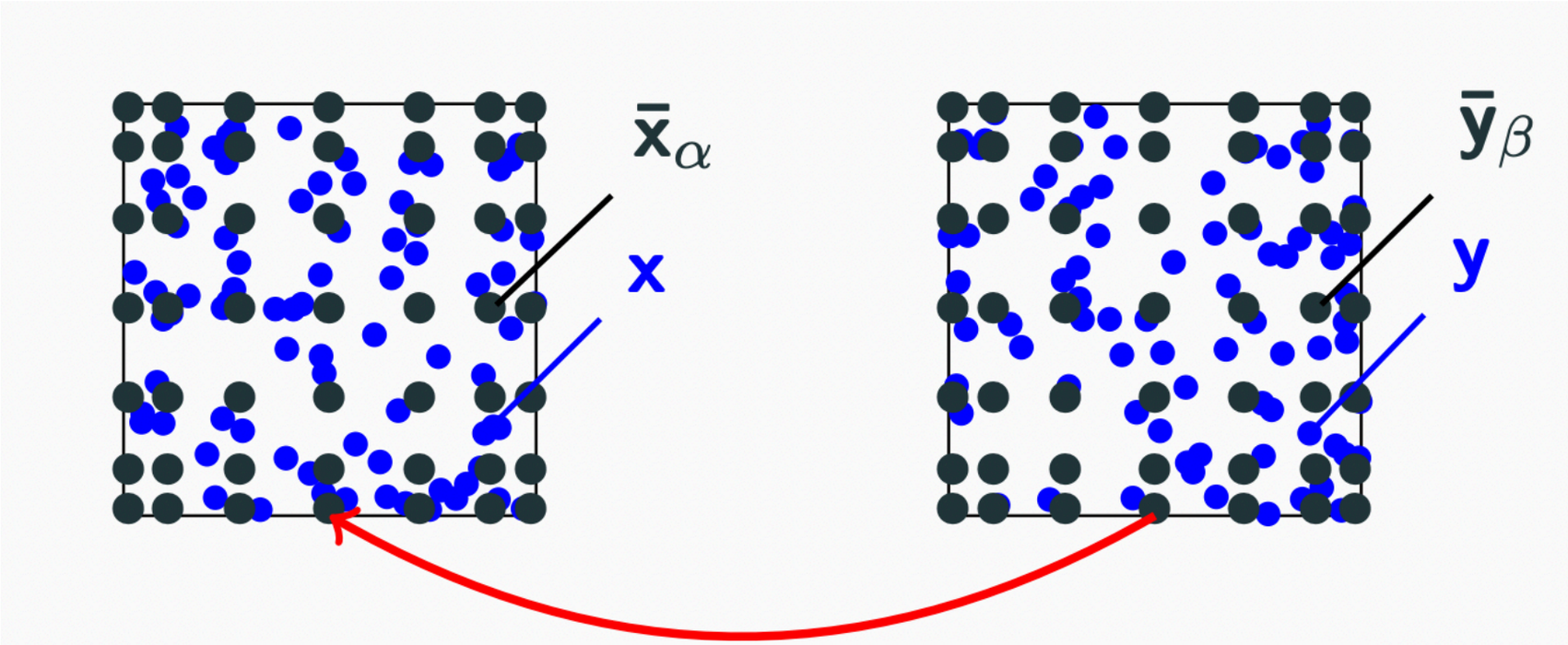
O(pN+p log p).

The objective can be viewed as pairwise attractive and repulsive forces between all cells

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4(F_{attr} + F_{rep}) = 4\left(\sum_{j \neq i} p_{ij} q_{ij} Z(\mathbf{y}_i - \mathbf{y}_j) - \sum_{j \neq i} q_{ij}^2 Z(\mathbf{y}_i - \mathbf{y}_j)\right)$$

Borrow ideas from N-body simulation!

Approximate pairwise forces with interpolant with fixed spacing (fast FFT-based acceleration available)



$$\tilde{\mathbf{k}}(\mathbf{x}, \mathbf{y}) = \sum_{|\alpha| \leq p} S(\bar{\mathbf{x}}_\alpha, \mathbf{x}) \sum_{|\beta| \leq p} \mathbf{k}(\bar{\mathbf{x}}_\alpha, \bar{\mathbf{y}}_\beta) S(\bar{\mathbf{y}}_\beta, \mathbf{y})$$

O(p^2+p N).

Fast interpolation-based t-SNE for improved visualization of single-cell RNA-seq data

Fast interpolation-based t-SNE O(pN+p log p).

Further acceleration with FFT (avoiding the p^2 computation):

$\mathbf{k}(\bar{\mathbf{x}}_\alpha, \bar{\mathbf{y}}_\beta)$. between equidistant points can be embedded into a Toeplitz matrix (many relative distances are repeated)

Example Toeplitz matrix

$$\begin{bmatrix} a_0 & a_{-1} & a_{-2} & a_{-3} \\ a_1 & a_0 & a_{-1} & a_{-2} \\ a_2 & a_1 & a_0 & a_{-1} \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \qquad \begin{bmatrix} 7 & 11 & 5 & 6 \\ 3 & 7 & 11 & 5 \\ 8 & 3 & 7 & 11 \\ 1 & 8 & 3 & 7 \end{bmatrix}$$

Toeplitz matrix can be embedded into a circulant matrix like this

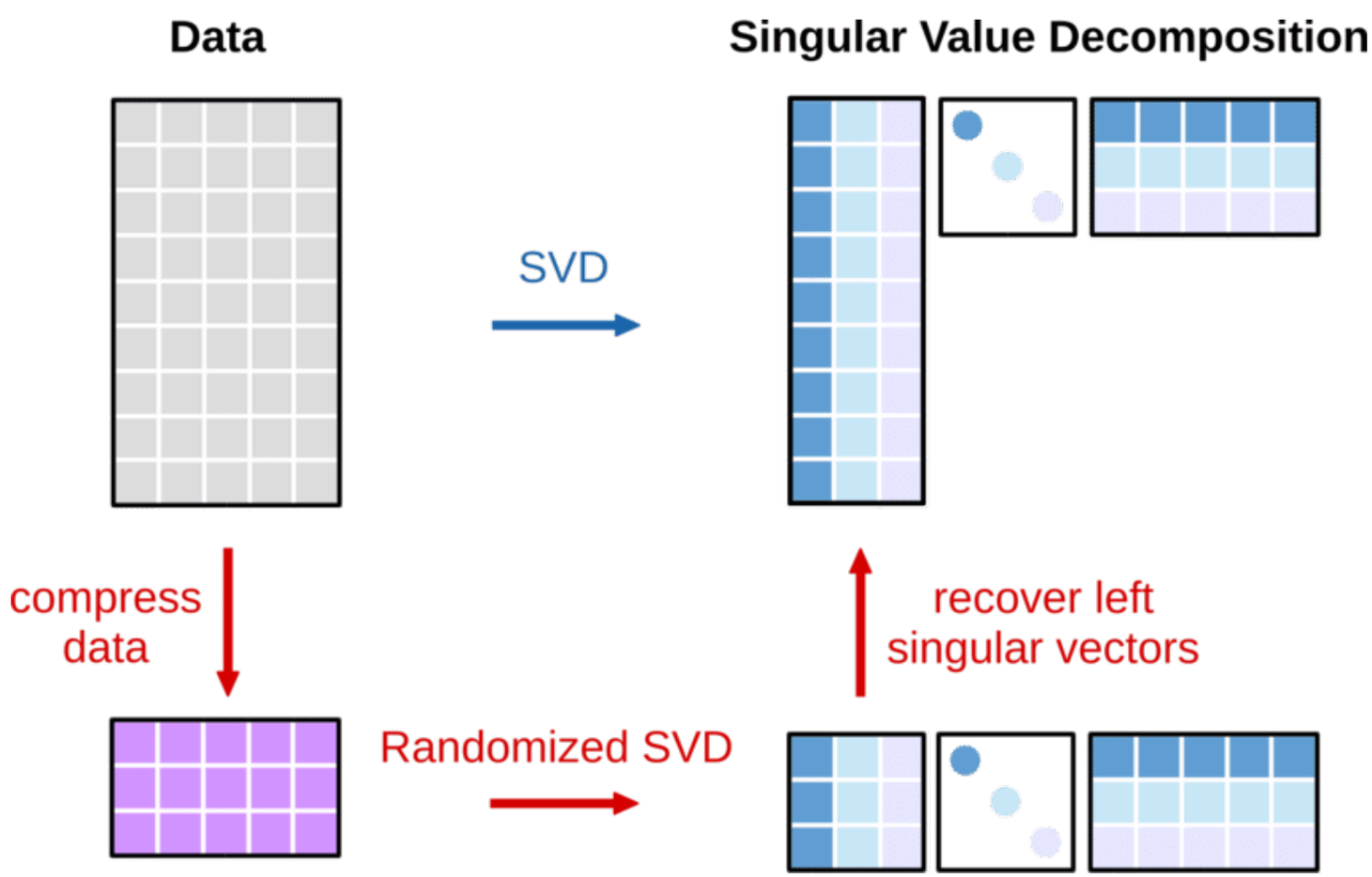
$$C_8 = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & a_{-3} & \mathbf{a_0} & \mathbf{a_3} & \mathbf{a_2} & \mathbf{a_1} \\ a_1 & a_0 & a_{-1} & a_{-2} & \mathbf{a_{-3}} & \mathbf{a_0} & \mathbf{a_3} & \mathbf{a_2} \\ a_2 & a_1 & a_0 & a_{-1} & \mathbf{a_{-2}} & \mathbf{a_{-3}} & \mathbf{a_0} & \mathbf{a_3} \\ a_3 & a_2 & a_1 & a_0 & \mathbf{a_{-1}} & \mathbf{a_{-2}} & \mathbf{a_{-3}} & \mathbf{a_0} \\ \mathbf{a_0} & \mathbf{a_3} & \mathbf{a_2} & \mathbf{a_1} & a_0 & a_{-1} & a_{-2} & a_{-3} \\ \mathbf{a_{-3}} & \mathbf{a_0} & \mathbf{a_3} & \mathbf{a_2} & a_1 & a_0 & a_{-1} & a_{-2} \\ \mathbf{a_{-2}} & \mathbf{a_{-3}} & \mathbf{a_0} & \mathbf{a_3} & a_2 & a_1 & a_0 & a_{-1} \\ \mathbf{a_{-1}} & \mathbf{a_{-2}} & \mathbf{a_{-3}} & \mathbf{a_0} & a_3 & a_2 & a_1 & a_0 \end{bmatrix}$$

Circulant matrix - vector multiplication can be accelerated with FFT in O(n log n) time instead of O(n^2)

$C_n x = \text{DFT}^{-1}(\text{DFT}(a_n) \odot \text{DFT}(x))$ Each DFT or inverse DFT step take n log n time

Fast algorithm design: Randomized algorithms

Singular value decomposition m-by-n matrix A, m>n $O(mn^2)$



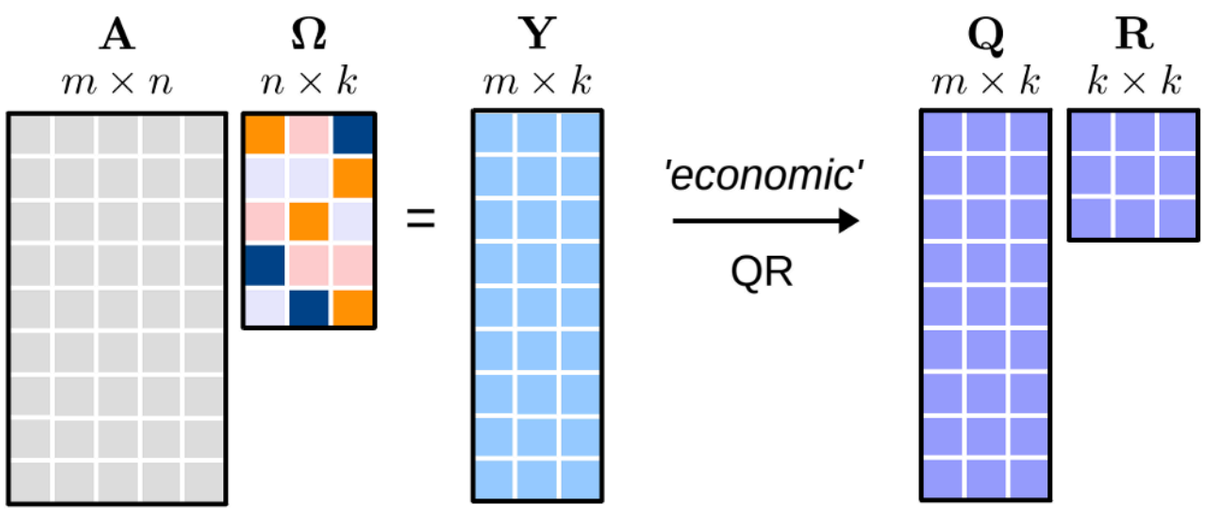
Fast algorithm design: Randomized algorithms

Randomized SVD (sketch)

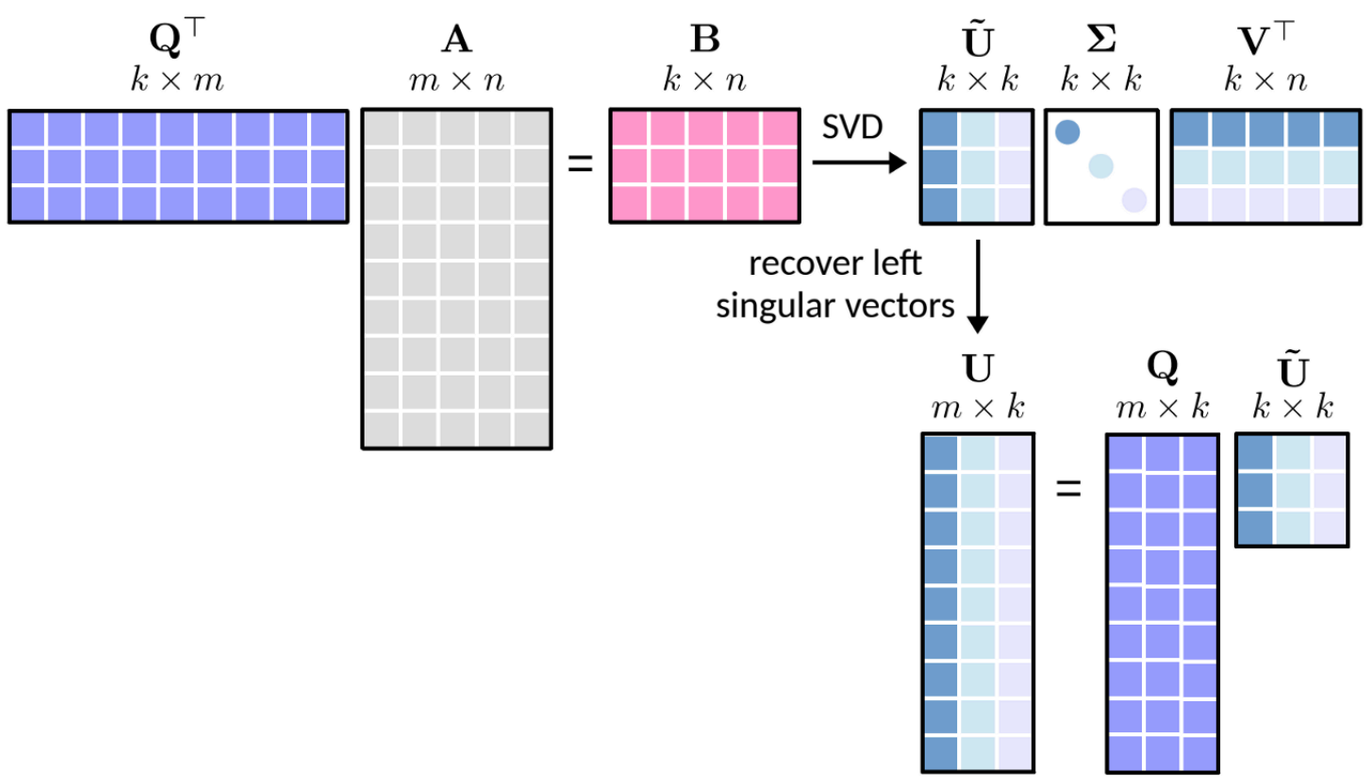
rank k decomposition for m-by-n matrix A, m>n

$$O(mnk + mk^2 + nk^2)$$

Step 1.



Step 2.



Why?

$$A \approx QQ^T A = Q(\tilde{U}\Sigma V^T)$$
$$\approx U\Sigma V^T$$

Reference: Halko, N., Martinsson, P.-G., & Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, *53*(2), 217–288.
Image source: N. Benjamin Erichson, Sergey Voronin, Steven L. Brunton, J. Nathan Kutz (2016). Randomized Matrix Decompositions Using R. arXiv:1608.02148

Computational complexity of common operations

Vector element wise product

Vector dot product (vector -> scalar)

Matrix multiplication (two n x n matrices):

Matrix-vector multiplication

Matrix inversion

Linear solve Ax = b

Sparse linear solve Ax = b, where A is sparse

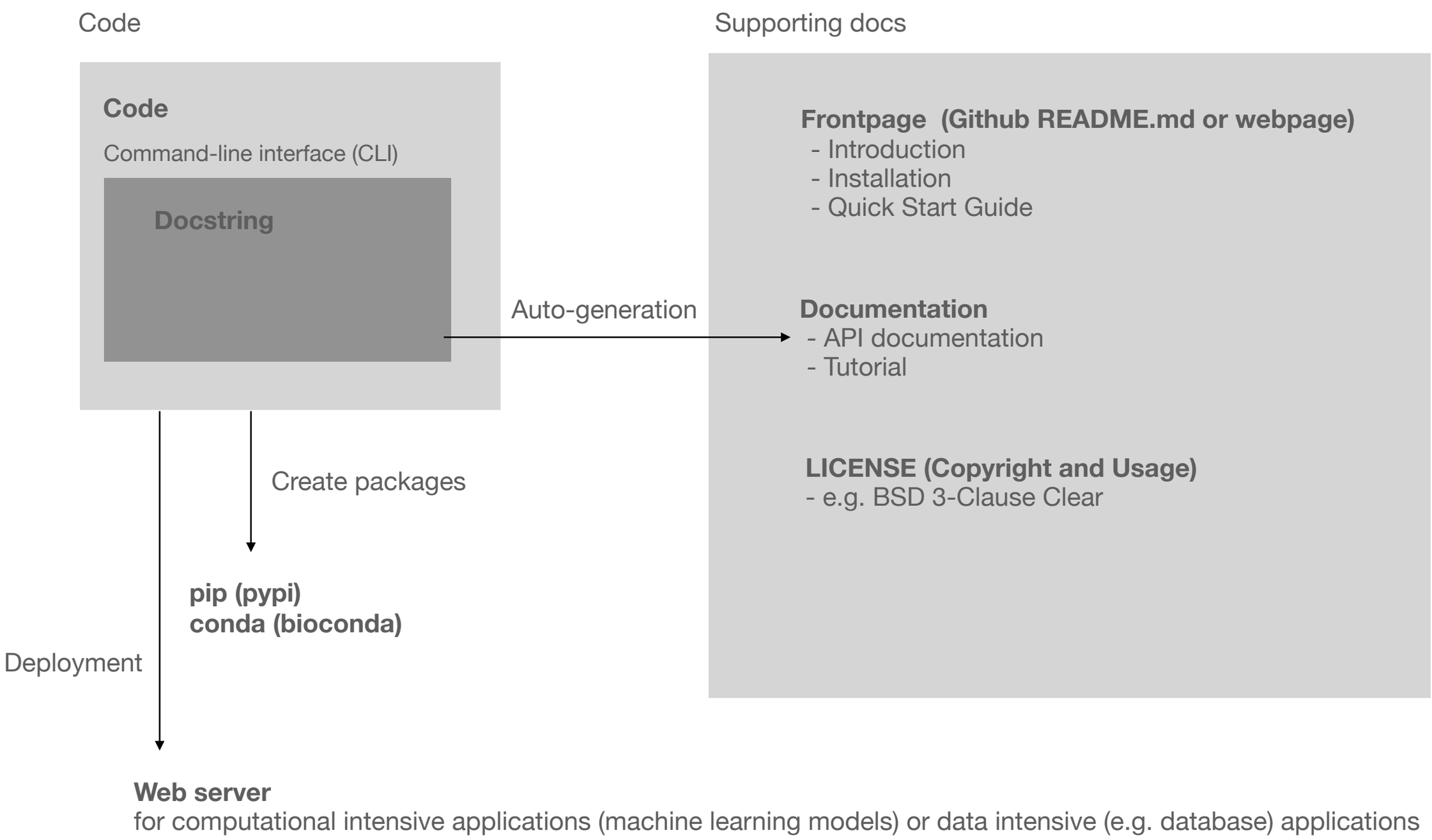
SVD

Nearest neighbor search

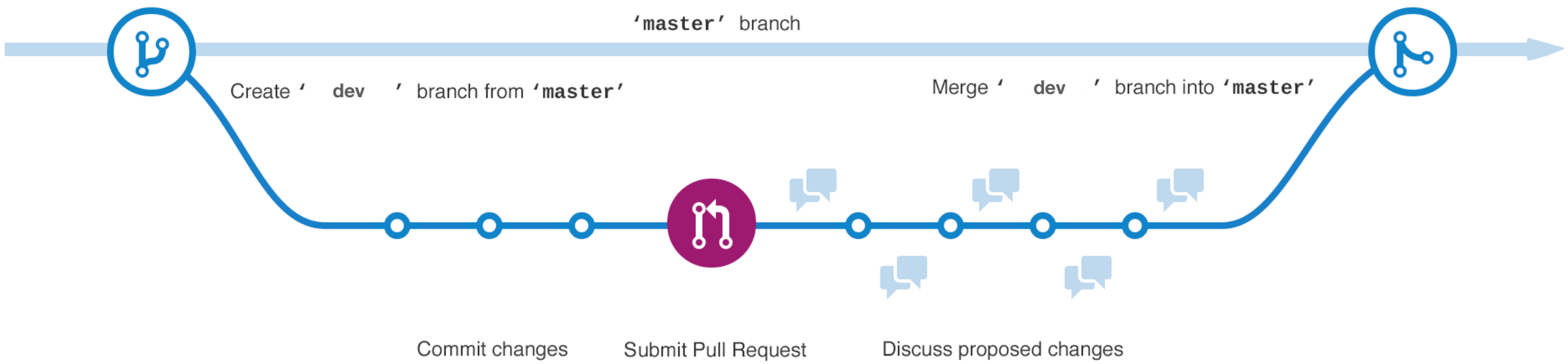
Practical note:

1. Utilize structure in the data (e.g. sparsity, Toeplitz) whenever possible
2. Avoid O(n^3) operations whenever possible (e.g. with randomized SVD)

Anatomy of python scientific software



Github workflow



Task for Day 2:

1. **Code review (set up Github, create dev branch and pull request)**
2. **Respond to code review (after approval by your reviewer, merge your code)**
3. **Write docstring. Optional: implement an CLI with capability to take input file and generate output file / plots; implement scikit-learn Estimator API**

Docopt : simplest way to make a CLI that parses command line input

```
"""Naval Fate.

Usage:
  naval_fate.py ship new <name>...
  naval_fate.py ship <name> move <x> <y> [--speed=<kn>]
  naval_fate.py ship shoot <x> <y>
  naval_fate.py mine (set|remove) <x> <y> [--moored | --drifting]
  naval_fate.py (-h | --help)
  naval_fate.py --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Moored (anchored) mine.
  --drifting    Drifting mine.

"""
from docopt import docopt


if __name__ == '__main__':
    arguments = docopt(__doc__, version='Naval Fate 2.0')
    print(arguments)
```

https://github.com/jazzband/docopt-ng

http://try.docopt.org/