University of Colorado **Boulder**

Randomization methods for big-data
(other than SGD)

Stephen Becker, CU Applied Math dept

Applied Math and Statistics colloquium, Colorado School of Mines

Fri Sept 19 2025

Joint work with several authors, mostly former PhD students:

(and colleagues at CU & the Colorado School of Mines)

Farhad Pourkamali-Anaraki
(Asst. Prof at UC Denver)

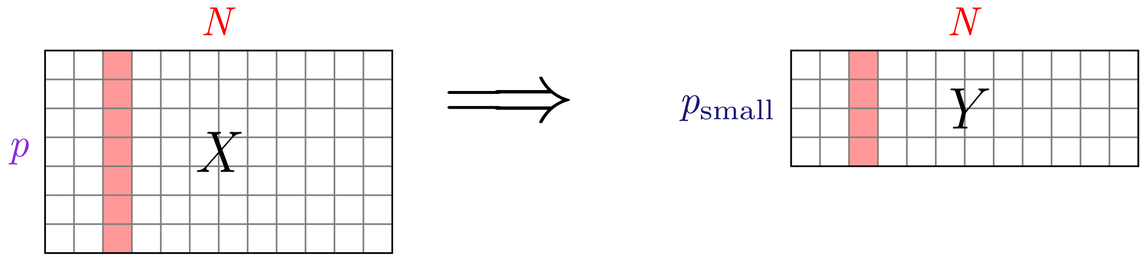
David Kozak
(private sector)

Osman Asif Malik
(was Alvarez postdoc at Berkeley Labs, now private sector)

OUTLINE

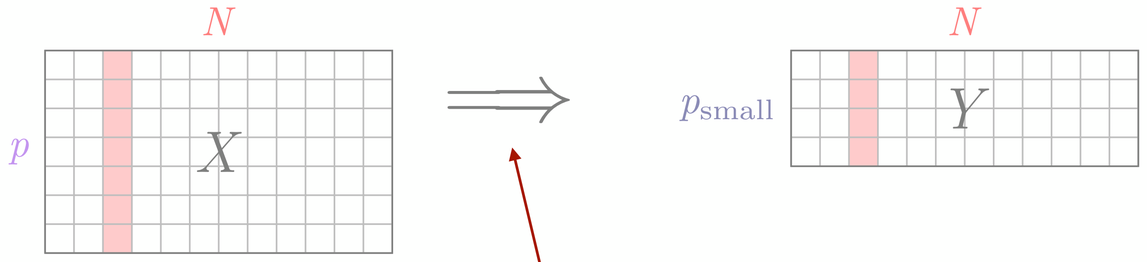
1. Randomized "sketches"
   a. Warmup: PCA
   b. Classical sketches
   c. Structured sketches
2. Applications
   a. Warmup: linear algebra
   b. K-means clustering
   c. Tensor factorizations
   d. Gradient-free optimization

Reducing the dimensionality of data



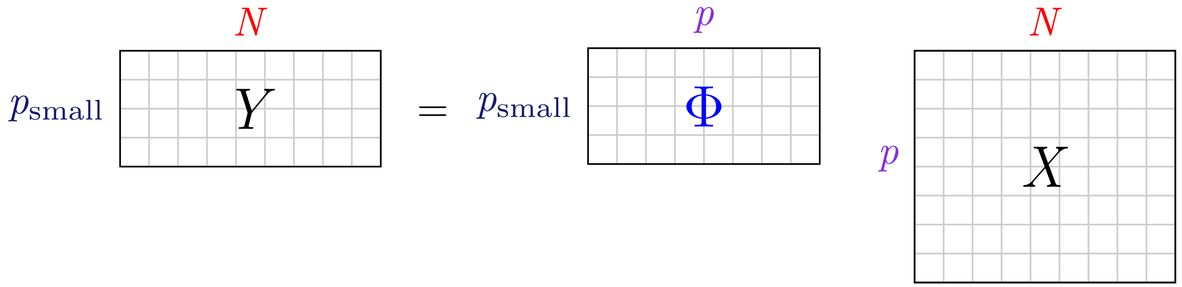
$x_i \in \mathbb{R}^p, \quad i = 1, \ldots, N$ is converted to $y_i \in \mathbb{R}^{p_{\text{small}}}, \quad i = 1, \ldots, N$

Reducing the dimensionality of data

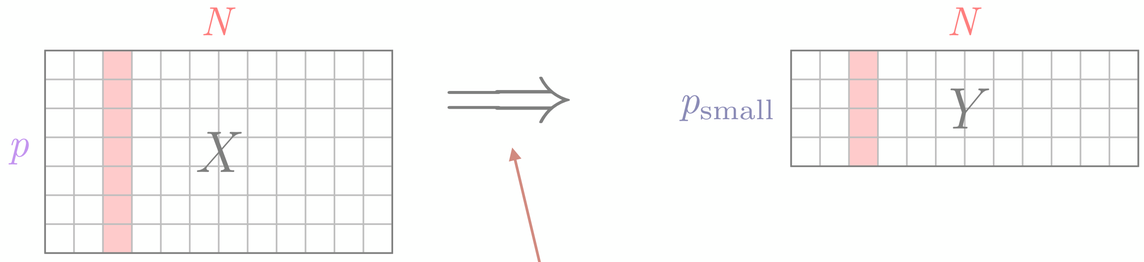


$x_i \in \mathbb{R}^p, \quad i = 1, \ldots, N$ is converted to $y_i \in \mathbb{R}^{p_{\text{small}}}, \quad i = 1, \ldots, N$

this reduction is often linear, in which case can write as

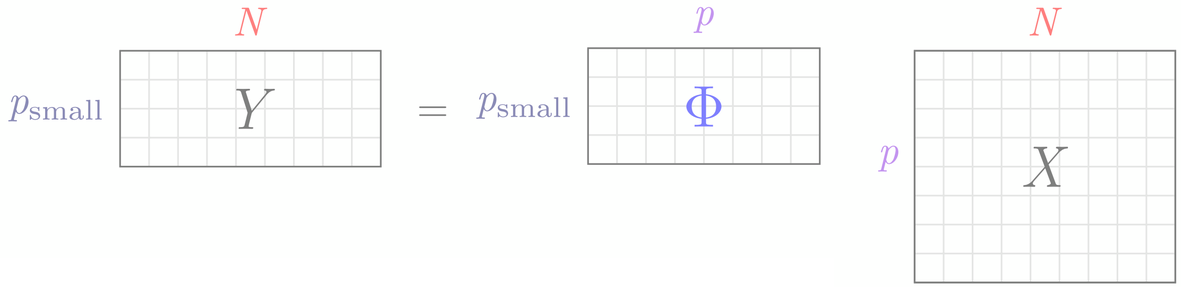


Reducing the dimensionality of data



$x_i \in \mathbb{R}^p, \quad i = 1, \ldots, N$ is converted to $y_i \in \mathbb{R}^{p_{\text{small}}}, \quad i = 1, \ldots, N$

this reduction is often linear, in which case can write as



Why?

- Faster computation
  - Especially if the complexity of subsequent processing is $\mathcal{O}(p^2 N)$
  - If $p_{\text{small}} = 0.05p$ then speedup is $400\times$
- Denoising (remove irrelevant components)
- Fewer degrees-of-freedom reduces chance of overfitting models
- Visual interpretation (e.g., $p_{\text{small}} = \{2, 3\}$) ⟵ This is "multidimensional scaling". Specialized techniques (tSNE, UMAP) are best

... or for storage reasons

(especially to create **one-pass methods**)

Method 1: PCA (non-linear)

Principal Component Analysis (aka Hotelling or Karhunen-Loeve transform)

Take SVD

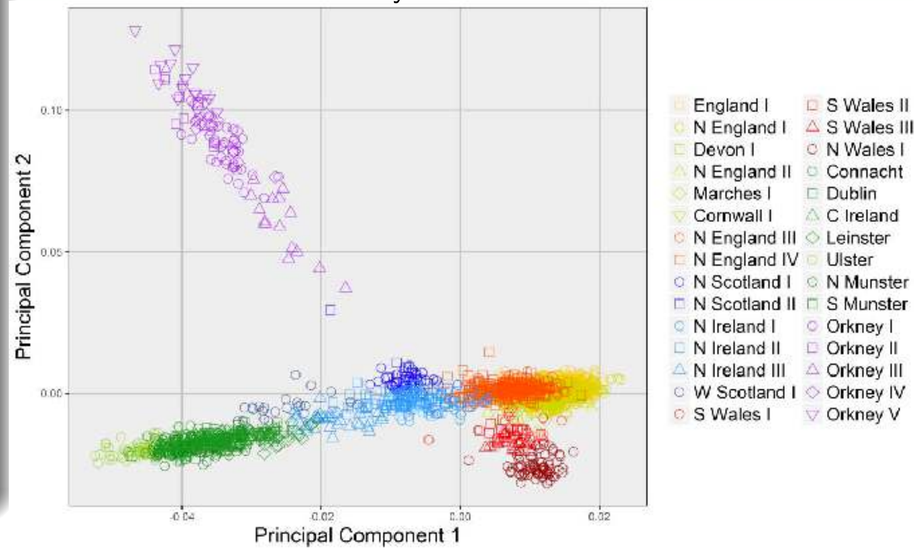
$$X = \begin{bmatrix} U_1 \, U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}$$

where $\|\Sigma_2\| \approx$ small. Then PCA takes the form of

$$Y \leftarrow \Phi_{\text{PCA}} \cdot X$$

where $\Phi_{\text{PCA}} = {U_1}^T$.

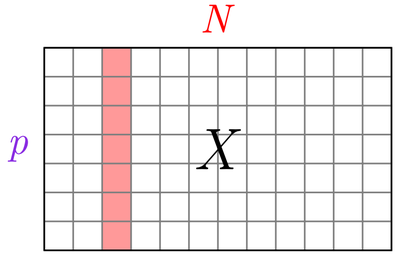
Principal component analysis of the fineStructure co-ancestry matrix



https://commons.wikimedia.org/wiki/File:PCA_of_British_people.png
Gilbert, E., O'Reilly, S., Merrigan, M. et al. The Irish DNA Atlas: Revealing Fine-Scale Population Structure and History within Ireland. Sci Rep 7, 17199 (2017)

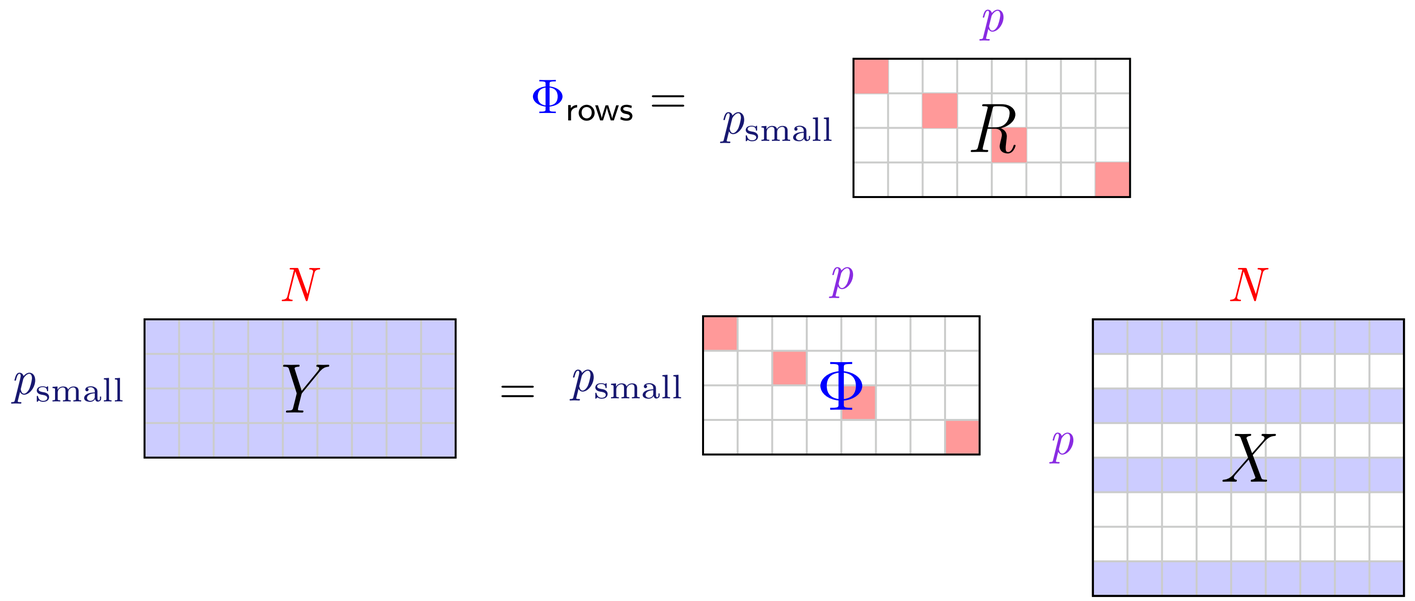
Note:

- Costly but linear in N: ($\mathcal{O}(p^2 N)$ direct, or about $\mathcal{O}(p_{\text{small}} p N)$ Krylov)
- Non-linear, since Φ_{PCA} is determined by X
- Need to process all of X before we can apply $y_i \leftarrow \Phi_{\text{PCA}} \cdot x_i$



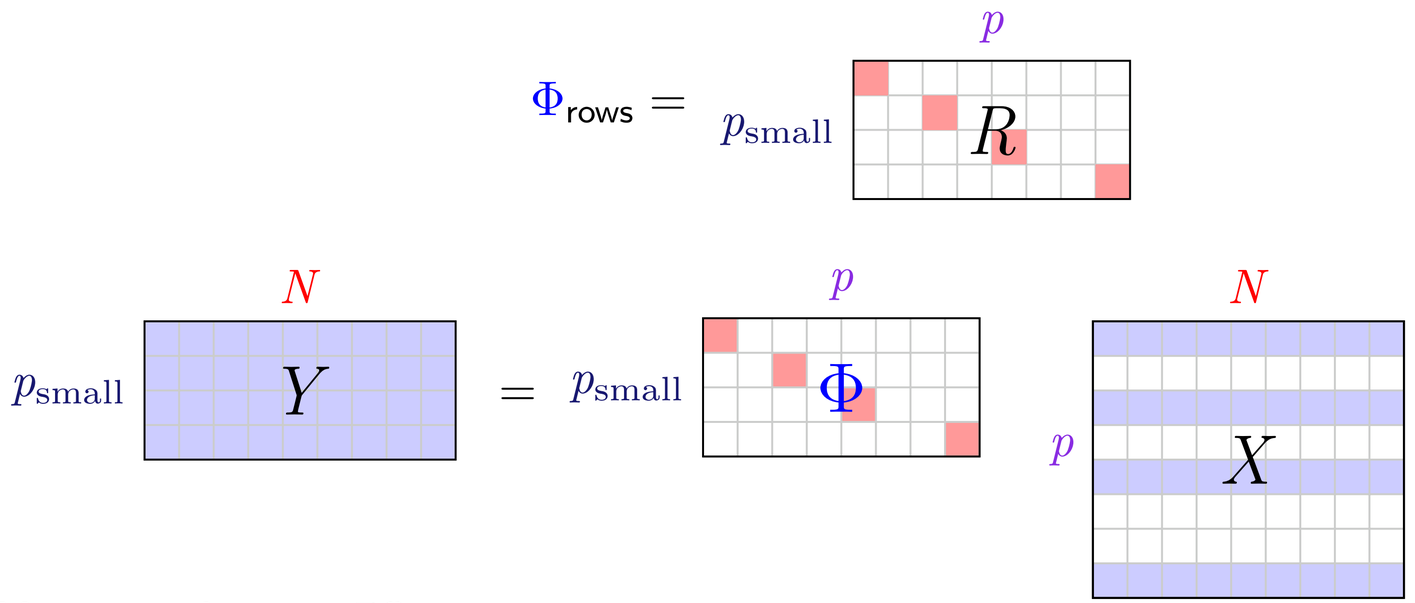
Method 2: sub-sample dimensions (i.e., rows)

R *randomly* samples rows



Method 2: sub-sample dimensions (i.e., rows)

R *randomly* samples rows



How to choose R?

- Uniformly at random (... poor performance) ← "oblivious"
- According to row-norm (... better, not great, and slower)
- According to *leverage scores* based on partial SVD (... better, slow) ← "non-oblivious" (like PCA)

One benefit: for many applications, this avoids computing all entries of the RHS in the first place!

That's sometimes the bottleneck...

Method 2: sub-sample dimensions (i.e., rows)

R *randomly* samples rows

p_{sma}

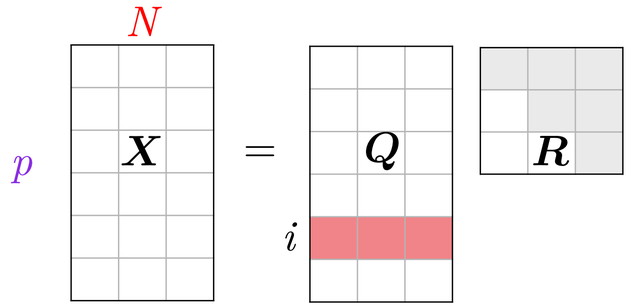
How

- U
- Ac
- According to *leverage scores* based on partial SVD (. . . better, slow)

"oblivious"

"non-oblivious"

Leverage scores



i^{th} leverage score is Euclidean norm squared of the i^{th} row of $\boldsymbol{Q}$

[For keen observers: this isn't useful if $N > p$. What to do then?
Can essentially prove separately for all pairs of points (so $N = 2$)
and then take union bound (not an issue, since failure decays fast).
Technically, usually prove via subspace embeddings]

Method 3: dense random matrix

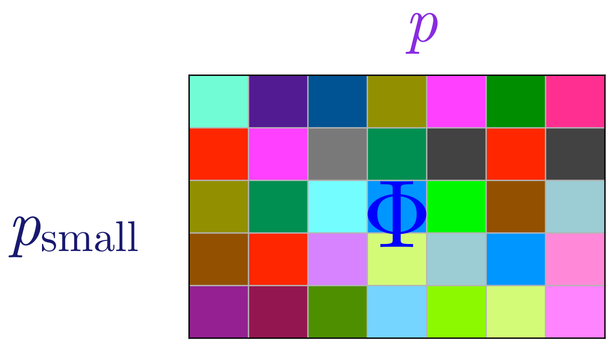
Theorem (Johnson-Lindenstrauss, 1984 (and Indyk-Motwani, 1998))

Choose $\Phi = \Phi_{randn}$ *with* $p_{small} \propto \varepsilon^{-2} \log N$ *iid rows each* $\mathcal{N}(0, p/p_{small})$*, then for all* $x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p$*,*

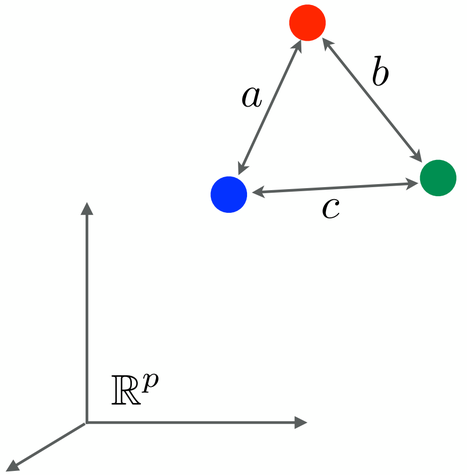
$$1 - \varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2}{\|x_i - x_j\|_2} \leq 1 + \varepsilon$$

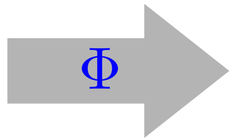
with constant probability.

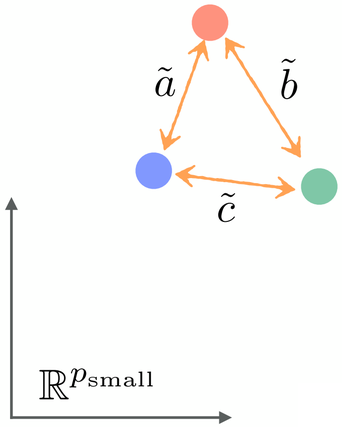
"gold standard" for accuracy among randomized data-oblivious sketches (for any Euclidean norm results)

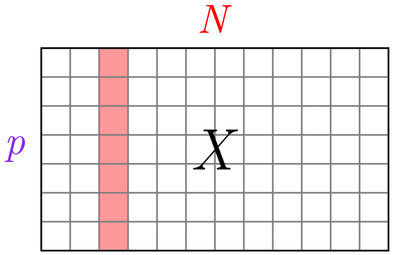


$\Phi_{ij} \sim \mathcal{N}(0, p/p_{\text{small}})$









Method 3: dense random matrix

Theorem (Johnson-Lindenstrauss, 1984 (and Indyk-Motwani, 1998))

Choose $\Phi = \Phi_{randn}$ *with* $p_{small} \propto \varepsilon^{-2} \log N$ *iid rows each* $\mathcal{N}(0, p/p_{small})$*, then for all* $x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p$*,*

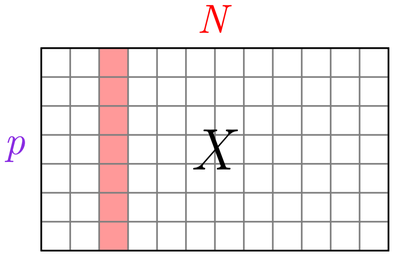
$$1 - \varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2}{\|x_i - x_j\|_2} \leq 1 + \varepsilon$$

with constant probability.

- Φ is linear and data-independent (unlike PCA)
- but Φ is not orthogonal (unlike PCA) (though variants can be made to be orthogonal)
- Independent of original dimension p
- Independent of conditioning of X (unlike PCA)
- Probabilistic (unlike PCA)

Usually not practical since too costly since computing ΦX is $\mathcal{O}(pp_{\text{small}}N)$

Depends on what its purpose is...



Method 3: dense random matrix

Theorem (Johnson-Lindenstrauss, 1984 (and Indyk-Motwani, 1998))

Choose $\Phi = \Phi_{randn}$ *with* $p_{small} \propto \varepsilon^{-2} \log N$ *iid rows each* $\mathcal{N}(0, p/p_{small})$*, then for all* $x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p$*,*

$$1 - \varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2}{\|x_i - x_j\|_2} \leq 1 + \varepsilon$$

with constant probability.

Bipartite Expander graphs

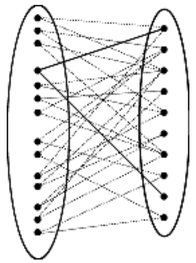


image: De Castro, IEEE TIT 2010

example

CountSketch

example

- Φ is linear and data-independent (unlike PCA)
- but Φ is not orthogonal (unlike PCA)
- Independent of original dimension p
- Independent of conditioning of X (unlike PCA)
- Probabilistic (unlike PCA)

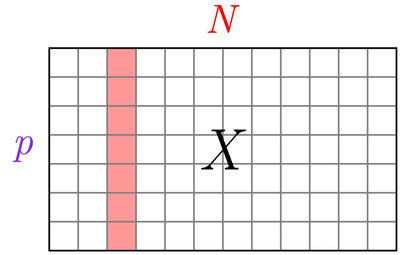
Usually not practical since too costly since computing ΦX is $\mathcal{O}(pp_{\text{small}}N)$

Extensions of **applicability**, for example:

- entire subspaces and manifolds (using covering-number arguments)

Many **extensions**, for example:

- sub-Gaussian entries, and/or dependent rows (e.g., columns uniform on sphere)
- dependent columns, e.g., Haar measure on orthogonal matrices
- sparse matrix (worse performance though)



Method 4: Fast Johnson-Lindenstrauss Transforms

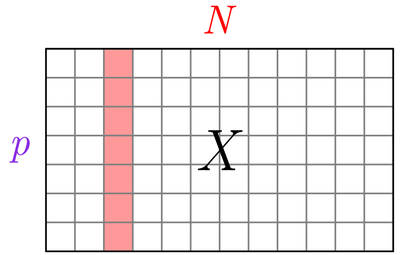
R samples rows; $\mathcal{F}$ Fourier-like; D diag. w/ random ± 1 entries ("Rademacher")

(uniformly)

$$\Phi_{\text{FFT}} = p_{\text{small}} \underbrace{\begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}}_{R} \; p \underbrace{\begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}}_{\mathcal{F}} \; p \underbrace{\begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}}_{D}$$

Orthogonal/unitary, ideally with small max entry

(so *identity matrix* is bad)



Method 4: Fast Johnson-Lindenstrauss Transforms

R samples rows; $\mathcal{F}$ Fourier-like; D diag. w/ random ± 1 entries ("Rademacher")
(uniformly)

$$\Phi_{\text{FFT}} = p_{\text{small}} \underbrace{\boxed{R}}_{p} \; p \underbrace{\boxed{\mathcal{F}}}_{p} \; p \underbrace{\boxed{D}}_{p}$$

Cost:

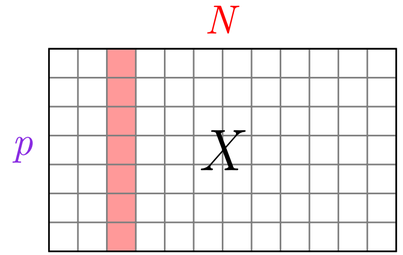
- $\mathcal{F}(x)$ costs $p \log p$ (vs p^2 naively)
- hence $\mathcal{O}(pN \log p)$ to compute ΦX

Guarantees: almost as good as classical Johnson-Lindenstrauss
First guarantees in the Fast Johnson-Lindenstrauss paper

Ailon and Chazelle, "Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform", STOC 2006

Think of $\mathcal{F}$ as FFT or DCT or Hadamard transform

also known under many names, e.g., Random Orthogonal System (**ROS**), FJLT



Method 4: Fast Johnson-Lindenstrauss Transforms

R samples rows; $\mathcal{F}$ Fourier-like; D diag. w/ random ± 1 entries ("Rademacher")

$$\Phi_{\text{FFT}} = \underset{p_{\text{small}} \times p}{R} \; \underset{p \times p}{\mathcal{F}} \; \underset{p \times p}{D}$$

Cost:

- $\mathcal{F}(x)$ costs $p \log p$
- hence $\mathcal{O}(pN \log p)$

Guarantees: almost as g

First guarantees in the Fast Johnson-Lindenstrauss paper

Ailon and Chazelle, "Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform", STOC 2006

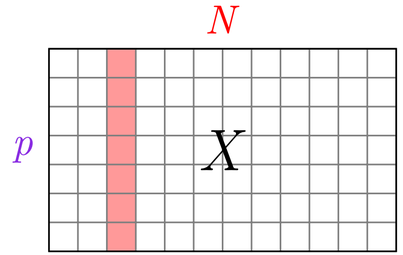
Think of $\mathcal{F}$ as FFT or DCT or Hadamard transform

also known under many names, e.g., Random Orthogonal System, FJLT

How to get guarantees?

Basic idea is that after applying the first two steps, resulting matrix has (almost) uniform leverage scores...

... so uniform subsampling is (almost) leverage score sampling.



Method 4: Fast Johnson-Lindenstrauss Transforms

R samples rows; $\mathcal{F}$ Fourier-like; D diag. w/ random ± 1 entries ("Rademacher")

More intuition

Suppose we want to estimate $\|\boldsymbol{x}\|_2^2$ for $\boldsymbol{x} \in \mathbb{R}^p$ by sampling just p_{small} entries (and multiplying by p/p_{small}). Wlog, let $\|\boldsymbol{x}\|_2 = 1$.

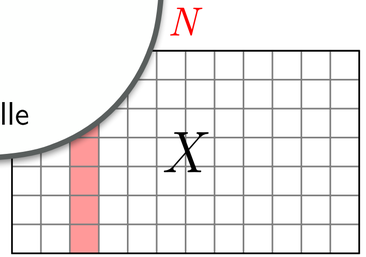
- High variance if $\boldsymbol{x}$ has a few large components
- Intuition: the matrix $\mathcal{F}D$ flattens out input vectors
  - Components of $\boldsymbol{x}$ can be as large as 1
  - Average component of $\boldsymbol{x}$ is $p^{-1/2}$
- Define $\boldsymbol{y} = \mathcal{F}D\boldsymbol{x}$, so $\|\boldsymbol{y}\|_2 = \|\boldsymbol{x}\|_2$. Then $y_1 = \sum_{j=1}^p \mathcal{F}_{1,j}\varepsilon_j x_j$
  - $\mathbb{E}\, y_1 = 0$ since $\mathbb{E}\, \varepsilon_j = 0$
  - $\text{Var}(y_1) = p^{-1}$ since $\mathcal{F}_{1,j}^2 \le \eta = p^{-1}$ and $\|\boldsymbol{x}\|_2 = 1$ and ε_j independent
- Hoeffding inequality:

$$\mathbb{P}\left(|y_1| \ge t\right) \le 2e^{-pt^2/2}$$

- So chance that one of y_j for $j = 1, \ldots, p$ is $\ge t$ is less than $p2e^{-pt^2/2}$
- So with constant probability, can choose $t \approx \sqrt{p^{-1}\log(p)}$
- Conclusion: the max component of $\boldsymbol{y}$ is about the same as its average component. Subsampling $\boldsymbol{y}$ will have much less variance than subsampling $\boldsymbol{x}$.

from "Improved analysis of the subsampled randomized Hadamard transform" (Tropp, 2010); see also Ailon and Chazelle

also kn

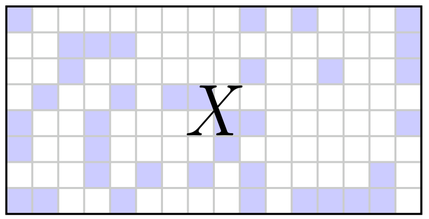


Method 5: subsample entries of a matrix

Keep $x_i' \in \mathbb{R}^p$ but with only p_{small} nonzeros

Idea goes back to landmark Achlioptas/McSherry paper (2001)

$$X' = \quad p \overbrace{\quad\quad\quad\quad}^{N} \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} X$$



This is the odd-one-out so far, because it's not the same linear operator applied to every column

Method 6: "**precondition**," *then* subsample entries of a matrix

Our new twist: first precondition with Random Orthogonal System (ROS)

(i.e., first 2 matrices from the Fast JLT)

$$Y = \underset{p}{\underset{\uparrow}{\mathcal{F}}} \; \underset{p}{\underset{\uparrow}{D}} \; \underset{N}{\underset{\uparrow}{X}}$$

$$Y' = \underset{N}{\underset{\uparrow}{Y}}$$

Why precondition?

Easy to see that better than uniform sampling of X is **non-uniform** sampling, with probability proportional to magnitude of entry

cf. Achlioptas, Z. Karnin, and E. Liberty 2013

Disadvantage of weighted sampling is the extra pass through data

Why precondition?

Easy to see that better than uniform sampling of X is **non-uniform** sampling, with probability proportional to magnitude of entry

cf. Achlioptas, Z. Karnin, and E. Liberty 2013

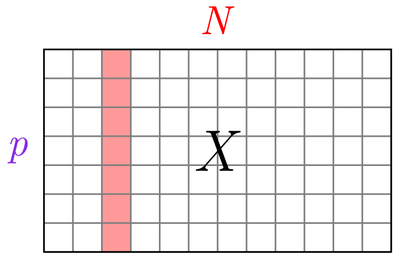
Disadvantage of weighted sampling is the extra pass through data

In our analysis, we have terms like (want this small)

$$\|X\|_{\text{max-entry}} = \max_{\substack{i=1,\ldots,N \\ j=1,\ldots,p}} |X_{ij}|$$

Assuming $\|x_i\|_2 = 1$,

- it is possible for $\|X\|_{\text{max-entry}} = 1$ (BAD)
- best case is $\|X\|_{\text{max-entry}} = 1/\sqrt{p}$



Why precondition?

Easy to see that better than uniform sampling of X is **non-uniform** sampling, with probability proportional to magnitude of entry

cf. Achlioptas, Z. Karnin, and E. Liberty 2013

Disadvantage of weighted sampling is the extra pass through data

In our analysis, we have terms like (want this small)

$$\|X\|_{\text{max-entry}} = \max_{\substack{i=1,\dots,N \\ j=1,\dots,p}} |X_{ij}|$$

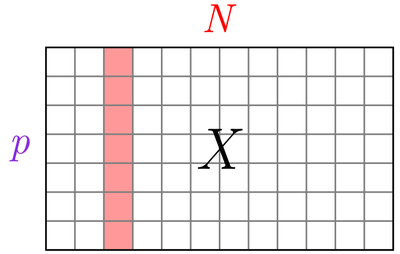
Assuming $\|x_i\|_2 = 1$,

- it is possible for $\|X\|_{\text{max-entry}} = 1$ (BAD)
- best case is $\|X\|_{\text{max-entry}} = 1/\sqrt{p}$

Benefit of preconditioning
After applying ROS, exponentially small chance that

$$\|Y\|_{\text{max-entry}} > \sqrt{\log(Np)}/\sqrt{p}$$

N.B. Since $\mathcal{F}D$ is unitary, bounds in spectral/Frobenius norm are unchanged



Theory: estimating the mean

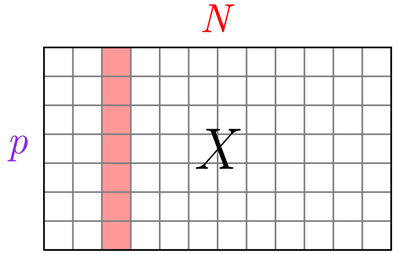
Given a true mean $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, and our estimate of it $\hat{\bar{x}}$ from sampled data,

Theorem (Pourkamali-Anaraki & B., *IEEE Trans. Info Theory* 2017)

$\mathbb{E}\,\hat{\bar{x}} = \bar{x}$ *and* $\|\hat{\bar{x}} - \bar{x}\|_\infty \leq t$ *with probability greater than*

$$1 - 2p\,exp\left(\frac{-N\gamma t^2/2}{\|X\|^2_{\text{max-row}} + t/3\|X\|_{\text{max-entry}}}\right)$$

where $\gamma = p_{small}/p$ *(simplifying to* $p_{small} \ll p \ll N$*)*



Theory: estimating the mean

Given a true mean $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, and our estimate of it $\hat{\bar{x}}$ from sampled data,

Theorem (Pourkamali-Anaraki & B., *IEEE Trans. Info Theory* 2017)

$\mathbb{E}\,\hat{\bar{x}} = \bar{x}$ and $\|\hat{\bar{x}} - \bar{x}\|_\infty \le t$ with probability greater than

$$1 - 2p\exp\left(\frac{-N\gamma t^2/2}{\|X\|^2_{\text{max-row}} + t/3\|X\|_{\text{max-entry}}}\right)$$

where $\gamma = p_{small}/p$ *(simplifying to $p_{small} \ll p \ll N$)*

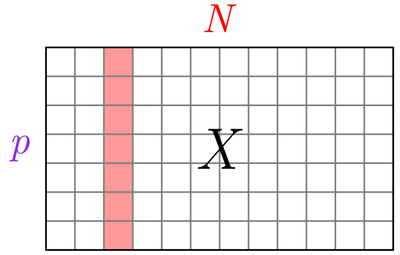
If X has normalized columns, then $\|X\|_{\text{max-entry}} = 1$ and $\|X\|_{\text{max-row}} = \sqrt{N}$ are possible, which is bad.

Lemma

If X is preconditioned, then *(for $Np = 10^{10}$, $\sqrt{\log(2Np) + 1000} = 32$)*

$$\mathbb{P}\left\{\|X\|_{\text{max-entry}} \ge \frac{\sqrt{2}}{\sqrt{p}} \cdot \sqrt{\log(2Np) + 1000}\right\} \le .001$$

$$\mathbb{P}\left\{\|X\|_{\text{max-row}} \ge \frac{\sqrt{2N}}{\sqrt{p}} \cdot \sqrt{\log(2Np) + 1000}\right\} \le .001$$



Theory: estimating the mean

Given a tru

Theor

$\mathbb{E}\,\hat{\bar{x}} =$

where

If X ha
possibl

Lemm

If X is

Probability theory aside

(symmetric Bernoulli, aka Rademacher)

Let $y_i = \pm 1$ be iid Bernoulli, $\hat{\bar{x}} = \frac{1}{N}\sum_{i=1}^{N} y_i$ (transform of Binomial)

- $\mathbb{E}(\hat{\bar{x}}) = 0$
- $\mathrm{Var}(\hat{\bar{x}}) = 1/N$

Chebyshev, Markov inequalities

$$\mathbb{P}(|\hat{\bar{x}}| \geq t) \leq \frac{1}{Nt^2}$$

Example: $N = 10^4$ and $t = 0.1$, $\mathbb{P}(|\hat{\bar{x}}| \geq t) \leq 0.01$

Markov
$X \geq 0,\ \mathbb{P}(X \geq t) \leq \frac{\mu}{t}$

Chebyshev
$\mathbb{P}\left(|X - \mu| \geq t\sigma^2\right) \leq \frac{1}{t^2}$

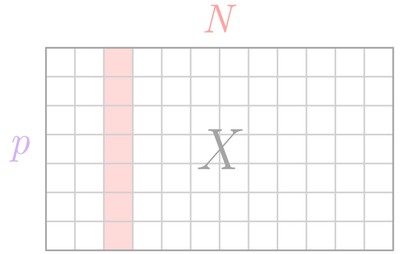
Concentration ineq.

Bernstein, Hoeffding, Chernoff

$$\mathbb{P}(|\hat{\bar{x}}| \geq t) \leq 2\exp\left(\frac{-Nt^2/2}{1 + t/3}\right)$$

Example: $N = 10^4$ and $t = 0.1$, $\mathbb{P}(|\hat{\bar{x}}| \geq t) \leq 2 \cdot 10^{-21}$

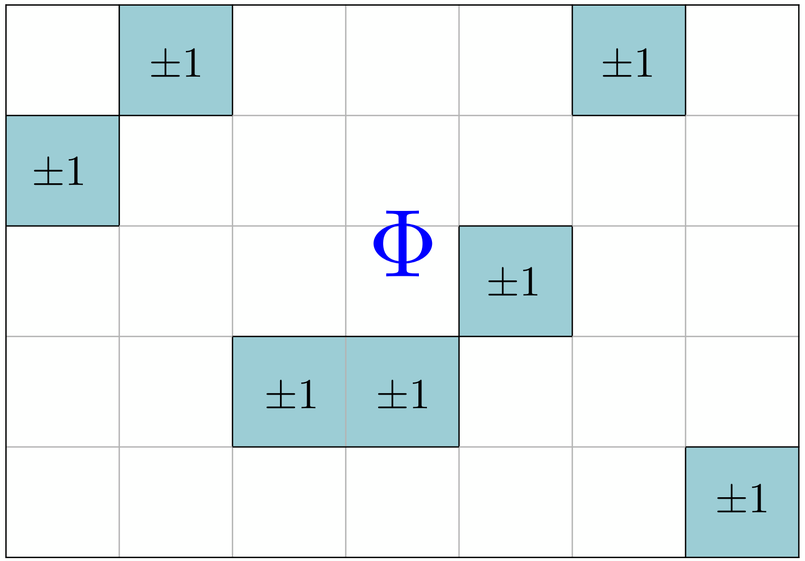
intuition: central limit theorem says random sums of iid rv should look Gaussian, and for a Gaussian, we have exponential concentration

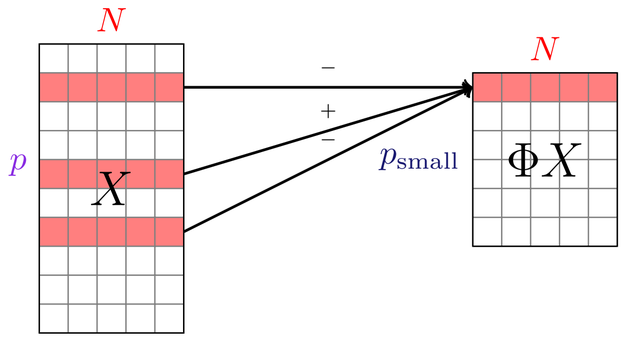


Method 7: CountSketch

p

p_{small}





Introduced in Charikar et al. (2004), more analysis in, e.g., Clarkson and Woodruff (2017)

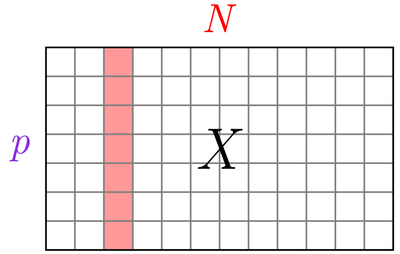
Based on **hash functions**
(as a side-effect, doesn't need full iid random variables for analysis to work)

See "appendix" of these slides for more details

Every column has exactly 1 nonzero entry, location chosen uniformly at random (and value is a Rademacher r.v.)

Has some guarantees, though not as good as a Gaussian (and not a JLT)
... but it's very fast to apply
cost: nnz(X), so no more than pN (vs $p_{\text{small}}pN$ with a Gaussian)

new linear algebra application in: Malik & B. "Fast randomized matrix and tensor interpolative decomposition using CountSketch", *Adv. Comp. Math* (2020)



Method 8: TensorSketch

Introduced in Pham, & Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

$$\Phi_{\mathcal{T}} : \mathbb{R}^{p} \to \mathbb{R}^{p_{\text{small}}}$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \quad p = p^{(1)} \cdot p^{(2)}$$

Not (yet!) related to tensors

Two tricks:

- **Computationally**, combine small sketches in a convenient way
- For **analysis**, we lose independence, but CountSketch analysis didn't require full independence!

Method 8: TensorSketch

Introduced in Pham, & Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

$$\Phi_{\mathcal{T}} : \mathbb{R}^{p} \to \mathbb{R}^{p_{\text{small}}}$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \quad p = p^{(1)} \cdot p^{(2)}$$

Kronecker/tensor product of vectors

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad \boldsymbol{a} \otimes \boldsymbol{b} = \begin{bmatrix} a_1 \boldsymbol{b} \\ a_2 \boldsymbol{b} \\ a_3 \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_1 b_2 \\ a_1 b_3 \\ a_2 b_1 \\ a_2 b_2 \\ a_2 b_3 \\ a_3 b_1 \\ a_3 b_2 \\ a_3 b_3 \end{bmatrix}$$

Method 8: TensorSketch

Introduced in Pham, & Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

$$\Phi_{\mathcal{T}} : \mathbb{R}^{p} \to \mathbb{R}^{p_{\text{small}}}$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \quad p = p^{(1)} \cdot p^{(2)}$$

Kronecker/tensor product of vectors

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad \boldsymbol{a} \otimes \boldsymbol{b} = \begin{bmatrix} a_1 \boldsymbol{b} \\ a_2 \boldsymbol{b} \\ a_3 \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_1 b_2 \\ a_1 b_3 \\ a_2 b_1 \\ a_2 b_2 \\ a_2 b_3 \\ a_3 b_1 \\ a_3 b_2 \\ a_3 b_3 \end{bmatrix}$$

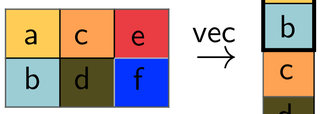
another equivalent definition:

$$\boldsymbol{a} \otimes \boldsymbol{b} = \text{vec}_{\text{col}}\left(\boldsymbol{b}\boldsymbol{a}^T\right)$$

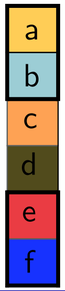
$$\boldsymbol{b}\boldsymbol{a}^T = \begin{bmatrix} b_1 a_1 & b_1 a_2 & b_1 a_3 \\ b_2 a_1 & b_2 a_2 & b_2 a_3 \\ b_3 a_1 & b_3 a_2 & b_3 a_3 \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_2 b_1 & a_3 b_1 \\ a_1 b_2 & a_2 b_2 & a_3 b_2 \\ a_1 b_3 & a_2 b_3 & a_3 b_3 \end{bmatrix}$$

vec: $\mathbb{R}^{p^{(1)} \times p^{(2)}} \to \mathbb{R}^{p^{(1)} p^{(2)}}$

mat = vec^{-1}



vec $\to$



more generally,

$$(\boldsymbol{A} \otimes \boldsymbol{B})\, \text{vec}_{\text{col}}(\boldsymbol{X}) = \text{vec}_{\text{col}}\left(\boldsymbol{B}\boldsymbol{X}\boldsymbol{A}^T\right)$$

TensorSketch: applying to matrices

From vector case $\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)}$ to matrix case $\mathbf{A} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(N)}$

Recall the Kronecker product:

$$\underset{I \times J}{\mathbf{A}} \otimes \underset{K \times L}{\mathbf{B}} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{IK \times JL}$$

$$= \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_1 \otimes \mathbf{b}_2 & \mathbf{a}_1 \otimes \mathbf{b}_3 & \cdots & \mathbf{a}_J \otimes \mathbf{b}_{L-1} & \mathbf{a}_J \otimes \mathbf{b}_L \end{bmatrix}$$

Khatri-Rao product $\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 \cdots \mathbf{a}_L \otimes \mathbf{b}_L \end{bmatrix} \in \mathbb{R}^{IK \times L} \quad (J = L)$

pointwise multiplication (aka Hadamard product)

Observe $\mathbf{a} \circ \mathbf{b} = \left(\mathbf{a}^\top \odot \mathbf{b}^\top\right)^\top$

and with some work, $\Phi_{\mathcal{T}}(\mathbf{A}) = \mathrm{FFT}^{-1}\left(\left(\bigodot_{n=1}^{N}\left(\mathrm{FFT}\left(\mathbf{S}^{(n)}\mathbf{A}^{(n)}\right)\right)^\top\right)^\top\right)$

if $\mathbf{A} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(N)}$

small countSketch

FFTs are doing circular convolution, i.e., multiplying polynomials modulo a fixed modulus. See appendix

TensorSketch: complexity analysis

$$\Phi_{\mathcal{T}} : \mathbb{R}^{p} \to \mathbb{R}^{p_{\text{small}}}$$

$$p = p^{(1)} \cdot p^{(2)}$$

Complexity (per mat-vec):

CountSketch applied as TensorSketch: $\mathcal{O}\big(p^{(1)} + p^{(2)} + p_{\text{small}} \log p_{\text{small}}\big)$ vs CountSketch applied naively: $\mathcal{O}\big(p^{(1)} \cdot p^{(2)}\big)$ vs generic dense matrix multiply: $\mathcal{O}\big(p^{(1)} \cdot p^{(2)} \cdot p_{\text{small}}\big)$

small CountSketches; polynomial multiplication via FFTs

Savings grow as we have more tensor products

If $p = p^{(1)} p^{(2)} \cdots p^{(q)}$

$$\mathcal{O}\big(p^{(1)} + p^{(2)} + \ldots p^{(q)} + p_{\text{small}} \log p_{\text{small}}\big) \quad \text{vs} \quad \mathcal{O}\big(p^{(1)} p^{(2)} \cdots p^{(q)}\big)$$

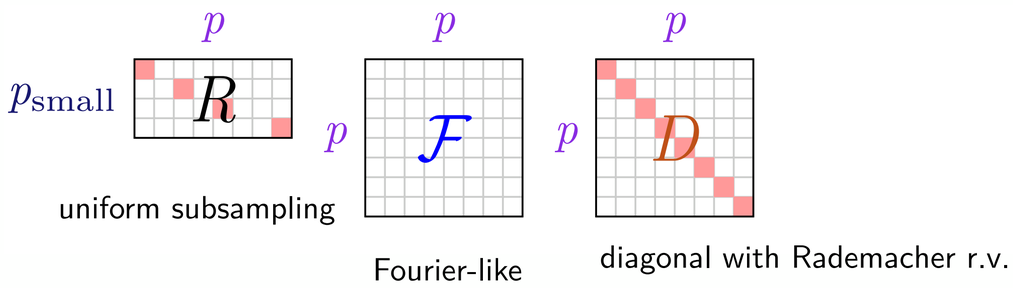
(Even larger savings if input factor matrices are **sparse**)

Method 9: Kronecker Fast Johnson-Lindenstrauss sketch

As for TensorSketch, suppose each column of data looks like $\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)}$

sizes: $p = p^{(1)} \cdot p^{(2)}$

Recall the Fast Johnson-Lindenstrauss (FJLT): $\Phi = R\mathcal{F}D$



The **Kronecker** FJLT has a similar structure:

$$\Phi_{\text{KFJLT}} = R\left(\mathcal{F}^{(1)}D^{(1)} \otimes \mathcal{F}^{(2)}D^{(2)}\right)$$

($\mathcal{F}^{(1)}D^{(1)}$: size $p^{(1)} \times p^{(1)}$; $\mathcal{F}^{(2)}D^{(2)}$: size $p^{(2)} \times p^{(2)}$)

which is efficient to apply since

$$R\left(\mathcal{F}^{(1)}D^{(1)} \otimes \mathcal{F}^{(2)}D^{(2)}\right)\cdot\left(\boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)}\right) = R\left(\left(\mathcal{F}^{(1)}D^{(1)}\boldsymbol{v}^{(1)}\right) \otimes \left(\mathcal{F}^{(2)}D^{(2)}\boldsymbol{v}^{(2)}\right)\right)$$

and uniform subsampling can be done implicitly without forming this

- Battaglino, Ballard, Kolda, "A practical randomized CP tensor decomposition", *SIAM J. Matrix Anal. Appl.* (2018)
- Jin, Kolda, Ward. "Faster Johnson–Lindenstrauss transforms via Kronecker products," *Information and Inference: A Journal of the IMA* (2021)
- Malik & B. "Guarantees for the Kronecker fast Johnson–Lindenstrauss transform using a coherence and sampling argument," *Lin. Alg. & its Applications* (2020)

Method 9: Kronecker Fast Johnson-Lindenstrauss sketch

$$\Phi_{\mathrm{KFJLT}} = R\left(\mathcal{F}^{(1)}D^{(1)} \otimes \mathcal{F}^{(2)}D^{(2)} \otimes \ldots \otimes \mathcal{F}^{(q)}D^{(q)}\right) \qquad p = p^{(1)}p^{(2)}\ldots p^{(q)}$$

Theorem (Thm 4.2 in Malik & B. 2020)

If all datapoints have a Kronecker structure, $p_{\mathrm{small}} > \frac{16}{2}\frac{4^q}{\varepsilon^2}\log\left(\frac{4N^2(q+1)}{\delta}\right)\log\left(\frac{4p^{(1)}N^2(q+1)}{\delta}\right)\cdots\log\left(\frac{4p^{(q)}N^2(q+1)}{\delta}\right)$

and $\Phi = \Phi_{\mathrm{KFJLT}}$

then with probability at least $1-\delta$

$$\forall x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p \qquad 1-\varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2^2}{\|x_i - x_j\|_2^2} \leq 1+\varepsilon$$

proof idea: FJLT keeps leverage scores almost uniform. With Kronecker product structure, this is (almost) true also, due to properties of leverage scores of Kronecker products

in fact, can extend. See: Malik, Xu, Cheng, B., Doostan, Narayan. "Fast Algorithms for Monotone Lower Subsets of Kronecker Least Squares Problems", https://arxiv.org/abs/2209.05662

- Malik & B. "Guarantees for the Kronecker fast Johnson–Lindenstrauss transform using a coherence and sampling argument," *Lin. Alg. & its Applications* (2020)

Method 9: Kronecker Fast Johnson-Lindenstrauss sketch

$$\Phi_{\text{KFJLT}} = R\left(\mathcal{F}^{(1)}D^{(1)} \otimes \mathcal{F}^{(2)}D^{(2)} \otimes \ldots \otimes \mathcal{F}^{(2)}D^{(q)}\right) \qquad p = p^{(1)}p^{(2)}\ldots p^{(q)}$$

Theorem (Thm 4.2 in Malik & B. 2020)

If all datapoints have a Kronecker structure, $p_{\text{small}} > \frac{16}{2}\frac{4^q}{\varepsilon^2}\log\left(\frac{4N^2(q+1)}{\delta}\right)\log\left(\frac{4p^{(1)}N^2(q+1)}{\delta}\right)\cdots\log\left(\frac{4p^{(q)}N^2(q+1)}{\delta}\right)$

and $\Phi = \Phi_{\text{KFJLT}}$

then with probability at least $1-\delta$

$$\forall x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p \qquad 1-\varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2^2}{\|x_i - x_j\|_2^2} \leq 1+\varepsilon$$

no dependence on p

for reference:

Theorem (Johnson-Lindenstrauss, 1984 (and Indyk-Motwani, 1998))

Choose $\Phi = \Phi_{randn}$ with $p_{small} \propto \varepsilon^{-2}\log N$ iid rows each $\mathcal{N}(0, p/p_{small})$, then for all $x_i, x_j \in \{x_1, \ldots, x_N\} \subset \mathbb{R}^p$,

$$1-\varepsilon \leq \frac{\|\Phi x_i - \Phi x_j\|_2}{\|x_i - x_j\|_2} \leq 1+\varepsilon$$

with constant probability.

OUTLINE

1. Randomized "sketches"
   a. Warmup: PCA
   b. Classical sketches
   c. Structured sketches
2. Applications
   a. Warmup: linear algebra
   b. K-means clustering
   c. Tensor factorizations
   d. Gradient-free optimization

One application of sketching: least-squares

Least-squares

Solve $x_{\mathrm{LS}} = \operatorname{argmin} \|Ax - b\|$, A is $p \times N$ with $p \gg N$

Approach 1: randomization to quickly find preconditioner BLENDENPIK, LSRN

"sketch-to-precondition"

- If $A = QR$ is a QR-decomposition, then AR^{-1} is well-conditioned
- Idea: do QR-decomp on reduced-dimension matrix ΦA
- On large, very ill-conditioned matrices (and tall), about $4\times$ faster than LAPACK

One application of sketching: least-squares

Least-squares

Solve $x_{\mathrm{LS}} = \mathrm{argmin} \|Ax - b\|$, A is $p \times N$ with $p \gg N$

Approach 1: randomization to quickly find preconditioner BLENDENPIK, LSRN

"sketch-to-precondition"

- If $A = QR$ is a QR-decomposition, then AR^{-1} is well-conditioned
- Idea: do QR-decomp on reduced-dimension matrix ΦA
- On large, very ill-conditioned matrices (and tall), about $4\times$ faster than LAPACK

Approach 2: directly solve sketched problem Sarlos, Woodruff, Mahoney, etc.

"sketch-to-solve"

- Directly sove $\min_x \|\Phi(Ax - b)\|$
- Theoretical bounds on objective error if Φ has $p_{\mathrm{small}} = N\mathrm{polylog}(N)$ rows
  - i.e., p_{small} is independent of p

For a sub-sampling sketch, this lets you avoid even computing some entries of b, which can be very useful when each entry of b is, e.g., the output of a simulation

One application of sketching: least-squares

Least-squares

Solve $x_{\text{LS}} = \text{argmin} \|Ax - b\|$, A is $p \times N$ with $p \gg N$

Approach 1: randomization to quickly find preconditioner BLENDENPIK, LSRN

- If $A = QR$ is a QR-decomposition, then AR^{-1} is well-conditioned
- Idea: do QR-decomp on reduced-dimension matrix ΦA
- On large, very ill-conditioned matrices (and tall), about $4\times$ faster than LAPACK

Approach 2: directly solve sketched problem Sarlos, Woodruff, Mahoney, etc.

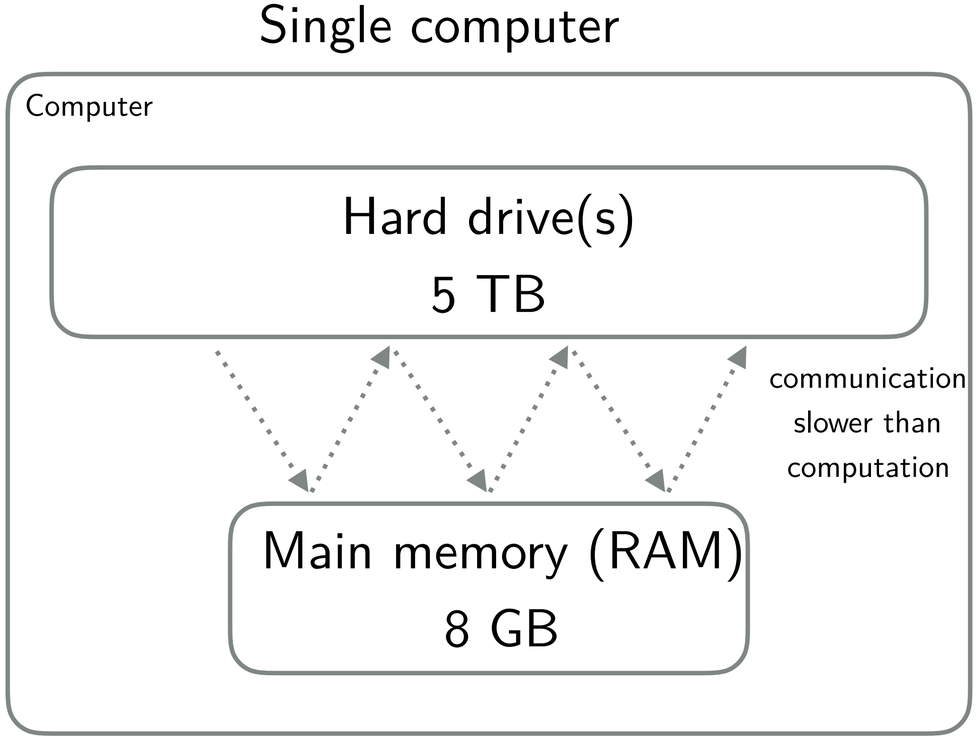
- Directly sove $\min_x \|\Phi(Ax - b)\|$
- Theoretical bounds on objective error if Φ has $p_{\text{small}} = N\text{polylog}(N)$ rows
  - i.e., p_{small} is independent of p

Other applications: SVDs, matrix multiplies, Nystrom/CUR/Interpolative Decompositions, QR without pivoting

OUTLINE

1. Randomized "sketches"
   a. Warmup: PCA
   b. Classical sketches
   c. Structured sketches
2. Applications
   a. Warmup: linear algebra
   b. K-means clustering
   c. Tensor factorizations
   d. Gradient-free optimization

Interlude: one-pass methods



Often want to minimize number of **passes** through a dataset if it's large

- *communication cost* can be significant
- in the extreme case, new data is constantly **streaming** in and we could never store it all

Think of a **pass** as a "for" loop through the dataset

Streaming is a special case of a "**one-pass**" method

Example: google indexing the web is streaming

1. Google finds a website and downloads the site
2. Applies algorithm to index/categorize/rank it
3. Discards the website (keeps only meta-data, ranking, etc.)
4. Never has to store entire WWW *all at once*

Interlude: one-pass methods

Example: computing the sample variance $\hat{\sigma}^2$

2-pass method

pass #1 $\quad \hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$

pseudocode

$$\begin{cases} \hat{\mu} = 0 \\ \texttt{for } i = 1, \dots, n \\ \qquad \hat{\mu} \leftarrow \hat{\mu} + x_i \\ \hat{\mu} \leftarrow \frac{1}{n}\hat{\mu} \end{cases}$$

pass #2 $\quad \hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \hat{\mu})^2$

pseudocode

$$\begin{cases} \hat{\sigma}^2 = 0 \\ \texttt{for } i = 1, \dots, n \\ \qquad \hat{\sigma}^2 \leftarrow \hat{\sigma}^2 + (x_i - \hat{\mu})^2 \\ \hat{\sigma}^2 \leftarrow \frac{1}{n-1}\hat{\sigma}^2 \end{cases}$$

1-pass method

motivation: $\quad \mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

pass #1

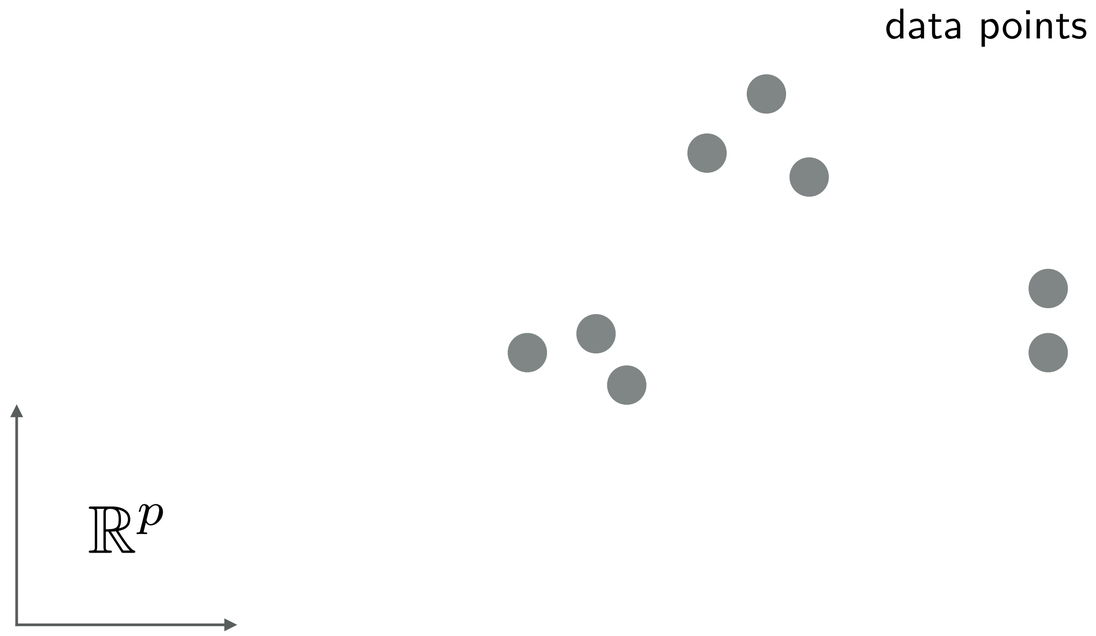
$$\begin{cases} \hat{\mu} = 0, \ s = 0 \\ \texttt{for } i = 1, \dots, n \\ \qquad \hat{\mu} \leftarrow \hat{\mu} + x_i \\ \qquad s \leftarrow s + x_i^2 \\ \hat{\sigma}^2 \leftarrow \frac{1}{n-1}(s - \hat{\mu}^2) \end{cases}$$

Interlude: one-pass methods

A major application of **sketching** is to create **one-pass** algorithms:

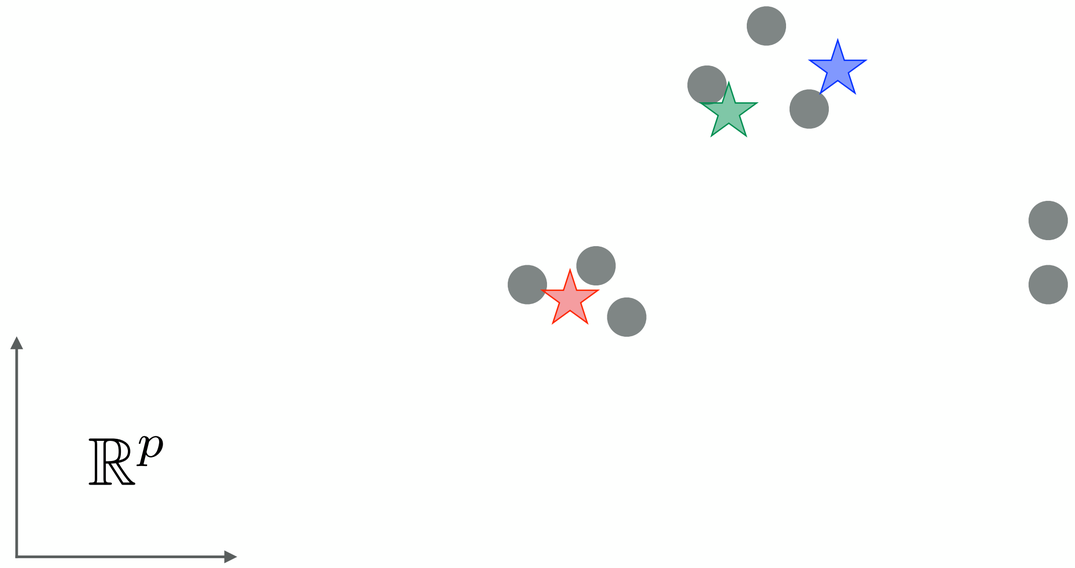
we spend one-pass to apply the sketch, and the sketched data is then small enough that we don't "count it" in our budget anymore

Machine Learning 101: K-means clustering



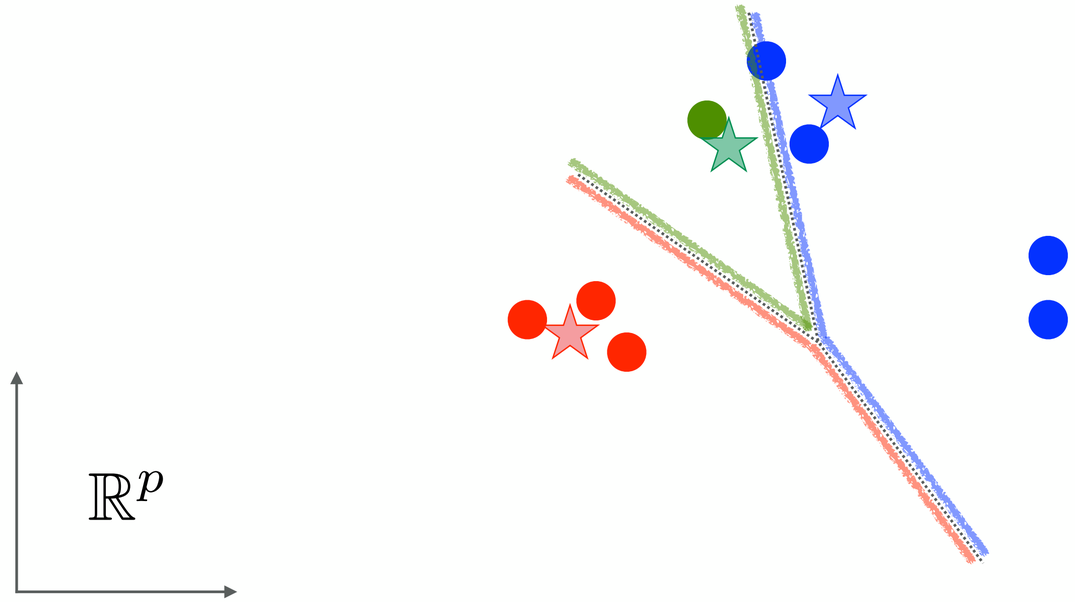
Machine Learning 101: K-means clustering

Step: initialize with guesses for cluster **centers**



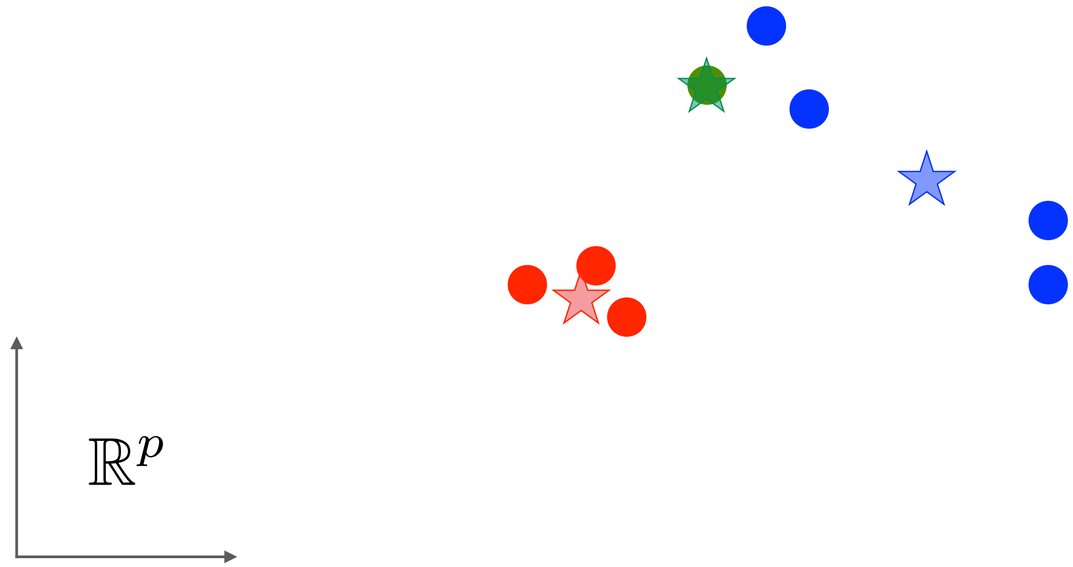
Machine Learning 101: K-means clustering

Step: update **assignments**



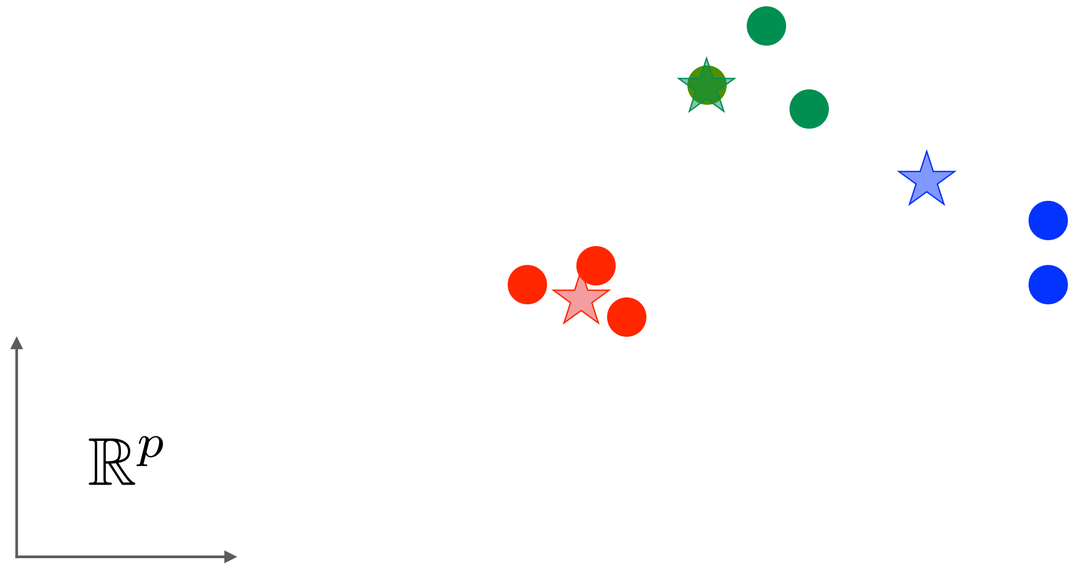
Machine Learning 101: K-means clustering

Step: recompute **cluster means**



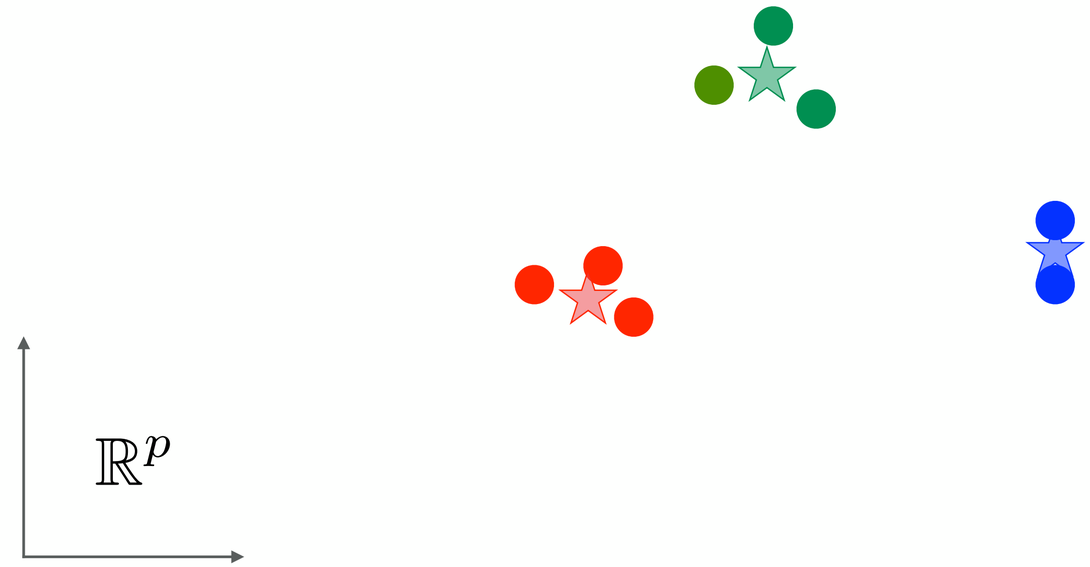
Machine Learning 101: K-means clustering

Step: update **assignments**



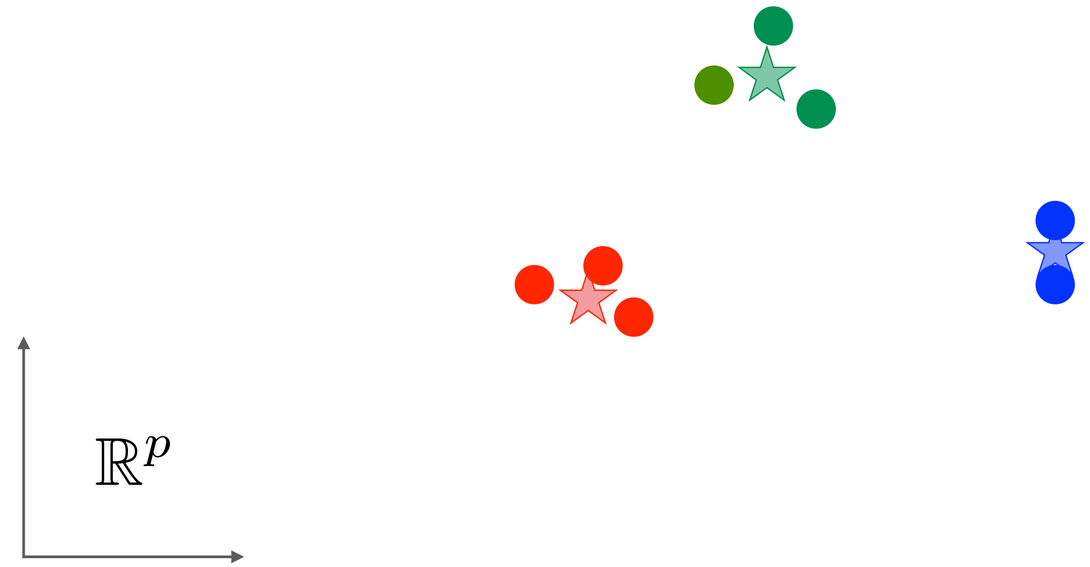
Machine Learning 101: K-means clustering

Step: recompute **cluster means**



Machine Learning 101: K-means clustering

Step: recompute **cluster means**



Theorem:

K-means procedure (Lloyd's algorithm) converges*

Proof:

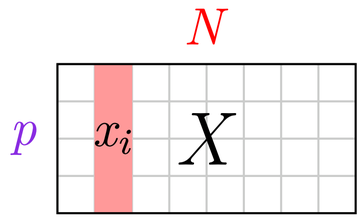
There are only a finite number of assignments, and we can never cycle since each iteration is an improvement

* *but it need not converge to the optimal solution!*

K-means clustering: complexity

Hard K-means / Lloyd's algorithm:
$\{x_i\}_{i=1,\dots,N} \subset \mathbb{R}^p$

1. Update cluster centers $\mu_k \in \mathbb{R}^p$ for $k = 1, \dots, K$
2. Update assignments $c_i \in \{1, \dots, K\}$ for $i = 1, \dots, N$



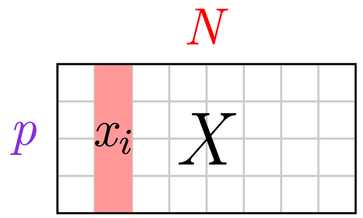
Let $X = (x_i)_{i=1,\dots,N}$
To update assignment of x_i:

- Compute $\|x_i - \mu_k\|$ (cost: p flops).
- For all k, and all i, this means $\mathcal{O}(KNp)$ cost.

K-means clustering: complexity

Hard K-means / Lloyd's algorithm:
$\{x_i\}_{i=1,\dots,N} \subset \mathbb{R}^p$

1. Update cluster centers $\mu_k \in \mathbb{R}^p$ for $k = 1, \dots, K$
2. Update assignments $c_i \in \{1, \dots, K\}$ for $i = 1, \dots, N$



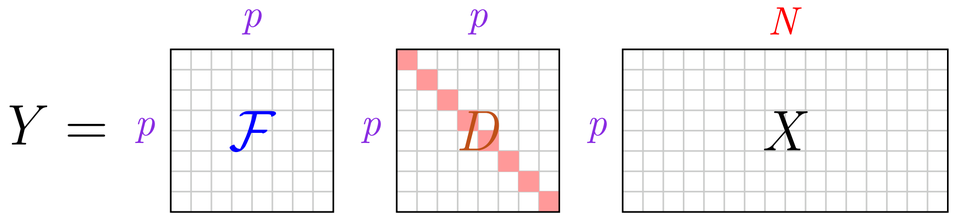
Let $X = (x_i)_{i=1,\dots,N}$
To update assignment of x_i:

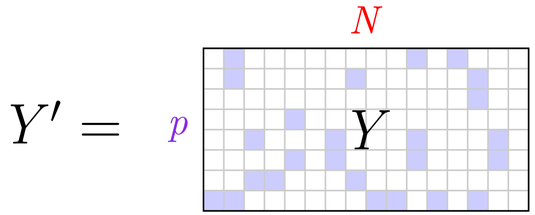
- Compute $\|x_i - \mu_k\|$ (cost: p flops).
- For all k, and all i, this means $\mathcal{O}(KNp)$ cost.

Because we're estimating the mean, we can use the theorems discussed earlier!

Our sampling is suitable for **streaming** data

Our idea: apply "preconditioned" entry-wise subsampling





Given a true mean $\bar{x} = \frac{1}{N}\sum_{i=1}^N x_i$, and our estimate of it $\hat{\bar{x}}$ from sampled data,

Theorem (Pourkamali-Anaraki & B., *IEEE Trans. Info Theory* 2017)
$\mathbb{E}\,\hat{\bar{x}} = \bar{x}$ *and* $\|\hat{\bar{x}} - \bar{x}\|_\infty \le t$ *with probability greater than*

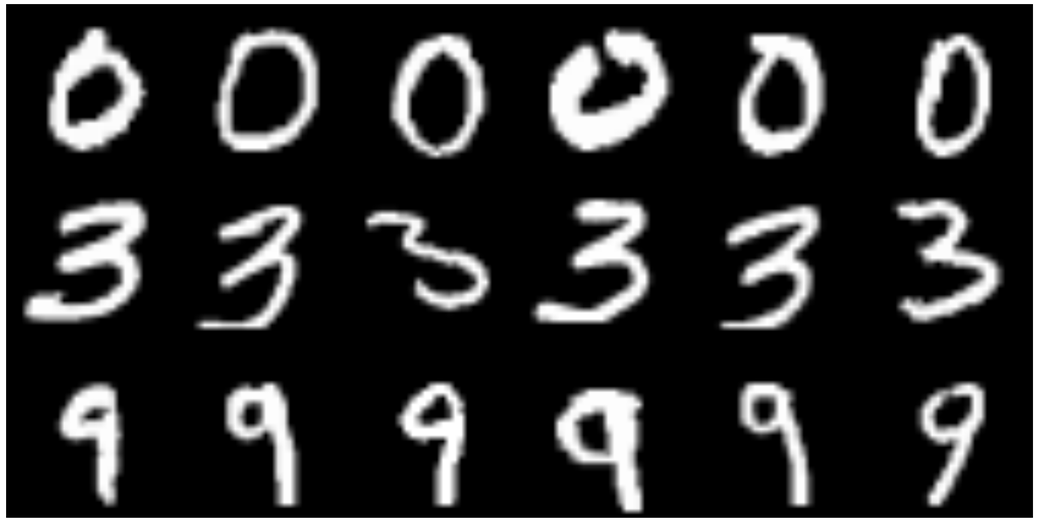
$$1 - 2p\exp\left(\frac{-N\gamma t^2/2}{\|X\|^2_{\text{max-row}} + t/3\|X\|_{\text{max-entry}}}\right)$$

where $\gamma = p_{small}/p$ *(simplifying to* $p_{small} \ll p \ll N$*)*

MNIST handwritten digits

Goal: recover cluster centers, using K-means clustering algorithm

Data: $N = 21{,}002$ examples of 28×28 pixel images ($p = 784$)



Experiments: MNIST handwritten digits

(a) true cluster centers

(b) K-means, many passes

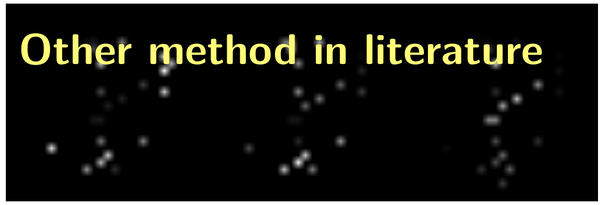
Experiments: MNIST handwritten digits

(a) true cluster centers

(b) K-means, many passes

(c) sparsified K-means, 1 pass, no preconditioning

(g) feature extraction, 1 pass



(h) feature selection, 3 passes

Experiments: MNIST handwritten digits

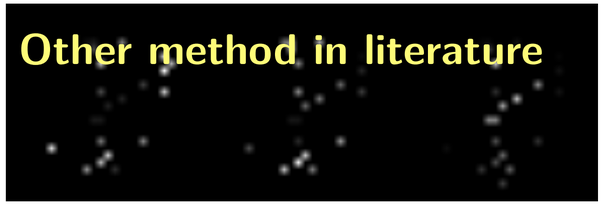
(a) true cluster centers

(b) K-means, many passes

(c) sparsified K-means, 1 pass, no preconditioning

(g) feature extraction, 1 pass

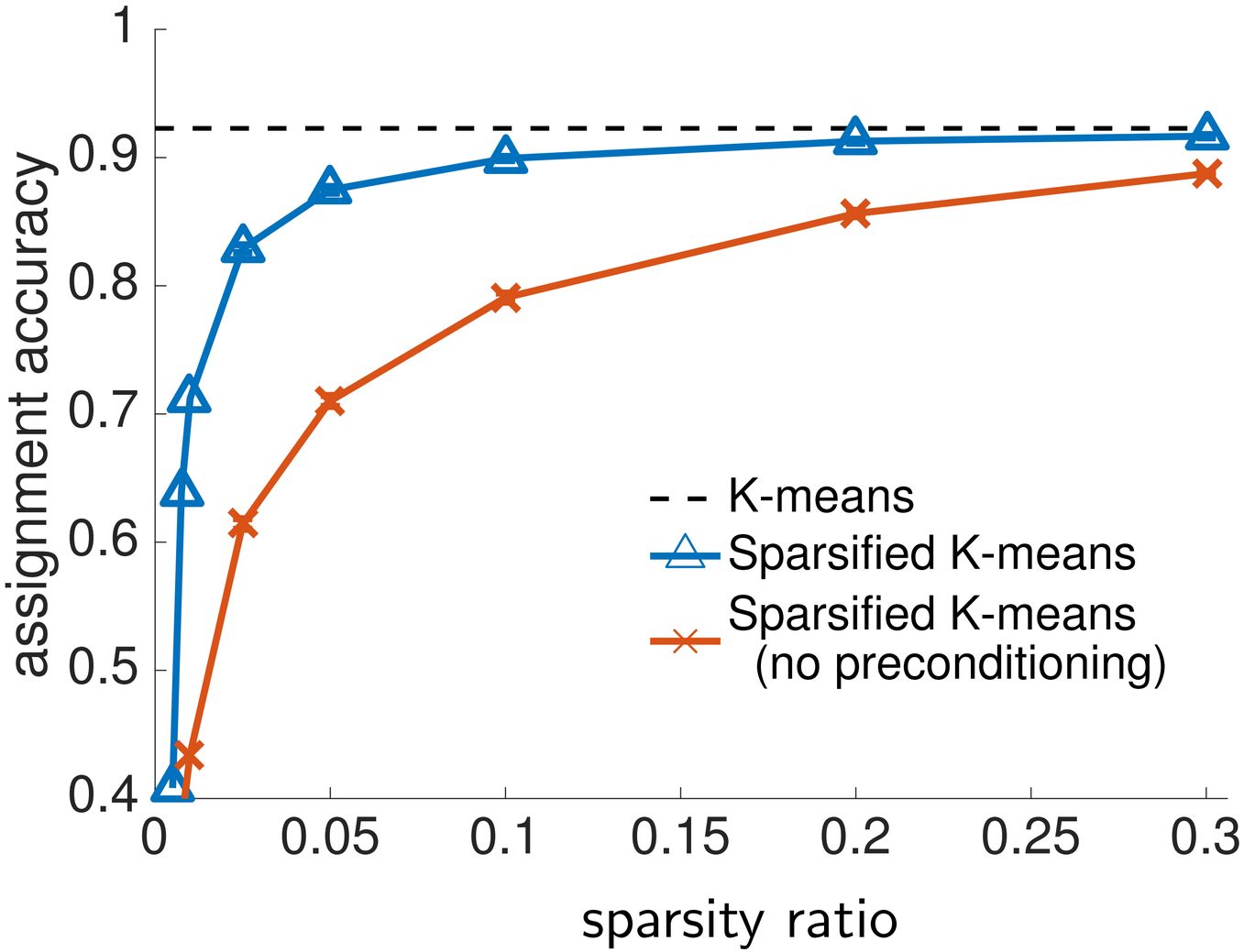
(d) sparsified K-means, 1 pass, preconditioned



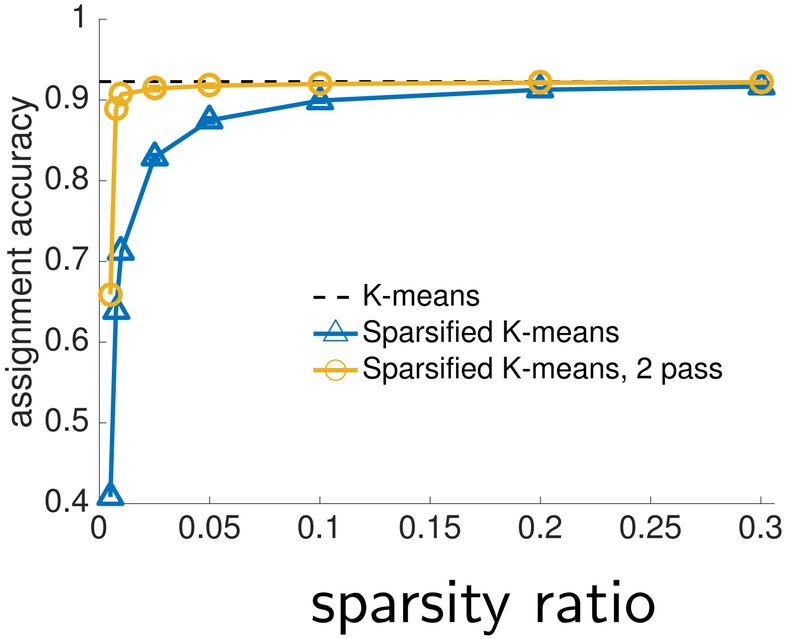
(h) feature selection, 3 passes

Accuracy on MNIST

Now look at **classification** accuracy (averaged over 50 trials)



"two-pass" version even more accurate



Speed on "Infinite MNIST"

"Infinite MNIST", $p = 784$, but now $N = 9{,}631{,}605$ instead of $21{,}002$

56 GB data, split into 58 chunks to keep RAM usage around 1 GB

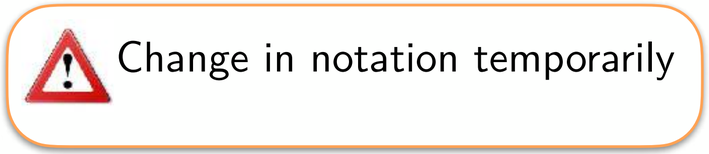
	Time to find assignments		Time to update all centers		Combined time	
Algorithm	Absolute	Speedup	Absolute	Speedup	Absolute	Speedup
K-means	130.0s	1×	150.8s	1×	280.8s	1×
Sparsified	1.3s	100×	5.7s	26.4×	7.0s	40.1×

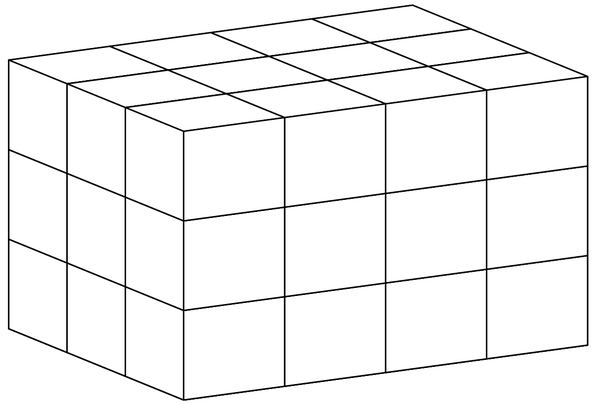
at 20x undersampling; accuracy is 89% (vs 92% for K-means)

OUTLINE

1. Randomized "sketches"
   a. Warmup: PCA
   b. Classical sketches
   c. Structured sketches
2. Applications
   a. Warmup: linear algebra
   b. K-means clustering
   c. Tensor factorizations
   d. Gradient-free optimization

Tensor Background: what is a tensor?





- A *tensor* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is an array of dimension N, also called an *N-way tensor* or *order N tensor*.
- A matrix $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2}$ is a 2-way tensor.
- A vector $\mathbf{x} \in \mathbb{R}^{I_1}$ is a 1-way tensor.
- A scalar $x \in \mathbb{R}$ is a 0-way tensor.

Examples of tensors

For example, [Kolda & Sun, '08] consider these datasets:

- **The Enron dataset** has size $1K \times 1K \times 1.1K \times 200$ and consists of elements of the form (user,user,keyword,day). Element (i, j, k, l) is 1 if user i sent an email to user j with keyword k on day l, 0 otherwise.
- **The DBLP dataset** has size $5K \times 1K \times 1K$ and consists of elements of the form (author,conference,keyword). Element (i, j, k) is 1 if author i published a paper at conference j containing the keyword k in the title, 0 otherwise.

These are examples of **sparse** tensors

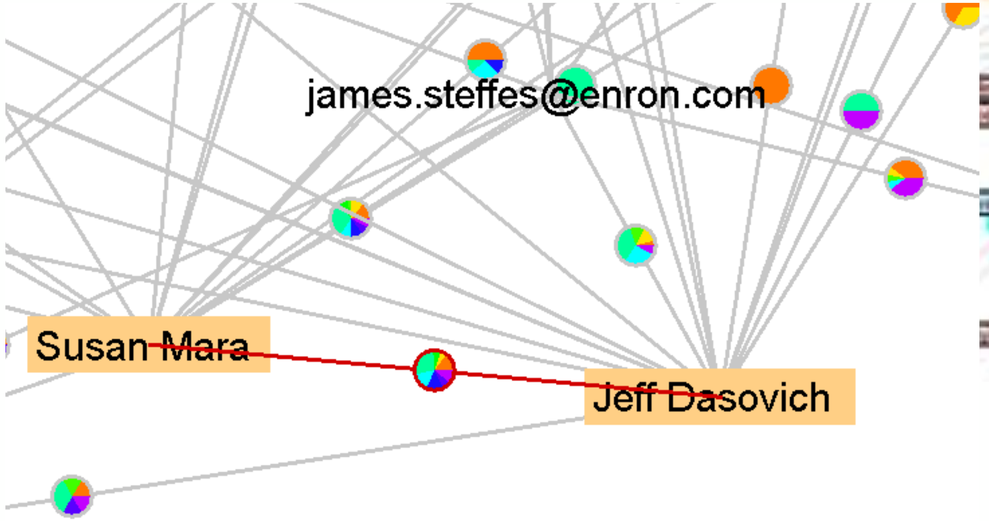


Image: https://homes.cs.washington.edu/~jheer/projects/enron/

Examples of tensors

For example, [Kolda & Sun, '08] consider these datasets:

- **The Enron dataset** has size $1K \times 1K \times 1.1K \times 1$ consists of elements of the form (user,user,keywor Element (i, j, k, l) is 1 if user i sent an email to us keyword k on day l, 0 otherwise.
- **The DBLP dataset** has size $5K \times 1K \times 1K$ and elements of the form (author,conference,keyword). (i, j, k) is 1 if author i published a paper at confer containing the keyword k in the title, 0 otherwise.

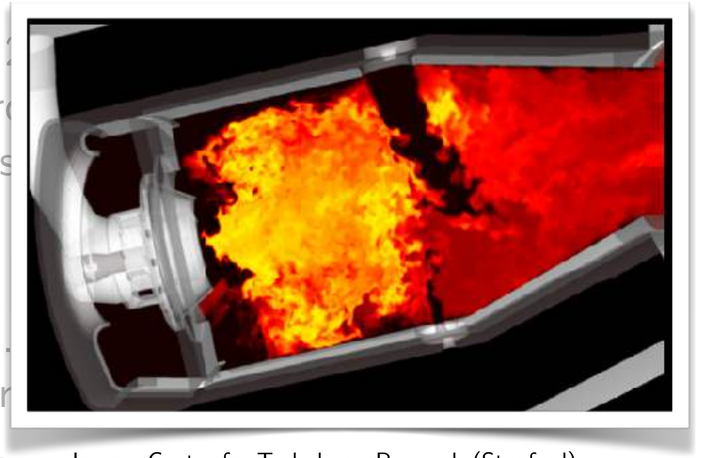
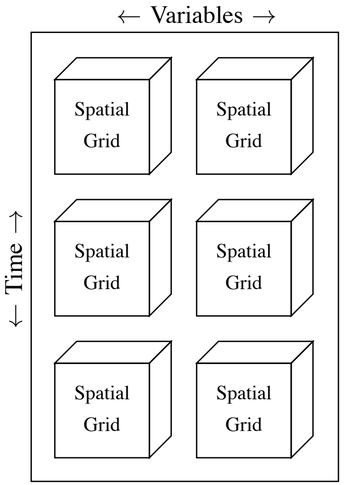


Image: Center for Turbulence Research (Stanford)

[Austin et al., '16] consider tensor data produced by high-fidelity combustion simulations with the following properties:

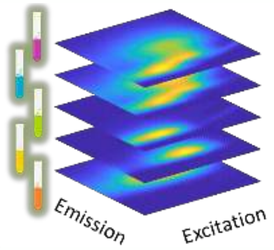
- 3-dimensional spatial grid with 512 points per dimension
- 64 variables are tracked per grid point
- 128 time steps 512 x 512 x 512 x 64 x 128 (8 TB)



This is a **dense** tensor

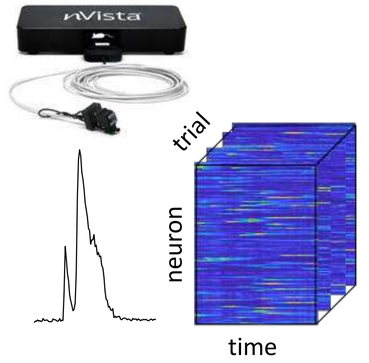
... we'll want methods that work with both sparse and dense tensors

Examples of tensors



For example, [Kolda & Sun, '08] consider these datasets:

- **The Enron dataset** has size $1K \times 1K \times 1.1K \times 200$ and consists of elements of the form (user,user,keyword,day). Element (i, j, k, l) is 1 if user i sent an email to user j with keyword k on day l, 0 otherwise.
- **The DBLP dataset** has size $5K \times 1K \times 1K$ and consists of elements of the form (author,conference,keyword). Element (i, j, k) is 1 if author i published a paper at conference j containing the keyword k in the title, 0 otherwise.



[Austin et al., '16] consider tensor data produced by high-fidelity combustion simulations with the following properties:

- 3-dimensional spatial grid with 512 points per dimension
- 64 variables are tracked per grid point
- 128 time steps 512 x 512 x 512 x 64 x 128 (8 TB)

Others:

- Chemometrics (emission x excitation x samples, for fluorescence spectroscopy)
- Neuroscience (neuron x time x trial/stimulus, for calcium imaging)
- Criminology (day x hour x location x crime, e.g., Chicago crime dataset)

Images: Tamara Kolda

Examples of tensors

For example, [Kolda & Sun, '08] consider these datasets:

- **The Enron d**
  consists of
  Element (
  keyword *k*
- **The DBL**
  elements
  (i, j, k) is
  containing

[Austin et al.,
combustion sir

- 3-dimensi
- 64 variable
- 128 time steps

512 x 512 x 512 x 64 x 128 (8 TB)

Others:

- Chemometrics (emission x excitation x samples, for fluorescence spectroscopy)
- Neuroscience (neuron x time x trial/stimulus, for calcium imaging)
- Criminology (day x hour x location x crime, e.g., Chicago crime dataset)

One-pass methods

All these tensors have a **time** or **sample** index

As time/samples increase, we don't want to store old ones

i.e., want a **streaming/one-pass method**

THIS IS A BIG MOTIVATION FOR SKETCHING

Tensor operations 1: matricization

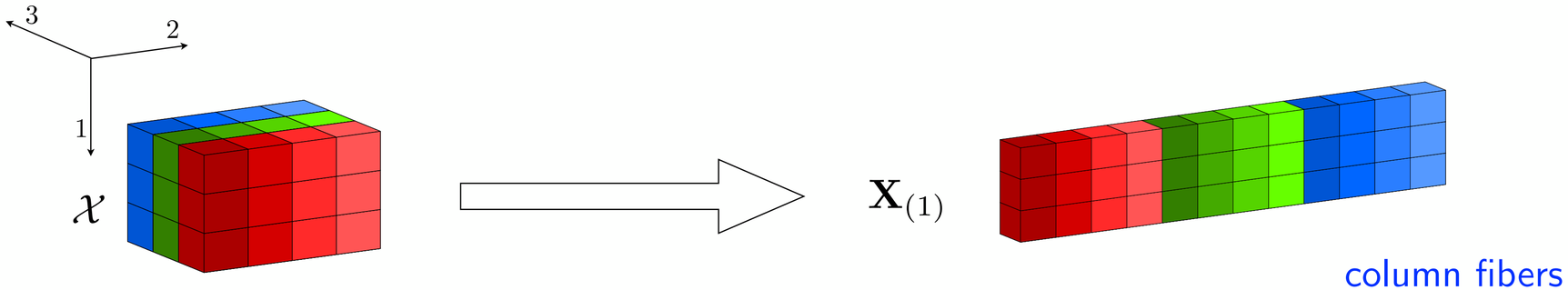
Matricization (i.e., flattening) turns a tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$

Same idea as turning a matrix (e.g., an image) into a vector

Tensor operations 1: matricization

Matricization (i.e., flattening) turns a tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$

Same idea as turning a matrix (e.g., an image) into a vector

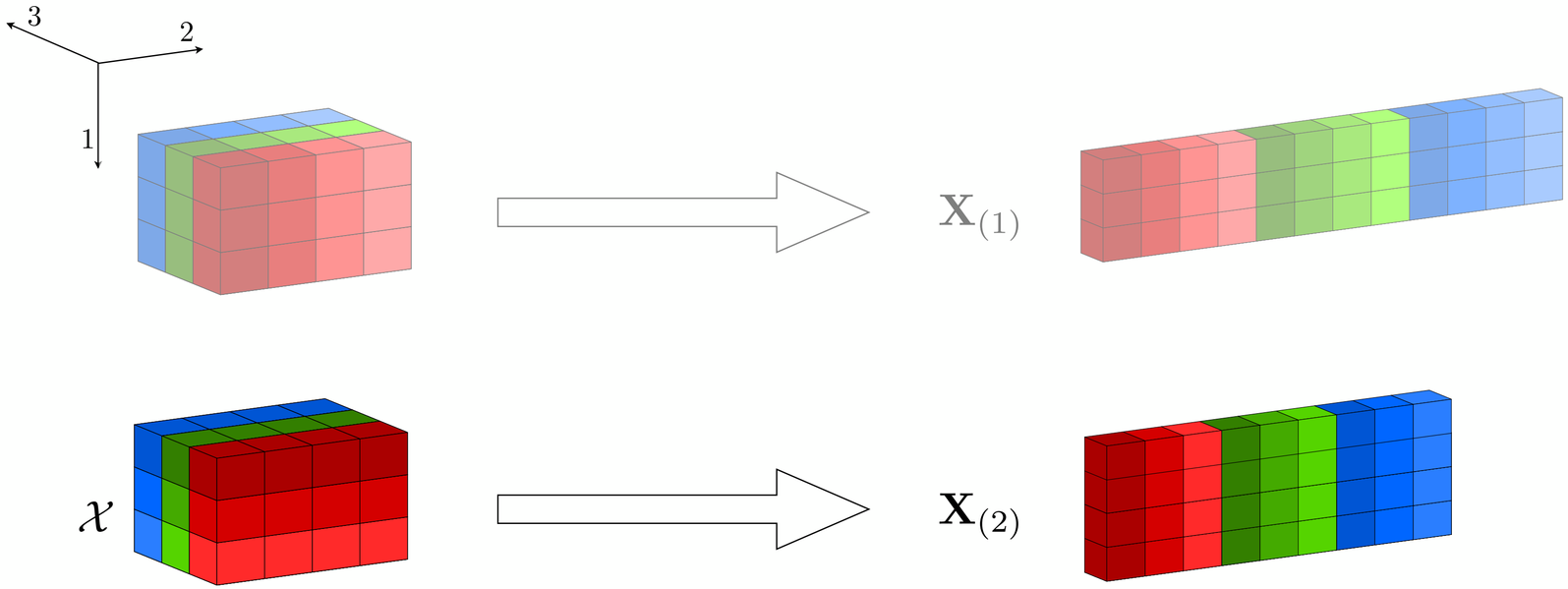


column fibers

Tensor operations 1: matricization

Matricization (i.e., flattening) turns a tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$

Same idea as turning a matrix (e.g., an image) into a vector

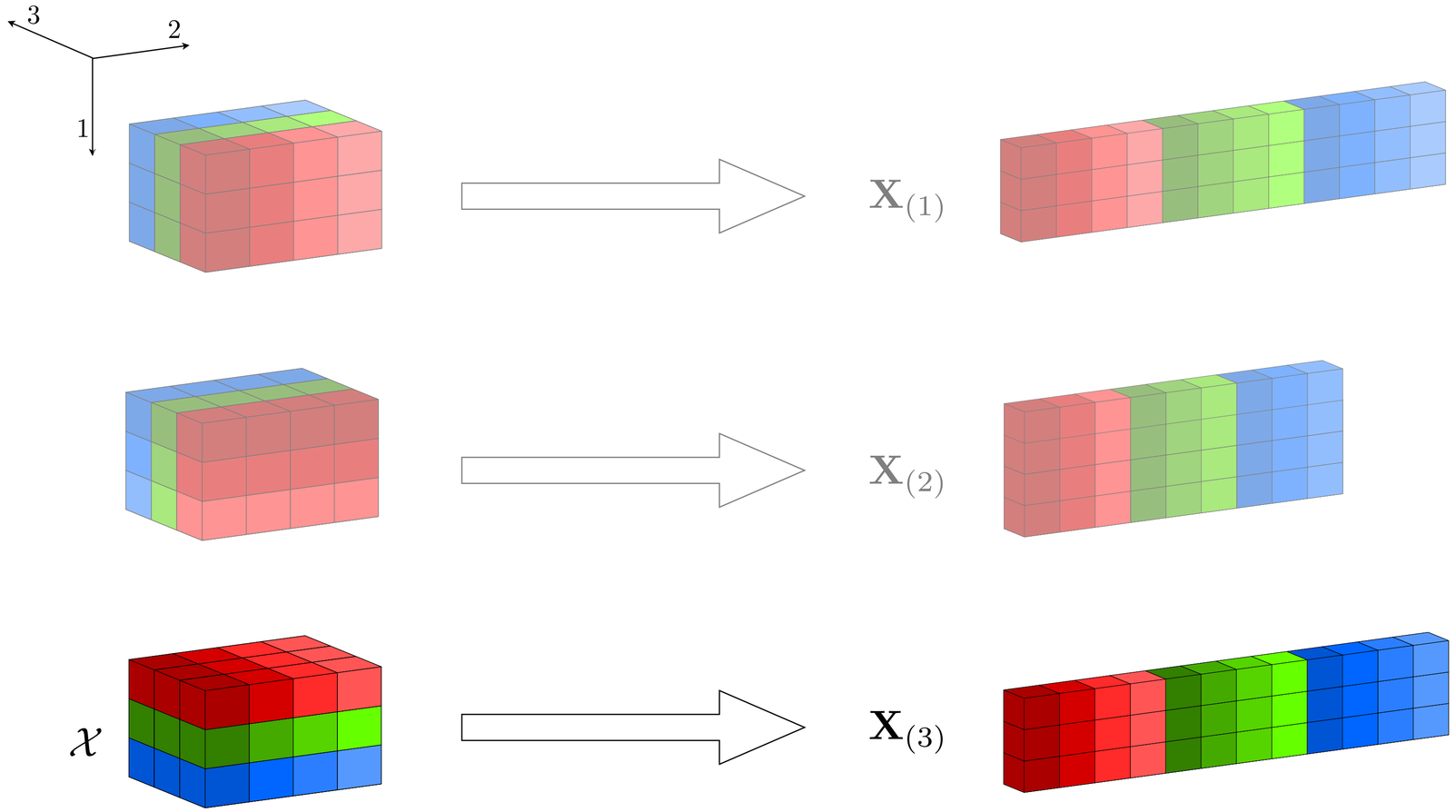


row fibers

Tensor operations 1: matricization

Matricization (i.e., flattening) turns a tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$

Same idea as turning a matrix (e.g., an image) into a vector



"tube" fibers

Tensor operations 2: tensor-times-matrix

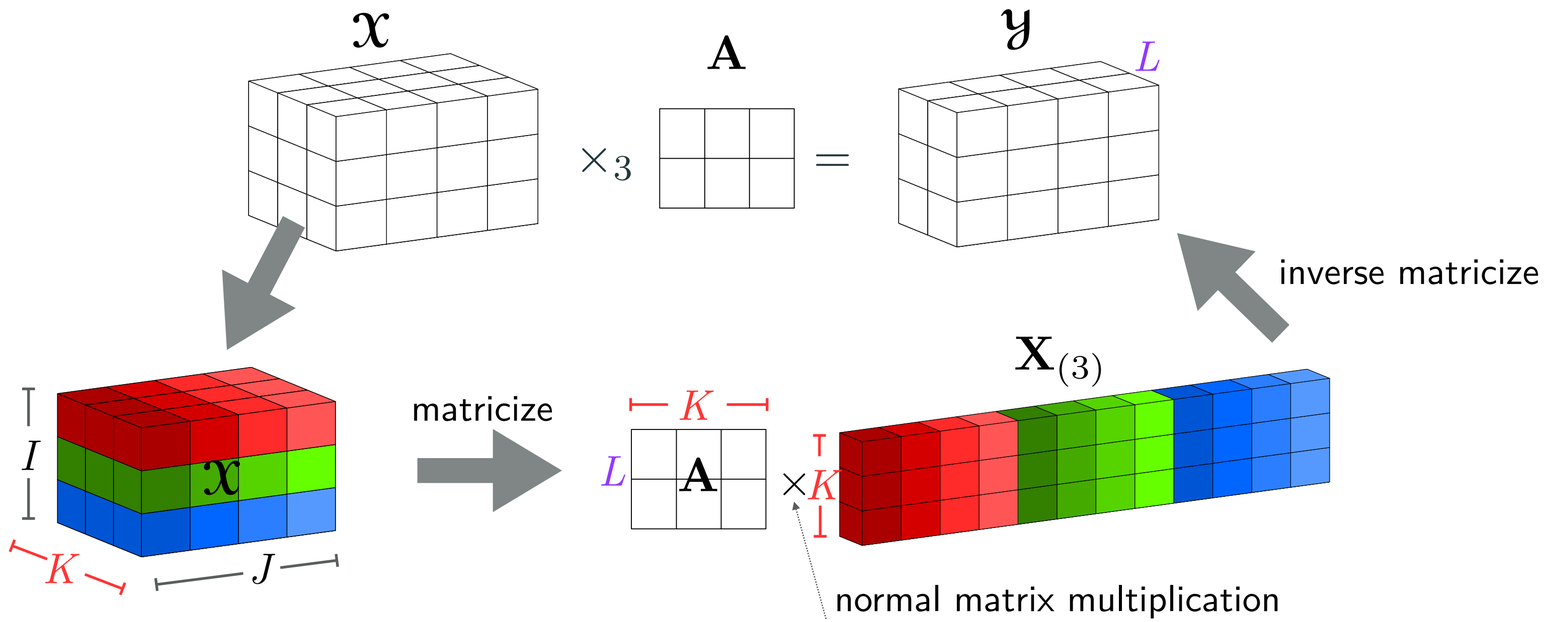
Example:

- Let $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and $\mathbf{A} \in \mathbb{R}^{L \times K}$.
- We define $\mathcal{Y} \overset{\text{def}}{=} \mathcal{X} \times_3 \mathbf{A} \in \mathbb{R}^{I \times J \times L}$ elementwise by

$$y_{ijl} = \sum_{k=1}^{K} x_{ijk} a_{kl}.$$

- In matrix terms, $\mathbf{Y}_{(3)} = \mathbf{A}\mathbf{X}_{(3)}$.

Conveniently, order doesn't matter*
$\mathcal{X} \times_1 \mathbf{A_1} \times_2 \mathbf{A_2} = \mathcal{X} \times_2 \mathbf{A_2} \times_1 \mathbf{A_1}$

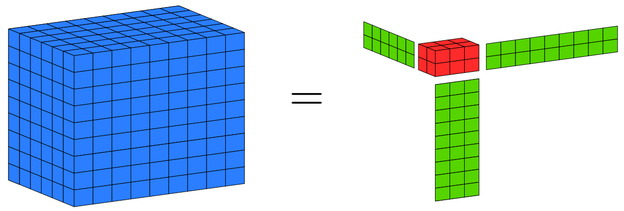


The Tucker decomposition

A Tucker decomposition of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is of the form

$$\mathcal{X} = \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} =: [\![\mathcal{G}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!] \text{ (shorthand notation)}$$

where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and each $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$.

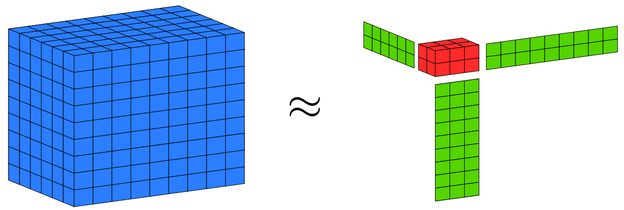


$\mathcal{G}$ is the *core tensor* and $\mathbf{A}^{(n)}$, $n = 1, 2, \ldots, N$, are *factor matrices*.

We say that $\mathcal{X}$ is a rank-$(R_1, R_2, \ldots, R_N)$ tensor.

Applications of the Tucker decomposition

- **Data compression.** Storing the red and green objects require less storage than the full blue tensor.
- **Data analysis.** Use rows of factor matrices as feature vectors—*unsupervised learning*.



Computing the Tucker decomposition

Given a data tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, we can find a decomposition $\mathcal{X} \approx [\![\mathcal{G}; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$ by solving the optimization problem

$$\min_{\mathcal{G}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}} \|\mathcal{X} - [\![\mathcal{G}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]\|^2, \tag{4}$$

where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ for $n = 1, 2, \ldots, N$.

- Nonlinear!
- A common approach to this issue is to use *alternating least squares (ALS)*: Minimize with respect to each $\mathbf{A}^{(n)}$ and $\mathcal{G}$ one at a time. Repeat!
- With ALS, each subproblem is linear and easily solved.

i.e., Gauss-Siedel style

ALS same as *higher-order orthogonal iteration* (HOOI)
and if you do just one iteration, it's like *higher-order SVD* (HOSVD)

Computing the Tucker decomposition

The algorithm looks like this. Repeat the following until convergence:

Repeat until convergence

1. For $n = 1, \dots, N$, update (10)

$$\mathbf{A}^{(n)} = \underset{\mathbf{A} \in \mathbb{R}^{I_n \times R_n}}{\arg\min} \left\| \left(\bigotimes_{\substack{i=N \\ i \neq n}}^{1} \mathbf{A}^{(i)} \right) \mathbf{G}_{(n)}^{\top} \mathbf{A}^{\top} - \mathbf{X}_{(n)}^{\top} \right\|_F^2 . \quad (11)$$

2. Update $\mathcal{G} = \mathcal{X} \times_1 \mathbf{A}^{(1)\top} \times_2 \mathbf{A}^{(2)\top} \cdots \times_N \mathbf{A}^{(N)\top}$. (12)

$$= \underset{\mathcal{Z} \in \mathbb{R}^{R_1 \times \cdots \times R_N}}{\arg\min} \left\| \left(\bigotimes_{i=N}^{1} \mathbf{A}^{(i)} \right) \mathbf{z}_{(:)} - \mathbf{x}_{(:)} \right\|_2^2 .$$

(Line 12 follows if the factor matrices are orthogonalized)

Make extensive use of this identity:

$$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} \quad \Leftrightarrow$$

$$\mathbf{Y}_{(n)} = \mathbf{A}^{(n)} \mathbf{X}_{(n)} \left(\mathbf{A}^{(N)} \otimes \cdots \otimes \mathbf{A}^{(n+1)} \otimes \mathbf{A}^{(n-1)} \otimes \cdots \otimes \mathbf{A}^{(1)} \right)^{\top}$$

Computing the Tucker decomposition

The algorithm looks like this. Repeat the following until convergence:

1. For $n = 1, \ldots, N$, update (10)

$$\mathbf{A}^{(n)} = \underset{\mathbf{A} \in \mathbb{R}^{I_n \times R_n}}{\arg\min} \left\| \left(\bigotimes_{\substack{i=N \\ i \neq n}}^{1} \mathbf{A}^{(i)} \right) \mathbf{G}_{(n)}^{\top} \mathbf{A}^{\top} - \mathbf{X}_{(n)}^{\top} \right\|_F^2 . \quad (11)$$

2. Update $\mathcal{G} = \mathcal{X} \times_1 \mathbf{A}^{(1)\top} \times_2 \mathbf{A}^{(2)\top} \cdots \times_N \mathbf{A}^{(N)\top}$. (12)

Issues when dealing with large, sparse tensors:

- The Kronecker product matrix is huge.
- The series of tensor-times-matrix products is very costly, and can require lots of additional memory when $\mathcal{X}$ is sparse.
- We address this problem by using *TensorSketch*.

Aside: *we initially thought that **solving** the least-squares problem would be an issue, and something to fix with sketching, but this is not exactly the case* → (partly because of many clever tricks due to orthogonality of A)

Our modified algorithm: Tucker-TS

We simply sketch each of these least-squares problems with TensorSketch operators $\mathbf{S}^{(n)}$, $n = 1, \ldots, N+1$, which are defined at the start of the algorithm:

Repeat until convergence

1. For $n = 1, \ldots, N$, update

$$\mathbf{A}^{(n)} = \underset{\mathbf{A} \in \mathbb{R}^{I_n \times R_n}}{\arg\min} \left\| \left(\mathbf{S}^{(n)} \bigotimes_{\substack{i=N \\ i \neq n}}^{1} \mathbf{A}^{(i)} \right) \mathbf{G}_{(n)}^{\top} \mathbf{A}^{\top} - \mathbf{S}^{(n)} \mathbf{X}_{(n)}^{\top} \right\|_F^2 . \tag{18}$$

(and orthogonalize)

2. Update

$$\mathcal{G} = \underset{\mathcal{Z} \in \mathbb{R}^{R_1 \times \cdots \times R_N}}{\arg\min} \left\| \left(\mathbf{S}^{(N+1)} \bigotimes_{i=N}^{1} \mathbf{A}^{(i)} \right) \mathbf{z}_{(:)} - \mathbf{S}^{(N+1)} \mathbf{x}_{(:)} \right\|_2^2 . \tag{19}$$

sketched once at very beginning

Tricks:

- don't draw new sketches each iteration
- the TensorSketches have CountSketches/hashes in common $\mathbf{S}^{(n)}, \mathbf{S}^{(n')}$ (each is leave-one-out)
- G update solved via conjugate gradient (CG) algo, and can prove it is well-conditioned
- advanced tricks and variants in the paper

Numerical experiment: sparse 3D tensor

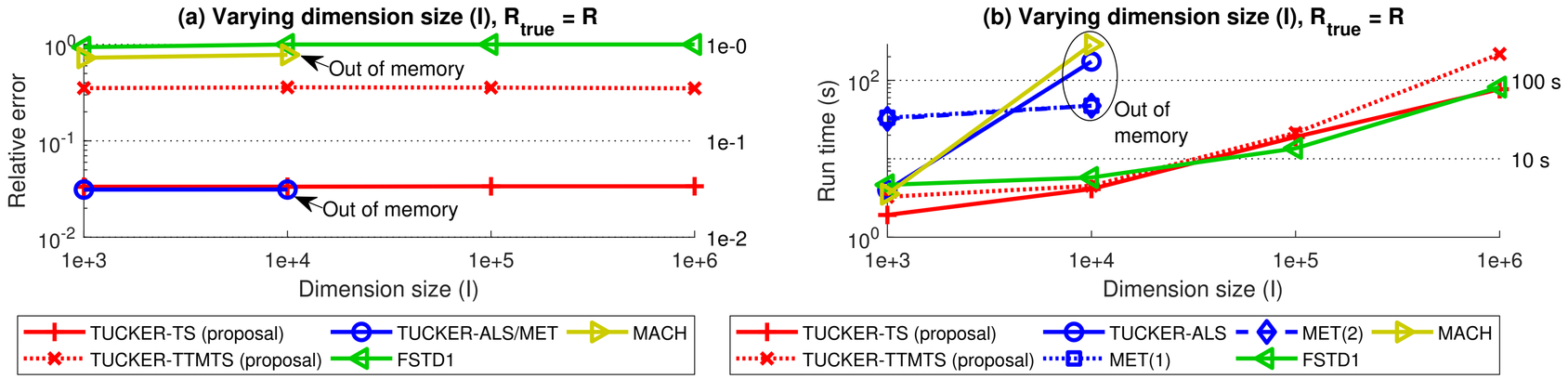


Figure 3: Relative error and run time for random sparse 3-way tensors with varying dimension size I and $\text{nnz}(\mathcal{Y}) \approx$ 1e+6. Both the true and target ranks are $(10, 10, 10)$.

Tucker-TS approximates least-squares like this:

$$\begin{aligned}\text{argmin}_x \frac{1}{2}\|Ax-b\|_2^2 &\approx \text{argmin}_x \frac{1}{2}\|\Phi(Ax-b)\|_2^2 \\ &= (A^\top \Phi^\top \Phi A)^{-1} A^\top \Phi^\top \Phi b\end{aligned}$$

Tucker-TTMTS approximates least-squares like this:

$$\begin{aligned}\text{argmin}_x \frac{1}{2}\|Ax-b\|_2^2 &= (A^\top A)^{-1} A^\top b \\ &\approx (A^\top A)^{-1} A^\top \Phi^\top \Phi b\end{aligned}$$

(and exploit orthogonality here)

Video

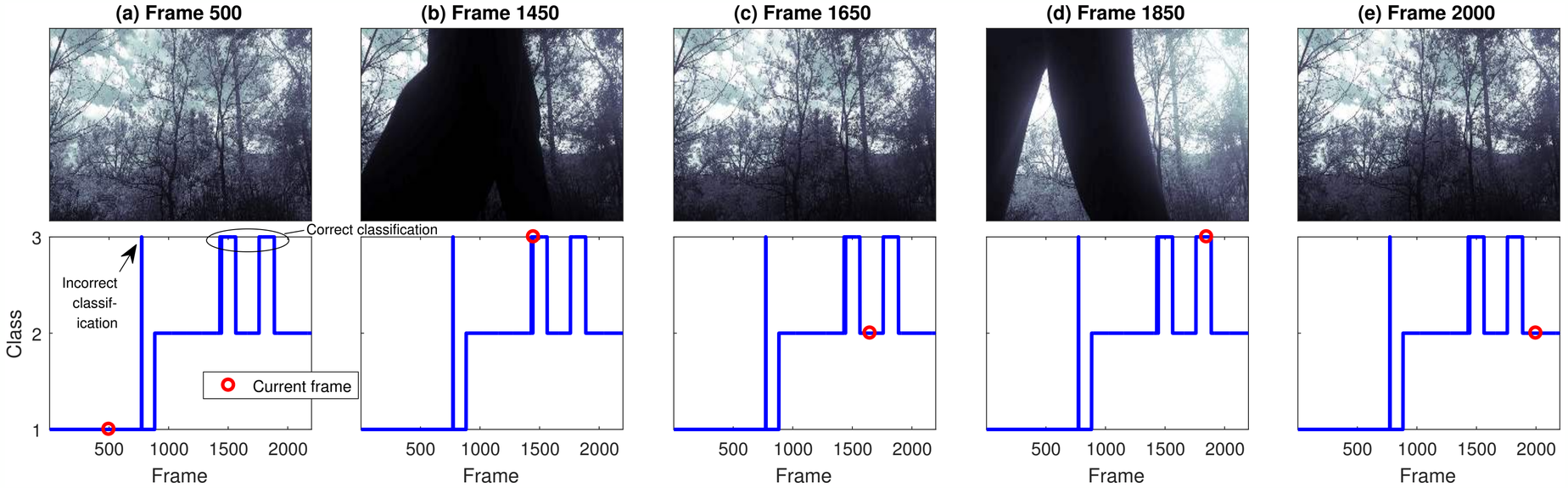


Figure 6: Five sample frames with their assigned classes. The frames (b) and (d) contain a disturbance.

38 GB video (can't all fit into RAM): 2,200 frames, each of size 1,080 by 1,980 pixels

Compute rank (10,10,10) factorization, 30 iterations max

Find 3 factor matrices — take the one corresponding to time and run k-means clustering

We load chunk-by-chunk and do the sketches (in one pass)

(used Tucker-TTMTS variant)

OUTLINE

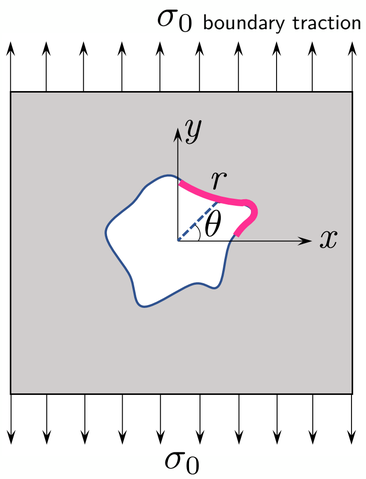
1. Randomized "sketches"
   a. Warmup: PCA
   b. Classical sketches
   c. Structured sketches
2. Applications
   a. Warmup: linear algebra
   b. K-means clustering
   c. Tensor factorizations
   d. Gradient-free optimization

Motivating Application: shape optimization

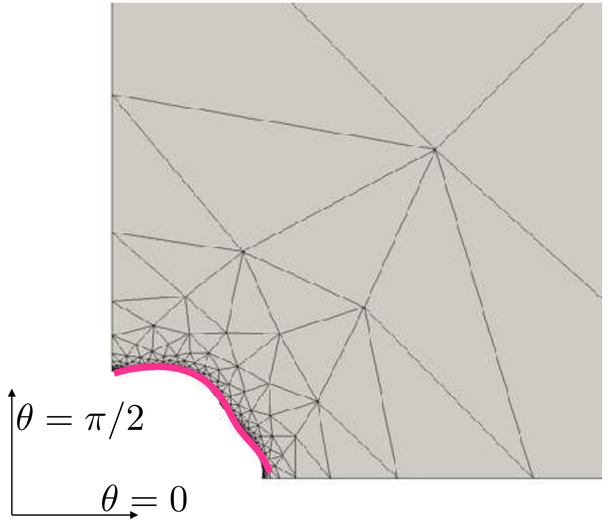
Forward problem: find vertical stress σ_y

linear elasticity PDE

(boundary conditions depend on shape of the hole)



conforming finite element mesh, used in FEniCS



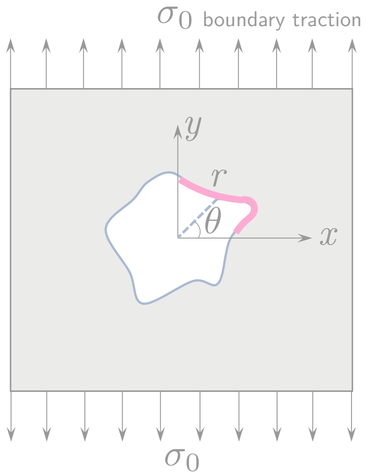

Change in notation temporarily

Motivating Application: shape optimization

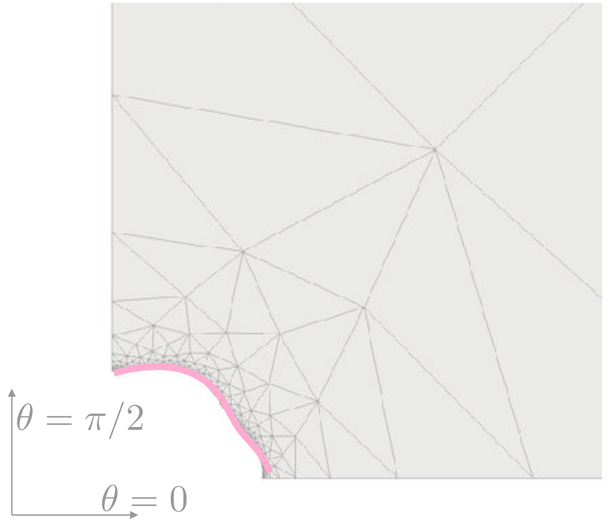
Forward problem: find vertical stress σ_y

linear elasticity PDE

(boundary conditions depend on shape of the hole)



conforming finite element mesh, used in FEniCS



Inverse problem: what shape minimizes the vertical stress?

parameterize the shape of the hole as follows, which automatically enforces a constant area constraint

$$r(\theta) = \frac{1}{2\pi} + \delta \sum_{k=0}^{d/2-1} \frac{1}{\sqrt{2k+1}} \left(x_{2k+1} \sin((2k+1)\cdot\theta) + x_{2k+2} \cos((2k+1)\cdot\theta) \right)$$

optimization variable:

$x \in \mathbb{R}^d$

Generic PDE-constrained optimization

implicitly saying that u solves the PDE

$$\min_{u,x} \mathcal{L}(u) \quad \text{subject to} \quad \phi(u,x) = 0$$

Examples:

$\dot{u} = \Delta u,\ u(0) = h$ "x" is the initial condition

$\ddot{u} = c^2 \Delta u,\ u(0) = h$ "x" is a parameter

$\Delta u = 0,\ u(\Gamma) = h$ "x" is the boundary condition

$\mathcal{L}(u)$ is the loss which penalizes something like:

- deviation from observations
- drag
- mass
- cost of materials
- compliance

etc.

Generic PDE-constrained optimization

implicitly saying that u solves the PDE

$$\min_{u,x} \mathcal{L}(u) \quad \text{subject to} \quad \phi(u,x) = 0$$

$$\phi(u,x) = 0 \quad \Longrightarrow \quad u = u(x)$$

Rewrite: $\min_x f(x) \stackrel{\text{def}}{=} \mathcal{L}(u(x))$

... but finding the gradient is tricky:

$$\nabla f(x) = \frac{\partial \mathcal{L}}{\partial u} \cdot \frac{\partial u}{\partial x}$$

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

- ... so if we can evaluate $f(x)$ numerically, we can find the gradient
- (this applies if $f : \mathbb{R}^d \to \mathbb{R}$; $f : \mathbb{R}^d \to \mathbb{R}^q$ with $q \gg 1$ is another story)

Note: we're assuming derivative exists, just hard to actually calculate

This is *not* non-smooth optimization

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✘ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

`Jax, PyTorch, Tensorflow, …`

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✘ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

✘ Adjoint state method requires a method to solve adjoint PDE

(and have to parallelize for HPC)

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✘ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

✘ Adjoint state method requires a method to solve adjoint PDE

- difficult to maintain in large code bases, e.g., 4D-var for weather codes

✘ Slow if used for intermediate calculations involving some $f : \mathbb{R}^d \to \mathbb{R}^q$

- e.g., seismic inversion with many observations

Why not just find gradients automatically?

✓ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✗ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

✗ Adjoint state method requires a method to solve adjoint PDE

- difficult to maintain in large code bases, e.g., 4D-var for weather codes

✗ Slow if used for intermediate calculations involving some $f : \mathbb{R}^d \to \mathbb{R}^q$

- e.g., seismic inversion with many observations

✗ Possible memory explosion

- e.g., time-dependent problems. Check-pointing schemes somewhat helpful

Example: hyper-parameter optimization in deep learning

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✘ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

✘ Adjoint state method requires a method to solve adjoint PDE

- difficult to maintain in large code bases, e.g., 4D-var for weather codes

✘ Slow if used for intermediate calculations involving some $f : \mathbb{R}^d \to \mathbb{R}^q$

- e.g., seismic inversion with many observations

✘ Possible memory explosion

- e.g., time-dependent problems. Check-pointing schemes somewhat helpful

✘ Requires access to original source code

Why not just find gradients automatically?

✔ The **adjoint state method** and **reverse-mode automatic differentiation** can automatically calculate gradients in about the same time (~4x) as a function evaluation

✘ Requires specialized/restricted libraries/code (`dolfin-adjoint/FEniCS`, `autograd`)

✘ Adjoint state method requires a method to solve adjoint PDE

- difficult to maintain in large code bases, e.g., 4D-var for weather codes

✘ Slow if used for intermediate calculations involving some $f : \mathbb{R}^d \to \mathbb{R}^q$

- e.g., seismic inversion with many observations

✘ Possible memory explosion

- e.g., time-dependent problems. Check-pointing schemes somewhat helpful

✘ Requires access to original source code

✘ Assumes a computational structure

- inapplicable for physical observations (wind farms; *rollout* in AI)

Baseline Algorithms (for comparison)

$f : \mathbb{R}^d \to \mathbb{R}$

These days, often called "zeroth order" optimization

Algorithm Gradient Descent via Finite Differences

1: **for** $k = 1, 2, \ldots$ **do**
2: Estimate $g_k \approx \nabla f(x_k)$ ▷ Use finite differences
3: $x_{k+1} \leftarrow x_k - \eta_k g_k$ ▷ For appropriate step-size η_k

✔ ignoring finite-difference error, enjoys well-understood convergence

✘ requires $d+1$ function evaluations per iter.

Why not use traditional Derivative Free Optimization (DFO) methods?

Answer: most classical DFO methods don't scale well with dimension

Note: there are many other DFO methods... but we won't discuss in this talk

From heuristics to theoretically based, from local to global

- Heuristics: Nelder-Mead, genetic algorithms, particle swarm optimization
  - e.g., CMA-ES, Covariance matrix adaptation evolution strategy
- Simulated Annealing (typical as heuristic)
- Bayesian Optimization
  - Model is Gaussian Process, with acquisition function (exploration/exploitation tradeoff)
- DFO-TR (Trust-region)
  - Model is polynomial

Baseline Algorithms (for comparison)

$f : \mathbb{R}^d \to \mathbb{R}$

Algorithm Gradient Descent via Finite Differences

1: **for** $k = 1, 2, \ldots$ **do**
2: Estimate $g_k \approx \nabla f(x_k)$ ▷ Use finite differences
3: $x_{k+1} \leftarrow x_k - \eta_k g_k$ ▷ For appropriate step-size η_k

✔ ignoring finite-difference error, enjoys well-understood convergence

✘ requires $d+1$ function evaluations per iter.

Algorithm Randomized Coordinate Descent (CD)

1: **for** $k = 1, 2, \ldots$ **do**
2: Choose $j \in \{1, 2, \ldots, d\}$ at random
3: $g_k = e_j e_j^T \nabla f(x_k)$
4: $x_{k+1} \leftarrow x_k - \eta_k g_k$ ▷ For appropriate step-size η_k (or exact minimization… depends on structure)

✔ just 1 function evaluation per iteration

✘ poor convergence properties, slow rates

Stochastic Subspace Descent

directional derivative $qq^T \nabla f(x_k) = \left(\lim_{h \to 0} \frac{f(x_k + h \cdot q) - f(x_k)}{h} \right) q$

Assume we can compute this!

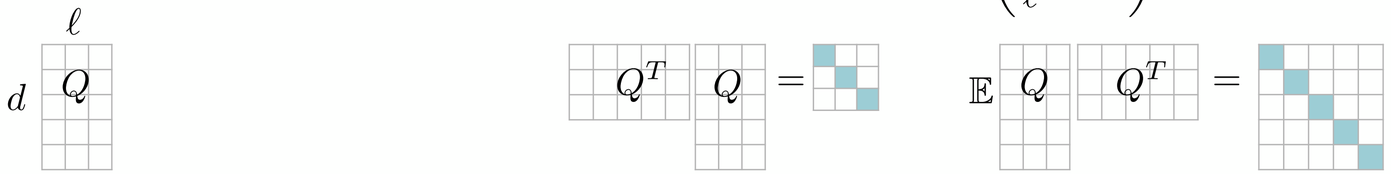
e.g.,

1) forward finite diff
2) forward-mode AD

Stochastic Subspace Descent

directional derivative $qq^T \nabla f(x_k) = \left(\lim_{h \to 0} \frac{f(x_k + h \cdot q) - f(x_k)}{h} \right) q$

$Q = [q_1, q_2, \ldots, q_\ell] \sim \text{Haar}(d \times \ell)$ $\quad Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T \right) = I_{d \times d}$



One benefit: in the limit $\ell = d$, $QQ^T = I_{d \times d}$, and so we'll recover the full gradient (for Gaussians, this is only true in expectation)

Stochastic Subspace Descent

directional derivative $qq^T \nabla f(x_k) = \left(\lim_{h \to 0} \frac{f(x_k + h \cdot q) - f(x_k)}{h} \right) q$

$Q = [q_1, q_2, \ldots, q_\ell] \sim \text{Haar}(d \times \ell)$ $\quad Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T \right) = I_{d \times d}$

Algorithm "Stochastic Subspace Descent" (SSD)

1: **for** $k = 1, 2, \ldots$ **do**
2: Draw $Q \sim \text{Haar}(d \times \ell)$ or *any* generic SSD
3: $x_{k+1} \leftarrow x_k - \eta_k \frac{d}{\ell} Q Q^T \nabla f(x_k)$

Generic SSD $\boxed{Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T \right) = I_{d \times d}}$

Both **Haar** and **Coordinate Descent** methods are valid generic SSD

Stochastic Subspace Descent

directional derivative $qq^T \nabla f(x_k) = \left(\lim_{h \to 0} \frac{f(x_k + h \cdot q) - f(x_k)}{h} \right) q$

$Q = [q_1, q_2, \ldots, q_\ell] \sim \text{Haar}(d \times \ell)$ $\quad Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T \right) = I_{d \times d}$

Algorithm "Stochastic Subspace Descent" (SSD)

1: **for** $k = 1, 2, \ldots$ **do**
2: Draw $Q \sim \text{Haar}(d \times \ell)$
3: $x_{k+1} \leftarrow x_k - \eta_k \frac{d}{\ell} Q Q^T \nabla f(x_k)$

We call Q a "Haar" distributed r.v. (i.e., the Haar measure over orthogonal matrices), but really care about QQ^T which is a projection matrix (onto $\text{col}(Q)$).

We get Q via Gram-Schmidt (or appropriately modified `QR`) on a Gaussian G, and note $\text{col}(Q) = \text{col}(G)$ w.p. 1, so our update is equivalent to

$$x_{k+1} \leftarrow x_k - \eta_k \frac{d}{\ell} \mathcal{P}_{\text{col}(G)} \left(\nabla f(x_k) \right)$$

and hence the term "stochastic subspace".

First theory results (for *generic* SSD)

Theorem (Kozak, Becker, Tenorio, Doostan '20)

Assume: minimizer attained, gradient Lipschitz, stepsize η_k chosen appropriately.

constant L ← η_k ; $\eta = \frac{\ell}{d}\frac{1}{L}$

1. If f is **convex**,

$$\mathbb{E}f(x_k) - f^\star \leq 2\frac{d}{\ell}\frac{L}{k}R^2 = \mathcal{O}(k^{-1})$$

constant μ ← 2. If f is **not convex** but satisfies the **Polyak-Lojasiewicz** inequality,

$$\mathbb{E}f(x_k) - f^\star \leq \rho^k(f(x_0) - f^\star) = \mathcal{O}(\rho^k) \qquad \text{and} \qquad f(x_k) \xrightarrow{\text{a.s.}} f^\star$$

3. If f is **strongly convex**, statements of 2 above hold, and also

$$x_k \xrightarrow{\text{a.s.}} \operatorname{argmin}_x f(x)$$

4. If f is **not convex** (nor PL),

$$\min_{k' \in \{0,\ldots,k\}} \mathbb{E}\|\nabla f(x_{k'})\|^2 \leq \frac{d}{\ell}\frac{2L(f(x_0) - f^\star)}{k+1}$$

SSD is a special type of SGD, but results are much better than generic SGD analysis

$f^\star \stackrel{\text{def}}{=} \min_x f(x)$

$\rho = 1 - \frac{\mu}{L}\frac{\ell}{d}$

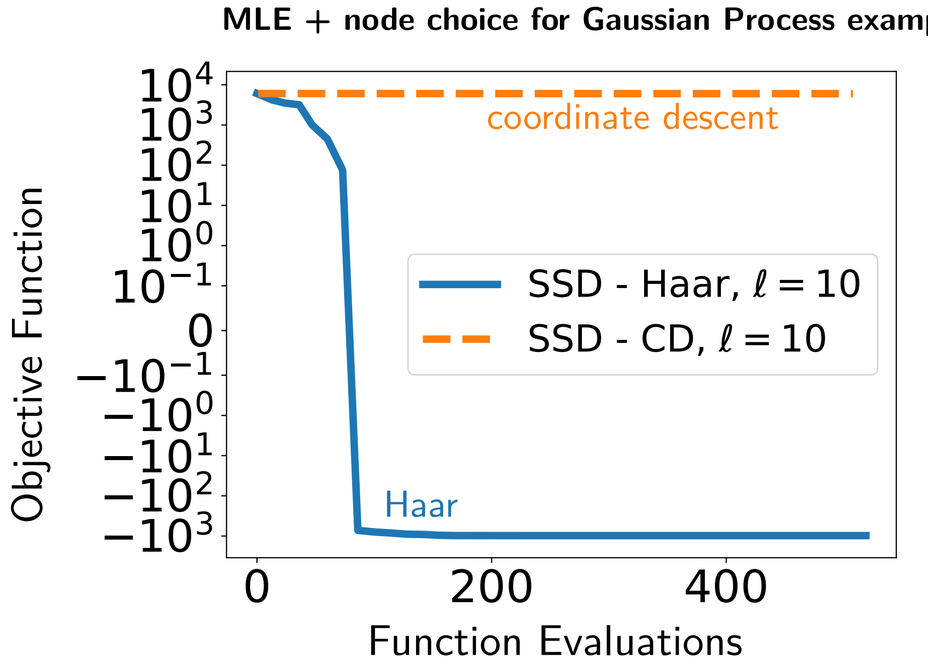
$d =$ ambient dimension

$\ell =$ # directional derivs

Generic SSD $\quad Q^TQ = I_{\ell\times\ell}, \quad \mathbb{E}\left(\frac{d}{\ell}QQ^T\right) = I_{d\times d}$

$\frac{d}{\ell} = 1$ is gradient descent

Numerical Results: better than expected

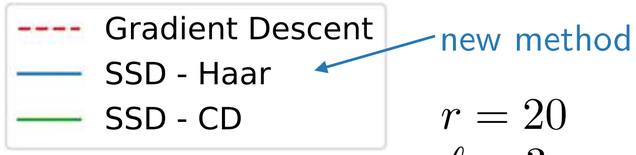


Observation: sometimes SSD (with Haar) drastically outperforms randomized coordinate descent (CD)

Generic SSD $\quad Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T\right) = I_{d \times d}$

Numerical Results: better than expected

Haar SSD drastically outperforms randomized coordinate descent (CD)

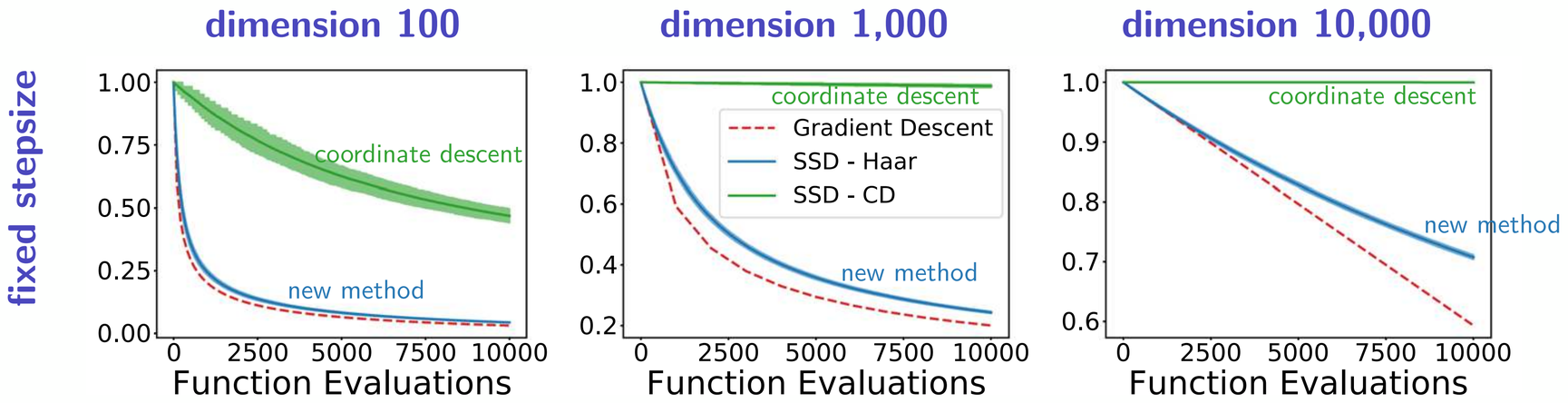


$r = 20$
$\ell = 3$

We can force it to happen by making a problem with low "intrinsic" dimension, e.g., Nesterov's "worst function in the world"

$$f_{\lambda,r}(\mathbf{x}) = \lambda((x_1^2 + \sum_{i=1}^{r-1} (x_i - x_{i+1})^2 + x_r^2)/2 - x_1)/4,$$

This has **intrinsic dimension** of r



Numerical Results: better than expected

Haar SSD drastically outperforms randomized coordinate descent (CD)

Gradient Descent
SSD - Haar
SSD - CD

new method

$r = 20$
$\ell = 3$

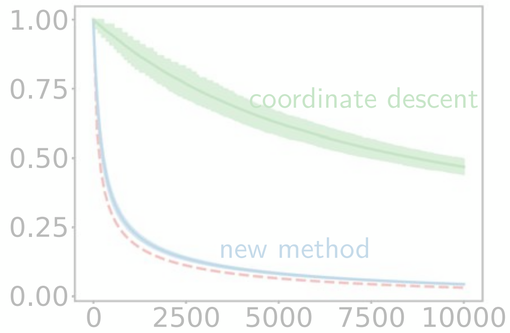
We can force it to happen by making a problem with low "intrinsic" dimension, e.g., Nesterov's "worst function in the world"

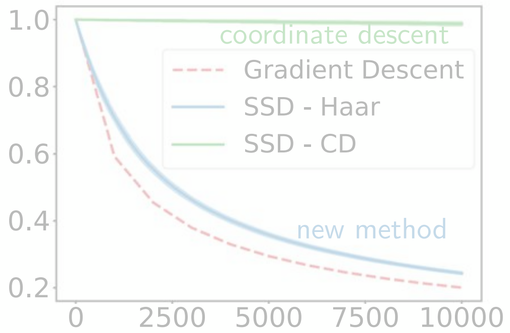
$$f_{\lambda,r}(\mathbf{x}) = \lambda((x_1^2 + \sum_{i=1}^{r-1} (x_i - x_{i+1})^2 + x_r^2)/2 - x_1)/4,$$

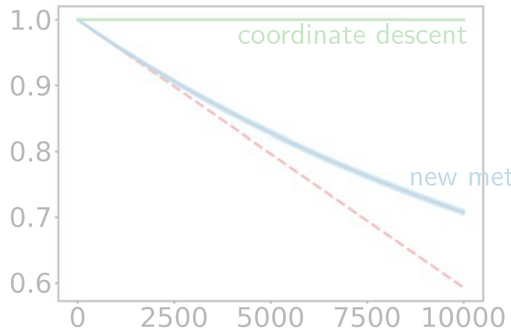
This has **intrinsic dimension** of r

dimension 100 | **dimension 1,000** | **dimension 10,000**

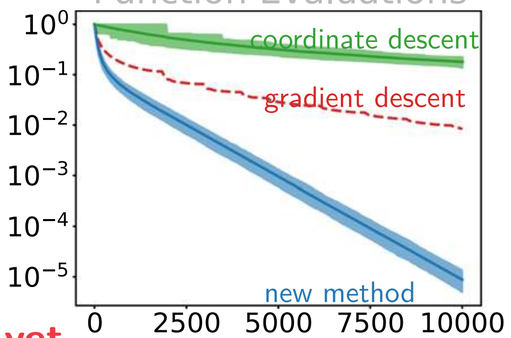
fixed stepsize

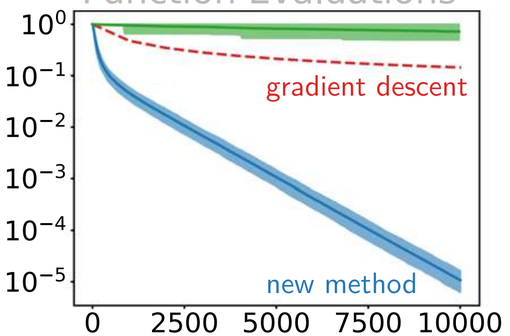


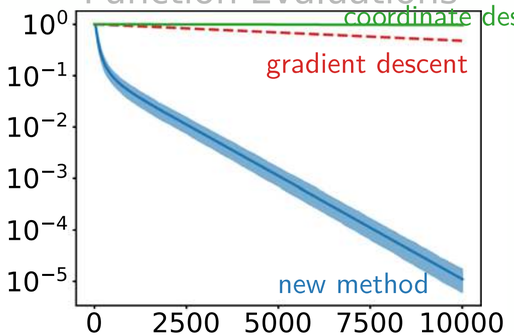




line search







no theory yet

Theory: explain better-than-expected results

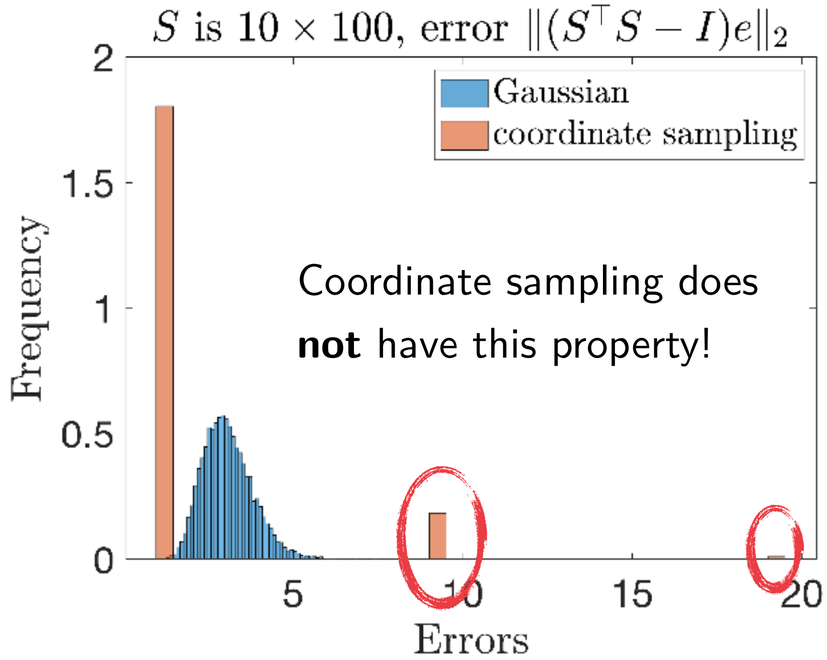
Previous theorem didn't actually rely on properties of Haar distribution, just generic Q:

$$Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T\right) = I_{d \times d}$$

Tighter analysis using concentration-of-measure:

Lemma 2 (Johnson-Lindenstrauss style embedding, from Kozak, Becker, Tenorio '19, Lemma 1)**.** $\forall \epsilon \in (0,1)$, *if* $\ell \gtrsim \epsilon^{-2}$, $Q \sim \text{Haar}(d \times \ell)$, *then* $\forall 0 \neq g \in \mathbb{R}^d$,

$$1 - \epsilon \leq \frac{d}{\ell} \frac{\|Q^T g\|^2}{\|g\|^2} \leq 1 + \epsilon \quad \textit{w/ prob. } \delta \geq 0.8$$



Recall...

$d =$ ambient dimension

$\ell =$ # directional derivs

Theory: explain better-than-expected results

Previous theorem didn't actually rely on properties of Haar distribution, just

$$Q^T Q = I_{\ell \times \ell}, \quad \mathbb{E}\left(\frac{d}{\ell} Q Q^T\right) = I_{d \times d}$$

Tighter analysis using concentration-of-measure:

Lemma 2 (Johnson-Lindenstrauss style embedding, from Kozak, Becker, Tenorio '19, Lemma 1)**.** $\forall \epsilon \in (0,1)$, *if* $\ell \gtrsim \epsilon^{-2}$, $Q \sim \mathrm{Haar}(d \times \ell)$, *then* $\forall 0 \neq g \in \mathbb{R}^d$,

$$1 - \epsilon \leq \frac{d}{\ell} \frac{\|Q^T g\|^2}{\|g\|^2} \leq 1 + \epsilon \quad \textit{w/ prob. } \delta \geq 0.8$$

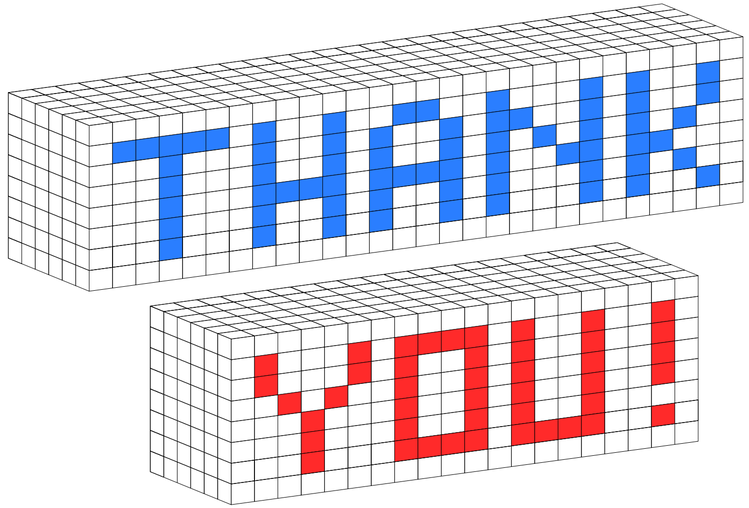
Theorem 3 (Kozak, Becker, Tenorio '19, Thm. 1)**.** *If f is strongly convex and ∇f is Lipschitz continuous, then for an appropriate stepsize η_k, the sequence (x_k) generated by SSD (with $Q \sim$ Haar), for $k > 100$, satisfies*

$$f(x_k) - f^\star \leq (1 + (1-\epsilon)\rho)^{k/2} (f(x_0) - f^\star) \quad \textit{with probability } \geq 0.998,$$

where $\rho < 1$ depends on ℓ, d and the Lipschitz and strong convexity parameters.

due to possibility of failure of JL

error in JL embedding

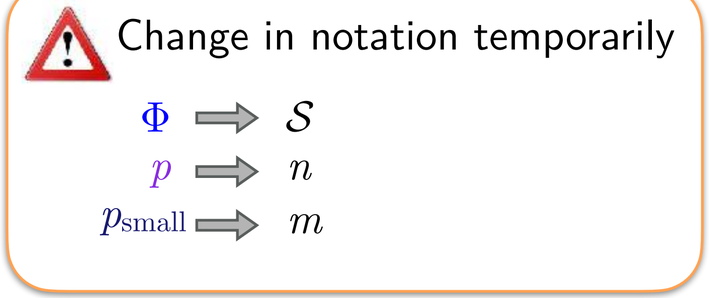


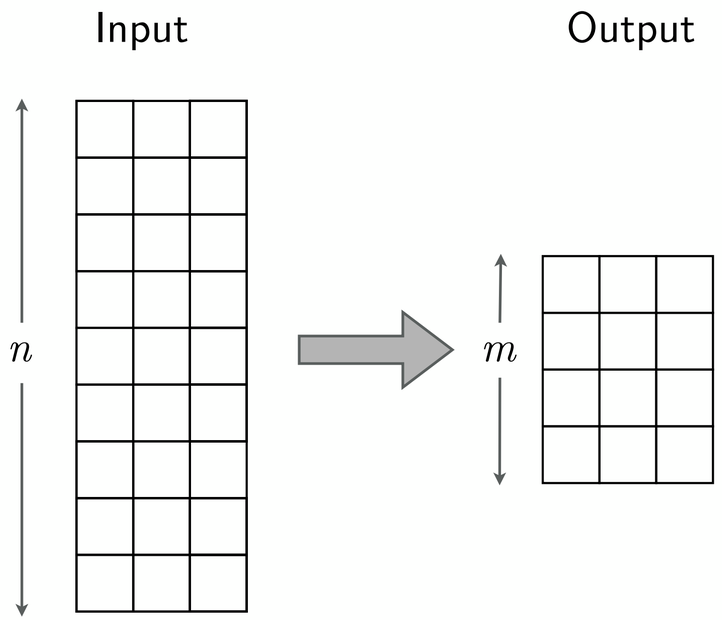
Papers and code available at https://amath.colorado.edu/faculty/becker/

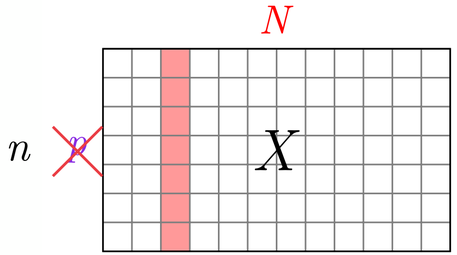
Method 7: CountSketch

Introduced in Charikar et al. (2004), more analysis in, e.g., Clarkson and Woodruff (2017)

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator







Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Notation: $[n] = (0, 1, \ldots, n-1)$

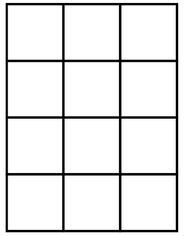
(use 0-based indexing)

Step 1:

Multiply by random sign*

$s : [n] \to \{\pm 1\}$
(uniform)

+1	+1	+1
+1	+1	+1
−1	−1	−1
+1	+1	+1
−1	−1	−1
−1	−1	−1
−1	−1	−1
+1	+1	+1
−1	−1	−1



* technically doesn't have to be fully random, but must be 2-wise independent

$$s(i) \perp s(j) \quad \text{if } i \neq j$$

Method 7: CountSketch

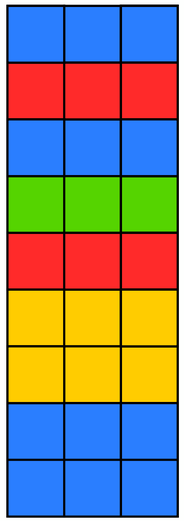
$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

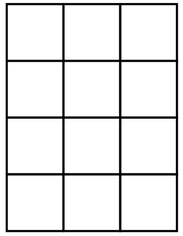
Step 2:

Assign an output row to every input row (randomly or with a hash function)

$h : [n] \to [m]$

(uniform)





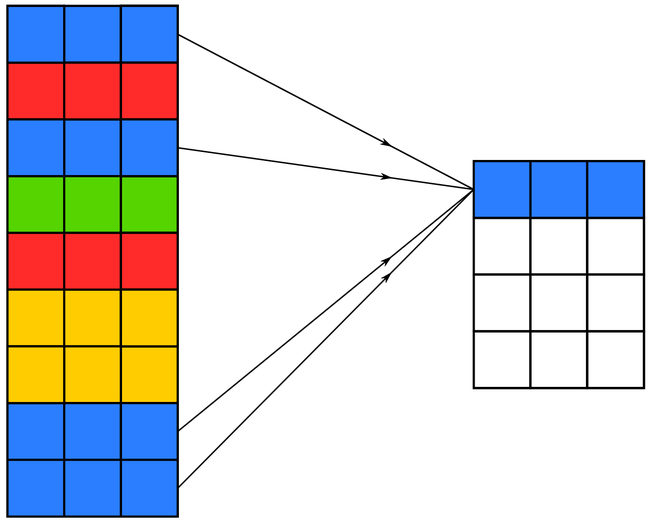
Again, don't need *all* of $h[i]$ to be independent, only pairwise independent (but easy enough to make them all independent)

Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Step 3:

Each output row is the sum of corresponding input rows

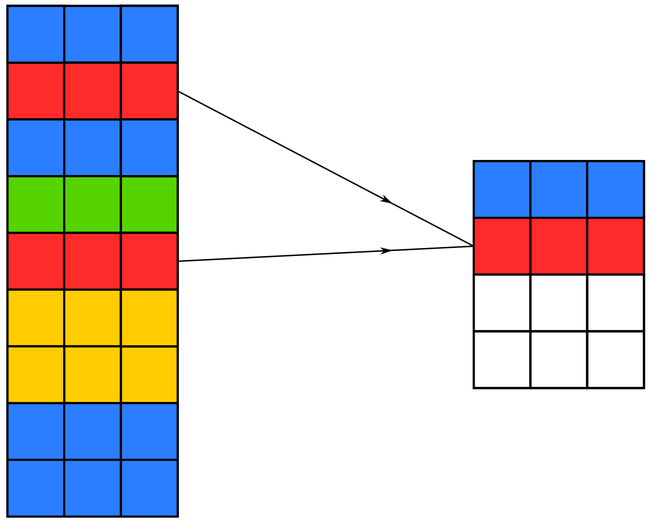


Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Step 3:

Each output row is the sum of corresponding input rows

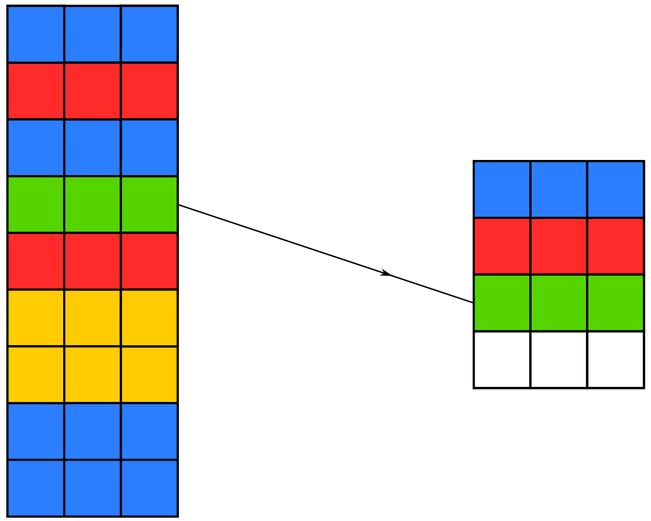


Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Step 3:

Each output row is the sum of corresponding input rows

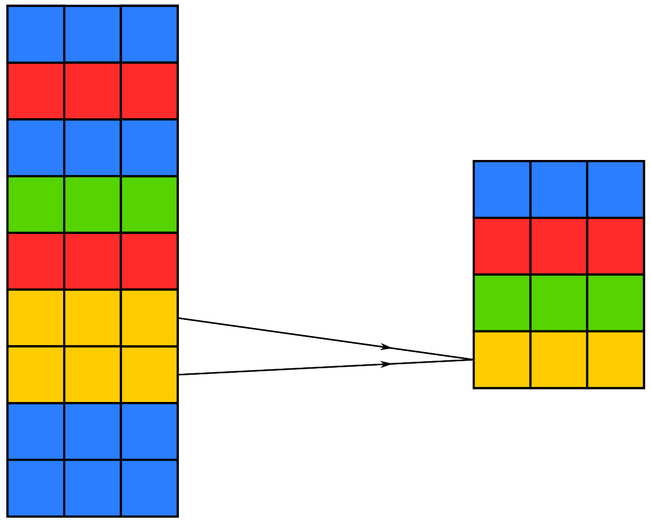


Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Step 3:

Each output row is the sum of corresponding input rows



Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

Complexity analysis: every input element is touched once

... so linear complexity

(and can exploit sparsity)

Method 7: CountSketch

$\mathcal{S} : \mathbb{R}^n \to \mathbb{R}^m$ linear operator

A formula for the output:

$$\boldsymbol{u} = \mathcal{S}(\boldsymbol{v}),\ u_i = \sum_{j|h(j)=i} s(j) v_j$$

$$s : [n] \to \{\pm 1\}$$
$$h : [n] \to [m] \quad \text{hash}$$

... and note that the following polynomial has the output as its coefficients:

$$p(x) \stackrel{\text{def}}{=} \sum_{j \in [n]} s(j) \cdot v_j \cdot x^{h(j)} = \sum_{i \in [m]} x^i \underbrace{\left(\sum_{j|h(j)=i} s(j) v_j \right)}_{u_i} = \sum_{i \in [m]} u_i \cdot x^i$$

(this will be key shortly...)

Method 8: TensorSketch

Introduced in Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

Change in notation temporarily

$\Phi \Rightarrow \mathcal{S}$

$p \Rightarrow n$

$p_{\text{small}} \Rightarrow m$

$$\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \qquad n = n^{(1)} \cdot n^{(2)}$$

Not (yet!) related to tensors

TensorSketch

Introduced in Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

$$\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \qquad n = n^{(1)} \cdot n^{(2)}$$

Kronecker/tensor product

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad \boldsymbol{a} \otimes \boldsymbol{b} = \begin{bmatrix} a_1\boldsymbol{b} \\ a_2\boldsymbol{b} \\ a_3\boldsymbol{b} \end{bmatrix} = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_1b_3 \\ a_2b_1 \\ a_2b_2 \\ a_2b_3 \\ a_3b_1 \\ a_3b_2 \\ a_3b_3 \end{bmatrix}$$

TensorSketch

Introduced in Pagh (2013), more analysis in, e.g., Diao, Zong, Sun, Woodruff (2018)

TensorSketch is just CountSketch when the input can be written as a tensor product (for a special choice of the hash and sign functions)

$$\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$$

$$\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)} \qquad \text{where size is} \qquad n = n^{(1)} \cdot n^{(2)}$$

Kronecker/tensor product

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad \boldsymbol{a} \otimes \boldsymbol{b} = \begin{bmatrix} a_1 \boldsymbol{b} \\ a_2 \boldsymbol{b} \\ a_3 \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_1 b_2 \\ a_1 b_3 \\ a_2 b_1 \\ a_2 b_2 \\ a_2 b_3 \\ a_3 b_1 \\ a_3 b_2 \\ a_3 b_3 \end{bmatrix}$$

another equivalent definition:

$$\boldsymbol{a} \otimes \boldsymbol{b} = \mathrm{vec}_{\mathrm{col}} \left(\boldsymbol{b} \boldsymbol{a}^T \right)$$

$$\boldsymbol{b} \boldsymbol{a}^T = \begin{bmatrix} b_1 a_1 & b_1 a_2 & b_1 a_3 \\ b_2 a_1 & b_2 a_2 & b_2 a_3 \\ b_3 a_1 & b_3 a_2 & b_3 a_3 \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_2 b_1 & a_3 b_1 \\ a_1 b_2 & a_2 b_2 & a_3 b_2 \\ a_1 b_3 & a_2 b_3 & a_3 b_3 \end{bmatrix}$$

$$\mathrm{vec} : [n^{(1)}] \times [n^{(2)}] \to [n^{(1)} \cdot n^{(2)}]$$
$$\mathrm{vec}^{-1} = \mathrm{mat} : [n^{(1)} \cdot n^{(2)}] \to [n^{(1)}] \times [n^{(2)}]$$

more generally,

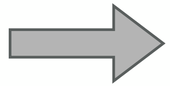
$$(\boldsymbol{A} \otimes \boldsymbol{B}) \, \mathrm{vec}_{\mathrm{col}} (\boldsymbol{X}) = \mathrm{vec}_{\mathrm{col}} \left(\boldsymbol{B} \boldsymbol{X} \boldsymbol{A}^T \right)$$

TensorSketch

Pick hash functions in a decomposable way:

Input has structure $\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)}$ with sizes $n = n^{(1)} \cdot n^{(2)}$

build small hash functions: $h^{(1)} : [n^{(1)}] \to [m]$, $h^{(2)} : [n^{(2)}] \to [m]$



$$h : [n] \to [m]$$

$$h(j_1, j_2) = h^{(1)}(j_1) + h^{(2)}(j_2) \mod m$$

(we're being loose about how we write the input, since we can use the mat/vec bijection as needed)

Fact: as long as $h^{(i)}$ is fully independent (or at least 3-wise independent), then h is 2-wise independent

TensorSketch

Pick hash functions in a decomposable way:

Input has structure $\boldsymbol{v} = \boldsymbol{v}^{(1)} \otimes \boldsymbol{v}^{(2)}$ with sizes $n = n^{(1)} \cdot n^{(2)}$

build small hash functions: $h^{(1)} : [n^{(1)}] \to [m]$, $h^{(2)} : [n^{(2)}] \to [m]$

$$h : [n] \to [m]$$

$$h(j_1, j_2) = h^{(1)}(j_1) + h^{(2)}(j_2) \mod m$$

Fact: h is 2-wise independent

(we're being loose about how we write the input, since we can use the mat/vec bijection as needed)

same trick for sign functions: $s^{(1)} : [n^{(1)}] \to \{\pm 1\}$, $s^{(2)} : [n^{(2)}] \to \{\pm 1\}$

$$s : [n] \to \{\pm 1\}$$

$$s(j_1, j_2) = s^{(1)}(j_1) \cdot s^{(2)}(j_2)$$

still pairwise independent, as needed for theory!

TensorSketch

what's the point? Huge computational speedup.

$$h^{(1)} : [n^{(1)}] \to [m]$$
$$h^{(2)} : [n^{(2)}] \to [m]$$

$$p^{(1)}(x) \stackrel{\text{def}}{=} \sum_{j \in [n^{(1)}]} s^{(1)}(j) \cdot v_j^{(1)} \cdot x^{h^{(1)}(j)} = \sum_{i \in [m]} u_i^{(1)} \cdot x^i$$
$$p^{(2)}(x) \stackrel{\text{def}}{=} \sum_{j \in [n^{(2)}]} s^{(2)}(j) \cdot v_j^{(2)} \cdot x^{h^{(2)}(j)} = \sum_{i \in [m]} u_i^{(2)} \cdot x^i$$

Let's compute this: $p^{(1)}(x) \cdot p^{(2)}(x) \mod x^m - 1$

TensorSketch

what's the point? Huge computational speedup.

$$p^{(1)}(x) \cdot p^{(2)}(x) = \left(\sum_{j_1 \in [n^{(1)}]} s^{(1)}(j_1) \cdot v_{j_1}^{(1)} \cdot x^{h^{(1)}(j_1)} \right) \cdot \left(\sum_{j_2 \in [n^{(2)}]} s^{(2)}(j_2) \cdot v_{j_2}^{(2)} \cdot x^{h^{(2)}(j_2)} \right)$$

TensorSketch

what's the point? Huge computational speedup.

$$p^{(1)}(x) \cdot p^{(2)}(x) = \left(\sum_{j_1 \in [n^{(1)}]} s^{(1)}(j_1) \cdot v_{j_1}^{(1)} \cdot x^{h^{(1)}(j_1)} \right) \cdot \left(\sum_{j_2 \in [n^{(2)}]} s^{(2)}(j_2) \cdot v_{j_2}^{(2)} \cdot x^{h^{(2)}(j_2)} \right)$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} s^{(1)}(j_1) \cdot s^{(2)}(j_2) \cdot v_{j_1}^{(1)} \cdot v_{j_2}^{(2)} \cdot x^{h^{(1)}(j_1)} \cdot x^{h^{(2)}(j_2)}$$

TensorSketch

what's the point? Huge computational speedup.

$$p^{(1)}(x) \cdot p^{(2)}(x) = \left(\sum_{j_1 \in [n^{(1)}]} s^{(1)}(j_1) \cdot v_{j_1}^{(1)} \cdot x^{h^{(1)}(j_1)} \right) \cdot \left(\sum_{j_2 \in [n^{(2)}]} s^{(2)}(j_2) \cdot v_{j_2}^{(2)} \cdot x^{h^{(2)}(j_2)} \right)$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} s^{(1)}(j_1) \cdot s^{(2)}(j_2) \cdot v_{j_1}^{(1)} \cdot v_{j_2}^{(2)} \cdot x^{h^{(1)}(j_1)} \cdot x^{h^{(2)}(j_2)}$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} \mathcal{S}(j_1, j_2) \cdot v_{j_1 j_2} \cdot x^{h^{(1)}(j_1) h^{(2)}(j_2)}$$

because input has tensor product structure

because of how we defined our sketch

TensorSketch

what's the point? Huge computational speedup.

$$p^{(1)}(x) \cdot p^{(2)}(x) = \left(\sum_{j_1 \in [n^{(1)}]} s^{(1)}(j_1) \cdot v^{(1)}_{j_1} \cdot x^{h^{(1)}(j_1)} \right) \cdot \left(\sum_{j_2 \in [n^{(2)}]} s^{(2)}(j_2) \cdot v^{(2)}_{j_2} \cdot x^{h^{(2)}(j_2)} \right)$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} s^{(1)}(j_1) \cdot s^{(2)}(j_2) \cdot v^{(1)}_{j_1} \cdot v^{(2)}_{j_2} \cdot x^{h^{(1)}(j_1)} \cdot x^{h^{(2)}(j_2)}$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} \mathcal{S}(j_1, j_2) \cdot v_{j_1 j_2} \cdot x^{h^{(1)}(j_1) h^{(2)}(j_2)}$$

$$\equiv \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} \mathcal{S}(j_1, j_2) \cdot v_{j_1 j_2} \cdot x^{h(j_1, j_2)} \mod x^m - 1$$

$$x^{k\ell} = x^{dm + (k\ell \mod m)} = x^{dm + (k\ell \mod m)} = (x^m)^d x^{k\ell \mod m} \equiv x^{k\ell \mod m} \mod x^m - 1$$

Recall $x^m \equiv 1 \mod x^m - 1$

TensorSketch

what's the point? Huge computational speedup.

$$p^{(1)}(x) \cdot p^{(2)}(x) = \left(\sum_{j_1 \in [n^{(1)}]} s^{(1)}(j_1) \cdot v_{j_1}^{(1)} \cdot x^{h^{(1)}(j_1)} \right) \cdot \left(\sum_{j_2 \in [n^{(2)}]} s^{(2)}(j_2) \cdot v_{j_2}^{(2)} \cdot x^{h^{(2)}(j_2)} \right)$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} s^{(1)}(j_1) \cdot s^{(2)}(j_2) \cdot v_{j_1}^{(1)} \cdot v_{j_2}^{(2)} \cdot x^{h^{(1)}(j_1)} \cdot x^{h^{(2)}(j_2)}$$

$$= \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} \mathcal{S}(j_1, j_2) \cdot v_{j_1 j_2} \cdot x^{h^{(1)}(j_1) h^{(2)}(j_2)}$$

$$\equiv \sum_{j_1 \in [n^{(1)}], j_2 \in [n^{(2)}]} \mathcal{S}(j_1, j_2) \cdot v_{j_1 j_2} \cdot x^{h(j_1, j_2)} \mod x^m - 1$$

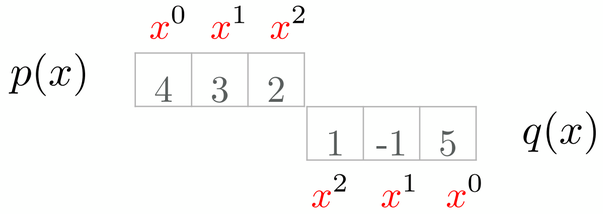
$$= p(x)$$

coefficients are the output of the CountSketch!

TensorSketch

How to multiply polynomials?

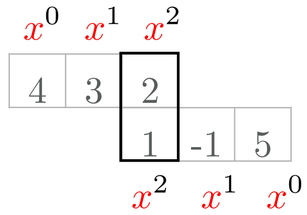
$$\underset{p(x)}{(2x^2 + 3x + 4)} \cdot \underset{q(x)}{(x^2 - x + 5)}$$



TensorSketch

How to multiply polynomials?

$$\left(2x^2 + 3x + 4\right) \cdot \left(x^2 - x + 5\right)$$



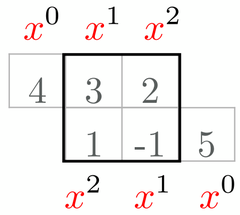
$$= 2x^4$$

$$2 \cdot 1$$

TensorSketch

How to multiply polynomials?

$$\left(2x^2 + 3x + 4\right) \cdot \left(x^2 - x + 5\right)$$



$$= 2x^4 + x^3$$

$$3 \cdot 1 + 2 \cdot (-1) = 1$$

TensorSketch

How to multiply polynomials?

$$(2x^2 + 3x + 4) \cdot (x^2 - x + 5)$$

x^0	x^1	x^2
4	3	2
1	-1	5
x^2	x^1	x^0

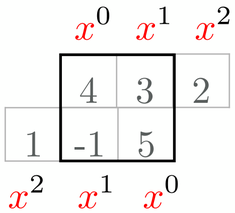
$$= 2x^4 + x^3 + 11x^2$$

$$4 \cdot 1 + 3 \cdot (-1) + 2 \cdot 5 = 11$$

TensorSketch

How to multiply polynomials?

$$\left(2x^2 + 3x + 4\right) \cdot \left(x^2 - x + 5\right)$$



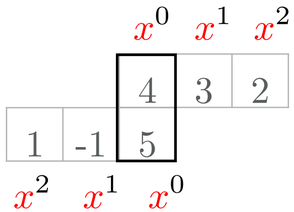
$$= 2x^4 + x^3 + 11x^2 + 11x$$

$$4 \cdot (-1) + 3 \cdot 5 = 11$$

TensorSketch

How to multiply polynomials?

$$\left(2x^2 + 3x + 4\right) \cdot \left(x^2 - x + 5\right)$$



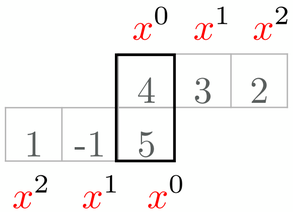
$$= 2x^4 + x^3 + 11x^2 + 11x + 20$$

$$4 \cdot 5 = 20$$

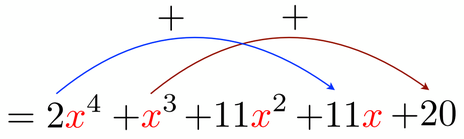
TensorSketch

How to multiply polynomials?

$$(2x^2 + 3x + 4) \cdot (x^2 - x + 5)$$



$$4 \cdot 5 = 20$$



$$\equiv 11x^2 + 13x + 21 \mod x^3 - 1$$

... this is circular convolution!

TensorSketch: complexity analysis

Convolution Theorem (for circular convolutions $*$)

pointwise multiplication

$$\mathcal{F}(g * h) = \mathcal{F}(g)\mathcal{F}(h) \quad \text{i.e.} \quad g * h = \mathcal{F}^{-1}\Big(\mathcal{F}(g)\mathcal{F}(h)\Big)$$

Discrete Fourier Transform (implemented via FFT)

TensorSketch: complexity analysis

Convolution Theorem (for circular convolutions $*$)

$$\mathcal{F}(g * h) = \mathcal{F}(g)\mathcal{F}(h) \quad \text{i.e.} \quad g * h = \mathcal{F}^{-1}\Big(\mathcal{F}(g)\mathcal{F}(h)\Big)$$

Complexity:

$\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$

$n = n^{(1)} \cdot n^{(2)}$

CountSketch applied as TensorSketch: $\mathcal{O}(n^{(1)} + n^{(2)} + m \log m)$ vs CountSketch applied naively: $\mathcal{O}(n^{(1)} n^{(2)})$ vs generic dense matrix multiply: $\mathcal{O}(n^{(1)} n^{(2)} m)$

small CountSketches

polynomial multiplication via FFTs

Savings grow as we have more tensor products

If $n = n^{(1)} n^{(2)} \cdots n^{(q)}$

$$\mathcal{O}(n^{(1)} + n^{(2)} + \ldots + n^{(q)} + m \log m) \quad \text{vs} \quad \mathcal{O}(n^{(1)} n^{(2)} \cdots n^{(q)})$$