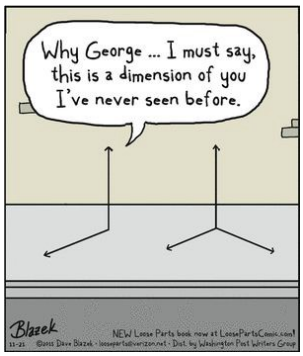
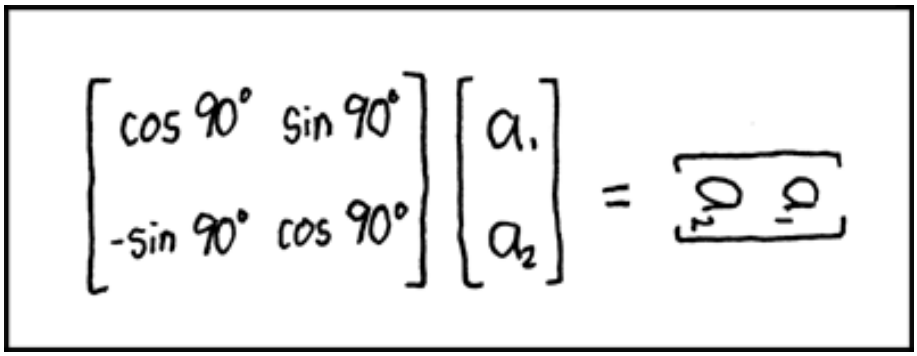
Notes: Computational Linear Algebra





1 Pre-lecture work

Problem 1. (optional) Watch the 3blue1brown's videos on linear algebra:

`https://www.3blue1brown.com/essence-of-linear-algebra-page`

The eigenvalue video is the most important:

`https://www.youtube.com/watch?v=PFDu9oVAE-g`

2 Review: Big-O/Θ/Ω Notation

Definition 1. Let f, g be (possibly multivariate) functions from $(\mathbb{R}^+)^d \to \mathbb{R}^+$, where $d > 0$ is an integer that specifies the number of input variables. Then,

1. If $\lim_{\mathbf{x}\to\infty} \frac{f(\mathbf{x})}{g(\mathbf{x})} < \infty$, then we say $f = O(g)$.
2. If $\lim_{\mathbf{x}\to\infty} \frac{f(\mathbf{x})}{g(\mathbf{x})} > 0$, then we say $f = \Omega(g)$.
3. We say that $f = \Theta(g)$ if both $f = O(g)$ and $f = \Omega(g)$.

Intuitively, you should think of O as $\leq$, Ω as $\geq$, and Θ as $=$.

Problem 2. Complete each equation below by adding the symbol O if $f = O(g)$, Ω if $f = \Omega(g)$, or Θ if $f = \Theta(g)$. The first row is completed for you as an example.

f(n)		g(n)
1	$=$	$O(n)$
$3n \log n$	$=$	n^2
1	$=$	$1/n$
$\log_2 n$	$=$	$\log_3 n$
$\log n$	$=$	$\frac{1}{\log n}$
$5 \cdot 10^{30}$	$=$	$\log n$
$\log n$	$=$	$\log(n^2)$
2^n	$=$	3^n
$\frac{1}{n}$	$=$	$\sqrt{\frac{1}{n}}$
$\log n$	$=$	$(\log n)^2$

Problem 3. Complete each equation below by adding the symbol O if $f = O(g)$, Ω if $f = \Omega(g)$, or Θ if $f = \Theta(g)$. If f cannot be related to g using asymptotic notation, draw a slash through the equals sign. The first row is completed for you as an example.

f(a,b,c)		g(a,b,c)
a^2b	$\neq$	ab^2
$a \log b$	$=$	a^b
a^2bc^3	$=$	$a^2b^2c^3$
ab	$=$	b
$\frac{a}{b}$	$=$	$\frac{a}{b^2}$
$\frac{a}{b}$	$=$	$(\frac{a}{b})^2$
a^b	$=$	b^a
a^b	$=$	$(\log a)^c$
a^b	$=$	$(1+c)^a$

Problem 4. Simplify the following expressions:

1. $O\Big((n^2 + n\log n)(n^3 + \log n)\Big)$

2. $\Omega\Big((3.45n + n)(\log n^2)\Big)$

3. $\Theta\Big(n(1 + \log n) + n^{3.2} + \log 2^n\Big)$

4. $O(ab^2 + 3a^2b + ab + 10b)$

5. $O(a + b + c + ab + bc + abc)$

3 BLAS/LAPACK

The *Basic Linear Algebra Subprogram* (BLAS) and *Linear Algebra Package* (LAPACK) are the primary low-level tools of computational linear algebra. These high performance libraries are written in Assembly, C, and Fortran. All programming languages (R, Python, Matlab, etc.) convert your code into basic BLAS/LAPACK function calls, and rely on these libraries for good performance.

There are many different versions of BLAS/LAPACK depending on:

1. The format of matrices (dense, sparse, diagonal, toplitz, etc.)
2. The computer architecture (Intel vs ARM, 8/16/32/64/128 bit, parallel CPUs, GPUs, clusters, etc.)

In this class, we will be using the PyTorch library, and not directly interfacing with BLAS/LAPACK. It is still important to understand BLAS/LAPACK to understand the performance of your algorithms. PyTorch supports Dense and Sparse formats on both parallel CPU and CUDA architectures.

3.1 Dense vs Sparse Matrices

Read the intro of this wikipedia article:

`https://en.wikipedia.org/wiki/Sparse_matrix`

We will be using $\text{nnz}(A)$ to denote the sparsity of $A : \mathbb{R}^{m \times n}$. If A is stored as a sparse matrix, the memory usage is $\Theta(\text{nnz}(A))$. If A is stored as a dense matrix, the memory usage is $\Theta(mn)$.

Problem 5. Prove or provide a counter example. For both sparse and dense matrices, the storage requirements are $O(mn)$.

Problem 6. A computer has 8GB of RAM. What is the largest dense vector of 64bit floating point numbers that can fit in memory? What about 32bit?

3.2 BLAS Level 1

BLAS Level 1 routines are for vector-vector operations. These include the dot product, vector norms, addition, and scalar multiplication.

These are fully implemented in PyTorch for both sparse and dense vectors. For dense vectors of length n, that is $\Theta(n)$. For sparse vectors, all of operations of a single vector $\mathbf{x}$ take time $\Theta(\text{nnz}(\mathbf{x}))$ and all operations of two vectors $\mathbf{x}$ and $\mathbf{y}$ take time $\Theta(\text{nnz}(\mathbf{x}) + \text{nnz}(\mathbf{y}))$.

Problem 7. What is the runtime of the following expressions:

1. $5 \cdot \mathbf{x}$

2. $0 \cdot \mathbf{x}$

3. $\mathbf{x}^T\mathbf{y}$

4. $\|\mathbf{x}+\mathbf{y}\|_1$

Problem 8. What is the formula for the following vector norms:

1. $\|\mathbf{x}\|_1 =$

2. $\|\mathbf{x}\|_2 =$

3. $\|\mathbf{x}\|_\infty =$

4. $\|\mathbf{x}\|_p =$

3.3 BLAS Level 2

BLAS Level 2 routines are for vector-matrix operations. The main example is the product $A\mathbf{x}$, where $A : \mathbb{R}^{m\times n}$ and $\mathbf{x} : \mathbb{R}^n$. PyTorch fully supports these operations in dense matrices, but has only partial support in sparse matrices. The sparse matrix support is via BLAS Level 3, so we will only discuss that.

3.4 BLAS Level 3

BLAS Level 3 routines are for matrix-matrix operations. Let $A : \mathbb{R}^{m\times n}$ and $B : \mathbb{R}^{n\times o}$. Note that when $o = 1$, then B is a vector, and so BLAS Level 3 operations can be used to implement BLAS Level 2 operations.

For dense matrices, the matrix multiplication AB takes time $O(mno)$. This is often referred to as a "cubic" runtime because when the multiplied matrices are square, and thus $m = n = o$, the runtime is $O(n^3)$. There are good algorithms that take potentially much less time than this, with the best known taking time $O(n^{2.373})$. No BLAS implementation that I know of actually implements this algorithm, and Strassen's algorithm with runtime $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$ is the most commonly used algorithm. Currently, we know that matrix multiplication takes at least time $\Omega(m^2 \log m)$, and it is an open problem what the optimal runtime for matrix multiplication is. The following wikipedia link has more details:

`https://en.wikipedia.org/wiki/Matrix_multiplication#Computational_complexity`

For sparse matrices, PyTorch has only limited support for BLAS Level 2/3. In particular, PyTorch allows the first matrix to be either sparse or dense, but the second matrix must be dense. The runtime in the sparse case is $O(\text{nnz}(A)no)$. The output is always a dense tensor.

For both sparse and dense matrices, the runtime of computing the transpose is $\Theta(1)$.

Problem 9. What is the runtime of the following expressions? (Assume that the dimensions match so that the expressions are well defined.)

1. $A + B$

2. AB

3. $B^T A^T$

4. $A\mathbf{x}$

5. $\mathbf{x}^T\mathbf{x}$

6. $\mathbf{x}\mathbf{x}^T$

7. $A\mathbf{x} + \mathbf{x}$

8. $(A + I)\mathbf{x}$

Problem 10. Many systems that implement sparse matrices offer only limited support in the same way that PyTorch does. The reason is that the sparsity of the matrix product AB is impossible to predict from the sparsity of just A and B. In this problem, you will prove this fact by example. Your task is to construct matrices A and B such that

$$\text{nnz}(AB) = O(1) \qquad \text{and} \qquad \text{nnz}(B^T A^T) = O(mno). \tag{1}$$

Problem 11. (Optional) Strassen's algorithm is a famous divide and conquer algorithm for fast matrix multiplication. It is the most commonly implemented algorithm in BLAS libraries because it has both good constant factors and good asymptotic performance. Lookup this algorithm and understand why it works. This is a problem that every computer science undergrad should do at some point in their undergrad career.

3.5 MCOP

The *Matrix Chain Ordering Problem* (MCOP) is the problem of finding the optimal ordering of parentheses for a given sequence of n matrices. There is a classic dynamic programming solution to this problem that takes time $\Theta(n^3)$. Many more complicated algorithms exist for solving MCOP, the best of which currently takes time $\Theta(n \log n)$. As far as I know, the best lower bound for MCOP is $\Omega(n)$, and it is therefore an open problem whether MCOP can be solved in linear time.

Note: The Θ and Ω notation above is correct. It should be obvious why these values are consistent with each other and do not contradict the definitions of Θ and Ω. If it's not obvious to you, you should ask and get this clarified.

Problem 12. (Optional) Give the dynamic programming solution for MCOP. This is a problem that every computer science undergrad should do at some point in their undergrad career.

Problem 13. Matrix multiplication is famously not commutative, but it is associative. In this problem, we investigate the computational consequences of exploiting this associativity. Let $A : \mathbb{R}^{m \times n}$, $B : \mathbb{R}^{n \times o}$, $C : \mathbb{R}^{o \times p}$.

1. What is the runtime of computing $(AB)C$?

2. What is the runtime of computing $A(BC)$?

3. Under what conditions would you choose the former parenthesization over the latter?

4. Recall that in PyTorch, only the first matrix in a matrix multiplication can be sparse, and the second must be dense. It is still possible to compute the product ABC if one of the matrices is dense and the other two are sparse. For example, if A and B are sparse, then the parenthesization $A(BC)$ can be computed in PyTorch. How should you rewrite the multiplication ABC so that it can be computed when both A and C are sparse?

   Hint: You can swap the order of matrix multiplications by using the fact that for any two matrices X and Y, $XY = (Y^T X^T)^T$.

Problem 14. Parenthesization becomes critical when the matrices are vectors. What is the optimal way to parenthesize the expressions:

1. $\mathbf{x}^T\mathbf{x}\mathbf{x}^T\mathbf{x}$?

2. $\mathbf{x}\mathbf{x}^T\mathbf{x}\mathbf{x}^T$?

Problem 15. Based on your solution to Problem 14 above, what is the asymptotically fastest way to compute the following expression

$$\prod_{i=1}^{n}(\mathbf{x}\mathbf{x}^T) \tag{2}$$

3.6 LAPACK

LAPACK provides more complex matrix operations than BLAS, and LAPACK functions are typically internally written in terms of BLAS operations. PyTorch has full support for dense matrices, and no support for sparse matrices.

Let $A : \mathbb{R}^{n\times n}$. Then the matrix inverse A^{-1} and matrix determinant $\det(A)$ can both be reduced to matrix multiplication problems. The runtime is therefore the same as matrix multiplication, which we talk about as $O(n^3)$ even though $O(n^{2.373})$ algorithms exist. The runtime of computing the top k eigenvectors and eigenvalues is $O(n^2k)$.

Problem 16. Let $\mathbf{x} : \mathbb{R}^n$ and $A : \mathbb{R}^{n\times n}$. What is the formula for the following matrix norms? (Note that the $\|\cdot\|_2$ and $\|\cdot\|_p$ norms are defined differently for matrices than vectors.)

1. $\|A\|_2$

2. $\|A\|_p$

3. $\|A\|_F$

Problem 17. Use the formulas above to calculate the runtime of computing the following matrix norms.

1. $\|A\|_2$

2. $\|A\|_F$

We do not have enough tools yet for calculating the runtime of generic $\|\cdot\|_p$ norms, and that's why I didn't ask you to provide that runtime.

Problem 18. Let $A : \mathbb{R}^{m \times n}$, $\mathbf{x} : \mathbb{R}^n$, and $\lambda : \mathbb{R}$. What is the runtime for computing the following matrix expressions assuming A and $\mathbf{x}$ are dense?

1. $(AA^T)^{-1}A\mathbf{x}$

2. $x^T A^T (A^T A)^{-1}$

3. $AA^T A\mathbf{x} + \lambda \|\mathbf{x}\|_2^2$

4. $\|(\lambda I + A)\mathbf{x}\mathbf{x}^T\|_F$

3.7 (Optional) Tensors

Tensors are higher order versions of vectors and matrices, and PyTorch is designed to handle tensors efficiently. There are many open problems related to computing tensor operations efficiently, but PyTorch implements its tensor operations by "reducing" them to BLAS/LAPACK operations. When you use the `einsum` function, PyTorch compiles your Einstein summation notation into the most efficient possible combination of BLAS/LAPACK operations. That is why I personally always use `einsum` whenever possible, and I recommend you do to. `einsum` is not compatible with sparse tensors.