

# The Tensor Cookbook

Thomas Dybdahl Ahle

August 30, 2024

# Chapter 1

## Introduction

**What is this?** These pages are a guide to tensors, using the visual language of “ensor diagrams”. For illustrating the generality of the approach, I’ve tried to closely follow the legendary “Matrix Cookbook”. As such, most of the presentation is a collection of facts (identities, approximations, inequalities, relations, ...) about tensors and matters relating to them. You won’t find many results not in the original cookbook, but hopefully the diagrams will give you a new way to understand and appreciate them.

**It’s ongoing:** The Matrix Cookbook is a long book, and not all the sections are equally amenable to diagrams. Hence I’ve opted to skip certain sections and shorten others. Perhaps in the future, I, or others, will expand the coverage further.

For example, while we cover all of the results on Expectation of Linear Combinations and Gaussian moments, we skip the section on general multi-variate distributions. I have also had to rearrange the material a bit, to avoid having to introduce all the notation up front.

**Complex Matrices and Covariance** Tensor diagrams (or networks) are currently most often seen in Quantum Physics. Here most values are complex numbers, which introduce some extra complexity. In particular transposing a matrix now involves taking the conjugate (flipping the sign of the imaginary part), which introduces the need for co- and contra-variant tensors. None of this complexity is present with standard real valued matrices, as is common e.g. in Machine Learning applications. For simplicity I have decided to not include these complexities.

**Tensorgrad** The symbolic nature of tensor diagrams make the well suited for symbolic computation.

**Advantages of Tensor Diagram Notation:** Tensor diagram notation has many benefits compared to other notations:

Various operations, such as a trace, tensor product, or tensor contraction can be expressed simply without extra notation. Names of indices and tensors can often be omitted. This saves time and lightens the notation, and is especially useful for internal indices which exist mainly to be summed over. The order of the tensor resulting from

a complicated network of contractions can be determined by inspection: it is just the number of unpaired lines. For example, a tensor network with all lines joined, no matter how complicated, must result in a scalar.

**Etymology** The term "tensor" is rooted in the Latin word *tensio*, meaning "tension" or "stretching," derived from the verb *tendere*, which means "to stretch" or "to extend." It was first introduced in the context of mathematics in the mid-19th century by William Rowan Hamilton in his work on quaternions, where it referred to the magnitude of a quaternion. The modern usage of "tensor" was later established by Gregorio Ricci-Curbastro and Tullio Levi-Civita in their development of tensor calculus, a framework that generalizes the concept of scalars, vectors, and matrices to more complex, multidimensional entities. [1, 2].

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Notation and Nomenclature . . . . .	6
1.2	Tensor Diagrams . . . . .	6
1.3	The Copy Tensor . . . . .	7
1.4	Sums of Tensors . . . . .	8
1.5	Trace . . . . .	9
1.6	Eigenvalues . . . . .	9
<b>2</b>	<b>Simple Derivatives</b>	<b>10</b>
2.1	Derivatives of Matrices, Vectors and Scalar Forms . . . . .	10
2.1.1	First Order . . . . .	10
2.1.2	Second Order . . . . .	11
2.1.3	Higher-order and non-linear . . . . .	11
2.2	Derivatives of Traces . . . . .	12
2.2.1	First Order . . . . .	12
2.2.2	Second Order . . . . .	12
2.2.3	Higher Order . . . . .	13
2.2.4	Other . . . . .	14
<b>3</b>	<b>Statistics and Probability</b>	<b>15</b>
3.0.1	Definition of Moments . . . . .	15
3.0.2	Expectation of Linear Combinations . . . . .	15
3.0.3	Weighted Scalar Variable . . . . .	16
3.0.4	Gaussian Moments . . . . .	16
<b>4</b>	<b>Kronecker and Vec Operator</b>	<b>17</b>
4.1	Flattening . . . . .	17
4.2	The Kronecker Product . . . . .	18
4.3	The Vec Operator . . . . .	19
4.4	General Matrifcation . . . . .	20
4.4.1	The Lyapunov Equation . . . . .	20
4.4.2	Encapsulating Sum . . . . .	20
4.5	The Hadamard Product . . . . .	21
4.6	Khatri–Rao product . . . . .	21
4.6.1	Stacking . . . . .	22
4.7	Stackexchange Problems . . . . .	22

<b>5</b>	<b>Functions</b>	<b>23</b>
5.1	The Chain Rule . . . . .	24
5.1.1	The Chain Rule . . . . .	24
5.1.2	Taylor . . . . .	25
<b>6</b>	<b>Determinant and Inverses</b>	<b>26</b>
6.1	Determinant . . . . .	26
6.2	Inverses . . . . .	27
<b>7</b>	<b>Advanced Derivatives</b>	<b>28</b>
7.1	Derivatives of vector norms . . . . .	28
7.1.1	Two-norm . . . . .	28
7.2	Derivatives of matrix norms . . . . .	29
7.3	Derivatives of Structured Matrices . . . . .	29
7.3.1	Symmetric . . . . .	29
7.3.2	Diagonal . . . . .	29
7.3.3	Toeplitz . . . . .	29
7.4	Derivatives of a Determinant . . . . .	29
7.5	General forms . . . . .	29
7.6	Linear forms . . . . .	29
7.7	Square forms . . . . .	29
7.8	Derivatives of an Inverse . . . . .	29
7.9	Derivatives of Eigenvalues . . . . .	29
<b>8</b>	<b>Special Matrices</b>	<b>30</b>
8.0.1	Block matrices . . . . .	30
8.0.2	The Discrete Fourier Transform Matrix . . . . .	30
8.0.3	Fast Kronecker Multiplication . . . . .	30
8.0.4	Hermitian Matrices and skew-Hermitian . . . . .	34
8.0.5	Idempotent Matrices . . . . .	34
8.0.6	Orthogonal matrices . . . . .	34
8.0.7	Positive Definite and Semi-definite Matrices . . . . .	34
8.0.8	Singleentry Matrix, The . . . . .	34
8.0.9	Symmetric, Skew-symmetric/Antisymmetric . . . . .	34
8.0.10	Toeplitz Matrices . . . . .	34
8.0.11	Units, Permutation and Shift . . . . .	34
8.0.12	Vandermonde Matrices . . . . .	34
<b>9</b>	<b>Decompositions</b>	<b>35</b>
9.1	Higher-order singular value decomposition . . . . .	35
9.2	Rank Decomposition . . . . .	35
9.2.1	Border Rank . . . . .	35
9.3	Fast Matrix Multiplication . . . . .	35

<b>10 Machine Learning Applications</b>	<b>38</b>
10.1 Least Squares . . . . .	38
10.2 Hessian of Cross Entropy Loss . . . . .	38
10.3 Convolutional Neural Networks . . . . .	38
10.4 Transformers / Attention . . . . .	38
10.5 Tensor Sketch . . . . .	38
<b>11 Tensor Algorithms</b>	<b>39</b>
11.1 Optimal Contractions . . . . .	39
<b>12 Tensorgrad</b>	<b>40</b>
12.1 Isomorphisms . . . . .	40
12.1.1 In Products . . . . .	40
12.1.2 In Sums . . . . .	41
12.1.3 In Evaluation . . . . .	42
12.1.4 In Variables . . . . .	42
12.1.5 In Constants . . . . .	42
12.1.6 In Functions . . . . .	43
12.1.7 In Derivatives . . . . .	43
12.1.8 Other . . . . .	43
12.2 Renaming . . . . .	43
12.3 Evaluation . . . . .	43
12.3.1 Products . . . . .	44
12.4 Simplification Rules . . . . .	45
<b>13 Appendix</b>	<b>47</b>

## 1.1 Notation and Nomenclature

$$[P] = \begin{cases} 1 & \text{if } P \\ 0 & \text{otherwise} \end{cases}.$$

## 1.2 Tensor Diagrams

Tensor diagrams are simple graphs (or “networks”) where nodes represent variables (e.g. vectors or matrices) and edges represent contractions (e.g. matrix multiplication or inner products.) The follow table shows how some basic operations can be written with tensor diagrams:

Dot product	$a-b$	$y = \sum_i a_i b_i$	$[\cdot \cdot \cdot \cdot] \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$	$= y$
Outer product	$-a \quad b-$	$Y_{i,j} = a_i b_j$	$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} [\cdot \cdot \cdot \cdot]$	$= -Y-$
Matrix-Vector	$-A-b$	$y_i = \sum_j A_{i,j} b_j$	$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$	$= -y$
Matrix-Matrix	$-A-B-$	$Y_{i,k} = \sum_j A_{i,j} B_{j,k}$	$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$	$= -Y-$

We think of vectors and matrices as tensors of order 1 and 2. The order corresponds to the number of dimensions in their  $[\cdot \cdot \cdot]$  visualization above, e.g. a vector is a 1-dimensional list of numbers, while a matrix is a 2-dimensional grid of numbers. The order also determines the degree of the node representing the variable in the tensor graph.

Diagram notation becomes more interesting when you have tensors of order 3 and higher. An order 3 tensor is a cube or numbers, or stack of matrices. E.g. we can write this as  $T \in \mathbb{R}^{n \times m \times k}$ , so  $T_i \in \mathbb{R}^{m \times k}$  is a matrix for  $i = 1 \dots n$ . Of course we could slice  $T$  along the other axes too, so  $T_{:,j} \in \mathbb{R}^{n \times k}$  and  $T_{::,\ell} \in \mathbb{R}^{n \times m}$  are matrices too.

A matrix having two outgoing edges means there are two ways you can multiply a vector onto it, either on the left:  $x^T M$ , or on the right:  $Mx$ . In graph notation we just write  $x-M-$  and  $-M-x$ . An order 3 tensor has three edges, so we can multiply it with a vector in three ways:

$$\begin{array}{c} | \\ T \\ \swarrow \quad \searrow \\ \quad x \end{array} \quad \text{and} \quad \begin{array}{c} | \\ T \\ \swarrow \quad \searrow \\ x \quad \quad \end{array} \quad \text{and} \quad \begin{array}{c} x \\ | \\ T \\ \swarrow \quad \searrow \\ \quad \quad \end{array}$$

To be perfectly precise about what each one means, we should give the edges labels. For example we would write  $\begin{array}{c} | \\ T \\ \swarrow \quad \searrow \\ \quad x \end{array}$  to specify the matrix  $\sum_i T_i x_i$ . However, often the edge in question will be clear from the context, which is part of what makes tensor diagram notation cleaner than, say, Einstein sum notation.

$$Y_{i,j} = \sum_{k,l,m,n,o} A_{i,k} B_{l,n,o} C_{j,k,l,m} D_{m,n} E_o \Leftrightarrow \begin{array}{c} \diagup \quad \diagdown \\ j \quad i \\ \text{Y} \end{array} = \begin{array}{c} \begin{array}{ccccc} & i & & & \\ & \diagdown & & \diagup & \\ & A & & B & \\ k & | & l & | & n \\ \diagup & & \diagdown & & \diagup \\ j & C & D & E & o \end{array} \end{array}$$

The *key principle* of tensor diagrams is that *edge contraction is associative*. This means you can contract any edge in any order you prefer. This can be seen from the sum representation above, which can be reordered to sum over  $k, l, m, n$  in any order.

The computational price for different contraction orders can be widely different. Unfortunately it's not computationally easy to find the optimal order. See section 11.1 for algorithms to find the best contraction order, and approximate contraction methods.

Note that tensor graphs are not always connected. We already saw that the outer product of two vectors can be written  $-a \ b-$ . This is natural from the sum representation: No edges simply means no sums. So here  $y_{i,j} = a_i b_j$ , which is exactly the outer product  $y = a \otimes b$ .

### 1.3 The Copy Tensor

A particularly important tensor is the “copy” tensor, also known as the “diagonal”, “Kronecker delta” or “spider” tensor. The simplest version is the all-ones vector, which we write as  $\circ-$ . That is  $\circ_i = 1$ . The general order- $n$  tensor is 1 on the diagonal, 0 everywhere else:

$$\circ_{i,j,k,\dots} = \begin{cases} 1 & \text{if } i = j = k = \dots \\ 0 & \text{otherwise} \end{cases}$$

Or, using Iversonian notation,  $\circ_{i,j,k,\dots} = [i = j = k = \dots]$ . We see the order-2 copy-tensor,  $\circ-\circ = I$ , is just the identity matrix, so we can simply remove it from graphs like this:

$$-A-\circ-B- = -A-B-$$

Higher order copy-tensors are very useful, because they let us turn the simple tensor graphs into hyper-graphs. A simple example of how we can use this is the diagonal matrix  $D_a$ , which has  $a$  on the diagonal and 0 elsewhere. We can write this as

$$D_a = \begin{array}{c} | \\ \diagup \quad \diagdown \\ \circ \\ \diagup \quad \diagdown \\ a \end{array}$$

Why? Because  $(D_a)_{i,j} = \sum_k \circ_{i,j,k} a_k = \sum_k [i = j = k] a_k = [i = j] a_i$ . Similarly the Hadamard product,  $(a \circ b)_i = a_i b_i$ , can be written

$$a \circ b = \begin{array}{c} | \\ \diagup \quad \diagdown \\ \circ \\ \diagup \quad \diagdown \\ a \quad b \end{array}$$

Now, let's see why everyone loves copy tensors by using it to prove the identity  $D_a D_b = D_{a \circ b}$  by “copy tensor manipulation”:

$$D_a D_b = \begin{array}{c} | \quad | \\ \diagup \quad \diagdown \quad \diagup \quad \diagdown \\ \circ \quad \circ \\ \diagup \quad \diagdown \quad \diagup \quad \diagdown \\ a \quad b \end{array} = \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \\ \circ \\ \diagup \quad \diagdown \\ a \quad b \end{array} = \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \\ \circ \\ \diagup \quad \diagdown \\ a \quad b \end{array} = D_{a \circ b}.$$



You can verify this using the sum representation.

The general rule at play is that any connected sub-graph of copy-tensors can be combined into a single one. Sometimes we are even lucky enough that this simplification leaves us with an identity matrix we can remove too:

$$S \text{ --- } \text{---} \text{---} \text{---} T = S \text{ --- } \text{---} T = S \text{ ---} T \text{ ---}$$

The only time you have to be a bit careful is when the resulting tensor has order 0. Depending on how you define the order-0 copy tensor,  $\circ$ , you may or may not have the identity  $\circ - \circ = \circ$ .

Lots of other constructions that require special notation (like diagonal matrices or Hadamard products) with normal vector notation can be unified using the copy tensor. In the Matrix Cookbook they define the order-4 tensor  $J$ , which satisfies  $J_{i,j,k,l} = [i = k][j = l]$  and which we'd write as  $J = \begin{smallmatrix} \circ & \circ \\ \circ & \circ \end{smallmatrix}$ , and satisfies, for example,  $\frac{dX}{dX} = J$ . Using “tensor products” you could write  $J = I \otimes I$ . Note that  $J$  is different from the order-4 copy-tensor,  $\text{---}\text{---}\text{---}\text{---}$ .

## 1.4 Sums of Tensors

Tensor products can express any linear function. That is  $f$  such that  $f(ax, by) = abf(x, y)$ . Unfortunately not all operations on tensors are linear. Even something as simple as a sum of two vectors,  $x + y$ , can not be displayed with a simple contraction graph. (Note that this is not linear because  $ax + by \neq ab(x + y)$ .)

To handle this important operation, Penrose suggesting simply writing the two graphs with a plus sign between them, such as  $-x + -y$ . Note that this is itself an order-1 tensor, even though it may look like there are two free edges. If we want to multiply the sum with another tensor, we can use parentheses like  $-M - (-x + -y)$ .

It can be helpful to use named edges when dealing with sums, to make it clear how the edges are matched up. Sums and tensor products interact nicely, with a general form of the distributive law:


$$\left( \begin{smallmatrix} \circ & i \\ R & k \\ U & j \end{smallmatrix} \begin{smallmatrix} T \\ M \end{smallmatrix} + \begin{smallmatrix} \circ & k \\ V & j \end{smallmatrix} \right) = \begin{smallmatrix} \circ & T \\ U & R \end{smallmatrix} M + \begin{smallmatrix} \circ & R \\ V & U \end{smallmatrix}$$

When adding tensors that don't have the same number of edges, or have edges with different names, we can use “broadcasting”. Say we want to add a matrix  $M$  and a vector  $x$ . What does it even mean? If we want to add  $x$  to every row of  $M$ , we write  $\frac{i}{j} M + \frac{i}{j} \circ x$ .

This is because  $\frac{i}{j} \circ x$  is an outer product between  $x$  and the all one vector, which is a matrix in which every row is the same. Similarly, if we want to add  $x$  to every column, we could use the matrix  $\frac{i}{j} x$ .

Note that we typically don't talk about "rows" or "columns" when dealing with tensors, but simply use the name edge (sometimes axis) of the tensor. When using named edges, operations from classical vector notation like "transpose" can also be removed. The matrix  $X^T$  is simply  $X$  where the left and right edge have been swapped. But if the edges are named, we don't have to keep track on "where the edge is" at all.

## 1.5 Trace

The "trace" of a square matrix is defined  $\text{Tr}(A) = \sum_i A_{i,i}$ . In tensor diagram notation, that corresponds to a self-edge: . The Matrix Cookbook has a list of identities using traces. Let's reproduce them with tensor diagrams:

$$\sum_{i=1}^n A_{ii} = \text{Tr}(A) = \text{Tr}(AI) \quad \text{A} = \text{A} \text{---} \circ \quad (11)$$

$$\text{Tr}(A) = \sum_i \lambda_i = \langle 1, \lambda \rangle \quad \text{A} = \overbrace{Q \text{---} \circ \text{---} Q^{-1}}^{\lambda} \quad (12)$$

$$\text{Tr}(A) = \text{Tr}(A^T) \quad \text{A} = \text{A} \quad (13)$$

$$\text{Tr}(AB) = \text{Tr}(BA) \quad \text{A} \text{---} \text{B} = \text{B} \text{---} \text{A} \quad (14)$$

$$\text{Tr}(A + B) = \text{Tr}(A) + \text{Tr}(B) \quad \overbrace{(\text{A} + \text{B}) \text{---} \circ} = \overbrace{\text{A} \text{---} \circ} + \overbrace{\text{B} \text{---} \circ} \quad (15)$$

$$\begin{aligned} \text{Tr}(ABC) &= \text{Tr}(BCA) \\ &= \text{Tr}(CAB) \end{aligned} \quad \begin{aligned} \overbrace{\text{A} \text{---} \text{B} \text{---} \text{C}} &= \overbrace{\text{B} \text{---} \text{C} \text{---} \text{A}} \\ &= \overbrace{\text{C} \text{---} \text{A} \text{---} \text{B}} \end{aligned} \quad (16)$$

$$\begin{aligned} a^T a &= \text{Tr}(aa^T) \\ a \text{---} a &= \text{Tr}(-a \text{---} a \text{---}) \\ &= \overbrace{a \text{---} a} \end{aligned} \quad (17)$$

## 1.6 Eigenvalues

TODO: What else does the Matrix Cookbook include here?  $\text{---} A \text{---} = -Q \text{---} \underset{\lambda}{\circ} \text{---} Q^{-1} \text{---}$  where  $\lambda_i$  is the  $i$ th eigenvalue of  $A$ .

## Chapter 2

# Simple Derivatives

### 2.1 Derivatives of Matrices, Vectors and Scalar Forms

#### 2.1.1 First Order

The Matrix Cookbook defines the single-entry matrix  $J^{i,j} \in R^{n \times n}$  as the matrix which is zero everywhere except in the entry  $(i, j)$  in which it is 1. Alternatively we could write  $J_{n,m}^{i,j} = [i = n][j = m]$ .

$$\frac{\partial x^T a}{\partial x} = \frac{\partial a^T x}{\partial x} = a \quad (x - a) \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} = (x) \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} - a = \nearrow a \quad (69)$$

$$\frac{\partial a^T X b}{\partial X} = ab^T \quad (a - X - b) \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} = a - (X) \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} - b = a - \nearrow b \quad (70)$$

$$\frac{\partial X}{\partial X_{i,j}} = J^{i,j} \quad (-X - \begin{smallmatrix} \nearrow \\ \bullet \\ \nearrow \end{smallmatrix} \begin{smallmatrix} i \\ j \end{smallmatrix}) = -\begin{smallmatrix} \nearrow \\ \bullet \\ \nearrow \end{smallmatrix} \begin{smallmatrix} i \\ j \end{smallmatrix} \quad (73)$$

$$\begin{aligned} \frac{\partial (XA)_{i,j}}{\partial X_{m,n}} &= (J^{m,n} A)_{i,j} & (\begin{smallmatrix} i \\ - \end{smallmatrix} X - A \begin{smallmatrix} \nearrow \\ \bullet \\ \nearrow \end{smallmatrix} \begin{smallmatrix} m \\ n \end{smallmatrix}) &= - (X) \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} - A - \\ & & &= \begin{smallmatrix} i \\ \nearrow \end{smallmatrix} \begin{smallmatrix} m \\ \nearrow \end{smallmatrix} A \begin{smallmatrix} \nearrow \\ \bullet \end{smallmatrix} \end{aligned} \quad (74)$$

## 2.1.2 Second Order

$$\begin{aligned} \frac{\partial}{\partial X_{i,j}} \sum_{k,l,m,n} X_{k,l} X_{m,n} &= (\sum_{k,l} X_{k,l})^2 & \left( \begin{array}{c} \circ - X - \circ \\ \circ - X - \circ \end{array} \right) \begin{array}{c} \nearrow \\ \searrow \end{array} &= \begin{array}{c} \circ - X - \circ \\ \circ - (X) - \circ \end{array} + \begin{array}{c} \circ - (X) - \circ \\ \circ - X - \circ \end{array} \\ &= 2 \sum_{k,l} X_{k,l} & &= 2 \begin{array}{c} \circ - X - \circ \\ \circ - \begin{array}{c} \nearrow \\ \searrow \end{array} - \circ \end{array} \end{aligned} \quad (76)$$

$$\begin{aligned} \frac{\partial b^T X^T X c}{\partial X} &= X(b c^T + c b^T) & (b - X^T - X - c) \begin{array}{c} \nearrow \\ \searrow \end{array} &= b - X^T - (X) \begin{array}{c} \nearrow \\ \searrow \end{array} - c \\ & & &+ b - (X^T) \begin{array}{c} \nearrow \\ \searrow \end{array} - X - c \\ & & &= b - X^T - \begin{array}{c} \nearrow \\ \searrow \end{array} - c \\ & & &+ b - \begin{array}{c} \nearrow \\ \searrow \end{array} - X - c \\ & & &= -X - (-b c' + -c b') \end{aligned} \quad (77)$$

$$\begin{aligned} \frac{\partial}{\partial x} (Bx + b)^T C (Dx + d) &= B^T C (Dx + d) \\ &+ D^T C^T (Bx + b) \end{aligned} \quad (78)$$

$$\begin{aligned} \frac{\partial}{\partial X_{i,j}} (X^T B X)_{k,l} &= \delta_{l,j} (X^T B)_{k,i} & (\begin{array}{c} \circ - X^T - B - X - \circ \end{array}) \begin{array}{c} \nearrow \\ \searrow \end{array} &= \begin{array}{c} \circ - X^T - B - \begin{array}{c} \nearrow \\ \searrow \end{array} - \circ \end{array} \\ &+ \delta_{k,j} (B X)_{i,l} & &+ \begin{array}{c} \circ - \begin{array}{c} \nearrow \\ \searrow \end{array} - B - X - \circ \end{array} \end{aligned} \quad (79)$$

$$\frac{\partial}{\partial X_{i,j}} X^T B X = X^T B J^{i,j} + J^{j,i} B X \quad (\text{same as above}) \quad (80)$$

$$\begin{aligned} \frac{\partial}{\partial x} x^T B x &= (B + B^T) x & (x - B - x) \begin{array}{c} \nearrow \\ \searrow \end{array} &= -B - x + x - B - \\ & & &= x - \begin{array}{c} \nearrow \\ \searrow \end{array} \left( \begin{array}{c} \circ - B - \circ \\ \circ - B - \circ \end{array} \right) \end{aligned} \quad (81)$$

$$\frac{\partial}{\partial X} b^T X^T D X c = D^T X b c^T + D X c b^T \quad \dots \quad (82)$$

$$\frac{\partial}{\partial X} (Xb + c)^T D (Xb + c) = (D + D^T) (Xb + c) b^T \quad \dots \quad (83)$$

Assume  $W$  is symmetric, then... (84) - (88)

## 2.1.3 Higher-order and non-linear

$$\frac{\partial (\mathbf{X}^n)_{kl}}{\partial X_{ij}} = \sum_{r=0}^{n-1} (\mathbf{X}^r \mathbf{J}^{ij} \mathbf{X}^{n-1-r})_{kl}$$

For proof of the above, see B.1.3.

$$\frac{\partial}{\partial \mathbf{X}} \mathbf{a}^T \mathbf{X}^n \mathbf{b} = \sum_{r=0}^{n-1} (\mathbf{X}^r)^T \mathbf{a} \mathbf{b}^T (\mathbf{X}^{n-1-r})^T$$

## 2.2 Derivatives of Traces

Assume  $F(\mathbf{X})$  to be a differentiable function of each of the elements of  $\mathbf{X}$ . It then holds that

$$\frac{d\text{Tr}(F(x))}{dX} = f(X)^T,$$

where  $f(\cdot)$  is the scalar derivative of  $F(\cdot)$ .

TODO: To show this with tensor diagrams, we first need to introduce our notation for functions.

### 2.2.1 First Order

$$\begin{aligned} \frac{\partial}{\partial X} \text{Tr}(X) = I \quad & \left( \overbrace{X} \right) \nearrow = \overbrace{(X)} \searrow \\ & = \curvearrowright \\ & = -\circ- \end{aligned} \quad (99)$$

$$\begin{aligned} \frac{\partial}{\partial X} \text{Tr}(XA) = A^T \quad & \left( \overbrace{X-A} \right) \nearrow = \overbrace{(X)} \searrow - A \\ & \curvearrowright \leftarrow A \\ & = -A^T - \end{aligned} \quad (100)$$

$$\begin{aligned} \frac{\partial}{\partial X} \text{Tr}(AXB) = A^T B^T \quad & \left( \overbrace{A-X-B} \right) \nearrow = \overbrace{A-(X)} \searrow - B \\ & = \overbrace{A} \rightarrow \leftarrow \overbrace{B} \\ & = -A^T - B^T - \end{aligned} \quad (101)$$

Continues for (102-105). The last one uses the Kronecker product, which we may have to introduce first.

### 2.2.2 Second Order

$$\begin{aligned} \frac{\partial}{\partial X} \text{Tr}(X^2) = 2X^T \quad & \left( \overbrace{X-X} \right) \nearrow = \overbrace{(X)} \searrow - X + \overbrace{X-(X)} \searrow \\ & = \curvearrowright \leftarrow X + \overbrace{X} \rightarrow \curvearrowright \\ & = 2 - X^T - \end{aligned}$$

More:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}^2 \mathbf{B}) &= (\mathbf{XB} + \mathbf{BX})^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}^T \mathbf{BX}) &= \mathbf{BX} + \mathbf{B}^T \mathbf{X} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{BXX}^T) &= \mathbf{BX} + \mathbf{B}^T \mathbf{X} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{XX}^T \mathbf{B}) &= \mathbf{BX} + \mathbf{B}^T \mathbf{X} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{XBX}^T) &= \mathbf{XB}^T + \mathbf{XB} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{BX}^T \mathbf{X}) &= \mathbf{XB}^T + \mathbf{XB} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}^T \mathbf{XB}) &= \mathbf{XB}^T + \mathbf{XB} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{AXBX}) &= \mathbf{A}^T \mathbf{X}^T \mathbf{B}^T + \mathbf{B}^T \mathbf{X}^T \mathbf{A}^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}^T \mathbf{X}) &= \frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{XX}^T) = 2\mathbf{X} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{B}^T \mathbf{X}^T \mathbf{CXB}) &= \mathbf{C}^T \mathbf{XBB}^T + \mathbf{CXBB}^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}[\mathbf{X}^T \mathbf{BXC}] &= \mathbf{BXC} + \mathbf{B}^T \mathbf{XC}^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{AXBX}^T \mathbf{C}) &= \mathbf{A}^T \mathbf{C}^T \mathbf{XB}^T + \mathbf{CAXB} \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}[(\mathbf{AXB} + \mathbf{C})(\mathbf{AXB} + \mathbf{C})^T] &= 2\mathbf{A}^T (\mathbf{AXB} + \mathbf{C}) \mathbf{B}^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X} \otimes \mathbf{X}) &= \frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}) \text{Tr}(\mathbf{X}) = 2 \text{Tr}(\mathbf{X}) \mathbf{I}
\end{aligned}$$

### 2.2.3 Higher Order

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{X}^k) &= k (\mathbf{X}^{k-1})^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{AX}^k) &= \sum_{r=0}^{k-1} (\mathbf{X}^r \mathbf{A} \mathbf{X}^{k-r-1})^T \\
\frac{\partial}{\partial \mathbf{X}} \text{Tr}[\mathbf{B}^T \mathbf{X}^T \mathbf{CXX}^T \mathbf{CXB}] &= \mathbf{CXX}^T \mathbf{CXBB}^T \\
&\quad + \mathbf{C}^T \mathbf{XBBB}^T \mathbf{X}^T \mathbf{C}^T \mathbf{X} \\
&\quad + \mathbf{CXBB}^T \mathbf{X}^T \mathbf{CX} \\
&\quad + \mathbf{C}^T \mathbf{XX}^T \mathbf{C}^T \mathbf{XBB}
\end{aligned}$$

**2.2.4 Other**

$$\frac{\partial}{\partial \mathbf{X}} \text{Tr}(\mathbf{A}\mathbf{X}^{-1}\mathbf{B}) = -(\mathbf{X}^{-1}\mathbf{B}\mathbf{A}\mathbf{X}^{-1})^T = -\mathbf{X}^{-T}\mathbf{A}^T\mathbf{B}^T\mathbf{X}^{-T}$$

Assume  $\mathbf{B}$  and  $\mathbf{C}$  to be symmetric, then

$$\begin{aligned} \frac{\partial}{\partial \mathbf{X}} \text{Tr}[(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1}\mathbf{A}] &= -(\mathbf{C}\mathbf{X}(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1})(\mathbf{A} + \mathbf{A}^T)(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1} \\ \frac{\partial}{\partial \mathbf{X}} \text{Tr}[(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{B}\mathbf{X})] &= -2\mathbf{C}\mathbf{X}(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{B}\mathbf{X}(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1} \\ \frac{\partial}{\partial \mathbf{X}} \text{Tr}[(\mathbf{A} + \mathbf{X}^T\mathbf{C}\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{B}\mathbf{X})] &= -2\mathbf{B}\mathbf{X}(\mathbf{X}^T\mathbf{C}\mathbf{X})^{-1} \\ &\quad - 2\mathbf{C}\mathbf{X}(\mathbf{A} + \mathbf{X}^T\mathbf{C}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{B}\mathbf{X}(\mathbf{A} + \mathbf{X}^T\mathbf{C}\mathbf{X})^{-1} \\ &\quad + 2\mathbf{B}\mathbf{X}(\mathbf{A} + \mathbf{X}^T\mathbf{C}\mathbf{X})^{-1} \end{aligned}$$

See [7].

$$\frac{\partial \text{Tr}(\sin(\mathbf{X}))}{\partial \mathbf{X}} = \cos(\mathbf{X})^T$$

## Chapter 3

# Statistics and Probability

### 3.0.1 Definition of Moments

Let  $x \in \mathbb{R}^n$  is a random variable. We write  $m = E[x] \in \mathbb{R}^n$  for the expectation and  $M = \text{Var}[x] = E[(x - m)(x - m)^T]$  for the covariance (when these quantities are defined.)

In tensor diagrams, we will use square brackets:

$$m = [-x] \quad \text{and} \quad M = [-(x \ominus m) \quad (x \div m) -]$$

Note we used the German minus,  $\div$ , to distinguish subtraction from contraction edges.

We can also define the third and fourth centralized moment tensors

$$M_3 = \begin{bmatrix} (x \div m) - \\ (x \div m) - \\ (x \div m) - \end{bmatrix} \quad \text{and} \quad M_4 = \begin{bmatrix} (x \div m) - \\ (x \div m) - \\ (x \div m) - \\ (x \div m) - \end{bmatrix}.$$

### 3.0.2 Expectation of Linear Combinations

General principle: The “linearity of expectation” lets you pull out all parts of the graph not involving  $X$ .

#### Linear Forms

$$E[AXB + C] = AE[X]B + C \quad \begin{bmatrix} -A-X-B- \\ + -C- \end{bmatrix} = \begin{bmatrix} -A-[X]-B- \\ + -C- \end{bmatrix} \quad (312)$$

$$\begin{aligned} \text{Var}[Ax] &= A\text{Var}[x]A^T \\ \begin{bmatrix} A-x \div [A-x] \\ A-x \div [A-x] \end{bmatrix} &= \begin{bmatrix} A-(x \div m) \\ A-(x \div m) \end{bmatrix} \\ &= A \begin{bmatrix} (x \div m) \\ (x \div m) \end{bmatrix} \\ &= -A-M_2-A- \end{aligned} \quad (313)$$



**Quadratic Forms**

$$\begin{aligned}
E[x^T A x] &= \text{Tr}(A \Sigma) + \mu^T A \mu \\
[x - A - x] &= [(x \div \mu) - A - (x \div \mu) + \mu - A - \mu] \\
&= \left[ \begin{array}{c} (x \div m) \\ (x \div m) \end{array} \right] A + \mu - A - \mu \\
&= \overbrace{\Sigma - A} + \mu - A - \mu
\end{aligned}$$

**Cubic Forms****3.0.3 Weighted Scalar Variable**

Let  $y = w^T x$ , and let  $m = E[y]$ , then

$$\begin{aligned}
E[y] &= m = w^T \mu \\
E[(y - m)^2] &= w - M_2 - w \\
E[(y - m)^3] &= w - \overset{w}{M_3} - w \\
E[(y - m)^4] &= w - \overset{w}{M_4} - w
\end{aligned}$$

For specific distributions, like  $x$  Gaussian, we can often reduce the moment tensors further. Khintchine's inequality also gives a way to bound all of these in terms of  $E[(y - m)^2]$ .

**3.0.4 Gaussian Moments**

**Mean and covariance of linear forms**

**Mean and variance of square forms**

**Cubic forms**

**Mean of Quartic Forms**

**Gaussian Integration by Parts**

General principle for Gaussian expectations.

## Chapter 4

# Kronecker and Vec Operator

### 4.1 Flattening

Flattening is a common operation for programmers. In the language of numpy, we may write `np.ones((2,3,4)).reshape(2, 12)` to flatten a shape (2,3,4) tensor into a shape (2,12) matrix. Similarly, in mathematical notation,  $\text{vec}(X)$  is commonly used to denote the flattening of a matrix into a vector.

Typically the main reason to do this is as a cludge for dealing with bad general notation for tensors. Hence, with tensor diagrams, we can avoid this operation entirely. However, it is still interesting to see how tensor diagrams can make a lot of properties of flattening much more transparent.

To begin with we note that flattening is a linear operation, and hence can be represented as a simple tensor. We'll use a triangle to denote this:

$$\triangleright_{i,j,k} = \begin{array}{c} i \\ \diagup \\ \diagdown \\ j \end{array} \triangleright^k = [i + jn = k].$$

Here  $n$  is the dimension of the  $i$  edge. Note we use a double line to denote the output of the flattening operation. This is simply a syntactic choice to remind ourselves that the output is a bundle of two edges.

Using this notation we can write

$$\text{vec}(X)_k = \sum_{i,j} \triangleright_{i,j,k} X_{i,j} = X \begin{array}{c} \text{---} \triangleright^k \\ \text{---} \end{array}.$$

The basic property of  $\triangleright$  is that opposing triangles cancel:

$$\begin{array}{c} \text{and} \end{array} \begin{array}{c} \begin{array}{c} \text{---} \triangleright^k \text{---} \triangleleft^k \text{---} \\ \text{---} \end{array} \\ \begin{array}{c} \triangleleft^k \text{---} \triangleright^k \text{---} \\ \text{---} \end{array} \end{array} = \begin{array}{c} \text{---} \\ \text{---} \end{array}.$$

## 4.2 The Kronecker Product

The Kronecker product of an  $m \times n$  matrix  $A$  and an  $r \times q$  matrix  $B$ , is an  $mr \times nq$  matrix,  $A \otimes B$  defined as

$$A \otimes B = \begin{bmatrix} A_{1,1}B & A_{1,2}B & \cdots & A_{1,n}B \\ A_{2,1}B & A_{2,2}B & \cdots & A_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1}B & A_{m,2}B & \cdots & A_{m,n}B \end{bmatrix}.$$

Using index notation we can also write this as  $(A \otimes B)_{p(r-1)+v, q(s-1)+w} = A_{rs}B_{vw}$ , but it's pretty hard to read.

In tensor notation the Kronecker Product is simply the outer product of two matrices, flattened “on both sides”:  $A \otimes B = \begin{array}{c} \text{A} \\ \text{B} \end{array} \begin{array}{c} \text{ } \\ \text{ } \end{array}$ .

The Kronecker product has the following properties:

$$A \otimes (B + C) = A \otimes B + A \otimes C \quad \begin{array}{c} \text{A} \\ \text{(B+C)} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \end{array} + \begin{array}{c} \text{A} \\ \text{C} \end{array} \quad (506)$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \quad \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \end{array} \quad (508)$$

$$aA \otimes bB = ab(A \otimes B) \quad \begin{array}{c} a \text{ A} \\ b \text{ B} \end{array} = \begin{array}{c} ab \\ \text{A} \\ \text{B} \end{array} \quad (509)$$

$$(A \otimes B)^T = A^T \otimes B^T \quad \begin{array}{c} \text{A} \\ \text{B} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \end{array} \quad (510)$$

$$(A \otimes B)(C \otimes D) = AC \otimes BD \quad \begin{array}{c} \text{A} \\ \text{B} \end{array} \begin{array}{c} \text{C} \\ \text{D} \end{array} = \begin{array}{c} \text{A - C} \\ \text{B - D} \end{array} \quad (511)$$

$$(A \otimes I)(I \otimes B) = A \otimes B \quad \begin{array}{c} \text{A} \\ \text{B} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \end{array} \quad (511b)$$

$$\text{Tr}(A \otimes B) = \text{Tr}(A)\text{Tr}(B) \quad \begin{array}{c} \text{A} \\ \text{B} \end{array} = \begin{array}{c} \text{A} \\ \text{B} \end{array} \quad (515)$$

Here the last equation shows an interesting general property of  $\triangleright$ :

This is easier to see when we consider that  $V = \overline{\text{M} \circ \text{vec}}$  represents the tensor where  $V_{i,j,i,j} = M_{i,j}$  and 0 otherwise. So flattening  $V$  on both sides is the same as  $\text{diag}(\text{vec}(M))$ , which is the rhs of (4.1).

The vec-operator applied on a matrix  $A$  stacks the columns into a vector, i.e. for a  $2 \times 2$  matrix

At the start of the chapter we showed how to represent the `vec`-operator using the flattening tensor:  $\text{vec}(X) = X \begin{array}{c} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \end{array}$ . The Matrix Cookbook gives the following properties of the `vec`-operator:

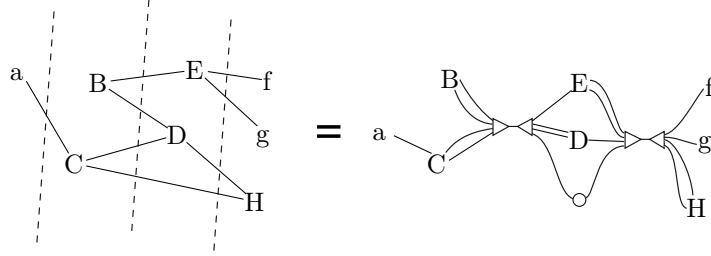
$$\text{vec}(A + B) = \text{vec}(A) + \text{vec}(B) \quad (A+B) \rhd \rhd = A \rhd \rhd + B \rhd \rhd \quad (522)$$

$$\text{vec}(aA) = a \text{vec}(A) \quad aA \rhd = a A \rhd \quad (523)$$

$$\begin{aligned} a^T X B X^T c &= \text{vec}(X)^T (B \otimes ca^T) \text{vec}(X) \quad a-X-B-X-c = X \begin{array}{c} \text{B} \\ \swarrow \quad \searrow \\ a \quad b \end{array} X \quad (524) \\ &= X \rhd \rhd \rhd \rhd \begin{array}{c} \text{B} \\ \swarrow \quad \searrow \\ a \quad b \end{array} X \end{aligned}$$

## 4.4 General Matrifaction

The last equation is an example of a general idea: Any tensor network can be transformed into a series of matrix multiplications by applying the  $\text{vec}$ -operator to all tensors and the flattening tensor to all edges. For example, the following complicated graph:



Can be written as a simple vector-matrix-matrix-vector product,  $aM_1M_2b$ , where  $M_1 = \text{vec}(B) \otimes C'$ ,  $M_2 = E' \otimes D' \otimes I$  and  $b = f \otimes g \otimes \text{vec}(H)$ , where  $C'$ ,  $D'$  and  $E'$  are rank 3 tensors flattened on one side, and  $\text{vec}(B)$  is interpreted as a matrix with a single column.

### 4.4.1 The Lyapunov Equation

A nice application of Kronecker product rewritings is to solve equations like

$$AX + XB = C. \quad (272)$$

We use the rewriting  $\text{vec}(AX + XB) = (I \otimes A + B^T \otimes I)\text{vec}(X)$ , which follows from the tensor diagram massaging:

$$\left( \begin{array}{c} -A-X- \\ + \quad -X-B- \end{array} \right) \rhd = X \begin{array}{c} \text{A} \\ \swarrow \quad \searrow \end{array} \rhd + X \begin{array}{c} \text{B} \\ \swarrow \quad \searrow \end{array} \rhd = X \rhd \left( \begin{array}{c} \text{A} \\ \swarrow \quad \searrow \\ \text{B} \end{array} \right) =$$

after which we can take the normal matrix inverse to get

$$\text{vec}(X) = (I \otimes A + B^T \otimes I)^{-1} \text{vec}(C). \quad (273)$$

### 4.4.2 Encapsulating Sum

This is a generalization of the previous equation.

$$\sum_n A_n X B_n = C \quad (274)$$

$$\text{vec}(X) = \left( \sum_n B_n^T \otimes A_n \right)^{-1} \text{vec}(C) \quad (275)$$

## 4.5 The Hadamard Product

The Hadamard product, also known as element-wise multiplication, is not described in the Matrix Cookbook. Yet, it is a very useful operation, and has some interesting properties in connection with the Kronecker product.

We define the Hadamard product of two  $2 \times 2$  matrices  $A$  and  $B$  as

$$A \circ B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \circ \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{12}B_{12} \\ A_{21}B_{21} & A_{22}B_{22} \end{bmatrix}.$$

In tensor notation, the Hadamard product can be represented using two rank-3 copy tensors:  $- \circ \begin{smallmatrix} A \\ B \end{smallmatrix} \circ -$ . Some properties of the Hadamard product are:

$$x^T (A \circ B) y = \text{tr}(A^T D_x B D_y) \quad x - \circ \begin{smallmatrix} A \\ B \end{smallmatrix} \circ - y = \overbrace{A^T - \circ - B - \circ}^{\substack{x \\ y}}$$

$$(A \otimes B) \circ (C \otimes D) = (A \circ C) \otimes (B \circ D) \quad \circ \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array} \circ = \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array} \circ$$

To see why the last equation holds, it suffices to follow the double lines to see that  $A$  and  $C$  both use the “upper” part of the double edge, while  $B$  and  $D$  use the lower part.

## 4.6 Khatri–Rao product

Also known as the column-wise Kronecker, row-wise Kronecker or “Face-splitting Product”. We use the symbols  $*$  and  $\bullet$  for the column and row-wise Kronecker products, respectively.

$$A * B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{12}B_{12} \\ A_{11}B_{21} & A_{22}B_{22} \\ A_{21}B_{11} & A_{22}B_{12} \\ A_{21}B_{21} & A_{22}B_{22} \end{bmatrix}$$

$$A \bullet B = \dots = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} & A_{12}B_{11} & A_{12}B_{12} \\ A_{21}B_{21} & A_{21}B_{22} & A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

In terms of tensor diagrams, these products correspond simply to flattening the product on one side, and using a copy tensor on the other:

$$A * B = \begin{array}{c} \text{A} \\ \text{B} \end{array} \circ -$$

$$A \bullet B = - \circ \begin{array}{c} \text{A} \\ \text{B} \end{array}$$

Clearly the two are identical up to transpose. Indeed,  $(A * B)^T = B^T \bullet A^T$  and  $(A \bullet B)^T = B^T * A^T$ .

There are multiple “mixed product” identities:

$$(A \bullet B)(C \otimes D) = (AC) \bullet (BD) \quad - \circ \begin{array}{c} A \\ \diagdown \diagup \\ B \end{array} \begin{array}{c} C \\ \diagdown \diagup \\ D \end{array} = - \circ \begin{array}{c} A - C \\ \diagdown \diagup \\ B - D \end{array}$$

$$(Ax) \circ (By) = (A \bullet C)(x \otimes y) \quad - \circ \begin{array}{c} A - x \\ \diagdown \diagup \\ B - y \end{array} = - \circ \begin{array}{c} A \\ \diagdown \diagup \\ B \end{array} \begin{array}{c} x \\ \diagdown \diagup \\ y \end{array}$$

### 4.6.1 Stacking

Can be part of Kronecker section

From Josh: Proposition 2.5. For any field  $\mathbb{F}$ , integers  $d_1, d_2, d_3, d_4$  and matrices  $X_1 \in \mathbb{F}^{d_1 \times d_2}$ ,  $X_2 \in \mathbb{F}^{d_2 \times d_3}$ ,  $X_3 \in \mathbb{F}^{d_1 \times d_4}$ , and  $X_4 \in \mathbb{F}^{d_4 \times d_3}$ , we have

$$X_1 \times X_2 + X_3 \times X_4 = (X_1 \mid X_3) \times \begin{pmatrix} X_2 \\ X_4 \end{pmatrix},$$

where we are writing  $\mid$  to denote matrix concatenation.

With tensor diagrams we can write stacking along a new axis  $i$  as

$$\text{stack}_i(X, Y) = \begin{array}{c} e^{(0)-i} \\ \text{---} X \text{---} \end{array} + \begin{array}{c} e^{(1)-i} \\ \text{---} Y \text{---} \end{array}$$

where  $e_i^{(i)} = 1$  and 0 elsewhere.

Fro this we easily get the identity

$$(A \mid C) \begin{pmatrix} B \\ D \end{pmatrix} = \text{stack}_i(A, C) \text{stack}_i(B, D) \quad (4.2)$$

$$= \left( \begin{array}{c} e^{(0)-i} \\ \text{---} A \text{---} \end{array} + \begin{array}{c} e^{(1)-i} \\ \text{---} C \text{---} \end{array} \right) \left( \begin{array}{c} e^{(0)-i} \\ \text{---} B \text{---} \end{array} + \begin{array}{c} e^{(1)-i} \\ \text{---} D \text{---} \end{array} \right) \quad (4.3)$$

$$= \begin{array}{c} e^{(0)-} e^{(0)} \\ \text{---} A \text{---} B \text{---} \end{array} + \begin{array}{c} e^{(1)-} e^{(1)} \\ \text{---} C \text{---} D \text{---} \end{array} \quad (4.4)$$

$$= AB + CD \quad (4.5)$$

TODO: Relation to direct sum, which is basically stacking + flattening. Or maybe it's nicer to do it by hadamard producting with the  $e_i$  vector. Also notice that this is what quantum comp people call “controlling”.

Also have a bunch of properties like  $A \otimes (B \oplus C) = A \otimes B \oplus A \otimes C$ .

## 4.7 Stackexchange Problems

## Chapter 5

# Functions

Function to scalar	$f : \mathbb{R}^n \rightarrow \mathbb{R}$	$f(x) \in \mathbb{R}$	$\{f \leftarrow x\}$
Function to vector	$g : \mathbb{R}^n \rightarrow \mathbb{R}^m$	$g(x) \in \mathbb{R}^m$	$\{g' \leftarrow x\}$
Element-wise function	$h : \mathbb{R}^n \rightarrow \mathbb{R}$	$h(x) \in \mathbb{R}^m$	$\{h \quad x'\}$
Vector times vector function	$u : \mathbb{R}^n \rightarrow \mathbb{R}^m, v \in \mathbb{R}^m$	$v^T u(x) \in \mathbb{R}$	$\overbrace{v \quad \{u \leftarrow x\}}$
Vector times matrix function	$A : \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}, v \in \mathbb{R}^n$	$A(x)v \in \mathbb{R}^m$	$\overbrace{\{A \leftarrow x\}^m} v$
Batch function application	$f : \mathbb{R}^d \rightarrow \mathbb{R}, X \in \mathbb{R}^{b \times d}$	$f(X) \in \mathbb{R}^b$	$\{f \leftarrow X\}^b$

As an example of a more complicated function, let  $\exp : \mathbb{R} \rightarrow \mathbb{R}$  be the element-wise exponential function, and  $\text{pow}^{-1} : \mathbb{R} \rightarrow \mathbb{R}$  be the element-wise inverse power function. Then we can write  $\text{softmax} : \mathbb{R}^d \rightarrow \mathbb{R}^d$  as the tensor diagram:

$$\text{softmax}(x) = \{ \exp \quad x' \} \quad \{ \text{pow}^{-1} \quad \{ \exp \quad x' \} \circ \}.$$

Note in particular how  $\{ \exp \quad x' \} \circ$  is the diagram way of writing  $\sum_i \exp(x_i)$ . Alternatively we could have used a function  $\text{sum} : \mathbb{R}^n \rightarrow \mathbb{R}$ , but the more we can express in terms of tensor diagrams, the more we can use the powerful tensor diagram calculus.

Stuff about analytical matrix functions. Such as Exponential Matrix Function. I'd rather talk about my general function notation. And maybe about Taylor series?



## 5.1 The Chain Rule

Sometimes the objective is to find the derivative of a matrix which is a function of another matrix.

E.g.  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$

Standard chain rule. Here we let  $f \in \mathbb{R}^d \rightarrow \mathbb{R}$  be a scalar function, and  $v \in \mathbb{R}^d \rightarrow \mathbb{R}^d$  be a vector function as used in backprop.

Visualization of the Chain Rule:  $J_{f \circ v}(x) = \nabla f(v(x)) J_v(x)$ .

$$\left( \left\{ \overleftarrow{f} \{ v \leftarrow x \} \right\} \right) \cdot \bullet = \left\{ \overleftarrow{f} \bullet \{ v \leftarrow x \} \right\} \left\{ v \bullet \leftarrow x \right\}$$

### 5.1.1 The Chain Rule

Let  $f \in \mathbb{R}^d \rightarrow \mathbb{R}$  be a scalar function, and  $v \in \mathbb{R}^d \rightarrow \mathbb{R}^d$  be a vector function, as used in backprop. Then we can write the chain rule as:

Using standard notation:  $J_{f \circ v}(x) = \nabla f(v(x)) J_v(x)$ .

The second derivative, (or Hessian Chain rule):

Using standard notation:  $H_{f \circ v}(x) = Dv(x)^T \cdot D^2 f(v(x)) \cdot Dv(x) + \sum_{k=1}^d \frac{\partial f}{\partial u_k}(v(x)) \frac{\partial^2 v_k}{\partial x \partial x^T}(x)$ .

$$\frac{\partial A(x)}{\partial x} = (x^T \otimes I) \frac{\partial}{\partial x} \text{vec}[A(x)] + A(x) \quad \left( x \left\{ \overleftarrow{A} \leftarrow x \right\} \right) \cdot \bullet = x \left( \left\{ \overleftarrow{A} \leftarrow x \right\} \right) \cdot \bullet + \left( x \right) \cdot \bullet \left\{ \overleftarrow{A} \leftarrow x \right\} \\ = x \left\{ \overleftarrow{A} \leftarrow x \right\} \cdot \bullet + \left\{ \overleftarrow{A} \leftarrow x \right\}$$

are common. All pixel-adaptive filters like non-local means, bilateral, etc, and the so-called attention mechanism in transformers can be written this way

Gradient of this  $f(x)$  is important & has a form worth remembering...

**Pseudo-linear form**

Maybe this should just be an example in a table?

Derivation of Peyman Milanfar's gradient

$$\begin{aligned}
 d[\mathbf{f}(\mathbf{x})] &= d[\mathbf{A}(\mathbf{x})\mathbf{x}] \\
 &= d[\mathbf{A}(\mathbf{x})]\mathbf{x} + \mathbf{A}(\mathbf{x})d\mathbf{x} \\
 &= \text{vec}\{d[\mathbf{A}(\mathbf{x})]\mathbf{x}\} + \mathbf{A}(\mathbf{x})d\mathbf{x} \\
 &= \text{vec}\{\mathbf{Id}[\mathbf{A}(\mathbf{x})]\mathbf{x}\} + \mathbf{A}(\mathbf{x})d\mathbf{x} \\
 &= (\mathbf{x}^T \otimes \mathbf{I}) \text{vec}\{d[\mathbf{A}(\mathbf{x})]\} + \mathbf{A}(\mathbf{x})d\mathbf{x} \\
 &= (\mathbf{x}^T \otimes \mathbf{I}) D \text{vec}[\mathbf{A}(\mathbf{x})]d\mathbf{x} + \mathbf{A}(\mathbf{x})d\mathbf{x} \\
 &= [(\mathbf{x}^T \otimes \mathbf{I}) D \text{vec}[\mathbf{A}(\mathbf{x})] + \mathbf{A}(\mathbf{x})] d\mathbf{x}
 \end{aligned}$$

**5.1.2 Taylor**

For an n-times differentiable function  $v : \mathbb{R}^d \rightarrow \mathbb{R}^d$  we can write the Taylor expansion:

$$\begin{aligned}
 v(x + \varepsilon) &\approx v(x) + \left[ \frac{\partial}{\partial x} v(x) \right] \varepsilon + \frac{1}{2} \left[ \frac{\partial}{\partial x} \left[ \frac{\partial}{\partial x} v(x) \right] \varepsilon \right] \varepsilon + \frac{1}{6} \left[ \frac{\partial}{\partial x} \left[ \frac{\partial}{\partial x} \left[ \frac{\partial}{\partial x} v(x) \right] \varepsilon \right] \varepsilon \right] \varepsilon + \dots \\
 &= v(x) + \left[ \frac{\partial}{\partial x} v(x) \right] \varepsilon + \frac{1}{2} (I \otimes \varepsilon) \left[ \frac{\partial \text{vec}}{\partial x} \left[ \frac{\partial v(x)}{\partial x} \right] \right] \varepsilon + \frac{1}{6} (I \otimes \varepsilon \otimes \varepsilon) \left[ \frac{\partial \text{vec}}{\partial x} \left[ \frac{\partial \text{vec}}{\partial x} \left[ \frac{\partial v(x)}{\partial x} \right] \right] \right] \varepsilon + \dots
 \end{aligned}$$

Or with indices:

$$v_i(x + \varepsilon) \approx v_i(x) + \sum_j \frac{\partial v_i(x)}{\partial x_j} \varepsilon_j + \frac{1}{2} \sum_{j,k} \frac{\partial^2 v_i(x)}{\partial x_j \partial x_k} \varepsilon_j \varepsilon_k + \frac{1}{6} \sum_{j,k,\ell} \frac{\partial^3 v_i(x)}{\partial x_j \partial x_k \partial x_\ell} \varepsilon_j \varepsilon_k \varepsilon_\ell$$

Or diagrams:

$$\begin{array}{c} v \\ \uparrow \\ (x + \varepsilon) \end{array} \approx \begin{array}{c} v \\ \uparrow \\ x \end{array} + \begin{array}{c} \varepsilon \\ \bullet \\ \uparrow \\ x \end{array} \begin{array}{c} \circ \\ v \end{array} + \frac{1}{2} \begin{array}{c} \varepsilon \\ \bullet \\ \uparrow \\ x \end{array} \begin{array}{c} \circ \\ \bullet \\ v \end{array} + \frac{1}{6} \begin{array}{c} \varepsilon \\ \bullet \\ \uparrow \\ x \end{array} \begin{array}{c} \circ \\ \bullet \\ \bullet \\ v \end{array} + \dots$$

TODO: Examples based on idempotent matrices etc.

## Chapter 6

# Determinant and Inverses

### 6.1 Determinant

It's convenient to write the determinant in tensor notation as

$$\det(A) = \frac{1}{n!} \overline{\overline{A \cdots A}}$$

where  $\overline{\overline{i_1 \ i_2 \ \cdots \ i_n}} = \varepsilon_{i_1, \dots, i_n}$  is the rank- $n$  Levi-Civita tensor defined by

$$\varepsilon_{i_1, \dots, i_n} = \begin{cases} \text{sign}(\sigma) & \sigma = (i_1, \dots, i_n) \text{ is a permutation} \\ 0 & \text{otherwise.} \end{cases}$$

To see that the definition makes sense, let's first consider

$$\det(I) = \frac{1}{n!} \overline{\overline{\begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}}} = \frac{1}{n!} \sum_{i_1, \dots, i_n, j_1, \dots, j_n} \varepsilon_{i_1, \dots, i_n} \varepsilon_{j_1, \dots, j_n} [i = j] = \frac{1}{n!} \sum_{i_1, \dots, i_n} \varepsilon_{i_1, \dots, i_n}^2 = 1.$$

In general we get from the permutation definition of the determinant:

$$\begin{aligned} \overline{\overline{A \cdots A}} &= \sum_{i_1, \dots, i_n, j_1, \dots, j_n} \varepsilon_{i_1, \dots, i_n} \varepsilon_{j_1, \dots, j_n} A_{i_1, j_1} \cdots A_{i_n, j_n} \\ &= \sum_{\sigma, \tau} \text{sign}(\sigma) \text{sign}(\tau) A_{\sigma_1, \tau_1} \cdots A_{\sigma_n, \tau_n} \\ &= \sum_{\sigma} \text{sign}(\sigma) \sum_{\tau} \text{sign}(\tau) A_{\sigma_1, \tau_1} \cdots A_{\sigma_n, \tau_n} \\ &= \sum_{\sigma} \text{sign}(\sigma)^2 \det(A) \\ &= n! \det(A). \end{aligned}$$

The definition generalizes to Cayley's "hyper determinants" by . . . .

A curious property is that

$$\begin{array}{|c|} \hline A \\ \hline \end{array} \cdots \begin{array}{|c|} \hline A \\ \hline \end{array} = \begin{array}{|c|} \hline \cdots \\ \hline \end{array} \begin{array}{|c|} \hline A \\ \hline \end{array} \cdots \begin{array}{|c|} \hline A \\ \hline \end{array}$$

$$(18) \quad \det(A) = \prod_i \lambda_i \quad \dots$$

$$(19) \quad \det(cA) = c^n \det(A) \quad \begin{array}{|c|} \hline cA \\ \hline \end{array} \cdots \begin{array}{|c|} \hline cA \\ \hline \end{array} = c^n \begin{array}{|c|} \hline A \\ \hline \end{array} \cdots \begin{array}{|c|} \hline A \\ \hline \end{array}$$

$$(20) \quad \det(A) = \det(A^T) \quad \dots$$

$$(21) \quad \det(AB) = \det(A)\det(B) \quad \begin{array}{|c|} \hline A \\ \hline \end{array} \cdots \begin{array}{|c|} \hline A \\ \hline \end{array} \begin{array}{|c|} \hline B \\ \hline \end{array} \cdots \begin{array}{|c|} \hline B \\ \hline \end{array} = \begin{array}{|c|} \hline A \\ \hline \end{array} \cdots \begin{array}{|c|} \hline A \\ \hline \end{array} \begin{array}{|c|} \hline B \\ \hline \end{array} \cdots \begin{array}{|c|} \hline B \\ \hline \end{array}$$

$$(22) \quad \det(A^{-1}) = 1/\det(A) \quad \dots$$

$$(23) \quad \det(A^n) = \det(A)^n \quad \dots$$

$$(24) \quad \det(I + uv^T) = 1 + u^T v \quad \dots$$

## 6.2 Inverses

Might be reduced, unless cofactor matrices have a nice representation?

## Chapter 7

# Advanced Derivatives

### 7.1 Derivatives of vector norms

#### 7.1.1 Two-norm

$$\frac{d}{dx} \|x - a\|_2 = \frac{x - a}{\|x - a\|_2} \quad (7.1)$$

$$\frac{d}{dx} \frac{x - a}{\|x - a\|_2} = \frac{I}{\|x - a\|_2} - \frac{(x - a)(x - a)^T}{\|x - a\|_2^3} \quad (7.2)$$

$$\frac{d}{dx} \|x\|_2^2 = \frac{d}{dx} \|x^T x\|_2 = 2x \quad (7.3)$$

## 7.2 Derivatives of matrix norms

## 7.3 Derivatives of Structured Matrices

### 7.3.1 Symmetric

### 7.3.2 Diagonal

### 7.3.3 Toeplitz

## 7.4 Derivatives of a Determinant

## 7.5 General forms

## 7.6 Linear forms

## 7.7 Square forms

## 7.8 Derivatives of an Inverse

## 7.9 Derivatives of Eigenvalues

# Chapter 8

## Special Matrices

### 8.0.1 Block matrices

Stuff like Schur complements is interesting. But can we say anything useful using tensor diagrams?

### 8.0.2 The Discrete Fourier Transform Matrix

I think FFT can be nicely described with diagrams

Let's start with the Hadamard matrix:  $H_n = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n}$ . Hm. It's just a bunch of matrices below each other, kinda boring.

What about the FFT? Does that require a bit more?

### 8.0.3 Fast Kronecker Multiplication

Say we want to compute  $(A_1 \otimes A_2 \cdots A_n)x$ , where  $A_i$  is a  $a_i \times a_i$  matrix, and  $x \in \mathbb{R}^{a_1 a_2 \cdots a_n}$ . If we first compute the Kronecker product, and then the matrix-vector multiplication, this would take  $(a_1 \cdots a_n)^2$  time.

Instead we can reshape  $x$  into a  $a_1 \times \cdots a_n$  tensor and perform the multiplication

$$\begin{array}{c} \overset{a_1}{\text{---}} A_1 \searrow \\ \overset{a_2}{\text{---}} A_2 \text{---} \\ \vdots \nearrow \\ \underset{a_n}{\text{---}} A_n \end{array} X$$

by contracting the  $a_i$  edges one by one. This takes time

$$a_1^2(a_2 \cdots a_n) + a_2^2(a_1 a_3 \cdots a_n) + \cdots + a_n^2(a_1 \cdots a_{n-1}) = (a_1 + \cdots + a_n)(a_1 \cdots a_n),$$

which is the basis of many fast algorithm as we will see.

The Hadamard matrix is defined as  $H_{2^n} = H_2^{\otimes n} = H_2 \otimes \cdots \otimes H_2$  where  $H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ . For example

$$H_4 = H_2 \otimes H_2 = \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

$$H_{2^n} = \begin{array}{c} \text{---} \triangleleft \begin{array}{c} H_2 \\ \vdots \\ H_7 \end{array} \triangleright \text{---} \end{array}.$$
$$H_{2^n}x = \begin{bmatrix} H_{2^{n-1}} & H_{2^{n-1}} \\ H_{2^{n-1}} & -H_{2^{n-1}} \end{bmatrix} \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix},$$

Alternatively we could just use the general fact, as described above, where  $a_i = 2$  for all  $i$ . Then the “fast Kronecker multiplication” method takes time  $(a_1 a_2 \cdots a_n)(a_1 + a_2 + \cdots a_n) = 2n \log_2 n$ .

The Discrete Fourier Matrix is defined by  $(F_N)_{i,j} = \omega^{ij}$ , where  $\omega = e^{-2\pi i/N}$ :

$$F_N = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}.$$

$$= \left\{ \exp \frac{2\pi i}{N} \begin{matrix} \text{arange}(N) \\ \vdots \\ \text{arange}(N) \end{matrix} \right\}$$

TODO: Show how the matrix can be written with the function notation.



The Good-Thomas Fast Fourier Transformer (FFT) uses a decomposition based on the Chinese Remainder Theorem:

$$F_N = \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{P_1} \\ \vdots \\ \boxed{P_1} \end{array} \begin{array}{c} F_{p_1^{i_1}} \\ F_{p_2^{i_2}} \\ \vdots \\ F_{p_n^{i_n}} \end{array} \begin{array}{c} \boxed{P_2} \\ \vdots \\ \boxed{P_2} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array},$$

where  $N = p_1^{i_1} p_2^{i_2} \cdots p_n^{i_n}$  is the prime factorisation of  $N$ , and  $P_1$  and  $P_2$  are some permutation matrices.

Using fast Kronecker multiplication, the algorithm this takes  $(p_1^{i_1} + \cdots + p_n^{i_n})N$  time. By padding  $x$  with zeros, we can increase  $N$  by a constant factor to get a string of  $n = O(\log(N)/\log \log(N))$  primes, the sum of which is  $\sim n^2/\log n = O(\log(N)^2)$ . The complete algorithm thus takes time  $O(N \log(N)^2)$ . Next we will see how to reduce this to  $O(N \log N)$ .

The classical Cooley-Tukey FFT algorithm uses a recursion:

$$F_N = \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & D_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} \begin{bmatrix} \text{even-odd} \\ \text{permutation} \end{bmatrix},$$

where  $D_N = [1, w^N, w^{2N}, \dots]$ . The even-odd permutation moves all the even values to the start. If we reshape  $I_{2^n}$  as  $I_2 \otimes \cdots \otimes I_2$ , this permutation is just  $P_N = \text{---}$ , or in pytorch: `x.permute([3,0,1,2])`. Also note that  $\begin{bmatrix} I & I \\ I & -I \end{bmatrix} = H_2 \otimes I$  and  $\begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} = I_2 \otimes F_{N/2}$ . So we can write in tensor diagram notation:

$$F_N = \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{H_2} \\ \vdots \\ \boxed{H_2} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{F_{N/2}} \\ \vdots \\ \boxed{F_{N/2}} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} = \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{H_2} \\ \vdots \\ \boxed{H_2} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{H_2} \\ \vdots \\ \boxed{H_2} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{H_2} \\ \vdots \\ \boxed{H_2} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$

$D_N \qquad D_{N/2^0} \quad D_{N/2^1} \quad D_{N/2^2}$

Since one can multiply with the permutation and diagonal matrices in linear time, the  $O(n \log n)$  time complexity follows from the same argument as for Hadamard.

Note there are a bunch of symmetries, such as by transposing (horizontal flip), since the matrix is symmetric. Or by pushing the permutation to the left side.

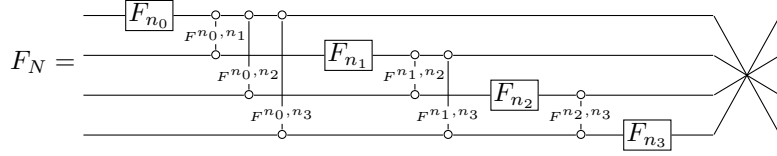
We don't have to split the matrix in half, we can also split it in thirds, fourths, etc. With this generalized Cooley-Tukey algorithm, we get the following diagram:

$$F_N = \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{F_{n_0}} \\ \vdots \\ \boxed{F_{n_0}} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{F_{n_1}} \\ \vdots \\ \boxed{F_{n_1}} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{F_{n_2}} \\ \vdots \\ \boxed{F_{n_2}} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{F_{n_3}} \\ \vdots \\ \boxed{F_{n_3}} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$

$F^{(n_0, n_1 n_2 n_3)} F^{(n_1, n_2 n_3)} F^{(n_2, n_3)}$

where  $n_0 n_1 n_2 n_3 = N$ . Here we replaced the  $D_N$  matrix with the “generalized” Fourier matrix  $F^{(a,b)}$  matrix, which is defined as  $F_{j,k}^{(a,b)} = e^{-2\pi i j k / (ab)}$ , and we reshaped as necessary. In the simple case of where we split in just two parts of roughly  $n_1 = n_2 = \sqrt{N}$ , this is also called “Four step FFT” or Bailey’s FFT algorithm.

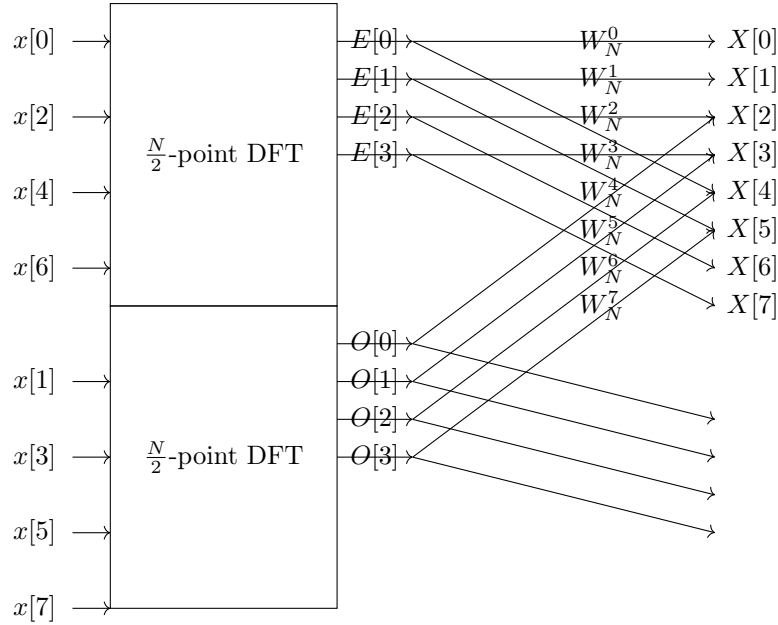
We can use the property  $F_{a,bc} = F_{a,b} \bullet F_{a,c}^{1/b}$  to simplify the diagram further:



In the simple case where we split in 2 every time, this is also called the “Quantum FFT” algorithm.

We hid some stuff above, namely that the matrices should be divided by different  $N$ s.

Note that this figure may look different from some FFT diagrams you have seen. These typically look like this:



and have  $2^n$  rows. The tensor diagram only has  $n$  rows (or  $\log_2 N$ ).

### Multi-dimensional Fourier Transform

This is just taking the Fourier transform along each axis.

**8.0.4 Hermitian Matrices and skew-Hermitian**

Complex. Skip

**8.0.5 Idempotent Matrices**

Skip

**8.0.6 Orthogonal matrices**

Skip

**8.0.7 Positive Definite and Semi-definite Matrices**

Skip

**8.0.8 Singleentry Matrix, The**

Describes the matrix  $J$ . All of this is trivial with diagrams.

**8.0.9 Symmetric, Skew-symmetric/Antisymmetric**

Could introduce Penrose's symmetric tensors here?

**8.0.10 Toeplitz Matrices**

Could talk about the convolution tensor here...

**8.0.11 Units, Permutation and Shift**

Not that interesting...

**8.0.12 Vandermonde Matrices**

Does this have a nice description? Not a lot of properties are given in the Cookbook.

## Chapter 9

# Decompositions

### 9.1 Higher-order singular value decomposition

Say we have an order  $n$  tensor  $A$ . We “unfold”  $A$  along each dimension. This means pulling the edge  $i$  to the left, and flattening the rest to the right. Then we compute the SVD,  $USV$ . Here  $U$  is a square matrix, which we keep. We multiply the  $i$ th edge of  $A$  by  $U^T$  (which is also the inverse of  $U$ ). The result is a “core” tensor as well as a sequence of  $U$  tensors. If we want a more compact SVD, we can make each  $U$  low rank, like normal SVD. There is also the “Interlacing computation” where we multiply the  $U^T$ s onto  $A$  as we go along.

For order 3 tensors, this method is called a “Tucker decomposition”.

If the “core matrix” is diagonal, this is called tensor rank decomposition. If we were good at that, we could use it to factor  $I^{\otimes 3}$  to get better matrix multiplication algorithms. Unfortunately tensor rank decomposition is NP hard.

I guess HOSVD gives a rank decomposition if we diagonalize the core tensor. It just won’t be an efficient one.

### 9.2 Rank Decomposition

#### 9.2.1 Border Rank

The border rank of a tensor is the smallest rank of a tensor that is close to it. TODO: Example where the border rank is much smaller than the rank.

### 9.3 Fast Matrix Multiplication

Strassen defines 3 tensors of shape  $7 \times 2 \times 2$ :

$$S_A = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \end{bmatrix}$$
$$S_B = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}, & \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \end{bmatrix}$$

$$W = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, & \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

These tensors have the neat property that they factor  $I_2 \otimes I_2 \otimes I_2$ :

To multiply two matrices,  $A$  and  $B$ , faster than the normal  $n^3$  time, we reshape them as block matrices, shape  $(2, \frac{n}{2}, 2, \frac{n}{2})$  and use Strassen's tensor:

$$= A = B = \text{Diagram of } A \text{ and } B \text{ as block matrices} = \text{Diagram of } W \text{ tensor contraction}.$$

Contracting the edges in the right order, uses only  $7/8n^3 + O(n^2)$  operations.

If we instead reshape to  $(2, 2, \dots, 2)$ ,

$$AB = \text{Diagram of } A \text{ and } B \text{ as block matrices},$$

and using Strassen's tensor along each axis reduces the work by  $(7/8)^{\log_2(n)}$ , giving us matrix multiplication in time  $n^{3+\log_2(7/8)} = n^{2.80735}$ .

Contracting the double edges,  $S_A - A$  and  $S_B - B$ , is both  $O(n^2)$  time.

It remains to verify that this is actually faster than the naive matrix multiplication: Contracting  $S_A - A$  takes  $7 \cdot 2^2(n/2)^2$  operations, and likewise  $S_B - B$ . Next we contract  $S_A A - S_B B$  which takes  $7(n/2)^3$  time. And finally we contract the edge with  $W$  which takes  $2^2 \cdot 7(n/2)^2$ . The important term is the cubic  $7/8n^3$ , which if instead done recursively, leads to the classical  $O(n^{\log_2 7})$  algorithm.

FIXME: What “edge with  $W$ ”? I think we have to/want to contract the hyperedge with  $W$  immediately?

### Other

If we instead wrote  $A$  and  $B$  using  $(n, m)$  and  $(m, p)$  shaped blocks, we could factor  $I_n \otimes I_m \otimes I_p$  and get a matrix multiplication algorithm using the same approach as the Strassen  $(2, 2, 2)$  tensors above. Lots of papers have investigated this problem, which has led to the best algorithms by Josh Alman and others. For example, Deep Mind found a rank 47 factorization of  $I_3 \otimes I_4 \otimes I_5$ .

Maybe a more interesting example is the  $(4, 4, 4)$  tensor, for which they find a rank 47 factorization. This an easy way to create a rank 49 is to take Strassen and double it. Would this be a nice thing to show? Maybe too messy? Well, actually their rank 47 construction only works in the “modular” case. Then  $(3, 4, 5)$  is general.

## Chapter 10

# Machine Learning Applications

10.1 Least Squares

10.2 Hessian of Cross Entropy Loss

10.3 Convolutional Neural Networks

10.4 Transformers / Attention

10.5 Tensor Sketch

## Chapter 11

# Tensor Algorithms

### 11.1 Optimal Contractions



# Chapter 12

## Tensorgrad

Implementation details

### 12.1 Isomorphisms

There is actually a concept of “tensor isomorphism”, but it’s basically just the same as graph isomorphism.

We need to understand isomorphisms in many different parts of the code.

#### 12.1.1 In Products

- Cancelling / combining equal parts of a product This is actually extra hard, because you have to collect a subset of nodes that constitute isomorphic subgraphs. Right now we hack this a bit by just considering separate components of the product.

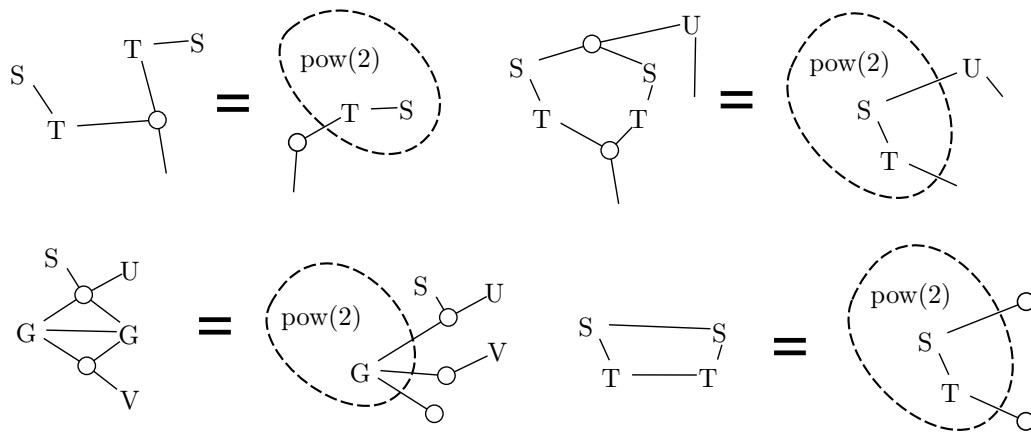


Figure 12.1: Combining equal parts of a product

Basically the problem is:

1. You are given a multigraph  $G$  with nodes  $V$  and edges  $E$ .
2. Nodes and edges are all labeled.
3. You are to find two disjoint subsets  $V_1$  and  $V_2$  of  $V$  such that the subgraphs  $G_1$  and  $G_2$  induced by  $V_1$  and  $V_2$  are isomorphic. Also, under the isomorphism, the labels of the nodes and edges in  $G_1$  and  $G_2$  are the same.

The problem is probably NP-hard, but it might still have an algorithm that's faster than  $2^n$  trying all subsets. In particular, we might modify the VF2 algorithm, which iteratively tries to match nodes in  $G_1$  and  $G_2$ . The NetworkX library already has a GraphMatcher, which searches for isomorphic subgraphs. It might be extendable to our problem... But honestly I don't know if we even want to solve this problem in the most general, since it corresponds a bit to factoring the graph. And we don't do factoring, just as we don't do inverse distribution.

In either case, it's clear that we need to be able to compare nodes and edges for isomorphism.

Also, the basic usecase of isomorphism canonaization in products is simply to compute the canonical product itself from its parts. Part of our approach here is taking the outer edges and turning them into nodes, so they can be colored.

### 12.1.2 In Sums

When deciding whether  $A + B$  is equal to  $2A$  we need to check if  $A$  and  $B$  are isomorphic. But we also need to do this under the current renaming of the edges. That's why you can't just transform  $A + A^T = 2A$ .

The way it actually works in my code is

```

1 def key_fn(t: Tensor):
2     # Align tensor edges to have the same order, using Sum's order as
      reference.
3     canons = [t.canonical_edge_names[t.edges.index(e)] for e in self.edges]
4     return hash((t.canon,) + tuple(canons))
5
6 ws_tensors = TensorDict(key_fn=key_fn, default_fn=int)
7 for w, t in zip(weights, tensors):
8     ws_tensors[t] += w
9 ws_tensors = [(w, t) for t, w in ws_tensors.items()]

```

which says that I'm using for a hash, the canonical form of the tensor, plus the canonical form of the edges in the order of the edges in the sum. These are basically the orbits, meaning that if the summed tensor has a symmetry, we are allowed to "flip" it to make the summands isomorphic.

In the "compute canonical" method, we do more or less the same, but we also include the weights.

```

1 def _compute_canonical(self):
2     hashes = []
3     for e in self.edges:
4         canons = [t.canonical_edge_names[t.edges.index(e)] for t in self.
      tensors]
5         hashes.append(hash(("Sum",) + tuple(sorted(zip(self.weights, canons)
      )))))

```

```

6     base = hash(("Sum", len(self.tensors)))
7     hashes = [hash((base, h)) for h in hashes]
8     return base, hashes

```

In the future we want to use symmetry groups instead. What would be the symmetry group of a sum? It's the diagonal of the product of the symmetry groups of the summands. How can we find the generators of this group? Maybe we should just construct some joint graph and then find the automorphisms of that graph.

Alternatively we can use sympy. It is not known whether this problem is solvable in polynomial time. I think Babai proved that it is quasi-polynomial but not with a practical algorithm. Incidentally the problems of intersections of subgroups, centralizers of elements, and stabilizers of subsets of  $\{1, \dots, n\}$  have been proved (by Eugene Luks) to be polynomially equivalent.

Actually making a graph and using nauty is a really good idea, since it would be able to detect that  $A + A^T$  is symmetric. Just taking the intersection of the automorphism groups of the summands would not find that.

Another option is to convert the sum to a function... But no, that's weird. That would require me to support functions with arbitrary numbers of inputs, which is not currently the case.

### 12.1.3 In Evaluation

When evaluating a tensor, we can look at the graph of the tensor and see if it's isomorphic to a previously evaluated tensor. This is an example where we don't really need a canonical form, but an approximate hash plus vf2 would be fine. Also note that in this case we don't care about the edge renaming, because we can just rename the edges before we return the tensor. E.g. if we have already evaluated  $A$ , we can use that to get  $A^T$  easily.

### 12.1.4 In Variables

In variables we include the name of the variable in the hash. Basically we assume that variables named the same refer to the same data.

```

1     base = hash(("Variable", self.name))
2     return base, [hash((base, e)) for e in self.original_edges]

```

For the original canonical edge names, we use the edge names before renaming. This means, in the case of  $A^T$  that will have the same hash as  $A$ . But because it's renamed, the `t.index` call in the Sum will flip the edges.

We could imagine variables taking an automorphism group as an argument, which would allow us to define variables with different symmetries. Such as a symmetric matrix  $A$  where  $A + A^T$  is actually  $2A$ .

### 12.1.5 In Constants

When computing the canonical form of a constant, like `Zero` or `Copy` we don't care about the edge names. I guess because the constants we use are all maximally symmetric? We currently include the constants `tag`, which is the hash of the variable that it came from, if any.

### 12.1.6 In Functions

One issue is that while the original names are usually part of the function definition, the new edges added by differentiation are often automatically generated based on the context, so they shouldn't really be part of the canonical form.

In contrast to Sum, we don't sort the canons here, since the order of the inputs matters.

Maybe functions should be allowed to transform the symmetry group? E.g. if we have a function that takes a symmetric matrix and returns a symmetric matrix, we should be able to use the symmetry group of the input to simplify the output.

### 12.1.7 In Derivatives

All we do is hashing the tensor and the wrt. And then add new edges for the derivative.

### 12.1.8 Other

For some tensors there might be edge dimension relations that aren't equivalences. For example, a flatten tensor would have the "out" edge dimension equal to the product of the "in" edge dimensions.

In a previous version I had every tensor register a "callback" function. Whenever an edge dimension "became available", the tensor would get a chance to emit new edge dimensions. However, this was a lot more work for each tensor to implement, and not needed for any of the existing tensors.

## 12.2 Renaming

This is an important part of the code.

## 12.3 Evaluation

An important part of evaluation is determining the dimension of each edge. To do this, I'm basically creating a full graph of the tensor, using a function called `edge_equivalences` which a list of tuples  $((t_1, e_1), (t_2, e_2))$ , indicating that edge  $e_1$  of tensor  $t_1$  is equivalent to edge  $e_2$  of tensor  $t_2$ . Note that the same edge name can appear multiple times in the graph, so we need to keep track of the tensor as well.

For variables, since the user gives edge dimensions in terms of variables, it's important to keep track of renamed edge names:

```
1 for e1, e2 in zip(self.original_edges, self.edges):
2     yield (self, e1), (self, e2)
```

For constants, there might be some equivalences based on tensors that the constant was derived from.

```
1 def edge_equivalences(self):
2     if self.link is not None:
3         yield from self.link.edge_equivalences()
4         for e in self.link.edges:
```

```

5         if e in self.edges:
6             yield (self, e), (self.link, e)

```

For the copy tensor, everything is equivalent:

```

1 def edge_equivalences(self):
2     yield from super().edge_equivalences()
3     for e in self.edges[1:]:
4         yield (self, self.edges[0]), (self, e)

```

For functions we can't really say anything about the edges of the function itself (`self.edges_out`), but at least we can say something about the broadcasted edges.

```

1 for t, *inner_edges in self.inputs:
2     yield from t.edge_equivalences()
3     for e in t.edges:
4         if e not in inner_edges:
5             yield (t, e), (self, e)

```

We could maybe also say that input edges with the same name are equivalent?

For products, we look at each edge  $(t_1, e, t_2)$  and yield  $(t_1, e), (t_2, e)$ . However for the free edges,  $(t, e)$ , we match them with ourselves,  $(t, e), (self, e)$ .

```

1 def edge_equivalences(self):
2     pairs = defaultdict(list)
3     for t in self.tensors:
4         yield from t.edge_equivalences()
5         for e in t.edges:
6             pairs[e].append(t)
7     for e, ts in pairs.items():
8         if len(ts) == 1:
9             yield (self, e), (ts[0], e)
10        else:
11            t1, t2 = ts
12            yield (t1, e), (t2, e)

```

Similarly, for sums, everything is just matched with ourselves:

```

1 def edge_equivalences(self):
2     for t in self.tensors:
3         yield from t.edge_equivalences()
4         for e in t.edges:
5             yield (t, e), (self, e)

```

Finally, we use BFS to propagate the edge dimensions from the variables (which are given by the user) to the rest of the graph.

Why is it even necessary for non-variables to know the edge dimensions? Mostly because of copy tensors, which we use for hyper edges, and have to construct. Could we get rid of this if we computed hyper-edges more efficiently without copy's? There are also sometimes "detached" copies...

Also, an alternative idea would be to actually construct the full graph. I originally didn't think this would be possible because of the Sum's which aren't really graphs. But maybe with the new approach of using nauty, we could actually do this.

### 12.3.1 Products

We simply evaluate the tensors in the product and give them to einsum.

## 12.4 Simplification Rules

There are a bunch of these.

Mostly we can do everything in a single depth-first pass, but a few times we need to do multiple passes. That can be done with the full-simplify method, which repeatedly calls simplify until nothing changes.

# Bibliography

- [1] John Aldrich. *The Origins of Mathematical Words: A Comprehensive Dictionary of Latin, Greek, and Arabic Roots*. Mathematical Association of America, 2010.
- [2] William Rowan Hamilton. *Lectures on Quaternions*. Hodges and Smith, 1853.

## Chapter 13

# Appendix

Contains some proofs, such as of equation 524 or 571. They are pretty long and could be useful for contrasting with the diagram proofs.