Basic mathematics of Deep Learning

BOAZ 19th Analyze Session

Jae Hyuk Sung

00 Index

Contents

01 Matrix Calculation

What is vector?

- *앞으로 소개할 내용은 수학에서 정의하는 Vector와 Matrix의 엄밀한 정의와 다름을 기억하세요.*
- Vector은 여러 숫자들을 관리하기 편하게 하나로 모아 놓은 구조체로 생각하시면 편합니다.
  - 행벡터(Row Vector)과 열벡터(Column Vector)라는 용어를 조금 더 많이 보게 될 것입니다.
  - 표기로는 보통 굵은 알파벳을 사용합니다.
  - 총 n개의 원소를 행벡터와 열벡터로 표현하면, 다음과 같습니다.

Row Vector

$$\mathbf{v} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

Column Vector

$$\mathbf{v} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

01 Matrix Calculation

What is matrix?

- 앞의 벡터는 1차원, 즉 단순하게 "한 방향"으로의 원소 나열을 의미한다면, 행렬은 2차원, 즉 가로와 세로로 원소들을 나열하는 도구입니다.
  - 행렬이 가지는 의미는 수들의 나열이 아니긴 합니다만, 이러한 내용은 알 필요가 없으므로 생략하겠습니다.
- 이때 가로를 행, 세로를 열이라고 칭합니다. 가로에 있는 원소의 개수가 열의 크기, 세로에 있는 원소의 개수가 행의 크기가 됩니다.
  - 이 두개를 종합하면, 행렬의 크기라고 칭합니다. 즉, 행렬의 크기가 $m \times n$이라는 것은 세로에 m개, 가로에 n개의 수가 있는 행렬입니다.
  - 정식 표현으로는 다음과 같이 작성합니다.

$$\begin{array}{ccc} & \begin{array}{cccccc} \phantom{a_{11}} & \phantom{a_{12}} & & j\text{-th column} & & \phantom{a_{1n}} \end{array} & \\ A = & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix} & \begin{array}{c} \\ \\ \\ \leftarrow i\text{-th row} \\ \\ \\ \end{array} \end{array}$$

01 Matrix Calculation

Relation between matrix and vector

- 이때, 행렬은 가로와 세로로 쌓는 것이고, 벡터는 가로와 세로 중 한 방향으로 쌓는 것이기 때문에 벡터를 이용하면 행렬을 1차원으로 표현할 수 있습니다.
- 다른 말로, 앞에서 익혔던 Column vector(혹은 Row vector)를 쌓으면, 곧 행렬이 됩니다.
- 즉, 행렬 A는 다음과 같이 쓸 수 있습니다.

$$A = \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \cdots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{w_1} \\ \mathbf{w_2} \\ \vdots \\ \mathbf{w}_m \end{bmatrix}$$

- 이때, $\mathbf{v}_i$는 m개의 숫자가, $\mathbf{w}_j$ 는 n개의 숫자가 들어 있는 벡터입니다.
- 특히 행렬을 열 벡터만을 이용하여 표현하는 경우는 굉장히 많습니다. (일반적으로 벡터라고 칭하면 열벡터를 의미하기 때문입니다.)

01 Matrix Calculation

Matrix Operation ①: Addition, Scalar multiplication, Transpose

- 행렬에는 대표적으로 ① 행렬 간의 더하기 ② 스칼라 곱 ③ 전치 ④ 행렬 간의 곱이라는 4가지의 연산이 존재합니다.
  - 따로 언급하진 않았지만, 행렬 간의 빼기는 더하기와 스칼라 곱을 이용하면 정의가 가능합니다.
- 스칼라 곱은 원소 하나와 행렬 하나가 필요하며, 전치는 아무 조건이 필요 없습니다.
  - 전치는 따로 설명하지 않지만, 굉장히 자주 사용되는 연산 중 하나입니다. 추후 내적을 설명할 때 다시 등장합니다.
- 행렬 간의 더하기는, 같은 크기의 두 개의 행렬이 필요합니다.

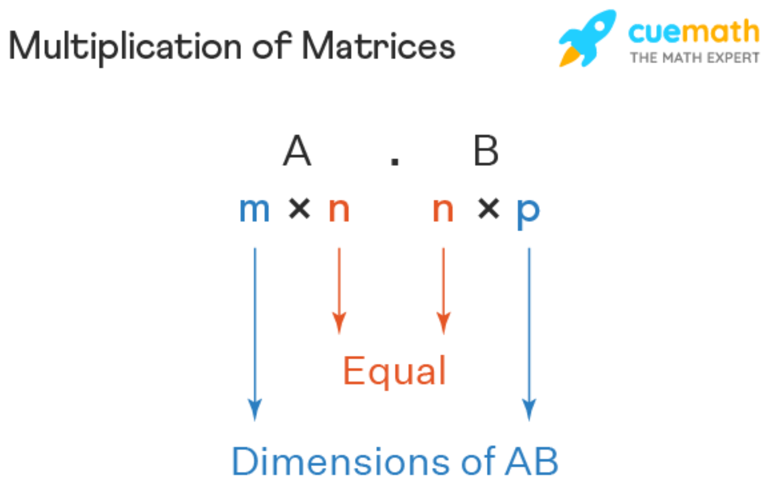
Operations performed on matrices

Operation	Definition	Example
Addition	The *sum* **A**+**B** of two *m*-by-*n* matrices **A** and **B** is calculated entrywise: $(\mathbf{A}+\mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$, where $1 \le i \le m$ and $1 \le j \le n$.	$\begin{bmatrix}1 & 3 & 1\\ 1 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 5\\ 7 & 5 & 0\end{bmatrix} = \begin{bmatrix}1+0 & 3+0 & 1+5\\ 1+7 & 0+5 & 0+0\end{bmatrix} = \begin{bmatrix}1 & 3 & 6\\ 8 & 5 & 0\end{bmatrix}$
Scalar multiplication	The product *c***A** of a number *c* (also called a scalar in the parlance of abstract algebra) and a matrix **A** is computed by multiplying every entry of **A** by *c*: $(c\mathbf{A})_{i,j} = c \cdot \mathbf{A}_{i,j}$. This operation is called *scalar multiplication*, but its result is not named "scalar product" to avoid confusion, since "scalar product" is sometimes used as a synonym for "inner product".	$2 \cdot \begin{bmatrix}1 & 8 & -3\\ 4 & -2 & 5\end{bmatrix} = \begin{bmatrix}2\cdot 1 & 2\cdot 8 & 2\cdot -3\\ 2\cdot 4 & 2\cdot -2 & 2\cdot 5\end{bmatrix} = \begin{bmatrix}2 & 16 & -6\\ 8 & -4 & 10\end{bmatrix}$
Transposition	The *transpose* of an *m*-by-*n* matrix **A** is the *n*-by-*m* matrix $\mathbf{A}^T$ (also denoted $\mathbf{A}^{tr}$ or ${}^t\mathbf{A}$) formed by turning rows into columns and vice versa: $(\mathbf{A}^T)_{i,j} = \mathbf{A}_{j,i}$.	$\begin{bmatrix}1 & 2 & 3\\ 0 & -6 & 7\end{bmatrix}^T = \begin{bmatrix}1 & 0\\ 2 & -6\\ 3 & 7\end{bmatrix}$

01 Matrix Calculation

Matrix Operation ②: Matrix Multiplication

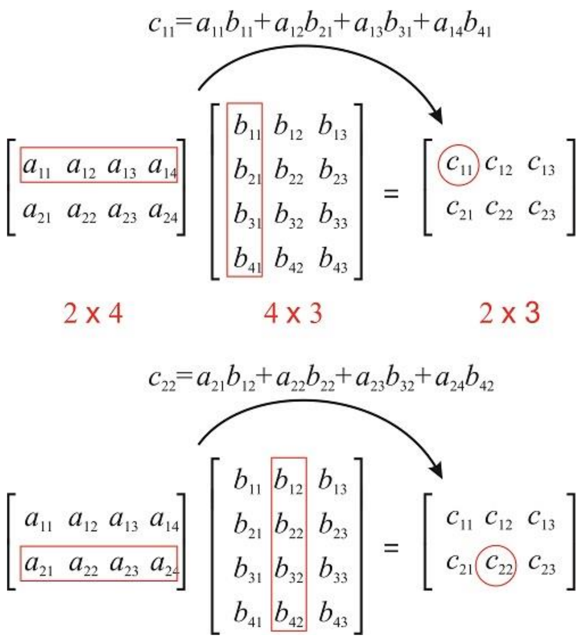
- 행렬 간의 곱은 (최소) 두 개의 행렬이 필요하며, 다른 연산들과 다르게 조건이 까다롭습니다.
- 행렬 곱이 정의가 되기 위해선, 곱하려는 두 행렬의 가운데 숫자가 같아야 합니다.
    - More Specifically, 크기가 $m_1 \times n_1$ 인 행렬 A 와 크기가 $m_2 \times n_2$ 인 행렬 B 의 곱 AB 가 정의되기 위해서는 가운데 숫자가 같야합니다.
    - 즉, $n_1 = m_2$ 이어야 하고 그렇게 해서 나온 행렬의 크기는 $m_1 \times n_2$ 가 됩니다.



01 Matrix Calculation

Matrix Operation ②: Matrix Multiplication

- 행렬 곱을 연산하는 방법은 가로 하나와 세로 하나를 온전히 잡고, 원소들을 차례차례 곱해서 더하면 됩니다.
- 예시를 보는 것이 더 빠르겠네요..
- 꼭 원소가 숫자 같은 것이 아니여도, 조건만 맞으면 행렬곱은 성립합니다. 또한, 행렬 곱은 결합 법칙과 분배 법칙은 성립하나, 교환 법칙은 성립하지 않습니다.



Property	Addition	Multiplication
Commutative	a + b = b + a	a ● b = b ● a
Associative	(a + b) + c = a + (b + c)	(a ● b) ● c = a ● (b ● c)
Distributive	a(b + c) = a ● b + a ● c (b + c)a = b ● a + c ● a a ● b + a ● c = a(b + c)	

01 Matrix Calculation

Hadamard product (Element-wise product)

- 행렬 곱은 크기가 서로 다른 것들을 일련의 방법으로 계산하는 연산이라면, 각 행렬의 원소마다 곱하는 방법도 필요할 것입니다.
    - 일반적인 행렬 곱으로는 행렬의 각 원소를 곱하는 연산을 수행할 수 없습니다. (Vectorization을 수행하면 가능할지도)
- 이럴 때 사용하는 것이 Hadamard product으로, 원소별로 계산하기 때문에 Element-wise operation이라고도 불립니다.
- 기존의 행렬 곱과 다르게, 교환 법칙 / 결합 법칙 / 분배 법칙이 모두 성립합니다.
- 기호로는 $\odot$을 사용합니다. 즉, 같은 크기를 가지는 두 행렬에 대해

$$(A \odot B)_{ij} = A_{ij} B_{ij}$$

가 성립합니다.

01 Matrix Calculation

Element-wise function / Some useful tricks!

- 직전의 Hadamard product과 같이 딥러닝에서는 행렬 각 원소마다 어떠한 특정한 함수를 취해주는 경우가 굉장히 많습니다.
  - 가장 대표적인 예로 softmax가 있고, Entry-wise norm도 이에 해당합니다.
- 그러기 위해서 Element-wise function을 선언하여 사용하는 경우가 굉장히 많습니다.
- 따로 정해진 표기는 없고, 나중에 사용하실 때 안 헷갈리기만 하면(…) 상관 없습니다.
- 이외에도, 행렬의 모든 합을 구하거나 / 행렬의 행마다 합을 구하거나 / 행렬의 열마다의 합을 구하거나 할 경우, 사용할 수 있는 useful한 trick이 있습니다.
  - 행렬의 모든 원소의 합을 구하는 공식(스칼라 반환): $\mathbf{1}_m^T A \mathbf{1}_n$
  - 행렬의 열마다의 합을 구하는 공식(열의 개수만큼 반환): $\mathbf{1}_m^T A$
  - 행렬의 행마다의 합을 구하는 공식(행의 개수만큼 반환): $A\mathbf{1}_n$

이때, $\mathbf{1}_m$은 모든 원소가 1인 $m \times 1$의 열벡터입니다.

02 Gaussian & Cross Entropy

Skip

- 정신 건강을 위해 생략합니다.
  - 해라면 할 수는 있는데.. 이건 모든 사람들이 Cross Entropy Loss에 대해 배운 후 시간이 나면 한 번 풀어보도록 하겠습니다.
  - Variational Auto Encoder에서 사용된 Bayesian Method의 가장 기초가 되는 Gaussian 관련하여 풀어볼 예정입니다.
  - 특히, Loss가 단순히 고정이 되어 있는 것이 아니라 분포의 입장에서 보면 Gaussian이 됨을 Maximum Lieklihood Estimation을 통해 전개해나갈 수 있습니다.
    - 보기만 해도 좀 그렇네요 하하
- 목차 고치기 귀찮아서 이렇게 하겠습니다,, ㅎ;

03 Chain Rule & Trace Trick

Motivation of Gradient & Definition

- 본격적으로 "학습"이라는 토픽을 배우기 시작하면은, 어떤 값을 기준으로 하여 어떤 방법으로 학습시키는 것인지에 대한 깨달음이 필요합니다.
- 일반적으로 지도 학습(Supervised Learning)에서는 a) 정답과 우리가 예측한 값들의 차이를 기준으로 b) 이것을 최소화하는 방향으로 c) 적당한 방법을 활용하여 매개변수들을 업데이트 하는 것을 목표로 합니다. (Non-Bayesian)
  - A)는 일반적으로 Loss, b)는 Optimization problem (Generally Convex Optimization Problem) c)는 Gradient Descent 라고 합니다.
- 여기에서 a)는 정의하기 나름이고, b)는 너무 수학적이기 때문에 이 자리에서 논하긴 힘들고, c) 정도는 어느 정도 알면 도움이 되기 위해 논하려고 합니다.
- 추후 편의를 위해, k개의 원소가 있는 벡터공간을 $\mathbb{R}^k$, 함수 $f : \mathbb{R}^k \to \mathbb{R}$로 정의 없이 사용하도록 합니다.
  - 함수의 의미는 벡터를 일종의 연산을 거쳐서 스칼라로 보내 버리는 함수입니다.
  - 우리가 일반적으로 사용하는 Loss를 반환하는 함수라고 생각하시면 됩니다!

03 Chain Rule & Trace Trick

What is Gradient? & Gradient Descent

- K차원의 벡터 p가 함수 f를 거쳤을 때, 어떠한 값을 가지고, 우리는 이 값이 0으로 만들게 함수 f의 매개변수들을 변경하고 싶습니다.
- 즉, 우리는 함수 f가 그려내는 그림에서 원래의 f(p)의 정보를 이용해서 f(p)=0으로 값을 감소시키고, 그 중에서도 가장 빠른 것을 찾고 싶습니다.
- 그러기 위해서는 어떤 방향으로 가는 것이 좋을까요? => Gradient
- Gradient는 어떠한 점에서 가장 증가율이 큰 것을 나타내는 벡터를 의미합니다. 즉, Gradient 방향으로 가면 가장 빠른 속도로 값을 늘릴 수 있다는 것이죠!
- 허나, 우리는 값을 감소하는 것이 목표입니다. 그렇기 때문에, Gradient가 알려주는 방향이 아닌, Gradient의 반대 방향으로 가야만 이 문제를 해결할 수 있습니다. => Gradient Descent의 원리입니다.
- Gradient는 반드시 입력의 크기와 같은 결과를 가집니다. 즉, K차원의 벡터였다면 gradient도 K차원이어야 합니다.
- 우리가 최적화하고 싶은 문제가 Convex하면 Gradient Descent를 이용하면 언젠가 반드시 최적에 도달함이 증명이 되어 있습니다.
  - 그러면, 일반적으로 (우리가 아는) Gradient Descent가 과연 가장 빠른 방법일까요?
  - 답은 아니오인데, 어떠한 반례 때문에 아닌지 한번 생각해보세요! 참고: https://dasu.tistory.com/69

03 Chain Rule & Trace Trick

Chain Rule

- Gradient를 계산하기 위해서는, 각 변수에 대한 편미분을 실시해야 합니다. 즉, 함수 f에 대한 각 변수들에 대한 변화량의 값이 필요합니다.
- 일반적으로 함수 f는 간단하게 적히지 않고, 함수들이 겹겹히 쌓여 있는 모양이 됩니다. 즉, 여러 함수들의 합성이 되어 함수 f를 구성하게 됩니다.
- 그렇기 때문에 함수를 미분할 때 각각의 함수를 미분하여 합쳐버리는 Chain Rule을 사용하면 편합니다. Chain Rule이란,

$$\frac{d}{dx}(f \circ g)(x) = (f' \circ g)(x) \cdot g'(x)$$

- 하지만, Gradient에서는 Chain-rule이 존재하지 않습니다! 따라서, Gradient를 이용한 Chain-rule을 위해서는, 다른 도구를 들고 와야합니다.
- 해결하는 도구로 Total derivate라는 것이 있고, 우리가 일반적으로 사용하는 미분 표기를 사용합니다. 즉, f에 대한 Total derivate는 df 로 적습니다.
- 동일하게 Chain Rule을 적용하면, $d(g \circ f)_a = dg_{f(a)} \cdot df_a$ 로 적을 수 있습니다.
- 정의에 의해 Total derivate는 Gradient과 밀접한 관련이 있습니다. 이를 통해 추후 Trace Trick을 활용합니다.

$$df = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} dx_i = (\nabla_{\mathbf{x}} f)^T \cdot d\mathbf{x}$$

03 Chain Rule & Trace Trick

Product rule

- x에 관한 두 함수 $f(x)$, $g(x)$가 있다고 가정합시다. 우리가 구하고자 하는 것은, $f(x)g(x)$에 대한 미분값입니다.
- 각 함수가 모두 관련이 있기 때문에, 단순히 하나씩 미분해서 더하면 되는 것이 아니라, 다른 방식으로 계산하여야 합니다! 일반적으로, 다음과 같이 계산합니다.

$$\frac{d}{dx}f(x)g(x) = f'(x)g(x) + f(x)g'(x)$$

- 위의 과정을 Product Rule이라고 하며, 벡터에도 동일하게 성립합니다. 이때, 벡터에 대해 Product Rule을 계산할 때는 순서를 반드시 지켜주어야 합니다.
- 이해를 위해 $\mathbf{y} = W\mathbf{x}$ 라는 간단한 선형 식을 생각해봅시다. 이때, 순서는 (앞에꺼 미분)*(뒤에꺼 그대로) + (앞에꺼 그대로)*(뒤에꺼 미분)입니다. 즉,

$$d\mathbf{y} = dW \cdot \mathbf{x} + W \cdot d\mathbf{x}$$

- 먼저 x에 대한 gradient를 구해 보자면 W와 x는 독립적이므로 dW의 값은 0이 됩니다. 따라서, $d\mathbf{y} = W \cdot d\mathbf{x}$ 로 적을 수 있습니다.
- 다음으로 W에 대한 gradient를 구해 보자면 W와 x는 독립적이므로 dx의 값은 0이 됩니다. 따라서, $d\mathbf{y} = dW \cdot \mathbf{x}$ 로 적을 수 있습니다.
- 순서가 다르기 때문에, 나중에 Chain rule을 사용하였을 때 앞쪽에 곱해질 지, 뒤쪽에 곱해질 지가 정해집니다! => 이 때문에 순서를 반드시 지켜야 합니다.

03 Chain Rule & Trace Trick

Differentation Rules

- 다음 규칙들은 자주 사용할 규칙들입니다. 나중에 굉장히 자주 사용되니 기억해두시면 편합니다.

  - $d(X \pm Y) = dX \pm dY$
  - $d(XY) = (dX)Y + XdY$
  - $dtr(X) = tr(dX)$
  - $d(X^T) = (dX)^T$
  - $d(X \odot Y) = (dX) \odot Y + X \odot dY$
  - $df(X) = f'(X) \odot dX$

- 이때, $f(X)$는 element-wise operation입니다.

03 Chain Rule & Trace Trick

Chain rule & Product rule

Backprop with Matrices

x: [N×D]

[2 1 -3]
[-3 4 2]

w: [D×M]

[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

[13 9 -2 -6]
[5 2 17 1]

dL/dy: [N×M]

[2 3 -3 9]
[-8 1 4 6]

$$\boxed{\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T}$$

$$\boxed{\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)}$$

03 Chain Rule & Trace Trick

Dot product

- 다시 선형대수학으로 돌아와서, 여러분들이 한번쯤은 들어봤을 내적(Dot product)이라는 개념에 대해 논해봅시다.
- 보통 (벡터의)내적이라고 하면, 벡터의 각 요소들끼리 다 곱한다음 더하는 것을 의미합니다.
  - 예를 들어 (1, 2, 3)과 (4, 5, 6)의 내적은 1*4 + 2*5 + 3*6=4 + 10 + 18=32가 됩니다.
- 상위 개념을 위해, 내적의 조건을 조금 더 까다롭게 들어가 봅시다. 과연 저것 뿐만이 벡터를 내적할 수 있는 유일한 방법일까요?
- 가장 중요한 것만 짚고 넘어가자면, 내적이라는 개념은 두 개의 입력을 이용하여 하나의 스칼라를 내는 교환법칙을 허용하는 연산입니다.
- 지금 저 위에 있는 것처럼 다 곱하고 더하고 나니까 숫자 하나만 띡 하고 나오니까, 이건 내적이라고 볼 수 있겠네요!
  - 물론 이게 벡터의 내적이 되기 위한 유일한 조건은 아닙니다. 여러 개 더 있습니다.
- 벡터는 내적이 잘 정의되니까 상관 없는데, 행렬에서도 비슷한 방법으로 내적을 정의할 수 있을까요?
  - 하지만, 벡터는 한 차원으로만 확장이 되어 있는 것이기 때문에 상관이 없지만, 행렬은 두 차원 모두 영향을 받기 때문에 위와 같은 정의는 힘들 듯 합니다.

03 Chain Rule & Trace Trick

Trace

- 특히, 앞쪽에서 가장 큰 문제는 교환 법칙이 성립해야 한다는 것입니다. 일반적으로, 행렬 곱은 교환 법칙이 성립하지가 않죠!
  - 그렇기 때문에 가장 간단한 방법인 "행렬 곱 이후 모든 원소를 다 더한다" 같은 것은 봉쇄가 되어 버렸습니다.
  - 뿐만 아니라, 같은 크기의 행렬에 대해서만 내적이 정의가 되어야 하므로, 행렬 곱을 함부로 할 수가 없습니다! (전치를 시켜야 합니다)
- 생각해보니, 앞쪽에서 이거와 굉장히 비슷한 연산을 한번 언급했습니다. Hadamard product는 1) 같은 크기에서 정의되고 2) 교환이 성립하죠.
  - 행렬을 하나의 스칼라 값을 만들어 주기 위해 모든 원소를 더합시다. 즉, $\sum_{i,\,j}(A \odot B)_{ij}$의 값을 계산하자는 것이죠.
  - 이를 행렬 곱을 이용하여 나타낼 수는 없을까요?
- 조금의 계산이 수반되어야 하지만, 정답만 듣고옵시다. 놀랍게도, A를 전치하여 B와 곱하나, B를 전치하여 A와 곱하나 각 대각선들의 합은 같게 됩니다.
  - 추가로 이 값은 앞에서 말한 Hadamard product 이후 모든 원소의 합과 같게 됩니다.
  - 이를 이용하기 위해 정사각행렬에서 대각선 원소들의 합을 들고 오는 연산자가 필요하고, 이를 Trace라고 부릅니다.

03 Chain Rule & Trace Trick

Frobenius inner product

- 앞쪽에서 정의가 되는 것들을 들고 왔으니 이걸 행렬의 내적으로 정의할 수 있겠네요!
    - 가장 많이 쓰이는 행렬 내적 중 하나로, Frobenius inner product라고 부릅니다. (같은 크기의 행렬이면 모두 사용가능합니다.)
    - 일반 내적과 헷갈리지 않기 위해 $\langle A,\ B\rangle_F$로 밑에 F를 달아 표기합니다. 정의는 $\langle A,\ B\rangle = \mathrm{Tr}(A^T B)$
- Trace와 내적에서 성립하는 몇몇 성질들을 증명 없이 알아보도록 하고, 이를 바로 사용해보도록 합시다.

- $\langle X,\ Y\rangle = \langle Y,\ X\rangle$
- $\langle aX,\ Y\rangle = a\langle X,\ Y\rangle = \langle X,\ aY\rangle$
- $\langle X+Z,\ Y\rangle = \langle X,\ Y\rangle + \langle Z,\ Y\rangle$
- $\langle X,\ Y\odot Z\rangle = \langle X\odot Y,\ Z\rangle$
- $\langle AC,\ BD\rangle = \langle A,\ BDC^T\rangle = \langle B^T AC,\ D\rangle$

- $a = tr(a)$
- $tr(A^T) = tr(A)$
- $tr(A \pm B) = tr(A) \pm tr(B)$
- $tr(A^T B) = tr(B^T A)$
- $tr(A^T(B\odot C)) = tr((A\odot B)^T C)$

03 Chain Rule & Trace Trick

Trace Trick

- 일반적으로 Loss는 벡터나 행렬이 아닌 스칼라의 반환 값을 가집니다. 즉, 실수 값을 낸다는 것이죠. 그 값을 앞으로 $\mathcal{L}$ 이라고 합시다.
- 하지만, $\mathcal{L}$ 에 대해 바로 입력 값 (혹은 중간 값)에 대한 gradient를 구하는 것은 쉽지 않습니다.
- 잠시 앞쪽으로 돌아가서, Total derivate와 Gradient의 관계를 먼저 봅시다.

$$df = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} dx_i = (\nabla_{\mathbf{x}} f)^T \cdot d\mathbf{x}$$

- 이 정의에서 f의 결과는 스칼라였으니, $\mathcal{L}$ 도 이 정의를 사용할 수 있겠군요! 즉, $d\mathcal{L}$ 에 대해 식을 정리할 수 있게 됩니다.
- 이때, 스칼라에 한해서는 Trace를 씌우나 안 씌우나 동일하므로, $tr(d\mathcal{L}) = d\mathcal{L}$ 이 성립하게 되겠군요. 식을 대입하면,

$$d\mathcal{L} = tr\left((\nabla_{\mathbf{x}} f)^T \cdot d\mathbf{x}\right)$$

이고, 이건 Frobenius inner product의 정의에 의해 다음과 같이 적을 수 있습니다. 이렇게 Gradient를 구하는 과정을 Trace Trick이라고 합니다.

$$d\mathcal{L} = \langle \nabla_{\mathbf{x}} f, \, d\mathbf{x} \rangle$$

03 Chain Rule & Trace Trick

Example: Trace Trick

$$f(X) = a^T \exp(Xb)$$

$$\begin{aligned} df &= \langle 1,\, d(a^T \exp(Xb)) \rangle \\ &= \langle 1,\, a^T d(\exp(Xb)) \rangle \\ &= \langle a,\, d(\exp(Xb)) \rangle \\ &= \langle a,\, \exp(Xb) \odot d(Xb) \rangle \\ &= \langle \exp(Xb) \odot a,\, (dX)b \rangle \\ &= \langle \{\exp(Xb) \odot a\} \cdot b^T,\, dX \rangle = \langle \nabla_X f,\, dX \rangle \end{aligned}$$

$$\therefore\ \nabla_X f = (\exp(Xb) \odot a) \cdot b^T$$

$$f(X) = \langle Y,\, MY \rangle,\ \ Y = \sigma(WX)$$

$$\begin{aligned} df &= d(tr(Y^T MY)) \\ &= tr(d(Y^T MY)) \\ &= tr((dY^T)MY + Y^T d(MY)) \\ &= tr((dY^T)MY) + tr(Y^T d(MY)) \\ &= \langle MY,\, dY \rangle + \langle Y,\, d(MY) \rangle \\ &= \langle MY,\, dY \rangle + \langle Y,\, MdY \rangle \\ &= \langle MY,\, dY \rangle + \langle M^T Y,\, dY \rangle \\ &= \langle (M + M^T)Y,\, dY \rangle \\ &= \langle (M + M^T)Y,\, d(\sigma(WX)) \rangle \\ &= \langle (M + M^T)Y,\, \sigma'(WX) \odot d(WX) \rangle \\ &= \langle \sigma'(WX) \odot ((M + M^T)Y),\, WdX \rangle \\ &= \langle W^T(\sigma'(WX) \odot ((M + M^T)Y)),\, dX \rangle = \langle \nabla_X f,\, dX \rangle \end{aligned}$$

$$\therefore\ \nabla_X f = W^T(\sigma'(WX) \odot ((M + M^T)Y))$$

03 Chain Rule & Trace Trick

Example: Useful Trace Trick

$$\mathcal{L} = f(Y),\ Y = WX$$

$$\begin{aligned} d\mathcal{L} &= \langle \nabla_Y f,\, dY \rangle \\ &= \langle \nabla_Y f,\, d(WX) \rangle \\ &= \langle \nabla_Y f,\, (dW)X + WdX \rangle \\ &= \langle \nabla_Y f,\, WdX \rangle \\ &= \langle W^T \nabla_Y f,\, dX \rangle = \langle \nabla_X f,\, dX \rangle \end{aligned}$$

$$\therefore\ \nabla_X f = {\color{blue}W^T \cdot \nabla_Y f}(\text{Upstream Gradient})$$

$$\mathcal{L} = f(Y),\ Y = WX$$

$$\begin{aligned} d\mathcal{L} &= \langle \nabla_Y f,\, dY \rangle \\ &= \langle \nabla_Y f,\, d(WX) \rangle \\ &= \langle \nabla_Y f,\, (dW)X \rangle \\ &= \langle \nabla_Y f,\, (dW)X \rangle \\ &= \langle \nabla_Y f \cdot X^T,\, dW \rangle = \langle \nabla_W f,\, dW \rangle \end{aligned}$$

$$\therefore\ \nabla_W f = {\color{red}\nabla_Y f}(\text{Upstream Gradient}){\color{red}\cdot X^T}$$

04 Numpy Tutorial

Numpy Tutorial

- 이 링크를 이용하여 준비해봅시다!
  - http://aikorea.org/cs231n/python-numpy-tutorial/
  - Or Jupyter Notebook on presenter's PC (Attention to Zoom!)
- Core Topics
  - Indexing(Integer, Boolean)
  - Broadcasting
  - Vector & Matrix shape / Reshape
  - Array with math

Thank you for your attention