THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
RELAXED SYSTEM LAB

Machine Learning Preliminary

COMP4901Y

Binhang Yuan

Audit Policy

RELAXED SYSTEM LAB

- You are always welcome to come to my class or view the online resource;
- If you want an audit credit in your HKUST transcript:
  - Get **<u>60% on the final exam</u>**.
  - No mid-term or homework is required.

Linear Algebra

Scalars

- Sample operations

$$c = a + b$$

$$c = a \cdot b$$

$$c = \sin a$$

- Length

$$|a| = \begin{cases} \mathrm{a} & if \;\; a > 0 \\ -a & otherwise \end{cases}$$

$$|a + b| \le |a| + |b|$$

$$|a \cdot b| = |a| \cdot |b|$$

Vector

- Vector in n-dimensions $\boldsymbol{a} \in \mathbb{R}^n$, $\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix}$
- Sample operations:

$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b} \ \textit{where}\ c_i = a_i + b_i$$

$$\boldsymbol{c} = \alpha \cdot \boldsymbol{b} \ \textit{where}\ c_i = \alpha b_i$$

$$\boldsymbol{c} = \sin \boldsymbol{a} \ \textit{where}\ c_i = \sin a_i$$

Vector

- Some properties of vector addition:
  - Commutative:

$$\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}, \qquad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$$

  - Associative:

$$(\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c} = \boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c}), \qquad \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^n$$

  - Distibutive:

$$\alpha(\boldsymbol{a} + \boldsymbol{b}) = \alpha\boldsymbol{a} + \alpha\boldsymbol{b}, \qquad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$$

Vector

- p-norm

$$\|\boldsymbol{a}\|_p = \left[\sum_{i=1}^{m} a_i^p\right]^{\frac{1}{p}}$$

- p=1, Manhattan norm

$$\|\boldsymbol{a}\|_1 = \sum_{i=1}^{m} |a_i|$$

- p=2, Euclidean norm

$$\|\boldsymbol{a}\|_2 = \left[\sum_{i=1}^{m} a_i^2\right]^{\frac{1}{2}}$$

$$\|\boldsymbol{a}\| \geq 0 \ \forall \ \boldsymbol{a}$$

$$\|\boldsymbol{a} + \boldsymbol{b}\| \leq \|\boldsymbol{a}\| + \|\boldsymbol{b}\|$$

$$\|\alpha \cdot \boldsymbol{b}\| \leq |\alpha| \|\boldsymbol{b}\|$$

- p= ∞, Euclidean norm

$$\|\boldsymbol{a}\|_\infty = \max(a_1, a_2, \dots, a_m)$$

Vector

- The span of a set of vectors:

$$\text{span}\{\boldsymbol{a_1}, \boldsymbol{a_2}, \dots, \boldsymbol{a_k}\} = \{\alpha_1 \boldsymbol{a_1} + \alpha_2 \boldsymbol{a_2} + \cdots + \alpha_k \boldsymbol{a_k} \,|\, \alpha_k \in \mathbb{R},\ \boldsymbol{a_k} \in \mathbb{R}^n\}$$

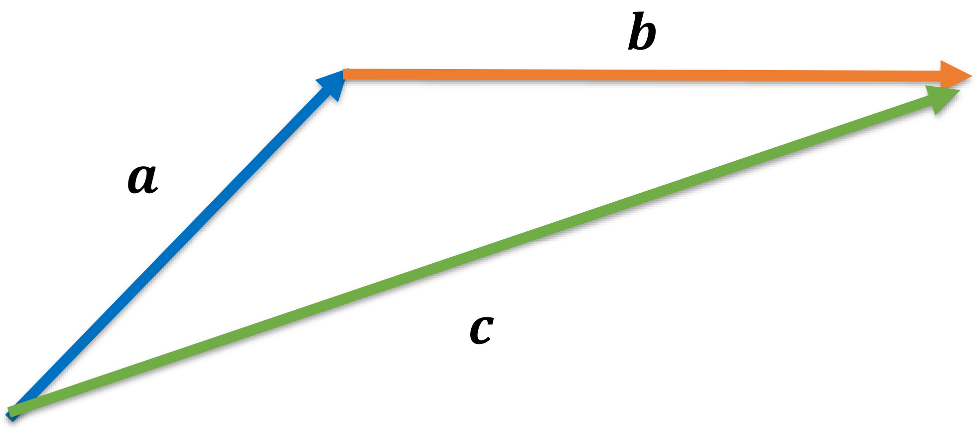
- Linear independence:

$$\alpha_1 \boldsymbol{a_1} + \alpha_2 \boldsymbol{a_2} + \cdots + \alpha_k \boldsymbol{a_k} = \boldsymbol{0} \Rightarrow \ \alpha_i = 0, \forall i$$

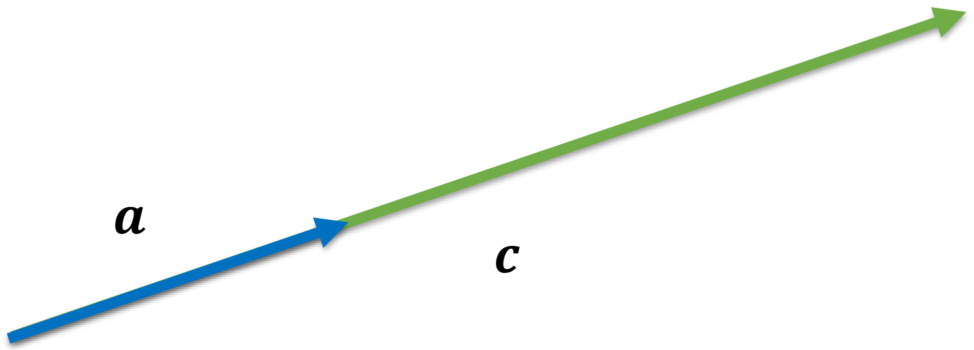
- How does k compare to n, the vector dimension?

$$k \leq n$$

Vector



$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$$



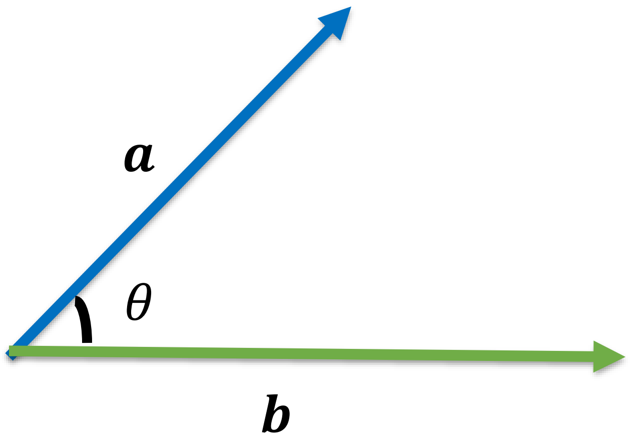
$$\boldsymbol{c} = \alpha \cdot \boldsymbol{a}$$

Mathematician's **'parallel for all do'**

Vectors

- Inner product

$$\boldsymbol{a}^T\boldsymbol{b} = \sum_i a_i b_i = \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\| \cdot \cos\theta$$



- Orthogonality

$$\boldsymbol{a}^T\boldsymbol{b} = \sum_i a_i b_i = 0$$

- If we have two vectors that are orthogonal with a third, their linear combination is, too.

Matrices

- Matrix in m, n-dimensions: $\boldsymbol{A} \in \mathbb{R}^{m \times n}$:

$$\mathbf{A} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix}$$

- Transpose of matrix:

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} A_{11} & \cdots & A_{m1} \\ \vdots & \ddots & \vdots \\ A_{1n} & \cdots & A_{mn} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

$$(\boldsymbol{A} + \boldsymbol{B})^T = \boldsymbol{A}^T + \boldsymbol{B}^T, \forall \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

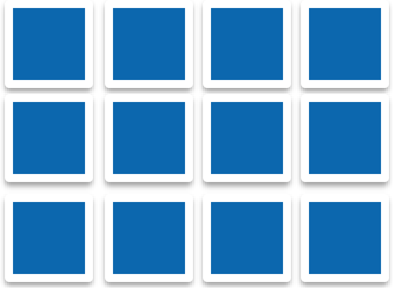
Matrices

- Simple operations

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B} \;\; where \; C_{ij} = A_{ij} + B_{ij}$$

$$\boldsymbol{C} = \alpha \cdot \boldsymbol{B} \;\; where \; C_{ij} = \alpha B_{ij}$$

$$\boldsymbol{C} = \sin \boldsymbol{A} \;\; where \; C_{ij} = \sin A_{ij}$$



Matrices

- Some properties of matrix addition:
  - Commutative:

$$\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{B} + \boldsymbol{A}, \qquad \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

  - Associative:

$$(\boldsymbol{A} + \boldsymbol{B}) + \boldsymbol{C} = \boldsymbol{A} + (\boldsymbol{B} + \boldsymbol{C}), \qquad \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{m \times n}$$

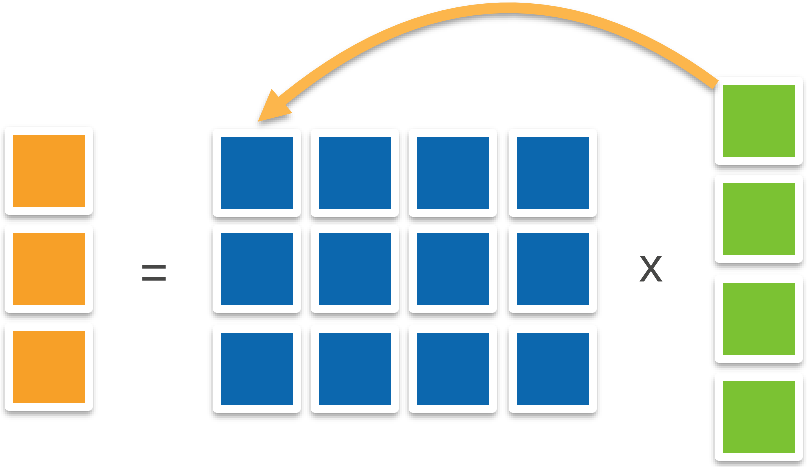
  - Distibutive:

$$\alpha(\boldsymbol{A} + \boldsymbol{B}) = \alpha\boldsymbol{A} + \alpha\boldsymbol{B}, \qquad \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

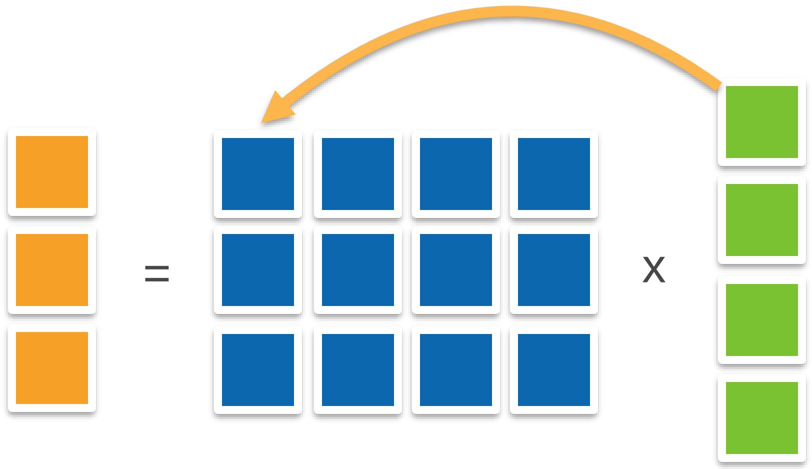
Matrices

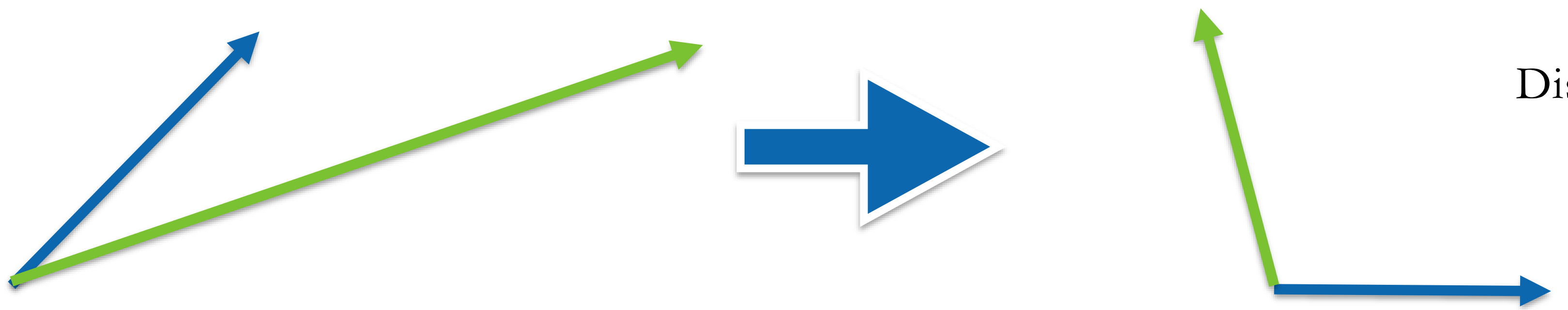
- Multiplications (matrix-vector), $\boldsymbol{c} = \boldsymbol{Ab}, \boldsymbol{c} \in \mathbb{R}^{m}, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{B} \in \mathbb{R}^{n}$

$$\begin{bmatrix} c_1 \\ \dots \\ c_m \end{bmatrix} = \boldsymbol{c} = \boldsymbol{Ab} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \text{ where } c_i = \sum_{j=1}^{n} A_{ij} b_j$$



Matrices



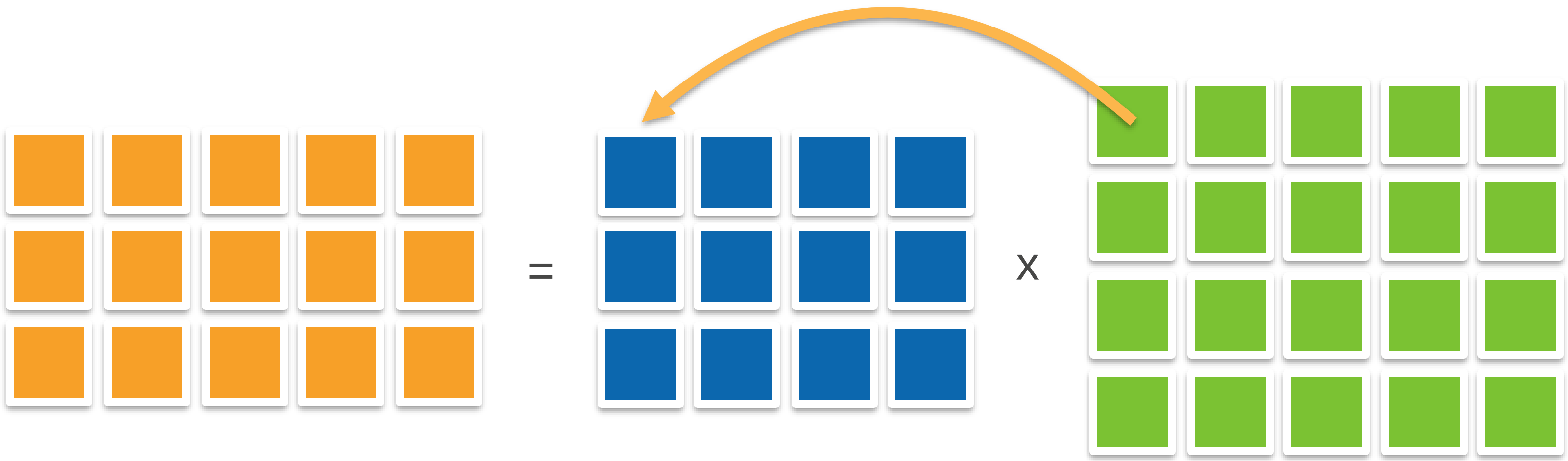


Distort the space.

Matrices

- Multiplications (matrix-matrix) $\boldsymbol{C} = \boldsymbol{AB}, \boldsymbol{C} \in \mathbb{R}^{m\times p}, \boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{B} \in \mathbb{R}^{n\times p}$

$$\begin{bmatrix} C_{11} & \cdots & C_{1p} \\ \vdots & \ddots & \vdots \\ C_{m1} & \cdots & C_{mp} \end{bmatrix} = \boldsymbol{C} = \boldsymbol{AB} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} B_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & A_{np} \end{bmatrix} \text{where } C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$$



Matrices

- Some properties of matrix multiplication:
  - Non-commutative!

$$\boldsymbol{AB} \neq \boldsymbol{BA}$$

  - Associative:

$$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC}), \qquad \forall \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$$

$$\alpha(\boldsymbol{AB}) = (\alpha\boldsymbol{A})\boldsymbol{B}, \qquad \forall \boldsymbol{A}, \boldsymbol{B}$$

  - Distibutive:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AB}, \qquad \forall \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$$

  - Transpose:

$$(\boldsymbol{AB})^T = \boldsymbol{B}^T\boldsymbol{A}^T$$

Matrix

RELAXED SYSTEM LAB

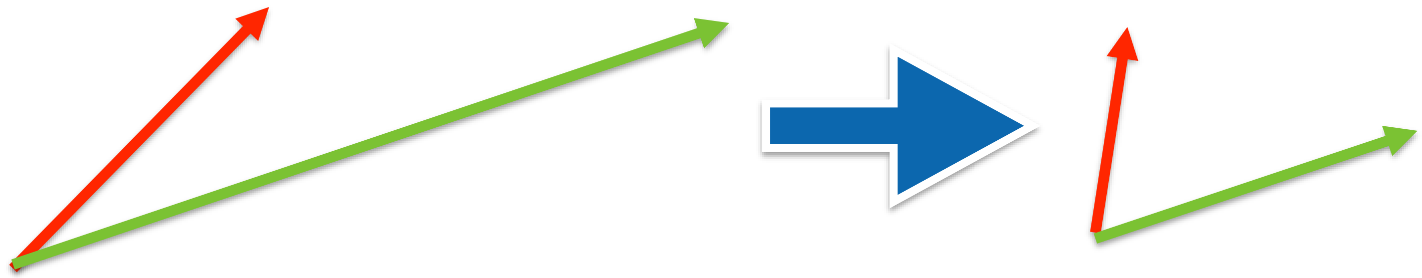
- Norm: $$\|\boldsymbol{A}\| = \sup\left\{\frac{\|\boldsymbol{Ax}\|_p}{\|\boldsymbol{x}\|_p}, \forall\, \boldsymbol{x} \neq 0\right\}$$
- Interpretation: <u>how much can the mapping induced by $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ can stretch vectors?</u>
- Choices depend on how to measure the length of vectors.

- Popular norms:
  - p=1, $\|\boldsymbol{A}\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$
  - p=∞, $\|\boldsymbol{A}\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|$
  - Frobenius norm: $\|\boldsymbol{A}\|_F = \left[\sum_{ij} A_{ij}^2\right]^{\frac{1}{2}}$
  - ~~p=2,~~ $\cancel{\|\boldsymbol{A}\|_2 = \sigma_{max}(\boldsymbol{A})}$ ~~(the largest singular value of matrix.)~~

Matrix

- Eigenvectors and egienvalue
  - Vectors that are not changed by the matrix ($\boldsymbol{x}$ is the vector, $\boldsymbol{\lambda}$ is the eginvalue):

$$\boldsymbol{Ax} = \lambda\boldsymbol{x}$$



  - For symmetric matrices, we can always find the eigenvalue and eigenvector.

Special Matrices $\boldsymbol{A} \in \mathbb{R}^{n \times n}$

- Symmetric matrix: $\boldsymbol{A}^T = \boldsymbol{A}$

$$A_{ij} = A_{ji}$$

- Antisymmetric matrix: $\boldsymbol{A}^T = -\boldsymbol{A}$

$$A_{ij} = -A_{ji}$$

- Positive definite:

$$\boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{A} \boldsymbol{x} \geq 0, \qquad \forall \boldsymbol{x}$$

Special Matrices

- Orthogonal Matrices:
  - All rows of the matrix are orthogonal to each other;
  - All rows of the matrix have unit length.

$$\sum_j U_{ij} U_{kj} = \delta_{ik}$$

  - Rewrite in matrix form:

$$\boldsymbol{U}\boldsymbol{U}^{\boldsymbol{T}} = \boldsymbol{I}$$

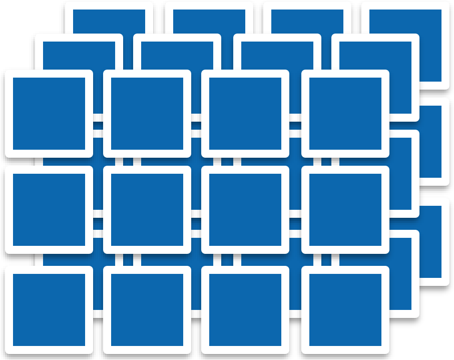
- Permutation Matrices:
  - There is only one 1 in each row or column:

$$P_{ij} = \begin{cases} 1 \; if \; and \; only \; if \; j = \pi(i) \\ 0. \qquad\qquad\qquad otherwise \end{cases}$$

Tensor

- A tensor is a collection of numbers labelled by indices.

- The rank of a tensor is the number of indices required to specify an entry in the tensor:
  - A vector is a rank–1 tensor;
  - A matrix is a rank–2 tensor.



Tensor

- Einstein summation convention:
  - Each index can appear at most twice in any term.
  - Repeated indices are implicitly summed over.
  - Each term must contain identical non-repeated indices.

$$M_{ij}v_j \equiv \sum_i M_{ij}v_j$$

Matrix multiplication between matrix M and vector v

$$M_{ij}u_jv_j$$

The index j appears three times in the first term.

$$T_{ijk}u_k + M_{ip}$$

The first term contains the non-repeated index j whereas the second term contains p.

Tensors in PyTorch

Creating Tensors

- Tensors are the central data abstraction in PyTorch.

Code	Output

`import torch`
`import math`
 	
`x = torch.empty(3, 4)`
`print(type(x))`
`print(x)`
	`<class 'torch.Tensor'>`
	`tensor([[-1.9609e-04,  4.5654e-41,  4.4115e-04,  0.0000e+00],`
	`        [ 8.0387e+26,  4.5654e-41,  8.9824e-06,  0.0000e+00],`
	`        [ 1.3431e-14,  0.0000e+00,  0.0000e+00,  0.0000e+00]])`

Creating Tensors

Code	Output

`import torch`
`import math`
 	
`x = torch.empty(3, 4)`
`print(type(x))`
`print(x)`
	`<class 'torch.Tensor'>`
	`tensor([[-1.9609e-04,  4.5654e-41,  4.4115e-04,  0.0000e+00],`
	`        [ 8.0387e+26,  4.5654e-41,  8.9824e-06,  0.0000e+00],`
	`        [ 1.3431e-14,  0.0000e+00,  0.0000e+00,  0.0000e+00]])`

- Create a tensor using one of the numerous factory methods attached to the torch module.
- The tensor itself is 2-dimensional, having 3 rows and 4 columns.
- The type of the object returned is torch.Tensor, which is an alias for torch.FloatTensor; by default, PyTorch tensors are populated with 32-bit floating point numbers.
- You will probably see some random-looking values when printing your tensor.
- The torch.empty() call allocates memory for the tensor, but does not initialize it with any values.
- What you're seeing is whatever was in memory at the time of allocation.

Creating Tensors

- Initialize your tensor with some value.

Code	Output

`zeros = torch.zeros(2, 3)`
`print(zeros)`
 	
`ones = torch.ones(2, 3)`
`print(ones)`
 	
`torch.manual_seed(1729)`
`random = torch.rand(2, 3)`
`print(random)`
	`tensor([[0., 0., 0.],`
	`        [0., 0., 0.]])`
	`tensor([[1., 1., 1.],`
	`        [1., 1., 1.]])`
	`tensor([[0.3126, 0.3791, 0.3087],`
	`        [0.0736, 0.4216, 0.0691]])`

Creating Tensors

- Manually setting your random number generator's seed assures reproducibility.

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td><pre>torch.manual_seed(1729)
random1 = torch.rand(2, 3)
print(random1)

random2 = torch.rand(2, 3)
print(random2)

torch.manual_seed(1729)
random3 = torch.rand(2, 3)
print(random3)

random4 = torch.rand(2, 3)
print(random4)</pre></td><td><pre>tensor([[0.3126, 0.3791, 0.3087],
        [0.0736, 0.4216, 0.0691]])
tensor([[0.2332, 0.4047, 0.2162],
        [0.9927, 0.4128, 0.5938]])
tensor([[0.3126, 0.3791, 0.3087],
        [0.0736, 0.4216, 0.0691]])
tensor([[0.2332, 0.4047, 0.2162],
        [0.9927, 0.4128, 0.5938]])</pre></td></tr>
</table>

Creating Tensors

- A pseudorandom number generator is a ***deterministic*** random bit generator:
  - An algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers.
  - The generated sequence is not truly random, it is completely determined by an initial value.

TORCH.MANUAL_SEED

torch.manual_seed(*seed*) [SOURCE]

Sets the seed for generating random numbers. Returns a *torch.Generator* object.

Parameters

seed (*int*) – The desired seed. Value must be within the inclusive range [*-0x8000_0000_0000_0000*, *0xffff_ffff_ffff_ffff*]. Otherwise, a RuntimeError is raised. Negative inputs are remapped to positive values with the formula *0xffff_ffff_ffff_ffff* + *seed*.

Return type

Generator

Creating Tensors

RELAXED SYSTEM LAB

- torch.shape contains a list of the extent of each dimension of a tensor.

Code	Output
`x = torch.empty(2, 2, 3)` `print(x.shape)`	`torch.Size([2, 2, 3])`
`empty_like_x = torch.empty_like(x)` `print(empty_like_x.shape)`	`torch.Size([2, 2, 3])`
`zeros_like_x = torch.zeros_like(x)` `print(zeros_like_x.shape)`	`torch.Size([2, 2, 3])`
`ones_like_x = torch.ones_like(x)` `print(ones_like_x.shape)`	`torch.Size([2, 2, 3])`
`rand_like_x = torch.rand_like(x)` `print(rand_like_x.shape)`	`torch.Size([2, 2, 3])`

Creating Tensors

- Create a tensor by specifying its data directly from a Python collection.

Code	Output

`some_constants = torch.tensor([[3.1415926, 2.71828], [1.61803, 0.0072897]])`
`print(some_constants)`
 	
`some_integers = torch.tensor((2, 3, 5, 7, 11, 13, 17, 19))`
`print(some_integers)`
 	
`more_integers = torch.tensor(((2, 4, 6), [3, 6, 9]))`
`print(more_integers)`
	`tensor([[3.1416, 2.7183],`
	`        [1.6180, 0.0073]])`
	`tensor([ 2,  3,  5,  7, 11, 13, 17, 19])`
	`tensor([[2, 4, 6],`
	`        [3, 6, 9]])`

Creating Tensors

- Setting the datatype of a tensor.

Code	Output

`a = torch.ones((2, 3), dtype=torch.int16)`
`print(a)`
 	
`b = torch.rand((2, 3), dtype=torch.float64) * 20.`
`print(b)`
 	
`c = b.to(torch.int32)`
`print(c)`
	`tensor([[1, 1, 1],`
	`        [1, 1, 1]], dtype=torch.int16)`
	`tensor([[ 0.9956,  1.4148,  5.8364],`
	`        [11.2406, 11.2083, 11.6692]], dtype=torch.float64)`
	`tensor([[ 0,  1,  5],`
	`        [11, 11, 11]], dtype=torch.int32)`

Creating Tensors

- Available data types include:
  - torch.bool
  - torch.int8
  - torch.uint8
  - torch.int16
  - torch.int32
  - torch.int64
  - torch.half
  - torch.float
  - torch.double
  - torch.bfloat

32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point 1	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
16-bit floating point 2	`torch.bfloat16`	`torch.BFloat16Tensor`	`torch.cuda.BFloat16Tenso r`

8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`

Math & Logic with PyTorch Tensors

RELAXED SYSTEM LAB

- Basic arithmetic: how tensors interact with simple scalars.

Code	Output

`ones = torch.zeros(2, 2) + 1`
`twos = torch.ones(2, 2) * 2`
`threes = (torch.ones(2, 2) * 7 - 1) / 2`
`fours = twos ** 2`
`sqrt2s = twos ** 0.5`
 	
`print(ones)`
`print(twos)`
`print(threes)`
`print(fours)`
`print(sqrt2s)`
	`tensor([[1., 1.],`
	`        [1., 1.]])`
	`tensor([[2., 2.],`
	`        [2., 2.]])`
	`tensor([[3., 3.],`
	`        [3., 3.]])`
	`tensor([[4., 4.],`
	`        [4., 4.]])`
	`tensor([[1.4142, 1.4142],`
	`        [1.4142, 1.4142]])`

Math & Logic with PyTorch Tensors

- Basic arithmetic operations between two tensors.

Code	Output

`powers2 = twos ** torch.tensor([[1, 2], [3, 4]])`
`print(powers2)`
 	
`fives = ones + fours`
`print(fives)`
 	
`dozens = threes * fours`
`print(dozens)`
	`tensor([[ 2.,  4.],`
	`        [ 8., 16.]])`
	`tensor([[5., 5.],`
	`        [5., 5.]])`
	`tensor([[12., 12.],`
	`        [12., 12.]])`

Math & Logic with PyTorch Tensors

- Tensor broadcasting:
  - Perform an operation between tensors that have similarities in their shapes;
  - E.g., a common example:
    - Multiply a tensor of learning weights by a batch of input tensors;
    - Apply the operation to each instance in the batch separately;
    - Return a tensor of identical shape.
- Four rule for broadcasting:
  - Each tensor must have at least one dimension (no empty tensors).
  - Comparing the dimension sizes of the two tensors, going ***from last to first***:
    - Each dimension must be equal, or
    - One of the dimensions must be of size 1, or
    - The dimension does not exist in one of the tensors.

Math & Logic with PyTorch Tensors

- Examples allow broadcasting.

Code	Output		

`a = torch.ones(4, 3, 2)`
 	
`# 3rd & 2nd dims identical to a, dim 1 absent`
`b = a * torch.rand(   3, 2)`
`print(b)`
 	
`# 3rd dim = 1, 2nd dim identical to a`
`c = a * torch.rand(   3, 1)`
`print(c)`
 	
`# 3rd dim identical to a, 2nd dim = 1`
`d = a * torch.rand(   1, 2)`
`print(d)`
	`tensor([[[0.6493, 0.2633],`
	`[0.4762, 0.0548],`
	`[0.2024, 0.5731]],`
	
	`[[0.6493, 0.2633],`
	`[0.4762, 0.0548],`
	`[0.2024, 0.5731]],`
	
	`[[0.6493, 0.2633],`
	`[0.4762, 0.0548],`
	`[0.2024, 0.5731]],`
	
	`[[0.6493, 0.2633],`
	`[0.4762, 0.0548],`
	`[0.2024, 0.5731]]])`
		`tensor([[[0.7191, 0.7191],`
		`[0.4067, 0.4067],`
		`[0.7301, 0.7301]],`
		
		`[[0.7191, 0.7191],`
		`[0.4067, 0.4067],`
		`[0.7301, 0.7301]],`
		
		`[[0.7191, 0.7191],`
		`[0.4067, 0.4067],`
		`[0.7301, 0.7301]],`
		
		`[[0.7191, 0.7191],`
		`[0.4067, 0.4067],`
		`[0.7301, 0.7301]]])`
			`tensor([[[0.6276, 0.7357],`
			`[0.6276, 0.7357],`
			`[0.6276, 0.7357]],`
			
			`[[0.6276, 0.7357],`
			`[0.6276, 0.7357],`
			`[0.6276, 0.7357]],`
			
			`[[0.6276, 0.7357],`
			`[0.6276, 0.7357],`
			`[0.6276, 0.7357]],`
			
			`[[0.6276, 0.7357],`
			`[0.6276, 0.7357],`
			`[0.6276, 0.7357]]])`

Math & Logic with PyTorch Tensors

RELAXED SYSTEM LAB

- Examples attempt at broadcasting that will fail.

Code

```
a = torch.ones(4, 3, 2)

b = a * torch.rand(4, 3)    # dimensions must match last-to-first

c = a * torch.rand(   2, 3) # both 3rd & 2nd dims different

d = a * torch.rand((0, ))   # can't broadcast with an empty tensor
```

Altering Tensors in Place

- Most binary operations on tensors will return a third, new tensor that takes a region of memory distinct from the other tensors.
- Most of the math functions have a version with an appended underscore (_) that will alter a tensor in place.

Code

```
a = torch.tensor([0, math.pi / 4, math.pi / 2, 3 * math.pi / 4])
print('a:')
print(a)
print(torch.sin(a))   # this operation creates a new tensor in memory
print(a)              # a has not changed

b = torch.tensor([0, math.pi / 4, math.pi / 2, 3 * math.pi / 4])
print('\nb:')
print(b)
print(torch.sin_(b))  # note the underscore
print(b)              # b has changed
```

Altering Tensors in Place

- Set up an out argument to specify a tensor to receive the output. If the out tensor is the correct shape and dtype, this can happen without a new memory allocation.

Code	Output

<pre>a = torch.rand(2, 2)
b = torch.rand(2, 2)
c = torch.zeros(2, 2)
old_id = id(c)
 	
print(c)
d = torch.matmul(a, b, out=c)
print(c)   # contents of c have changed
 	
assert c is d   # test c & d are same object, not just containing equal values
assert id(c) == old_id # make sure that our new c is the same object as the old one
 	
torch.rand(2, 2, out=c) # works for creation too!
print(c)   # c has changed again
assert id(c) == old_id # still the same object!</pre>
	<pre>tensor([[0., 0.],
	        [0., 0.]])
	tensor([[0.3653, 0.8699],
	        [0.2364, 0.3604]])
	tensor([[0.0776, 0.4004],
	        [0.9877, 0.0352]])</pre>

Copying Tensors

- As with any object in Python, assigning a tensor to a variable makes the variable a label of the tensor, and does not copy it.
- If you want a separate copy of the data to work on? The clone() method is there for you.

Code	Output

`a = torch.ones(2, 2)`
`b = a.clone()`
 	
`assert b is not a       # different objects in memory...`
`print(torch.eq(a, b))  # ...but still with the same contents!`
 	
`a[0][1] = 561           # a changes...`
`print(b)                # ...but b is still all ones`
	`tensor([[True, True],`
	`        [True, True]])`
	`tensor([[1., 1.],`
	`        [1., 1.]])`

Moving to GPU

- Define string or torch device handle to move the tensor to GPU.
- To do computation involving two or more tensors, all of the tensors must be on the ***same*** device.

Code	Output

<pre>if torch.cuda.is_available():
    my_device = torch.device('cuda’)
    x = torch.rand(2, 2,   device='cuda’)
    print(gpu_rand)
else:
    my_device = torch.device('cpu')
print('Device: {}'.format(my_device))
 	
y = torch.rand(2, 2, device=my_device)
print(y)</pre>
	<pre>tensor([[0.3344, 0.2640],
	        [0.2119, 0.0582]], device='cuda:0')
	
	Device: cuda
	tensor([[0.0024, 0.6778],
	        [0.2441, 0.6812]], device='cuda:0')</pre>

Manipulating Tensor Shapes - Squeeze

torch.squeeze(*input*, *dim=None*) → Tensor

Returns a tensor with all specified dimensions of `input` of size *1* removed.

For example, if *input* is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then the *input.squeeze()* will be of shape: $(A \times B \times C \times D)$.

When `dim` is given, a squeeze operation is done only in the given dimension(s). If *input* is of shape: $(A \times 1 \times B)$, `squeeze(input, 0)` leaves the tensor unchanged, but `squeeze(input, 1)` will squeeze the tensor to the shape $(A \times B)$.

- **NOTE**

The returned tensor shares the storage with the input tensor, so changing the contents of one will change the contents of the other.

- **WARNING**

If the tensor has a batch dimension of size 1, then *squeeze(input)* will also remove the batch dimension, which can lead to unexpected errors. Consider specifying only the dims you wish to be squeezed.

The squeeze() method has the in-place versions squeeze_().

Manipulating Tensor Shapes - Unsqueeze

RELAXED SYSTEM LAB

`torch.unsqueeze(input, dim)` → Tensor

Returns a new tensor with a dimension of size one inserted at the specified position.

The returned tensor shares the same underlying data with this tensor.

A `dim` value within the range `[-input.dim() - 1, input.dim() + 1)` can be used. Negative `dim` will correspond to `unsqueeze()` applied at `dim = dim + input.dim() + 1`.

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the index at which to insert the singleton dimension

The unsqueeze() method has the in-place versions unsqueeze_().

Manipulating Tensor Shapes - Reshape

RELAXED SYSTEM LAB

torch.reshape(*input*, *shape*) → Tensor

Returns a tensor with the same data and number of elements as `input`, but with the specified shape. When possible, the returned tensor will be a view of `input`. Otherwise, it will be a copy. Contiguous inputs and inputs with compatible strides can be reshaped without copying, but you should not depend on the copying vs. viewing behavior.

See `torch.Tensor.view()` on when it is possible to return a view.

A single dimension may be -1, in which case it's inferred from the remaining dimensions and the number of elements in `input`.

Parameters

- **input** (*Tensor*) – the tensor to be reshaped
- **shape** (*tuple of int*) – the new shape

reshape() will return a *view* on the tensor to be changed: a separate tensor object looking at the same underlying region of memory. That means any change made to the source tensor will be reflected in the view on that tensor, unless you clone() it.

Einstein Notation in PyTorch

RELAXED SYSTEM LAB

Code

```
a = torch.rand(2,3)
b = torch.rand(3,4)
c = torch.einsum("ik,kj->ij", a, b)
```

- Einstein summation in PyTorch:
  - *free index:* index on the right-hand side (e.g., i, j in the above example).
  - *summation index*: index only on the left-hand side, index to be summed over (e.g., k in the above example).
- <u>Execution:</u>
  - <u>Repeated indices among different input operands are multiplied.</u>
  - <u>Summation indices are summed over.</u>
  - <u>The indices on the output side can be permutated.</u>
  - <u>If the right-hand side is ignored, the indices that appear only once on the left-hand side will be placed on the right-hand side by default.</u>

TORCH.EINSUM

`torch.einsum(equation, *operands) → Tensor` [SOURCE]

Sums the product of the elements of the input `operands` along dimensions specified using a notation based on the Einstein summation convention.

Einsum allows computing many common multi-dimensional linear algebraic array operations by representing them in a short-hand format based on the Einstein summation convention, given by `equation`. The details of this format are described below, but the general idea is to label every dimension of the input `operands` with some subscript and define which subscripts are part of the output. The output is then computed by summing the product of the elements of the `operands` along the dimensions whose subscripts are not part of the output. For example, matrix multiplication can be computed using einsum as *torch.einsum("ij,jk->ik", A, B)*. Here, j is the summation subscript and i and k the output subscripts (see section below for more details on why).

Equation:

The `equation` string specifies the subscripts (letters in [*a-zA-Z*]) for each dimension of the input `operands` in the same order as the dimensions, separating subscripts for each operand by a comma (','), e.g. *'ij,jk'* specify subscripts for two 2D operands. The dimensions labeled with the same subscript must be broadcastable, that is, their size must either match or be *1*. The exception is if a subscript is repeated for the same input operand, in which case the dimensions labeled with this subscript for this operand must match in size and the operand will be replaced by its diagonal along these dimensions. The subscripts that appear exactly once in the `equation` will be part of the output, sorted in increasing alphabetical order. The output is computed by multiplying the input `operands` element-wise, with their dimensions aligned based on the subscripts, and then summing out the dimensions whose subscripts are not part of the output.

Optionally, the output subscripts can be explicitly defined by adding an arrow ('->') at the end of the equation followed by the subscripts for the output. For instance, the following equation computes the transpose of a matrix multiplication: 'ij,jk->ki'. The output subscripts must appear at least once for some input operand and at most once for the output.

Ellipsis ('...') can be used in place of subscripts to broadcast the dimensions covered by the ellipsis. Each input operand may contain at most one ellipsis which will cover the dimensions not covered by subscripts, e.g. for an input operand with 5 dimensions, the ellipsis in the equation *'ab...c'* cover the third and fourth dimensions. The ellipsis does not need to cover the same number of dimensions across the `operands` but the 'shape' of the ellipsis (the size of the dimensions covered by them) must broadcast together. If the output is not explicitly defined with the arrow ('->') notation, the ellipsis will come first in the output (left-most dimensions), before the subscript labels that appear exactly once for the input operands. e.g. the following equation implements batch matrix multiplication *'...ij,...jk'*.

A few final notes: the equation may contain whitespaces between the different elements (subscripts, ellipsis, arrow and comma) but something like *'. . .'* is not valid. An empty string *''* is valid for scalar operands.

Einstein Notation in PyTorch

RELAXED SYSTEM LAB

- The dimensions labeled with the same subscript must be broadcastable, that is, their size must either match or be 1.

Code	Output

`a_scaler = torch.ones(1)`
`a_vec = torch.ones(3)`
`B = torch.arange(9).reshape(3,3).float()`
 	
`print(B)`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`

`torch.einsum("i,ij",a_scaler,B)`	`tensor([ 9., 12., 15.])`
`torch.einsum("i,ij",a_vec,B)`	`tensor([ 9., 12., 15.])`
`torch.einsum("j,ij",a_scaler,B)`	`tensor([ 3., 12., 21.])`
`torch.einsum("j,ij",a_vec,B)`	`tensor([ 3., 12., 21.])`

Einstein Notation in PyTorch

- The subscripts that appear exactly once in the equation will be part of the output, sorted <u>in increasing alphabetical order</u>.

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td>

```
A = torch.eye(3)
B = torch.arange(9).reshape(3,3).float()

print(A)

print(B)

torch.einsum("ij,jk",A,B)

torch.einsum("kj,ji",A,B)
```

</td><td>

```
# A
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
# B
tensor([[0., 1., 2.],
        [3., 4., 5.],
        [6., 7., 8.]])

tensor([[0., 1., 2.],
        [3., 4., 5.],
        [6., 7., 8.]])

tensor([[0., 3., 6.],
        [1., 4., 7.],
        [2., 5., 8.]])
```

</td></tr>
</table>

Einstein Notation in PyTorch

- Optionally, the output subscripts can be *explicitly* defined by adding an arrow ('->') at the end of the equation followed by the subscripts for the output.

Code	Output

`A = torch.eye(3)`
`B = torch.arange(9).reshape(3,3).float()`
 	
`print(A)`
	`# A`
	`tensor([[1., 0., 0.],`
	`        [0., 1., 0.],`
	`        [0., 0., 1.]])`

`print(B)`	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`

`torch.einsum("kj,ji",A,B)`	`tensor([[0., 3., 6.],`
	`        [1., 4., 7.],`
	`        [2., 5., 8.]])`

`torch.einsum("kj,ji->ki",A,B)`	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`

Einstein Notation in PyTorch

RELAXED SYSTEM LAB

- Ellipsis ('…') can be used in place of subscripts to broadcast the dimensions covered by the ellipsis.

Code	Output
`A = torch.randn(2, 3, 4, 5)` `torch.einsum('...ij->...ji', A).shape`	`# A.shape:` torch.Size([2, 3, 5, 4])

Einstein Notation in PyTorch

- If a subscript is repeated for the same input operand, in which case the dimensions labeled with this subscript for this operand must match in size and the operand will be replaced by its diagonal along these dimensions.
- Example: Get the matrix's diagonal elements.

Code	Output

`B = torch.arange(9).reshape(3,3).float()`
 	
`print(B)`
 	
`torch.einsum("ii->i",B)`
 	
`torch.einsum("ii",B)`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`
	
	`tensor([0., 4., 8.])`
	
	`tensor(12.)`

Einstein Notation in PyTorch

- Example: Matrix Transpose.

Code	Output

`B = torch.arange(9).reshape(3,3).float()`
 	
`print(B)`
 	
`torch.einsum("ij->ji",B)`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`
	
	`tensor([[0., 3., 6.],`
	`        [1., 4., 7.],`
	`        [2., 5., 8.]])`

Einstein Notation in PyTorch

- Example: Matrix Summation.

Code	Output

`B = torch.arange(9).reshape(3,3).float()`
 	
`print(B)`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`

`torch.einsum("ij->",B)`	`tensor(36.)`
`torch.einsum("ij->i",B)`	`tensor([ 3., 12., 21.])`
`torch.einsum("ij->j",B)`	`tensor([ 9., 12., 15.])`

Einstein Notation in PyTorch

- Example: Matrix element-wise multiplication.

Code	Output

`A = torch.eye(3)`
`B = torch.arange(9).reshape(3,3).float()`
 	
`print(A)`
 	
`print(B)`
 	
`torch.einsum("ij,ij->ij",A,B)`
	`# A`
	`tensor([[1., 0., 0.],`
	`        [0., 1., 0.],`
	`        [0., 0., 1.]])`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`
	
	`tensor([[0., 0., 0.],`
	`        [0., 4., 0.],`
	`        [0., 0., 8.]])`

Einstein Notation in PyTorch

- Example: dot product.

Code	Output
`a = torch.arange(1,5).float()` `b = torch.arange(4).float()`	`# a` `tensor([1., 2., 3., 4.])`
`print(a)`	`# b` `tensor([0., 1., 2., 3.])`
`Print(b)`	`tensor(20.)`
`torch.einsum(”i,i ->",a,b)`	

Einstein Notation in PyTorch

RELAXED SYSTEM LAB

- Example: outer product.

Code	Output

`a = torch.arange(1,5).float()`
`b = torch.arange(4).float()`
 	
`print(a)`
 	
`Print(b)`
 	
`torch.einsum("i,j -> ij",a,b)`
	`# a`
	`tensor([1., 2., 3., 4.])`
	
	`# b`
	`tensor([0., 1., 2., 3.])`
	
	`tensor([[ 0.,  1.,  2.,  3.],`
	`        [ 0.,  2.,  4.,  6.],`
	`        [ 0.,  3.,  6.,  9.],`
	`        [ 0.,  4.,  8., 12.]])`

Einstein Notation in PyTorch

- Example: Matrix multiplication.

Code	Output

`A = torch.eye(3)`
`B = torch.arange(9).reshape(3,3).float()`
 	
`print(A)`
 	
`print(B)`
 	
`torch.einsum("ij,jk->ik",A,B)`
	`# A`
	`tensor([[1., 0., 0.],`
	`        [0., 1., 0.],`
	`        [0., 0., 1.]])`
	`# B`
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`
	
	`tensor([[0., 1., 2.],`
	`        [3., 4., 5.],`
	`        [6., 7., 8.]])`

Einstein Notation in PyTorch

- Example: Batch matrix multiplication.

Code	Output

`A = torch.ones(2,3,2)`
`B = torch.arange(12).reshape(2,2,3).float()`
 	
`print(A)`
 	
`print(B)`
 	
`torch.einsum('ijk,ikl->ijl',A,B)`
	`# A`
	`tensor([[[1., 1.],`
	`         [1., 1.],`
	`         [1., 1.]],`
	`        [[1., 1.],`
	`         [1., 1.],`
	`         [1., 1.]]])`
	`# B`
	`tensor([[[ 0.,  1.,  2.],`
	`         [ 3.,  4.,  5.]],`
	`        [[ 6.,  7.,  8.],`
	`         [ 9., 10., 11.]]])`
	
	`tensor([[[ 3.,  5.,  7.],`
	`         [ 3.,  5.,  7.],`
	`         [ 3.,  5.,  7.]],`
	`        [[15., 17., 19.],`
	`         [15., 17., 19.],`
	`         [15., 17., 19.]]])`

References

- https://d2l.ai/
- https://www.dr-qubit.org/teaching/summation_delta.pdf
- https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html