

Deep Learning Programming

Lecture 2.1: Pytorch Introduction

Sangryul Jeon

School of Computer Science and Engineering

srjeonn@pusan.ac.kr

PART 2: Basic Operations on Tensors

1. Tensor의 모양변경2

- 1.1 cat() 함수를 활용한 Tensor들 간의 연결
- 1.2 expand() 메서드를 활용한 Tensor의 크기 확장
- 1.3 repeat() 메서드를 활용한 Tensor의 크기 확장
- 1.4 학습정리

2. Tensor의 기초 연산

- 2.1 Tensor의 산술연산
- 2.2 Tensor의 비교연산
- 2.3 Tensor의 논리연산
- 2.4 학습정리

1.

Tensor의 모양변경2

이번 챕터에서는 Tensor의 모양을 변경하는 함수와 메서드에 대해 살펴봅니다.

stack() 함수와 cat() 함수

- 4강에서 학습한 stack() 함수는 새로운 차원을 생성하여 Tensor들을 결합함
- 하지만 cat() 함수는 새로운 차원을 추가하는 것이 아닌 기존의 차원을 유지하면서 Tensor들을 연결함

1.1 cat() 함수를 활용한 Tensor들 간의 연결

1. Tensor의 모양변경2

cat() 함수를 활용한 Tensor들 간의 연결에 대한 표현

- $b = \text{torch.tensor}([[0, 1], [2, 3]])$
- $c = \text{torch.tensor}([[4, 5]])$
와 같이 연결하고자 하는 2개의 2-D Tensor를 생성함

$$b = \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array}$$

$$c = \begin{array}{|c|c|}\hline 4 & 5 \\ \hline \end{array}$$

cat() 함수를 활용한 Tensor들 간의 연결에 대한 표현

- dim = 0을 기준으로 Tensor b와 c를 연결하는 [코드 표현](#)
 - `d = torch.cat((b, c))`

$$d = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline 4 & 5 \\ \hline \end{array}$$

cat() 함수를 활용한 Tensor 간의 연결에 대한 표현

- dim = 1을 기준으로 Tensor b와 c를 연결하는 [코드 표현](#)
 - `e= torch.cat((b, c), 1)` → Error

0	1
2	3

4	5
---	---

cat() 함수를 활용한 Tensor 간의 연결에 대한 표현

- dim = 1을 기준으로 Tensor b와 c를 연결하는 코드 표현
 - `e = torch.cat((b, c.reshape(2, 1)), 1)`

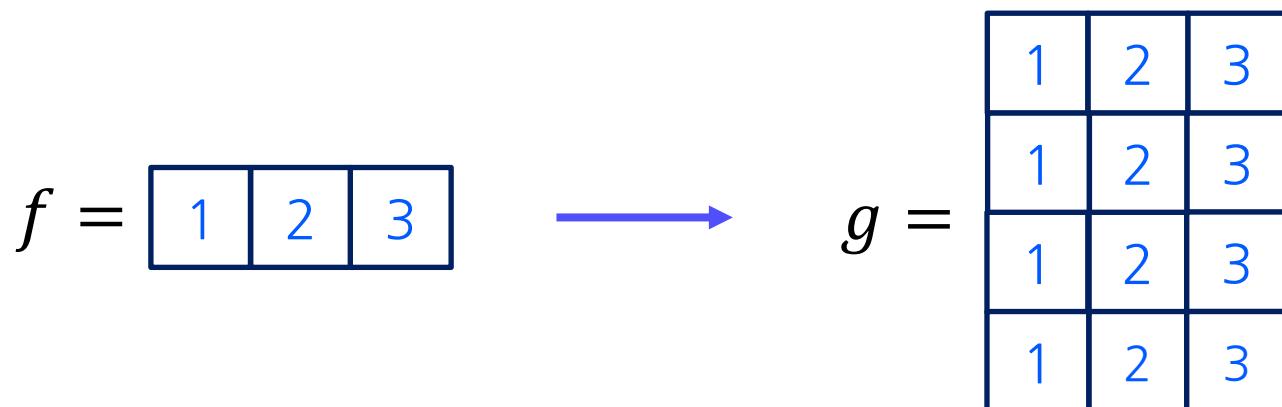
$$e = \begin{array}{|c|c|c|} \hline 0 & 1 & 4 \\ \hline 2 & 3 & 5 \\ \hline \end{array}$$

1.2 expand() 메서드를 활용한 Tensor의 크기 확장

1. Tensor의 모양변경2

expand() 메서드를 활용한 Tensor의 크기를 확장하는 표현

- 주어진 Tensor의 차원이 크기가 1일 때, 해당 차원의 크기를 확장함
- $f = \text{torch.tensor}([[1, 2, 3]])$ 인 크기가 (1, 3)인 2-D Tensor를 생성했을 때,
- 주어진 Tensor f 의 크기를 (4, 3)로 확장하는 코드 표현
 - $g = f.expand(4, 3)$



repeat() 메서드를 활용한 Tensor의 크기 확장

- expand() 메서드를 활용하는 경우 주어진 Tensor의 차원 중 일부의 크기가 1이어야 하는 제한점이 있음
- [repeat\(\) 메서드](#)는 Tensor의 요소들을 반복해서 크기를 확장하는데 사용하며, expand() 메서드와는 다르게 Tensor의 차원 중 일부의 크기가 1이어야 하는 제약이 없음
- 하지만 repeat() 메서드는 추가 메모리를 할당하기 때문에 메모리를 할당하지 않는 expand() 메서드보다 메모리 효율성이 떨어짐

repeat() 메서드를 활용한 Tensor의 크기를 확장하는 코드 표현

- `h = torch.tensor([[1, 2], [3, 4]])`인 2-D Tensor를 생성했을 때,
- 주어진 Tensor `h`를 `dim = 0` 축으로 2번 반복하고, `dim = 1`축으로 3번 반복하여 Tensor의 크기를 확장하는 [코드 표현](#)
 - `i = h.repeat(2, 3)`

1.3 repeat() 메서드를 활용한 Tensor의 크기 확장

1. Tensor의 모양변경2

repeat() 메서드를 활용한 Tensor의 크기를 확장하는 공간에서 표현

$$h = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$



$$i = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$$

Tensor의 모양변경2

- `cat()` 함수를 활용하면 차원의 축방향에 따라 Tensor들을 연결할 수 있다.
- `expand()` 메서드를 활용하면 주어진 Tensor의 차원 중 일부의 크기가 1일 때, 해당 차원의 크기를 확장할 수 있다.
- `repeat()` 메서드를 활용하면 Tensor의 요소들을 반복하는 방식으로 크기를 확장할 수 있다.

2.

Tensor의 기초 연산

이번 챕터에서는 Tensor의 산술 연산, 비교 연산, 논리 연산에 대해 살펴봅니다.

더하기 연산

- 크기가 동일한 두 Tensor a, b의 더하기 연산은 각 요소들을 더한 값으로 출력함
- a = torch.tensor([[1, 2],
[3, 4]]) b = torch.tensor([[1, 3],
[5, 7]]) 2-D Tensor를 생성했을 때,
 - 두 Tensor를 더하는 코드 표현: `torch.add(a, b)`

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 3 \\ \hline 5 & 7 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 5 \\ \hline 8 & 11 \\ \hline \end{array}$$

in-place 방식을 활용한 더하기 연산

- [in-place 방식](#)은 메모리를 절약하며 Tensor의 값을 업데이트할 수 있는 연산
- $a = \text{torch.tensor}([[1, 2], [3, 4]])$, $b = \text{torch.tensor}([[1, 3], [5, 7]])$ 2-D Tensor를 생성했을 때,
 - 두 Tensor를 in-place 방식으로 더하는 [코드 표현](#): `a.add_(b)`
- in-place 방식의 연산은 추가적인 메모리 할당이 필요 없기 때문에 메모리 사용량을 줄일 수 있다는 장점이 있지만, 이후에 배울 [autograd](#)와의 호환성 측면에서 문제를 일으킬 수 있음

크기가 다른 Tensor의 더하기 연산

- $c = \text{torch.tensor}([[1, 2], [4, 5]])$, $d = \text{torch.tensor}([1, 3])$ 크기가 다른 두 Tensor를 더하게 되면,
[4, 5]))
- Tensor d가 Tensor c와 같은 크기로 확장하여 요소들을 연산하게 됨
 - 두 Tensor를 더하는 코드 표현: `torch.add(c, d)`

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 3 \\ \hline 1 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 5 \\ \hline 5 & 8 \\ \hline \end{array}$$

2.1 Tensor의 산술연산

2. Tensor의 기초 연산

빼기 연산

- 크기가 동일한 두 Tensor e, f의 빼기 연산은 각 요소들을 뺀 값으로 출력함
 - e = torch.tensor([[2, 3], [4, 5]]) f = torch.tensor([[2, 4], [6, 8]]) 2-D Tensor를 생성했을 때,
 - Tensor f에서 Tensor e를 빼는 코드 표현: `torch.sub(f, e)`
 - Tensor f에서 Tensor e를 `in-place` 방식으로 빼는 코드 표현: `f.sub_(e)`

$$\begin{array}{|c|c|} \hline 2 & 4 \\ \hline 6 & 8 \\ \hline \end{array} - \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array}$$

크기가 다른 Tensor의 빼기 연산

- $g = \text{torch.tensor}([[2, 3], [5, 6]])$, $h = \text{torch.tensor}([[1], [4]])$, 크기가 다른 두 Tensor를 빼게 되면,
Tensor h가 Tensor g와 같은 크기로 확장한 후 연산이 이루어짐
- Tensor g에서 Tensor h를 빼는 코드 표현: `torch.sub(g, h)`

$$\begin{array}{|c|c|} \hline 2 & 3 \\ \hline 5 & 6 \\ \hline \end{array} - \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 4 & 4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 1 & 2 \\ \hline \end{array}$$

스칼라곱 연산

- 스칼라곱은 Tensor의 각 요소에 동일한 스칼라 값을 곱하는 연산을 의미함
- $i = 2$ 인 스칼라를 생성하고, $j = \text{torch.tensor}([[1, 2], [3, 4]])$ 2-D Tensor를 생성했을 때,
 - 스칼라 i 와 Tensor j 를 스칼라곱하는 코드 표현: `torch.mul(i, j)`

2·1	2·2
2·3	2·4

요소별 곱하기 연산(Hadamard product, Element wise product)

- 크기가 동일한 두 Tensor k, l의 요소별 곱하기 연산은 각 요소들을 곱한 값으로 출력함
- k = torch.tensor([[1, 2],
[1, 3]])
l = torch.tensor([[2, 3],
[1, 4]]) 2-D Tensor를 생성했을 때,
 - Tensor k와 Tensor l을 요소별로 곱하는 코드 표현: `torch.mul(k, l)`
 - Tensor k와 Tensor l을 in-place 방식으로 요소별로 곱하는 코드 표현: `k.mul_(l)`

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 1 & 3 \\ \hline \end{array} \circ \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 1 & 4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 6 \\ \hline 1 & 12 \\ \hline \end{array}$$

크기가 다른 Tensor의 요소별 곱하기 연산

- $m = \text{torch.tensor}([[1, 4], [7, 8]])$, $n = \text{torch.tensor}([2, 3])$ 크기가 다른 두 Tensor를 곱하게 되면,
Tensor n이 Tensor m과 같은 크기로 확장한 후 연산이 이루어짐
 - Tensor m과 Tensor n을 곱하는 코드 표현: [torch.mul\(m, n\)](#)

$$\begin{array}{|c|c|} \hline 1 & 4 \\ \hline 7 & 8 \\ \hline \end{array} \circ \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 12 \\ \hline 14 & 24 \\ \hline \end{array}$$

요소별 나누기 연산

- 크기가 동일한 두 Tensor o와 p의 요소별 나누기 연산은 각 요소들을 나눈 값으로 출력함
- $o = \text{torch.tensor}([[18, 9], [10, 4]])$, $p = \text{torch.tensor}([[6, 3], [5, 2]])$ 2-D Tensor를 생성했을 때,
 - Tensor o를 Tensor p로 나누는 코드 표현: `torch.div(o, p)`
 - Tensor o를 Tensor p로 *in-place* 방식으로 나누는 코드 표현: `o.div_(p)`

$$\begin{array}{|c|c|} \hline 18 & 9 \\ \hline 10 & 4 \\ \hline \end{array} \div \begin{array}{|c|c|} \hline 6 & 3 \\ \hline 5 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 3. & 3. \\ \hline 2. & 2. \\ \hline \end{array}$$

크기가 다른 Tensor의 요소별 나누기 연산

- $q = \text{torch.tensor}([[12, 8], [9, 4]])$, $r = \text{torch.tensor}([3, 2])$ 크기가 다른 두 Tensor를 나누게 되면,
[9, 4]]])
- Tensor r 이 Tensor q 와 같은 크기로 확장한 후 연산이 이루어짐
 - Tensor q 를 Tensor r 로 나누는 코드 표현: `torch.div(q, r)`

$$\begin{array}{|c|c|} \hline 12 & 8 \\ \hline 9 & 4 \\ \hline \end{array} \quad \div \quad \begin{array}{|c|c|} \hline 3 & 2 \\ \hline 3 & 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|} \hline 4. & 4. \\ \hline 3. & 2. \\ \hline \end{array}$$

2.1 Tensor의 산술연산

2. Tensor의 기초 연산

요소별 거듭제곱 연산

- $s = \text{torch.tensor}([[1, 2], [3, 4]])$ 2-D Tensor를 생성했을 때,
 - Tensor s 의 각 요소들에 대해 n 제곱을 계산하는 코드 표현: `torch.pow(s, n)`

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \quad \ast\ast \quad n \quad = \quad \begin{array}{|c|c|} \hline 1^n & 2^n \\ \hline 3^n & 4^n \\ \hline \end{array}$$

요소별 거듭제곱 연산

- Tensor t의 각각의 요소들을 Tensor u의 각각의 요소들의 값만큼 거듭제곱한 값으로 출력됨
- t = torch.tensor([[5, 4],
[3, 2]]) u = torch.tensor([[1, 2],
[3, 4]]) 2-D Tensor를 생성했을 때,
 - Tensor t의 요소들을 Tensor u의 대응요소들만큼 거듭제곱하는 [코드 표현](#): `torch.pow(t, u)`
 - in-place 방식으로 요소별 거듭제곱하는 [코드 표현](#): `t.pow_(u)`

$$\begin{array}{|c|c|} \hline 5 & 4 \\ \hline 3 & 2 \\ \hline \end{array} \quad ** \quad \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|} \hline 5 & 16 \\ \hline 27 & 16 \\ \hline \end{array}$$

2.1 Tensor의 산술연산

2. Tensor의 기초 연산

요소별 거듭제곱근 연산

- $s = \text{torch.tensor}([[1, 2], [3, 4]])$ 2-D Tensor를 생성했을 때,
 - Tensor s 의 각 요소들에 대한 n 제곱근을 계산하는 코드 표현: `torch.pow(s, 1/n)`

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \quad ** \quad \frac{1}{n} \quad = \quad \begin{array}{|c|c|} \hline \sqrt[n]{1} & \sqrt[n]{2} \\ \hline \sqrt[n]{3} & \sqrt[n]{4} \\ \hline \end{array}$$

2.2 Tensor의 비교연산

2. Tensor의 기초 연산

비교연산

- $v = \text{torch.tensor}([1, 3, 5, 7])$, $w = \text{torch.tensor}([2, 3, 5, 7])$ 1-D Tensor를 생성했을 때,
 - Tensor v 의 요소들과 Tensor w 의 대응요소들이 같은지를 비교하는 [코드 표현](#): `torch.eq(v, w)`
 - 결과는 Boolean Tensor로 출력함

$$v = \begin{array}{|c|c|c|c|} \hline 1 & 3 & 5 & 7 \\ \hline \end{array}$$

$$w = \begin{array}{|c|c|c|c|} \hline 2 & 3 & 5 & 7 \\ \hline \end{array}$$

2.2 Tensor의 비교연산

2. Tensor의 기초 연산

비교연산

- $v = \text{torch.tensor}([1, 3, 5, 7])$, $w = \text{torch.tensor}([2, 3, 5, 7])$ 1-D Tensor를 생성했을 때,
 - Tensor w 의 요소들과 Tensor v 의 대응요소들이 다른지를 비교하는 [코드 표현](#): `torch.ne(v, w)`

$$v = \begin{array}{|c|c|c|c|} \hline 1 & 3 & 5 & 7 \\ \hline \end{array}$$
$$w = \begin{array}{|c|c|c|c|} \hline 2 & 3 & 5 & 7 \\ \hline \end{array}$$

2.2 Tensor의 비교연산

2. Tensor의 기초 연산

비교연산

- $v = \text{torch.tensor}([1, 3, 5, 7])$, $w = \text{torch.tensor}([2, 3, 5, 7])$ 1-D Tensor를 생성했을 때,
 - Tensor v 의 요소들이 Tensor w 의 대응요소들보다 큰지를 비교하는 [코드 표현](#): `torch.gt(v, w)`

$$v = \begin{array}{|c|c|c|c|} \hline 1 & 3 & 5 & 7 \\ \hline \end{array}$$
$$w = \begin{array}{|c|c|c|c|} \hline 2 & 3 & 5 & 7 \\ \hline \end{array}$$

2.2 Tensor의 비교연산

2. Tensor의 기초 연산

비교연산

- $v = \text{torch.tensor}([1, 3, 5, 7])$, $w = \text{torch.tensor}([2, 3, 5, 7])$ 1-D Tensor를 생성했을 때,
 - Tensor v 의 요소들이 Tensor w 의 대응요소들보다 크거나 같은지를 비교하는
코드 표현: `torch.ge(v, w)`

$$v = \begin{array}{|c|c|c|c|} \hline 1 & 3 & 5 & 7 \\ \hline \end{array}$$

$$w = \begin{array}{|c|c|c|c|} \hline 2 & 3 & 5 & 7 \\ \hline \end{array}$$

비교연산

- $v = \text{torch.tensor}([1, 3, 5, 7])$, $w = \text{torch.tensor}([2, 3, 5, 7])$ 1-D Tensor를 생성했을 때,
 - Tensor v 의 요소들이 Tensor w 의 대응요소들보다 작은지를 비교하는
코드 표현: `torch.lt(v, w)`
 - Tensor v 의 요소들이 Tensor w 의 대응요소들보다 작거나 같은지를 비교하는
코드 표현: `torch.le(v, w)`

기초논리

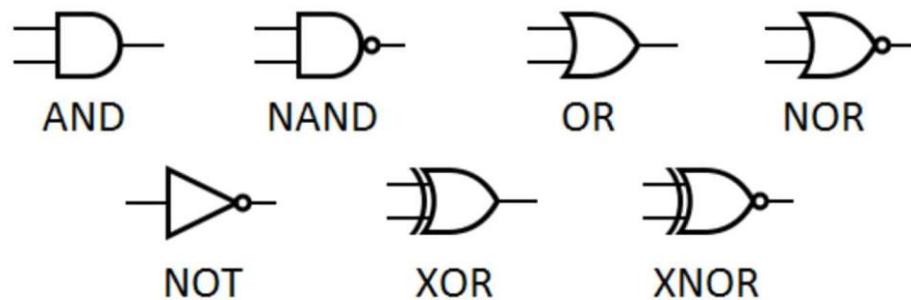
- 논리 연구란 타당하지 않은 논증으로부터 타당한 논증을 구별하는 데 사용되는 원리와 방법을 의미함(Lin & Lin, 1973)
- 이러한 논리는 전문적으로 사용되고 있는 용어 “명제”로부터 시작함
- 명제란 참, 거짓 중 어느 한 경우이되 동시에 양쪽은 아닌 서술문(주장)을 뜻함(Lin & Lin, 1973)
 - $2+1$ 은 3 과 같다(참인 명제)
 - $2>3$ (거짓인 명제)
 - 새우버터구이는 맛있다(명제 X)

2.3 Tensor의 논리연산

2. Tensor의 기초 연산

기초논리

- 여기서는 논리곱(AND), 논리합(OR), 배타적 논리합(XOR) 연산에 대해서 살펴보고자 함
- 이러한 논리곱, 논리합, 배타적 논리합 연산은 전자회로를 구성하는 가장 기본적인 논리 게이트가 됨
 - 게이트란 디지털 회로의 기본 구성요소 중 하나



출처: <https://fineproxy.org/ko/wiki/or-logic-gate/>

논리곱(AND) 연산

- 논리곱(AND)이란 입력된 신호가 모두 참일 때, 출력이 참이 되는 연산
- x 명제가 참이고 y 명제가 참일 때, $x \wedge y$ 명제는 참
 - ex) $x: 2+1=3$ (참), $y: 1+2=3$ (참)일 때, $x \wedge y : 2+1=3$ 이고 $1+2=3$ (참)
- x 명제가 참이고 y 명제가 거짓일 때, $x \wedge y$ 명제는 거짓
 - ex) $x: 2+1=3$ (참), $y: 1+2=4$ (거짓)일 때, $x \wedge y : 2+1=3$ 이고 $1+2=4$ (거짓)
- x 명제가 거짓이고 y 명제가 참일 때, $x \wedge y$ 명제는 거짓
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=3$ (참)일 때, $x \wedge y : 2+1=5$ 이고 $1+2=3$ (거짓)
- x 명제가 거짓이고 y 명제가 거짓일 때, $x \wedge y$ 명제는 거짓
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=4$ (거짓)일 때, $x \wedge y : 2+1=5$ 이고 $1+2=4$ (거짓)

2.3 Tensor의 논리연산

2. Tensor의 기초 연산

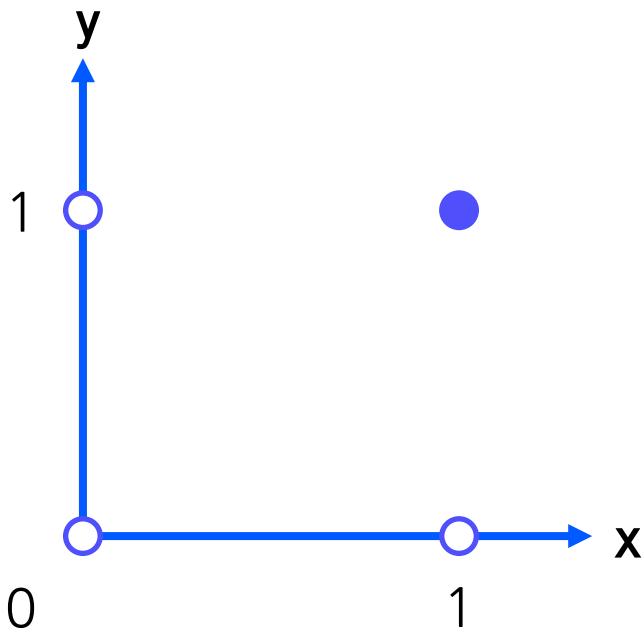
논리곱(AND) 연산

- 논리곱 연산에 대한 진리표 표현

x	y	$x \wedge y$
T	T	T
T	F	F
F	T	F
F	F	F

논리곱(AND) 연산

- 논리곱 연산에 대한 [그래프 표현](#)



논리곱(AND) 연산

- x 명제와 y 명제에 대한 진리표의 결과를 출력하는 [코드 표현](#)
 - `torch.logical_and(x, y)`

논리합(OR) 연산

- 논리합(OR)이란 입력된 신호 중 하나라도 참일 때, 출력이 참이 되는 연산
- x 명제가 참이고 y 명제가 참일 때, $x \vee y$ 명제는 참
 - ex) $x: 2+1=3$ (참), $y: 1+2=3$ (참)일 때, $x \vee y : 2+1=3$ 또는 $1+2=3$ (참)
- x 명제가 참이고 y 명제가 거짓일 때, $x \vee y$ 명제는 참
 - ex) $x: 2+1=3$ (참), $y: 1+2=4$ (거짓)일 때, $x \vee y : 2+1=3$ 또는 $1+2=4$ (참)
- x 명제가 거짓이고 y 명제가 참일 때, $x \vee y$ 명제는 참
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=3$ (참)일 때, $x \vee y : 2+1=5$ 또는 $1+2=3$ (참)
- x 명제가 거짓이고 y 명제가 거짓일 때, $x \vee y$ 명제는 거짓
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=4$ (거짓)일 때, $x \vee y : 2+1=5$ 또는 $1+2=4$ (거짓)

2.3 Tensor의 논리연산

2. Tensor의 기초 연산

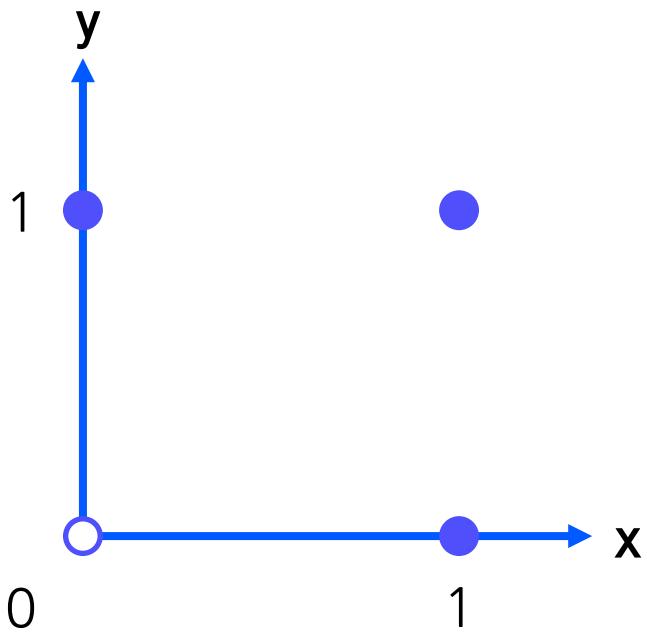
논리합(OR) 연산

- 논리합 연산에 대한 진리표 표현

x	y	$x \vee y$
T	T	T
T	F	T
F	T	T
F	F	F

논리합(OR) 연산

- 논리합 연산에 대한 그래프 표현



논리합(OR) 연산

- x 명제와 y 명제에 대한 진리표의 결과를 출력하는 코드 표현
 - `torch.logical_or(x, y)`

배타적 논리합(XOR) 연산

- 배타적 논리합(XOR)이란 입력된 신호가 하나만 참일 때, 출력이 참이 되는 연산
- x 명제가 참이고 y 명제가 참일 때, $x \oplus y$ 명제는 거짓
 - ex) $x: 2+1=3$ (참), $y: 1+2=3$ (참)일 때, $x \oplus y : 2+1=3$ 이고 $1+2=3$ 중 하나가 참(거짓)
- x 명제가 참이고 y 명제가 거짓일 때, $x \oplus y$ 명제는 참
 - ex) $x: 2+1=3$ (참), $y: 1+2=4$ (거짓)일 때, $x \oplus y : 2+1=3$ 이고 $1+2=4$ 중 하나가 참(참)
- x 명제가 거짓이고 y 명제가 참일 때, $x \oplus y$ 명제는 참
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=3$ (참)일 때, $x \oplus y : 2+1=5$ 이고 $1+2=3$ 중 하나가 참(참)
- x 명제가 거짓이고 y 명제가 거짓일 때, $x \oplus y$ 명제는 거짓
 - ex) $x: 2+1=5$ (거짓), $y: 1+2=4$ (거짓)일 때, $x \oplus y : 2+1=5$ 이고 $1+2=4$ 중 하나가 참(거짓)

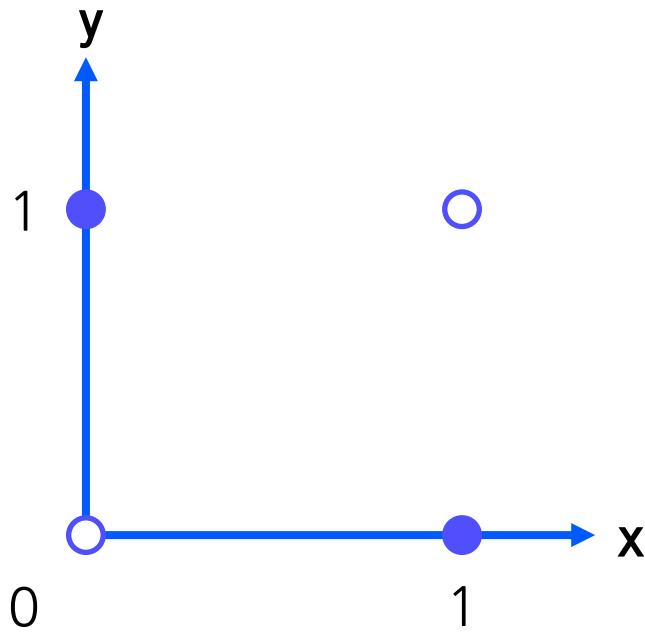
배타적 논리합(XOR) 연산

- 배타적 논리합 연산에 대한 진리표 표현

x	y	$x \oplus y$
T	T	F
T	F	T
F	T	T
F	F	F

배타적 논리합(XOR) 연산

- 배타적 논리합 연산에 대한 [그래프 표현](#)



배타적 논리합(XOR) 연산

- x 명제와 y 명제에 대한 진리표의 결과를 출력하는 코드 표현
 - `torch.logical_xor(x, y)`

Tensor의 기초 연산

- Tensor들의 요소별 산술연산 방법에는 함수 방식과 in-place 방식이 있다.
- Tensor들 간의 요소들을 비교할 수 있는 비교연산으로 `torch.eq()`, `torch.ne()`, `torch.gt()`, `torch.ge()`, `torch.lt`, `torch.le()` 등이 있다.
- 명제를 포함하고 있는 Tensor들의 논리를 연산할 수 있는 방법으로 논리곱, 논리합, 배타적 논리합 등이 있다.

Thank you

Prof. Jeon, Sangryul

Computer Vision Lab.

Pusan National University, Korea

Tel: +82-51-510-2423

Web: <http://sr-jeon.github.io/>

E-mail: srjeonn@pusan.ac.kr