Tensor Manipulation

상태	Basic ML
담당자	

💡 매 연산마다 Tensor의 크기를 인지하고 있어야 제대로 된 구현이 가능하다!

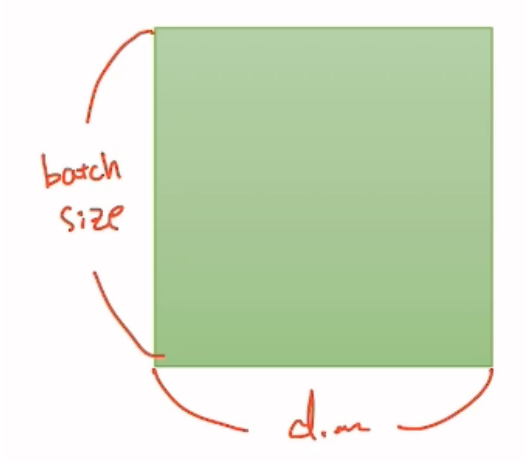
PyTorch Tensor Shape Convention
PyTorch Tensor
Broadcasting
Frequently Used Operations in PyTorch
- Matrix Multiplication vs Multiplication
- Mean
- Sum
- Max and Argmax
- View(Reshape)
- Squeeze
- Unsqueeze
- Concatenate
- Stacking
- Type Casting
- Ones and Zeros
- In-place Operation

🤔 Torch tensor의 연산에 관한 직관적인 이해(3차원)

PyTorch Tensor Shape Convention

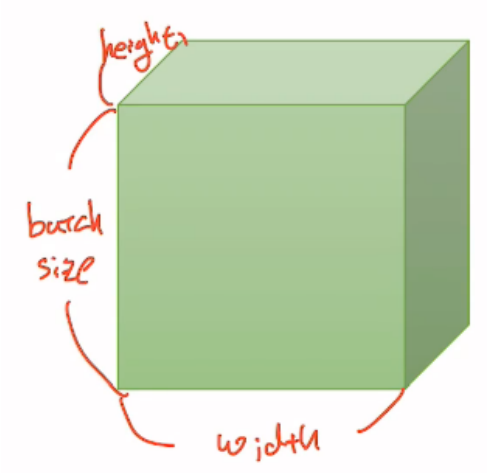
(세로 값, 가로 값, 깊이) 순으로 표현하는 것이 PyTorch의 Tensor Shape Convention

- 2D Tensor(Typical Simple Setting) : $|t| = (batch\ size, dimension)$



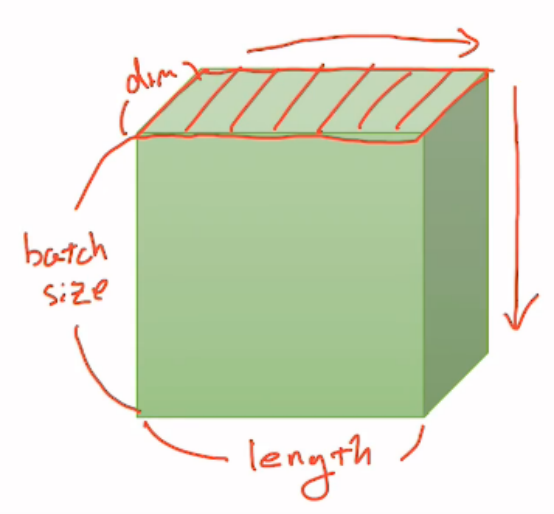
- 일반적인 CV 분야에서의 응용 : $|t| = (bath\ size, width, height)$

  '(가로 x 세로) 로 구성된 이미지가 bath size만큼 쌓여 있다'



- NLP에서의 응용 : $|t| = (bath\ size, length, dimension)$

  '문장이 특정 단어 벡터의 차원으로 bath size 만큼 쌓여 있다'



▼ NLP 분야의 3D 텐서 예제로 이해하기

```
[['나는 사과를 좋아해'], ['나는 바나나를 좋아해'], ['나는 사과를 싫어해'], ['나는 바나나를 싫어해']]
```

(4 x 1 x 2) Tensor

문장들의 리스트를 인풋으로 받아도 파이썬은 문장이 몇개의 단어로 구성되어 있는지 이해하지 못한다.

```
[['나는', '사과를', '좋아해'], ['나는', '바나나를', '좋아해'], ['나는', '사과를', '싫어해'], ['나는', '바나나를', '싫어해']]
```

(4 x 3 x 2) Tensor

띄어쓰기를 기준으로 단어별로 재구성

```
'나는' = [0.1, 0.2, 0.9]
'사과를' = [0.3, 0.5, 0.1]
'바나나를' = [0.3, 0.5, 0.2]
'좋아해' = [0.7, 0.6, 0.5]
'싫어해' = [0.5, 0.6, 0.7]
```

각 단어를 3차원의 벡터로 변환한 다음 대입하면

```
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.5, 0.6, 0.7]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.5, 0.6, 0.7]]]
```

(4 x 3 x 3) Tensor

연산을 수행할 batch size를 2로 설정하면 주어진 데이터는 batch#1, batch#2로 나누어진다.

```
#batch1
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.7, 0.6, 0.5]]]
```

```
#batch2
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.5, 0.6, 0.7]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.5, 0.6, 0.7]]]
```

(2 x 3 x 3), (2 x 3 x 3) Tensor : (batch size, 문장 길이, 단어 벡터의 차원)

PyTorch Tensor

💡 PyTorch에서 tensor를 다루는 방식은 NumPy와 상당히 유사하다

```
t = torch.FloatTensor([[1., 2., 3.],
                       [4., 5., 6.],
                       [7., 8., 9.],
                       [10., 11., 12.]
                      ])
print(t)
```

```
print(t.dim())  # rank
print(t.size()) # shape
print(t[:, 1]) # n x m 형태의 Tensor에서 m의 1번째 인덱스에 해당하는 모든 element
print(t[:, 1].size())
print(t[:, :-1]) # n x m 형태의 Tensor에서 m의 맨 마지막 인덱스에 해당하는 것만 뺀 모든 element

>>>
2
torch.Size([4, 3])
tensor([ 2.,  5.,  8., 11.])
torch.Size([4])
tensor([[ 1.,  2.],
        [ 4.,  5.],
        [ 7.,  8.],
        [10., 11.]])
```

```
t = torch.FloatTensor([[[[1, 2, 3, 4],
                         [5, 6, 7, 8],
                         [9, 10, 11, 12]],
                        [[13, 14, 15, 16],
                         [17, 18, 19, 20],
                         [21, 22, 23, 24]]
                        ]])

print(t.dim())  # rank  = 4
print(t.size()) # shape = (1, 2, 3, 4)

>>
4
torch.Size([1, 2, 3, 4])
```

▼ NumPy, PyTorch Shape 예제

Shape 예제	분류	설명	샘플
(8,)	1차원 텐서	배열 형태로 8개의 요소로 구성되어 있음	[1 2 3 4 5 6 7 8]
(2,4)	2차원 텐서	두 개 그룹으로 나누고 각 그룹은 4개의 요소를 갖고 있는 구조	[[1, 2, 3, 4], [5, 6, 7, 8]]
(2,2,2)	3차원 텐서	2개의 구룹으로 나누고, 각 그룹 별로 각각 4개의 요소로 2개 그룹으로 분할 됨	[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

4차원 텐서 `torch.Size([1, 2, 3, 4])` : 3x4 tensor가 2개의 그룹으로 분할된 것이 1개의 그룹으로 묶여 있음

```
t = torch.FloatTensor([[[[1, 2, 3, 4],
                         [5, 6, 7, 8],
                         [9, 10, 11, 12]],
                        [[13, 14, 15, 16],
                         [17, 18, 19, 20],
                         [21, 22, 23, 24]]
                        ]])
```

예를 들어

```
t = torch.FloatTensor([[[[1, 2, 3, 4],
                         [5, 6, 7, 8],
                         [9, 10, 11, 12]],

                        [[13, 14, 15, 16],
                         [17, 18, 19, 20],
                         [21, 22, 23, 24]]
                        ],

                        [[[1, 2, 3, 4],
                          [5, 6, 7, 8],
                          [9, 10, 11, 12]],

                        [[13, 14, 15, 16],
                         [17, 18, 19, 20],
                         [21, 22, 23, 24]]]])
```

의 경우 3x4 tensor가 2개의 그룹으로 분할된 것이 다시 2개의 그룹으로 묶여 있으므로 `torch.Size([2, 2, 3, 4])`

Broadcasting

💡 다른 크기의 tensor간 연산이 가능하도록 자동적으로 크기를 맞추어 주는 기능
단, 아무런 주의 없이 broadcasting을 수행하면 디버깅하기 어려워질 수 있으니 유의!

```
# Vector + scalar
m1 = torch.FloatTensor([[1, 2]]) # 1 x 2
```

```
m2 = torch.FloatTensor([3]) # 3 -> [[3, 3]]
print(m1 + m2)

>>
tensor([[4., 5.]])
```

```
# 2 x 1 Vector + 1 x 2 Vector
m1 = torch.FloatTensor([[1, 2]]) # 2 x 1
m2 = torch.FloatTensor([[3], [4]]) # 1 x 2
print(m1 + m2) # -> 2 x 2

>>
tensor([[4., 5.],
        [5., 6.]])
```

Frequently Used Operations in PyTorch

Matrix Multiplication vs Multiplication

```
print()
print('-------------')
print('Mul vs Matmul')
print('-------------')
m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1.matmul(m2)) # 2 x 1

m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1 * m2) # 2 x 2
print(m1.mul(m2))

>>
-------------
Matmul vs Mul
-------------
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor([[ 5.],
        [11.]])
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor([[1., 2.],
        [6., 8.]])
tensor([[1., 2.],
        [6., 8.]])
```

`matmul`로 실행한 전자의 경우 행렬곱(matrix multiplication)이 수행되었으며, `mul` 로 실행한 후자(element-wise-multiplication)의 경우 broadcasting이 수행되었다.

Mean

> `torch.mean`(***input***, , ***dtype=None***) → **Tensor**
> `Tensor.mean`(***dim=None***, ***keepdim=False***, , ***dtype=None***) → **Tensor**

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)

>>
tensor([[1., 2.],
        [3., 4.]])
```

평균을 원하는 차원에 대해서 수행할 수 있다.

```
print(t.mean()) # 전체 차원에 대해
print(t.mean(dim=0)) # 첫 번째 차원(행) 제거 -> 열 차원만 보존
print(t.mean(dim=1)) # 두 번째 차원(열) 제거 -> 행 차원만 보존
print(t.mean(dim=-1)) # 마지막 차원(여기서는 열) 제거

>>
tensor(2.5000)
tensor([2., 3.])
tensor([1.5000, 3.5000])
tensor([1.5000, 3.5000])
```

Sum

> `torch.sum`(*input*, , *dtype=None*) → **Tensor**
> `Tensor.sum`(*dim=None*, *keepdim=False*, *dtype=None*) → **Tensor**

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)

>>
tensor([[1., 2.],
        [3., 4.]])

print(t.sum())
print(t.sum(dim=0))
print(t.sum(dim=1))
print(t.sum(dim=-1))


>>

tensor(10.)
tensor([4., 6.])
tensor([3., 7.])
tensor([3., 7.])
```

Max and Argmax

> `torch.max`(*input*) → **Tensor** 혹은 `Tensor.max`(*dim=None*, *keepdim=False*) : 텐서의 가장 큰 `값` 을 리턴
> `torch.argmax`(*input*) → **LongTensor 혹은**

> `Tensor.argmax` (***dim=None*, *keepdim=False***) → **LongTensor** : 텐서의 가장 큰 값의 `인덱스`를 리턴

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)

>>
tensor([[1., 2.],
        [3., 4.]])
```

```
torch.max(t, dim=0) # 이 예시의 경우, 수직 방향으로 연산

>>
(tensor([3., 4.]), tensor([1, 1]))

torch.max(t, dim=1)

>>
(tensor([2., 4.]), tensor([1, 1]))
```

View(Reshape)

> `view` 함수를 이용해 텐서의 형태를 변경해 줄 수 있다.

```
t = np.array([[[0, 1, 2],
               [3, 4, 5]],

              [[6, 7, 8],
               [9, 10, 11]]])
ft = torch.FloatTensor(t)
print(ft.shape)

>>
torch.Size([2, 2, 3])
```

이 텐서의 사이즈는 2 x 2 x 3인데, 암튼 n x n 으로 바꾸면서 두 번째 차원이 3개의 element를 가지게끔 하고 싶다.

```
print(ft.view([-1, 3]))
print(ft.view([-1, 3]).shape)

>>
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
torch.Size([4, 3])
```

암튼 n x n x n 으로 바꾸면서 두, 세번째 차원이 각각 1개와 3개의 element를 가지게끔 하고 싶다.

```
print(ft.view([-1, 1, 3]))
print(ft.view([-1, 1, 3]).shape)

>>
```

```
tensor([[[ 0.,  1.,  2.]],

        [[ 3.,  4.,  5.]],

        [[ 6.,  7.,  8.]],

        [[ 9., 10., 11.]]])
torch.Size([4, 1, 3])
```

Squeeze

> Dimension의 element가 1개만 있는 경우에, 해당 dimension을 없애주는 역할을 한다.

```
ft = torch.Tensor([[[0, 1, 2]]
                   ])
print(ft.shape)
print(ft.squeeze().shape)

>>
torch.Size([1, 1, 3])
torch.Size([3])
```

`squeeze(dim=)` 옵션을 통해 해당 dimension의 element가 1개일 경우에 squeeze하도록 옵션을 설정할 수 있다.

```
ft = torch.Tensor([[[0, 1, 2]]])
print(ft.shape)
print(ft.squeeze(dim=1).shape)

>>
torch.Size([1, 1, 3])
torch.Size([1, 3])
```

Unsqueeze

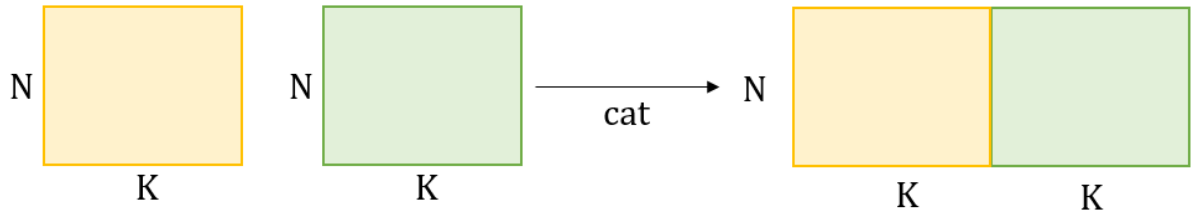
> 원하는 dimension에 대해 unsqueeze한다. 원하는 dimension에 `1`을 넣는다.

```
ft = torch.Tensor([0, 1, 2])
print(ft.unsqueeze(dim=0))
print(ft.unsqueeze(dim=0).shape)

>>
tensor([[0., 1., 2.]])
torch.Size([1, 3])
```

Concatenate

> 텐서를 이어 붙인다. 이어붙이는 텐서의 크기가 꼭 같을 필요는 없다.

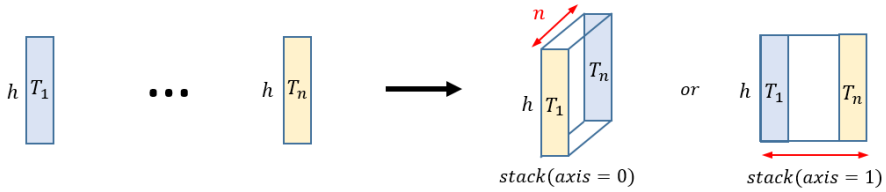



```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[5, 6, 7], [8, 9, 10]])
torch.cat([x, y], dim=1)

>>
tensor([[ 1,  2,  5,  6,  7],
        [ 3,  4,  8,  9, 10]])
```

Stacking

> `torch.stack` (***tensors***, ***dim=0***, , ***out=None***) → **Tensor**
> Concat을 편리하게 이용할 수 있는 함수, '텐서를 지정하는 dimension으로 확장하여 쌓아라!'



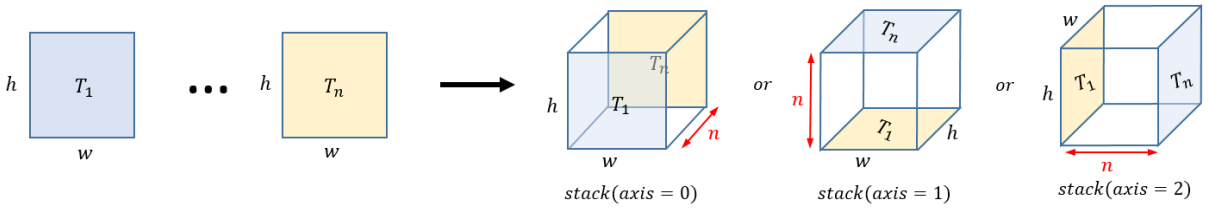

```
x = torch.FloatTensor([1, 4])
y = torch.FloatTensor([2, 5])
z = torch.FloatTensor([3, 6])

print(torch.stack([x, y, z]))  # stack axis=0 인 경우

>>
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])

print(torch.stack([x, y, z], dim=1)) # stack axis=1 인 경우

>>
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```



Type Casting

```
lt = torch.LongTensor([1, 2, 3, 4]) # 64-bit integer (signed)
print(lt)

>>
tensor([1, 2, 3, 4])

bt = torch.ByteTensor([True, False, False, True])
print(bt)

>>
tensor([1, 0, 0, 1], dtype=torch.uint8)
```

여기서 `.long` 은 정수 값, `.float`은 실수 값을 의미한다.

Ones and Zeros

> 💡 팁! `torch.ones_like` 혹은 `torch.zeros_like` 함수를 이용하면 같은 디바이스(CPU, GPU, multi-GPU)에 텐서를 선언해 준다.

0 혹은 1로만 가득찬 동일 사이즈, 동일 shape의 텐서를 생성한다.

```
x = torch.FloatTensor([[0, 1, 2], [2, 1, 0]])

print(torch.ones_like(x))
print(torch.zeros_like(x))

>>
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In-place Operation

> operation 뒤에 `_`을 붙임으로써 메모리에 새로 선언하지 않고 정답 값을 기존의 텐서에 넣는다. 메모리 추가 할당이 없으므로 메모리 효율화와 속도 향상을 기대할 수 있다.
>
> ex) a += b 처럼 변수에 할당된 주소를 재사용한다.

```
x = torch.FloatTensor([[1, 2], [3, 4]])
print(x.mul(2.))
print(x)
>>
tensor([[2., 4.],
        [6., 8.]])
tensor([[1., 2.],
        [3., 4.]])

print(x.mul_(2.))
print(x)
>>
tensor([[2., 4.],
```

```
        [6., 8.]])
tensor([[2., 4.],
        [6., 8.]])
```

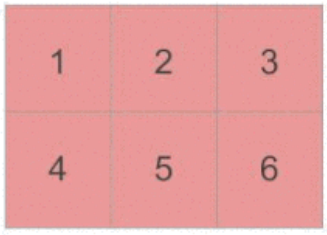
🤔 Torch tensor의 연산에 관한 직관적인 이해(3차원)

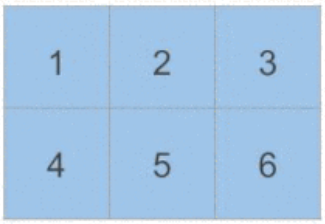
```
>>
y = torch.tensor([
     [
       [1, 2, 3],
       [4, 5, 6]
     ],
     [
       [1, 2, 3],
       [4, 5, 6]
     ],
     [
       [1, 2, 3],
       [4, 5, 6]
     ]
   ])
>>
y.shape
torch.Size([3, 2, 3])
```

2 x 3 크기의 matrix가 3개 생성된 것을 확인할 수 있다. 즉, 여기서의 `dim` 은 `0`, `1`, `2` 이고 (이를 인덱스처럼 생각해 보면 쉽다) 각각의 옵션을 넣어 `torch.sum()` 을 수행한 결과를 예상해 보자

- `torch.sum(y, dim=0)`

  여기서의 0번째 인덱스, 3개의 2D 텐서를 서로 더한다. (Collapse 3 elements)






```
>>
tensor([[ 3,  6,  9],
        [12, 15, 18]])
```

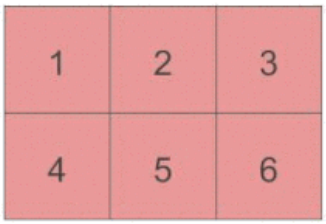
- `torch.sum(y, dim=1)`

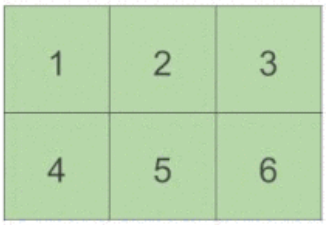
  여기서의 1번째 인덱스, 행에 대해 축소한다 (Collapse the rows)


```
>>
tensor([[5, 7, 9],
        [5, 7, 9],
        [5, 7, 9]])
```

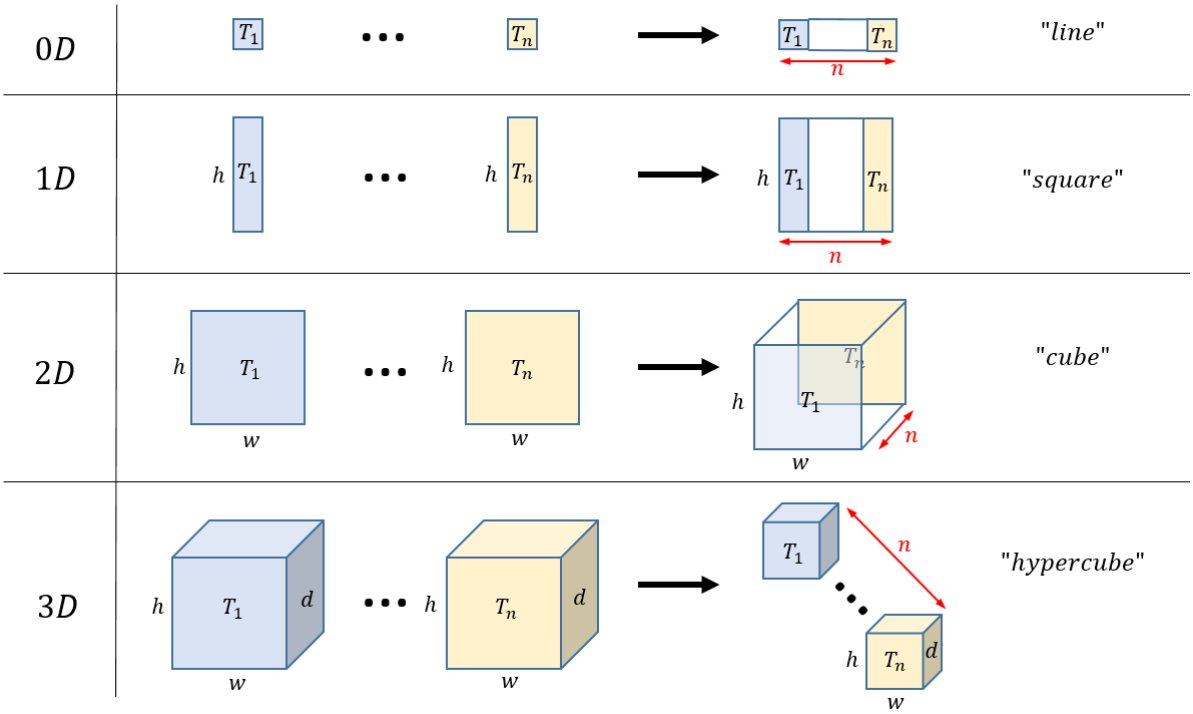
- `torch.sum(y, dim=2)`

  Collapse the columns
  여기서의 2번째 인덱스, 열에 대해 축소한다 (Collapse the columns)






```
>>
tensor([[ 6, 15],
        [ 6, 15],
        [ 6, 15]])
```

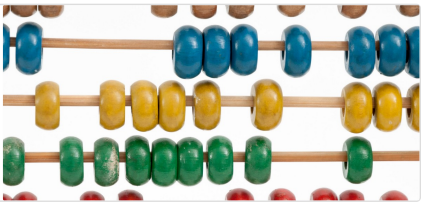


Source

Understanding dimensions in PyTorch

However, th more important problem was, as I said, the direction of each dimension. Here's what I mean. When we describe the shape of a 2D tensor, we say that it contains some rows and some columns. So for a 2x3 tensor we've 2 rows and 3 columns: We specify at first the

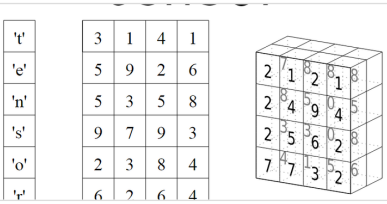
https://towardsdatascience.com/understanding-dimensions-in-pytorch-6edf9972d3be



torch.sum()에 대한 정리

그렇다면 3차원에서의 torch.sum()은 어떻게 작용되는가? 2x3x4의 배열을 생성해보자. 3x4크기의 matrix가 2개 생성된 것을 확인 할 수 있다. 그럼 2차원일 때 처럼 axis=0을 넣고 계산해보자. 뭔가 생각했던 계산이 나오지 않았다. 처음에 axis=0으로 넣었을 때 필자는 column들끼리 계산인 가 나와야 한다

https://velog.io/@reversesky/torch.sum%EC%97%90-%EB%8C%80%ED%95%B4-%EC%95%8C%EC%95%84%EB%B3%B4%EC%9E%90



Use of torch.stack()

t1 = torch.tensor([1,2,3]) t2 = torch.tensor([4,5,6]) t3 = torch.tensor([7,8,9]) torch.stack((t1,t2,t3),dim=1) When implementing the torch.stack(), I can't understand how stacking is done for dif...

https://stackoverflow.com/questions/69220221/use-of-torch-stack