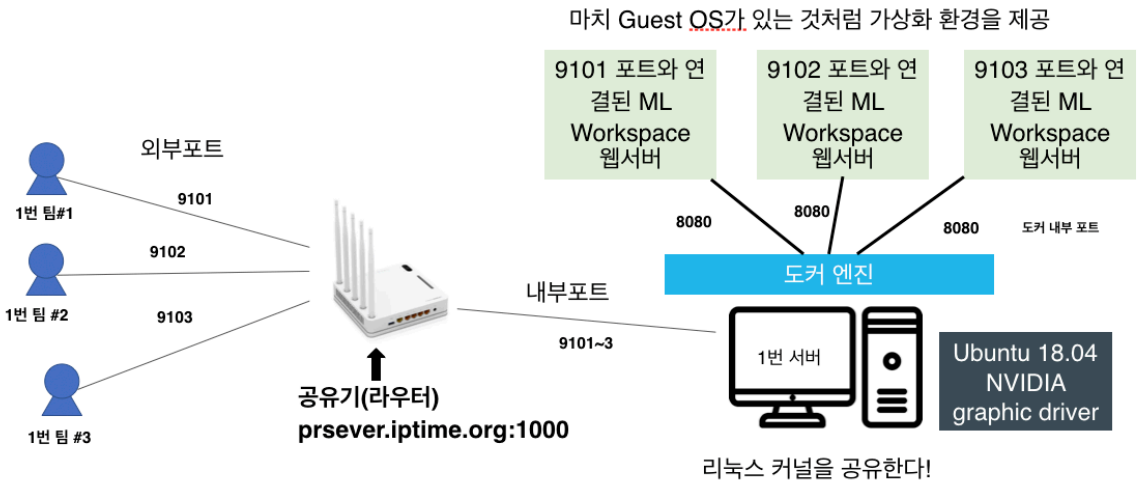
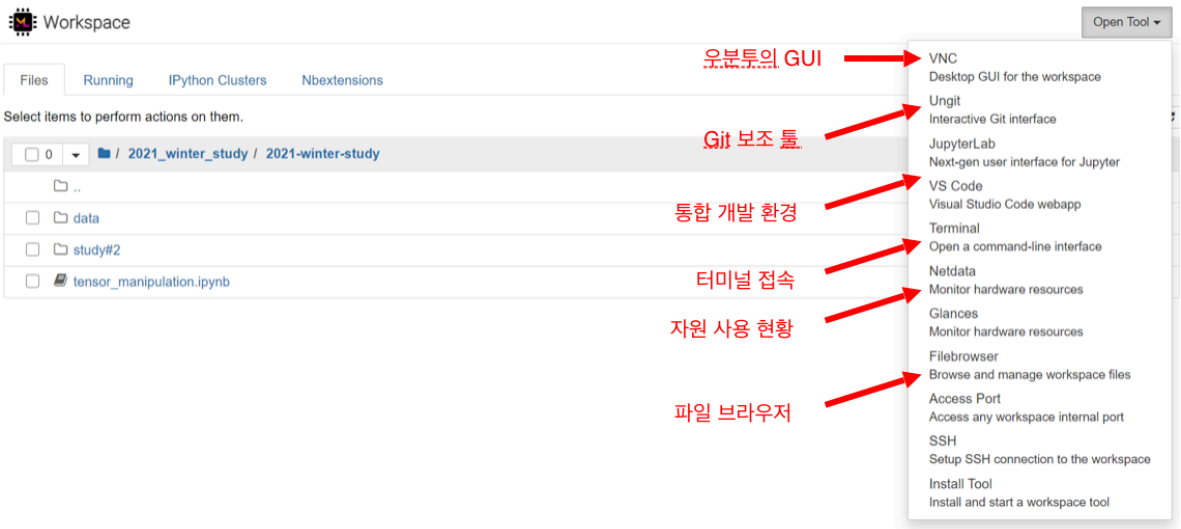
# ML workspace_02

## 웹서버 접속 프로세스

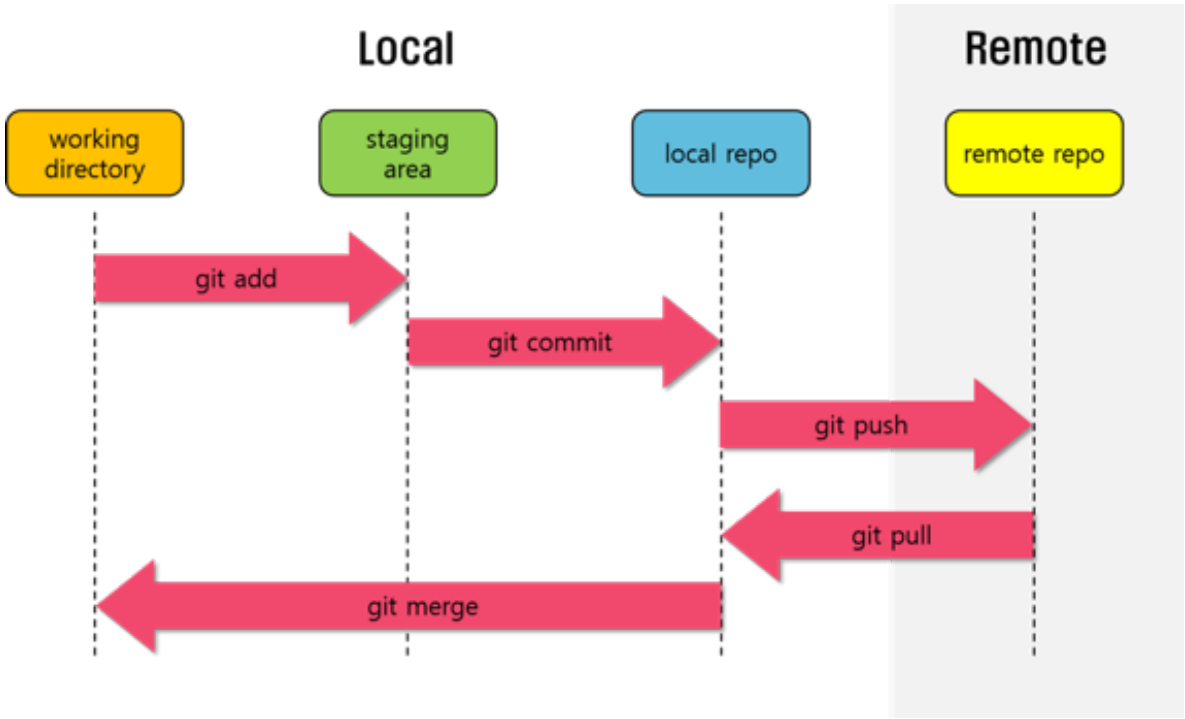



첫번째 포트 포워딩은 공유기에서, 두번째 포트 포워딩은 도커 명령을 통해 수행됨

## ML workspace 웹서버의 전체적인 기능들




## GIT실습코드를 가져와 자신의 프로젝트에 병합




**open folder -> workspace -> 2021_winter_study** // VS code에서 폴더 열기

**git status** // 자신의 git 상태 확인하기

**git add .** //

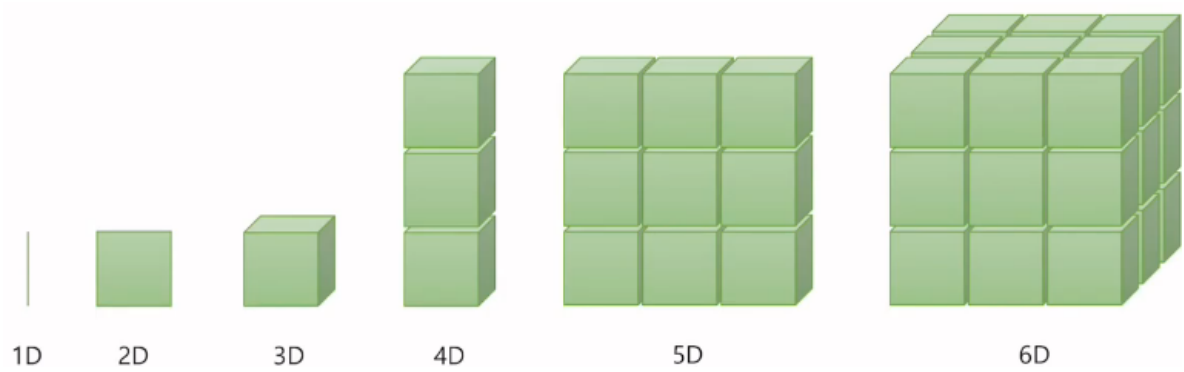
**git commit –m "메시지"** // 커밋하기

**git remote add origin https://github.com/KUNYOUNGLEE/2021-winter-study.git** // 자신의 코드에 병합하기

**git pull origin main** //

**git clone https://github.com/KUNYOUNGLEE/2021-winter-study.git** // 코드 복제하기 (workspace에서)

## PyTorch에서 Tensor다루기

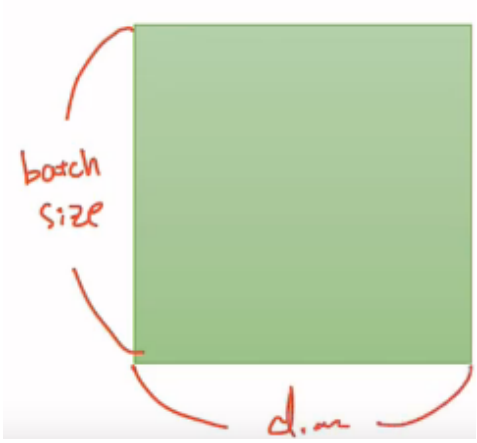



딥 러닝에서 가장 기본적인 단위는 **벡터, 행렬, 텐서**

차원이 없는 값 - **스칼라**
**1차원**으로 구성된 값 - **벡터**
**2차원**으로 구성된 값 - **행렬(Matrix)**
**3차원**으로 구성된 값 - **텐서(Tensor)**

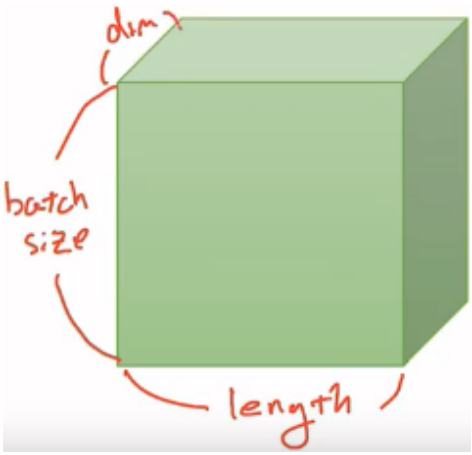
4차원은 3차원의 텐서를 위로 쌓아 올린 모습
5차원은 그 4차원을 다시 옆으로 확장한 모습
6차원은 5차원을 뒤로 확장한 모습

## 2차원 Tensor




// 행의 크기가 batch size, 열의 크기가 dim

## 3차원 Tensor




// 세로 batch size, 가로 너비(width), 차원 (dim)

### 1) 1차원 배열

## 1D Array with NumPy

[5]
```
t = np.array([0., 1., 2., 3., 4., 5., 6.]) # 1차원 배열
print(t)
```
```
[0. 1. 2. 3. 4. 5. 6.]
```

[6]
```
print('Rank  of t: ', t.ndim) # 차원의 수
print('Shape of t: ', t.shape) # 1행 7열
```
```
Rank  of t:  1
Shape of t:  (7,)
```

[7]
```
print('t[0] t[1] t[-1] = ', t[0], t[1], t[-1]) # Element 요소
print('t[2:5] t[4:-1]  = ', t[2:5], t[4:-1])   # Slicing 슬라이싱
print('t[:2] t[3:]     = ', t[:2], t[3:])      # Slicing 슬라이싱
```
```
t[0] t[1] t[-1] =  0.0 1.0 6.0
t[2:5] t[4:-1]  =  [2. 3. 4.] [4. 5.]
t[:2] t[3:]     =  [0. 1.] [3. 4. 5. 6.]
```

**인덱싱의 개수 = 차원의 수**

**.ndim** // 몇 차원인지 출력
**.shape** // 크기를 출력 **# (7, )은 (1, 7)을 의미함**

# **슬라이싱**은 [시작 번호 : 끝 번호]라고 했을 때, **끝 번호에 해당하는 것은 포함 X**

## 2) 2차원 배열

2D Array with NumPy

[8]
```
t = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.], [10., 11., 12.]]) # 2차원 배열
print(t)
```

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

[9]
```
print('Rank  of t: ', t.ndim) # 2차원
print('Shape of t: ', t.shape) # 4행 3열
```

```
Rank  of t:  2
Shape of t:  (4, 3)
```

[10]
```
print(t[:,2]) # 3열의 모든 속성 출력
print(type(t[:,2])) # 배열의 타입 출력
print(np.shape(t[:,2])) # 2열의 행 개수 출력
```

```
[ 3.  6.  9. 12.]
<class 'numpy.ndarray'>
(4,)
```

## 3) 1차원 PyTorch

# PyTorch is like NumPy (but better)

## 1D Array with PyTorch

[11]

```
t = torch.Tensor([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

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

[12]

```
print(t.dim())  # rank
print(t.shape)  # shape
print(t.size()) # shape
print(t[0], t[1], t[-1])  # Element
print(t[2:5], t[4:-1])    # Slicing
print(t[:2], t[3:])       # Slicing
```

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

**dim()** // 현재 텐서의 차원 출력
**shape(), size()** // 크기 출력

### 4) 2차원 PyTorch

[13]

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

```
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]])
```

[14]

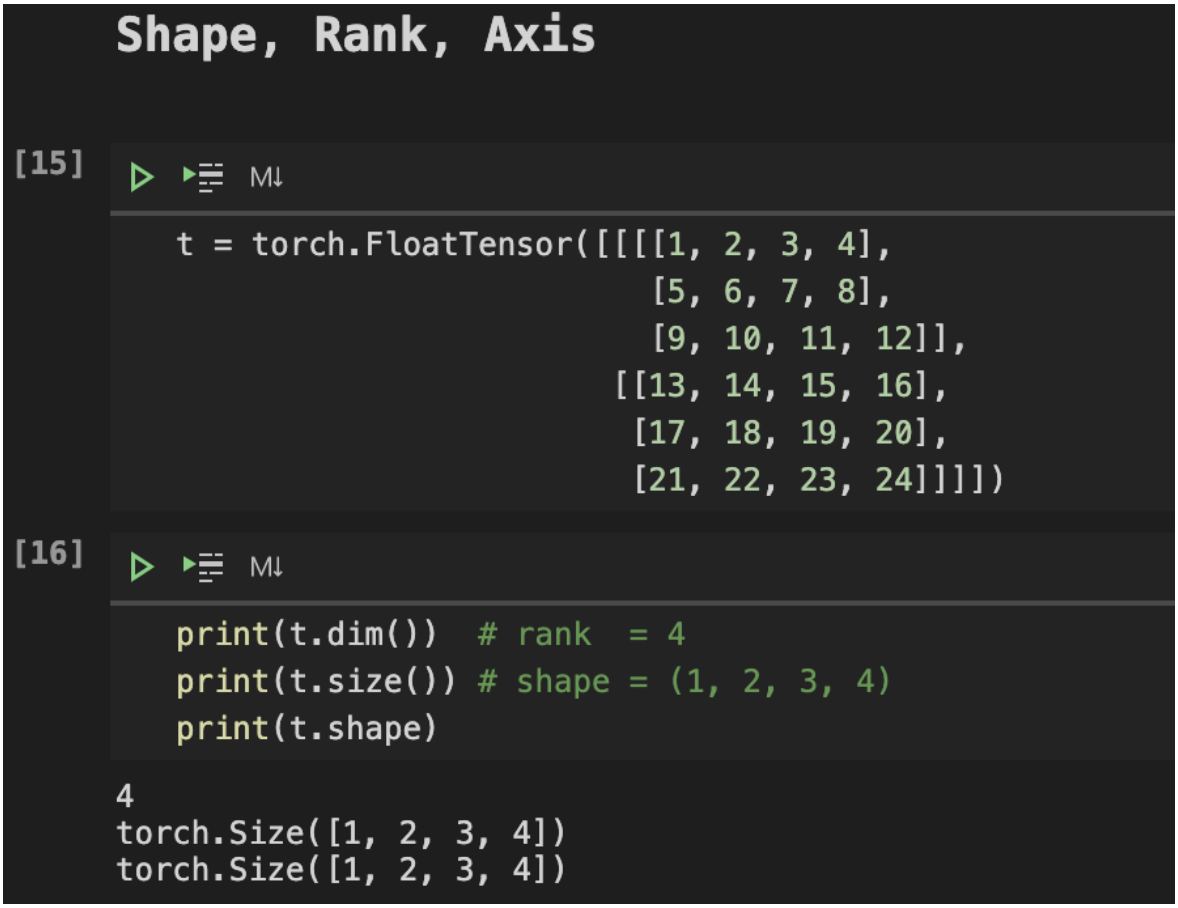
```
print(t.dim())  # rank
print(t.size()) # shape
print(t[:, 1])
print(t[:, 1].size())
print(t[:, :-1])
```

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

# 현재 텐서의 차원은 2차원, (4, 3)크기를 가짐
**print(t[:, 1])** // 첫번째 차원을 전체 선택한 상황에서 두번째 차원의 첫번째 것만 가져온다.
(=두번째 열에 있는 모든 값을 가져온 상황)

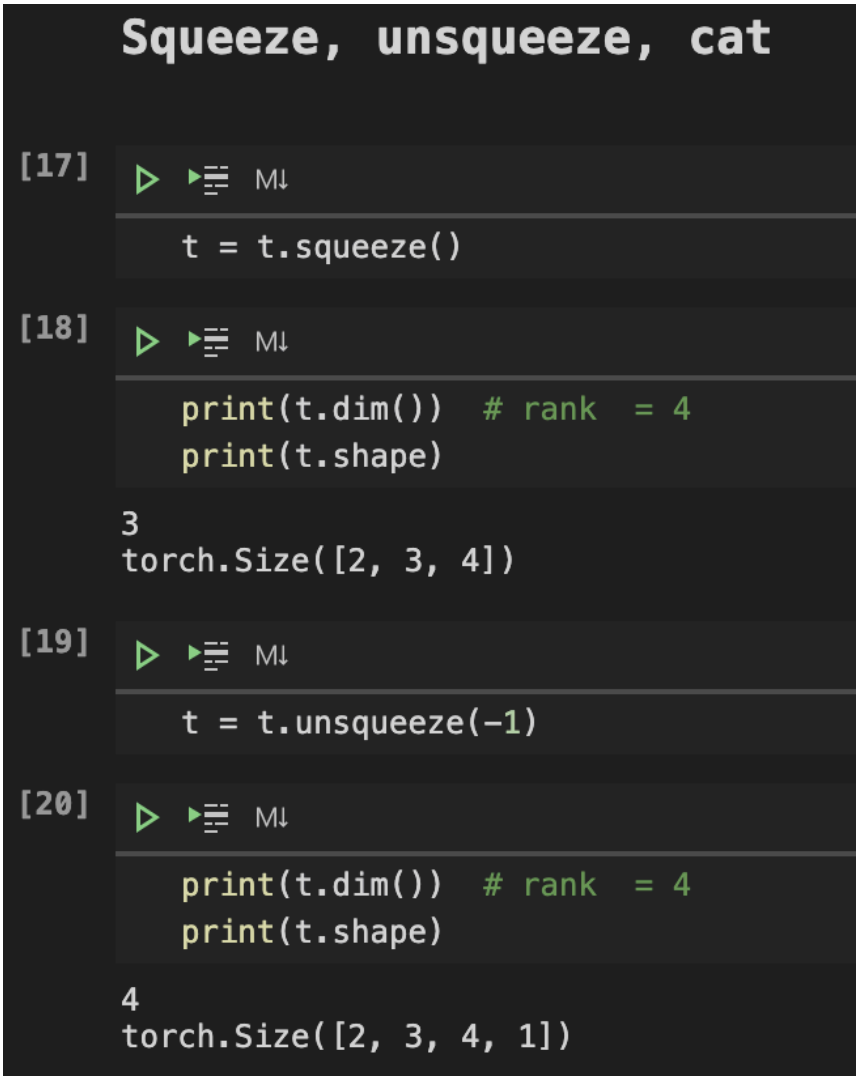
**print(t[:, :-1])** // 첫번째 차원을 전체 선택한 상황에서 두번째 차원에서는 맨 마지막에서 첫번째를 제외하고 다 가져온다.

## 5) Shape, Rank, Axis




// 2차원 배열 2개 = 차원

### 6) Squeeze, unsqueeze, cat




# **squeeze**는 차원이 1인 경우에는 해당 차원을 제거
# **unsqueeze**는 **squeeze**와 정반대입니다. 특정 위치에 1인 차원을 추가

```
    t = t.squeeze().unsqueeze(0)
    print(t)
```

```
tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.]],

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

[22]

```
    print(torch.cat([t, t], dim=0))
    print(torch.cat([t, t], dim=0).shape
```

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

### 7) broadcast

두 행렬 A, B의 행렬의 덧셈, 뺄셈을 할 때에는 두 행렬의 크기가 같아야하고,
곱셈을 할 때에는 A의 마지막 차원과 B의 첫번째 차원이 같아야한다.

# 파이토치에서는 자동으로 크기를 맞춰서 연산을 수행하는 **브로드캐스팅** 기능을 제공

## broadcast

[23]

```
# Same shape
m1 = torch.FloatTensor([[3, 3]])
m2 = torch.FloatTensor([[2, 2]])
print(m1 + m2)

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

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

```
tensor([[5., 5.]])
tensor([[4., 5.]])
tensor([[3., 3.],
        [3., 3.]])
```

[2, 1]
==> [[2, 1],
[2, 1]]

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

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

[1, 2]
==> [[1, 2],
[1, 2]]
[3]
[4]
==> [[3, 3],
[4, 4]]

**# 브로드캐스팅 사용시 단점**

브로드캐스팅은 자동으로 수행되므로 사용자는 나중에 원하는 결과가 나오지 않았더라도 어디서 문제가 발생했는지 찾기가 굉장히 어려울 수 있습니다.

## 8) Mean, Sum

**mean, sum**

[24]

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

print("=======mean======")
print(t.mean())
print(t.mean(dim=0))
print(t.mean(dim=1))
print(t.mean(dim=-1))

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

```
tensor([[1., 2.],
        [3., 4.]])
=======mean======
tensor(2.5000)
tensor([2., 3.])
tensor([1.5000, 3.5000])
tensor([1.5000, 3.5000])
=======sum======
tensor(10.)
tensor([4., 6.])
tensor([3., 7.])
tensor([3., 7.])
```

**# 인자로 dim을 준다면 해당 차원을 제거**한다는 의미
**# dim=0**은 **첫번째 차원, dim=1**은 두번째 차원, **dim=-1**은 마지막 차원
# 행렬에서 **첫번째 차원은 '행'을 의미, 두번째 차원은 열을 의미**

**print(t.mean(dim=0))** // 인자로 **dim=0을 주면 첫번째 차원**을 제거한다. 즉, 열만 남기겠다는 의미이다. 기존 행렬의 크기는 (2, 2)였지만 이를 수행하면 열의 차원만 보존되면서 (1, 2)가 된다.

```
# 실제 연산 과정
t.mean(dim=0)은 입력에서 첫번째 차원을 제거한다.

[[1., 2.],
 [3., 4.]]

1과 3의 평균을 구하고, 2와 4의 평균을 구한다.
결과 ==> [2., 3.]
```

**print(t.mean(dim=1))** // 인자로 dim=1을 주면 두번째 차원을 제거한다. 즉, 열의 차원을 제거하고, 행만 남기겠다는 의미이다.

**print(t.mean(dim=-1))** // dim=-1를 주는 경우는 마지막 차원을 제거한다. 즉, 열의 차원을 제거한다는 의미와 같습니다.

```
print(t.sum()) # 단순히 원소 전체의 덧셈을 수행
print(t.sum(dim=0)) # 행을 제거
print(t.sum(dim=1)) # 열을 제거
print(t.sum(dim=-1)) # 열을 제거
```

## 9) Named tensors

### color to gray

## Named tensors

### color to gray

[25]

```
img_t = torch.randn(3, 5, 5)
batch_t = torch.randn(2, 3, 5, 5)
weights = torch.tensor([0.2126, 0.7152, 0.0722]) # RGB weights

# naive color to gray
img_gray_naive = img_t.mean(-3)
batch_gray_naive = batch_t.mean(-3)
print(img_gray_naive.shape, batch_gray_naive.shape)
```

```
torch.Size([5, 5]) torch.Size([2, 5, 5])
```

[26]

```
#weighted color to gray
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)
img_weights = (img_t * unsqueezed_weights)
batch_weights = (batch_t * unsqueezed_weights)
img_gray_weighted = img_weights.sum(-3)
batch_gray_weighted = batch_weights.sum(-3)
print(img_gray_weighted.shape, batch_gray_weighted.shape)
```

```
torch.Size([5, 5]) torch.Size([2, 5, 5])
```

### Max, Argmax

## Max and Argmax

[27]

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
print(t.max()) # Returns one value: max
print(t.max(dim=0)) # Returns two values: max and argmax
print('Max: ', t.max(dim=0)[0])
print('Argmax: ', t.max(dim=0)[1])
```

```
tensor([[1., 2.],
        [3., 4.]])
tensor(4.)
torch.return_types.max(
values=tensor([3., 4.]),
indices=tensor([1, 1]))
Max:  tensor([3., 4.])
Argmax:  tensor([1, 1])
```

print(t.max(dim=0)) // 행의 차원을 제거한다는 의미이므로 (1, 2) 텐서를 만듭니다. 결과는 [3, 4]입니다.
그런데 [1, 1]이라는 값도 함께 리턴되었습니다. max에 dim 인자를 주면 argmax도 함께 리턴하는 특징 때문입니다. 첫번째 열에서 3의 인덱스는 1이었습니다. 두번째 열에서 4의 인덱스는 1이었습니다. 그러므로 [1, 1]이 리턴됩니다.

```
# [1, 1]가 무슨 의미인지 봅시다. 기존 행렬을 다시 상기해봅시다.
[[1, 2],
 [3, 4]]
첫번째 열에서 0번 인덱스는 1, 1번 인덱스는 3입니다.
두번째 열에서 0번 인덱스는 2, 1번 인덱스는 4입니다.
다시 말해 3과 4의 인덱스는 [1, 1]입니다.
```

# 만약 두 개를 함께 리턴받는 것이 아니라 max 또는 argmax만 리턴받고 싶다면 다음과 같이 리턴값에도 인덱스를 부여하면 됩니다. 0번 인덱스를 사용하면 max 값만 받아올 수 있고, 1번 인덱스를 사용하면 argmax 값만 받아올 수 있습니다.

```
print('Max: ', t.max(dim=0)[0])
print('Argmax: ', t.max(dim=0)[1])
```

```
Max:  tensor([3., 4.])
Argmax:  tensor([1, 1])
```

**View**

```
    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])
```

[29]

```
    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])
```

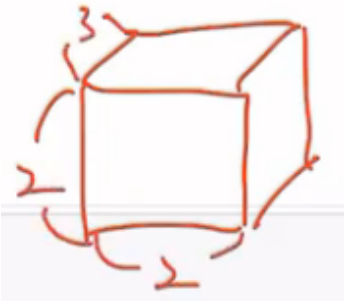
[69]

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

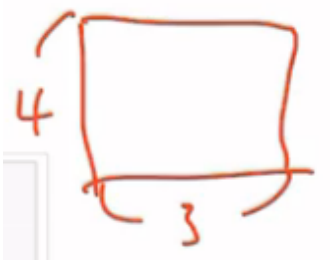
```
tensor([[[ 0.,  1.,  2.]],

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

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

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

파이토치 텐서의 **뷰(View)**는 넘파이에서의 리쉐이프(Reshape)와 같은 역할 (=텐서의 크기(Shape)를 변경해주는 역할)

**# view로 텐서의 크기를 변경하더라도 원소의 수는 유지되어야 한다**



**view([-1, 3])** // -1은 첫번째 차원은 사용자가 잘 모르겠으니 파이토치에 맡기겠다는 의미이고, 3은 두번째 차원의 길이는 3을 가지도록 하라는 의미입니다. 다시 말해 현재 3차원 텐서를 2차원 텐서로 변경하되 (?, 3)의 크기로 변경하라는 의미입니다. 결과적으로 (4, 3)의 크기를 가지는 텐서를 얻었습니다.




3차원 텐서에서 3차원 텐서로 차원은 유지하되, 크기(shape)를 바꾸는 작업
(2 × 2 × 3) = (? × 1 × 3) = 12를 만족해야 하므로 ?는 4가 됩니다.

**In-place Operation**

## In-place Operation

[73]

```
x = torch.FloatTensor([[1, 2], [3, 4]])
print(x.mul(2.))
print(x)
print(x.mul_(2.))
print(x)
print(x.zero_())
print(x)
print(x.add_(99))
print(x)
```

```
tensor([[2., 4.],
        [6., 8.]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[2., 4.],
        [6., 8.]])
tensor([[2., 4.],
        [6., 8.]])
tensor([[0., 0.],
        [0., 0.]])
tensor([[0., 0.],
        [0., 0.]])
tensor([[999., 999.],
        [999., 999.]])
tensor([[999., 999.],
        [999., 999.]])
```