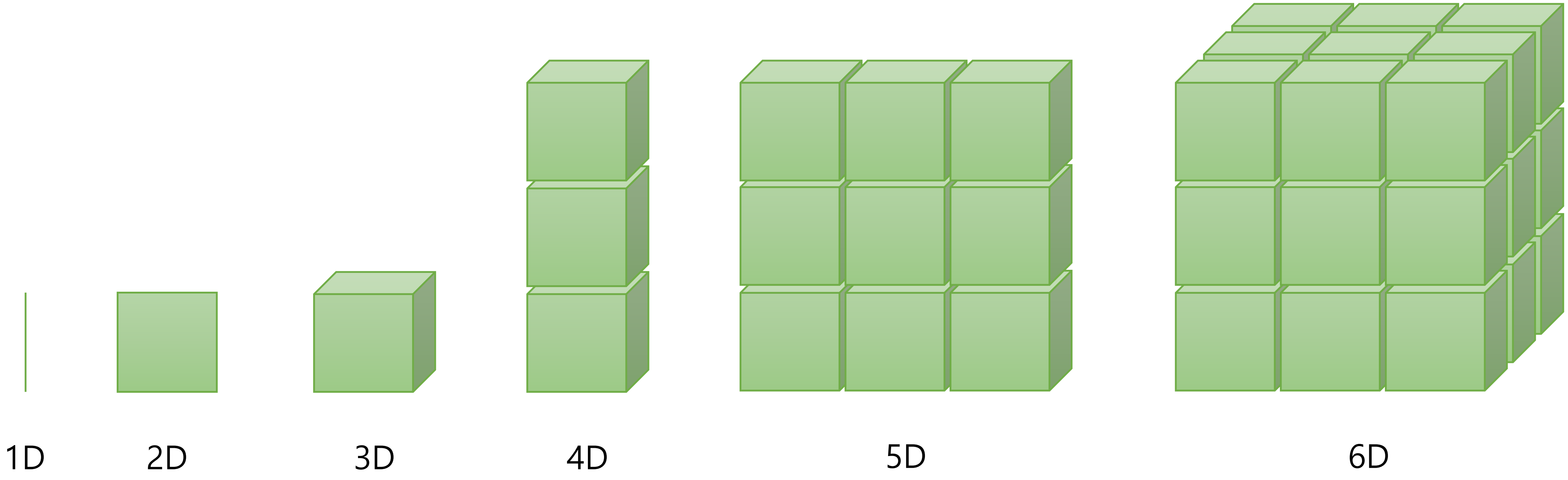
PyTorch
Basic Tensor Manipulation

Kim, Ki Hyun

Vector, Matrix and Tensor



Import

Imports

Run `pip install -r requirements.txt` in terminal to install all required Python packages.

In [1]:
```
import numpy as np
import torch
```

NumPy Review

1D Array with NumPy

In [2]:
```python
t = np.array([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

```
[0. 1. 2. 3. 4. 5. 6.]
```

In [3]:
```python
print('Rank  of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

```
Rank  of t:  1
Shape of t:  (7,)
```

In [4]:
```python
print('t[0] t[1] t[-1] = ', t[0], t[1], t[-1]) # Element
print('t[2:5] t[4:-1]   = ', t[2:5], t[4:-1])  # Slicing
print('t[:2] t[3:]      = ', t[:2], t[3:])     # Slicing
```

```
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.]
```

NumPy Review

2D Array with NumPy

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

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

In [6]:
```
print('Rank  of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

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

PyTorch Tensor

1D Array with PyTorch

In [7]:
```
t = torch.FloatTensor([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

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

In [8]:
```
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.])
```

PyTorch Tensor

2D Array with PyTorch

In [9]:
```
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.]])
```

In [10]:
```
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.]])
```

PyTorch Tensor

Shape, Rank, Axis

In [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]]]])
```

In [12]:
```
print(t.dim())  # rank  = 4
print(t.size()) # shape = (1, 2, 3, 4)
```

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

Multiplication vs Matrix Multiplication

Mul vs. Matmul

In [13]:
```
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))
```

```
-------------
Mul vs Matmul
-------------
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.]])
```

Broadcasting

Broadcasting

Carelessly using broadcasting can lead to code hard to debug.

In [14]:
```
# Same shape
m1 = torch.FloatTensor([[3, 3]])
m2 = torch.FloatTensor([[2, 2]])
print(m1 + m2)
```

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

In [15]:
```
# Vector + scalar
m1 = torch.FloatTensor([[1, 2]])
m2 = torch.FloatTensor([3]) # 3 -> [[3, 3]]
print(m1 + m2)
```

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

In [16]:
```
# 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.]])
```

Mean

Mean

In [17]:
```
t = torch.FloatTensor([1, 2])
print(t.mean())
```

```
tensor(1.5000)
```

In [18]:
```
# Can't use mean() on integers
t = torch.LongTensor([1, 2])
try:
    print(t.mean())
except Exception as exc:
    print(exc)
```

```
Can only calculate the mean of floating types. Got Long instead.
```

You can also use `t.mean` for higher rank tensors to get mean of all elements, or mean by particular dimension.

In [19]:
```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

In [20]:
```
print(t.mean())
print(t.mean(dim=0))
print(t.mean(dim=1))
print(t.mean(dim=-1))
```

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

Sum

Sum

In [21]:
```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

In [22]:
```
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

Max and Argmax

In [23]:
```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

The `max` operator returns one value if it is called without an argument.

In [24]:
```
print(t.max()) # Returns one value: max
```

```
tensor(4.)
```

The `max` operator returns 2 values when called with dimension specified. The first value is the maximum value, and the second value is the argmax: the index of the element with maximum value.

In [25]:
```
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([3., 4.]), tensor([1, 1]))
Max:  tensor([3., 4.])
Argmax:  tensor([1, 1])
```

In [26]:
```
print(t.max(dim=1))
print(t.max(dim=-1))
```

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

View (Reshape)

View

This is a function hard to master, but is very useful!

In [27]:
```
t = torch.FloatTensor([[[0, 1, 2],
                        [3, 4, 5]],
                       [[6, 7, 8],
                        [9, 10, 11]]])
print(t.shape)
```

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

In [28]:
```
print(t.view([-1, 3]))
print(t.view([-1, 3]).shape)
```

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

In [29]:
```
print(t.view([-1, 1, 3]))
print(t.view([-1, 1, 3]).shape)
```

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

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

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

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

Squeeze

Squeeze

In [30]:
```
t = torch.FloatTensor([[0], [1], [2]])
print(t)
print(t.shape)
```

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

In [31]:
```
print(t.squeeze())
print(t.squeeze().shape)
```

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

Unsqueeze

Unsqueeze

In [32]:
```python
t = torch.Tensor([0, 1, 2])
print(t.shape)
```

```
torch.Size([3])
```

In [33]:
```python
print(t.unsqueeze(0))
print(t.unsqueeze(0).shape)
```

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

In [34]:
```python
print(t.view(1, -1))
print(t.view(1, -1).shape)
```

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

In [35]:
```python
print(t.unsqueeze(1))
print(t.unsqueeze(1).shape)
```

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

In [36]:
```python
print(t.unsqueeze(-1))
print(t.unsqueeze(-1).shape)
```

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

Scatter

Scatter (for one-hot encoding)

Scatter is a very flexible function. We only discuss how to use it to get a one-hot encoding of indices.

In [36]:
```
lt = torch.LongTensor([[0], [1], [2], [0]])
pp.pprint(lt)
```

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

In [37]:
```
one_hot = torch.zeros(4, 3) # batch_size = 4, classes = 3
one_hot.scatter_(1, lt, 1)
pp.pprint(one_hot)
```

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

Type Casting

Casting

In [38]:
```
lt = torch.LongTensor([1, 2, 3, 4])
print(lt)
```
```
tensor([1, 2, 3, 4])
```

In [39]:
```
print(lt.float())
```
```
tensor([1., 2., 3., 4.])
```

In [40]:
```
bt = torch.ByteTensor([True, False, False, True])
print(bt)
```
```
tensor([1, 0, 0, 1], dtype=torch.uint8)
```

In [41]:
```
print(bt.long())
print(bt.float())
```
```
tensor([1, 0, 0, 1])
tensor([1., 0., 0., 1.])
```

Concatenate

Concatenation

In [43]:
```
x = torch.FloatTensor([[1, 2], [3, 4]])
y = torch.FloatTensor([[5, 6], [7, 8]])
```

In [44]:
```
print(torch.cat([x, y], dim=0))
print(torch.cat([x, y], dim=1))
```

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

Stacking

Stacking

In [45]:
```
x = torch.FloatTensor([1, 4])
y = torch.FloatTensor([2, 5])
z = torch.FloatTensor([3, 6])
```

In [46]:
```
print(torch.stack([x, y, z]))
print(torch.stack([x, y, z], dim=1))
```

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

In [47]:
```
print(torch.cat([x.unsqueeze(0), y.unsqueeze(0), z.unsqueeze(0)], dim=0))
```

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

Ones and Zeros

Ones and Zeros Like

In [44]:
```
x = torch.FloatTensor([[0, 1, 2], [2, 1, 0]])
print(x)
```

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

In [45]:
```
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

In-place Operation

In [50]:
```
x = torch.FloatTensor([[1, 2], [3, 4]])
```

In [51]:
```
print(x.mul(2))
print(x)
print(x.mul_(2.))
print(x)
```

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