PyTorch Basics

TEAMLAB director

최성철

PyTorch Operations

numpy + AutoGrad

numpy + AutoGrad

Tensor

- 다차원 Arrays 를 표현하는 PyTorch 클래스
- 사실상 numpy의 ndarray와 동일

  (그러므로 TensorFlow의 Tensor와도 동일)
- Tensor를 생성하는 함수도 거의 동일

Tensor

numpy - ndarray

```
import numpy as np
n_array = np.arange(10).reshape(2,5)
print(n_array)
print("ndim :", n_array.ndim, "shape :", n_array.shape)
```

pytorch - tensor

```
import torch
t_array = torch.FloatTensor(n_array)  print(t_array)
print("ndim :", t_array.ndim, "shape :", t_array.shape)
```

Array to Tensor

Tensor 생성은 list나 ndarray를 사용 가능

data to tensor

```
data = [[3, 5],[10, 5]]
x_data = torch.tensor(data)  x_data
```

ndarray to tensor

```
nd_array_ex = np.array(data)
tensor_array = torch.from_numpy(nd_array_ex)  tensor_array
```

Tensor data types

기본적으로 tensor가 가질 수 있는 data 타입은 numpy와 동일

Data types

Torch defines 10 tensor types with CPU and GPU variants which are as follows:

Data type	dtype	CPU tensor	GPU tensor
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.BFloat16Tensor`
32-bit complex	`torch.complex32`		
64-bit complex	`torch.complex64`		
128-bit complex	`torch.complex128` or `torch.cdouble`		
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`
Boolean	`torch.bool`	`torch.BoolTensor`	`torch.cuda.BoolTensor`
quantized 8-bit integer (unsigned)	`torch.quint8`	`torch.ByteTensor`	/
quantized 8-bit integer (signed)	`torch.qint8`	`torch.CharTensor`	/
quantized 32-bit integer (signed)	`torch.qfint32`	`torch.IntTensor`	/
quantized 4-bit integer (unsigned) 3	`torch.quint4x2`	`torch.ByteTensor`	/

numpy like operations

기본적으로 PyTorch의 대부분의 사용법이 그대로 적용됨

```
data = [[3, 5, 20],[10, 5, 50], [1, 5, 10]]
x_data = torch.tensor(data)

x_data[1:]
# tensor([[10, 5, 50],
#         [ 1, 5, 10]])

x_data[:2, 1:]
# tensor([[ 5, 20],
#         [ 5, 50]])

x_data.flatten()
# tensor([ 3, 5, 20, 10, 5, 50, 1, 5, 10])
```

```
torch.ones_like(x_data)

# tensor([[1, 1, 1],
#         [1, 1, 1],
#         [1, 1, 1]])

x_data.numpy()
# array([[3, 5, 20],
#        [10, 5, 50],
#        [1, 5, 10]], dtype=int64

x_data.shape
# torch.Size([3, 3])

x_data.dtype
# torch.int64
```

numpy like operations

pytorch의 tensor는 GPU에 올려서 사용가능

```
x_data.device
# device(type='cpu')

if torch.cuda.is_available():
x_data_cuda = x_data.to('cuda')
x_data_cuda.device
# device(type='cuda', index=0)
```

Tensor handling

view, squeeze, unsqueeze 등으로 tensor 조정가능

- view: reshape과 동일하게 tensor의 shape을 변환
- squeeze: 차원의 개수가 1인 차원을 삭제 (압축)
- unsqueeze: 차원의 개수가 1인 차원을 추가

Tensor handling

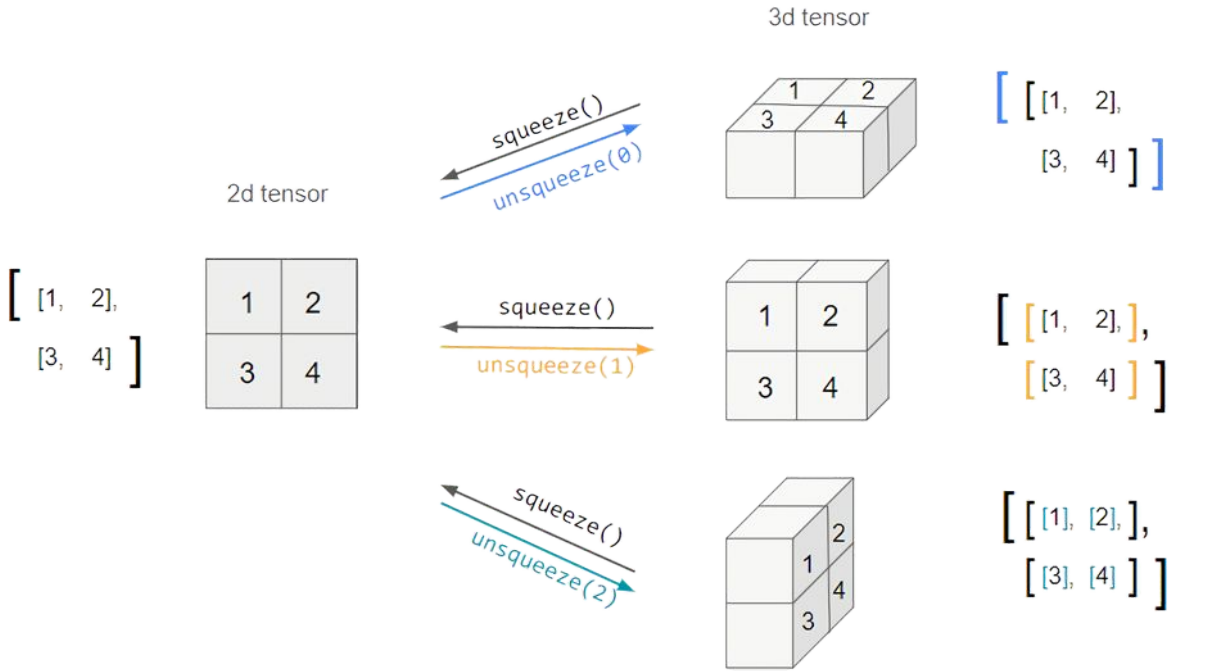

```
tensor_ex = torch.rand(size=(2, 3, 2))
tensor_ex
 # tensor([[[0.7466, 0.5440],
 #          [0.7145, 0.2119],
 #          [0.8279, 0.0697]],

 #         [[0.8323, 0.2671],
 #          [0.2484, 0.8983],
 #          [0.3228, 0.2254]]])


 tensor_ex.view([-1, 6])
 # tensor([[0.7466, 0.5440, 0.7145, 0.2119, 0.8279, 0.0697],
 #         [0.8323, 0.2671, 0.2484, 0.8983, 0.3228, 0.2254]])

 tensor_ex.reshape([-1, 6])
 # tensor([[0.7466, 0.5440, 0.7145, 0.2119, 0.8279, 0.0697],
 #         [0.8323, 0.2671, 0.2484, 0.8983, 0.3228, 0.2254]])
```

Tensor handling

view와 reshape은 contiguity 보장의 차이

```
a = torch.zeros(3, 2)
b = a.view(2, 3)
a.fill_(1)

a = torch.zeros(3, 2)
b = a.t().reshape(6)
a.fill_(1)
```

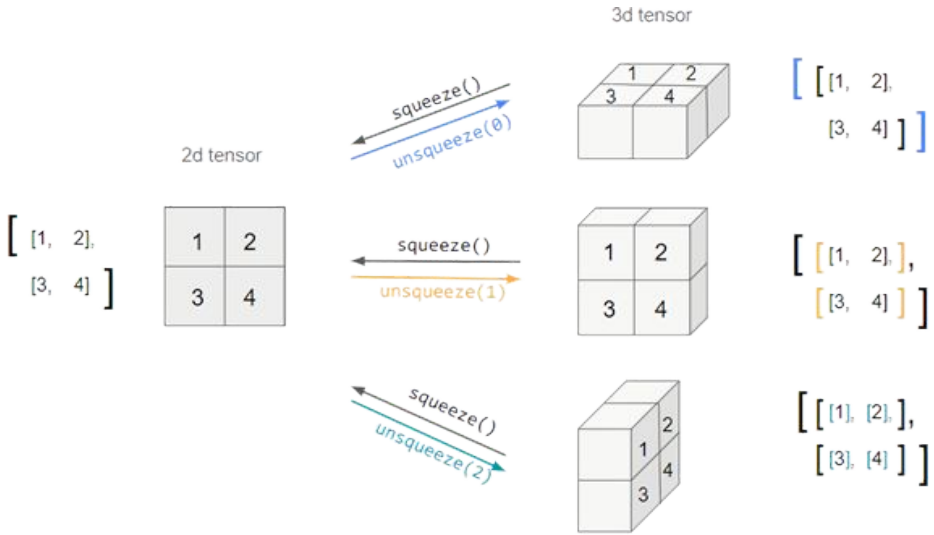
Tensor handling



https://bit.ly/3CgkVWK

Tensor handling


```
tensor_ex = torch.rand(size=(2, 1, 2))
tensor_ex.squeeze()
# tensor([[0.8510, 0.8263],
#         [0.7602, 0.1309]])

tensor_ex = torch.rand(size=(2, 2))
tensor_ex.unsqueeze(0).shape

# torch.Size([1, 2, 2])


tensor_ex.unsqueeze(1).shape

# torch.Size([2, 1, 2])


tensor_ex.unsqueeze(2).shape

# torch.Size([2, 2, 1])
```



https://bit.ly/3CgkVWK

Tensor operations

기본적인 tensor의 operations는 numpy와 동일

```
n1 = np.arange(10).reshape(2,5)
t1 = torch.FloatTensor(n1)
```

```
t1 - t1
# tensor([[10., 11., 12., 13., 14.],
#         [15., 16., 17., 18., 19.]])
```

```
t1 + t1
# tensor([[ 0.,  2.,  4.,  6.,  8.],
#         [10., 12., 14., 16., 18.]])
```

```
t1 + 10
# tensor([[10., 11., 12., 13., 14.],
#         [15., 16., 17., 18., 19.]])
```

Tensor operations

행렬곱셈 연산은 함수는 dot이 아닌 mm 사용

```
n2 = np.arange(10).reshape(5,2)
t2 = torch.FloatTensor(n2)

t1.mm(t2)
# tensor([[ 60., 70.],
#         [160., 195.]])

t1.dot(t2)
# RuntimeError

t1.matmul(t2)
# tensor([[ 60., 70.],
#         [160., 195.]])
```

```
a = torch.rand(10)
b = torch.rand(10)
a.dot(b)


a = torch.rand(10)
b = torch.rand(10)
a.mm(b)
```

Tensor operations

mm과 matmul은 broadcasting 지원 차이

```
a = torch.rand(5, 2, 3)
b = torch.rand(5)
a.mm(b)
```

```
a = torch.rand(5, 2, 3)
b = torch.rand(3)
a.matmul(b)
```

```
a[0].mm(torch.unsqueeze(b,1))
a[1].mm(torch.unsqueeze(b,1))
a[2].mm(torch.unsqueeze(b,1))
a[3].mm(torch.unsqueeze(b,1))
a[4].mm(torch.unsqueeze(b,1))
```

Tensor operations for ML/DL formula

nn.functional 모듈을 통해 다양한 수식 변환을 지원함

```
import torch
import torch.nn.functional as F

tensor = torch.FloatTensor([0.5, 0.7, 0.1])
h_tensor = F.softmax(tensor, dim=0)
h_tensor
# tensor([0.3458, 0.4224, 0.2318])

y = torch.randint(5, (10,5))
y_label = y.argmax(dim=1)

torch.nn.functional.one_hot(y_label)
```

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

AutoGrad

PyTorch의 핵심은 자동 미분의 지원 → backward 함수 사용

```
w = torch.tensor(2.0, requires_grad=True)
y = w**2
z = 10*y + 25
z.backward()
w.grad
```

$$y = w^2$$

$$z = 10 * y + 25$$

$$z = 10 * w^2 + 25$$

AutoGrad

PyTorch의 핵심은 자동 미분의 지원 → backward 함수 사용

$$Q = 3a^3 - b^2$$

$$\frac{\partial Q}{\partial a} = 9a^2$$

$$\frac{\partial Q}{\partial b} = -2b$$

```
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
Q = 3*a**3 – b**2
external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)
```

```
a.grad
# a.grad
b.grad
# tensor([-12., -8.])
```

End of Document
Thank You.