Pytorch tutorial

What is pytorch?

- Numpy를 대체하며 GPU연산을 지원하는 tensor 구조 지원
  CUDA(NDVIA에서 gpu를 통한 연산을 위해 만든 API)와 cuDNN(CUDA를 활용하여 딥러닝 gpu연산을 지원하는 API)을 활용하여 연산 가속
  일반적으로 CUDA와 cuDNN을 활용하는 연산은 CPU연산의 15배정도 가속을 가져옴.
- backward() 함수를 통한 그래프 미분 연산 지원

Pytorch vs Tensorflow

둘 모두 GPU를 활용하는 framework

- Tensorflow
  `Define and Run` 방식
  연산 그래프를 미리 만들어두고, 실제 연산시 값을 전달하여 결과를 얻음(정적)
  직관적이지 않고 그래프를 정의하는 부분과 실행하는 부분이 분리되어 코드가 길어짐
  최적화에서 장점
- Pytorch
  `Define by Run` 방식
  연산 그래프를 미리 만들어두지 않고 값이 할당되어 전달되는 과정에서 그래프가 작성(동적)
  직관적이고 간단한 코드
  (Pytorch측의 주장으로는)Tensorflow보다 평균 2.5배정도 빠른 속도(적어도 밀리지는 않음)

Tensor

Tensor: Numpy의 ndarray와 유사한 matrix자료구조. GPU연산 가속이 가능

```
1 import torch
2 import numpy as np
```

Tensor 생성

```
1 not_initialized = torch.empty(3, 4)
2 not_initialized
```

```
tensor([[-9.7406e-23,  3.0742e-41,  3.3631e-44,  0.0000e+00],
        [        nan,  3.0742e-41,  1.1578e+27,  1.1362e+30],
        [ 7.1547e+22,  4.5828e+30,  1.2121e+04,  7.1846e+22]])
```

```
random_initialized = torch.rand(3, 4)
random_initialized
```

```
tensor([[0.2220, 0.6558, 0.8770, 0.1879],
        [0.5791, 0.5067, 0.0954, 0.1268],
        [0.5590, 0.6974, 0.3883, 0.5308]])
```

```
zero_initialized = torch.zeros(3, 4)
zero_initialized
```

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

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

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

```
arange_Initiaized = torch.arange(10)
arange_Initiaized
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Tensor와 numpy간의 변환

```
x = torch.ones(10)
x, type(x)
```

```
(tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), torch.Tensor)
```

```
y = x.numpy()
y, type(y)
```

```
(array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32), numpy.ndarray)
```

```
z = torch.from_numpy(y)
z
```

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

주의사항 - tensor와 ndarray는 메모리공간을 공유함

[ ] ↳ 숨겨진 셀 3개

Tensor 특성

[ ] ↳ 숨겨진 셀 2개

Tensor reshape vs view

[ ] ↳ 숨겨진 셀 4개

Tensor의 배열 연산

[ ] ↳ 숨겨진 셀 20개

Aggregation

[ ] ↳ 숨겨진 셀 9개

Broadcasting

[ ] ↳ 숨겨진 셀 2개

AUTOGRAD: 자동미분

pytorch 신경망의 중심. backward() 를 가능케 하는 패키지
Tensor 의 모든 연산에 대해 자동 미분을 제공

pytorch.Tensor class에는 .required_grad 속성이 존재
.required_grad 를 True로 세팅하면 해당 tensor를 기반으로 이루어신 모든 연산을 추적(track)하기 시작함
모든 연산이 완료된 후, backward() 를 호출하면 모든 gradient가 자동으로 계산(Tensor.grad 속성에 gradient가 누적됨)

Autograd의 다른 중요 class는 Function class
Tensor 와 Function class는 연결되어 있으며, 모든 연산을 encode하여 acyclic graph를 생성
각각의 tensor는 .grad_fn 속성을 가지며 이는 Tensor 를 생성한 Function 을 참조함

```
1 x = torch.ones(5, 2, requires_grad=True)
2 # x = torch.arange(10, dtype=torch.float32).reshape(5, 2)
3 # x.requires_grad = True
4 x
```

```
4 x
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]], requires_grad=True)
```

```
1 y = x + 10
```

```
1 y
```

```
tensor([[11., 11.],
        [11., 11.],
        [11., 11.],
        [11., 11.],
        [11., 11.]], grad_fn=<AddBackward0>)
```

y는 연산의 결과로 생성된 tensor이므로 grad_fn을 가짐

```
1 y.grad_fn
```

```
<AddBackward0 at 0x7f79f3e89f50>
```

```
1 z = y * y * 10
```

```
1 out = z.mean()
```

```
1 z, out
```

```
(tensor([[1210., 1210.],
         [1210., 1210.],
         [1210., 1210.],
         [1210., 1210.],
         [1210., 1210.]], grad_fn=<MulBackward0>),
 tensor(1210., grad_fn=<MeanBackward0>))
```

변화도(Gradient) 계산)

```
1 out.backward(retain_graph=True)
```

```
1 print(x.grad)
```

```
tensor([[22., 22.],
        [22., 22.],
        [22., 22.],
        [22., 22.],
        [22., 22.]])
```

▶ Jacobian matrix 활용

[ ] ↳ 숨겨진 셀 4개

▼ DataLoader

torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0,...)

데이터를 학습모델에 통째로 넣지 않고 batch 단위로 나누어 학습시킬 때 활용.
모든 데이터를 나누고 나눈 데이터를 차례로 넣어주는 과정을 대신해줌

- 용어
  - epoch: 모든 training data 혹은 모든 training data를 학습시킨 상태
  - batch: batch_size만큼의 training data만 학습시키기 위한 소분 데이터

```
x = [[73, 80, 75],
   [93, 88, 93],
   [89, 91, 90],
   [96, 98, 100],
   [73, 66, 70]]
x = torch.FloatTensor(x)
x
```

```
tensor([[ 73.,  80.,  75.],
        [ 93.,  88.,  93.],
        [ 89.,  91.,  90.],
        [ 96.,  98., 100.],
        [ 73.,  66.,  70.]])
```

```
data_loader = torch.utils.data.DataLoader(x,
                            batch_size=2,
                            shuffle=True,
                            num_workers=0)
```

```
epochs = 2
for epoch in range(epochs):
    for step, batch in enumerate(data_loader):
        print(step, batch)
```

```
0 tensor([[73., 80., 75.],
        [89., 91., 90.]])
1 tensor([[ 73.,  66.,  70.],
        [ 96.,  98., 100.]])
2 tensor([[93., 88., 93.]])
0 tensor([[ 96.,  98., 100.],
        [ 93.,  88.,  93.]])
1 tensor([[73., 80., 75.],
```

```
        [73., 66., 70.]])
2 tensor([[89., 91., 90.]])
```

CUDA Tensor

GPU로 이동

```
1 if torch.cuda.is_available():
2     x = x.to(torch.device('cuda'))
3     print(type(x))
4     print(x.device)
```

```
<class 'torch.Tensor'>
cuda:0
```

생성과 함께 gpu에 할당

```
1 if torch.cuda.is_available():
2     k = torch.full((2, 3), 1, device=torch.device('cuda'))
3     print(type(k))
4     print(k.device)
```

```
<class 'torch.Tensor'>
cuda:0
```

CPU로 이동

```
1 if torch.cuda.is_available():
2     x = x.to(torch.device('cpu'))
3     print(type(x))
4     print(x.device)
```

```
<class 'torch.Tensor'>
cpu
```

```
1
```

✓ 0초 오전 11:33에 완료됨