Pytorch

발표 주제

이 동헌

2018.03.02.

Pytorch

- Python을 기반으로 하는 Scientific Computing패키지
  - GPU를 활용하기 위한 Numpy의 대체제로 사용
  - 유연성과 스피드를 제공하는 딥러닝 연구 플랫폼

- Pytorch에서 연산을 위한 변수는 모두 Tensor로 선언

Tensor

- 텐서(Tensor)
  - Numpy의 ndarrays와 유사한것으로 속도를 빠르게 하기 위하여 GPU에서 사용할 수 있는 것
  - 초기화 되지 않은 5x3행렬 생성

```
x = torch.empty(5, 3)
print(x)
```

```
tensor(1.00000e-04 *
       [[-0.0000,  0.0000,  1.5135],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000]])
```

  - 랜덤으로 초기화 된 행렬 생성

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[ 0.6291,  0.2581,  0.6414],
        [ 0.9739,  0.8243,  0.2276],
        [ 0.4184,  0.1815,  0.5131],
        [ 0.5533,  0.5440,  0.0718],
        [ 0.2908,  0.1850,  0.5297]])
```

Tensor

- 0으로 채워지고 long 데이터 타입을 가지는 행렬 생성
- ※dtype = 데이터타임

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

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

- 입력데이터로부터 젝접 텐서 생성

```
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([ 5.5000,  3.0000])
```

- 텐서의 크기 확인

```
print(x.size())
```

```
torch.Size([5, 3])
```

Tensor

- 이미 존재하는 텐서를 기반으로 새로운 텐서 생성

```
x = x.new_ones(5, 3, dtype=torch.double)      # new_* methods take in sizes
print(x)

x = torch.randn_like(x, dtype=torch.float)    # override dtype!
print(x)                                      # result has the same size
```

```
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]], dtype=torch.float64)
tensor([[-0.2183,  0.4477, -0.4053],
        [ 1.7353, -0.0048,  1.2177],
        [-1.1111,  1.0878,  0.9722],
        [-0.7771, -0.2174,  0.0412],
        [-2.1750,  1.3609, -0.3322]])
```

Tensor Operations

- 더하기-1

```
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

- 더하기-2

```
print(torch.add(x, y))
```

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

Tensor Operations

- 더하기-3 : 파라미터로 결과 텐서 이용

```
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

x와 y텐서의 합 결과가 result텐서에 저장

- 더하기-4 : 제자리

```
y.add_(x)
print(y)
```

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

y텐서에 x텐서와의 합 결과가 덮어쓰기 됨

Tensor Operations

- 사이즈 변경

```
x = torch.randn(4,4)
y = x.view(16)
z = x.view(-1, 8)
print(x)
print(y)
print(z)
```

```
tensor([[ 0.0069,  0.9910,  1.0710, -0.2402],
        [-1.1811,  1.0306,  0.1867, -0.0986],
        [-0.0474, -0.3365,  0.2849, -1.0639],
        [-1.7261, -0.2660,  0.8515,  0.6763]])
tensor([ 0.0069,  0.9910,  1.0710, -0.2402, -1.1811,  1.0306,  0.1867,
        -0.0986, -0.0474, -0.3365,  0.2849, -1.0639, -1.7261, -0.2660,
         0.8515,  0.6763])
tensor([[ 0.0069,  0.9910,  1.0710, -0.2402, -1.1811,  1.0306,  0.1867,
         -0.0986],
        [-0.0474, -0.3365,  0.2849, -1.0639, -1.7261, -0.2660,  0.8515,
          0.6763]])
```

x.size = 4x4, y.size = 1x16, z.size = 2x8

Autograd mechanics

- Autograd는 backprop을 위한 미분 값을 자동으로 계산
- 이를 위하여 사용하는 변수 => Variable
- backprop 연산을 수행할 Tensor를 Variable로 감싸야한다.
- 모든 Variable은 2가지 flags를 가지며, 둘다 하위 그래프를 제외한 gradient 계산을 통해 효율성을 증가시킨다.
  - requires_grad : 해당 Tensor가 gradient를 필요로 하는지
  - volatile
- CNN과 RNN의 weight의 경우와는 달리 Vairable은 requires_grad가 false로 기본설정이 되어있다.

Autograd mechanics

- Variable의 형태

  data : Tensor형태의 데이터가 담기는 공간

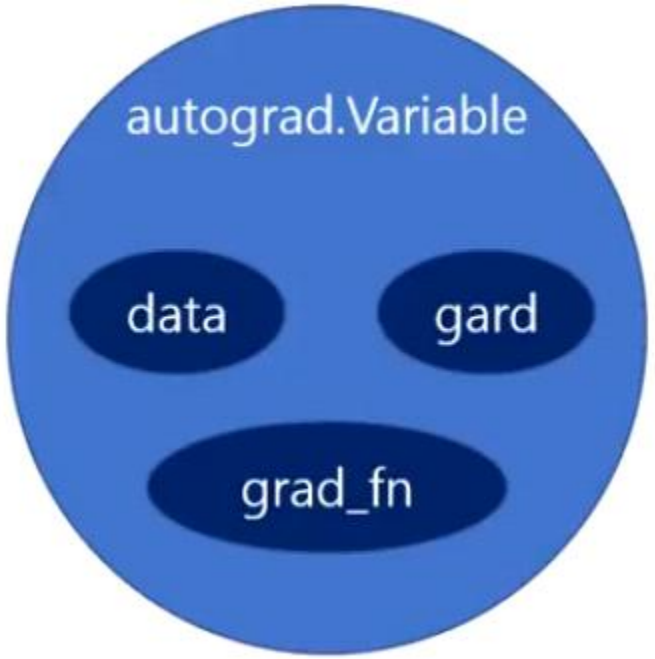
  grad : Data가 거쳐온 Layer에 대한 미분값이 축적되는 공간

  grad_fn : 미분 값을 계산한 함수에 대한 정보

  => 미분 값을 계산할 때 어떠한 함수를 사용했는지

```
a = torch.ones(2,2)
a = Variable(a, requires_grad=True)
print(a)
print("-------a.data-------")
print(a.data)
print("-------a.grad-------")
print(a.grad)
print("-------a.grad_fn----")
print(a.grad_fn)
```

```
tensor([[ 1.,  1.],
        [ 1.,  1.]])
-------a.data-------
tensor([[ 1.,  1.],
        [ 1.,  1.]])
-------a.grad-------
None
-------a.grad_fn----
None
```



Autograd mechanics

- Tensor 생성

```
a = torch.ones(2,2)
a = Variable(a, requires_grad=True)
print(a)
```

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

- a를 이용하여 연산 수행

```
b=a+2
c=b**2
out=c.sum()
print(out)
```

```
tensor(36.)
```

a -> b -> c -> out

- a를 업데이트하기 위해서는 ∂out/∂a 연산을 해주어야 한다.
- 이 값을 계산하여 저장하는 장소가 a.grad인데 현재 a의 grad는 none으로 비어있는 상태이다.

Autograd mechanics

- 앞서 a.grad를 채워주기 위하여 실행하는 함수가 backword()라는 함수이다.

```
out.backward()

print("---after a.grad----")
print(a.grad)
```

```
---after a.data----
tensor([[ 1.,  1.],
        [ 1.,  1.]])
---after a.grad----
tensor([[ 6.,  6.],
        [ 6.,  6.]])
---after a.grad_fn----
None
```

그림과 같이 a.grad가 채워진걸 볼 수 있다.

하지만 a가 직접적으로 연산을 수행하지 않았기 때문에

a.grad_fn은 none상태를 유지한다.

Autograd mechanics

```
print("---after b.data----")
print(b.data)
print("---after b.grad----")
print(b.grad)
print("---after b.grad_fn----")
print(b.grad_fn)
```

```
---after b.data----
tensor([[ 3.,  3.],
        [ 3.,  3.]])
---after b.grad----
None
---after b.grad_fn----
<AddBackward0 object at 0x0000028DFC285048>
```

- 반면에 b(a+2)의 grad_fn을 보게 되면 AddBackward가 채워져 있는데 이는 a에 대하여 더하기 여산에 대한 Backward를 했다는 의미이다.

Autograd mechanics

```
x=torch.ones(3)
x=Variable(x, requires_grad=True)
y=(x**2)
z=y*3
print(z)
grad=torch.Tensor([0.1,1,10])
z.backward(grad)
```

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

```
print("----x.data----")
print(x.data)
print("----x.grad----")
print(x.grad)
print("----x.grad_fn----")
print(x.grad_fn)
```

```
----x.data----
tensor([ 1.,  1.,  1.])
----x.grad----
tensor([  0.6000,   6.0000,  60.0000])
----x.grad_fn----
None
```

- y=x2, z=3*y 따라서 z = 3*x^2이다.
- $\partial z/\partial x$ = 3*2*x인데 x=1이기 때문에 6이 된다
- 그러나 x.grad를 보게되면 0.6, 6.0, 60이 들어있다.
- 이는 grad=torch.Tensor([0.1, 1, 10])때문인데 Tensor를 backward의 인자로 주게되면 grad의 결과에 Tensor가 곱해진 결과를 주게된다.

참고

- http://bob3rdnewbie.tistory.com/314?category=780658

- https://www.youtube.com/watch?v=E0R9Xf_GyUc