en-ch4.3_Automatic_Differentiation

2019년 10월 8일

1 Chapter 4 - THE PRELIMINARIES: A CRASHCOURSE

1.1 4.3 Automatic Differentiation

autograd package 는 자동으로 미분값을 계산해주고 backproaation 을 쉽게 할수 있도록 도와줌.

```
In [1]: from mxnet import autograd, nd
```

4.3.1 A simple Example

MXnet 의 autograd 의 기본적인 사용법을 설명

- 간단한 예로서, 벡터 x 로 $y = 2x^\top x$ 미분하려고 함. 벡터 x 를 초기화하고 할당함

```
In [2]: x = nd.arange(4)
        x
```

```
Out[2]:
        [0. 1. 2. 3.]
        <NDArray 4 @cpu(0)>
```

- ndarray 의 attach_grad method 를 호출하여 기울기를 저장할 수 있다.

```
In [3]: # 해당 함수 불릴 때, x.grad 값 만들어짐. 호출 안하면 x.grad 값은 None
        x.attach_grad()
```

- 이제 y를 계산하고 MXnet은 바로 연산 그래프를 생성할 것이다. 마치 MXnet 이 레코딩 장치를 켜고 생성되는 변수들을 바로 캡쳐한것과 같다. 계산 그래프를 만드는 것은 적지않은 계산양이 필요하다는 것을 주목해라. 그렇기에 MXnet with autograd.record(): block 안에서만 그래프를 만들 것이다.

```
In [4]: with autograd.record():
            y = 2 * nd.dot(x, x)
        y

Out[4]:
        [28.]
        <NDArray 1 @cpu(0)>
```

- x 는 길이 4 인 vector 이고 nd.dot 이 inner product 룰 수행할 것이다. 따라서 y 는 scalar 값이 나온다 다음으로 우리는 backward function 을 호출하므로 모든 input 의 기울기를 자동적으로 찾을 수 있다.

```
In [5]: y.backward()
```

- x 에 대한 함수 $y = 2x^\top x$ 의 기울기는 4x 이여야 한다. mxnet 에 의해 생성된 기울기가 맞는 값인지 확인해보자.

```
In [6]: print(x)
        print(x.grad)
        print(x.grad - 4 * x)


[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>

[ 0.  4.  8. 12.]
<NDArray 4 @cpu(0)>

[0. 0. 0. 0.]
<NDArray 4 @cpu(0)>
```

- 만일 x 가 다른 부분에서 기울기 계산이 수행되었다면 이전의 x.grad 값은 덮어쓰여진다.

```
In [7]: with autograd.record():
            y = x.norm()
        y.backward()
        x.grad
```

```
Out[7]:
        [0.         0.26726124 0.5345225  0.80178374]
        <NDArray 4 @cpu(0)>
```

4.3.2 Backward for Non-scalar Variable

- y 가 scalar 가 아닐 경우 기울기는 고차원의 tensor 이고 계산이 복잡할 수 있다. 다행히도 머신러닝과 딥러닝 모두에서, 종종 scalar 값이 되는 loss function 의 기울기 만을 계산한다. y 가 scalar 가 아닐 때, mxnet 은 기본적으로 새로운 변수를 얻기 위해 y 안에 element 를 합한 다음, 현재의 dydx 에서 x 에 대한 분석적 기울기를 찾을 것이다.

```
In [8]: with autograd.record(): # y is a vector
            y = x * x
        print('y vector : ', y)
        y.backward()
        print('x.grad : ', x.grad)

        u = x.copy()
        u.attach_grad()

        with autograd.record(): # v is scalar
            v = (u * u).sum()
        print('v scalar : ', v)
        v.backward()
        print('u.grad : ', u.grad)

        x.grad - u.grad

y vector :
[0. 1. 4. 9.]
<NDArray 4 @cpu(0)>
x.grad :
[0. 2. 4. 6.]
<NDArray 4 @cpu(0)>
v scalar :
[14.]
<NDArray 1 @cpu(0)>
```

```
u.grad :
[0. 2. 4. 6.]
<NDArray 4 @cpu(0)>
```

Out[8]:

```
[0. 0. 0. 0.]
<NDArray 4 @cpu(0)>
```

4.3.3 Detach Computations

- 계산 그래프의 일부를 계산 밖으로 옮길 수 있다. y = f(x), z = g(y) 를 가정했을 때, u = y.detach() 를 호출할 때, 이 동작은 새로운 변수를 리턴하고 u 가 어떻게 계산되었는지를 잊게함. u 를 constant 같이 취급함. 아래의 backward 계산 예제는 x 에 대한 미분계산을 x^3/x 대신에 ux/x 로 실행함. $ux/x = u$

In [9]:
```
# 상단에서 x.attach_grad() 호출되어 있는 상태.
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
print('x : ', x)
print('u : ', u)
print('z : ', z)

z.backward()

print('x.grad : ', x.grad)

x.grad - u
```

```
x :
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
u :
[0. 1. 4. 9.]
<NDArray 4 @cpu(0)>
z :
```

```
[ 0.  1.  8. 27.]
<NDArray 4 @cpu(0)>
x.grad :
[0. 1. 4. 9.]
<NDArray 4 @cpu(0)>
```

Out[9]:

```
[0. 0. 0. 0.]
<NDArray 4 @cpu(0)>
```

- y 의 계산은 여전히 기록되고 있으므로 우리는 y.backward 를 호출하여 $y/x = 2x$ 를 얻을 수 있다.

```
In [10]: y.backward()
         print('x : ', x)
         print('y : ', y)
         print('x.grad : ', x.grad)


         x.grad - 2*x
```

```
x :
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
y :
[0. 1. 4. 9.]
<NDArray 4 @cpu(0)>
x.grad :
[0. 2. 4. 6.]
<NDArray 4 @cpu(0)>
```

Out[10]:

```
[0. 0. 0. 0.]
<NDArray 4 @cpu(0)>
```

- 결국 detach 함수를 통해 autograd.record scope 안에서 여러 단계의 계산이 있을 경우, 일부 미분 계산 선택적으로 하는 것이 가능.

4.3.4 Attach Gradients to Internal Variables

- attach grad 를 실행하면 내재적으로 detach 가 실행됨. 아래 예제에서 u 를 다른 변수 기반으로 계산한다면 backward 계산에서 y 는 아예 사용되지 않게 됨.

In [11]:
```
# x.attach_grad() 는 상단에서 호출되어 있음.

y = nd.ones(4) * 2
print('x :', x)
print('y :', y)

y.attach_grad()

# z = x* y + x  -> dz/dx

# z = u + x -> dz/dx
with autograd.record():
    u = x * y

    # implicitly run u = u.detach(), u = x * y 가 z.backward 에는 포함되지 않는다.
    u.attach_grad()

    z = u + x

z.backward() # 실질적으로 'u + x' 에 대한 미분값 만을 계산함.
print('u :', u)
print('z :', z)

print('\nx.grad :', x.grad)
print('\nu.grad :', u.grad)
print('\ny.grad :', y.grad)
```

```
x :
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
y :
[2. 2. 2. 2.]
```

```
<NDArray 4 @cpu(0)>
u :
[0. 2. 4. 6.]
<NDArray 4 @cpu(0)>
z :
[0. 3. 6. 9.]
<NDArray 4 @cpu(0)>

x.grad :
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>

u.grad :
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>

y.grad :
[0. 0. 0. 0.]
<NDArray 4 @cpu(0)>
```

4.3.5 Head gradients *Detach* 를 통해 여러 개의 계산식에 대한 각각의 미분값을 구할 수 있게 됨.

- u = f(x), z = g(u) 라고 가정했을 때 체인룰에 의해 다음과 같이 표현된다.

$$\frac{dz}{dx} = \frac{dz}{du}\frac{du}{dx}$$

- head gradient $\frac{dz}{du}$ 를 구하기 위해, 우리는 먼저 u.detach 를 실행한 후 z.backward 를 실행한다.

```
In [12]: y = nd.ones(4) * 2

         print('x :', x)
         print('y :', y)

         with autograd.record():
             u = x * y
             v = u.detach() # u still keeps the computation graph
```

```
            v.attach_grad()
            z = v + x

        print('\nu = x * y :', u)
        print('\nz = u + x :', z)

        print('\n\nrun `z.backward`')
        z.backward()

        print('\n\nx.grad : ',x.grad)
        print('\nv.grad = dz/du = d(u + x)/du = ',v.grad)
```

```
x :
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
y :
[2. 2. 2. 2.]
<NDArray 4 @cpu(0)>

u = x * y :
[0. 2. 4. 6.]
<NDArray 4 @cpu(0)>

z = u + x :
[0. 3. 6. 9.]
<NDArray 4 @cpu(0)>


run `z.backward`


x.grad :
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>

v.grad = dz/du = d(u + x)/du =
```

```
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>
```

```
In [13]: u.backward(v.grad) # v.grad 집어넣으면 결국 dz/du * du/dx = dz/dx 의 결과가 나옴.

         print('dz/du = ', v.grad)
         print('\n du/dx = d(x * y)/dx = y = ', y)
         print('\n ')
         print('\n x.grad --> dz/dx = ', x.grad)
```

```
dz/du =
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>

 du/dx = d(x * y)/dx = y =
[2. 2. 2. 2.]
<NDArray 4 @cpu(0)>



 x.grad --> dz/dx =
[2. 2. 2. 2.]
<NDArray 4 @cpu(0)>
```

4.3.6 Computing the Gradient of Python Control Flow

- 자동 미분 기능을 사용함으로서 얻을 수 있는 또 하나의 장점은 조건문, 반복문이 포함된 함수를 통해서도 미분값을 구할 수 있다는 점. (단 입력된 값에 따라 미분값이 달라짐)

```
In [14]: def f(a):
             b = a * 2

             while b.norm().asscalar() < 1000:
                 b = b * 2

             if b.sum().asscalar() > 0:
```

```
        c = b
    else:
        c = 100 * b

    return c
```

```
In [17]: a = nd.random.normal(shape=1)
         # a = nd.array([-1.3])
         a.attach_grad()

         with autograd.record():
             d = f(a)

         d.backward()
         # d.backward()
```

```
In [19]: print('a : ', a)
         print('d : ', d)
         print('a.grad : ', a.grad)

         # 결과값 d, 입력값 a 이므로 아래와 같은 수식이 성립.
         print(a.grad == (d / a))
```

```
a :
[0.7740038]
<NDArray 1 @cpu(0)>
d :
[1585.1598]
<NDArray 1 @cpu(0)>
a.grad :
[2048.]
<NDArray 1 @cpu(0)>

[1.]
<NDArray 1 @cpu(0)>
```

4.3.7 Training Mode and Prediction Mode

- record 함수 호출될 때 기본적으로 train_mode 로 변경됨. 참고 : https://mxnet.incubator.apache.org/api/python/docs/_modules/mxnet/autograd.html#record
- section 6.6 dropout, 9.5 bacth normalization 에서 running mode 이용한 테크닉 설명됨.

```
In [87]: print(autograd.is_training())

         # record 함수 호출 시 default 로 train_mode = True 로 동작
         # with autograd.record(train_mode=False):
         with autograd.record():
             print(autograd.is_training())
```

```
False
True
```

```
In [77]: with autograd.predict_mode():
             print(autograd.is_training())

         with autograd.train_mode():
             print(autograd.is_training())
```

```
False
True
```

4.3.8 Summary

- MXNet provides an autograd package to automate the derivation process. To do so, we first attach gradients to variables, record the computation, and then run the backward function.
- We can detach gradients and pass head gradients to the backward function to control the part of the computation will be used in the backward function.
- The running modes of MXNet include the training mode and the prediction mode. We can determine the running mode by autograd.is_training().
