en-ch4.1_Data_Manapulation

2019년 10월 8일

1 Chapter 4 - THE PRELIMINARIES: A CRASHCOURSE

1.1 4.1 Data Manapulation

4.1.1 Getting Started

```
In [2]: from mxnet import nd
```

- NDArrays represent (possibly multi-dimensional) arrays of numerical values. NDArrays with one axis cor- respond (in math-speak) to vectors. NDArrays with two axes correspond to matrices. For arrays with more than two axes, mathematicians do not have special names---they simply call them ***tensors***

```
In [3]: x = nd.arange(12)
        x

Out[3]:
        [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
        <NDArray 12 @cpu(0)>

In [4]: x.shape

Out[4]: (12,)

In [5]: x.reshape((3, 4))

Out[5]:
        [[ 0.  1.  2.  3.]
         [ 4.  5.  6.  7.]
         [ 8.  9. 10. 11.]]
        <NDArray 3x4 @cpu(0)>
```

- We can invoke this capability by placing ***-1*** for the dimension that we would like NDArray to automatically infer. In our case, instead of x.reshape((3, 4)), we could have equivalently used x.reshape((-1, 4)) or x. reshape((3, -1)).

```
In [7]: x.reshape((3, -1))

Out[7]:
        [[ 0.  1.  2.  3.]
         [ 4.  5.  6.  7.]
         [ 8.  9. 10. 11.]]
        <NDArray 3x4 @cpu(0)>

In [7]: x.reshape((4, -1))

Out[7]:
        [[ 0.  1.  2.]
         [ 3.  4.  5.]
         [ 6.  7.  8.]
         [ 9. 10. 11.]]
        <NDArray 4x3 @cpu(0)>

In [8]: x.reshape((6, -1))

Out[8]:
        [[ 0.  1.]
         [ 2.  3.]
         [ 4.  5.]
         [ 6.  7.]
         [ 8.  9.]
         [10. 11.]]
        <NDArray 6x2 @cpu(0)>

In [9]: nd.empty((3, 4))

Out[9]:
        [[1.1e-44 0.0e+00 0.0e+00 0.0e+00]
         [0.0e+00 0.0e+00 0.0e+00 0.0e+00]
         [0.0e+00 0.0e+00 0.0e+00 0.0e+00]]
        <NDArray 3x4 @cpu(0)>
```

- The empty method just grabs some memory and hands us back a matrix without setting the values of any of its entries. his is very effcient but it means that the entries might take any arbitrary values, ***including very big ones!***

4.1.2 Operations

```
In [10]: x = nd.array([1, 2, 4, 8])
         print(x)

         # f: ones_like(x) -> x 와 같은 shape 를 가진 1로 채워진 matrix 생성
         y = nd.ones_like(x) * 2
         print(y)
```

```
[1. 2. 4. 8.]
<NDArray 4 @cpu(0)>

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

```
In [11]: print('x =', x)
         print('x + y', x + y)
         print('x - y', x - y)
         print('x * y', x * y)
         print('x * y', x ** y)
         print('x / y', x / y)
```

```
x =
[1. 2. 4. 8.]
<NDArray 4 @cpu(0)>
x + y
[ 3.  4.  6. 10.]
<NDArray 4 @cpu(0)>
x - y
[-1.  0.  2.  6.]
<NDArray 4 @cpu(0)>
x * y
```

```
[ 2.  4.  8. 16.]
<NDArray 4 @cpu(0)>
x * y
[ 1.  4. 16. 64.]
<NDArray 4 @cpu(0)>
x / y
[0.5 1.  2.  4. ]
<NDArray 4 @cpu(0)>
```

```
In [12]: # 오일러수에 x 를 지수 입력으로, 지수함수 계산한 값을 리턴
         x, x.exp()

Out[12]: (
          [1. 2. 4. 8.]
          <NDArray 4 @cpu(0)>,
          [2.7182817e+00 7.3890562e+00 5.4598148e+01 2.9809580e+03]
          <NDArray 4 @cpu(0)>)

In [13]: x = nd.arange(12).reshape((3,4))
         y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
         print(x)
         print(y)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 3x4 @cpu(0)>

[[2. 1. 4. 3.]
 [1. 2. 3. 4.]
 [4. 3. 2. 1.]]
<NDArray 3x4 @cpu(0)>
```

```
In [14]: nd.dot(x, y.T)
```

```
Out[14]:
         [[ 18.  20.  10.]
          [ 58.  60.  50.]
          [ 98. 100.  90.]]
         <NDArray 3x3 @cpu(0)>
```

- We can also merge multiple NDArrays. For that, we need to tell the system along which dimension to merge. The example below merges two matrices along dimension 0 (along rows) and dimension 1 (along columns) respectively.

```
In [15]: nd.concat(x, y, dim=0)

Out[15]:
         [[ 0.  1.  2.  3.]
          [ 4.  5.  6.  7.]
          [ 8.  9. 10. 11.]
          [ 2.  1.  4.  3.]
          [ 1.  2.  3.  4.]
          [ 4.  3.  2.  1.]]
         <NDArray 6x4 @cpu(0)>

In [103]: nd.concat(x, y, dim=1)

Out[103]:
          [[ 0.  1.  2.  3.  2.  1.  4.  3.]
           [ 4.  5.  6.  7.  1.  2.  3.  4.]
           [ 8.  9. 10. 11.  4.  3.  2.  1.]]
          <NDArray 3x8 @cpu(0)>

In [104]: x == y

Out[104]:
          [[0. 1. 0. 1.]
           [0. 0. 0. 0.]
           [0. 0. 0. 0.]]
          <NDArray 3x4 @cpu(0)>

In [105]: x.sum()
```

```
Out[105]:
          [66.]
          <NDArray 1 @cpu(0)>

In [106]: # L2 norm
          x.norm().asscalar()

Out[106]: 22.494442
```

4.1.3 Broadcast Mechanism

```
In [10]: a = nd.arange(3).reshape((3, 1))
         b = nd.arange(2).reshape((1, 2))
         a, b

Out[10]: (
          [[0.]
           [1.]
           [2.]]
          <NDArray 3x1 @cpu(0)>,
          [[0. 1.]]
          <NDArray 1x2 @cpu(0)>)

In [11]: a + b

Out[11]:
         [[0. 1.]
          [1. 2.]
          [2. 3.]]
         <NDArray 3x2 @cpu(0)>

In [109]: c = nd.arange(12).reshape((3, 2, 2))
          d = nd.arange(4).reshape((2, 2))
          c, d

Out[109]: (
           [[[ 0.  1.]
             [ 2.  3.]]
```

```
             [[ 4.  5.]
              [ 6.  7.]]

             [[ 8.  9.]
              [10. 11.]]]
            <NDArray 3x2x2 @cpu(0)>,
            [[0. 1.]
             [2. 3.]]
            <NDArray 2x2 @cpu(0)>)
```

```
In [110]: c + d
```

```
Out[110]:
           [[[ 0.  2.]
             [ 4.  6.]]

            [[ 4.  6.]
             [ 8. 10.]]

            [[ 8. 10.]
             [12. 14.]]]
           <NDArray 3x2x2 @cpu(0)>
```

-

4.1.4 indexing and Slicing

```
In [15]: x = nd.arange(12).reshape((3,4))
         print(x)
         print(x[1:3])
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 3x4 @cpu(0)>
```

```
[[ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 2x4 @cpu(0)>
```

```
In [16]: x[1, 2] = 9
         x
```

```
Out[16]:
         [[ 0.  1.  2.  3.]
          [ 4.  5.  9.  7.]
          [ 8.  9. 10. 11.]]
         <NDArray 3x4 @cpu(0)>
```

```
In [17]: x[0:2, :] = 12
         x
```

```
Out[17]:
         [[12. 12. 12. 12.]
          [12. 12. 12. 12.]
          [ 8.  9. 10. 11.]]
         <NDArray 3x4 @cpu(0)>
```

4.1.5 Saving Memory

- 매트릭스 변수 할당 시에 잘못하면 메모리 낭비가 심할 수 있으므로 참고

```
In [18]: y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
         y
```

```
Out[18]:
         [[2. 1. 4. 3.]
          [1. 2. 3. 4.]
          [4. 3. 2. 1.]]
         <NDArray 3x4 @cpu(0)>
```

```
In [19]: before = id(y)
         print(before)
```

```
        # 임시 메모리에 y + x 결과 먼저 할당 후에, y 에 해당 메모리를 가르키게 함. 이전 메모리는 ....
        y = y + x
        print(y)
        print(id(y))

        id(y) == before
```

```
140223971748864

[[14. 13. 16. 15.]
 [13. 14. 15. 16.]
 [12. 12. 12. 12.]]
<NDArray 3x4 @cpu(0)>
140223971749728
```

```
Out[19]: False
```

```
In [44]: z = y.zeros_like()
         print('id(z):', id(z))
         z[:] = x + y
         print('id(z):', id(z))
```

```
id(z): 139962319609360
id(z): 139962319609360
```

```
In [45]: before = id(z)
         nd.elemwise_add(x, y, out=z)
         id(z) == before
```

```
Out[45]: True
```

```
In [55]: before = id(x)
         x += y
         id(x) == before
```

```
Out[55]: True
```

4.1.6 Mutual Transformation of NDArray and Numpy

```
In [51]: import numpy as np

         a = x.asnumpy()
         print(type(a))

         b = nd.array(a)
         print(type(b))
```

```
<class 'numpy.ndarray'>
<class 'mxnet.ndarray.ndarray.NDArray'>
```
