파이썬으로 배우는 기계학습

Machine Learning with Python

제 2-2 강: 넘파이$NumPy$ 튜토리얼(2/3)

학습 목표

- 기계학습에서 왜 넘파이를 사용하는지 이해한다
- 넘파이 개념과 기본적인 사용법을 익힌다.

학습 내용

1. 배열의 형상 다루기
2. 배열들 합치기
3. 배열의 인덱싱과 슬라이싱
4. 배열과 벡터의 연산

In [54]:
```
import numpy as np
```

1. 배열의 형상(크기) 다루기

기존의 만들어진 배열의 내부 데이터는 보존한 채로 형상만 바꿀 수 있는 다양한 함수 혹은 메소드가 있습니다.

- reshape
- flatten
- ravel

- newaxis

예를 들어, 가장 대표적으로 `reshape` 메서드는 12개의 원소를 가진 1차원 행렬은 3x4 형상의 2차원 행렬로 만들 수 있습니다.

In [55]:
```
a = np.arange(12)
print(a.shape)
print(a)
```

```
(12,)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [56]:
```
b = a.reshape(3, 4)
b
```

Out[56]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

사용하는 원소의 갯수가 정해저 있기 때문에 `reshape` 메소드의 형상 튜플의 원소 중 하나는 -1이라는 숫자로 대체할 수 있습니다. -1을 넣으면 해당 숫자는 메소드 자체에서 자동으로 계산해서 처리합니다.

In [57]:
```
c = a.reshape(3, -1)
c
```

Out[57]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [58]:
```
d = a.reshape(2, 2, -1)
d
```

Out[58]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

In [59]:

```
e = a.reshape(-1, 2, 2)
print('a:', a)
print('e:', e)
```

```
a: [ 0  1  2  3  4  5  6  7  8  9 10 11]
e: [[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
```

다차원 배열을 1차원으로 펼치기 위해서는 flatten 나 ravel 메소드를 사용합니다.

In [60]:

```
f = e.flatten()
print('e:', e)
print('f:', f)
```

```
e: [[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
f: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [61]:
```
f = e.ravel()
print('e:', e)
print('f:', f)
```

```
e: [[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
f: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

배열 사용에서 주의할 점은 길이가 5인 1차원 배열과 행, 열의 갯수가 (5,1)인 2차원 배열 또는 행, 열의 갯수가 (1, 5)인 2차원 배열은 데이터가 같아도 엄연히 다른 종류의 객체라는 것입니다.

In [62]:
```
x = np.arange(5)
print(x.shape)
print(x)
```

```
(5,)
[0 1 2 3 4]
```

In [63]:
```
y = x.reshape(1, 5)
print(y.shape)
print(y)
```

```
(1, 5)
[[0 1 2 3 4]]
```

In [64]:
```
y = x.reshape(5, 1)
print(y.shape)
print(y)
```

```
(5, 1)
[[0]
 [1]
 [2]
 [3]
 [4]]
```

이렇게 같은 배열에 대해 차원만 1차원 증가시키는 경우에는 newaxis 명령을 사용하기도 한다.

In [65]:
```
y = x[:, np.newaxis]
print(y.shape)
print(y)
```

```
(5, 1)
[[0]
 [1]
 [2]
 [3]
 [4]]
```

2. 배열들 합치기

행의 수나 열의 수가 같은 두 개 이상의 배열을 연결하여(concatenate) 더 큰 배열을 만들 때는 다음과 같은 명령을 사용합니다.

1. hstack
2. vstack
3. dstack
4. stack
5. r_
6. c_
7. tile
8. concatenate

1. hstack

hstack 함수는 행의 수가 같은 두 개 이상의 배열을 옆으로 연결하여 열의 수가 더 많은 배열을 만듭니다. 연결할 배열은 하나의 리스트에 담아야 합니다.

In [66]:
```
a = np.ones((3, 2))
a
```

Out[66]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

In [67]:
```
b = np.zeros((3, 3))
b
```

Out[67]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [68]:
```
np.hstack([a, b])
```

Out[68]:
```
array([[1., 1., 0., 0., 0.],
       [1., 1., 0., 0., 0.],
       [1., 1., 0., 0., 0.]])
```

2. vstack

vstack 함수는 열의 수가 같은 두 개 이상의 배열을 위아래로 연결하여 행의 수가 더 많은 배열을 만듭니다. 연결할 배열은 마찬가지로 하나의 리스트에 담아야 합니다.

In [69]:
```
a = np.ones((2, 3))
a
```

Out[69]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [70]:
```
b = np.zeros((3, 3))
b
```

Out[70]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [71]:
```
np.vstack([a, b])
```

Out[71]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

3. dstack

dstack 함수는 제3의 축 즉, 행이나 열이 아닌 깊이(depth) 방향으로 배열을 합칩니다. 가장 안쪽의 원소의 차원이 증가합니다. 즉 가장 내부의 숫자 원소가 배열이 됩니다. shape 정보로 보자면 가장 끝에 값이 2인 차원이 추가되는 것입니다.이 예제의 경우에는 shape 변화가 2개의 (3 x 4) -> 1개의 (3 x 4 x 2)가 됩니다.

In [72]:
```
a = np.arange(12).reshape(3, 4)
a
```

Out[72]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [73]:
```
b = np.arange(12).reshape(3, 4)+50
b
```

Out[73]:
```
array([[50, 51, 52, 53],
       [54, 55, 56, 57],
       [58, 59, 60, 61]])
```

In [74]:
```
np.dstack([a, b])
```

Out[74]:
```
array([[[ 0, 50],
        [ 1, 51],
        [ 2, 52],
        [ 3, 53]],

       [[ 4, 54],
        [ 5, 55],
        [ 6, 56],
        [ 7, 57]],

       [[ 8, 58],
        [ 9, 59],
        [10, 60],
        [11, 61]]])
```

4. stack

stack 함수는 dstack 의 기능을 확장한 것으로 dstack 처럼 마지막 차원으로 연결하는 것이 아니라 사용자가 지정한 차원(축으로) 배열을 연결합니다. axis 인수(디폴트 0)를 사용하여 연결후의 회전 방향을 정합니다. 디폴트 인수값은 0이고 가장 앞쪽에 차원이 생성됩니다. 즉, 배열 두 개가 겹치게 되므로 연결하고자 하는 배열들의 크기가 모두 같아야 합니다.

다음 예에서는 axis=0 이므로 가장 값에 값이 2인 차원이 추가됩니다. 즉, shape 변화는 2개의 (3 x 4) -> 1개의 (2 x 3 x 4) 입니다.

In [75]:
```
c = np.stack([a, b])
c
```

Out[75]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[50, 51, 52, 53],
        [54, 55, 56, 57],
        [58, 59, 60, 61]]])
```

In [76]:
```
c.shape
```

Out[76]:
```
(2, 3, 4)
```

`axis` 인수가 1이면 두번째 차원으로 새로운 차원이 삽입됩니다. 다음 예에서 즉, shape 변화는 2개의 (3 x 4) -> 1개의 (3 x 2 x 4) 입니다.

In [77]:
```
c = np.stack([a, b], axis=1)
c
```

Out[77]:
```
array([[[ 0,  1,  2,  3],
        [50, 51, 52, 53]],

       [[ 4,  5,  6,  7],
        [54, 55, 56, 57]],

       [[ 8,  9, 10, 11],
        [58, 59, 60, 61]]])
```

In [78]:
```
c.shape
```

Out[78]:
```
(3, 2, 4)
```

5. _r

`r_` 메서드는 `hstack` 함수와 비슷하게 배열을 좌우로 연결합니다. 다만 메서드임에도 불구하고 소괄호(parenthesis, ())를 사용하지 않고 인덱싱과 같이 대괄호(bracket, `[]`)를 사용합니다. 이런 특수 메서드를 **인덱서(indexer)**라고 합니다.

In [79]:
```
np.r_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

Out[79]:
```
array([1, 2, 3, 4, 5, 6])
```

6. _c

`c_` 메서드는 배열의 차원을 증가시킨 후 좌우로 연결합니다. 만약 1차원 배열을 연결하면 2차원 배열이 됩니다.

In [80]:
```
np.c_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

Out[80]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

7. tile

`tile` 명령은 동일한 배열을 반복하여 연결한다.

In [81]:
```
a = np.array([[0, 1, 2], [3, 4, 5]])
np.tile(a, 2)
```

Out[81]:
```
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

In [82]:
```
np.tile(a, (3, 2))
```

Out[82]:
```
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5],
       [0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5],
       [0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

8. concatenate

두 배열이 같은 차원일 때 사용합니다. 예시를 통해서 살펴보면, 직감적으로 알 수 있습니다.

In [83]:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[7, 8], [9, 10], [11, 12]])

print(a.shape)
print(b.shape)

print(np.concatenate((a, b), axis=0))
print(np.concatenate((a, b.T), axis=1))
print(np.concatenate((a, b), axis=None))
```

```
(2, 2)
(3, 2)
[[ 1  2]
 [ 3  4]
 [ 7  8]
 [ 9 10]
 [11 12]]
[[ 1  2  7  9 11]
 [ 3  4  8 10 12]]
[ 1  2  3  4  7  8  9 10 11 12]
```

3. 배열의 인덱싱과 슬라이싱

기계학습을 하면서, 다차원 배열을 부분적으로 다루어야 할 때가 많습니다. 인덱싱과 슬라이싱이 필요한 것입니다.

배열의 인덱싱은 0부터 시작하고, 또한 음수 인덱싱도 가능하여 배열의 끝은 -1부터 시작합니다. 또한 범위를 지정할 때, 콜론(:)을 사용합니다. 예를 들어, `[:, :]` 2차원 배열의 모든 행과 열을 나타내고, `[:2]` 는 배열의 0, 1의 행, `[:, 1:3]` 은 1, 2열을 나타냅니다. 일반적으로 파이썬에서 범위(인덱싱)를 지정할 때 범위의 끝은 포함하지 않습니다.

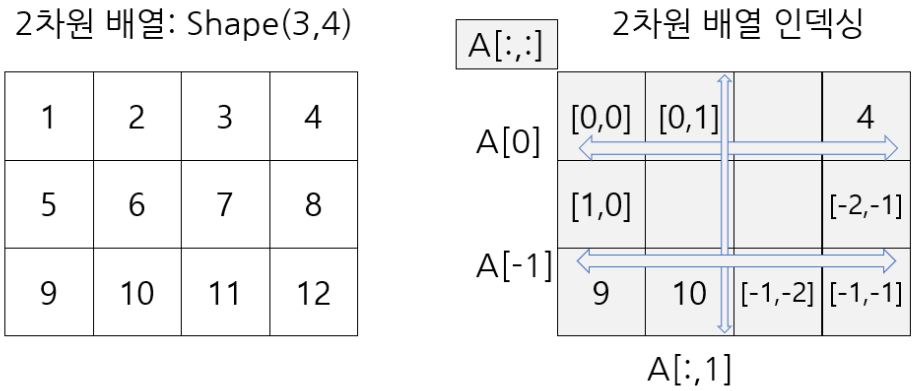

그림 1: 넘파이 배열 인덱싱

인덱싱 연습을 위해, 1 부터 12까지 수로 이루어진 형상이 (3, 4)인 배열을 생성해 봅시다.

In [27]:
```
import numpy as np
np_arr = np.arange(1, 13).reshape(3, 4)
print(np_arr)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

원소 인덱싱

원소 하나씩 접근하는 것은 일반적인 배열 인덱싱과 같습니다. 다만, 파이썬의 특이한 점은 인덱싱으로 음수를 사용하며, -1은 배열의 끝을 나타냅니다. 그러므로, 원소 12의 인덱싱은 `[-1, -1]` 이며, 이런 인덱싱은 2차원 배열의 크기가 바뀔지라도 항상 마지막 원소를 가리키는 편리함이 있습니다.

In [28]:
```
np_arr[-1, -1]
```

Out[28]: 12

행, 열 인덱싱

행의 인덱싱은 행의 인덱스로만 가능하지만, 열을 인덱싱할 때는 행의 위치에 콜론(:)을 지정해야 합니다. 다음은 배열 `np_arr` 의 첫째 행과 열을 인덱싱합니다

In [29]:
```
print(np_arr[0])
print(np_arr[:,0])
```

```
[1 2 3 4]
[1 5 9]
```

다음과 같이 배열 `np_arr` 의 마지막 행과 열을 인덱싱하여 출력하도록 코딩하십시오.

```
[ 9 10 11 12]
[ 4  8 12]
```

In [30]:
```
print(np_arr[-1])
print(np_arr[:, -1])
```

```
[ 9 10 11 12]
[ 4  8 12]
```

3.1 배열의 슬라이싱

콜론(:)을 사용하여 start:end와 같이 범위를 지정할 수 있습니다. end는 자신을 포함하지 않고, start가 생략이 되면 0부터를 의미하며, end가 생략되면 끝까지를 의미합니다.

자, 그러면 위 왼쪽 코너의 4개의 원소 즉 `[[1, 2], [5,6]]` 와 아래 오른쪽 코너 4개의 원소 즉 `[[7, 8], [11,12]]` 를 슬라이싱하여 각각 b, c에 저장을 시도하겠습니다.

그림 2: Numpy 배열 인덱싱

Example 1

아래의 4x4 배열에서 첫 세개의 열(4x3) X 와 마지막 한 개의 열(4x1) y로 슬라이싱 하여 출력하십시오.

```
X=[[1 0 0]
   [1 0 1]
   [1 1 0]
   [1 1 1]]

y=[0 1 1 0]
```

In [126]:
```
data = np.array([[1, 0, 0, 0], [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 0]])
print(data)
X, y = data[:, :3], data[:, 3]
print('X', X)
print('y', y)
```

```
[[1 0 0 0]
 [1 0 1 1]
 [1 1 0 1]
 [1 1 1 0]]
X [[1 0 0]
 [1 0 1]
 [1 1 0]
 [1 1 1]]
y [0 1 1 0]
```

Example 2.

다음 코드 셀에서 생성하는 3x4 배열에서 가장 가운데 있는 배열의 원소 6, 7를 인덱싱하여 출력하십시오.

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[6 7]]
```

In [ ]:
```
np_arr = np.arange(1, 13).reshape(3, 4)
print(np_arr)
print(np_arr[1:-1, 1:-1])
```

Example 3.

다음 코드 셀에서, 아래와 같이 주어진 배열의 가장자리의 행과 열을 제외한 나머지 배열을 인덱싱하여 다음과 같이 출력하십시오. (위의 문제와 답이 같을 수 있습니다.)

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]
 [25 26 27 28 29 30]]
[[ 8  9 10 11]
 [14 15 16 17]
 [20 21 22 23]]
```

In [ ]:

```
np_arr = np.arange(1, 31).reshape(5, 6)
print(np_arr)
print(np_arr[1:-1, 1:-1])
```

Example 4.

다음 코드 셀에서, 아래와 같이 주어진 배열의 짝수번째 열을 제외한 모든 요소를 포함하는 배열을 출력하십시오.

```
[[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]
 [17 18 19 20 21 22 23 24]]
[[ 1  3  5  7]
 [ 9 11 13 15]
 [17 19 21 23]]
```

In [ ]:

```
np_arr = np.arange(1, 25).reshape(3, 8)
print(np_arr)
print(np_arr[:, ::2])
```

3.2 서브배열은 원 배열의 view(뷰)

제목이 무슨 말인지 이해가 가지 않지만 그대로 진행해 봅시다. 여기서 원 배열은 `a` 배열입니다.

b 의 첫 원소를 값을 99 으로 바꾸고 b 를 출력하고, a 를 출력해봅시다. 그러면, a 배열도 수정된 것을 볼 수 있습니다. 유의해야 할 점입니다. 이러한 사실을 간과하면, 언제가 몇 시간이나 며칠 동안 디버깅을 해야 할지도 모릅니다.

In [84]:
```
a = np.arange(1, 13).reshape(3,4)
b = a[:2, :2]
print(b)
b[0, 0] = 99
print(a)
```

```
[[1 2]
 [5 6]]
[[99  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

원 배열에서 일부 잘라낸 배열을 서브배열^{subarray}라고 부르는 이유가 여기에 있습니다.

view(뷰)라는 말은 무슨 말인가요? C프로그래밍의 포인터 개념입니다. 포인터가 가리키고 있는 내용을 바꾸면 원래의 저장된 변수의 내용이 바뀌는 것과 같습니다.

3.3 배열의 복사

배열을 복사할 때도 조심해야 합니다. 다음과 같이 aa = a 를 하여 a 를 복사했다고 생각하면 잘못입니다. 여기서 aa 도 역시 view(뷰)입니다.

In [85]:
```
a = np.arange(1, 13).reshape(3,4)
aa = a
aa[0, 0] = 99
print(a)
```

```
[[99  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

그러면, 배열의 복사는 어떻게 해야 합니까? 배열 ndarray클래스가 제공하는 copy() 메소드를 사용해야 합니다.

In [86]:
```
a = np.arange(1, 13).reshape(3,4)
aa = a.copy()
aa[0, 0] = 99
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

3.4 행이나 열을 슬라이싱하기

두 가지 방법이 있습니다. 하나는 1차원 배열로 하는 방법 또 하나는 2차원 배열로 슬라이싱하는 방법입니다.

1. 두 번째 행([5 6 7 8])을 슬라이싱할 때 (4,) 혹은 (1, 4) 형상으로 슬라이싱하는 방법이 있습니다.
2. 두 번째 열([2 6 10])을 슬라이싱할 때도, (3,) 혹은 (3, 1) 형상으로 슬라이싱하는 방법이 있습니다.

In [87]:
```
a = np.arange(1, 13).reshape(3,4)
row1 = a[1]      # shape(4,)
print(row1)
```

```
[5 6 7 8]
```

In [88]:
```
a = np.arange(1, 13).reshape(3,4)
print(a)
row1 = a[1, :]      # shape(4,), sames as a[1]
row2 = a[1:2, :]    # shape(1, 4), same as a[1:2]
print(row1)
print(row2)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[5 6 7 8]
[[5 6 7 8]]
```

In [89]:
```
col1 = a[:, 1]      # shape(3,)
col2 = a[:, 1:2]    # shape(3, 1)
print(col1)
print(col2)
```

```
[ 2  6 10]
[[ 2]
 [ 6]
 [10]]
```

3.5 불린*Boolean* 배열 인덱싱

기계학습에서 종종 사용되는 인덱싱입니다. 어떤 조건을 만족하는 배열의 원소를 선택할 때 사용합니다. 분류를 한 결과가 배열로 저장되어 있을 때, 어떤 임계값보다 큰 모든 원소의 갯수나 인덱스를 구할 수 있습니다.

난수로 이루어진 배열에서 0.6 이상인 원소의 갯수와 인덱스를 찾아내는 코드입니다.

In [90]:
```
a = np.random.random(12)  # np.random.random(12).reshape(3,4)
index = a > 0.6
print(index)
print(np.sum(index))
print(np.argwhere(index))
```

```
[ True  True  True False  True False False  True False  True False False]
6
[[0]
 [1]
 [2]
 [4]
 [7]
 [9]]
```

4. 배열과 벡터의 연산

행렬은 여러 벡터로 구성된 데이터 셋입니다. 하나의 벡터 안에는 원소의 값이 같은 형식(type)이며, 이러한 벡터를 여러 개 모은 것이 행렬입니다. 원소가 3개인 벡터가 2개 있으면 2행, 3열의 행렬, 2x3 행렬입니다.

4.1 원소별 사칙 연산

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix}$$

In [91]:
```
import numpy as np

A = np.array([[1, 2],[3, 4]])
B = np.array([[5, 6],[7, 8]])
```

In [92]:
```
print(A + B)
```

```
[[ 6  8]
 [10 12]]
```

In [93]:
```
print(A - B)
```

```
[[-4 -4]
 [-4 -4]]
```

In [94]:
```
print(A * B)
```

```
[[ 5 12]
 [21 32]]
```

In [95]:
```
print(A / B)
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

4.2 행 벡터와 열 벡터

선형 대수학에서, 행 벡터는 행 행렬 1xm 행렬, 즉 m 원소들로 구성된 단일 행 행렬이며 다음과 같이 나타낼 수 있습니다.

$$\mathbf{w} = \begin{pmatrix} w_1 & w_2 & \cdots & w_m \end{pmatrix}$$

마찬가지로, 열 벡터는 열 행렬 mx1 행렬, 즉 m 원소들의 구성된 단일 열 행렬이며 다음과 같이 표시합니다.

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

행 벡터의 전치 행렬은 열 벡터이며, 열 벡터의 전치^{transpose} 행렬은 행 벡터입니다.

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_m \end{pmatrix}^T = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}, \qquad \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}^T = \begin{pmatrix} x_1 & x_2 & \cdots & x_m \end{pmatrix}$$

4.3 점곱(혹은 내적, inner product)

행 벡터와 열 벡터의 점곱

행 벡터 $\mathbf{w}$와 열 벡터 $\mathbf{x}$를 각각 원소별로 곱한 것을 합산함으로 내적(inner product, 혹은 점곱)을 구할 수 있습니다.

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x} &= \begin{pmatrix} w_1 & w_2 & \cdots & w_m \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \\ &= \sum_{i=1}^{m} w_i x_i \\ &= w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \end{aligned}$$

열 벡터와 행 벡터의 점곱

만약, 우리가 열(col) 벡터 `x` 와 행(row) 벡터 `w` 의 내적을 하면 어떻게 될까요?

$$\mathbf{x} \cdot \mathbf{w} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \cdot \begin{pmatrix} w_1 & w_2 & \cdots & w_m \end{pmatrix} = \begin{pmatrix} x_1 w_1 & x_1 w_2 & \cdots & x_1 w_m \\ x_2 w_1 & x_2 w_2 & \cdots & x_2 w_m \\ \cdots & & \cdots & \cdots \\ x_m w_1 & x_m w_2 & \cdots & x_m w_m \end{pmatrix}$$

넘파이 벡터의 형상

벡터를 1차원 배열로 쉽게 나타낼 수도 있지만, 만약 그렇게 한다면, 행(row)과 열(col)의 점곱은 스칼라 값을 얻을 수 있지만, 열(col)과 행(row) 벡터의 점곱은 얻을 수 없습니다. 행/열의 점곱과 열/행의 점곱을 온전히 계산하려면, 벡터를 1차원 배열이 아니라, 2차원 배열을 사용하여 **w**를 `1 x m` 형상으로, **x**를 `m x 1` 형상으로 생성해야 합니다.

그러면, 어떻게 하면,**w**, **x**를 **진정한** 열 벡터와 행 벡터로 생성할 수 있을까요?

reshape()를 사용하여, 원하는 shape을 지정하거나 벡터를 생성할 때 겹 `[]` 을 사용하여 2차원 배열로 만들어야 합니다. 다음과 같이 말입니다.

```
row_vector = np.array([[0, 1, 2, 3]])  col_vector = np.array([[4, 3, 2, 1]])
```

In [96]:

```
w = np.array(np.arange(4)).reshape(1, 4)
x = np.array(np.arange(4, 0, -1)).reshape(4, 1)
print(w)
print(x)
print(w.ndim)
```

```
[[0 1 2 3]]
[[4]
 [3]
 [2]
 [1]]
2
```

In [97]:
```
np.dot(x,w)
```

Out[97]:
```
array([[ 0,  4,  8, 12],
       [ 0,  3,  6,  9],
       [ 0,  2,  4,  6],
       [ 0,  1,  2,  3]])
```

In [98]:
```
np.dot(w, x)
```

Out[98]:
```
array([[10]])
```

결론적으로, 벡터는 1차원 배열처럼 보이는 2차원 배열입니다. 2차원 배열에서 행(row)이나 열(col)만 있는 배열입니다.

1차원 배열이 아니라, 2차원 배열로 벡터를 표시하면 전치(transpose도 작동합니다.

In [99]:
```
print(w.T)
print(x.T)
```

```
[[0]
 [1]
 [2]
 [3]]
[[4 3 2 1]]
```

1차원 배열은 벡터가 아니지만, 점곱이 가능합니다.

그런데, 만약, 우리가 두 개으 1차원 배열 `w = [0 1 2 3] x = [4 3 2 1]` 를 점곱하면 스칼라 10을 얻습니다. w, x 두 벡터가 모두 똑같이 shape이 (4,)이며, 서로 구별할 수 없으며, `np.dot(w, x)` 와 `np.dot(x, w)` 는 같은 결과를 도출합니다. 결론적으로, w, x는 벡터가 아니고 1차원 배열입니다.

In [100]:
```
import numpy as np
w = np.array(np.arange(0, 4))          # w = [0 1 2 3]
x = np.array(np.arange(4, 0, -1))      # x = [4 3 2 1]
print(w)
print(w.shape)
print(x)
print(x.shape)
print(np.dot(w, x))
```

```
[0 1 2 3]
(4,)
[4 3 2 1]
(4,)
10
```

In [101]:
```
np.dot(w, x)
```

Out[101]: 10

두 배열의 내적 결과는 나왔지만, 사실 행 벡터와 열 벡터의 점곱(dot product)를 한 것은 아닙니다. 왜냐하면, w, x는 1차원 배열이지 사실상 벡터는 아닙니다. 그 형상을 살펴보면, 모두 **똑같이 형상이 (4,)이기 때문입니다**.

4.3 행렬의 내적

Numpy ndarray 클래스의 `dot` 메소드를 사용해서 행렬의 내적을 구할 수 있습니다.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{pmatrix}$$

In [102]:
```
A = np.array([[1, 2],[3, 4]])
B = np.array([[5, 6],[7, 8]])
print(np.dot(A, B))
```

```
[[19 22]
 [43 50]]
```

In [103]:
```
print(A.dot(B))
```

```
[[19 22]
 [43 50]]
```

4.4 행렬의 전치*transpose*

기계학습 코딩을 하다보면, 종종 배열의 모양을 바꾸어야 할 할 때가 있습니다. 가장 간단한 예는 행렬의 주 대각선을 기준으로 대칭되는 원소끼리 뒤바꾸는 것입니다. 이를 **전치** 라고 하며 행렬을 전치하기 위해선, 간단하게 배열 객체의 'T' 속성을 사용하면 됩니다:

다음과 같이 두 행렬 A 와 B 가 있다고 합시다.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad B = (1 \quad 2 \quad 3)$$

위의 두 행렬을 넘파이 배열로 표현하면 다음과 같습니다. `array` 메소드의 인자로 1차원 파이썬 리스트를 넣으면, 크기가 리스트의 길이와 같은 벡터가 생성됩니다. 넘파이 배열의 형상을 확인하려면 `shape` 메소드를 사용하면 됩니다.

여기서 A, B의 rank는 각각 2, 1 입니다. rank가 1인 배열은 전치를 해도 아무런 변화가 없습니다.

먼저, 두 행렬을 넘파이 배열로 만들고 각각의 shape(형상)을 출력해봅시다.

In [104]:
```
A = np.array([[1,2,3],[4,5,6]])
B = np.array([1,2,3])
print(A.shape, B.shape)
```

```
(2, 3) (3,)
```

두 배열의 형상이 각각 (2,3), (3,) 배열인데, 서로 곱셈이 가능한지 코드로 시도해 봅시다

In [105]:
```
C = np.dot(A, B)
print(C)
print(C.shape)
```

```
[14 32]
(2,)
```

1차원 배열 형상(2,)의 결과를 얻었습니다.

만약에, 우리가 수학의 행렬의 내적처럼 두 행렬 (2 x 3), (3 x 1)을 내적하여 2차원 행렬(2, 1)을 구하길 원한다면 어떻게 해야 할까요?

$$A \bullet B^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \bullet \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 2 \times 2 + 3 \times 3 \\ 4 \times 1 + 5 \times 2 + 6 \times 3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \end{pmatrix}$$

우리가 수학적인 `reshape` 메소드를 사용해서 벡터를 행렬로 만들어줄 수 있습니다. `reshape(1,3)` 는 넘파이 배열을 크기 1 x 3 의 행렬로 형태를 바꾸겠다는 의미입니다.

In [106]:
```
A = np.array([[1,2,3],[4,5,6]])         # rank 2, shape (2, 3)
B = np.array([1,2,3]).reshape(1,3)      # rank 2, shape (1, 3)
C = np.dot(A, B.T)                      # rand 2, shape (2, 1)
print(C)
```

```
[[14]
 [32]]
```

학습 정리

1. 배열의 형상 다루기
2. 배열들 합치기
3. 배열의 인덱싱과 슬라이싱
4. 배열과 벡터의 연산

참고자료

- CS231n Convolutional Neural Networks for Visual Recognition, Python Numpy Tutorial (http://cs231n.github.io/python-numpy-tutorial/), Stanford University
- Python For Data Science Cheat Sheet NumPy Basics (https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf), DataCamp
- astroML (http://www.astroml.org/book_figures/appendix/fig_broadcast_visual.html)
- Python Numpy Tutorial - http://cs231n.github.io/ (http://cs231n.github.io/)
- 김태완 블로그: 파이썬 데이터 사이언스 Cheat Sheet (http://taewan.kim/post/numpy_cheat_sheet/)