2.2 벡터와 행렬의 연산

벡터와 행렬도 숫자처럼 덧셈, 뺄셈, 곱셈 등의 연산을 할 수 있다. 벡터와 행렬의 연산을 이용하면 대량의 데이터에 대한 계산을 간단한 수식으로 나타낼 수 있다. 물론 벡터와 행렬에 대한 연산은 숫자의 사칙 연산과는 몇 가지 다른 점이 있으므로 이러한 차이를 잘 알아야 한다.

이 절에서는 넘파이를 이용하여 벡터와 행렬의 연산을 실행하는 법도 공부한다. 다음처럼 넘파이와 맷플롯립 패키지가 임포트되어 있어야 한다.

In [1]:

```
import numpy as np
import matplotlib.pylab as plt
```

벡터/행렬의 덧셈과 뺄셈

같은 크기를 가진 두 개의 벡터나 행렬은 덧셈과 뺄셈을 할 수 있다. 두 벡터와 행렬에서 같은 위치에 있는 원소끼리 덧셈과 뺄셈을 하면 된다. 이러한 연산을 **요소별(element-wise) 연산**이라고 한다.

예를 들어 벡터 x와 y가 다음과 같으면,

$$x = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

벡터 x와 y의 덧셈 $x+y$와 뺄셈 $x-y$는 다음처럼 계산한다.

$$x + y = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 10+0 \\ 11+1 \\ 12+2 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 14 \end{bmatrix}$$

$$x - y = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 10-0 \\ 11-1 \\ 12-2 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}$$

벡터의 덧셈과 뺄셈을 넘파이로 계산하면 다음과 같다. 여기에서는 편의상 1차원 배열로 벡터를 표시하였다.

In [2]:

```
x = np.array([10, 11, 12, 13, 14])
y = np.array([0, 1, 2, 3, 4])
```

In [3]:

```
x + y
```

Out[3]:

```
array([10, 12, 14, 16, 18])
```

In [4]:

```
x - y
```

Out[4]:

```
array([10, 10, 10, 10, 10])
```

행렬도 같은 방법으로 덧셈과 뺄셈을 할 수 있다.

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} + \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 14 & 24 \\ 34 & 44 \end{bmatrix}$$

In [5]:

```
np.array([[5, 6], [7, 8]]) + np.array([[10, 20], [30, 40]]) - ₩
    np.array([[1, 2], [3, 4]])
```

Out[5]:

```
array([[14, 24],
       [34, 44]])
```

스칼라와 벡터/행렬의 곱셈

벡터 x 또는 행렬 A에 스칼라값 c를 곱하는 것은 **벡터 x 또는 행렬 A의 모든 원소에 스칼라값 c를 곱하는 것**과 같다.

$$c\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} cx_1 \\ cx_2 \end{bmatrix}$$

$$c\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} ca_{11} & ca_{12} \\ ca_{21} & ca_{22} \end{bmatrix}$$

브로드캐스팅

원래 덧셈과 뺄셈은 크기(차원)가 같은 두 벡터에 대해서만 할 수 있다. 하지만 벡터와 스칼라의 경우에는 관례적으로 다음처럼 1-벡터를 사용하여 스칼라를 벡터로 변환한 연산을 허용한다. 이를 **브로드캐스팅(broadcasting)**이라고 한다.

$$\begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} - 10 = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} - 10 \cdot \mathbf{1} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} - \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}$$

데이터 분석에서는 원래의 데이터 벡터 x가 아니라 그 데이터 벡터의 각 원소의 평균값을 뺀 **평균제거(mean removed) 벡터** 혹은 **0-평균(zero-mean) 벡터**를 사용하는 경우가 많다.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \rightarrow x - m = \begin{bmatrix} x_1 - m \\ x_2 - m \\ \vdots \\ x_N - m \end{bmatrix}$$

위 식에서 m은 샘플 평균이다.

$$m = \frac{1}{N}\sum_{i=1}^{N} x_i$$

선형조합

벡터/행렬에 다음처럼 스칼라값을 곱한 후 더하거나 뺀 것을 벡터/행렬의 **선형조합(linear combination)**이라고 한다. 벡터나 행렬을 선형조합해도 크기는 변하지 않는다.

$$c_1 x_1 + c_2 x_2 + c_3 x_3 + \cdots + c_L x_L = x$$

$$c_1 A_1 + c_2 A_2 + c_3 A_3 + \cdots + c_L A_L = A$$

$$c_1, c_2, \ldots, c_L \in \mathbf{R}$$

$$x_1, x_2, \ldots, x_L, x \in \mathbf{R}^M$$

$$A_1, A_2, \ldots, A_L, A \in \mathbf{R}^{M \times N}$$

벡터나 행렬의 크기를 직사각형으로 표시하면 다음과 같다.

$$c_1 \; x_1 \quad + \quad c_2 \; x_2 \quad + \quad \cdots \quad + \quad c_L \; x_L$$

$$c_1 \quad A_1 \quad + \quad c_2 \quad A_2 \quad + \quad \cdots \quad + \quad c_L \quad A_L$$

벡터와 벡터의 곱셈

행렬의 곱셈을 정의하기 전에 우선 두 벡터의 곱셈을 알아보자. 벡터를 곱셈하는 방법은 여러 가지가 있지만 여기서는 **내적(inner product)**에 대해서만 다룬다. 벡터 x와 벡터 y의 내적은 다음처럼 표기한다.

$$x^T y$$

내적은 다음처럼 점(dot)으로 표기하는 경우도 있어서 **닷 프로덕트(dot product)**라고도 부르고 $< x, y >$ 기호로 나타낼 수도 있다.

$$x \cdot y = \ < x, y > \ = x^T y$$

두 벡터를 내적하려면 다음과 같은 조건이 만족되어야 한다.

1. 우선 두 벡터의 차원(길이)이 같아야 한다.
2. 앞의 벡터가 행 벡터이고 뒤의 벡터가 열 벡터여야 한다.

이때 내적의 결과는 스칼라값이 되며 다음처럼 계산한다. 우선 같은 위치에 있는 원소들을 요소별 곱셈처럼 곱한 다음, 그 값들을 다시 모두 더해서 하나의 스칼라값으로 만든다.

$$x^T y = \begin{bmatrix} x_1 & x_2 & \cdots & x_N \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = x_1 y_1 + \cdots + x_N y_N = \sum_{i=1}^{N} x_i y_i$$

$$x \in \mathbf{R}^{N \times 1}$$

$$y \in \mathbf{R}^{N \times 1}$$

$$x^T y \in \mathbf{R}$$

다음은 두 벡터의 내적의 예다.

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad y = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

$$x^T y = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$$

넘파이에서 벡터와 행렬의 내적은 `dot()` 이라는 명령 또는 `@` (at이라고 읽는다)이라는 연산자로 계산한다. 2차원 배열로 표시한 벡터를 내적했을 때는 결과값이 스칼라가 아닌 2차원 배열이다.

In [6]:

```
x = np.array([[1], [2], [3]])
y = np.array([[4], [5], [6]])

x.T @ y  # 또는 np.dot(x.T, y)
```

Out[6]:

```
array([[32]])
```

넘파이에서는 1차원 배열끼리도 내적을 계산한다. 이때는 넘파이가 앞의 벡터는 행 벡터이고 뒤의 벡터는 열 벡터라고 가정한다.

In [7]:

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

x @ y  # 또는 np.dot(x, y)
```

Out[7]:

```
32
```

왜 벡터의 내적은 덧셈이나 뺄셈과 달리 이렇게 복잡하게 정의된 것일까? 그 이유는 데이터 분석을 할 때 이러한 연산이 필요하기 때문이다. 벡터의 내적을 사용하여 데이터를 분석하는 몇 가지 예를 살펴보자.

가중합

벡터의 내적은 가중합을 계산할 때 쓰일 수 있다. **가중합(weighted sum)**이란 복수의 데이터를 단순히 합하는 것이 아니라 각각의 수에 어떤 가중치 값을 곱한 후 이 곱셈 결과들을 다시 합한 것을 말한다.

만약 데이터 벡터가 $x = [x_1, \cdots, x_N]^T$이고 가중치 벡터가 $w = [w_1, \cdots, w_N]^T$이면 데이터 벡터의 가중합은 다음과 같다.

$$w_1 x_1 + \cdots + w_N x_N = \sum_{i=1}^{N} w_i x_i$$

이 값을 벡터 x와 w의 곱으로 나타내면 $w^T x$ 또는 $x^T w$ 라는 간단한 수식으로 표시할 수 있다.

$$\sum_{i=1}^{N} w_i x_i = \begin{bmatrix} w_1 & w_2 & \cdots & w_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = w^T x$$

$$= \begin{bmatrix} x_1 & x_2 & \cdots & x_N \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = x^T w$$

예를 들어 쇼핑을 할 때 각 물건의 가격은 데이터 벡터, 각 물건의 수량은 가중치로 생각하여 내적을 구하면 총 금액을 계산할 수 있다.

만약 가중치가 모두 1이면 일반적인 합(sum)을 계산한다.

$$w_1 = w_2 = \cdots = w_N = 1$$

또는

$$w = \mathbf{1}_N$$

이면

$$\sum_{i=1}^{N} x_i = \mathbf{1}_N^T x$$

연습 문제 2.2.1

A, B, C 세 회사의 주식은 각각 100만원, 80만원, 50만원이다. 이 주식을 각각 3주, 4주, 5주를 매수할 때 필요한 금액을 구하고자 한다.

(1) 주식의 가격과 수량을 각각 p 벡터, n 벡터로 표시하고 넘파이로 코딩한다.

(2) 주식을 매수할 때 필요한 금액을 곱셈으로 표시하고 넘파이 연산으로 그 값을 계산한다.

가중평균

가중합의 가중치값을 전체 가중치값의 합으로 나누면 **가중평균(weighted average)**이 된다. 가중평균은 대학교의 평균 성적 계산 등에 사용할 수 있다.

예를 들어 고등학교에서는 국어, 영어, 두 과목의 평균 점수를 구할 때 단순히 두 과목의 점수(숫자)를 더한 후 2으로 나눈다. 그러나 대학교에서는 중요한 과목과 중요하지 않는 과목을 구분하는 학점(credit)이라는 숫자가 있다. 일주일에 한 시간만 수업하는 과목은 1학점짜리 과목이고 일주일에 세 시간씩 수업하는 중요한 과목은 3학점짜리 과목이다. 1학점과 3학점 과목의 점수가 각각 100점, 60점이면 학점을 고려한 가중 평균(weighted average) 성적은 다음과 같이 계산한다.

$$\frac{1}{1+3} \times 100 + \frac{3}{1+3} \times 60 = 70$$

벡터로 표현된 N개의 데이터의 단순 평균은 다음처럼 생각할 수 있다.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{N}\mathbf{1}_N^T x$$

위 수식에서 보인 것처럼 x 데이터의 평균은 보통 $\bar{x}$라는 기호로 표기하고 "엑스 바(x bar)" 라고 읽는다.

다음은 넘파이로 평균을 계산하는 방법이다.

In [8]:

```
x = np.arange(10)
N = len(x)

np.ones(N) @ x / N
```

Out[8]:

4.5

현실적으로는 `mean()` 이라는 메서드를 사용하는 것이 편하다.

In [9]:

```
x.mean()
```

Out[9]:

4.5

연습 문제 2.2.2

벡터 x의 평균 제거 벡터는 다음과 같이 계산함을 증명하라.

$$x - \frac{1}{N}\mathbf{1}_N^T x \mathbf{1}_N$$

유사도

벡터의 곱셈(내적)은 두 벡터 간의 유사도를 계산하는 데도 이용할 수 있다. **유사도(similarity)는 두 벡터가 닮은 정도를 정량적으로 나타낸 값**으로 두 벡터가 비슷한 경우에는 유사도가 커지고 비슷하지 않은 경우에는 유사도가 작아진다. 내적을 이용하면 **코사인 유사도(cosine similarity)**라는 유사도를 계산할 수 있다. 추후 선형대수의 기하학적 의미를 공부할 때 코사인 유사도에 대해 살펴볼 것이다.

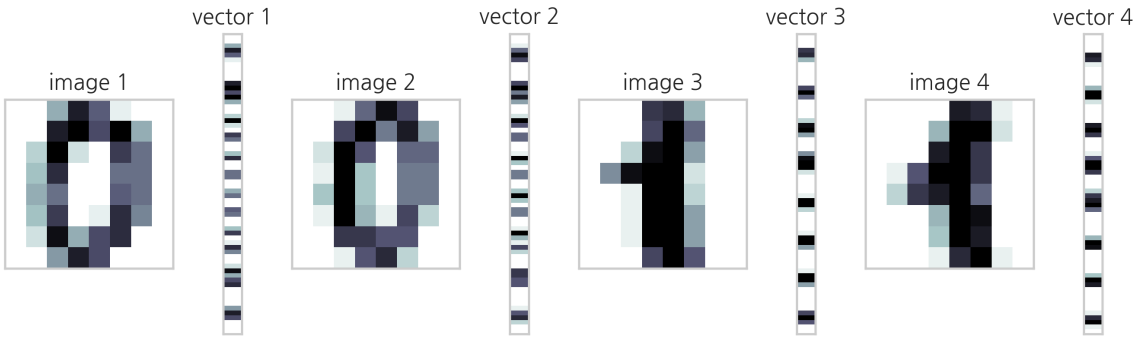
예를 들어 0과 1을 나타내는 MNIST 이미지에 대해 내적을 계산해보자.

In [10]:

```
from sklearn.datasets import load_digits
import matplotlib.gridspec as gridspec

digits = load_digits()
d1 = digits.images[0]
d2 = digits.images[10]
d3 = digits.images[1]
d4 = digits.images[11]
v1 = d1.reshape(64, 1)
v2 = d2.reshape(64, 1)
v3 = d3.reshape(64, 1)
v4 = d4.reshape(64, 1)

plt.figure(figsize=(9, 9))
gs = gridspec.GridSpec(1, 8, height_ratios=[1],
                       width_ratios=[9, 1, 9, 1, 9, 1, 9, 1])
for i in range(4):
    plt.subplot(gs[2 * i])
    plt.imshow(eval("d" + str(i + 1)), aspect=1,
               interpolation='nearest', cmap=plt.cm.bone_r)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.title("image {}".format(i + 1))
    plt.subplot(gs[2 * i + 1])
    plt.imshow(eval("v" + str(i + 1)), aspect=0.25,
               interpolation='nearest', cmap=plt.cm.bone_r)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.title("vector {}".format(i + 1))
plt.tight_layout()
plt.show()
```



"0" 이미지와 "0" 이미지, 또는 "1" 이미지와 "1" 이미지의 내적값은 다음과 같다.

In [11]:

```
(v1.T @ v2)[0][0], (v3.T @ v4)[0][0]
```

Out[11]:

(3064.0, 3661.0)

상대적으로 "0" 이미지와 "1" 이미지, 또는 "1" 이미지와 "0" 이미지의 내적값은 작다.

In [12]:

```
(v1.T @ v3)[0][0], (v1.T @ v4)[0][0], (v2.T @ v3)[0][0], (v2.T @ v4)[0][0]
```

Out[12]:

(1866.0, 1883.0, 2421.0, 2479.0)

연습 문제 2.2.3

다음 코드를 실행하면 MNIST 숫자 이미지 전체 데이터를 모두 벡터로 변환하여 하나의 넘파이 행렬 `X` 를 만든다. 이 행렬을 이용하여 다음 문제를 풀어라.

```
from sklearn.datasets import load_digits
X = load_digits().data
```

(1) 내적을 이용하여 첫 번째 이미지와 10번째 이미지의 유사도를 구하라.

(2) 내적을 이용하여 모든 이미지의 조합에 대해 유사도를 구하라. 어떻게 구현하는 것이 효율적일까? (힌트 : 이 문제는 뒤에서 배울 행렬과 행렬의 곱셈을 이용한다.)

선형회귀 모형

선형회귀 모형(linear regression model)이란 독립변수 x에서 종속변수 y를 예측하는 방법의 하나로 독립변수 벡터 x와 가중치 벡터 w와의 가중합으로 y에 대한 예측값 $\hat{y}$를 계산하는 수식을 말한다.

$$\hat{y} = w_1 x_1 + \cdots + w_N x_N$$

이 수식에서 기호 ^ 는 "캐럿(caret)"이라는 기호이다. $\hat{y}$는 "와이 햇(y hat)"이라고 읽는다.

이 수식은 다음처럼 벡터의 내적으로 나타낼 수 있다.

$$\hat{y} = w^T x$$

선형회귀 모형은 가장 단순하면서도 가장 널리 쓰이는 예측 모형이다.

예를 들어 어떤 아파트 단지의 아파트 가격을 조사하였더니 아파트 가격은 (1)면적, (2)층수, (3)한강이 보이는지의 여부, 즉 이 세 가지 특징에 의해 달라진다는 사실을 알게 되었다. 이때 이 단지 내의 아파트 가격을 예측하는 예측 모형을 다음과 같이 만들 수 있다.

- 면적(m^2)을 입력 데이터 x_1라고 한다.
- 층수를 입력 데이터 x_2라고 한다
- 한강이 보이는지의 여부를 입력 데이터 x_3라고 하며 한강이 보이면 $x_3 = 1$, 보이지 않으면 $x_3 = 0$이라고 한다.
- 출력 데이터 $\hat{y}$는 해당 아파트의 예측 가격이다.

위와 같이 입력 데이터와 출력 데이터를 정의하고 회귀분석을 한 결과, 아파트값이 다음과 같은 선형회귀 모형으로 나타난다고 가정하자. 이러한 모형을 실제로 찾는 방법은 나중에 회귀분석 파트에서 공부하게 된다.

$$\hat{y} = 500 x_1 + 200 x_2 + 1000 x_3$$

이 모형은 다음과 같이 해석할 수 있다.

- 면적이 1m^2 증가할수록 가격은 500만 원이 증가한다.
- 층수가 1층 높아질수록 가격은 200만 원이 증가한다.
- 한강이 보이는 집은 1,000만 원의 웃돈(프리미엄)이 존재한다.

위 식은 다음과 같이 벡터의 내적으로 고쳐 쓸 수 있다.

$$\hat{y} = \begin{bmatrix} 500 & 200 & 1000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = w^T x$$

즉, 위 선형회귀 모형은 다음 가중치 벡터로 대표된다.

$$w^T = \begin{bmatrix} 500 & 200 & 1000 \end{bmatrix}$$

인공신경망(artificial neural network)에서는 선형회귀 모형을 다음과 같은 그림으로 표현한다. 데이터는 노드(node) 혹은 뉴런(neuron)이라는 동그라미로 표시하고 곱셈은 선분(line)위에 곱할 숫자를 써서 나타낸다. 덧셈은 여러 개의 선분이 만나는 것으로 표시한다.

그림 2.2.1 인공신경망으로 표현한 선형회귀 모형

선형회귀 모형의 단점

선형회귀 모형은 비선형적인 현실 세계의 데이터를 잘 예측하지 못할 수 있다는 단점이 있다. 예를 들어 집값은 면적에 단순 비례하지 않는다. 소형 면적의 집과 대형 면적의 집은 단위 면적당 집값의 증가율이 다를 수 있다. 또한 저층이 보통 고층보다 집값이 싸지만 층수가 올라갈수록 정확히 층수에 비례하여 가격이 증가하지도 않는다.

이러한 현실 세계의 데이터와 선형회귀 모형의 괴리를 줄이기 위해 선형회귀 모형이 아닌 완전히 다른 모형을 쓰기보다는 선형회귀 모형을 기반으로 여러 기법을 사용해 수정한 모형을 사용하는 것이 일반적이다. 이러한 수정 선형회귀 모형에 대해서는 나중에 공부하게 된다.

제곱합

데이터의 분산(variance)이나 표준 편차(standard deviation) 등을 구하는 경우에는 각각의 데이터를 제곱한 뒤 이 값을 모두 더한 **제곱합(sum of squares)**을 계산해야 한다. 이 경우에도 벡터의 내적을 사용하여 $x^T x$로 쓸 수 있다.

$$x^T x = \begin{bmatrix} x_1 & x_2 & \cdots & x_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \sum_{i=1}^{N} x_i^2$$

행렬과 행렬의 곱셈

벡터의 곱셈을 정의한 후에는 이를 이용하여 행렬의 곱셈을 정의할 수 있다. 행렬과 행렬을 곱하면 행렬이 된다. 방법은 다음과 같다.

A 행렬과 B 행렬을 곱한 결과가 C 행렬이 된다고 하자. C의 i번째 행, j번째 열의 원소 c_{ij}의 값은 A 행렬의 i번째 행 벡터 a_i^T와 B 행렬의 j번째 열 벡터 b_j의 곱이다.

$$C = AB \;\rightarrow\; c_{ij} = a_i^T b_j$$

이 정의가 성립하려면 앞의 행렬 A의 열의 수가 뒤의 행렬 B의 행의 수와 일치해야만 한다.

$$A \in \mathbf{R}^{N\times L},\; B \in \mathbf{R}^{L\times M} \;\rightarrow\; AB \in \mathbf{R}^{N\times M}$$

다음은 4×3 행렬과 3×2을 곱하여 4×2을 계산하는 예다.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} (a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}) & (a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}) \\ (a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}) & (a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}) \\ (a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}) & (a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}) \\ (a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31}) & (a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32}) \end{bmatrix}$$

다음은 실제 행렬을 사용한 곱셈의 예다.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$C = AB = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

넘파이를 이용하여 행렬의 곱을 구할 때도 `@` 연산자 또는 `dot()` 명령을 사용한다.

In [13]:

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

Out[13]:

```
array([[22, 28],
       [49, 64]])
```

연습 문제 2.2.4

(1) A와 B가 위와 같을 때 AB를 연습장에 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다.

(2) 순서를 바꾸어 BA를 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다. BA가 AB와 같은가?

(3) A, B가 다음과 같을 때, AB, BA를 (계산이 가능하다면) 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다. AB, BA 모두 계산 가능한가?

$$A = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 7 \\ 5 & 8 \\ 6 & 9 \end{bmatrix}$$

(4) A, B가 다음과 같을 때, AB, BA를 (계산이 가능하다면) 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다. AB, BA 모두 계산 가능한가? BA의 결과가 AB와 같은가?

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

(5) A가 다음과 같을 때, AA^T와 A^TA를 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다. AA^T와 A^TA의 크기는 어떠한가? 항상 정방행렬이 되는가?

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(6) x가 다음과 같을 때, x^Tx와 xx^T를 손으로 계산하고 넘파이의 계산 결과와 맞는지 확인한다. x^Tx와 xx^T의 크기는 어떠한가? 어떤 것이 스칼라이고 어떤 것이 정방행렬인가?

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

인공 신경망은 내부적으로 다음과 같이 여러 개의 선형회귀 모형을 사용한다. 이 구조는 행렬과 벡터의 곱으로 나타낼 수 있다.

그림 2.2.2 인공신경망의 기본 구조

위 그림을 행렬식으로 표현하면 다음과 같다.

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\hat{y} = Wx$$

교환 법칙과 분배 법칙

행렬의 곱셈은 곱하는 행렬의 순서를 바꾸는 교환 법칙이 성립하지 않는다. 그러나 덧셈에 대한 분배 법칙은 성립한다.

$$AB \neq BA$$

$$A(B + C) = AB + AC$$

$$(A + B)C = AC + BC$$

A, B, C가 다음과 같을 때 위 법칙을 넘파이로 살펴보자.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix}$$

In [14]:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.array([[9, 8], [7, 6]])
```

AB 와 BA의 값은 다음처럼 다른 값이 나오므로 교환법칙이 성립하지 않음을 알 수 있다.

In [15]:

```
A @ B
```

Out[15]:

```
array([[19, 22],
       [43, 50]])
```

In [16]:

```
B @ A
```

Out[16]:

```
array([[23, 34],
       [31, 46]])
```

분배법칙은 다음과 같이 성립한다.

In [17]:

```
A @ (B + C)
```

Out[17]:

```
array([[42, 42],
       [98, 98]])
```

In [18]:

```
A @ B + A @ C
```

Out[18]:

```
array([[42, 42],
       [98, 98]])
```

In [19]:

```
(A + B) @ C
```

Out[19]:

```
array([[110,  96],
       [174, 152]])
```

In [20]:

```
A @ C + B @ C
```

Out[20]:

```
array([[110,  96],
       [174, 152]])
```

전치 연산도 마찬가지로 덧셈/뺄셈에 대해 분배 법칙이 성립한다.

$$(A+B)^T = A^T + B^T$$

전치 연산과 곱셈의 경우에는 분배 법칙이 성립하기는 하지만 전치 연산이 분배되면서 곱셈의 순서가 바뀐다.

$$(AB)^T = B^T A^T$$

$$(ABC)^T = C^T B^T A^T$$

마찬가지로 넘파이로 위 법칙이 성립하는지 살펴보자.

In [21]:

```
(A + B).T
```

Out[21]:

```
array([[ 6, 10],
       [ 8, 12]])
```

In [22]:

```
A.T + B.T
```

Out[22]:

```
array([[ 6, 10],
       [ 8, 12]])
```

In [23]:

```
(A @ B).T
```

Out[23]:

```
array([[19, 43],
       [22, 50]])
```

In [24]:

```
B.T @ A.T
```

Out[24]:

```
array([[19, 43],
       [22, 50]])
```

연습 문제 2.2.5

(1) 길이가 같은 일벡터 $1_N \in \mathbf{R}^N$와 행벡터 $x \in \mathbf{R}^N$의 곱은 행벡터 x를 반복하여 가지는 행렬과 같음을 보여라.

$$\mathbf{1}_N x^T = \begin{bmatrix} x^T \\ x^T \\ \vdots \\ x^T \end{bmatrix}$$

(2) 행렬 $X (X \in \mathbf{R}^{N \times M})$가 있을 때, 이 행렬의 각 열의 평균으로 이루어진 벡터 $\bar{x}\,(\bar{x} \in \mathbf{R}^M)$가 다음과 같음을 보여라.

$$\bar{x} = \frac{1}{N} X^T \mathbf{1}_N$$

(3) 행렬 $\bar{X}\,(\bar{X} \in \mathbf{R}^{N \times M})$는 동일한 벡터 $\bar{x}^T$를 N개 누적하여 만든 행렬이다. 즉 각 열의 모든 값이 그 열의 평균으로 이루어진 행렬이다.

$$\bar{X} = \begin{bmatrix} \bar{x}^T \\ \bar{x}^T \\ \vdots \\ \bar{x}^T \end{bmatrix}$$

이때 $\bar{X}$가 다음과 같음을 보여라.

$$\bar{X} = \frac{1}{N} \mathbf{1}_N \mathbf{1}_N^T X$$

(4) 다음 코드를 실행하면 붓꽃 전체 데이터를 모두 벡터로 변환하여 하나의 넘파이 행렬 X 를 만든다.

```
from sklearn.datasets import load_iris
X = load_iris().data
```

이 데이터로 행렬 $\bar{X}$의 값을 계산하라. 이 행렬은 첫 번째 열의 값이 모두 같은 값으로 붓꽃의 꽃받침의 길이(sepal length)의 평균이고 두 번째 열의 값이 모두 같은 값으로 붓꽃의 꽃받침의 폭(sepal width)의 평균, 이런 식으로 계산된 행렬이다.

곱셈의 연결

연속된 행렬의 곱셈은 계산 순서를 임의의 순서로 해도 상관없다.

$$ABC = (AB)C = A(BC)$$

$$ABCD = ((AB)C)D = (AB)(CD) = A(BCD) = A(BC)D$$

연습 문제 2.2.6

다음 행렬의 곱셈을 순서를 바꾸어 두 가지 방법으로 해본다.

$$\begin{bmatrix}1 & 2\end{bmatrix}\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}5\\6\end{bmatrix}$$

항등행렬의 곱셈

어떤 행렬이든 항등행렬을 곱하면 그 행렬의 값이 변하지 않는다.

$$AI = IA = A$$

넘파이로 다음과 같이 확인한다.

In [25]:

```
A = np.array([[1, 2], [3, 4]])
I = np.eye(2)
```

In [26]:

```
A @ I
```

Out[26]:

```
array([[1., 2.],
       [3., 4.]])
```

In [27]:

```
I @ A
```

Out[27]:

```
array([[1., 2.],
       [3., 4.]])
```

행렬과 벡터의 곱

그럼 이러한 행렬의 곱셈은 데이터 분석에서 어떤 경우에 사용될까? 행렬의 곱셈 중 가장 널리 쓰이는 것은 다음과 같은 형태의 행렬 M과 벡터 v의 곱이다.

$$Mv$$

벡터와 행렬의 크기를 직사각형으로 표시하면 다음과 같다.

$$M \qquad v = Mv$$

행렬과 벡터의 곱을 사용하는 몇가지 예를 살펴보자.

열 벡터의 선형조합

행렬 X와 벡터 w의 곱은 행렬 X를 이루는 열벡터 $c_1, c_2, \ldots, c_M$을 뒤에 곱해지는 벡터 w의 각 성분 $w_1, w_2, \ldots, w_M$으로 선형조합(linear combination)을 한 결과 벡터와 같다.

$$Xw = \begin{bmatrix} c_1 & c_2 & \cdots & c_M \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} = w_1 c_1 + w_2 c_2 + \cdots + w_M c_M$$

벡터의 크기를 직사각형으로 표시하면 다음과 같다.

$$\begin{bmatrix} c_1 & c_2 & \cdots & c_M \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} = w_1 \, c_1 \quad + \quad w_2 \, c_2 \quad + \quad \cdots \quad + \quad w_M \, c_M$$

연습 문제 2.2.7

다음 행렬 X와 벡터 w에 대해 곱 Xw가 열벡터 c_1, c_2, c_3의 선형조합 $w_1 c_1 + w_2 c_2 + w_3 c_3$가 됨을 실제 계산으로 증명하라.

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad w = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

연습 문제 2.2.8

벡터 v_1, v_2, v_3로 이루어진 행렬 V와 벡터 λ에 대해 다음 식이 성립함을 증명하라. 이 식에서 λ_1는 스칼라이다.

$$V\lambda = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ 0 \\ 0 \end{bmatrix} = \lambda_1 v_1$$

벡터의 선형조합은 다양한 분야에 응용된다. 예를 들어 두 이미지 벡터의 선형조합은 두 이미지를 섞어놓은 모핑(morphing) 효과를 얻는 데 사용할 수 있다.

In [28]:

```
from sklearn.datasets import fetch_olivetti_faces

faces = fetch_olivetti_faces()

f, ax = plt.subplots(1, 3)

ax[0].imshow(faces.images[6], cmap=plt.cm.bone)
ax[0].grid(False)
ax[0].set_xticks([])
ax[0].set_yticks([])
ax[0].set_title("image 1: $x_1$")

ax[1].imshow(faces.images[10], cmap=plt.cm.bone)
ax[1].grid(False)
ax[1].set_xticks([])
ax[1].set_yticks([])
ax[1].set_title("image 2: $x_2$")

new_face = 0.7 * faces.images[6] + 0.3 * faces.images[10]
ax[2].imshow(new_face, cmap=plt.cm.bone)
ax[2].grid(False)
ax[2].set_xticks([])
ax[2].set_yticks([])
ax[2].set_title("image 3: $0.7x_1 + 0.3x_2$")

plt.show()
```

여러 개의 벡터에 대한 가중합 동시 계산

벡터 하나의 가중합은 w^Tx 또는 x^Tw로 표시할 수 있다는 것을 배웠다. 그런데 만약 이렇게 w 가중치를 사용한 가중합을 하나의 벡터 x가 아니라 여러 벡터 $x_1, \cdots, x_M$ 개에 대해서 모두 계산해야 한다면 어떻게 해야 할까? 예를 들어 위와 같이 선형 회귀 모형을 사용하여 여러 데이터 $x_1, x_2, x_3, \cdots, x_N$ 개의 데이터 모두에 대해 예측값 $y_1, y_2, y_3, \cdots, y_N$을 한꺼번에 계산하고 싶다면 다음처럼 데이터 행렬 X를 사용하여 $\hat{y} = Xw$라는 수식으로 간단하게 표시할 수 있다.

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_M \end{bmatrix} = \begin{bmatrix} w_1 x_{1,1} + w_2 x_{1,2} + \cdots + w_N x_{1,N} \\ w_1 x_{2,1} + w_2 x_{2,2} + \cdots + w_N x_{2,N} \\ \vdots \\ w_1 x_{M,1} + w_2 x_{M,2} + \cdots + w_N x_{M,N} \end{bmatrix}$$

$$= \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M,1} & x_{M,2} & \cdots & x_{M,N} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$$

$$= \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_M^T \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$$

$$= Xw$$

즉.

$$\hat{y} = Xw$$

연습 문제 2.2.9

x_1, x_2가 다음과 같을 때,

$$x_1 = \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \end{bmatrix} \quad x_2 = \begin{bmatrix} x_{12} \\ x_{22} \\ x_{32} \end{bmatrix}$$

다음 등식이 성립함을 보인다.

$$Xw = \begin{bmatrix} x_1^T \\ x_2^T \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1^T w \\ x_2^T w \end{bmatrix}$$

잔차

선형 회귀분석(linear regression)을 한 결과는 가중치 벡터 w라는 형태로 나타나고 예측치는 이 가중치 벡터를 사용한 독립변수 데이터 레코드 즉, 벡터 x_i의 가중합 $w^T x_i$이 된다고 했다. 예측치와 실젯값(target) y_i의 차이를 **오차(error)** 혹은 **잔차(residual)** e_i라고 한다. 이러한 잔찻값을 모든 독립변수 벡터에 대해 구하면 잔차 벡터 e가 된다.

$$e_i = y_i - \hat{y}_i = y_i - w^T x_i$$

잔차 벡터는 다음처럼 $y - Xw$로 간단하게 표기할 수 있다.

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}$$

$$= \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} - \begin{bmatrix} x_1^T w \\ x_2^T w \\ \vdots \\ x_M^T w \end{bmatrix}$$

$$= y - Xw$$

$$e = y - Xw$$

잔차 제곱합

잔차의 크기는 잔차 벡터의 각 원소를 제곱한 후 더한 **잔차 제곱합(RSS: Residual Sum of Squares)**을 이용하여 구한다. 이 값은 e^Te로 간단하게 쓸 수 있으며 그 값은 다음처럼 계산한다.

$$\sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} (y_i - w^T x_i)^2 = e^T e = (y - Xw)^T (y - Xw)$$

연습 문제 2.2.10

분배 법칙을 사용하여 위 식 $(y - Xw)^T(y - Xw)$을 풀어쓰면 다음과 같아짐을 보여라.

$$(y - Xw)^T(y - Xw) = y^Ty - w^TX^Ty - y^TXw + w^TX^TXw$$

이차형식

위의 연습 문제에서 마지막 항은 w^TX^TXw라는 형태다. 이 식에서 X^TX는 정방행렬이 되므로 이 정방행렬을 A라고 이름 붙이면 마지막 항은 w^TAw와 같은 형태가 된다.

벡터의 **이차형식(Quadratic Form)**이란 이처럼 어떤 벡터와 정방행렬이 '행벡터 × 정방행렬 × 열벡터'의 형식으로 되어 있는 것을 말한다.

이 수식을 풀면 $i = 1, \ldots, N, j = 1, \ldots, N$에 대해 가능한 모든 i, j 쌍의 조합을 구한 다음 i, j에 해당하는 원소 x_i, x_j를 가중치 $a_{i,j}$와 같이 곱한 값 $a_{i,j}x_ix_j$의 총합이 된다.

$$x^TAx = \begin{bmatrix} x_1 & x_2 & \cdots & x_N \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} a_{i,j}x_ix_j$$

연습 문제 2.2.11

다음 3차원 벡터와 행렬에 대해 이차형식을 쓰고 값을 계산하라.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

예를 들어 $x = [1, 2, 3]^T$ 이고 A가 다음과 같다면

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

넘파이 에서 벡터의 이차형식은 다음처럼 계산한다.

In [29]:

```
x = np.array([1, 2, 3])
x
```

Out[29]:

```
array([1, 2, 3])
```

In [30]:

```
A = np.arange(1, 10).reshape(3, 3)
A
```

Out[30]:

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

In [31]:

```
x.T @ A @ x
```

Out[31]:

연습 문제 2.2.12

다음 식이 성립함을 증명하라.

$$x^T Ax = \frac{1}{2} x^T (A + A^T) x$$

부분행렬

다음과 같은 2차원 정방행렬 A, B가 있다.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

이때 두 행렬의 곱 AB는 A, B의 **부분행렬(submatrix)**을 이용하여 여러 방법으로 계산할 수 있다.

(1) 우선 앞에 곱해지는 행렬을 행벡터로 나누어 계산해도 된다.

$$A = \begin{bmatrix} a_1^T \\ a_2^T \end{bmatrix}$$

즉,

$$a_1^T = \begin{bmatrix} a_{11} & a_{12} \end{bmatrix}, \quad a_2^T = \begin{bmatrix} a_{21} & a_{22} \end{bmatrix}$$

이면

$$AB = \begin{bmatrix} a_1^T \\ a_2^T \end{bmatrix} B = \begin{bmatrix} a_1^T B \\ a_2^T B \end{bmatrix}$$

(2) 아니면 뒤에 곱해지는 행렬을 열벡터로 나누어 계산해도 된다.

$$B = \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

즉,

$$b_1 = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}, \quad b_2 = \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix}$$

이면

$$AB = A \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} Ab_1 & Ab_2 \end{bmatrix}$$

(3) 앞에 곱해지는 행렬을 열벡터로, 뒤에 곱해지는 행렬을 행벡터로 나누어 스칼라처럼 계산해도 된다.

$$AB = \begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} b_1^T \\ b_2^T \end{bmatrix} = a_1 b_1^T + a_2 b_2^T$$

벡터의 크기를 직사각형으로 표시하면 다음과 같다.

$$AB = \begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} b_1^T \\ b_2^T \end{bmatrix} = a_1 \quad b_1^T \quad + b_1 \quad b_2^T$$

여기에서는 2차원 행렬의 예를 들었지만 일반적인 N차원 행렬에서도 이 관계는 성립한다.

연습 문제 2.2.13

행렬 V는 열벡터 v_i $(i = 1, \ldots, N)$로 이루어진 정방행렬이다. V와 크기가 같은 다른 정방행렬 A, Λ이 있을 때 다음 식이 성립한다.

$$AV = A\begin{bmatrix} v_1 \cdots v_N \end{bmatrix} = \begin{bmatrix} Av_1 \cdots Av_N \end{bmatrix}$$

$$V\Lambda = \begin{bmatrix} v_1 \cdots v_N \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{bmatrix} = \begin{bmatrix} \lambda_1 v_1 \cdots \lambda_N v_N \end{bmatrix}$$

$N = 3$인 경우에 위 식이 성립함을 보여라.

연습 문제 2.2.14

부분행렬 공식 (3)으로부터 A가 행벡터 a_i^T $(i = 1, \ldots, N)$로 이루어진 N차 정방행렬일 때

$$A = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_N^T \end{bmatrix}$$

다음 관계가 성립한다.

$$A^T A = \begin{bmatrix} a_1 & a_2 & \cdots & a_N \end{bmatrix} \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_N^T \end{bmatrix} = \sum_{i=1}^{N} a_i a_i^T$$

$N = 3$인 경우에 위 식이 성립함을 보여라.