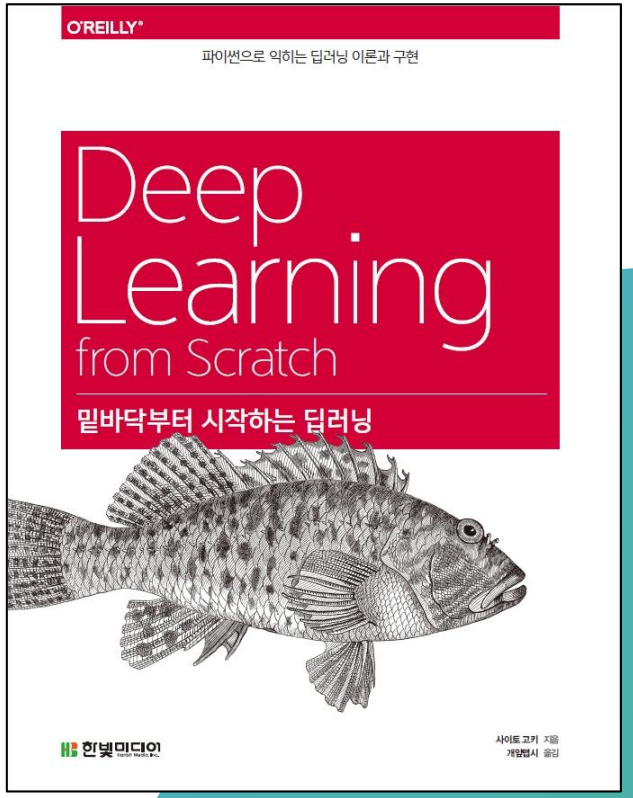
- CHAPTER 3 신경망
- CHAPTER 4 신경망 학습

밑바닥부터 시작하는 딥러닝



CHAPTER 3 신경망

신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기

Contents

SECTION 03 신경망

3.1.1 신경망의 예

신경망을 그림으로 나타내면 [그림 3 - 1]처럼 된다.
여기에서 가장 왼쪽 줄을 입력층 , 맨 오른쪽 줄을 출력층 , 중간 줄을 은닉층 이라고 함.
은닉층의 뉴런은 (입력층이나 출력층과 달리)사람 눈에는 보이지 않는다.
이 책에서는 입력층에서 출력층방향으로 차례로 0 층, 1 층, 2 층이라 하겠다(층 번호를 0 부터 시작하는 이유는 파이썬 배열의 인덱스도 0 부터 시작하여, 나중에 구현할 때 짝짓기 편하기 때문
[그림 3 - 1]에서는 0 층이 입력층, 1 층이 은닉층, 2 층이 출력층이 된다.

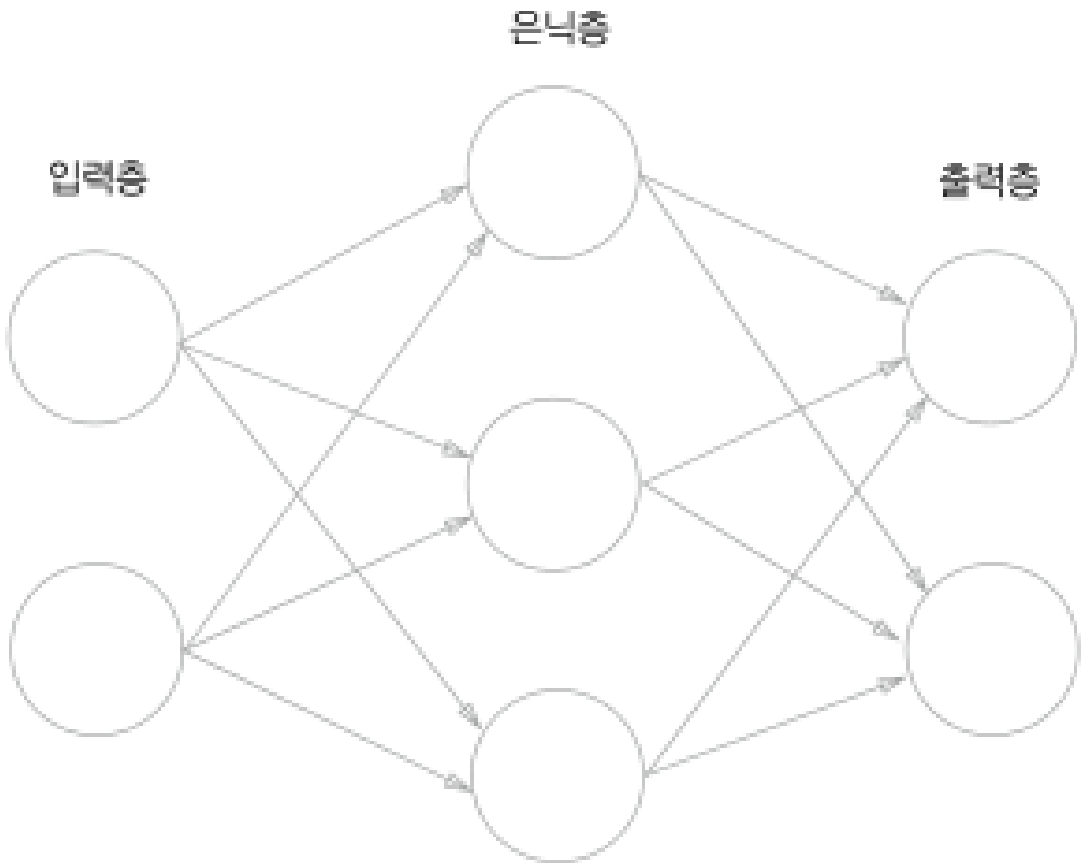
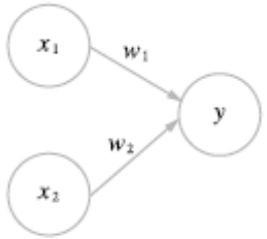


그림 3-1 신경망의 예

3.1.2 퍼셉트론 복습

h (x)라는 함수가 등장했는데, 이처럼 입력 신호의 총합을 출력 신호로 변환하는 함수를 일반적으로 활성화 함수 activation function 라 한다.
'활성화'라는 이름이 말해주듯 활성화 함수는입력 신호의 총합이 활성화를 일으키는지를 정하는 역할
[식 3 . 2]는 가중치가 곱해진 입력 신호의 총합을 계산하고, 그합을 활성화 함수에 입력해 결과를 내는 2 단계로 처리 된다. 그래서 이식은 다음 2단계로 나눌수 있습니다.

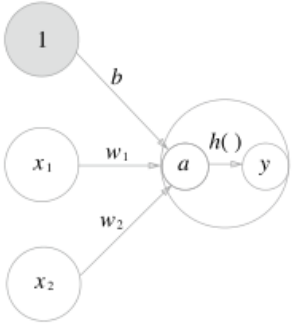
그림 3-2 퍼셉트론 복습



$$a = b + w_1x_1 + w_2x_2 \quad \text{[식 3.4}$$

$$y = h(a) \quad \text{[식 3.5}$$

[식 3 . 4]는 가중치가 달린 입력 신호와 편향의 총합을 계산하고, 이를 a 라 한다.
[식3 . 5]는 a 를 함수 h ()에 넣어 y 를 출력하는 흐름.
지금까지와 같이 뉴런을 큰 원(○)으로 그려보면 [식 3 . 4]와 [식 3 . 5]는 [그림 3 - 4]처럼 나타내 낼 수 있다



$$y = h(b + w_1x_1 + w_2x_2) \quad \text{[식 3.2]}$$

$$h(x) = \begin{cases} 0 \ (x \leq 0) \\ 1 \ (x > 0) \end{cases} \quad \text{[식 3.3]}$$

그림 3-4 활성화 함수의 처리 과정

3.1.3 활성화 함수의 등장

입력 신호의 총합을 출력 신호로 변환하는 함수를 일반적으로 활성화 함수activation function라 한다

[식 3.2]는 가중치가 곱해진 입력 신호의 총합을 계산하고, 그 합을 활성화 함수에 입력해 결과를 내는 2단계로 처리된다.

$$a = b + w_1 x_1 + w_2 x_2 \qquad [식\ 3.4]$$

$$y = h(a) \qquad [식\ 3.5]$$

그림 3-4 활성화 함수의 처리 과정

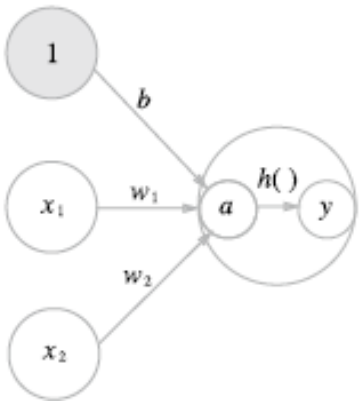


그림 **3-5** 왼쪽은 일반적인 뉴런, 오른쪽은 활성화 처리 과정을 명시한 뉴런(*a*는 입력 신호의 총합, *h*( )는 활성화 함수,*y*는 출력)

3.2.1 시그모이드 함수

시그모이드 함수sigmoid function를 나타낸 식

$$h(x) = \frac{1}{1 + \exp(-x)}$$ [식 3.6]

신경망에서는 활성화 함수로 시그모이드 함수를 이용하여 신호를 변환하고, 그 변환된 신호를다음 뉴런에 전달

3.2.2 계단 함수 구현하기

```
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)
```

```
>>> import numpy as np
>>> x = np.array([-1.0, 1.0, 2.0])
>>> x
array([-1.,  1.,  2.])
>>> y = x > 0
>>> y
array([False,  True,  True], dtype=bool)
```

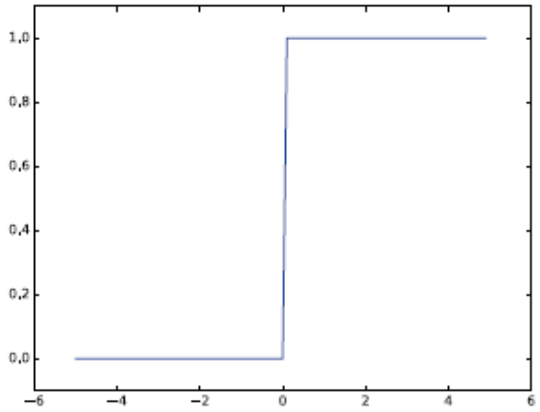
넘파이 배열에 부등호 연산을 수행하면 배열의 원소 각각에 부등호 연산을 수행한 bool 배열이 생성.
이 예에서는 배열 x의 원소 각각이 0보다 크면 True로, 0 이하면 False로 변환 한 새로운 배열 y가 생성

3.2.3 계단 함수의 그래프

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

def step_function(x):

    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1)  # y축의 범위 지정
plt.show()
```

np.arange(-5.0, 5.0, 0.1)은 -5.0에서 5.0 전까지 0.1 간격의 넘파이 배열을 생성.
[-5.0, -4.9, ..., 4.9]를 생성.step_function ( )은 인수로 받은 넘파이 배열의 원소 각각을 인수로 계단 함수 실행해, 그 결과를 다시 배열로 만들어 돌려준다

그림 3-6 계단 함수의 그래프



3.2.4 시그모이드 함수 구현하기

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

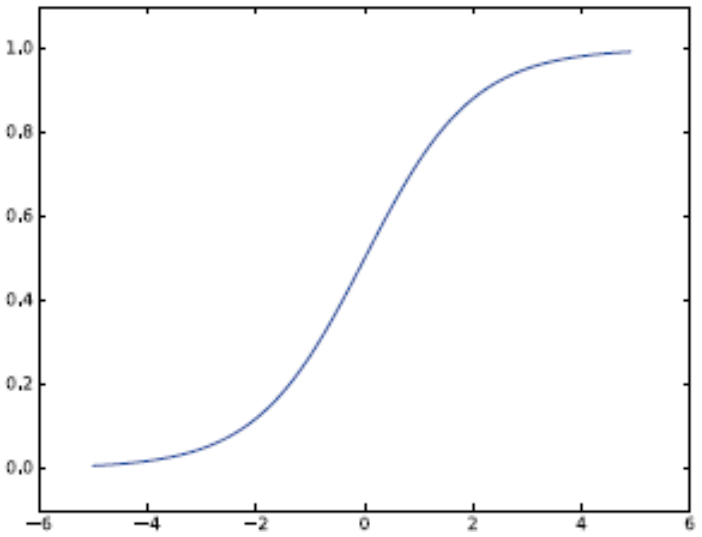
```
>>> x = np.array([-1.0, 1.0, 2.0])
>>> sigmoid(x)
array([ 0.26894142,  0.73105858,  0.88079708])

>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([ 2.,  3.,  4.])
>>> 1.0 / t
array([ 1.        ,  0.5       ,  0.33333333])
```

구현한 sigmoid 함수에서도 np.exp(-x)가 넘파이 배열을 반환하기 때문에 1 / (1 + np.exp(-x))도 넘파이 배열의 각 원소에 연산을 수행한 결과를 내어 준다

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # y축 범위 지정
plt.show()
```

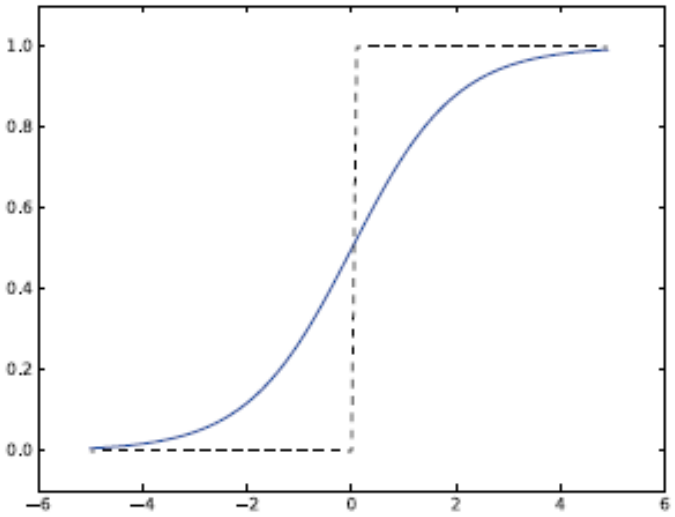
이 코드를 실행하면 [그림 3-7]의 그래프를 나타낸다

그림 3-7 시그모이드 함수의 그래프*



3.2.5 시그모이드 함수와 계단 함수 비교

그림 3-8 계단 함수(점선)와 시그모이드 함수(실선)



[그림 3-8]을 보고 가장 먼저 느껴지는 점은 '매끄러움'의 차이일 것.
시그모이드 함수는 부드러운 곡선이며 입력에 따라 출력이 연속적으로 변화.
한편, 계단 함수는 0을 경계로 출력이 갑자기 바뀐다.
시그모이드 함수의 이 매끈함이 신경망 학습에서 아주 중요한 역할.

3.2.6 비선형 함수

계단 함수와 시그모이드 함수의 공통점은 그 밖에도 있습니다. 중요한 공통점으로, 둘 모두는 비선형 함수.
시그모이드 함수는 곡선, 계단 함수는 계단처럼 구부러진 직선으로 나타나며, 동시에 비선형 함수로 분류.

활성화 함수를 설명할 때 비선형 함수와 선형 함수라는 용어가 자주 등장.
함수란 어떤 값을입력하면 그에 따른 값을 돌려주는 '변환기'.
이 변환기에 무언가 입력했을 때 출력이 입력의 상수 배 만큼 변하는 함수를 선형 함수라고 한다.

수식으로는 f (x) = ax + b 이고, 이때 a 와 b 는 상수이다.
선형 함수는 곧은 1 개의 직선이 된다.

한편, 비선형 함수는 문자 그대로 '선형이 아닌' 함수입니다. 즉, 직선 1 개로는 그릴 수 없는 함수를 말한다.

3.2.7 ReLU 함수

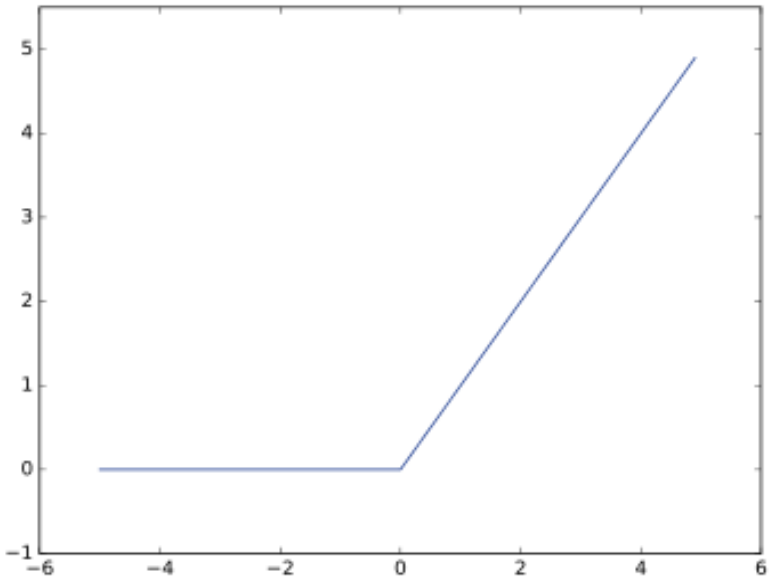


그림 3-9 ReLU 함수의 그래프

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$ [식 3.7]

수식으로는 [식 3 . 7]처럼 쓸 수 있다.

```
def relu(x):
    return np.maximum(0, x)
```

여기에서는 넘파이의 maximum 함수를 사용.
maximum 은 두 입력 중 큰 값을 선택해 반환하는 함수.
이번 장에서는 앞으로 시그모이드 함수를 활성화 함수로 사용했으나,
이 책 후반부는 주로ReLU 함수를 사용.

3.3.1 다차원 배열

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4])
>>> print(A)
[1 2 3 4]
>>> np.ndim(A)
1
>>> A.shape
(4,)
>>> A.shape[0]
4
```

이와 같이 배열의 차원 수는 np . ndim () 함수로 확인할 수 있다.
또, 배열의 형상은 인스턴스 변수인 shape 으로 알 수 있다.
이 예에서 A 는 1 차원 배열이고 원소 4 개로 구성되어. 한 가지, A . shape 이 튜플을 반환하는 것에 주의.
이는 1 차원 배열이라도 다차원 배열일 때와 통일된 형태로 결과를 반환하기 위함.

```
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> print(B)
[[1 2]
 [3 4]
 [5 6]]
>>> np.ndim(B)
2
>>> B.shape
(3, 2)
```

이때 처음 차원은 0 번째 차원, 다음 차원은 1 번째 차원에 대응
(파이썬의 인덱스는 0 부터 시작합니다). 2 차원 배열은 특히 행렬 matrix 이라고 부르고 [그림 3 - 10]과 같이 배열의 가로 방향을 행 row , 세로 방향을 열 column 이라고 한다

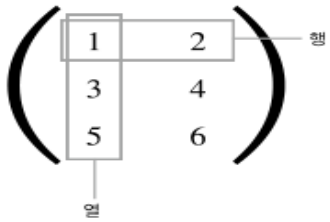


그림 3-10 2 차원 배열(행렬)의 행(가로)과 열(세로)

3.3.2 행렬의 곱

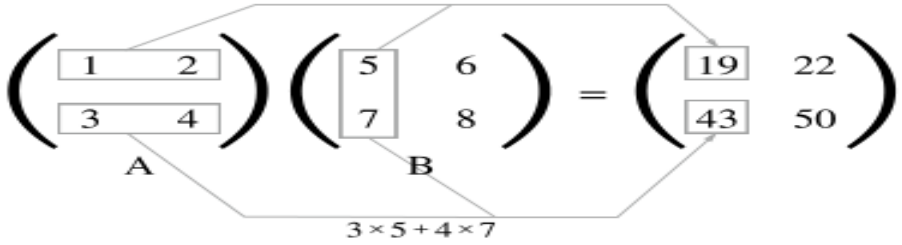


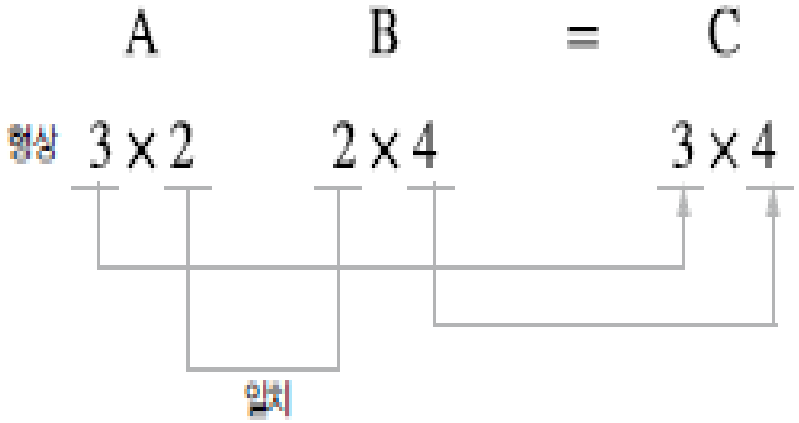
그림 3-11 행렬의 곱 계산 방법

```
>>> A = np.array([[1,2], [3,4]])
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

3.3.2 행렬의 곱

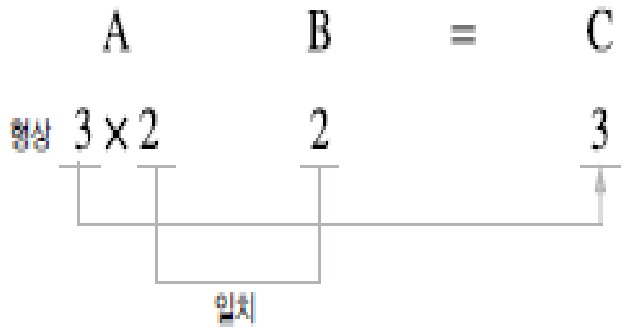
```
>>> A = np.array([[1,2,3], [4,5,6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> B.shape
(3, 2)
>>> np.dot(A, B)
array([[22, 28],
       [49, 64]])
```

그림 3-12 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시켜라.

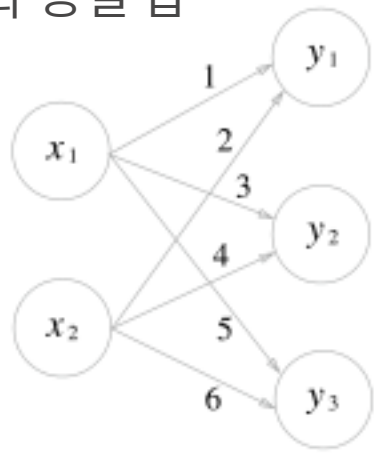



```
>>> C = np.array([[1,2], [3,4]])
>>> C.shape
(2, 2)
>>> A.shape
(2, 3)
>>> np.dot(A, C)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

그림 3-13 A가 2차원 행렬, B가 1차원 배열일 때도 대응하는 차원의 원소 수를 일치시켜라.



3.3.3 신경망에서의 행렬 곱



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

$$X \quad W \quad = \quad Y$$

$$2 \quad 2 \times 3 \quad 3$$

일치

그림 3-14 행렬의 곱으로 신경망의 계산을 수행.

이 구현에서도 X , W , Y 의 형상을 주의해서 보세요. 특히 X 와 W 의 대응하는 차원의 원소 수가 같아야 한다는 걸 잊지 말아야한다

```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

다차원 배열의 스칼라 곱을 구해주는 np . dot 함수를 사용하면 이처럼 단번에 결과 Y 를 계산할 수 있다.
Y 의 원소가 100 개든 1 , 000 개든 한 번의 연산으로 계산할 수 있다!

3.4 3층 신경망 구현하기

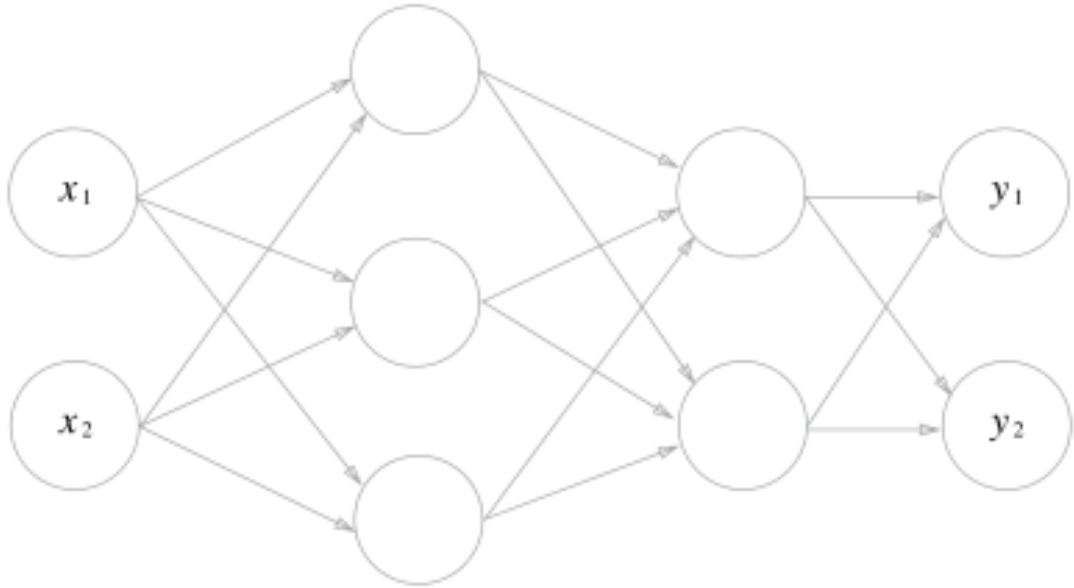


그림 3-15 3 층 신경망 : 입력층(0 층)은 2 개, 첫 번째 은닉층(1 층)은 3 개, 두 번째 은닉층(2 층)은 2 개, 출력층(3 층)은 2개의 뉴련으로 구성

3.4.1 표기법 설명

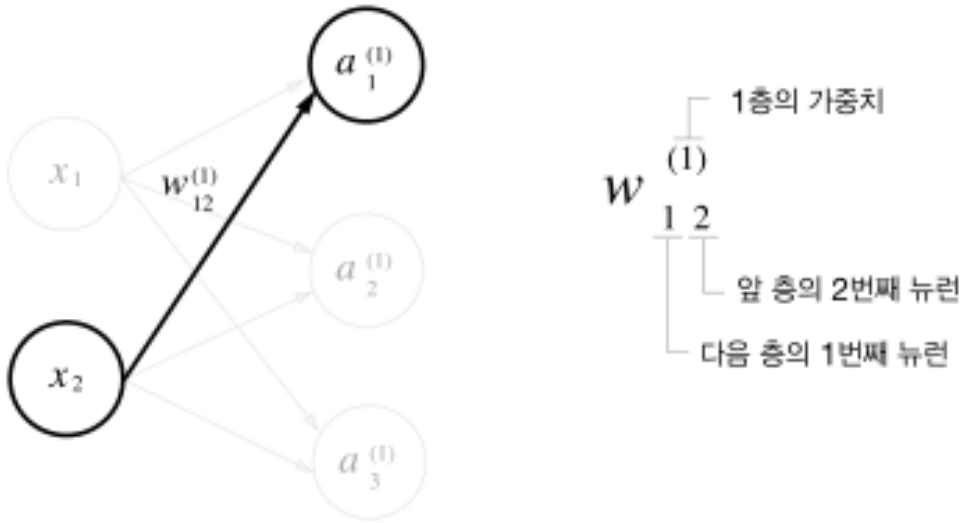


그림 3-16 중요한 표기

[그림 3 - 16]과 같이 가중치와 은닉층 뉴런의 오른쪽 위에는 ' (1) '이 붙어 있다.
이는 1 층의 가중치, 1 층의 뉴런 임을 뜻하는 번호.
또, 가중치의 오른쪽 아래의 두 숫자는 차례로 다음 층 뉴런과 앞 층 뉴런의 인덱스 번호

3.4.2 각 층의 신호 전달 구현하기

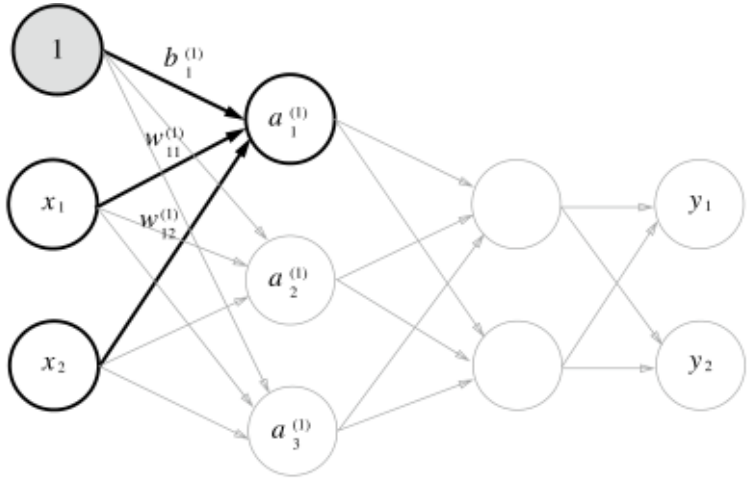


그림 3-17 입력층에서 1 층으로 신호 전달

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)} \quad [식 3.8]$$

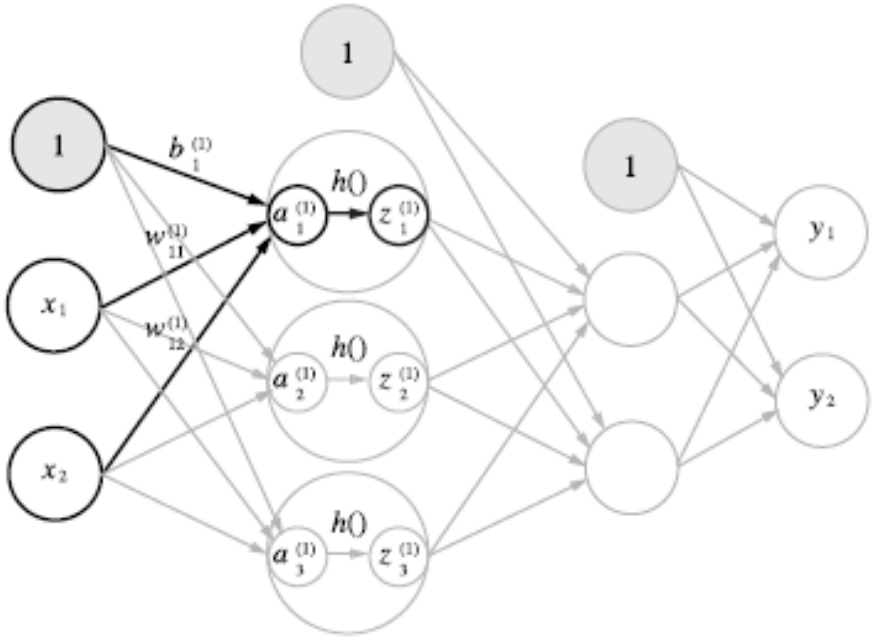
$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)} \quad [식 3.9]$$

$$\mathbf{A}^{(1)} = (a_1^{(1)} \; a_2^{(1)} \; a_3^{(1)}),\ \mathbf{X} = (x_1 \; x_2),\ \mathbf{B}^{(1)} = (b_1^{(1)} \; b_2^{(1)} \; b_3^{(1)})$$

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

print(W1.shape) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

A1 = np.dot(X, W1) + B1
```

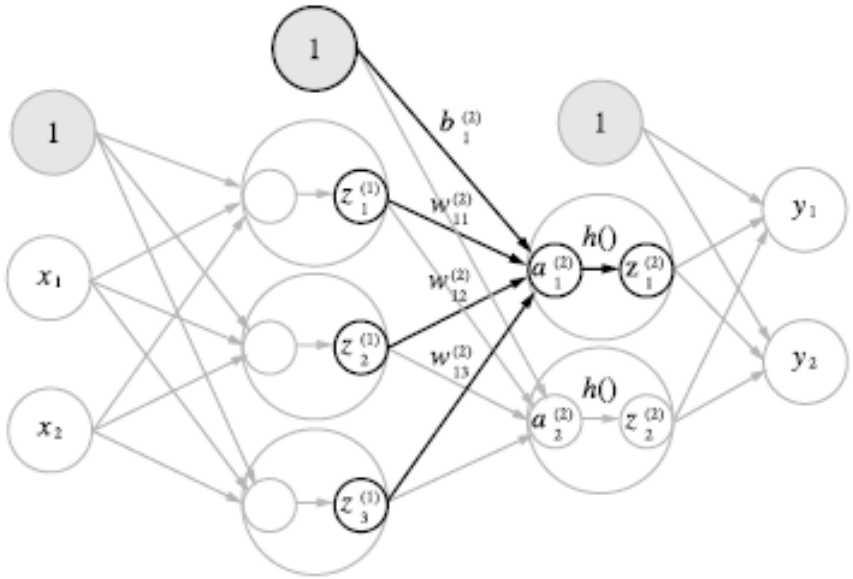
그림 3-18 입력층에서 1층으로의 신호 전달




```
Z1 = sigmoid(A1)

print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [0.57444252, 0.66818777, 0.75026011]
```

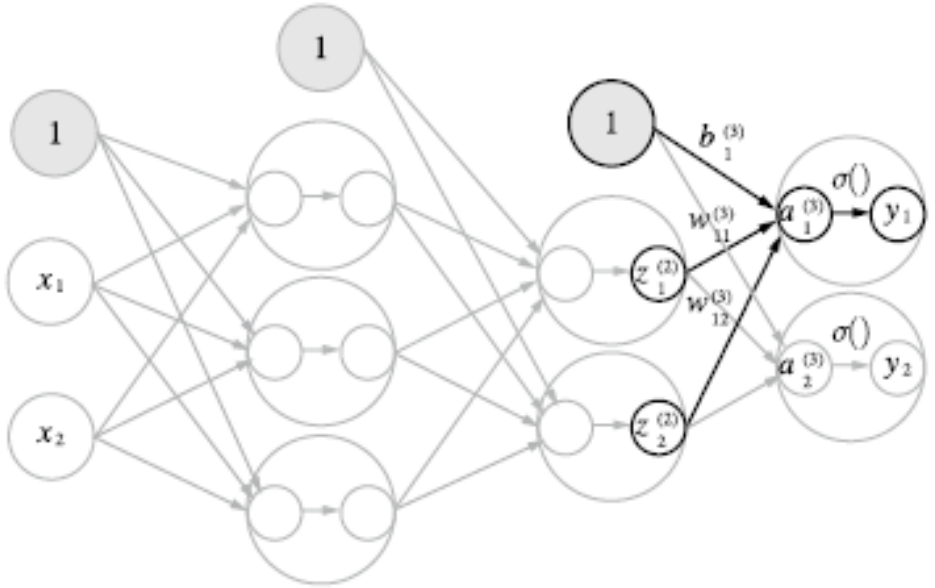
3.4.2 각 층의 신호 전달 구현하기

그림 3-19 1층에서 2층으로의 신호 전달



```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

print(Z1.shape) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

그림 3-20 2층에서 출력층으로의 신호 전달



```
def identity_function(x):
    return x

W3=np.array([[0.1,0.3],[0.2,0.4]])
B3=np.array([0.1,0.2])

A3=np.dot(Z2,W3)+B3
Y=identity_function(A3)

print('Y=',Y)
```

deep-learning-from-scratch/ch03/nn3layer.py

3.4.3 구현 정리

```
def init_network():
    network={}
    network['W1']=np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
    network['b1']=np.array([0.1,0.2,0.3])
    network['W2']=np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
    network['b2']=np.array([0.1,0.2])
    network['W3']=np.array([[0.1,0.3],[0.2,0.4]])
    network['b3']=np.array([0.1,0.2])

    return network

def forward(network,x):
    W1, W2, W3=network['W1'], network['W2'], network['W3']
    b1, b2, b3=network['b1'], network['b2'], network['b3']

    a1=np.dot(x,W1)+b1
    z1=sigmoid(a1)
    a2=np.dot(z1,W2)+b2
    z2=sigmoid(a2)
    a3=np.dot(z2,W3)+b3
    y=identity_function(a3)

    return y

network=init_network()
x=np.array([1.0,0.5])
y=forward(network,x)
print('y=',y)
```

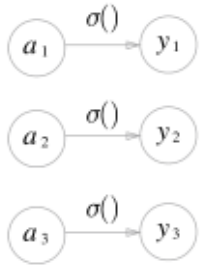
deep-learning-from-scratch/ch03/nn3layer.py

3.5 출력층 설계하기

일반적으로 회귀에는 항등함수, 분류에는 소프트맥스/시그모이드 함수

3.5.1 항등 함수와 소프트맥스 함수 구현하기

그림 3-21 항등 함수



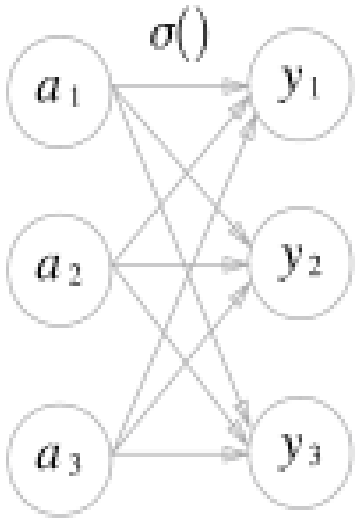
한편, 분류에서 사용하는 **소프트맥스 함수**softmax function의 식은 다음과 같습니다.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} \qquad [식\ 3.10]$$

그림 3-21 항등 함수

exp (x)는 e x 을 뜻하는 지수 함수 exponential function 이다
(e 는 자연상수). N 은 출력층의 뉴런 수,y k 는 그중 k 번째 출력임을 뜻한다.
[식 3 . 10]과 같이 소프트맥스 함수의 분자는 입력 신호 a k의 지수 함수, 분모는 모든 입력 신호의 지수 함수의 합으로 구성.

그림 3-22 소프트맥스 함수




```
>>> a = np.array([0.3, 2.9, 4.0])
>>>
>>> exp_a = np.exp(a) # 지수 함수
>>> print(exp_a)
[ 1.34985881 18.17414537 54.59815003]
>>>
>>> sum_exp_a = np.sum(exp_a) # 지수 함수의 합
>>> print(sum_exp_a)
74.1221542102
>>>
>>> y = exp_a / sum_exp_a
>>> print(y)
[ 0.01821127 0.24519181 0.73659691]
```

이 구현은 [식 3 . 10]의 소프트맥스 함수를 그대로 파이썬으로 표현

ch03/softmax.py

```
import numpy as np

def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a=np.sum(exp_a)
    y=exp_a/sum_exp_a

    return y

a=np.array([0.3, 2.9, 4.0])
y=softmax(a)
print(y)
```

3.5.2 소프트맥스 함수 구현 시 주의점

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} = \frac{C\exp(a_k)}{C\sum_{i=1}^{n}\exp(a_i)}$$
$$= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n}\exp(a_i + \log C)}$$
$$= \frac{\exp(a_k + C')}{\sum_{i=1}^{n}\exp(a_i + C')}$$

식 3 . 11]의 전개 과정을 살펴보자.
첫 번째 변형에서는 C 라는 임의의 정수를 분자와 분모 양쪽에 곱했다(양쪽에 같은 수를 곱했으니 결국 똑같은 계산).
그다음으로 C 를 지수함수 exp () 안으로 옮겨 logC 로 만듭니다. 마지막으로 logC 를 C '라는 새로운 기호로 바꾼다.

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # 소프트맥스 함수의 계산
array([ nan,  nan,  nan])        # 제대로 계산되지 않는다.
>>>
>>> c = np.max(a)                # c = 1010 (최댓값)
>>> a - c
array([  0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

이 예에서 보는 것처럼 아무런 조치 없이 그냥 계산하면 nan 이 출력된다 (nan 은 not a Number 의 약자).
하지만 입력 신호 중 최댓값(이 예에서는 c)을 빼주면 올바르게 계산할 수 있다.

ch03/softmax.py

```
import numpy as np

#def softmax(a):
#    exp_a = np.exp(a)
#    sum_exp_a=np.sum(exp_a)
#    y=exp_a/sum_exp_a
#
#    return y

def softmax(a):
    c=np.max(a)
    exp_a=np.exp(a-c) # prevent overflow
    sum_exp_a=np.sum(exp_a)
    y=exp_a/sum_exp_a

    return y

a=np.array([0.3, 2.9, 4.0])
y=softmax(a)
print(y)
a=np.array([1010, 1000, 990])
y=softmax(a)
print(y)
```

3.5.3 소프트맥스 함수의 특징

```
>>> a = np.array([0.3, 2.9, 4.0])
>>> y = softmax(a)
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
>>> np.sum(y)
1.0
```

보는 바와 같이 소프트맥스 함수의 출력은 0 에서 1 . 0 사이의 실수입니다.
또, 소프트맥스 함수출력의 총합은 1 이다.
출력 총합이 1 이 된다는 점은 소프트맥스 함수의 중요한 성질.

가령 앞의 예에서 y [0]의 확률은 0 . 018 (1 . 8 %), y [1]의 확률은 0 . 245 (24 . 5 %), y [2]의 확률은 0 . 737 (73 . 7 %)로 해석
그리고 이 결과 확률들로부터 " 2 번째 원소의 확률이가장 높으니, 답은 2 번째 클래스다"라고 할 수 있다.

혹은 " 74 %의 확률로 2 번째 클래스,25 %의 확률로 1 번째 클래스, 1 %의 확률로 0 번째 클래스다"와 같이 확률적인 결론도 낼 수있다.
소프트맥스 함수를 이용함으로써 문제를 확률적(통계적)으로 대응할 수 있다.

여기서 주의점으로, 소프트맥스 함수를 적용해도 각 원소의 대소 관계는 변하지 않는다.
이는 지수 함수 y = exp (x)가 단조 증가 함수이기 때문.

- 실제로 앞의 예에서는 a 의 원소들사이의 대소 관계가 y 의 원소들 사이의 대소 관계로 그대로 이어진다.
- 예를 들어 a 에서 가장큰 원소는 2 번째 원소이고, y 에서 가장 큰 원소도 2 번째 원소.

3.5.4 출력층의 뉴런 수 정하기

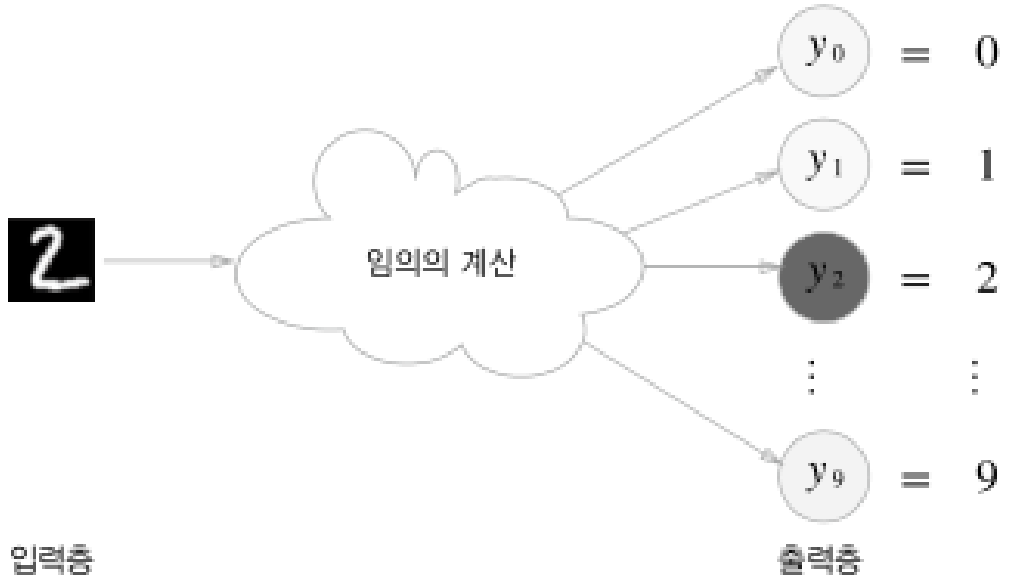


그림 3-23 출력층의 뉴런은 각 숫자에 대응한다

그림 3 - 23]의 예에서 출력층 뉴런은 위에서부터 차례로 숫자 0 , 1 , ..., 9 에 대응하며, 뉴런의회색 농도가 해당 뉴런의 출력 값의 크기를 의미.
이 예에서는 색이 가장 짙은 y 2 뉴런이가장 큰 값을 출력하는 것.
그래서 이 신경망이 선택한 클래스는 y 2 , 즉 입력 이미지를 숫자 ' 2 '로 판단했음을 의미.

3.6.1 MNIST 데이터셋

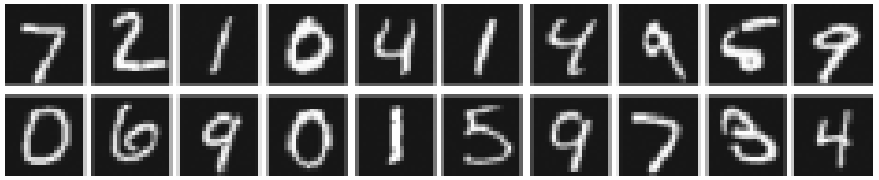


그림 3-24 MNIST 이미지 데이터셋의 예

코드를 보면 가장 먼저 부모 디렉터리의 파일을 가져올 수 있도록 설정하고 dataset/mnist.py의 load_mnist 함수를 임포트한다.
그런 다음 load_mnist 함수로 MNIST 데이터셋을 읽는다. load_mnist가 MNIST 데이터를 받아와야 하니 최초 실행 시에는 인터넷에 연결된 상태여야 한다.
두 번째부터는 로컬에 저장된 파일(pickle 파일)을 읽기 때문에 빠르게 끝난다.

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from dataset.mnist import load_mnist

# 처음 한 번은 몇 분 정도 걸립니다.
(x_train, t_train), (x_test, t_test) = \
    load_mnist(flatten=True, normalize=False)

# 각 데이터의 형상 출력
print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000,)
print(x_test.shape)  # (10000, 784)
print(t_test.shape)  # (10000,)
```

ch03/mnistRead.py

3.6.1 MNIST 데이터셋

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = \
    load_mnist(flatten=True, normalize=False)

img = x_train[0]
label = t_train[0]
print(label) # 5

print(img.shape)            # (784,)
img = img.reshape(28, 28)   # 원래 이미지의 모양으로 변형
print(img.shape)            # (28, 28)

img_show(img)
```

그림 3-25 MNIST 이미지 중 하나

여기서 주의 사항으로, flatten = True 로 설정해 읽어 들인 이미지는 1 차원 넘파이 배열로 저장되어 있다.
그래서 이미지를 표시할 때는 원래 형상인 28 × 28 크기로 다시 변형해야 한다.
Reshape () 메서드에 원하는 형상을 인수로 지정하면 넘파이 배열의 형상을 바꿀수 있다.
또한, 넘파이로 저장된 이미지 데이터를 PIL 용 데이터 객체로 변환해야 하며, 이변환은 Image . fromarray ()가 수행한다.

ch03/mnist_show.py

3.6.2 신경망의 추론 처리

```
def get_data():
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)

    return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

```
x, t = get_data()
network = init_network()

accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p = np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

드디어 이 MNIST 데이터셋을 가지고 추론을 수행하는 신경망을 구현할 차례. 이 신경망은 입력층 뉴런을 784개, 출력층 뉴런을 10개로 구성. 입력층 뉴런이 784개인 이유는 이미지 크기가 28×28=784이기 때문이고, 출력층 뉴런이 10개인 이유는 이 문제가 0에서9까지의 숫자를 구분하는 문제이기 때문

ch03/neuralnet_mnist.py

init _ network ()에서는 pickle 파일인 sample _ weight . pkl에 저장된 '학습된 가중치 매개변수'를 읽는다. 이 파일에는 가중치와 편향 매개변수가 딕셔너리 변수로 저장되어 있다.

3.6.2 신경망의 추론 처리

ch03/neuralnet_mnist.py

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import pickle
from dataset.mnist import load_mnist
from common.functions import sigmoid, softmax


def get_data():
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test


def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network


def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y


x, t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p= np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

3.6.3 배치 처리

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']
>>>
>>> x.shape
(10000, 784)
>>> x[0].shape
(784,)
>>> W1.shape
(784, 50)
>>> W2.shape
(50, 100)
>>> W3.shape
(100, 10)
```

우선 파이썬 인터프리터에서 앞서 구현한 신경망 각 층의 가중치 형상을 출력

그림 3-26 신경망 각 층의 배열 형상의 추이

[그림 3 - 26]을 전체적으로 보면 원소 784 개로 구성된 1 차원 배열(원래는 28 × 28 인 2 차원 배열)이 입력되어 마지막에는 원소가 10 개인 1 차원 배열이 출력되는 흐름.
이는 이미지데이터를 1 장만 입력했을 때의 처리 흐름

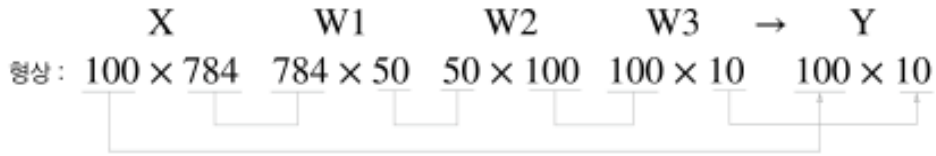


그림 3-27 배치 처리를 위한 배열들의 형상 추이

[그림 3 - 27]과 같이 입력 데이터의 형상은 100 × 784 , 출력 데이터의 형상은 100 × 10 이 된다.
이는 100 장 분량 입력 데이터의 결과가 한 번에 출력됨을 나타낸다.
가령 x [0]와 y [0]에는 0 번째 이미지와 그 추론 결과가, x [1]과 y [1]에는 1 번째의 이미지와 그 결과가 저장되는 식.

3.6.3 배치 처리

ch03/neuralnet_mnist_batch.py

```python
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록
import numpy as np
import pickle
from dataset.mnist import load_mnist
from common.functions import sigmoid, softmax


def get_data():
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test


def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network


def predict(network, x):
    w1, w2, w3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, w1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, w2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, w3) + b3
    y = softmax(a3)

    return y


x, t = get_data()
network = init_network()

batch_size = 100 # 배치 크기
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

CHAPTER 4 신경망 학습

손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기

Contents

4.1.1 데이터 주도 학습

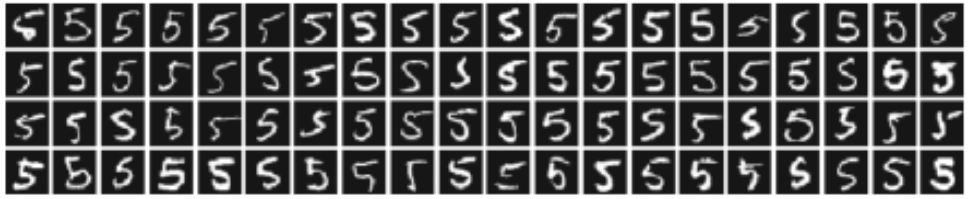


그림 4-1 손글씨 숫자 ' 5 '의 예 : 사람마다 자신만의 필체가 있다

기계학습은 데이터가 생명.
그래서 기계학습의 중심에는 데이터 가 존재합니다.
이처럼 데이터가 이끄는 접근 방식 덕에사람 중심 접근에서 벗어날 수 있다.

그림 4-2 규칙을 '사람'이 만드는 방식에서 '기계'가 데이터로부터 배우는 방식으로의 패러다임 전환 : 회색 블록은 사람

[그림 4 - 2]와 같이 신경망은 이미지를 '있는 그대로' 학습한다.
두 번째 접근 방식(특징과기계학습 방식)에서는 특징을 사람이 설계했지만, 신경망은 이미지에 포함된 중요한 특징까지도 '기계'가 스스로 학습할 것이다.

4.1.2 훈련 데이터와 시험 데이터

기계학습 문제는 데이터를 훈련 데이터 training data 와 시험 데이터 test data 로 나눠 학습과 실험을 수행하는 것이 일반적이다.
우선 훈련 데이터만 사용하여 학습하면서 최적의 매개변수를 찾는다.
그런 다음 시험 데이터를 사용하여 앞서 훈련한 모델의 실력을 평가하는 것이다

훈련 데이터와 시험 데이터를 나눠야 할것인가?
그것은 우리가 원하는 것은 범용적으로 사용할수 있는 모델이기 때문이다.
이 **범용 능력** 을 제대로 평가하기 위해 훈련 데이터와 시험 데이터를 분리하는 것이다.

그래서 데이터셋 하나로만 매개변수의 학습과 평가를 수행하면 올바른 평가가 될 수 없다.
수중의 데이터셋은 제대로 맞히더라도 다른 데이터셋에는 엉망인 일도 벌어진다.
참고로 한데이터셋에만 지나치게 최적화된 상태를 오버피팅 overfitting * 이라고 한다.
오버피팅 피하기는 기계학습의 중요한 과제이기도 하다

4.2 손실 함수

이 '행복 지표' 이야기는 하나의 비유지만, 실은 신경망 학습에서도 이와 같은 일을 수행.
신경망 학습에서는 현재의 상태를 '하나의 지표'로 표현한다
그리고 그 지표를 가장 좋게만들어주는 가중치 매개변수의 값을 탐색하는 것. '행복 지표'를 가진 사람이 그 지표를근거로 '최적의 인생'을 탐색하듯, 신경망도 '하나의 지표'를 기준으로 최적의 매개변수 값을 탐색

신경망 학습에서 사용하는 지표는 손실 함수 loss function * 라고 합니다. 이 손실 함수는 임의의 함수를 사용할 수도 있지만 일반적으로는 평균 제곱 오차와 교차 엔트로피 오차를 사용.

4.2.1 평균 제곱 오차

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2 \qquad [식 4.1]$$

여기서 y k 는 신경망의 출력(신경망이 추정한 값), t k 는 정답 레이블, k 는 데이터의 차원 수를 나타낸다.
이를테면 " 3 . 6 손글씨 숫자 인식" 예에서 y k 와 t k 는 다음과 같은 원소 10 개짜리 데이터이다.

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

이 배열들의 원소는 첫 번째 인덱스부터 순서대로 숫자 ' 0 ', ' 1 ', ' 2 ', ... 일 때의 값입니다. 여기에서 신경망의 출력 y 는 소프트맥스 함수의 출력입니다

이처럼 한 원소만 1 로 하고 그 외는 0 으로 나타내는 표기법을 원-핫 인코딩 이라 한다고 했습니다.

ch04/4_2_1.py

```python
import numpy as np

def mean_squared_error (y,t):
    return 0.5*np.sum((y-t)**2)

t=[0,0,1,0,0,0,0,0,0,0]
y=[0.1,0.05,0.6,0.0,0.05,0.1,0.0,0.1,0.0,0.0]
print(mean_squared_error(np.array(y),np.array(t)))
```

4.2.2 교차 엔트로피 오차

또 다른 손실 함수로서 교차 엔트로피 오차 cross entropy error , CEE 도 자주 이용한다. 교차 엔트로피 오차의 수식은 다음과 같다.

$$E = -\sum_k t_k \log y_k$$

[식 4.2]

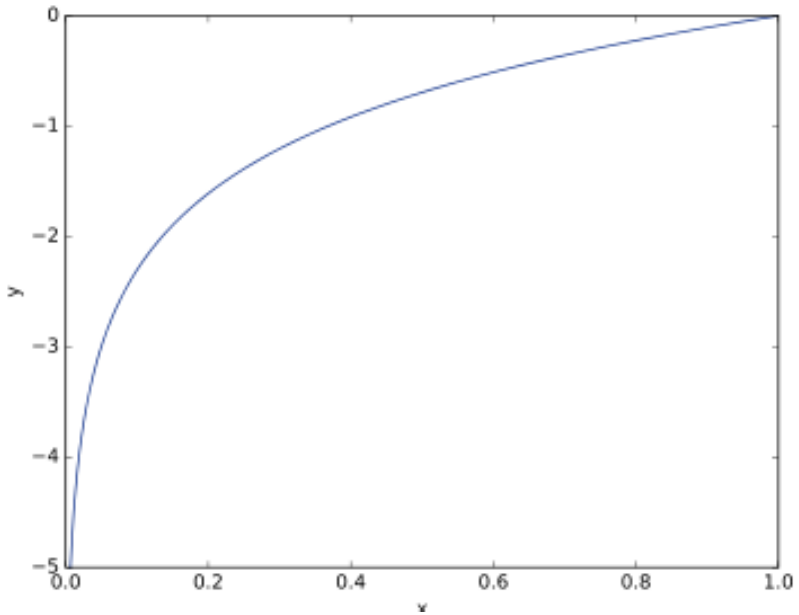


그림 4-3 자연로그 y = logx 의 그래프

ch04/4_2_2.py

```
import numpy as np

def cross_entropy_error (y,t):
    delta=1e-7
    return -np.sum(t*np.log(y+delta))

t=[0,0,1,0,0,0,0,0,0,0]
y=[0.1,0.05,0.6,0.0,0.05,0.1,0.0,0.1,0.0,0.0]
print(cross_entropy_error(np.array(y),np.array(t)))
```

이 그림에서 보듯이 x 가 1 일 때 y 는 0 이 되고 x 가 0 에 가까워질수록 y 의 값은 점점 작아진다.
[식 4 . 2]도 마찬가지로 정답에 해당하는 출력이 커질수록 0 에 다가가다가, 그 출력이 1 일 때 0이 된다.
반대로 정답일 때의 출력이 작아질수록 오차는 커진다.

4.2.3 미니배치 학습

기계학습 문제는 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아낸다.
이렇게 하려면 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 한다.
즉, 훈련 데이터가 100 개 있으면 그로부터 계산한 100 개의 손실 함수 값들의 합을 지표로 삼는 것입니다.

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \qquad [식 4.3]$$

이때 데이터가 N 개라면 t nk 는 n 번째 데이터의 k 번째 값을 의미한다 (y nk 는 신경망의 출력, t nk는 정답 레이블).
수식이 좀 복잡해 보이지만 데이터 하나에 대한 손실 함수인 [식 4 . 2]를 단순히 N 개의 데이터로 확장했을 뿐이다.

신경망 학습에서도 훈련 데이터로부터 일부만 골라 학습을 수행합니다. 이 일부를 미니배치 mini - batch 라고 한다.
가령 60 , 000 장의 훈련 데이터 중에서 100 장을 무작위로 뽑아 그 100 장만을 사용하여 학습하는 것이다.
이러한 학습 방법을 미니배치 학습 이라고 한다.

4.2.3 미니배치 학습

ch04/4_2_3.py

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from dataset.mnist import load_mnist

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

print(x_train.shape)
print(t_train.shape)

train_size=x_train.shape[0]
batch_size=10
batch_mask=np.random.choice(train_size,batch_size)
x_batch=x_train[batch_mask]
t_batch=t_train[batch_mask]

np.random.choice(60000,10)
```

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(t * np.log(y + 1e-7)) / batch_size
```

이 코드에서 y 는 신경망의 출력, t 는 정답 레이블이다.
y 가 1 차원이라면, 즉 데이터 하나당교차 엔트로피 오차를 구하는 경우는 reshape 함수로 데이터의 형상을 바꿔준다.
그리고 배치의 크기로 나눠 정규화하고 이미지 1 장당 평균의 교차 엔트로피 오차를 계산한다.

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

이 구현에서는 원-핫 인코딩일 때 t 가 0 인 원소는 교차 엔트로피 오차도 0 이므로, 그 계산은무시해도 좋다는 것이 핵심이다.
다시 말하면 정답에 해당하는 신경망의 출력만으로 교차 엔트로피 오차를 계산할 수 있다.
그래서 원-핫 인코딩 시 t * np . log (y)였던 부분을 레이블 표현일 때는 np . log (y [np . arange (batch _ size), t])로 구현한다

4.2.5 왜 손실 함수를 설정하는가?

정확도는 매개변수의 미소한 변화에는 거의 반응을 보이지 않고, 반응이 있더라도 그 값이 불연속적으로 갑자기 변화한다.
이는 '계단 함수'를 활성화 함수로 사용하지 않는 이유와도 들어맞는다.
만약 활성화 함수로 계단 함수를 사용하면 지금까지 설명한 것과 같은 이유로 신경망 학습이 잘 이뤄지지 않는다.
계단 함수의 미분은 [그림 4 – 4]와 같이 대부분의 장소(0이외의 곳)에서 0 이다.
그 결과, 계단 함수를 이용하면 손실 함수를 지표로 삼는 게 아무 의미가 없게 된다. 매개변수의 작은 변화가 주는 파장을 계단 함수가 말살하여 손실 함수의 값에는 아무런 변화가 나타나지 않기 때문이다

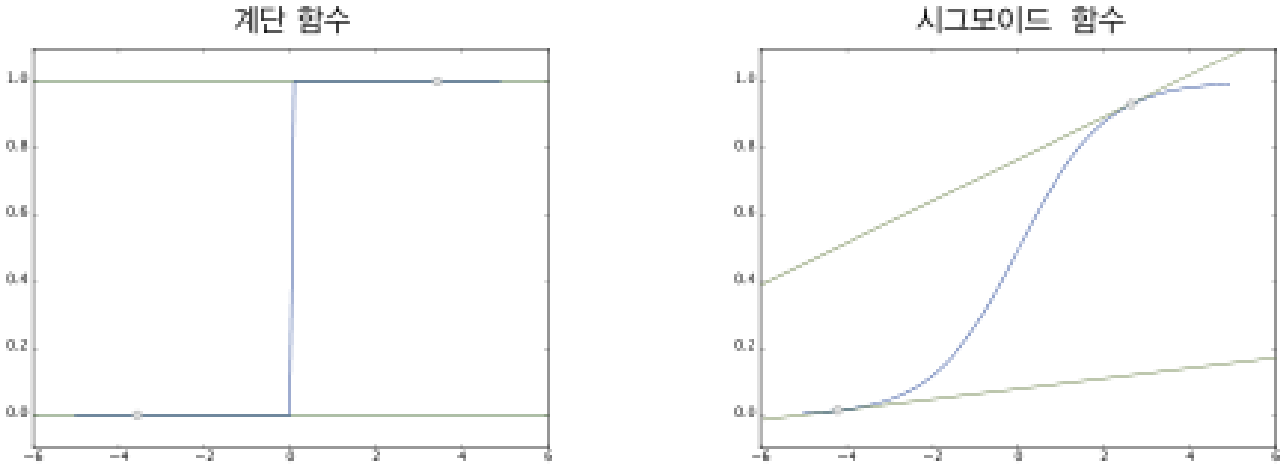


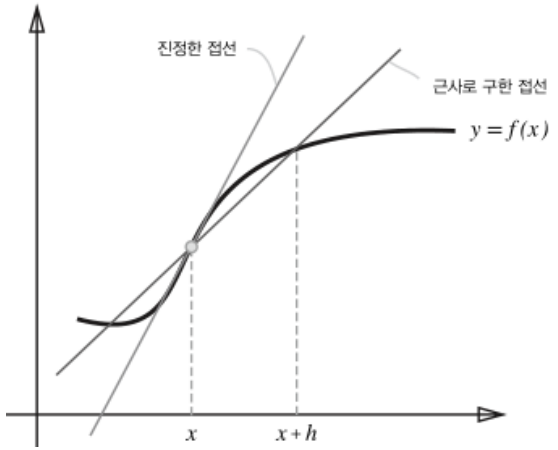
그림 4-4 계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0 이지만, 시그모이드 함수의 기울기(접선)는 0 이 아니다

4.3 수치 미분

4.3.1 미분

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$ [식 4.4]

[식 4 . 4]는 함수의 미분을 나타낸 식이다.
좌변은 f (x)의 x 에 대한 미분(x 에 대한 f (x)의 변화량)을 나타내는 기호이다. 결국, x 의 '작은 변화'가 함수 f (x)를 얼마나 변화시키느냐를 의미하며, 이때 시간의 작은 변화, 즉 시간을 뜻하는 h 를 한없이 0 에 가깝게 한다는 의미로 나타낸다.



[그림 4 - 5]와 같이 수치 미분에는 오차가 포함된다. 이 오차를 줄이기 위해 (x + h)와 (x -h)일 때의 함수 f 의 차분을 계산하는 방법을 쓰기도 한다. 이 차분은 x 를 중심으로 그 전후의차분을 계산한다는 의미에서 중심 차분 혹은 중앙 차분 이라 한다(한편,(x + h)와 x 의 차분은전방 차분 이라 함).

4.3.2 수치 미분의 예

$$y = 0.01x^2 + 0.1x$$ [식 4.5]

```
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

[식 4 . 5]를 파이썬으로 구현하면 다음과 같이 된 다.

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

x = np.arange(0.0, 20.0, 0.1) # 0에서 20까지 0.1 간격의 배열 x를 만든다.
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```

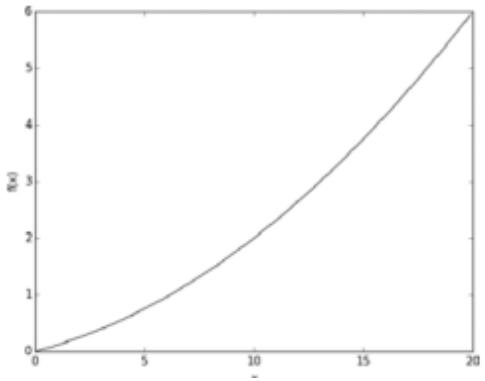


그림 4-6 식 f (x) = 0 . 01x 2 + 0 . 1x 의 그래프

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

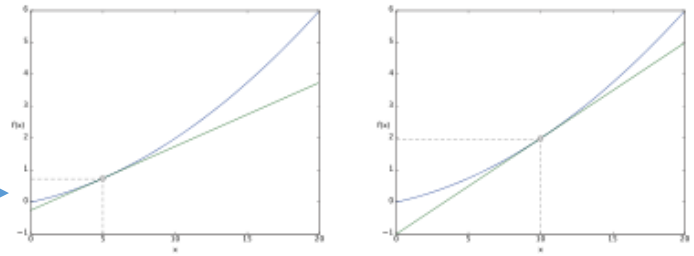
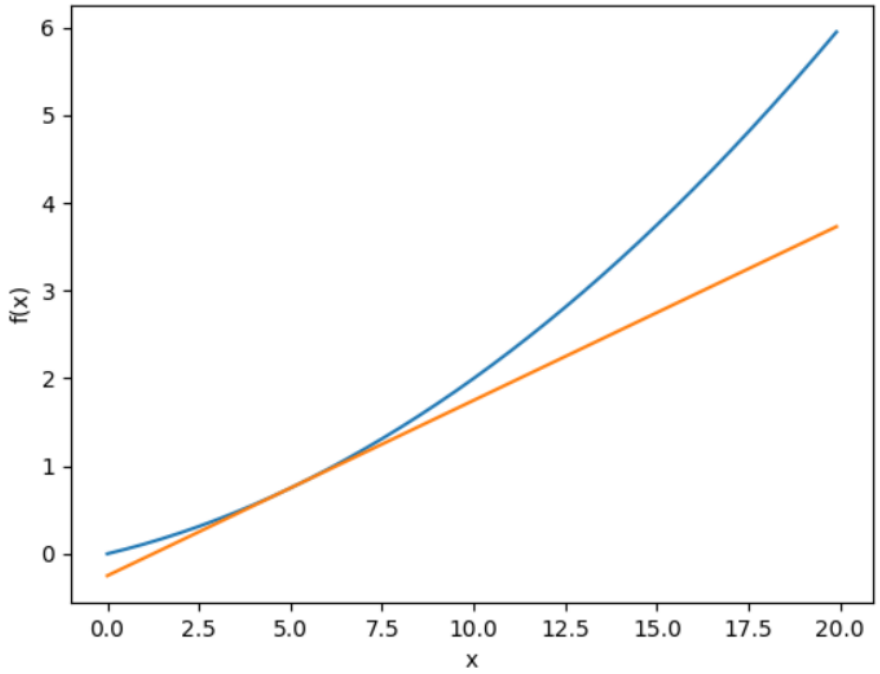


그림 4-7 x = 5 , x = 10 에서의 접선 : 직선의 기울기는 수치 미분에서 구한 값을 사용하였다.

4.3.2 수치 미분의 예

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


def numerical_diff(f, x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)


def function_1(x):
    return 0.01*x**2 + 0.1*x


def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")

tf = tangent_line(function_1, 5)
y2 = tf(x)

plt.plot(x, y)
plt.plot(x, y2)
plt.show()
```

$$y = 0.01x^2 + 0.1x$$



ch04/gradient_1d.py

4.3.3 편미분

$$f(x_0, x_1) = x_0^2 + x_1^2$$

[식 4.6]

```
def function_2(x):
    return x[0]**2 + x[1]**2
    # 또는 return np.sum(x**2)
```

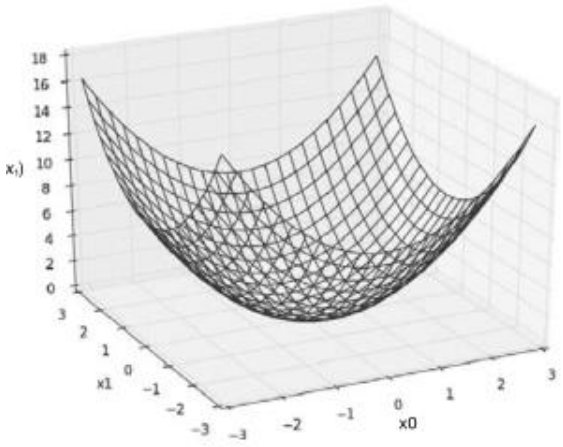


그림 4-8 f (x 0 , x 1) = x 0 2 + x 1 2 의 그래프

즉 x 0 와 x 1 중 어느 변수에 대한 미분이냐를 구별해야 한다.
덧붙여 이와 같이 변수가 여럿인 함수에 대한 미분을 편미분 이라고 한다.

4.4 기울기

```
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

    for idx in range(x.size):
        tmp_val = x[idx]
        # f(x+h) 계산
 x[idx] = tmp_val + h
 fxh1 = f(x)

 # f(x-h) 계산
 x[idx] = tmp_val - h
 fxh2 = f(x)

 grad[idx] = (fxh1 - fxh2) / (2*h)
 x[idx] = tmp_val # 값 복원
```

x.size=2이므로 idx=0,1이 된다.
Idx=0이면 x[0]값이 tmp_val+h와 tmp_val-h인 경우에 f(x)값이 계산되어 편미분이 된다.(x[1]은 입력값)
Idx=1이면 x[1]값이 tmp_val+h와 tmp_val-h인 경우에 f(x)값이 계산되어 편미분이 된다.(x[0]은 입력값)

편미분 값은 grad[idx]에 저장된다.

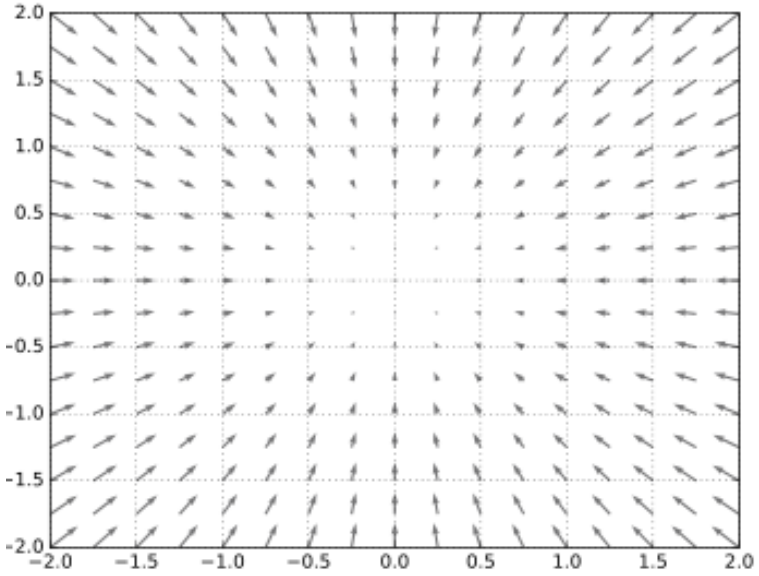


그림 4-9 f (x 0 , x 1) = x 0 2 + x 1 2 의 기울기

기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향

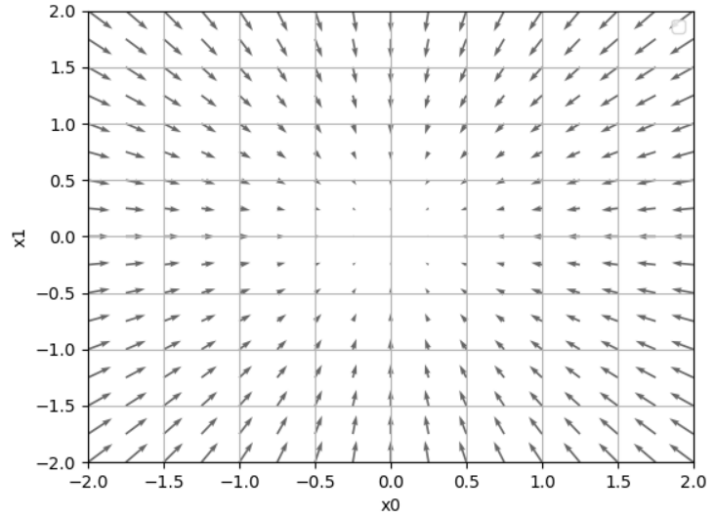
실제로는 [6 . 0000000000037801 , 7 . 9999999999991189]라는 값이 얻어지지만 [6 ., 8 .]으로 출력된다.
이는 넘파이 배열을 출력할 때 수치를 '보기 쉽도록' 가공하기 때문이다

4.4 기울기

```python
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D


def _numerical_gradient_no_batch(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

    for idx in range(x.size):
        tmp_val = x[idx]

        # f(x+h) 계산
        x[idx] = float(tmp_val) + h
        fxh1 = f(x)

        # f(x-h) 계산
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 값 복원

    return grad
```

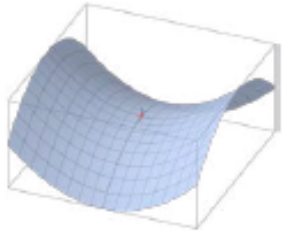
ch04/gradient_2d.py

```python
def numerical_gradient(f, X):
    if X.ndim == 1:
        return _numerical_gradient_no_batch(f, X)
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            grad[idx] = _numerical_gradient_no_batch(f, x)

        return grad


def function_2(x):
    if x.ndim == 1:
        return np.sum(x**2)
    else:
        return np.sum(x**2, axis=1)


def tangent_line(f, x):
    d = numerical_gradient(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

if __name__ == '__main__':
    x0 = np.arange(-2, 2.5, 0.25)
    x1 = np.arange(-2, 2.5, 0.25)
    X, Y = np.meshgrid(x0, x1)

    X = X.flatten()
    Y = Y.flatten()

    grad = numerical_gradient(function_2, np.array([X, Y]) )

    plt.figure()
    plt.quiver(X, Y, -grad[0], -grad[1],  angles="xy",color="#666666")#,headwidt
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.xlabel('x0')
    plt.ylabel('x1')
    plt.grid()
    plt.legend()
    plt.draw()
    plt.show()
```



4.4.1 경사법(경사 하강법)

매개변수 공간이 광대하여 어디가 최솟값이 되는 곳인지를 짐작할 수 없다.
이런 상황에서 기울기를 잘 이용해 함수의 최솟값(또는 가능한 한 작은 값)을 찾으려는 것이 경사법이다.



문제 : 경사법으로 f (x 0 , x 1) = x 02 + x 1 2 의 최솟값을 구하라

```
>>> def function_2(x):
...     return x[0]**2 + x[1]**2
...
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
array([ -6.11110793e-10,   8.14814391e-10])
```

※ 교재의 상세 과정을 참고하여 실습을 진행합니다.

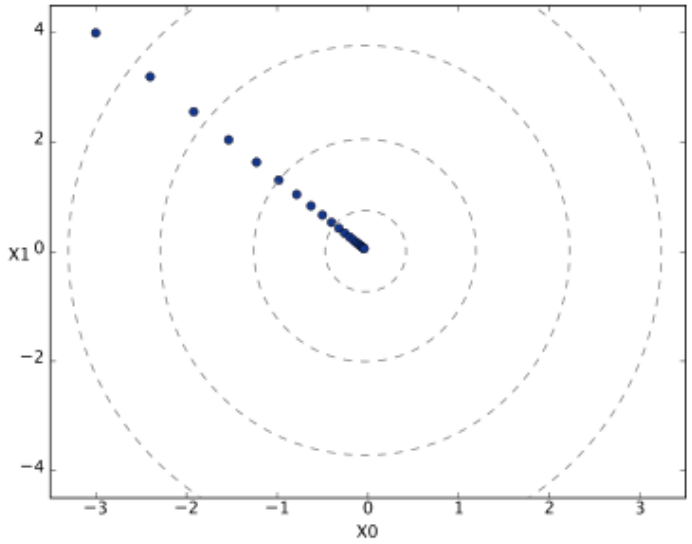
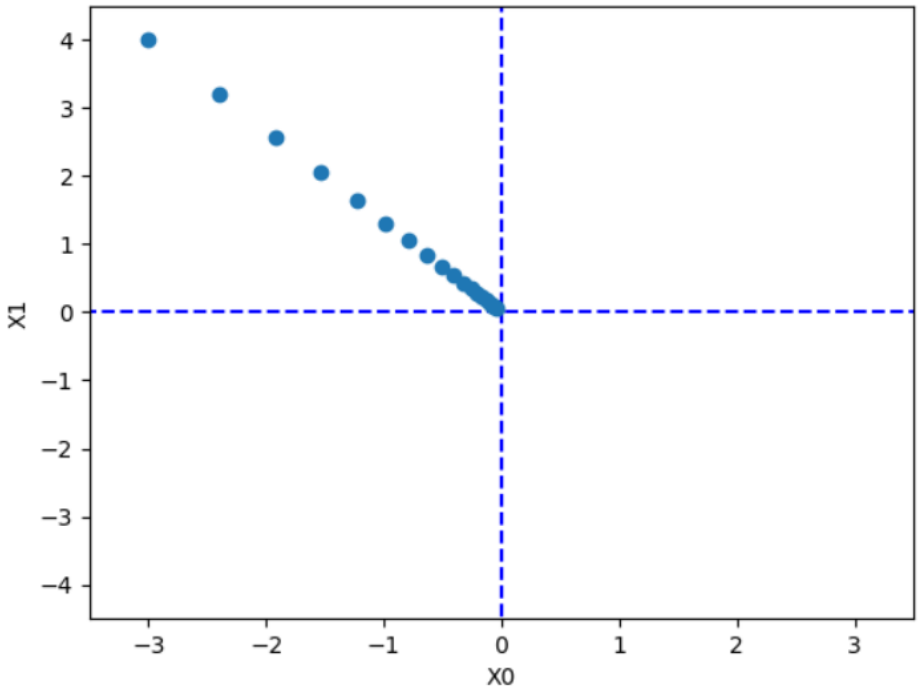


그림 4-10 경사법에 의한 f (x 0 , x 1) = x 0 2 + x 1 2 의 갱신 과정 : 점선은 함수의 등고선을 나타낸다

4.4.1 경사법(경사 하강법)

ch04/gradient_method.py




```python
import numpy as np
import matplotlib.pylab as plt
from gradient_2d import numerical_gradient


def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x = init_x
    x_history = []

    for i in range(step_num):
        x_history.append( x.copy() )

        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x, np.array(x_history)


def function_2(x):
    return x[0]**2 + x[1]**2

init_x = np.array([-3.0, 4.0])

lr = 0.1
step_num = 20
x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)

plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
plt.plot(x_history[:,0], x_history[:,1], 'o')

plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.show()
```

lr을 변경시켜 볼 것

4.4.2 신경망에서의 기울기

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

[식 4.8]

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

신경망 학습에서도 기울기를 구해야 한다.
여기서 말하는 기울기는 가중치 매개변수에 대한
손실 함수의 기울기입니다

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3) # 정규분포로 초기화

    def predict(self, x):
        return np.dot(x, self.W)

    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)

        return loss
```

여기에서는 common / functions . py 에 정의한 softmax 와
cross _ entropy _ error 메서드를이용한다.
그리고 common / gradient . py 에 정의한 numerical _ gradient
메서드도 이용 한 다.

4.4.2 신경망에서의 기울기

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient


class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3) # 정규분포로 초기화

    def predict(self, x):
        return np.dot(x, self.W)

    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)

        return loss

x = np.array([0.6, 0.9])
t = np.array([0, 0, 1])

net = simpleNet()

f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)

print(dW)
```

p.135의 def f(w)를 lamda 함수로 표현

ch04/gradient_simplenet.py

4.5 학습 알고리즘 구현하기

전제
신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다.
신경망 학습은 다음과 같이 4 단계로 수행한다.

1 단계 - 미니배치
훈련 데이터 중 일부를 무작위로 가져온다.
이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실함수 값을 줄이는 것이 목표.

2 단계 - 기울기 산출
미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다.

3 단계 - 매개변수 갱신
가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4 단계 - 반복
1 ~ 3 단계를 반복한다.

4.5.1 2층 신경망 클래스 구현하기

```
import sys, os
sys.path.append(os.pardir)
from common.functions import *
from common.gradient import numerical_gradient

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * \
                            np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * \
                            np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x : 입력 데이터, t : 정답 레이블
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x : 입력 데이터, t : 정답 레이블
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads
```

TwoLayerNet 의 구현은 스탠퍼드 대학교의 CS231n 수업에서 제공한 파이썬 소스 코드를 참고.

4.5.1 2층 신경망 클래스 구현하기

변수	설명
params	신경망의 매개변수를 보관하는 딕셔너리 변수(인스턴스 변수)
	params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향
	params['W2']는 2번째 층의 가중치, params['b2']는 2번째 층의 편향
grads	기울기 보관하는 딕셔너리 변수(numerical_gradient() 메서드의 반환 값)
	grads['W1']은 1번째 층의 가중치의 기울기, grads['b1']은 1번째 층의 편향의 기울기
	grads['W2']는 2번째 층의 가중치의 기울기, grads['b2']는 2번째 층의 편향의 기울기

표 4-1 TwoLayerNet 클래스가 사용하는 변수

메서드	설명
__init__(self, input_size, hidden_size, output_size)	초기화를 수행한다. 인수는 순서대로 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수
predict(self, x)	예측(추론)을 수행한다. 인수 x는 이미지 데이터
loss(self, x, t)	손실 함수의 값을 구한다. 인수 x는 이미지 데이터, t는 정답 레이블(아래 칸의 세 메서드의 인수들도 마찬가지)
accuracy(self, x, t)	정확도를 구한다.
numerical_gradient(self, x, t)	가중치 매개변수의 기울기를 구한다.
gradient(self, x, t)	가중치 매개변수의 기울기를 구한다. numerical_gradient()의 성능 개선판! 구현은 다음 장에서...

표 4-2 TwoLayerNet 클래스의 메서드

4.5.1 2층 신경망 클래스 구현하기

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

net = TwoLayerNet(input_size=784, hidden_size=100, output_size=10)
print(net.params['W1'].shape)
print(net.params['b1'].shape)
print(net.params['W2'].shape)
print(net.params['b2'].shape)

x=np.random.rand(100,784)
y=net.predict(x) #predict

t=np.random.rand(100,10)

grads=net.numerical_gradient(x,t)
print(grads['W1'].shape)
print(grads['b1'].shape)
print(grads['W2'].shape)
print(grads['b2'].shape)
```

ch04/4_5_1.py

4.5.2 미니배치 학습 구현하기

ch04/train_neuralnet.py

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)
```

4.5.3 시험 데이터로 평가하기

```
        # 매개변수 갱신
        for key in ('W1', 'b1', 'W2', 'b2'):
            network.params[key] -= learning_rate * grad[key]

        # 학습 경과 기록
        loss = network.loss(x_batch, t_batch)
        train_loss_list.append(loss)

        # 1에폭당 정확도 계산
        if i % iter_per_epoch == 0:
            train_acc = network.accuracy(x_train, t_train)
            test_acc = network.accuracy(x_test, t_test)
            train_acc_list.append(train_acc)
            test_acc_list.append(test_acc)
            print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 그래프 그리기
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```