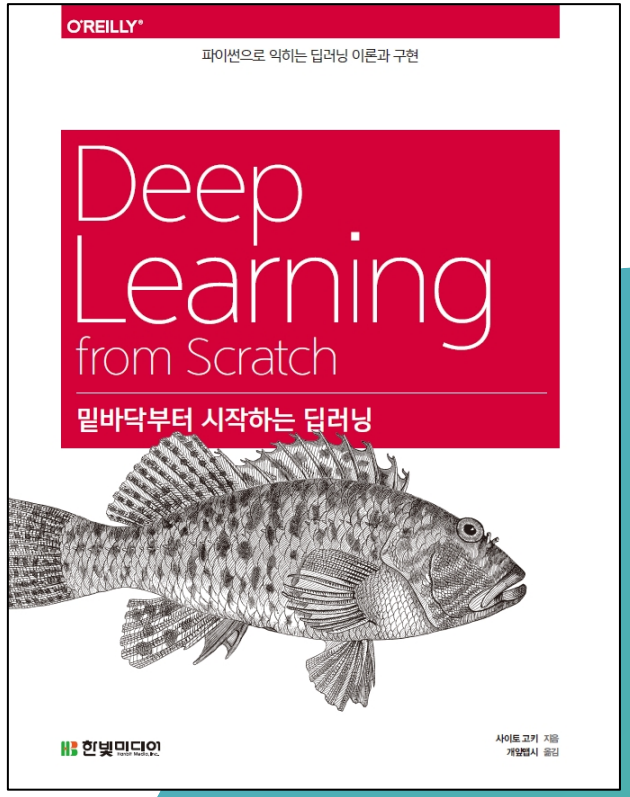
CHAPTER 3 신경망

밑바닥부터 시작하는 딥러닝



홍익대학교 컴퓨터공학과
홍길동

이 책의 학습 목표

- CHAPTER 1 파이썬에 대해 간략하게 살펴보고 사용법 익히기
- CHAPTER 2 퍼셉트론에 대해 알아보고 퍼셉트론을 써서 간단한 문제를 풀어보기
- CHAPTER 3 신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기
- CHAPTER 4 손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기
- CHAPTER 5 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법 배우기
- CHAPTER 6 신경망(딥러닝) 학습의 효율과 정확도를 높이기
- CHAPTER 7 CNN의 메커니즘을 자세히 설명하고 파이썬으로 구현하기
- CHAPTER 8 딥러닝의 특징과 과제, 가능성, 오늘날의 첨단 딥러닝에 대해 알아보기

Contents

Contents

CHAPTER 3 신경망

신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기

3.1.1 신경망의 예

가장 왼쪽 줄: 입력층 , 맨 오른쪽 줄: 출력층 , 중간 줄: 은닉층

은닉층의 뉴런은 (입력층이나 출력층과 달리)사람 눈에는 보이지 않음

입력층에서 출력층방향으로 차례로 0 층, 1 층, 2 층

(층 번호를 0 부터 시작하는 이유는 파이썬 배열의 인덱스도 0 부터 시작)

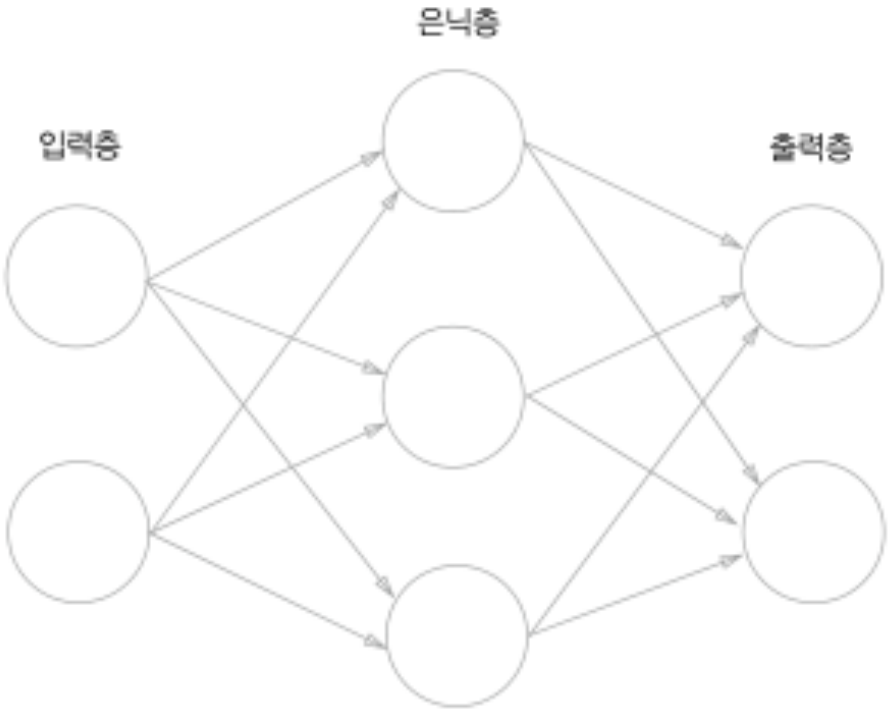
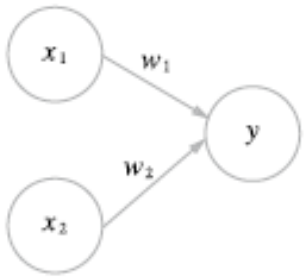


그림 3-1 신경망의 예

3.1.2 퍼셉트론 복습

[식 3 . 2]: 가중치가 곱해진 입력 신호의 총합을 계산, 그합을 활성화 함수에 입력해 결과를 내는 2 단계로 처리 .
이 식은 다음 2단계로 나눌수 있음

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]: 가중치가 달린 입력 신호와 편향의 총합을 계산하고, 이를 a 라 한다.
[식3 . 5]: a 를 함수 h ()에 넣어 y 를 출력.

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

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

그림 3-4 활성화 함수의 처리 과정

3.1.3 활성화 함수의 등장

h (x): 활성화 함수 activation function,
입력 신호의 총합을 출력 신호로 변환하는 함수
입력 신호의 총합이 활성화를 일으키는지를 정하는 역할

[식 3.2]는 가중치가 곱해진 입력 신호의 총합을 계산하고, 그 합을 활성화 함수에 입력해 결과를 내는 2단계로 처리
지금까지와 같이 뉴런을 큰 원(○)으로 그려보면 [식 3 . 4]와 [식 3 . 5]는 [그림 3 - 4]처럼 나타낼 수 있다

$$a = b + w_1x_1 + w_2x_2 \qquad [식\ 3.4]$$

$$y = h(a) \qquad [식\ 3.5]$$

그림 3-4 활성화 함수의 처리 과정

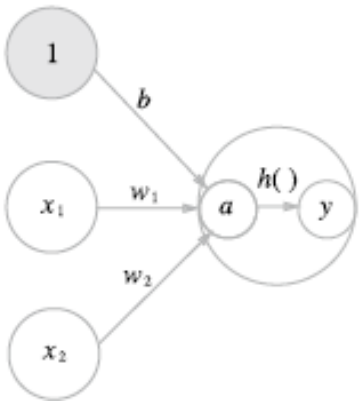


그림 **3-5** 왼쪽은 일반적인 뉴런, 오른쪽은 활성화 처리 과정을 명시한 뉴런(a는 입력 신호의 총합, $h( )$는 활성화 함수,y는 출력)

3.2.1 시그모이드 함수

시그모이드 함수sigmoid function를 나타낸 식

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

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)
```

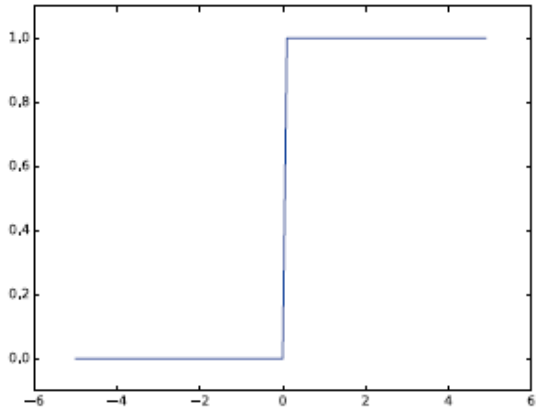
넘파이 배열에 부등호 연산을 수행하면 배열의 원소 각각에 부등호 연산을 수행한 bool 배열이 생성.
이 예에서는 배열 x의 원소 각각이 0보다 크면 True로, 0 이하면 False로 변환 한 새로운 배열 y가 생성

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])
```

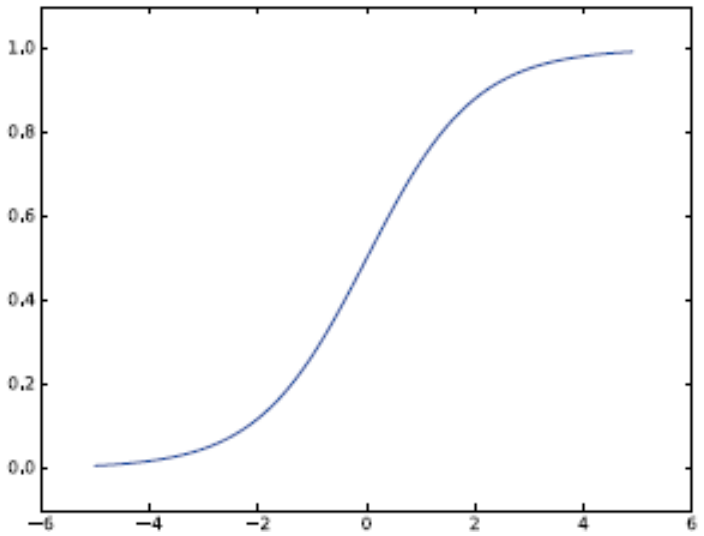
```
>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([ 2.,  3.,  4.])
>>> 1.0 / t
array([ 1.        ,  0.5       ,  0.33333333])
```

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()
```

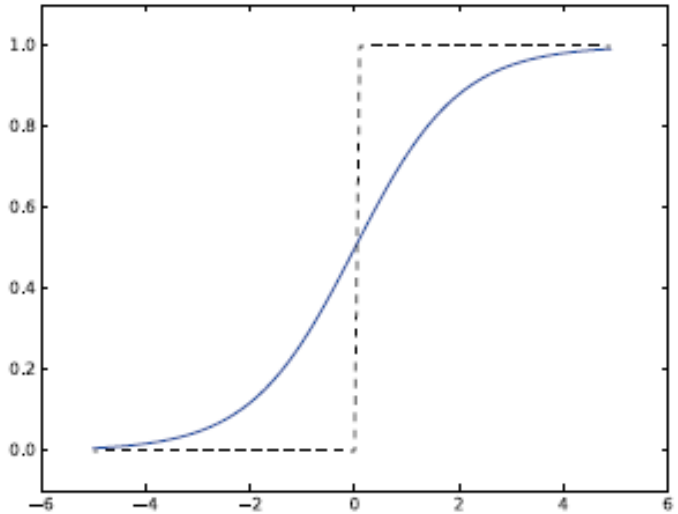
이 코드를 실행하면 [그림 3-7]의 그래프를 나타낸다

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 중
요한 역할.

3.2.6 비선형 함수

계단 함수와 시그모이드 함수의 중요한 공통점
둘 모두는비선형 함수.
시그모이드 함수는 곡선, 계단 함수는 계단처럼 구부러진 직선으로 나타나며, 동시에
비선형 함수로 분류.

함수: 어떤 값을입력하면 그에 따른 값을 돌려주는 '변환기'.
선형 함수: 이 변환기에 무언가 입력했을 때 출력이 입력의 상수배만
큼 변하는 함수

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

비선형 함수는 문자 그대로 '선형이 아닌' 함수. 즉, 직선 1 개로는 그릴 수 없는 함수

3.2.7 ReLU 함수

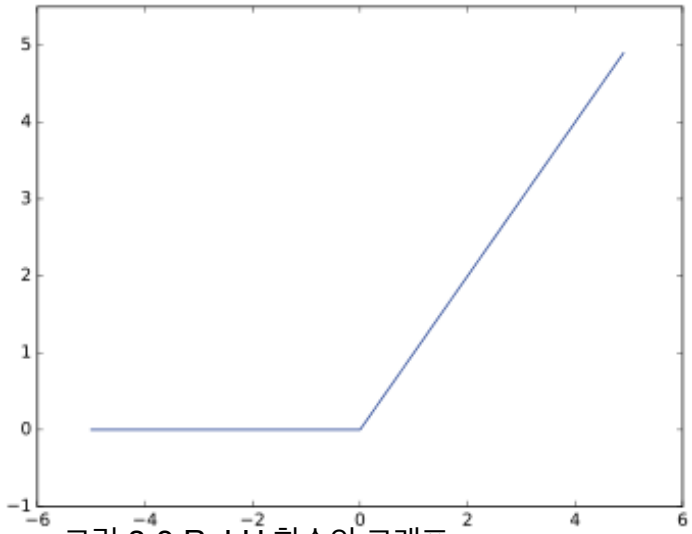


그림 3-9 ReLU 함수의 그래프

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

수식으로는 [식 3 . 7]처럼 쓸 수 있다.

```
def relu(x):
    
```

여기에서는 넘파이의 maximum 함수를 사용.
maximum 은 두 입력 중 큰 값을 선택해 반환하는 함수.
이번 장에서는 앞으로 시그모이드 함수를 활성화 함수로 사용했으나,
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.
A 는 1 차원 배열이고 원소 4 개로 구성.
A . shape 은 튜플을 반환.
- 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)
```

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

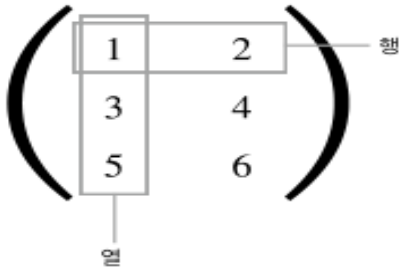


그림 3-10 2 차원 배열(행렬)의 행(가로)과 열(세로)

3.3.2 행렬의 곱

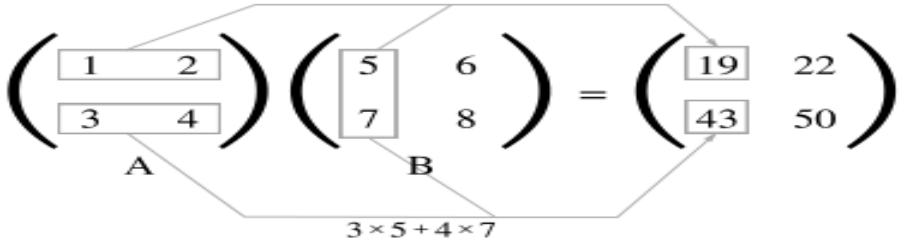


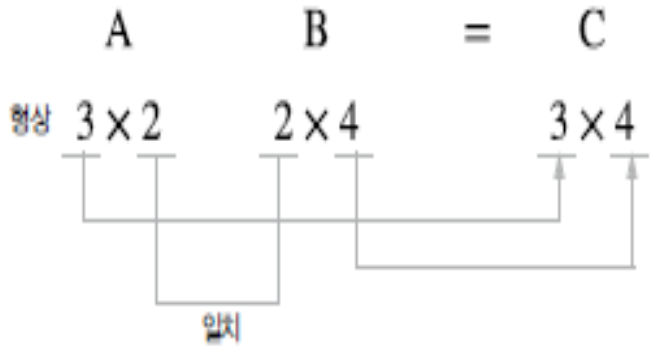
그림 3-11 행렬의 곱 계산 방법

```
>>> 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

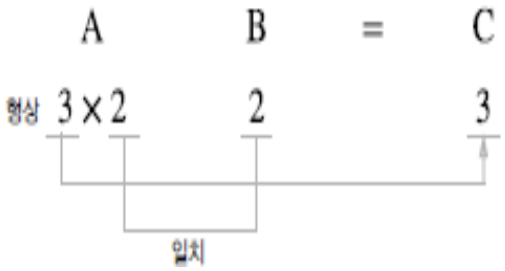
```
>>> A = np.array([[1,2,3], [4,5,6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> B.shape
(3, 2)
>>> np.dot(A, B)
array([[22, 28],
       [49, 64]])
```

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

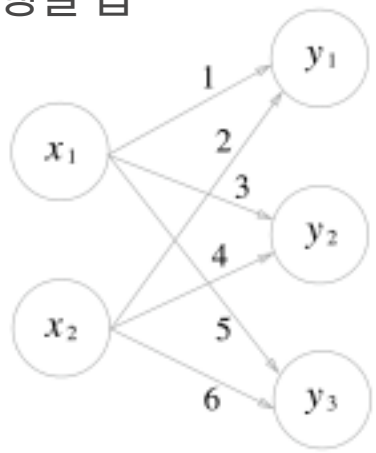



```
>>> C = np.array([[1,2], [3,4]])
>>> C.shape
(2, 2)
>>> A.shape
(2, 3)
>>> np.dot(A, C)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

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 스칼라곱을 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

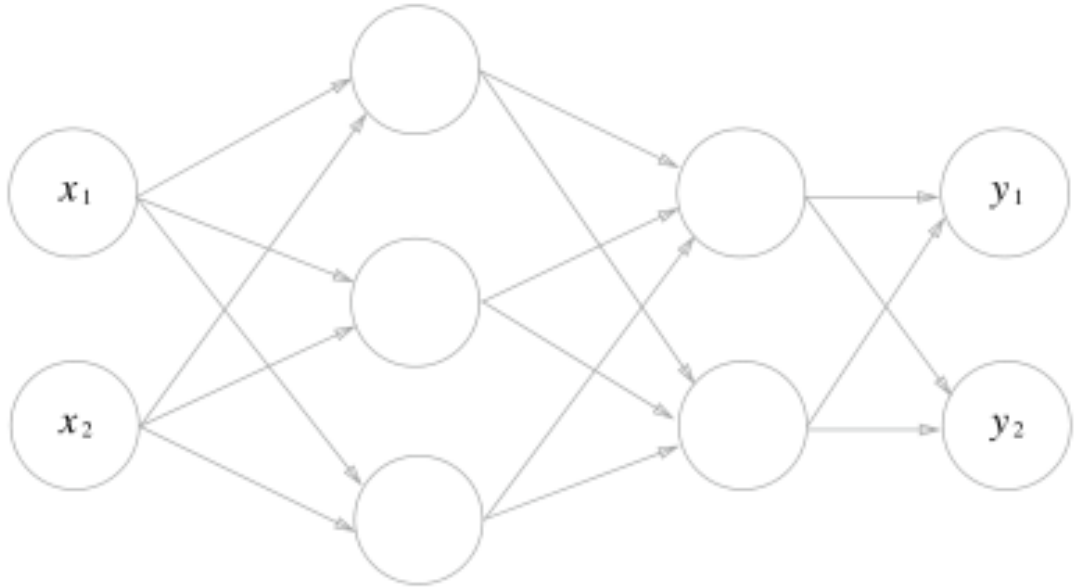


그림 3-15 3 층 신경망 : 입력층(0 층)은 2 개, 첫 번째 은닉층(1 층)은 3 개, 두 번째 은닉층(2 층)은 2 개, 출력층(3 층)은 2개의 뉴련으로 구성

3.4.1 표기법 설명

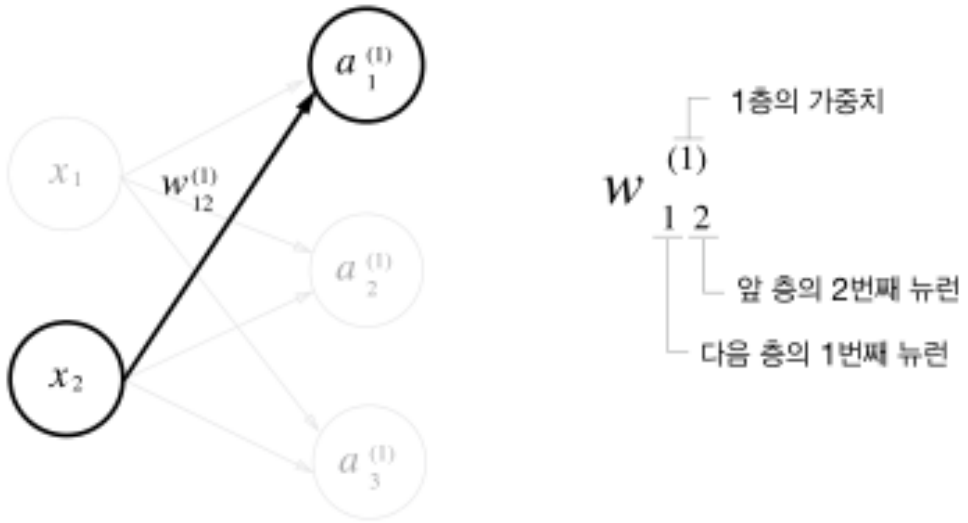


그림 3-16 중요한 표기

가중치와 은닉층 뉴런의 오른쪽 위 위첨자 (1) : 1 층의가중치, 1 층의 뉴런임을 뜻하는 번호.
가중치의 오른쪽 아래의 두 숫자는 차례로 다음 층 뉴런과 앞 층 뉴런의 인덱스 번호

3.4.2 각 층의 신호 전달 구현하기

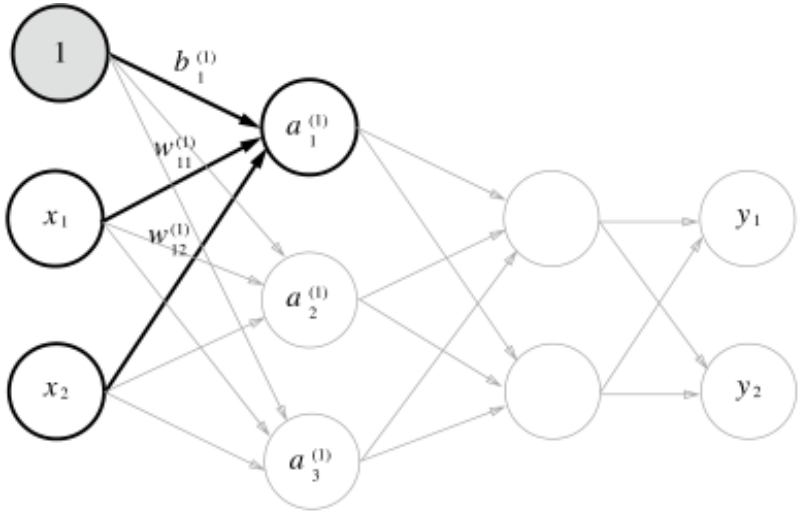


그림 3-17 입력층에서 1 층으로 신호 전달

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

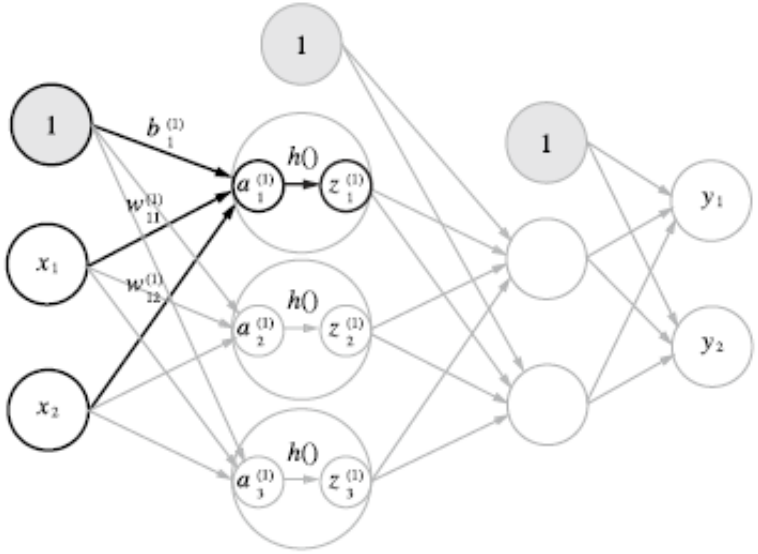
$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)} \quad [식 3.9]$$

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

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

```
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
```

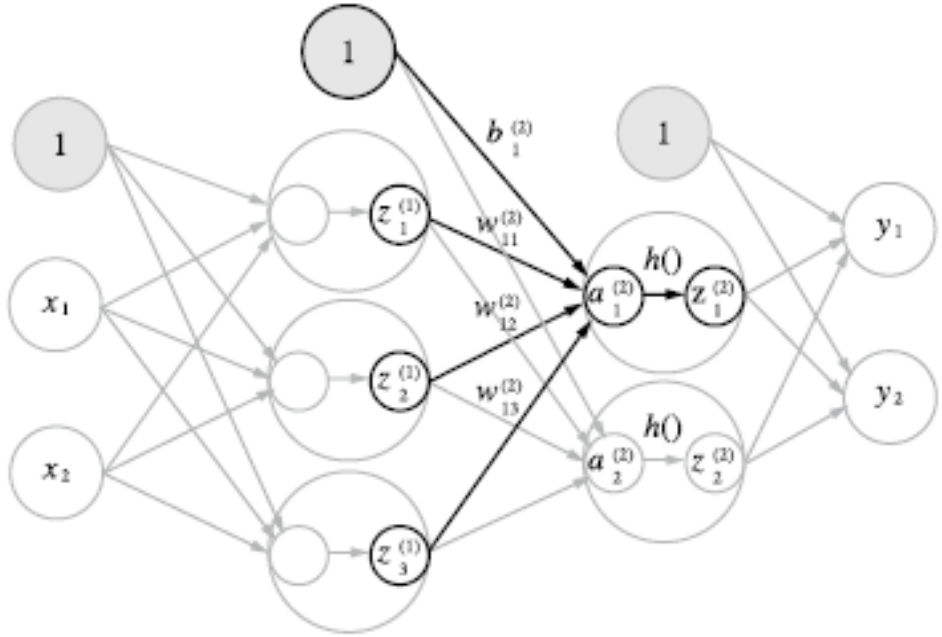
그림 3-18 입력층에서 1층으로의 신호 전달



3.4.2 각 층의 신호 전달 구현하기

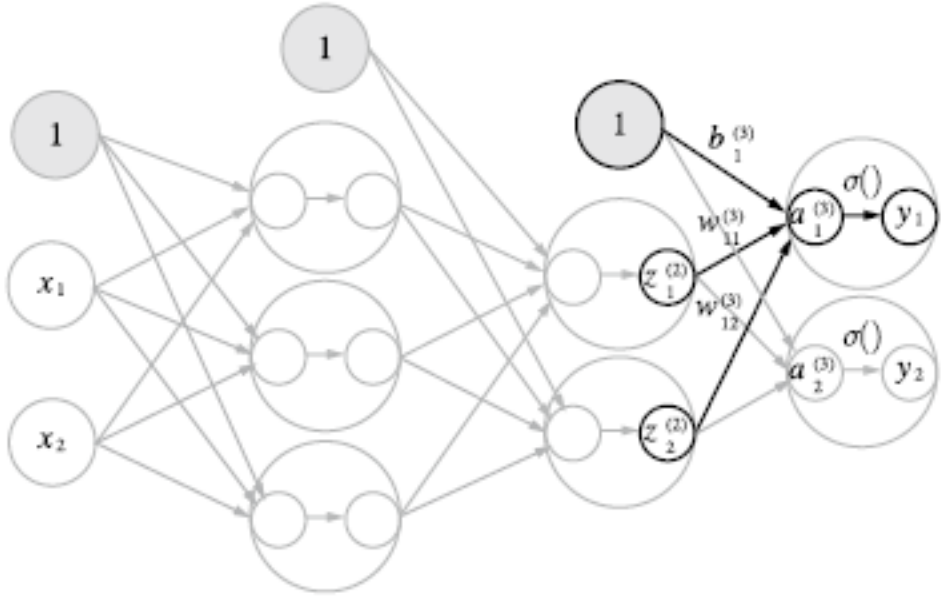
```
Z1 = sigmoid(A1)

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

그림 3-19 1층에서 2층으로의 신호 전달



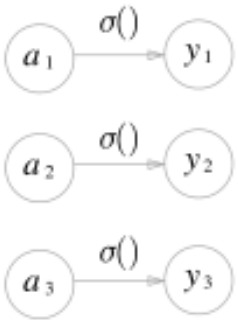

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

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



3.5.1 항등 함수와 소프트맥스 함수 구현하기

그림 3-21 항등 함수



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

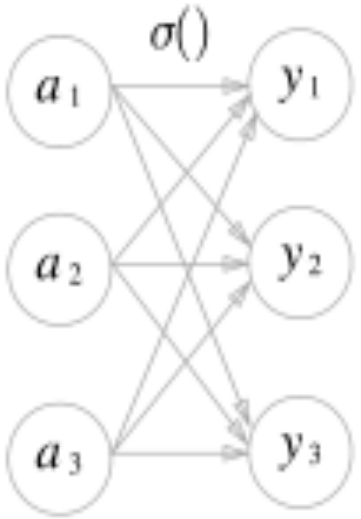
$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \qquad [식\ 3.10]$$

그림 3-21 항등 함수

exp (x)는 e x 을 뜻하는 지수 함수 exponential function(e 는 자연상수). N 은 출력층의 뉴런 수,y_k 는 그중 k 번째 출력.

[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

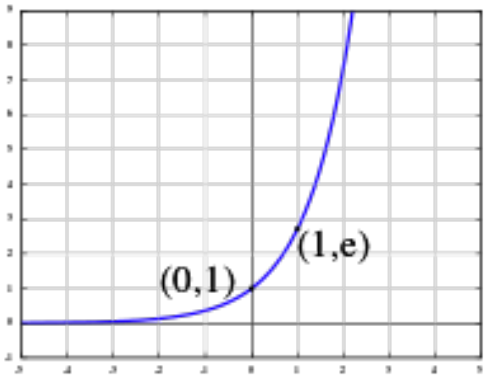
Softmax 함수는 classification
Identity (항등) 함수는 regression 에 사용

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 . 10의 변형
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.

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
```

Softmax 함수 구현하기

보는 바와 같이 소프트맥스 함수의 출력은 0 에서 1 . 0 사이의 실수
또, 소프트맥스 함수출력의 총합은 1.
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

```
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,)
```

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.

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.

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)))
```

MNIST 데이터셋을 가지고 추론을 수행하는 신경망을 구현.
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

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

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

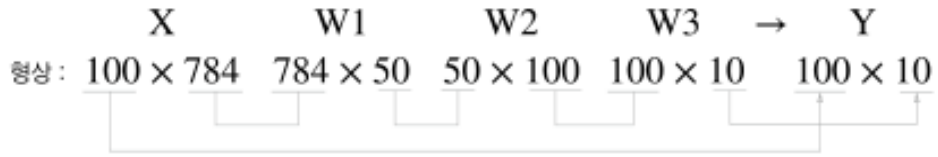


그림 3-27 배치 처리를 위한 배열들의 형상 추이

[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

배치처리로 계산 속도 향상
- 대부분의 라이브러리 및 CPU/GPU가 대용량 데이터 처리에 호율적
- IO 부하를 줄일 수 있음 (IO 가 병목)

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

batch_size=100
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])
```

axis=1
100x10 의 배열 중 1번째 차원을 구성하는 각 원소에서 (1번째 차원을 축으로)