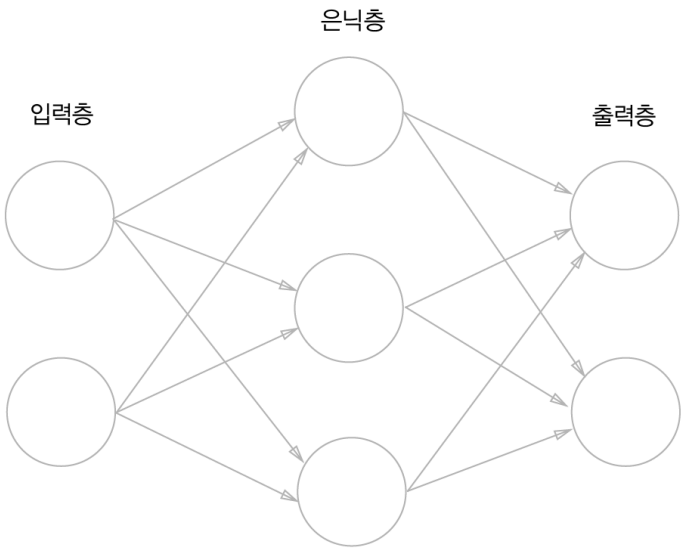
Chapter 03 신경망

3.1 퍼셉트론에서 신경망으로

퍼셉트론과 다른 점을 중심으로 신경망의 구조를 살펴보자

3.1.1 신경망의 예



가장 왼쪽 줄을 입력층, 맨 오른쪽 줄을 출력층, 중간 줄을 은닉층 이라 한다. 은닉층의 뉴런은 사람 눈에는 보이지 않는다.

> 가중치를 갖는 층은 2개뿐이기 때문에 '2층 신경망'이라고 한다.

3.1.2 퍼셉트론 복습

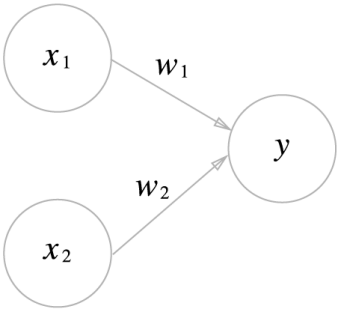


그림 3-2

x_1 과 x_2라는 두 신호를 입력받아 y를 출력하는 퍼셉트론이다. 이 퍼셉트론을 수식으로 나타내면 아래와 식 3.1과 같다.

$$y = \begin{cases} 0 & (b + w_1 x_1 + w_2 x_2 \leq 0) \\ 1 & (b + w_1 x_1 + w_2 x_2 > 0) \end{cases}$$

식 3.1

b는 편향, w_1과 w_2는 가중치를 나타내는 매개변수이다.

여기에 편향을 그래프에 명시한다면 아래 그림과 같이 된다.

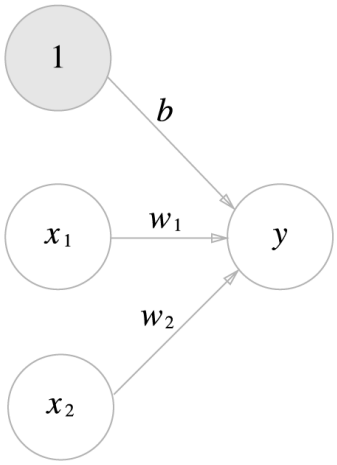


그림 3-3 편향을 명시한 퍼셉트론

[식 3,1]을 더 간결한 형태로 출력하려면 아래와 같이 표현할 수 있다.

$$y = h(b + w_1 x_1 + w_2 x_2)$$

[식 3.2]

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

[식 3.3]

3.1.3 활성화 함수의 등장

이처럼 입력 신호의 총합을 출력 신호로 변환하는 함수를 일반적으로 활성화 함수**(activation function)** 이라고 한다. 활성화 함수는 입력 신호의 총합이 활성화를 일으키는지 여부를 정한다.

식 3.2를 다시 작성하면 아래와 같다.

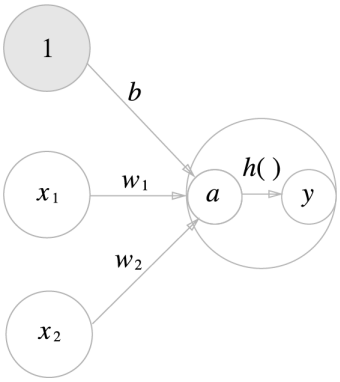
$$a = b + w_1 x_1 + w_2 x_2$$

[식 3.4]

$$y = h(a)$$

[식 3.5]

이를 그림으로 나타내면 다음과 같다.



[그림 3.4] 활성화 함수의 처리 과정

> **WARNING**
> 일반적으로 단순 퍼셉트론은 단층 네트워크에서 계단 함수(임계값을 경계로 출력이 바뀌는 함수)를 활성화 함수로 사용한 모델을 가리키고 다층 퍼셉트론은 신경망(여러 층으로 구성되고 시그모이드 함수 등의 매끈한 활성화 함수를 사용하는 네트워크)를 가리킨다.

3.2 활성화 함수

활성화 함수는 임계값을 경계로 출력이 바뀌는데, 이런 함수를 계단 함수(Step function)이라고 한다. 퍼셉트론에서는 활성화 함수로 계단 함수를 이용한다. 여기서 활성화 함수를 계단 함수에서 다른 함수로 변경하는 것이 신경망의 시작이다.

3.2.1 시그모이드 함수

$$h(x) = \frac{1}{1+\exp(-x)}$$

퍼셉트론과 신경망의 주된 차이는 활성화 함수 뿐이다. 그 외에 뉴런이 여러 층으로 이어지는 구조와 신호를 전달하는 방법은 기본적으로 앞에서 살펴본 퍼셉트론과 같다.

3.2.2 계단 함수 구현하기

In [2]:

```
import numpy as np
import pickle
```

In [4]:

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

In [5]:

```
def step_function(x):
    y = x > 0
    return y.astype(np.int) # astype을 통해서 Boolean type을 astype으로 바꿔준다.
```

In [6]:

```
import numpy as np
x = np.array([-1.0, 1.0, 2.0])
x
```

Out[6]:

```
array([-1.,  1.,  2.])
```

In [7]:

```
y = x > 0
y
```

Out[7]:

```
array([False,  True,  True])
```

In [8]:

```
y = y.astype(np.int)
y
```

Out[8]:

```
array([0, 1, 1])
```

In [9]:

```
step_function(np.array([1.0, 2.0]))
```

Out[9]:

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

3.2.3 계단 함수의 그래프

In [10]:

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

In [11]:

```
def step_function(x):
    return np.array(x>0, dtype=int)
```

In [12]:

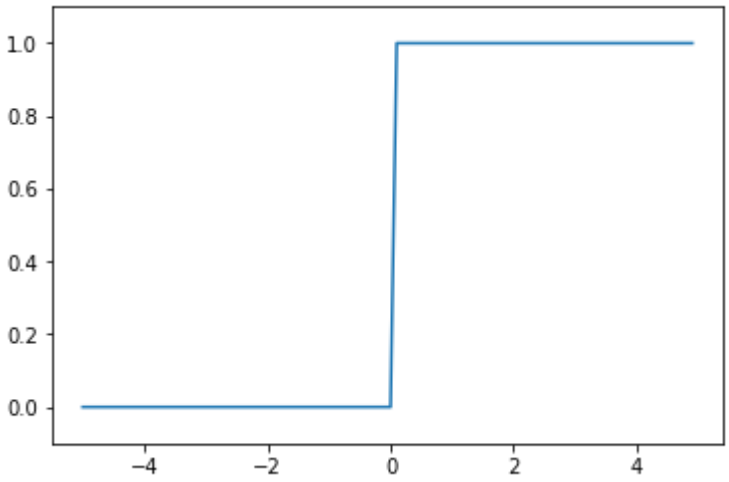
```
x = np.arange(-5.0, 5.0, 0.1)
```

In [13]:

```
y = step_function(x)
```

In [14]:

```
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # y축의 범위 지정
plt.show()
```



0을 경계로 계단처럼 출력이 0에서 1로 바뀜.

3.2.4 시그모이드 함수 구현하기

In [15]:

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

In [16]:

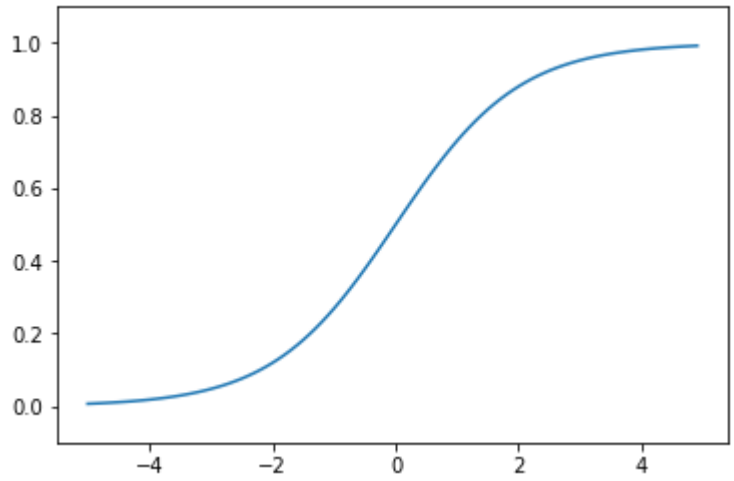
```
x = np.array([-1.0, 1.0, 2.0])
sigmoid(x)
```

Out[16]:

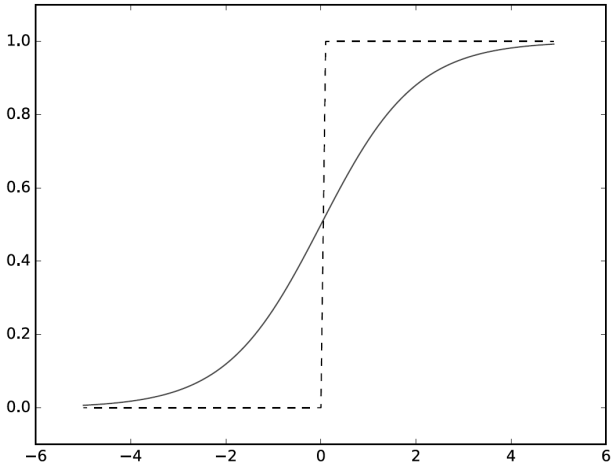
```
array([0.26894142, 0.73105858, 0.88079708])
```

In [17]:

```
x = np.arange(-5, 5, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # y축 범위 지정
plt.show()
```



3.2.5 시그모이드 함수와 계단 함수 비교



[그림 3.8] 계단 함수(점선)와 시그모이드 함수(실선)

시그모이드 함수는 부드러운 곡선이며 입력에 따라 출력이 연속적으로 변화한다. 한편 계단 함수는 0을 경계로 출력이 갑자기 바뀐다.

계단 함수가 0과 1중 하나의 값만 돌려주는 반면 시그모이드 함수는 실수(0.731..., 0.880... 등)을 돌려준다. 다시 말해 퍼셉트론에선느 뉴런 사이에 0 혹은 1이 흘렀다면, 신경망에서는 연속적인 실수가 흐른다.

3.2.6 비선형 함수

계단 함수와 시그모이드 함수 모두 비선형 함수이다. 신경망에서는 활성화 함수로 비선형 함수를 사용해야 하며, 선형 함수를 이용하면 신경망의 층을 깊게 하는 의미가 없다. 여기서 선형 함수는 f(x) = ax + b 여기서 a와 b는 상수. 층을 쌓는 혜택을 얻고 싶다면 활성화 함수로는 반드시 비선형 함수를 사용해야 함

$$y(x) = h(h(h(x))) \text{ where } h(x) = cx$$

$$y(x) = c * c * c * x$$

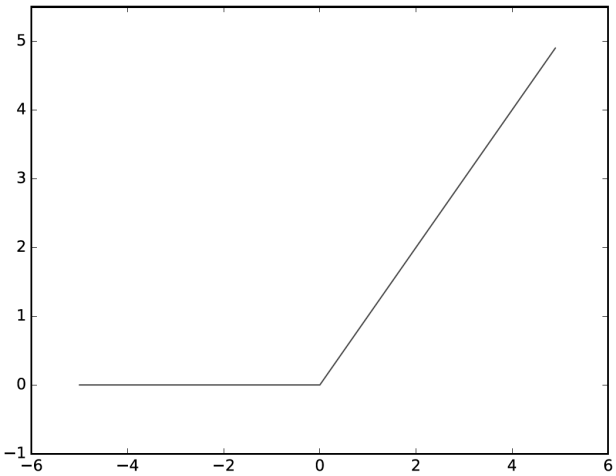
$$y(x) = ax, a = c^3$$

> **NOTE_** 함수란 어떤 값을 입력하면 그에 따른 값을 돌려주는 '변환기'이다. 이 변환기에 무언가 입력했을 때 출력이 입력의 상수배만큼 변하는 함수를 선형 함수 라고 한다. 수식으로는 f(x) = ax + b 이고 이때 a와 b는 상수이다. 그래서 선형 함수는 곧은 1개의 직선이 된다. 한편, 비선형 함수는 그대로 문자 그대로 '선형이 아닌 함수'이다. 즉 직선 1개로는 그릴 수 없는 함수를 뜻한다.

3.2.7 ReLU 함수

ReLU(Rectified Linear Unit)는 입력이 0을 넘으면 그 입력을 그대로 출력하고, 0 이하이면 0을 출력하는 함수

$$h(x) = x \ (x > 0)$$
$$h(x) = 0 \ (x \leq 0)$$



[그림 3.9] ReLU 함수의 그래프

In [18]:

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

3.3 다차원 배열의 계산

3.3.1 다차원 배열

배열의 차원 수는 np.dim() 함수로 확인 가능. 형상은 인스턴스 변수인 shape으로 알 수 있음.

In [19]:

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

```
[[1 2]
 [3 4]
 [5 6]]
```

In [20]:

```
np.ndim(B)
```

Out[20]:

```
2
```

In [21]:

```
B.shape
```

Out[21]:

```
(3, 2)
```

3.3.2 행렬의 내적(행렬 곱)

In [22]:

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

Out[22]:

```
(2, 2)
```

In [23]:

```
B = np.array([[5,6], [7,8]])
B.shape
```

Out[23]:

```
(2, 2)
```

In [24]:

```
np.dot(A, B)
```

Out[24]:

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

3.3.2 행렬의 내적(행렬 곱)

In [25]:

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

Out[25]:

```
(2, 2)
```

In [26]:

```
B = np.array([[5,6], [7,8]])
B.shape
```

Out[26]:

```
(2, 2)
```

In [27]:

```
np.dot(A, B)
```

Out[27]:

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

In [28]:

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

Out[28]:

```
(3, 2)
```

In [29]:

```
B = np.array([7, 8])
B.shape
```

Out[29]:

```
(2,)
```

In [30]:

```
np.dot(A, B)
```

Out[30]:

```
array([23, 53, 83])
```

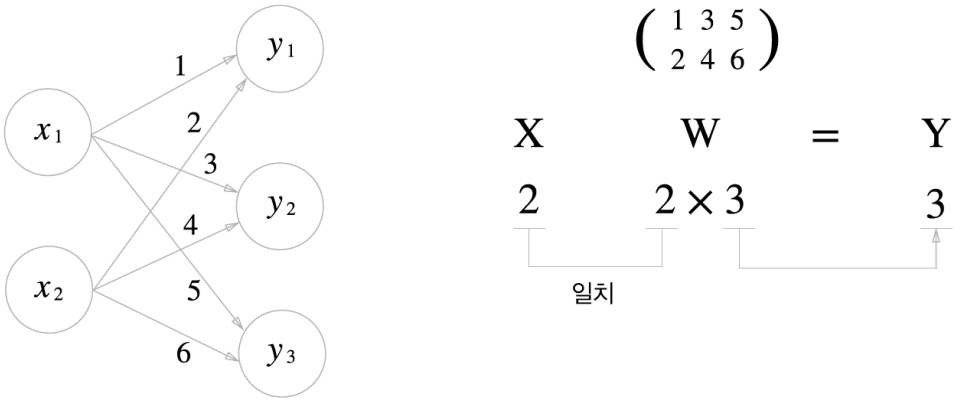
In [31]:

```
np.ndim(B)
```

Out[31]:

```
1
```

3.3.3 신경망의 내적



[그림 3.14] 행렬의 곱으로 신경망의 계산을 수행한다.

X, W, Y의 형상을 주의해서 보자. 특히 **X**와 **W**의 대응하는 차원의 원소 수가 같아야 한다.

In [32]:

```
X = np.array([1, 2])
X.shape
```

Out[32]:

```
(2,)
```

In [33]:

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

```
[[1 3 5]
 [2 4 6]]
```

In [34]:

```
W.shape
```

Out[34]:

```
(2, 3)
```

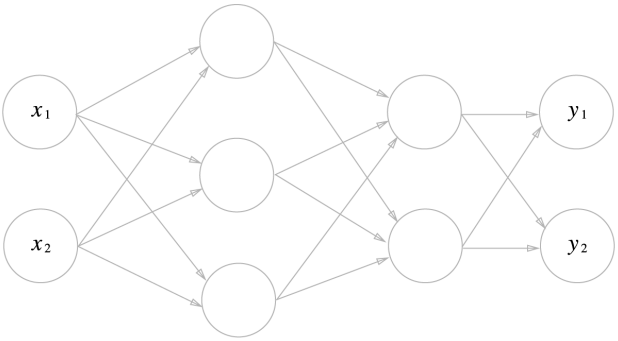
In [35]:

```
Y = np.dot(X, W)
print (Y)
```

```
[ 5 11 17]
```

다차원 배열의 내적을 구해주는 np.dot함수를 사용하면 이처럼 단변에 결과 Y를 계산할 수 있다. Y의 원소가 100개든, 1,000개든 한 번의 연산으로 계산할 수 있다. 만약 np.dot을 사용하지 않으면 Y의 원소를 하나하나 따져봐야 한다. 그래서 내적을 통해 한꺼번에 계산해 주는 기능은 신경망을 구현할 때 매우 중요하다.

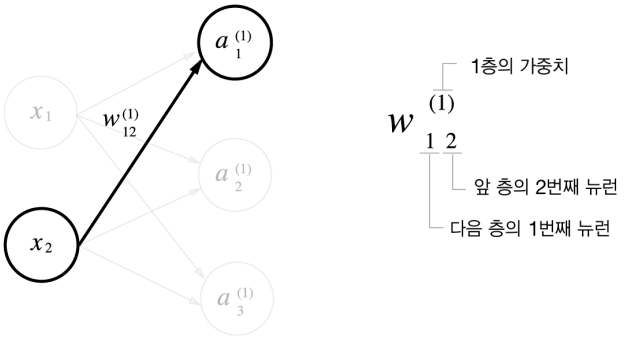
3.4 3층 신경망 구현하기



[그림 3.15] 3층 신경망: 입력층(0층)은 2개, 첫 번째 은닉층(1층)은 3개, 두 번째 은닉층(2층)은 2개, 출력층(3층)은 2개의 뉴런으로 구성된다.

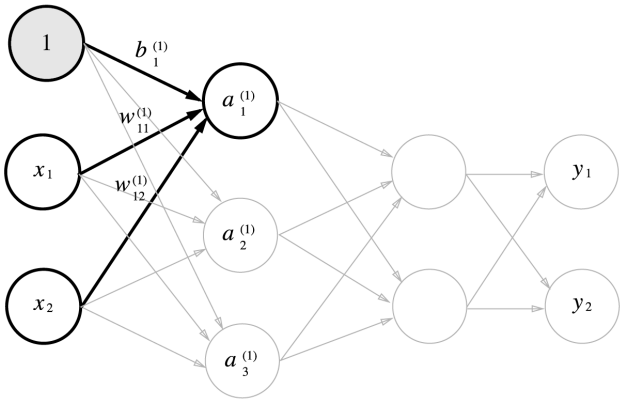
3.4.1 표기법 설명

$w_{12}^{(1)}$ 여기서 (1)은 1층의 가중치. 1은 다음 층의 1번째 뉴런, 2는 앞 층의 2번째 뉴런을 뜻함



[그림 3.16] 중요 표기

3.4.2 각 층의 신호 전달 구현하기



[그림 3.17] 입력층에서 1층으로 신호 전달

편향을 뜻하는 뉴런 1은 오른쪽 아래 인덱스가 한 ㅏ밖에 없다. 이는 앞 층의 편향 뉴런(뉴런1)이 하나 밖에 없기 때문이다.

$a_1^{(1)}$ 을 수식으로 나타내 보면 다음과 같다. $a_1^{(1)}$은 가중치를 곱한 신호 두 개와 편향을 합해서 다음과 같이 계산한다.

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}$$

[식 3.8]

행렬의 내적을 이용하면 1층의 '가중치 부분'을 다음 식처럼 간소화할 수 있다.

$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

[식 3.9]

각 $A^{(1)}$, **X**, $B^{(1)}$, $W^{(1)}$는 다음과 같다.

$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}$$

$$X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \end{pmatrix}$$

$$B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

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

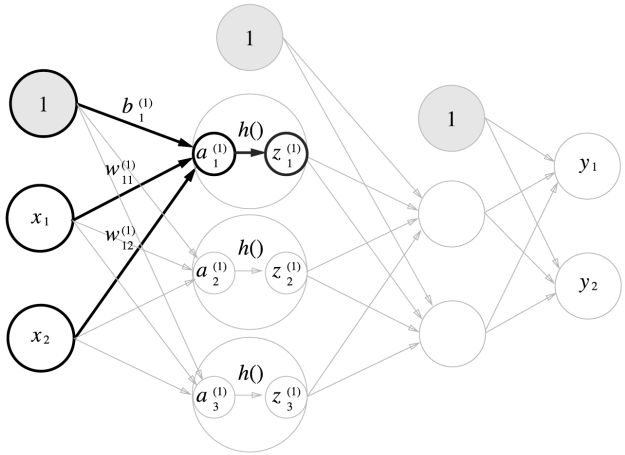
넘파이의 다차원 배열을 사용해서 [식 3.9]를 구현해 보자. (입력 신호, 가중치, 편향은 적당한 값으로 설정한다.)

In [36]:

```
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)
print(X.shape)
print(B1.shape)

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

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

이 활성화 함수의 처리를 그림으로 나타내면 아래 그림 3-18처럼 된다.



[그림 3.18] 입력층에서 1층으로 신호 전달

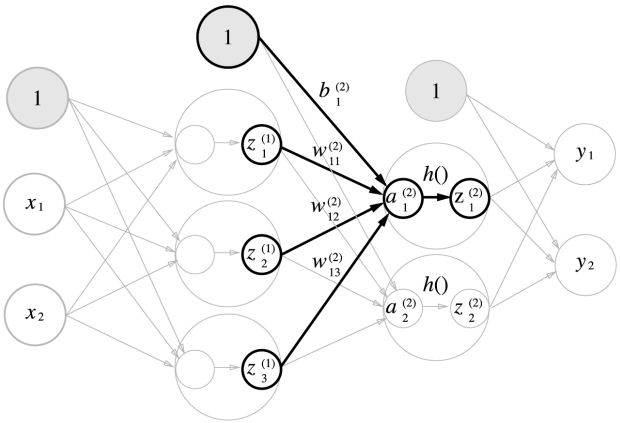
그림 3.18과 같이 은닉층에서의 가중치 합을 a로 표기하고 활성화 함수 h()로 변환된 신호를 z로 표기한다. 여기에서는 활성화 함수로 시그모이드 함수를 사용하기로 한다.

In [38]:

```
Z1 = sigmoid(A1)

print (A1)
print (Z1)
```

```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```

시그모이드는 넘파이 배열을 받아 같은 수의 원소로 구성된 넘파이 배열을 반환한다. 1층에서 2층으로 가는 과정과 그 구현을 살펴보자.



[그림 3.19] 1층에서 2층으로 신호 전달

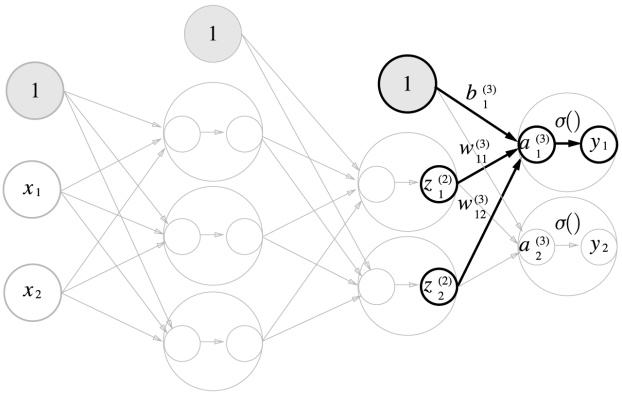
In [41]:

```
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)
print (W2.shape)
print (B2.shape)

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

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

이 구현은 1층의 출력 Z1이 2층의 입력이 된다는 점만 제외하면 조금 전의 구현과 똑같다. 이처럼 넘파이 배열을 사용하면서 층 사이의 신호 전달을 쉽게 구현할 수 있다.

마지막으로 2층에서 출력층으로 신호 전달을 한다. 출력층의 구현도 그동안의 구현과 거의 같다. 활성화 함수만 지금까지의 은닉층과 다르다.



[그림 3.20] 2층에서 출력층으로 신호 전달

In [42]:

```
def identity_function(x):
    return x
```

In [43]:

```
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)
```

항등 함수는 입력을 그대로 출력하는 함수이다. 그래서 이 예에서는 identity_function()을 굳이 정의할 필요가 없지만, 그동안의 흐름과 통일하기 위해 이렇게 구현하였다. 그림 3-20 에서는 출력층의 활성화 함수를 $\sigma()$로 표시하여 은닉층의 활성화 함수 h()와는 다름을 명시하였다.

NOTE

> 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 맞게 정한다. 예를 들어 회귀에는 항등 함수를, 2클래스 분류에는 시그모이드 함수를, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적이다.

3.4.3 구현 정리

In [56]:

```
def identify_function(x):
    return x
```

In [65]:

```
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
```

In [66]:

```
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 = identify_function(a3)

    return y
```

In [67]:

```
network = init_network()
```

In [68]:

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

In [69]:

```
print(y)
```

```
[ 0.31682708  0.69627909]
```

3.5 출력층 설계하기

항등 함수는 입력을 그대로 출력하는 함수. 여기서는 이를 출력층의 활성화 함수로 이용. 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 따라 달라질 수 있음. 회귀에서는 항등 함수를, 2클래스 분류에는 시그모이드 함수를, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적

3.5.1 항등 함수와 소프트맥스 함수 구현하기

항등 함수 는 입력을 그대로 출력한다. 이는 입력과 출력이 항상 같다는 뜻이다.
한편, 분류에서 사용하는 소프트맥스 함수(softmax function)은 다음과 같다.

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

[식 3.10]

exp(x)는 e^x 을 뜻하는 지수 함수.(e는 자연상수) n은 출력층의 뉴런 수, y_k는 그중 k번째 출력임을 뜻함. 소프트맥스의 출력은 모든 입력 신호로부터 화살표를 받는다. 출력층의 각 뉴런이 모든 입력 신호에서 영향을 받기 때문이다.

소프트맥스 함수 구현

In [78]:

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

    return y
```

In [79]:

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

In [80]:

```
softmax(a)
```

Out[80]:

```
array([ 0.01821127,  0.24519181,  0.73659691])
```

3.5.2 소프트맥스 함수 구현 시 주의점

지수 함수는 쉽게 아주 큰 값을 내뱉기 때문에, 위의 소프트맥스 함수에서는 오버플로 문제가 발생한다. 가령 e^{10} 은 2만 이상이고 e^{100}은 0이 40개가 넘는 큰 값이 된다. 이런 큰 값끼리 나눗셈을 하면 결과 수치가 불안정해진다.

In [87]:

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

    return y
```

여기서 C는 어떤 값을 대입해도 결과는 바뀌지 않지만, 주로 오버플로우를 막기 위해서는 입력 신호 중 최대값을 이용하는 것이 일반적이다.

In [83]:

```
a = np.array([1010, 1000, 990])
np.exp(a) / np.sum(np.exp(a))
```

```
/root/.pyenv/versions/3.5.3/envs/hrc/lib/python3.5/site-packages/ipyke
rnel_launcher.py:2: RuntimeWarning: overflow encountered in exp

/root/.pyenv/versions/3.5.3/envs/hrc/lib/python3.5/site-packages/ipyke
rnel_launcher.py:2: RuntimeWarning: invalid value encountered in true_
divide
```

Out[83]:

```
array([ nan,  nan,  nan])
```

In [84]:

```
c = np.max(a)
```

In [85]:

```
a - c
```

Out[85]:

```
array([  0, -10, -20])
```

In [86]:

```
np.exp(a - c) / np.sum(np.exp(a-c))
```

Out[86]:

```
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

3.5.3 소프트맥스 함수의 특징

softmax() 함수를 이용하면 신경망의 출력은 다음과 같이 계산할 수 있습니다.

In [88]:

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

```
[ 0.01821127  0.24519181  0.73659691]
```

In [89]:

```
np.sum(y)
```

Out[89]:

```
1.0
```

소프트맥스 함수의 출력은 0에서 1사이의 실수이고 출력 총합의 값은 1이다. 이 성질 더분에 소프트맥스 함수의 출력을

'확률'로 해석할 수 있다. 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 해석할 수 있다. 또한 소프트맥스 함수는 단조증가 함소이기 때문에 원소의 대소 관계는 변하지 않는다.

신경망을 이용한 분류에서는 일반적으로 가장 큰 출력을 내는 뉴런에 해당하는 클래스로만 인식한다. 따라서 신경망으로 분류를 할 때에는 출력층의 소프트맥스 함수를 생략해도 된다.

3.5.4 출력층의 뉴런 수 정하기

출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정해야 함. 분류에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적

3.6 손글씨 숫자 인식

이미 학습된 매개변수를 사용하여 학습 과정은 생략하고, 추론 과정만 구현한다. 이 추론 과정을 신경망의 순전파(forward propagation)이라고 한다.

3.6.1 MNIST 데이터셋

MNIST의 이미지 데이터는 28 X 28 크기의 회색조 이미지(1채널)이며, 각 픽셀은 0에서 255까지의 값을 취한다. 각 이미지에는 7, 2, 1와 같이 그 이미지가 실제 의미하는 숫자가 레이블로 붙어 있다.

파이썬에는 pickle(피클)이라는 기능이 있다. 실행 중에 특정 객체를 파일로 저장하는 기능인데, 저장해둔 pickle 파일을 로드하면 실행 당시의 객체를 즉시 복원할 수 있다.

In [91]:

```
import sys, os
```

In [97]:

```
sys.path.append('/root/entropylab/hrc/deep_learning_from_scratch_org/deep-learning-
```

In [98]:

```
from dataset.mnist import load_mnist
```

In [99]:

```
# 처음은 시간이 몇분 걸림
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize = False)
```

```
Converting train-images-idx3-ubyte.gz to NumPy Array ...
Done
Converting train-labels-idx1-ubyte.gz to NumPy Array ...
Done
Converting t10k-images-idx3-ubyte.gz to NumPy Array ...
Done
Converting t10k-labels-idx1-ubyte.gz to NumPy Array ...
Done
Creating pickle file ...
Done!
```

In [100]:

```
print(x_train.shape)
```

```
(60000, 784)
```

In [101]:

```
print(t_train.shape)
```

```
(60000,)
```

In [102]:

```
print(x_test.shape)
```

```
(10000, 784)
```

In [103]:

```
print(t_test.shape)
```

```
(10000,)
```

인수로는 normalize, flatten, one_hot_label 세가지가 있다.

첫번째 인수인 normalize는 0~1사이의 값으로 정규화 여부를 결정한다.
두번째 인수인 flatten은 입력 이미지를 평탄하게 한다.
세번째 인수인 one_hot_label은 레이블을 원-핫 인코딩 형태로 저장할지를 정한다. 여기서 False면 '7'이나 '2'와 같이 숫자 형태의 레이블을 저장하고, True 일때에는 레이블을 원-핫 인코딩하여 저장한다.

In [117]:

```
import sys, os
sys.path.append('/root/entropylab/hrc/deep_learning_from_scratch_org/deep-learning-
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))
    return pil_img

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

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

print (img.shape)
img = img.reshape(28, 28)
print (img.shape)

img_show(img)
```

```
5
(784,)
(28, 28)
```

Out[117]:

3.6.2 신경망의 추론 처리

이미지 크기가 28 X 28 = 784이고 0에서 9까지의 숫자를 구분하는 문제이기 때문에, 입력층 뉴런이 784개, 출력층 뉴런이 10개가 된다. 은닉층은 총 두개이고, 첫번째 은닉층에서 50개의 뉴런을 두번째 은닉층에서는 100개의 뉴런을 임의로 배치한다.

In [119]:

```
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
```

In [120]:

```
def init_network():
    with open ('../deep_learning_from_scratch_org/deep-learning-from-scratch/ch03/s
        network = pickle.load(f)

    return network
```

In [129]:

```
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
```

정확도 평가하기

In [126]:

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

In [127]:

```
network = init_network()
```

In [130]:

```
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p = np.argmax(y)
    if p == t[i]:
        accuracy_cnt +=1

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

```
Accuracy:0.9352
```

load_mnist 함수의 인수인 normalize를 True로 설정한다. 이에 0~255 범위인 각 픽셀의 값이 0.0~1.0 범위로 전환된다. (단순히 픽셀의 값을 255로 나눔)
이렇게 데이터를 특정 범위로 변환하는 처리를 정규화(normalization)이라 하고, 신경망의 입력 데이터에 특정 변환을 가하는 것을 전처리(pre-processing)라고 한다. 여기서는 입력 이미지 데이터에 대한 전처리 작업으로 정규화를 수행한 셈이다.

여기서는 픽셀의 값을 255로 나누는 단순한 정규화를 수행했지만, 데이터 전체의 분포를 고려해 데이터들을 0을 중심으로 분포하도록 이동시키거나 데이터의 확산 범위를 제한하는 정규화를 수행한다. 그 이외에도 전체 데이터를 균일하게 분포시키는 데이터 백색화(whitening)를 진행한다.

3.6.3 배치 처리

하나로 묶은 입력 데이터를 배치(batch)라 한다. 배치는 두가지 이점이 있다. 첫째는 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 최적화되어 있기 때문이다. 둘째는 커다란 신경망에서는 데이터 전송이 병목으로 작

용하는 경우가 있는데, 배치 처리를 함으로서 버스에 주는 부하를 줄일 수 있다. (느린 I/O를 통해 데이터를 읽는 횟수가 줄어, 빠른 CPU나 GPU로 순수 계산을 수행하는 비율이 높아진다. 즉 배치 처리를 수행함으로써 큰 배열로 이루어진 계산을 하게 되는데, 컴퓨터에서는 큰 배열을 한꺼번에 계산하는 것이 분할된 작은 배열을 여러 번 계산하는 것보다 빠르다.

배치 처리 구현

In [144]:

```
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)))
```

```
Accuracy:0.9352
```

argmax()는 최대값의 인덱스를 가지고 온다. axis=1은 100 X 10의 배열 중 1번째 차원을 구성하는 각 원소에서(1번째 차원을 축으로) 최대값의 인덱스를 찾도록 한 것이다. (인덱스가 0부터 시작하니 0번째 차원이 가장 처음 차원이다.)

In [3]:

```
x = np.array([[0.1, 0.8, 0.1], [0.3, 0.1, 0.6], \
              [0.2, 0.5, 0.3], [0.8, 0.1, 0.1]])
print(x)
```

```
[[0.1 0.8 0.1]
 [0.3 0.1 0.6]
 [0.2 0.5 0.3]
 [0.8 0.1 0.1]]
```

In [9]:

```
y = np.argmax(x, axis=0)
y
```

Out[9]:

```
array([3, 0, 1])
```

In [8]:

```
y = np.argmax(x, axis=1)
y
```

Out[8]:

```
array([1, 2, 1, 0])
```

== 연산자를 사용해 넘파이 배열끼리 비교하여 True/False로 구성된 bool 배열을 만들고, 이 결과 배열에서 True가 몇 개인지 센다.

In [157]:

```
y = np.array([1, 2, 1, 0])
t = np.array([1, 2, 0, 0])
print (y == t)
```

```
[ True  True False  True]
```

In [158]:

```
np.sum(y == t)
```

Out[158]:

```
3
```

이번 장에서 배운 내용

- 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU함수 같은 매끄럽게 변화하는 함수를 이용한다.
- 넘파이의 다차원 배열을 잘 사용하면 신경망을 효율적으로 구현할 수 있다.
- 기계학습 문제는 크게 회귀와 분류로 나눌 수 있다.
- 출력층의 활성화 함수로는 회귀에서는 주로 항등 함수를, 분류에서는 주로 소프트맥스 함수를 이용한다.
- 분류에서는 출력층의 뉴런 수를 분류하려는 클래스 수와 같게 설정한다.
- 입력 데이터를 묶은 것을 배치라 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 더 빠르게 얻을 수 있다.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: