

3. 신경망

앞 장에서 배운 퍼셉트론 관련해서 좋은 소식과 나쁜 소식이 있다.

좋은 소식은 퍼셉트론으로 복잡한 함수도 표현할 수 있다는 것이다. 그 예로 컴퓨터가 수행하는 복잡한 처리도 퍼셉트론으로 (이론 상) 표현할 수 있음을 앞 장에서 설명했다.

나쁜 소식은 가중치를 설정하는 작업(원하는 결과를 출력하도록 가중치 값을 적절히 정하는 작업)은 여전히 사람이 수동적으로 한다는 것이다. 앞 장에서는 AND, OR 게이트의 진리표를 보면서 우리 인간이 적절한 가중치 값을 정했다.

신경망은 이 나쁜 소식을 해결해준다. 데이터가 자동으로 가중치 매개변수의 적절한 값을 자동으로 학습해준다는 말이다. 이는 신경망의 아주 중요한 성질이다. 이번 장에서는 신경망의 개요를 설명하고, 신경망이 입력 데이터를 식별하는 처리 과정에 대해 자세히 알아보도록 하자.

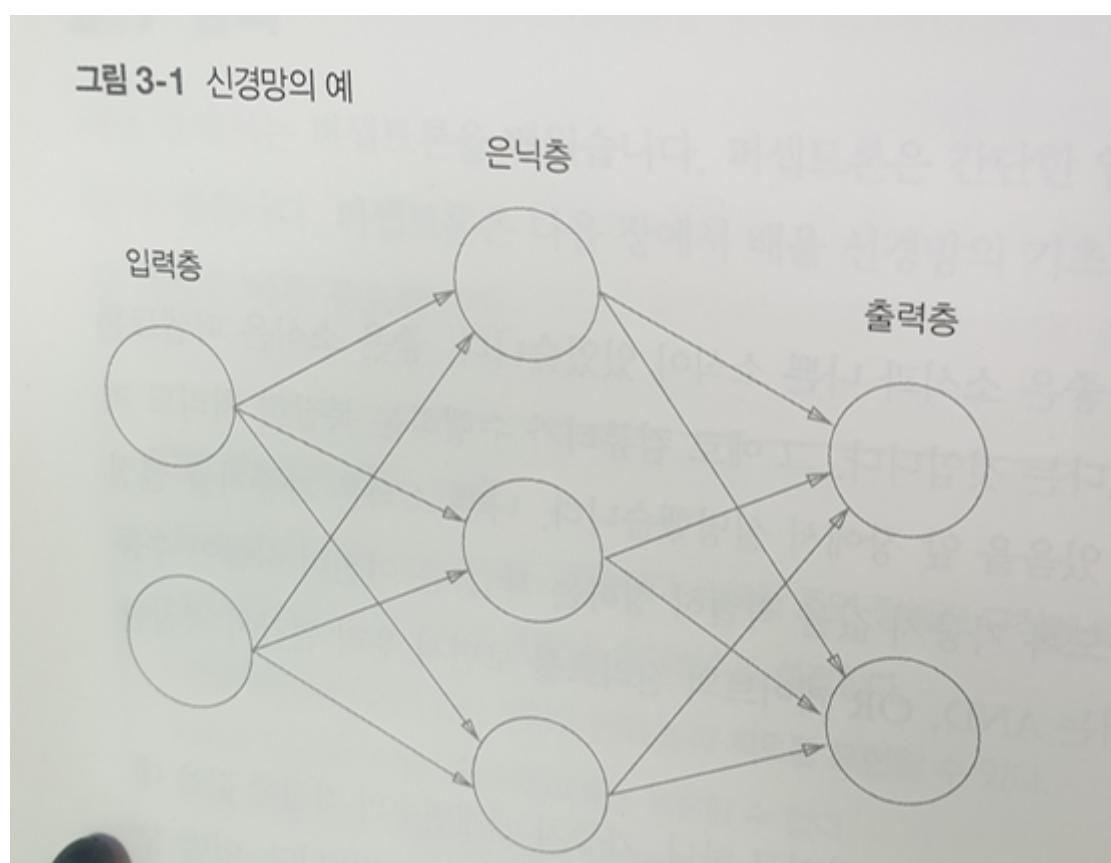
4장에서는 데이터에서 가중치 매개변수 값을 학습하는 방법을 다룬다.

3.1 퍼셉트론에서 신경망으로

신경망은 앞 장에서 설명한 퍼셉트론과 공통점이 많다. 이번 절에서는 퍼셉트론과 다른 점을 중심으로 신경망의 구조를 설명한다.

3.1.1 신경망의 예

아래 그림은 신경망을 나타낸 것이다. 여기에서 가장 왼쪽 줄을 **입력층**, 맨 오른쪽 줄을 **출력층**, 중간 줄을 **은닉층**이라고 한다. 은닉층의 뉴런은 (입력층이나 출력층과 달리) 사람 눈에는 보이지 않는다. 그래서 '은닉' 인 것이다. 또한 이 책에서는 입력층에서 출력층의 방향으로 차례로 0층, 1층, 2층이라 하겠다. (층 번호를 0부터 시작하는 이유는 파이썬 배열의 인덱스도 0부터 시작하여, 나중에 구현할 때 짹짓기 편하기 때문이다.) 아래 그림에서, 0층이 입력층, 1층이 은닉층, 2층이 출력층이 된다.



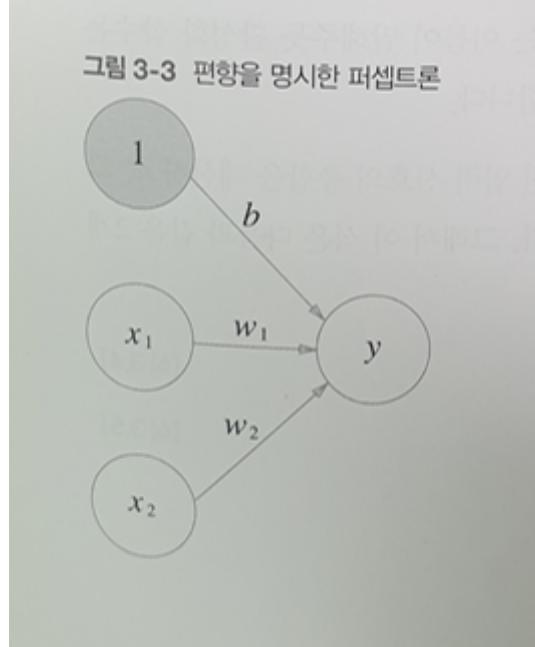
WARNING_ 신경망은 모두 3층으로 구성된다. 하지만 가중치를 갖는 층은 2개 뿐이기에 '2층 신경망'이라고 한다.

위 그림은 앞 장에서 배운 퍼셉트론과 특별히 달라 보이지 않는다. 실제로 뉴런이 연결되는 방식은 앞 장의 퍼셉트론에서 달라진 것이 없다. 신경망에서는 신호를 어떻게 전달할까?

3.1.2 퍼셉트론 복습

신경망의 신호 전달 방법을 보기 전에 퍼셉트론을 살짝 복습해보도록 한다.

아래 그림은 편향을 명시한 퍼셉트론이다.



위 그림에서 가중치가 b 이고 입력이 1인 뉴런이 추가 됐다. 이 퍼셉트론의 동작은 $x_1, x_2, 1$ 이라는 3개의 신호가 뉴런에 입력되어, 각 신호에 가중치를 곱한 후, 다음 뉴런에 전달 된다. 다음 뉴런에서는 이 신호들의 값을 더하여, 그 합이 0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력한다.

참고로, 편향의 입력 신호는 항상 1이기 때문에 그림에서는 해당 뉴런을 회색으로 채워 다른 뉴런과 구별했다.

그럼 식(3.1)

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

여기서 b 는 편향을 나타내는 매개변수로, 뉴런이 얼마나 쉽게 활성화되느냐를 제어합니다. 한편, w_1 과 w_2 는 각 신호의 가중치를 나타내는 매개변수로, 각 신호의 영향력을 제어한다.

그런데 위 그림의 네트워크에는 편향 b 가 보이지 않는다. 여기에 편향을 명시한다면 위 그림과 같이 나타낼 수 있다.

위 그림에서는 가중치가 b 이고 입력이 1인 뉴런이 추가되었다. 이 퍼셉트론의 동작은 $x_1, x_2, 1$ 이라는 3개의 신호가 뉴런에 입력되어, 각 신호에 가중치를 곱한 후, 다음 뉴런에 전달 된다.

다음 뉴런에서는 이 신호들의 값을 더하여, 그 합이 0이 넘으면 1을 출력하고, 그렇지 않으면 0을 출력한다. 참고로, 편향의 입력 신호는 항상 1이기 때문에 그림에서는 해당 뉴런을 회색으로 채워 다른 뉴런과 구별했다.

그럼 식(3.1)을 더 간결한 형태로 다시 작성해보자. 이를 위해 조건 분기의 동작 (0을 넘으면 1을 출력하고 그렇지 않으면 0 출력)을 하나의 함수로 나타낸다. 이 함수를 $h(x)$ 라고 하면 다음 식으로 나타낼 수 있다.

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

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

$$1 \quad (x > 0) \quad \text{식 (3.3)}$$

식 3.2는 입력 신호의 총합이 $h(x)$ 라는 함수를 거쳐 변환되어, 그 변환된 값이 y 의 출력이 됨을 보여준다. 그리고 위 식(3.3)의 $h(x)$ 함수는 입력이 0을 넘으면 1을 돌려주고 그렇지 않으면 0을 돌려준다.

3.1.3 활성화 함수의 등장

조금 전 $h(x)$ 라는 함수가 등장했는데, 이처럼 입력 신호의 총합을 출력 신호로 변환하는 함수를 일반적으로 **활성화 함수**라고 한다.

그럼 식 3.2를 다시 써보자. 식 3.2는 가중치가 곱해진 입력 신호의 총합을 계산하고, 그 합을 활성화 함수에 입력해 결과를 내는 2단계로 처리된다. 이에 이 식은 다음과 같은 2개의 식으로 나눌 수 있다.

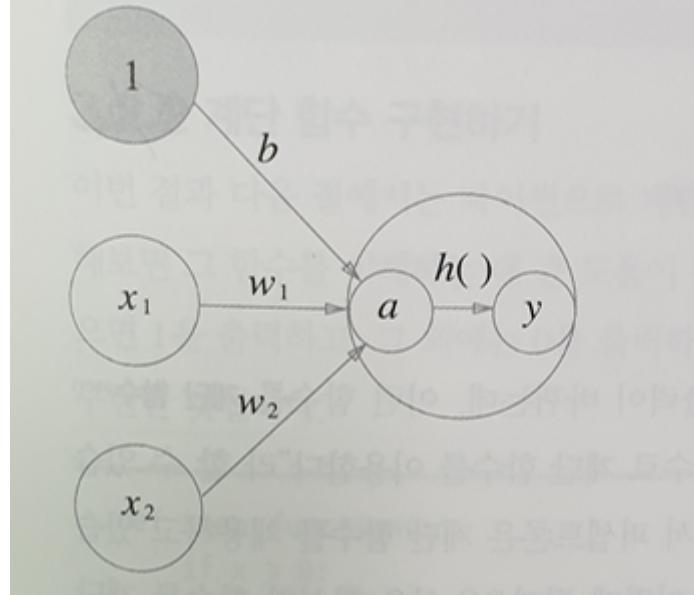
$$a = b + w_1x_1 + w_2x_2 \quad \text{식 (3.4)}$$

$$y = h(a) \quad \text{식 (3.5)}$$

식 3.4는 가중치가 달린 입력 신호와 편향의 총합을 계산하고, 이를 a 라고 한다. 그리고 식 3.5는 a 를 함수 $h()$ 에 넣어 y 를 출력하는 흐름이다.

지금까지와 같이 뉴런을 큰 원으로 그려보면 다음그림과 같이 표현할 수 있다.

그림 3-4 활성화 함수의 처리 과정

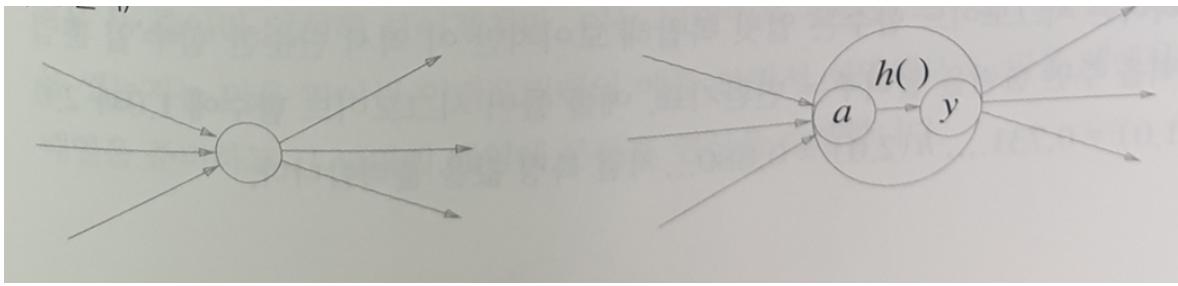


보다시피 위 그림에서는 기존 뉴런의 원을 키우고, 그 안에 활성화 함수의 처리 과정을 명시적으로 그렸다. 즉, 가중치 신호를 조합한 결과가 a 라는 노드가 되고, 활성화 함수 $h()$ 를 통하여 y 라는 노드로 변환되는 과정이 분명하게 나타나 있다. 참고로 이 책에서는 **뉴런과 노드**라는 용어를 같은 의미로 사용한다.

방금 a 와 y 의 원을 노드라고 했는데, 이는 지금까지 뉴런이라고 한 것과 같은 의미이다.

뉴런을 그릴 때 보통은 지금까지와 마찬가지로 아래 그림의 왼쪽처럼 뉴런을 하나의 원으로 그린다.

그리고 신경망의 동작을 더 명확히 드러내고자 할 때는 오른쪽 그림처럼 활성화 처리 과정을 명시하기도 한다.



그럼 계속해서 활성화 함수에 대해 알아보고자 한다. 이 활성화 함수가 퍼셉트론에서 신경망으로 가기 위한 길잡이다.

warning 이 책에서는 퍼셉트론이라는 말이 가리키는 알고리즘을 엄밀히 통일하지는 않았다. 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

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases} \quad (3.3)$$

식 3.3과 같은 활성화 함수는 임계값을 경계로 출력이 바뀌는데, 이런 함수를 **계단 함수**라 한다. 이에 '퍼셉트론에서는 활성화 함수로 계단 함수로 이용한다'라 할 수 있다. 즉, 활성화 함수로 쓸 수 있는 여러 후보 중에서 퍼셉트론은 계단 함수를 이용하고 있다. 그렇다면 계단 함수 이외의 함수를 사용하면 어떻게 될까? 실은 활성화 함수를 계단 함수에서 다른 함수로 변경하는 것이 신경망의 세계로 나아가는 열쇠이다.

3.2.1 시그모이드 함수

다음은 신경마에서 자주 이용하는 활성화 함수인 **시그모이드 함수**를 나타낸 식이다.

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

식 3.6에서 나타나는 시그모이드 함수는 얼핏 복잡해 보이지만 이 역시 단순한 '함수'일 뿐이다. 함수는 입력을 주면 출력을 돌려주는 변환기 개념이다. 예를 들어 시그모이드 함수에 1.0과 2.0을 입력하면 $h(1.0) = 0.731 \dots, h(2.0) = 0.880 \dots$ 처럼 특정 값을 출력한다.

신경망에서는 활성화함수로 시그모이드 함수를 이용하여 신호를 변환하고, 그 변환된 신호를 다음 뉴런에 전달한다. 사실 앞 장에서 본 퍼셉트론과 앞으로 볼 신경망의 주된 차이는 이 활성화 함수 뿐이다. 그 외에 뉴런이 여러 층으로 이어지는 구조와 신호를 전달하는 방법은 기본적으로 앞으로 살펴볼 퍼셉트론과 같다. 그러면 활성화 함수로 이용되는 시그모이드 함수를 계단 함수와 비교하면서 자세히 살펴보겠다.

3.2.2 계단 함수 구현하기

이번 절과 다음 절에서는 파이썬으로 계단 함수를 그려보겠다(함수의 형태로 눈으로 확인해보면 그 함수를 이해하는데 큰 도움이 된다). 계단 함수는 식 3.3과 같이 입력이 0을 넘으면 1을 출력하고, 그 외에는 0을 출력하는 함수이다. 다음은 이러한 계단 함수를 단순하게 구현한 것이다.

```
In [1]: def step_function(x):
....:     if x>0:
....:         return 1
....:     else:
....:         return 0
....:
```

이 구현은 단순하고 쉽지만, 인수 x 는 실수(부동소수점)만 받아들인다. 즉, $\text{step_function}(3.0)$ 은 되지만 넘파이 배열을 인수로 넣을 수는 없다. 가령 $\text{step_function}(\text{np.array}[1.0, 2.0])$ 는 안된다.

나는 앞으로를 위해 numpy array도 지원하도록 수정하고 싶다. 그러기 위해서는 다음과 같은 구현을 생각해야 한다.

```
In [2]: def step_function(x):
....:     y = x>0
....:     return y.astype(np.int)
```

겨우 두 줄이라 엉성해 보이겠지만, 이는 넘파이의 편리한 트릭을 사용한 덕분이다. 어떤 트릭을 썼는지는 다음 파이썬 인터프리터의 예를 보면서 설명하자. 다음 예에서는 x 라는 넘파이 배열을 준비하고 그 넘파이 배열에 부등호 연산을 수행한다.

```
In [4]: import numpy as np

In [5]: x=np.array([-1.0,1.0,2.0])

In [6]: x
Out[6]: array([-1.,  1.,  2.])

In [7]: y=x>0

In [8]: y
Out[8]: array([False,  True,  True])
```

넘파이 배열에 부등호 연산을 수행하면 배열의 원소 각각에 부등호 연산을 수행한 bool 배열이 생성된다. 이 예에서는 배열 x 의 원소 각각이 0보다 크면 True로, 0 이하면 False로 변환한 새로운 배열 y 가 생성된다.

이 y 는 bool 배열이다. 그런데 우리가 원하는 계단 함수는 0이나 1의 'int형'을 출력하는 함수다. 이에 배열 y 의 원소를 bool에서 int형으로 바꿔준다.

```
In [9]: y = y.astype(np.int)

In [10]: y
Out[10]: array([0, 1, 1])
```

이처럼 넘파이 배열의 자료형을 변환할 때는 `astype()` 메서드를 이용한다. 원하는 자료형 (이 예에서는 `np.int`)을 인수로 지정하면 된다. 그리고 파이썬에서는 bool을 int형으로 변환하면 True는 1, False는 0으로 변환한다. 이상이 계단 함수 구현에서 사용한 넘파이의 '트릭'이 있다.

3.2.3 계단 함수의 그래프

이제 앞에서 정의한 계단함수를 그래프로 그려보자. 이를 위해 `matplotlib` 라이브러리를 사용한다.

```
In [13]: def step_function(x):
```

```

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

In [14]: x = np.arange(-5.0, 5.0, 0.1)

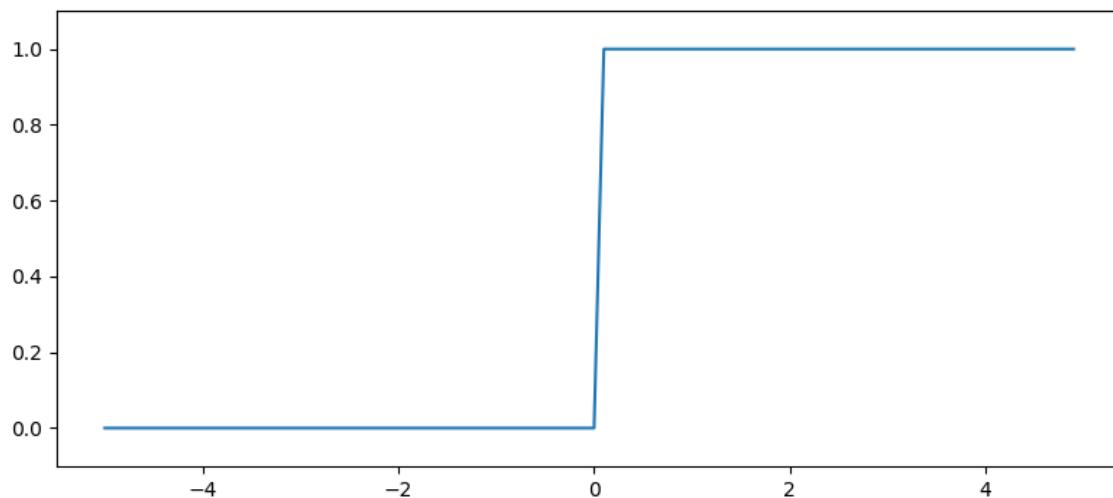
In [15]: y = step_function(x)

In [18]: %matplotlib
Using matplotlib backend: Qt5Agg

In [20]: import matplotlib.pyplot as plt

In [21]: plt.plot(x,y)
Out[21]: [

```



`np.arange(-5.0,5.0,0.1)`은 -5.0에서 5.0 전까지 0.1 간격의 넘파이 배열을 생성한다. 즉 $[-5.0, -4.9, -4.8, \dots, 4.9]$ 를 생성한다. `step_function()`은 인수로 받은 넘파이 배열의 원소 각각을 인수로 계단 함수 실행해, 그 결과를 다시 배열로 만들어 돌려준다. 이 x,y 배열을 그래프로 그리면 그에 대한 결과는 위와 같다.

3.2.4 시그모이드 함수 구현하기

이어서 시그모이드 함수를 구현하자.

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

위 식 3.6의 시그모이드 함수는 파이썬으로 다음과 같이 작성할 수 있다.

```

In [25]: def sigmoid(x):
....:     return 1/(1+np.exp(-x))

```

여기서 `np.exp(-x)`는 `exp(-x)` 수식에 해당한다. 이 구현에서 특별히 어려운 건 없고, 인수 x가 넘파이 배열이어도 올바른 결과가 나온다는 정도만 기억하자. 실제로 넘파이 배열을 제대로 처리하는지 살펴보자.

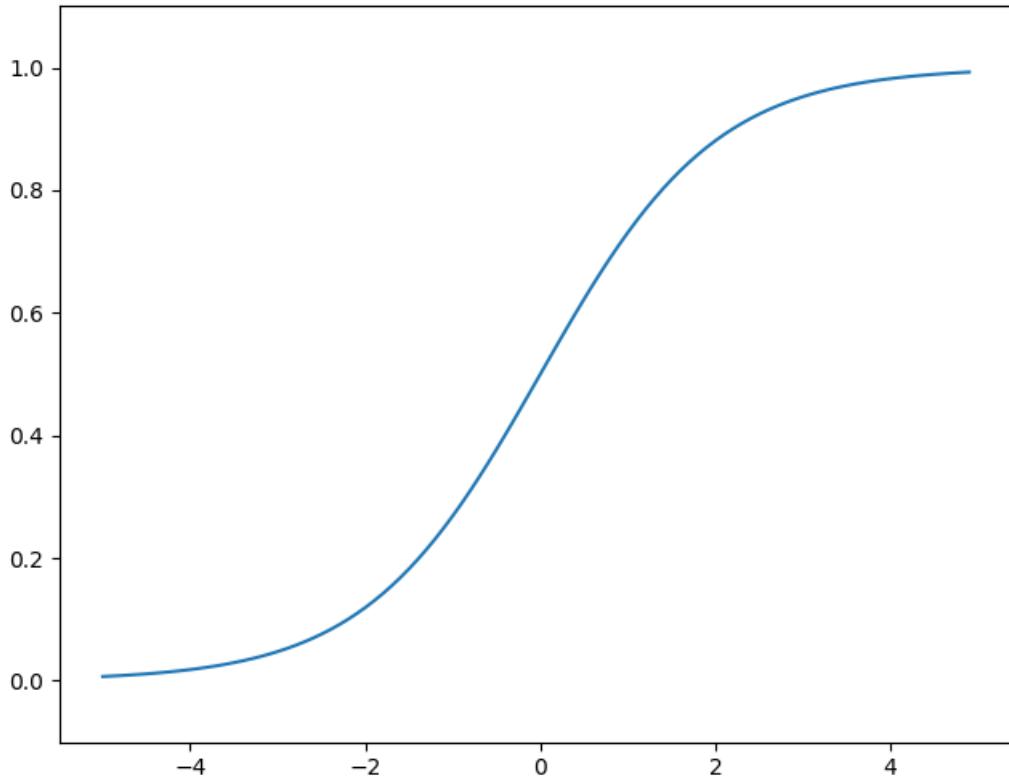
```
In [26]: x = np.array([-1.0, 1.0, 2.0])  
  
In [27]: sigmoid(x)  
Out[27]: array([0.26894142, 0.73105858, 0.88079708])
```

이 함수가 넘파이 배열도 훌륭히 처리해줄 수 있는 비밀은 넘파이의 브로드캐스트 기능이란, 넘파이 배열과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산으로 바꿔 수행하는 것이다. 복습 겸 구체적인 예를 보자.

```
In [28]: t = np.array([1.0, 2.0, 3.0])  
  
In [29]: 1.0 + t  
Out[29]: array([2., 3., 4.])  
  
In [30]: 1.0/t  
Out[30]: array([1.          , 0.5          , 0.33333333])
```

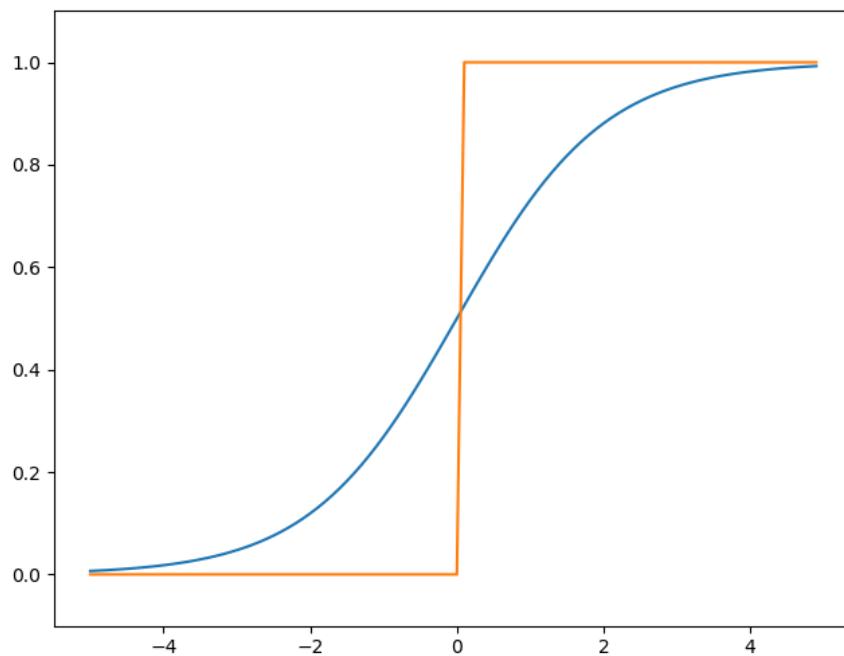
이 예에서는 스칼라값 1.0과 넘파이 배열 사이에서 수치 연산 (+와 /)을 해봤다. 결과적으로 스칼라값과 넘파이 배열의 각 원소 사이에서 연산이 이뤄지고, 연산 결과가 넘파이 배열로 출력됐다. 앞에서 구현한 sigmoid 함수에서도 $\text{np.exp}(-x)$ 가 넘파이 배열을 반환하기에 $1/(1+\text{np.exp}(-x))$ 도 넘파이 배열의 각 원소에 연산을 수행한 결과를 출력한다. 그럼 시그모이드 함수를 그래프로 그려보자. 그래프를 그리는 코드는 앞 절의 계단 함수 그리기 코드와 거의 같다.

```
In [31]: x = np.arange(-5.0, 5.0, 0.1)  
  
In [32]: y = sigmoid(x)  
  
In [33]: plt.plot(x,y)  
Out[33]: [<matplotlib.lines.Line2D at 0x16f19c70248>]  
  
In [34]: plt.plot(x,y)  
Out[34]: [<matplotlib.lines.Line2D at 0x16f19cafec8>]  
  
In [35]: plt.ylim(-0.1,1.1)  
Out[35]: (-0.1, 1.1)
```



3.2.5 시그모이드 함수와 계단 함수 비교

시그모이드 함수와 계단 함수를 비교해보자. 이 두 함수를 다음 그림에 그려봤다. 무엇이 다르고, 또 공통되는 성질은 무엇일까?



위 그림을 보고 가장 먼저 느껴볼 것은 매끄러움이다. 시그모이드 함수는 부드러운 곡선이며 입력에 따라 추력이 연속적으로 변화한다. 한편 계단 함수는 0을 경계로, 출력이 갑자기 바뀐다. 시그모이드 함수의 이 매끈함이 신경망 학습에서 아주 중요한 역할을 한다.

(역시 매끈함과 관련되지만) 계단 함수가 0과 1 중 하나의 값만 돌려주는 반면 시그모이드 함수는 실수 (0.731 ..., 0.880... 등)를 돌려준다는 점도 다르다. 다시 말해 퍼셉트론에서는 뉴런 사이에 0 혹은 1이 흘렀다면, 신경망에서는 연속적인 실수가 흐른다.

비유하자면 계단 함수는 '시시오도시'이고 시그모이드 함수는 '물레방아'와 비슷하다. 계단 함수는 시시오도시처럼 물을 쏟아내거나 쏟아내지 않는 (0 또는 1) 두 가지 움직임을 보여주며, 시그모이드 함수는 물레방아처럼 흘러온 물의 양에 비례해 흐르는 물의 양을 조절한다.

두 함수의 공통점도 찾아보자. 두 함수는 매끄러움이라는 점에서는 다르지만, 위 그림을 큰 관점에서 보면 둘은 같은 모양을 하고 있다. 둘 다 입력이 작을 때의 출력은 0에 가깝고 (혹은 0이고), 입력이 커지면 출력이 1에 가까워지는 (혹은 1이 되는) 구조다. 즉, 계단 함수와 시그모이드 함수는 입력이 중요하면 큰 값을 출력하고 입력이 중요하지 않으면 작은 값을 출력한다.

그리고 입력값의 관계없이 출력값은 무조건 0~1의 값이다.

3.2.6 비선형함수

계단 함수와 시그모이드 함수의 공통점은 그 밖에도 있다. 중요한 공통점으로, 둘 모두는 **비선형 함수**이다. 시그모이드 함수는 곡선, 계단 함수는 계단처럼 구부러진 직선으로 나타나며 동시에 비선형 함수로 분류된다.

Note_ 활성화 함수를 설명할 때 비선형 함수와 선형 함수라는 용어가 자주 등장한다. 함수란 어떤 값을 입력하면 그에 따른 값을 돌려주는 '변환기'이다. 이 변환기에 무언가 입력했을 때 출력이 입력의 상수배 만큼 변하는 함수를 **선형 함수**라고 한다. 수식으로는 $f(x) = ax + b$ 이고, 이때 a 와 b 는 상수이다. 그래서 선형 함수는 곧은 1개의 직선이 된다. 한편, **비선형 함수**는 문자 그대로 '선형이 아닌' 함수이다. 즉, 직선 1개로는 그릴 수 없는 함수를 말한다.

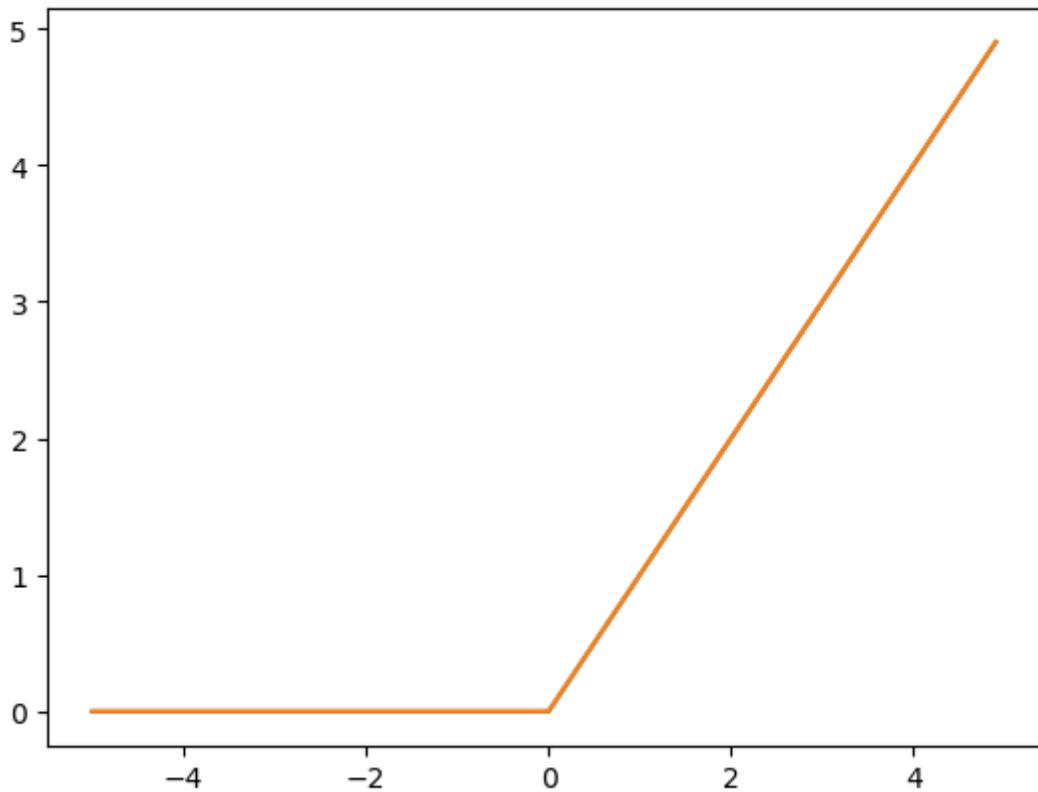
신경망에서는 활성화 함수로 비선형 함수를 사용해야 한다. 달리 말하면 선형 함수를 사용해서는 안된다. 왜? 그 이유는 선형함수를 이용하면 신경망의 층을 깊게 하는 의미가 없기 때문이다.

선형 함수의 문제는 층을 아무리 깊게 해도 '은닉층이 없는 네트워크'로도 똑같은 기능을 할 수 있다는 데 있다. 구체적으로 (약간직관적으로) 설명해주는 간단한 예를 생각해봤다. 선형 함수인 $h(x) = cx$ 를 활성화 함수로 사용한 3층 네트워크를 떠올려 봐라. 이를 식으로 나타내면 $y(x) = h(h(h(x)))$ 가 된다. 이 계산은 $y(x) = c * c * c * x$ 처럼 곱셈을 세번 수행하지만, 실은 $y(x) = ax$ 와 똑같은 식이다. $a = c^3$ 이라고 만 하면 끝이다. 즉, 은닉층이 없는 네트워크로 표현할 수 있다. 이 예처럼 선형함수를 이용해서는 여러 층으로 구성하는 이점을 살릴 수 없다. 그래서 층을 쌓는 혜택을 얻고 싶다면 활성화 함수로는 반드시 비선형 함수를 사용해야 한다.

3.2.7 ReLU 함수

지금까지 활성화 함수로서 계단 함수와 시그모이드 함수를 소개했다. 시그모이드 함수는 신경망 분야에서 오래전부터 이용해왔으나, 최근에는 **ReLU** 함수를 주로 이용한다.

ReLU는 입력이 0을 넘으면 그 입력을 그대로 출력하고, 0이하면 0을 출력하는 함수다.



수식으로는 다음과 같이 쓸 수 있다.

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ x & (x > 0) \end{cases} \quad (3.3)$$

그래프와 수식에서 보듯 ReLU는 간단한 함수이다.

```
In [46]: In [13]: def relu(x):  
....:     ....:         return np.maximum(0, x)  
....:     ....:  
....:     ....:  
....: In [14]: x = np.arange(-5.0, 5.0, 0.1)  
....:     ....:  
....: In [15]: y = relu(x)
```

여기서는 넘파이의 maximum 함수를 사용했다. maximum은 두 입력 중 큰 값을 선택해 반환하는 함수다. 이번 장에서는 앞으로 시그모이드 함수를 활성화 함수로 사용하지만 이 책 후반부는 주로 ReLU 함수를 사용한다.

3.3 다차원 배열의 계산

넘파이의 다차원 배열을 사용한 계산법을 숙달하면 신경망을 효율적으로 구현할 수 있다. 이에 이번 절에서는 넘파이의 다차원 배열 계산에 대해서 설명한 뒤 신경망을 구현해보겠다.

3.3.1 다차원 배열

다차원 배열도 그 기본은 '숫자의 집합'이다. 숫자가 한 줄로 늘어선 것이나 직사각형으로 늘어놓은 것, 3 차원으로 늘어놓은 것이나 (더 일반화한) N차원으로 나열하는 것을 통틀어 다차원 배열이라고 한다. 그럼 넘파이를 사용해서 다차원 배열을 작성해보겠다. 우선은 지금까지 봐온 1차원 배열이다.

```
In [48]: import numpy as np  
  
In [49]: A = np.array([1,2,3,4])  
  
In [50]: print(A)  
[1 2 3 4]  
  
In [51]: np.ndim(A)  
Out[51]: 1  
  
In [53]: A.shape  
Out[53]: (4,)  
  
In [54]: A.shape[0]  
Out[54]: 4
```

이와 같이 배열의 차원 수는 np.dim() 함수로 확인할 수 있다. 또, 배열의 형상은 인스턴스 변수인 shape으로 알 수 있다. 이 예에서 A는 1차원 배열이고 원소 4개로 구성되었다. 한 가지, A.shape이 튜플을 반환하는 것에 주의해라. 이는 1차원배열이라도 다차원 배열일 때와 통일된 형태로 결과를 반환하기 위함이다. 예를 들어 2차원 배열일 때는 (4,3), 3차원 배열일 때는 (4,3,2) 같은 튜플을 반환한다. 그래서 1차원 배열일 때도 결과를 튜플로 반환한다. 다음은 2차원 배열을 작성해보자.

```
In [55]: B = np.array([[1,2],[3,4],[5,6]])  
  
In [56]: print(B)  
[[1 2]  
 [3 4]  
 [5 6]]  
  
In [58]: np.ndim(B)  
Out[58]: 2  
  
In [59]: B.shape  
Out[59]: (3, 2)
```

여기에서는 '3x2 배열'인 B를 작성했다. 3x2 배열은 처음 차원에는 원소가 3개, 다음 차원에는 원소가 2개 있다는 의미이다. 이때 처음 차원은 0번째 차원, 다음 차원은 1번째 차원에 대응한다(파이썬의 인덱스는 0부터 시작이다).

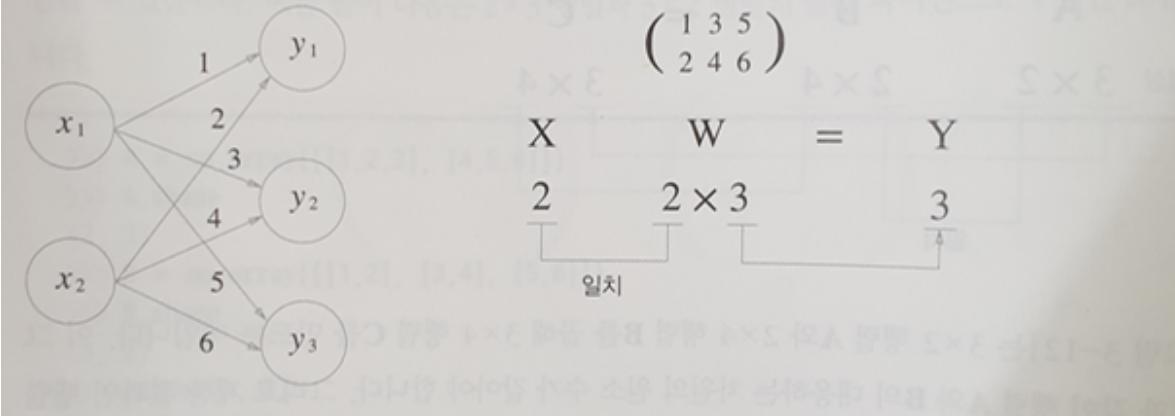
3.3.2 행렬의 곱

파이썬에서 행렬 곱은 np.dot()으로 연산한다.

3.3.3 신경망에서의 행렬 곱

그럼 넘파이 행렬을 써서 신경망을 구현해보겠다. 이번 예에서는 다음 그림의 간단한 신경망을 가정해보겠다. 이 신경망은 편향과 활성화 함수를 생략하고 가중치만 갖는다.

그림 3-14 행렬의 곱으로 신경망의 계산을 수행한다.



이 구현에서도 X, W, Y 의 형상을 주의해서 보라. 특히 X 와 W 의 대응하는 차원의 원소 수가 같아야 한다는 것을 잊지 마라.

```
In [60]: x = np.array([1,2])
In [61]: x.shape
Out[61]: (2,)

In [62]: w = np.array([[1,3,5],[2,4,6]])

In [63]: print(w)
[[1 3 5]
 [2 4 6]]

In [64]: w.shape
Out[64]: (2, 3)

In [65]: y = np.dot(x,w)

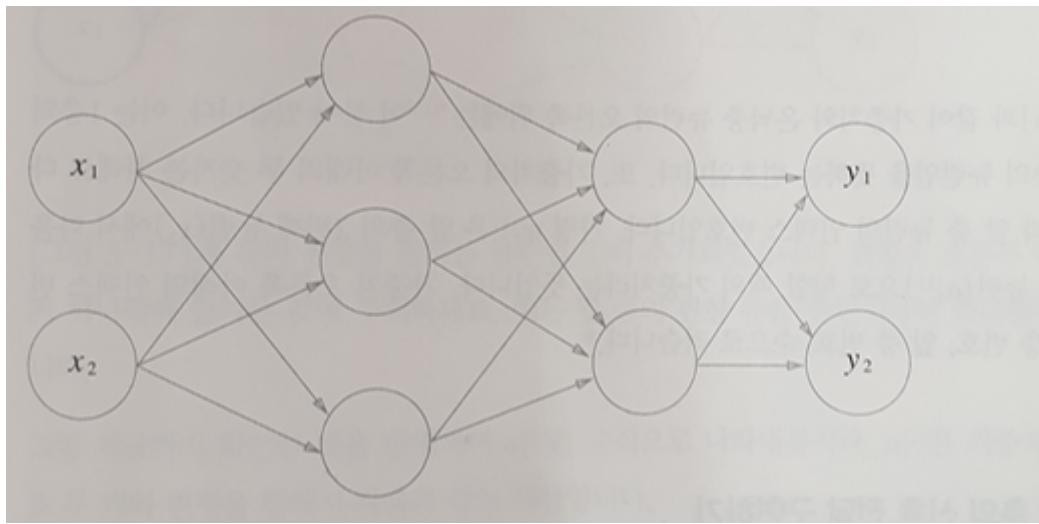
In [66]: print(y)
[ 5 11 17]
```

다차원 배열의 스칼라곱을 구해주는 `np.dot` 함수를 사용하면 이처럼 단번에 결과 Y 를 계산할 수 있다. Y 의 원소가 100개든 1,000개든 한 번의 연산으로 계산할 수 있다. 만약 `np.dot`을 사용하지 않으면 Y 의 원소를 하나하나 따져봐야 한다. (또는 `for`문을 사용해서 계산 해야하는데, 굉장히 귀찮겠다..)

그래서 행렬의 곱으로 한꺼번에 계산해주는 기능은 신경망을 구현할 때 매우 중요하다.

3.4 3층 신경망 구현하기

이제 더 그럴싸한 신경망을 구현해보자. 다음 그림은 3층 신경망에서 수행되는, 입력부터 출력까지의 처리(순방향 처리)를 구현해보겠다. 이를 위해 앞에서 설명한 넘파이의 다차원 배열을 사용하겠다. 넘파이 배열을 잘 쓰면 아주 적은 코드만으로 신경망의 순방향 처리를 완성할 수 있다.



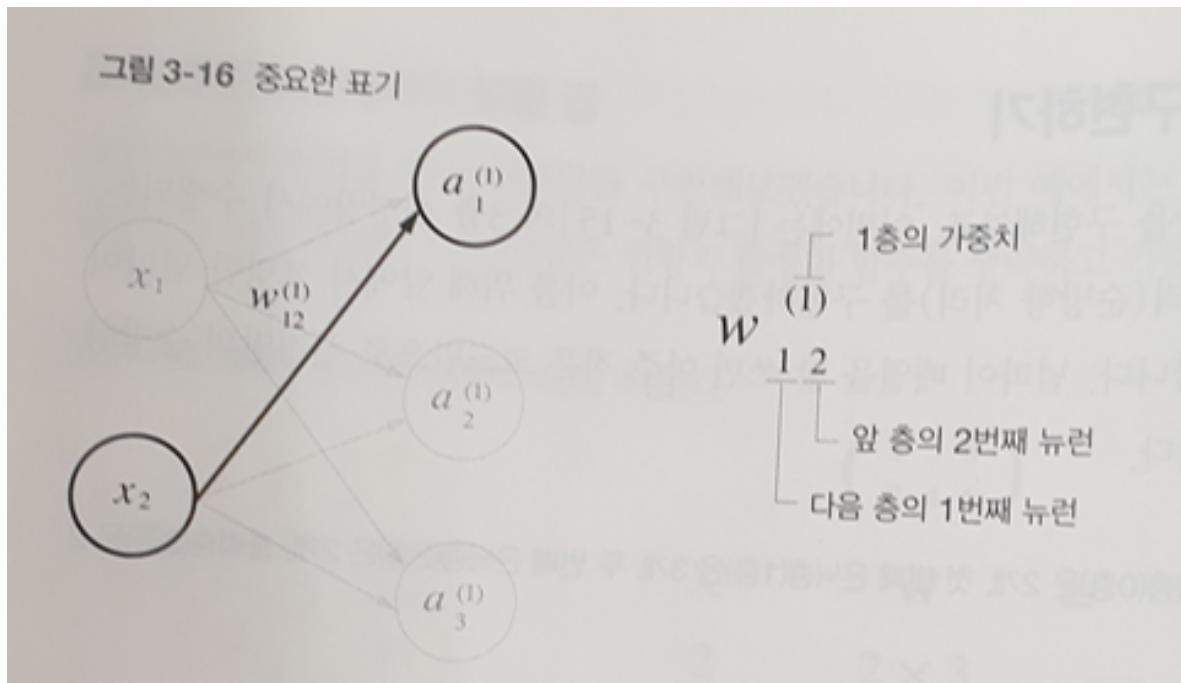
입력층(0층) 2개, 첫 번째 은닉층(1층) 3개, 두 번째 은닉층(2층) 2개, 출력층(3층) 2개의 뉴런으로 구성된다.

3.4.1 표기법 설명

이번 절에서는 신경망에서의 처리를 설명하며 $w_{12}^{(1)}, a_1^{(1)}$ 같은 표기법을 사용한다. 조금 복잡해 보일 수 있는데, 이번 절에서만 사용하는 표기법이다.

WARNING_이번 절의 핵심은 신경망에서의 계산을 행렬 계산으로 정리할 수 있다는 것이다. 신경망 각 층의 계산은 행렬의 곱으로 처리할 수(더 큰 관점에서 생각할 수) 있으니, 세세한 표기 규칙은 잊어버려도 앞으로의 설명을 이해하는 데 전혀 지장없다.

다음 그림을 보자.



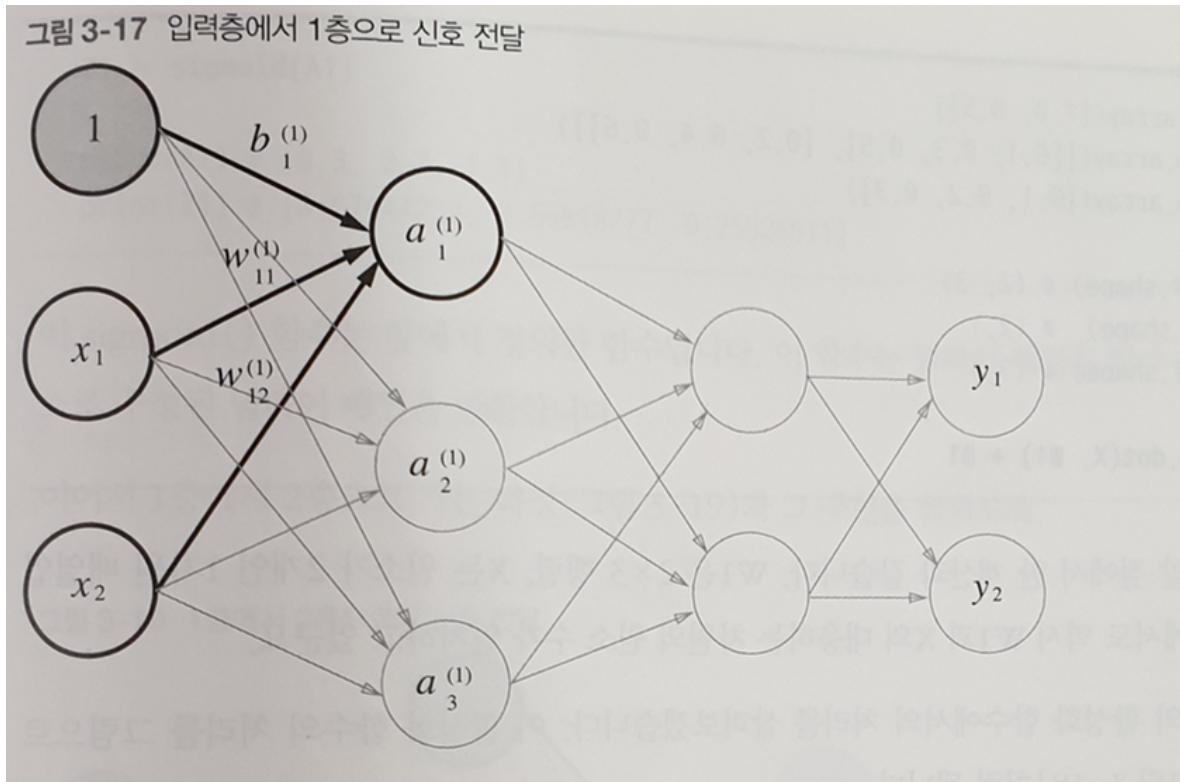
입력층의 뉴런 x_2 에서 다음 층의 뉴런 $a_1^{(1)}$ 으로 향하는 선 위에 가중치를 표시하고 있다.

이와 같이 가중치와 은닉층 뉴런의 오른쪽 위에는 ⁽¹⁾ 이 붙어 있다. 이는 1층의 가중치, 1층의 뉴런임을 뜻하는 번호이다. 또, 가중치의 오른쪽 아래로 | 두 숫자는 차례로 다음 층 뉴런과 앞 층 뉴런의 인덱스 번호다.

3.4.2 각 층의 신호 전달 구현하기

이번 절에서는 입력층에서 '1층의 첫 번째 뉴런'으로 가는 신호를 살펴보겠다.

다음 그림을 보자.



편향을 뜻하는 뉴런인 1이 추가 됐다. 편향은 오른쪽 아래 인덱스가 하나밖에 없다는 것에 주의해라. 이는 앞 층의 편향 뉴런(뉴런 1)이 하나뿐이기 때문이다.

그럼 지금까지 확인한 것을 반영하여 $a_1^{(1)}$ 을 수식으로 나타내보자. $a_1^{(1)}$ 은 가중치를 곱한 신호 두 개와 편향을 합해서 다음과 같이 계산한다.

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

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

$$A^{(1)} = XW^{(1)} + B^{(1)} \quad \text{--- --- --- --- --- 식 (3.9)}$$

이 때 $A^{(1)}, X, W^{(1)}, B^{(1)}$ 은 각각 다음과 같다.

$$A^{(1)} = (a_1^{(1)} \ a_2^{(1)} \ a_3^{(1)}), X = (x_1, x_2), B^{(1)} = (b_1^{(1)} \ b_2^{(1)} \ b_3^{(1)})$$
$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}$$

그럼 넘파이의 다차원 배열을 사용해서 식 3.9를 구현해보자.

```
In [67]: X = np.array([1.0, 0.5])
In [68]: w1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
In [69]: b1 = np.array([0.1, 0.2, 0.3])
In [70]: print(w1.shape)
(2, 3)
In [71]: print(X.shape)
(2,)
In [72]: print(b1.shape)
```

(3,)

```
In [73]: A1 = np.dot(X,W1)+B1
```

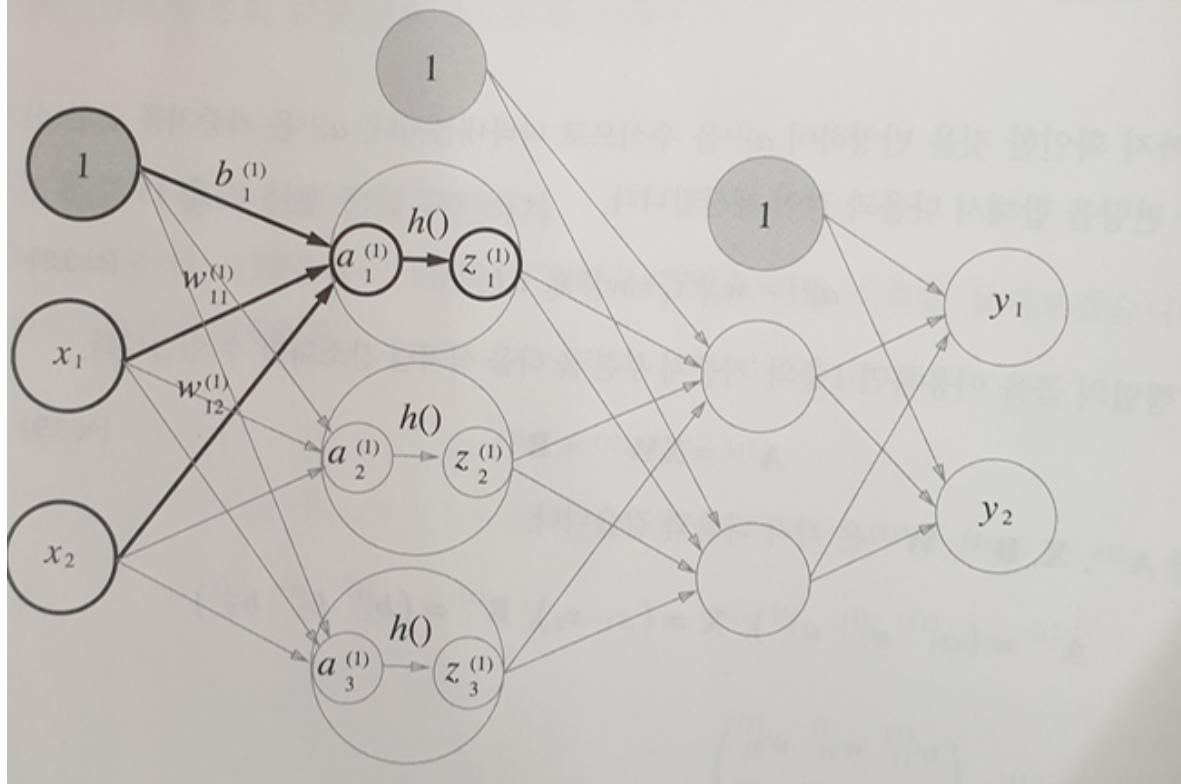
```
In [75]: A1
```

```
Out[75]: array([0.3, 0.7, 1.1])
```

이 계산은 앞 절에서 한 계산과 같다. W1은 2x3행렬, X는 원소가 2개인 1차원 배열이다. 여기에서도 W1과 X의 대응하는 차원의 원소 수가 일치하고 있다.

이어서 1층의 활성화 함수에서의 처리를 살펴봤다. 이 활성화 함수의 처리를 그림으로 나타내면 다음처럼 된다.

그림 3-18 입력층에서 1층으로의 신호 전달



위와 같이 은닉층에서의 가중치 합(가중 신호와 편향의 총합)을 a 로 표기하고 활성화 함수 $h()$ 로 변환된 신호를 z 로 표기한다. 여기에서는 활성화 함수로 시그모이드 함수를 사용하기로 한다. 이를 파이썬으로 구현하면 다음과 같다.

```
In [75]: A1
```

```
Out[75]: array([0.3, 0.7, 1.1])
```

```
In [76]: z1 = sigmoid(A1)
```

```
In [77]: print(z1)
```

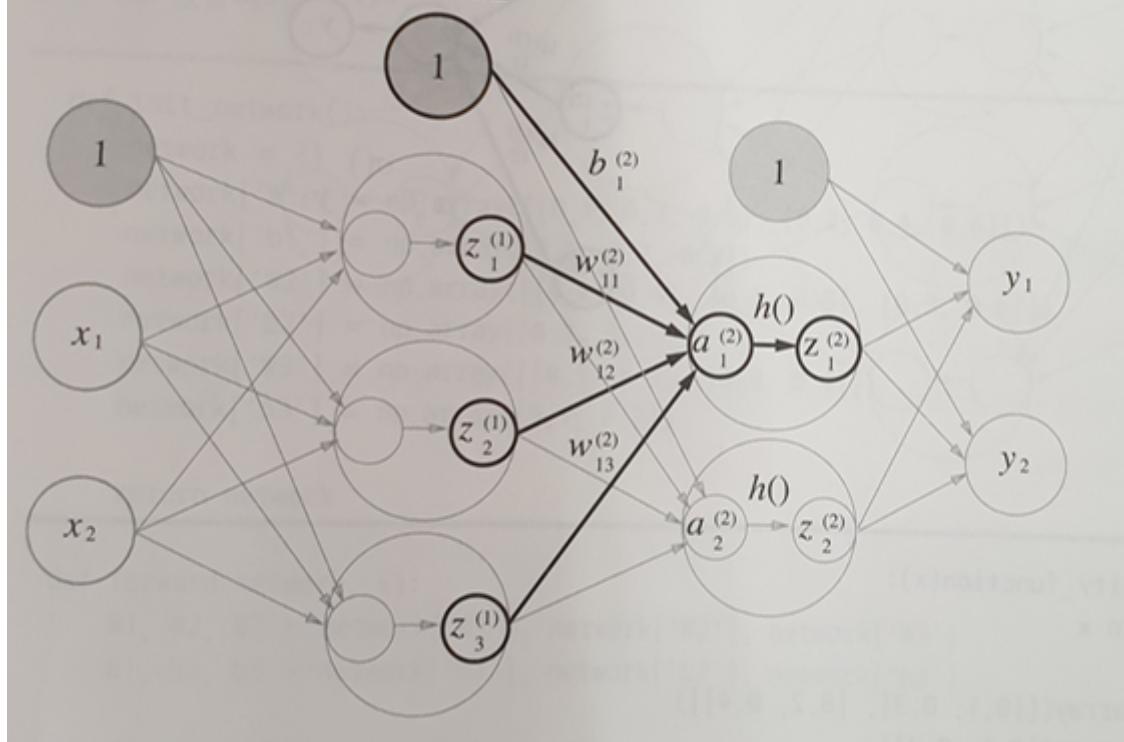
```
[0.3 0.7 1.1]
```

```
In [78]: print(z1)
```

```
[0.57444252 0.66818777 0.75026011]
```

이 `sigmoid()` 함수는 앞에서 정의한 함수이다. 이 함수는 넘파이 배열을 받아 같은 수의 원소로 구성된 넘파이 배열을 반환한다. 이어서 1층에서 2층으로 가는 과정은 다음 그림과 같다.

그림 3-19 1층에서 2층으로의 신호 전달



```
In [80]: w2 = np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
```

```
In [81]: b2 = np.array([0.1,0.2])
```

```
In [82]: print(z1.shape)
(3,)
```

```
In [83]: print(w2.shape)
(3, 2)
```

```
In [84]: print(b2.shape)
(2,)
```

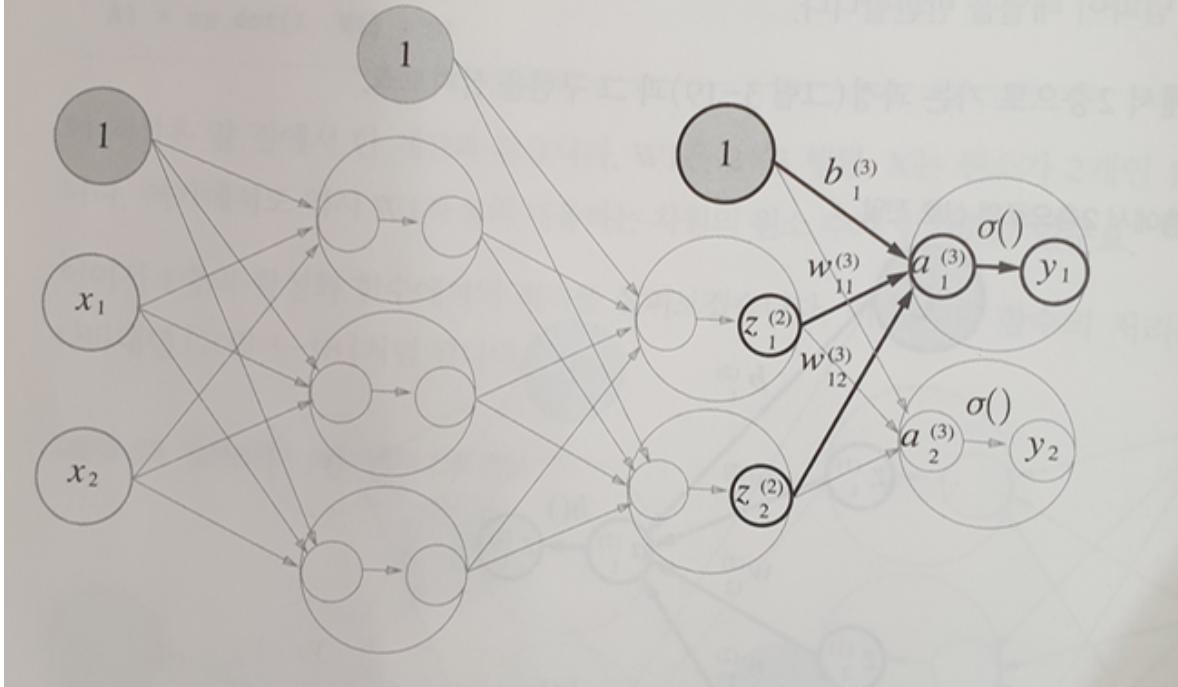
```
In [85]: A2 = np.dot(z1,w2)+b2
```

```
In [88]: z2 = sigmoid(A2)
```

이 구현은 1층의 출력 Z_1 이 2층의 입력이 된다는 점을 제외하면 조금 전의 구현과 똑같다. 이처럼 넘파이 배열을 사용하면서 층 사이의 신호 전달을 쉽게 구현할 수 있다. 마지막으로 2층에서 출력층으로의 신호 전달이다.

마지막으로 2층에서 출력층으로의 신호 전달이다. 다음 그림과 같다. 출력층의 구현도 그동안의 구현과 거의 같다. 딱 하나, 활성화 함수만 지금까지의 은닉층과 다르다.

그림 3-20 2층에서 출력층으로의 신호 전달



```
In [89]: def identity_function(x):
....:     return x
....:
In [90]: w3 = np.array([[0.1,0.3],[0.2,0.4]])
In [91]: b3 = np.array([0.1,0.2])
In [93]: A3 = np.dot(z2,w3) + b3
In [94]: Y = identity_function(A3)
```

여기에서는 항등 함수인 `identity_function()`을 정의하고, 이를 출력층의 활성화 함수로 이용했다. 항등 함수는 입력을 그대로 출력하는 함수다. 이에 이 예에서는 `identity_function()`을 굳이 정의할 필요는 없지만, 그동안의 흐름과 통일하기 위해 구현했다.

또한 그림에서는 출력층의 활성화 함수를 $\sigma()$ 을 표시하여 은닉층의 활성화 함수 $h(x)$ 와는 다름을 명시했다.

NOTE_ 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 맞게 정한다. 예를 들어 회귀에는 항등함수를, 2클래스 분류에는 시그모이드 함수를, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적이다. 출력층의 활성화 함수에 대해서는 다음에 다시 설명하겠다.

3.4.3 구현 정리

이로써 3층 신경망에 대한 설명은 끝이다. 그럼 다시 정리해보겠다. 신경망 구현의 관례에 따라 가중치만 W 과 같이 대문자로 쓰고, 그 외 편향과 중간 결과들은 모두 소문자로 쓰겠다.

```
In [98]: 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 [101]: 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
....:

In [104]: network = init_network()

In [105]: x = np.array([1.0,0.5])

In [106]: y = forward(network, x)

In [107]: print(y)
[0.31682708 0.69627909]

```

여기에서는 `init_network()`와 `forward()`라는 함수를 정의했다. `init_network()` 함수는 가중치와 편향을 초기화하고 이들을 딕셔너리 변수인 `network`에 저장한다. 이 딕셔너리 변수 `network`에는 각 층에 필요한 매개변수(가중치와 편향)를 저장한다. 그리고 `forward()` 함수는 입력 신호를 출력을 변환하는 처리과정을 모두 구현하고 있다.

함수 이름을 `forward`라고 한 것은 신호가 순방향(입력에서 출력 방향)으로 전달됨(순전파)을 알리기 위함이다.

앞으로 신경망 학습을 다룰 때 역방향(backward) 처리에 대해서도 알아볼 것이다.

3.5 출력층 설계하기

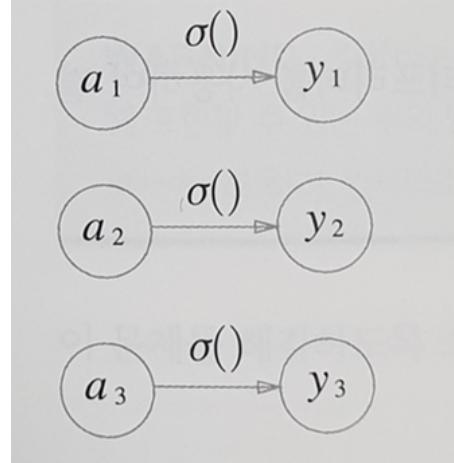
신경망은 분류와 회귀 모두에 이용할 수 있다. 다만 둘 중 어떤 문제냐에 따라 출력층에서 사용하는 활성화 함수가 달라진다. 일반적으로 회귀에는 항등 함수를, 분류에는 소프트맥스 함수를 사용한다.

NOTE_ 기계학습 문제는 분류와 회귀로 나뉜다. 분류는 데이터가 어느 클래스에 속하느냐는 문제이다. 사진 속 인물의 성별을 분류하는 문제가 여기에 속한다. 한편, 회귀는 입력 데이터에서 (연속적인) 수치를 예측하는 문제다. 사진 속 인물의 몸무게(57.4kg?)를 예측하는 문제가 회귀다.

3.5.1 항등 함수와 소프트맥스 함수 구현하기

항등 함수는 입력을 그대로 출력한다. 입력과 출력이 항상 같다는 뜻의 항등이다. 이에 출력층에서 항등 함수를 사용하면 입력신호가 그대로 출력 신호가 된다. 항등 함수 처리는 신경망 그림으로는 아래처럼 되겠다. 항등 함수에 의한 변환은 은닉층에서의 활성화 함수와 마찬가지로 화살표로 그린다.

그림 3-21 항등 함수



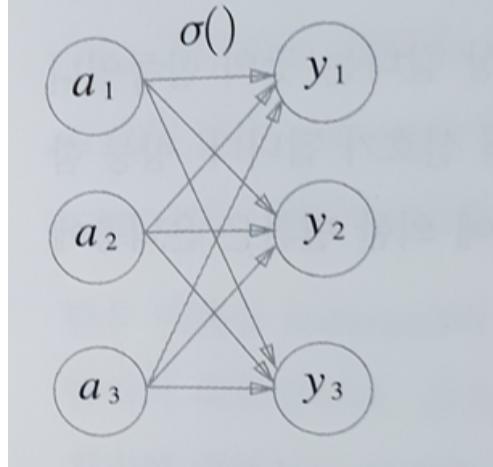
한편, 분류에서 사용하는 소프트맥스 함수의 식은 다음과 같다.

$$y_i = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} \quad \text{식 (3.10)}$$

n 은 출력층의 뉴런 수, y_i 는 그 중 k 번째 출력임을 뜻한다. 위 식과 같이 소프트맥스 함수의 분자는 입력 신호 a_k 의 지수함수, 분모는 모든 입력 신호의 지수 함수 합으로 구성된다.

이 소프트맥스 함수를 그림으로 나타내면 다음 그림처럼 된다. 그림과 같이 소프트맥스의 출력은 모든 입력 신호로부터 화살표를 받는다. 식 3.10의 분모에서 보듯, 출력층의 각 뉴런이 모든 입력 신호에서 영향을 받기 때문이다.

그림 3-22 소프트맥스 함수



그럼, 이상의 소프트맥스 함수를 구현해보자. 여기에서는 파이썬 인터프리터를 사용하여 결과를 하나씩 확인해가며 진행하겠다.

```
In [108]: a = np.array([0.3, 2.9, 4.0])
```

```
In [109]: exp_a = np.exp(a)
```

```
In [110]: print(exp_a)
[ 1.34985881 18.17414537 54.59815003]
```

```
In [111]: sum_exp_a = np.sum(exp_a)
```

```
In [112]: print(sum_exp_a)
```

```
74.1221542101633
```

```
In [113]: y = exp_a/sum_exp_a  
In [114]: print(y)  
[0.01821127 0.24519181 0.73659691]
```

이 구형은 식 3.10의 식을 그대로 파이썬에 붙인 것이다. 이제 이 식을 함수로 정의해두겠다.

```
In [115]: def softmax(a):  
....:     exp_a = np.exp(a)  
....:     sum_exp_a = np.sum(exp_a)  
....:     y = exp_a/sum_exp_a  
....:     return y  
....:
```

3.5.2 소프트맥스 함수 구현 시 주의점

앞 절에서 구현한 softmax() 함수의 코드는 위 식을 제대로 표현하고 있지만, 컴퓨터로 계산할 때는 결함이 있다. 바로 오버플로 문제다. 소프트맥스 함수는 지수 함수를 사용하는데, 지수 함수란 것이 쉽게 아주 큰 값을 내뱉는다. 가령 e^{10} 은 20,000이 넘고, e^{1000} 은 무한대를 뜻하는 inf가 되어 돌아온다. 그리고 이런 큰 값끼리 나눗셈을 하면 결과 수치가 '불안정'해진다.

WARNING_ 컴퓨터는 수를 4바이트나 8바이트와 같이 크기가 유한한 데이터로 다룬다. 다시 말해 표현 할 수 있는 수의 범위가 한정되어 너무 큰 값은 표현할 수 없다는 문제가 생긴다. 이것을 오버플로라고 하며, 컴퓨터로 수치를 계산할 때 주의해야 할 점이다.

이 문제를 해결하도록 소프트맥스 함수 구현을 개선해보자. 다음은 개선한 수식이다.

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

위 식의 전개 과정을 살펴보자. 첫 번째 변경에서는 C라는 임의의 정수를 분자와 분모 양쪽에 곱했다(양 쪽에 같은 수를 곱했으니 결국 똑같은 계산이다). 그 다음으로 C를 지수 함수 exp() 안으로 옮겨 log C로 만든다. 마지막으로 logC를 C'라는 새로운 기호로 바꾼다.

위 식이 말하는 것은 소프트맥스의 지수 함수를 계산할 때 어떤 정수를 더해도 (혹은 빼도) 결과는 바뀌지 않는다는 것이다. 여기서 C'에 어떤 값을 대입해도 상관없지만, 오버플로를 막을 목적으로는 입력 신호 중 최댓값을 이용하는 것이 일반적이다. 구체적인 예를 하나 들어보자.

```
In [144]: a = np.array([1010,1000,990])

In [145]: np.exp(a)/np.sum(np.exp(a))
Out[145]: array([nan, nan, nan])

In [146]: c = np.max(a)

In [147]: a-c
Out[147]: array([ 0, -10, -20])

In [148]: np.exp(a-c)/np.sum(np.exp(a-c))
Out[148]: array([9.99954600e-01, 4.53978686e-05, 2.06106005e-09])
```

이 예에서 보는 것처럼 아무런 조치 없이 그냥 계산하면 nan이 출력된다.(nan은 not a number의 약자다.) 하지만 입력 신호 중 최댓값(이 예에서는 c)을 빼주면 올바르게 계산할 수 있다. 이를 바탕으로 소프트 맥스 함수를 다시 구현하면 다음과 같다.

```
In [116]: 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
....:
```

3.5.3 소프트맥스 함수의 특징

softmax() 함수를 사용하면 신경망의 출력은 다음과 같이 계산할 수 있다.

```
In [125]: a = np.array([0.3,2.9,4.0])

In [126]: y = softmax(a)

In [127]: print(y)
[0.01821127 0.24519181 0.73659691]

In [128]: np.sum(y)
Out[128]: 1.0
```

보는 바와 같이 소프트맥스 함수의 출력은 0~1 사이의 실수이다. 또, 소프트맥스 함수 출력의 총합은 10이다. 출력 총합이 1이 된다는 점은 소프트맥스 함수의 중요한 성질이다. 이 성질 덕에 소프트맥스 함수의 출력을 '확률'로 해석할 수 있다.

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%)으로 해석할 수 있다. 이에 이 결과 확률들로부터 2번째 원소의 확률이 가장 높으니, 답은 2번째 클래스다" 혹은 "74%의 확률로 2번째 클래스, 25%의 확률로 1번째 클래스, 1%의 확률로 0번째 클래스다"와 같이 확률적인 결론도 낼 수 있다.

즉, 소프트맥스 함수를 이용함으로써 문제를 확률적(통계적)으로 대응할 수 있다는 것이다.

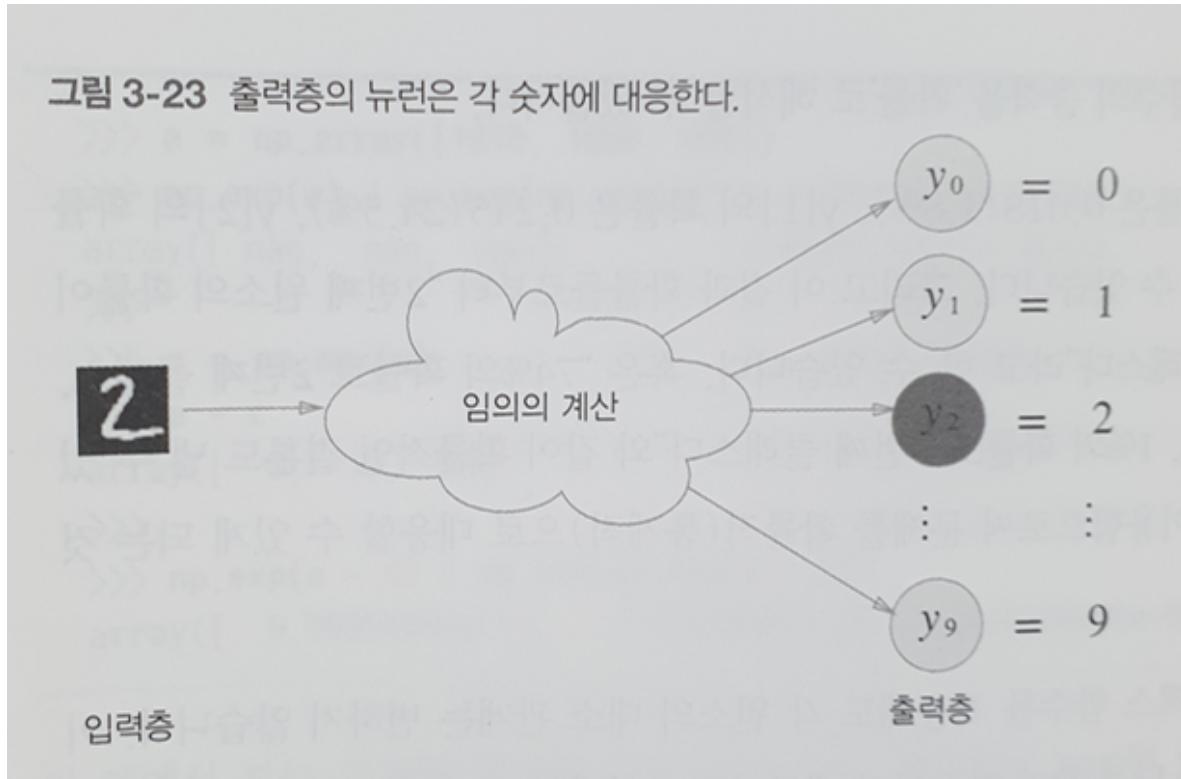
여기서 주의점으로, 소프트맥스 함수를 적용해도 각 원소의 대소 관계는 변하지 않는다. 이는 지수함수 $y = \exp(x)$ 가 단조 증가함수이기 때문이다. 실제로 앞의 예에서는 a의 원소들 사이의 대소 관계가 y의 원소들 사이의 대소 관계로 그대로 이어진다. 예를 들어 a에서 가장 큰 원소는 2번째 원소이고, y에서 가장 큰 원소도 2번째 원소이다.

신경망을 이용한 분류에서는 일반적으로 가장 큰 출력을 내는 뉴런에 해당하는 클래스로만 인식한다. 그리고 소프트맥스 함수를 적용해도 출력이 가장 큰 뉴런의 위치는 달라지지 않는다. 결과적으로 신경망으로 분류할 때는 출력층의 소프트맥스 함수를 생략해도 된다. 현업에서도 지수 함수 계산에 드는 자원 낭비를 줄이고자 출력층의 소프트맥스 함수는 생략하는 것이 일반적이다.

Note_ 기계학습의 문제 풀이는 학습과 추론의 두 단계를 거쳐 이뤄진다. 학습 단계에서 모델을 학습하고 (직업 훈련을 받고), 추론 단계에서 앞서 학습한 모델로 미지의 데이터에 대해 추론(분류)을 수행한다(현장에 나가 진짜 일을 한다) 방금 설명한 대로, 추론 단계에서는 출력층의 소프트맥스 함수를 생략하는 것이 일반적이다. 한편, 신경망을 학습시킬 때는 출력층에서 소프트맥스 함수를 사용한다.

3.5.4 출력층의 뉴런 수 정하기

출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정해야 한다. 분류에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적이다. 예를 들어 입력 이미지를 숫자 0~9 중 하나로 분류하는 문제라면 다음 그림처럼 출력층의 뉴런을 10개로 설정한다.



위 그림의 예에서 출력층 뉴런은 위에서부터 차례로 1, ..., 9에 대응하며, 뉴런의 회색 농도가 해당 뉴런의 출력 값의 크기를 의미한다. 이 예에서는 색이 가장 짙은 y_2 뉴런이 가장 큰 값을 출력하는 것이다. 그래서 이 신경망이 선택한 인덱스는 y_2 , 즉 입력 이미지를 숫자 '2'로 판단했음을 의미한다.

3.6 손글씨 숫자 인식

신경망의 구조를 배웠으니 실전 예에 적용해보자. 바로 손글씨 숫자 분류이다. 이번 절에서는 이미 학습된 매개변수를 사용하여 학습 과정은 생략하고, 추론과정만 구현할 것이다. 이 추론 과정은 신경망의 순전파라고도 한다.

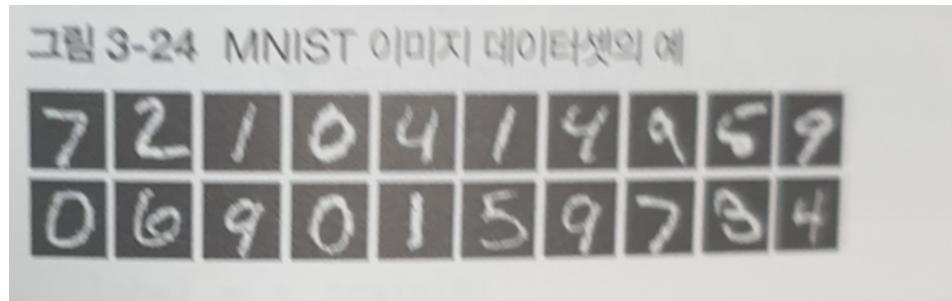
Note_ 기계학습과 마찬가지로 신경망도 두 단계를 거쳐 문제를 해결한다. 먼저 훈련 데이터(학습 데이터)를 사용해 가중치 매개변수를 학습하고, 추론 단계에서는 앞서 학습한 매개변수를 사용하여 입력 데이터를 분류한다.

3.6.1 MNIST 데이터셋

이번 예에서 사용할 데이터셋은 MNIST라는 손글씨 숫자 이미지 집합이다. MNIST는 기계학습 분야에서 아주 유명한 데이터셋으로, 간단한 실험부터 논문으로 발표되는 연구까지 다양한 곳에서 이용되고 있다.

이미지 인식이나 기계학습 논문들을 읽다보면 실험용 데이터로 자주 등장하는 것을 확인할 수 있을 것이다.

MNIST 데이터셋은 0~9까지의 숫자 이미지로 구성된다. 훈련 이미지가 60,000장, 시험 이미지가 10,000장 준비되어 있다. 일반적으로 이 훈련 이미지들을 사용하여 모델을 학습하고, 학습한 모델로 시험 이미지를 얼마나 정확하게 분류하는지를 평가한다.



MNIST의 이미지 데이터는 28×28 크기의 회색조 이미지이며, 각 픽셀은 0~255 값을 취한다.

각 이미지에는 또한 '7', '2', '1'과 같이 그 이미지가 실제 의미하는 숫자가 레이블로 붙어 있다.

이 책에서는 MNIST 데이터 셋을 내려받아 이미지를 넘파이 배열로 변환해주는 파이썬 스크립트를 제공한다. mnist.py 파일에 정의된 load_mnist() 함수를 이용하면 MNIST 데이터를 다음과 같이 아주 쉽게 가져올 수 있다.

```
#mnist_show.py
In [132]: import sys, os
....: sys.path.append(os.pardir) #부모 디렉터리의 파일을 가져올 수 있도록 설정
....: from dataset.mnist import load_mnist
....: (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 [133]: print(x_train.shape)
(60000, 784)

In [134]: print(t_train.shape)
(60000,)

In [135]: print(x_test.shape)
(10000, 784)

In [136]: print(t_test.shape)
(10000,)
```

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

WARNING `mnist.py` 파일은 이 책 예제 소스의 `dataset` 디렉터리에 있고, 이 파일을 이용하는 다른 예제들은 각각 `ch01`, `ch02`.. 디렉터리에서만 수행한다고 가정한다. 즉, 각 예제에서 `mnist.py` 파일을 찾으려면 부모 디렉터리로부터 시작해야 해서 `sys.path.append(os.pardir)` 문장을 추가한 것이다.

`load_mnist` 함수는 읽은 MNIST 데이터를 "(훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)"

형식으로 반환한다. 인수로는 `normalize`, `flatten`, `one_hot_label` 세 가지를 설정할 수 있다. 세 인수 모두 `bool` 값이다.

첫 번째 인수인 `normalize`는 입력 이미지의 픽셀 값을 0.0~1.0 사이의 값을 정규화할지를 정한다.

`False`로 설정하면 입력 이미지의 픽셀은 원래 값 그대로 0~255 사이의 값을 유지한다.

두 번째 인수인 `flatten`은 입력 이미지를 평탄하게, 즉 1차원 배열을 만들지를 정한다. `False`로 설정하면 입력 이미지를 $1 \times 28 \times 28$ 의 3차원 배열로, `True`로 설정하면 784개의 원소로 이루어진 1차원 배열로 저장한다.

세 번째 인수인 `one_hot_label`은 레이블을 원-핫 인코딩 형태로 저장할지를 정한다. 원-핫 인코딩이란, 예를 들어 $[0,0,1,0,0,0,0]$ 처럼 정답을 뜻하는 원소만 1이고 (hot하고) 나머지는 모두 0인 배열이다. `one_hot_label`이 `False`면 '7'이나 '2'와 같이 숫자 형태의 레이블을 저장하고 `True`일 때는 레이블을 원-핫 인코딩하여 저장한다.

Note 파일에는 `pickle`이라는 편리한 기능이 있다. 이는 프로그램 실행 중에 특정 객체를 파일로 저장하는 기능이다. 저장해둔 `pickle` 파일을 로드하면 실행 당시에 객체를 즉시 복원할 수 있다. MNIST 데이터셋을 읽는 `load_mnist()` 함수에서도 (2번째 이후의 읽기 시) `pickle`을 이용한다.

`pickle` 덕분에 MNIST 데이터를 순식간에 준비할 수 있다.

```
def load_mnist(normalize=True, flatten=True, one_hot_label=False):
    """MNIST 데이터셋 읽기

    Parameters
    -----
    normalize : 이미지의 픽셀 값을 0.0~1.0 사이의 값으로 정규화할지 정한다.
    one_hot_label :
        one_hot_label이 True면, 레이블을 원-핫(one-hot) 배열로 돌려준다.
        one-hot 배열은 예를 들어 [0,0,1,0,0,0,0,0,0]처럼 한 원소만 1인 배열이다.
    flatten : 입력 이미지를 1차원 배열로 만들지를 정한다.

    Returns
    -----
    (훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)
    """
    if not os.path.exists(save_file):
        init_mnist()

    with open(save_file, 'rb') as f:
        dataset = pickle.load(f)

    if normalize:
        for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].astype(np.float32)
            dataset[key] /= 255.0

    if one_hot_label:
        dataset['train_label'] = _change_one_hot_label(dataset['train_label'])
        dataset['test_label'] = _change_one_hot_label(dataset['test_label'])

    if not flatten:
```

```

    for key in ('train_img', 'test_img'):
        dataset[key] = dataset[key].reshape(-1, 1, 28, 28)

    return (dataset['train_img'], dataset['train_label']),
            (dataset['test_img'], dataset['test_label'])

```

그럼 데이터도 확인할 겸 MNIST 이미지를 화면으로 불러보겠다. 이미지 표시에는 PIL(python Image Library) 모듈을 사용한다.

다음 코드를 실행하면 첫 번째 훈련 이미지가 모니터 화면에 출력된다.



```

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
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) # 형상을 원래 이미지의 크기로 변형
print(img.shape) # (28, 28)

img_show(img)

```

여기서 주의 사항으로, flatten = True로 설정해 읽어 들인 이미지는 1차원 넘파이 배열로 저장되어 있다. 그래서 이미지를 표시할 때는 원래 형상인 28 X 28 크기로 다시 변형해야 한다. reshape() 메서드에 원하는 형상을 인수로 지정하면 넘파이 배열의 형상을 바꿀 수 있다. 또한, 넘파이로 저장된 이미지 데이터를 PIL용 데이터 객체로 변환해야 하며, 이 변환은 Image.fromarray()가 수행한다.

3.6.2 신경망의 추론 처리

드디어 이 MNIST 데이터셋을 가지고 추론을 수행하는 신경망을 구현할 차례다. 이 신경망은 입력층 뉴런을 784개, 출력층 뉴런을 10개로 구성하다. 입력층 뉴런이 784개인 이유는 이미지 크기가 $28 \times 28 = 784$ 이기 때문이고, 출력층의 뉴런이 10개인 이유는 이 문제가 0~9까지의 숫자를 구분하는 문제이기 때문이다. 한편, 은닉층은 총 두 개로, 첫 번째 은닉층에는 50개의 뉴런을, 두 번째 은닉층에는 100개의 뉴런을 배치할 것이다. 여기서 50과 100은 임의로 정한 값이다.

이제 순서대로 작업을 처리해줄 세 함수인 `get_data()`, `init_network()`, `predict()`를 정의하겠다.

```
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
```

`init_network()`에서는 `pickle` 파일인 `sample_weight.pkl`이 저장하고 있는 '학습된 가중치 매개변수'를 읽는다. 이 파일에는 가중치와 평향 매개변수가 딕셔너리 변수로 저장되어 있다. 나머지 두 함수는 따로 설명안하겠다.

그럼 이 세 함수를 사용해 신경망에 의한 추론을 수행해보고, 정확도(분류가 얼마나 올바른가)도 평가해보자.

```

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)))

```

가장 먼저 MNIST 데이터셋을 얻고 네트워크를 생성한다. 이어서 for 문을 돌려 x에 저장된 이미지 데이터를 1장씩 꺼내 predict() 함수로 분류한다. predict() 함수는 각 레이블의 확률을 넘파이 배열로 반환한다. 예를 들어 [0.1, 0.3, 0.2, ..., 0.04] 같은 배열이 반환되며, 이는 이미지가 숫자 '0'일 확률이 0.1, '1'일 확률이 0.3 ... 식으로 해석된다.

그런 다음 np.argmax() 함수로 이 배열에서 값이 가장 큰(확률이 가장 높은) 원소의 인덱스를 구한다.

이것이 바로 예측 결과다. 마지막으로, 신경망이 예측한 답변과 정답 레이블을 비교하여 맞힌 숫자(accuracy_cnt)를 세고, 이를 전체 이미지 숫자로 나눠 정확도를 구한다.

이 코드를 실행하면 "Accuracy : 0.9353"라고 출력한다. 올바르게 분류한 비율이 93.52%라는 뜻이다.

이번 장의 목표는 학습된 신경망을 돌려보는 것끼리, 정확도에 대해서는 고민하지 않겠지만

다음 장부터는 신경망 구조와 학습 방법을 궁리하여 이 정확도를 더 높여갈 것이다. 마지막에는 99% 이상 까지 도달할 예정이다.

또한, 이 예에서는 load_mnist 함수의 인수인 normalize를 True로 설정했다.

normalize를 True로 설정하면 0~255 범위인 각 픽셀의 값을 0.0~1.0 범위로 변환한다(단순히 픽셀의 값을 255로 나눈다). 이처럼 데이터를 특정 데이터를 특정 범위로 변환하는 처리를 **정규화** 라 하고, 신경망의 입력 데이터에 특정 변환을 가하는 것을 **전처리** 라고 한다. 여기서는 입력 이미지 데이터에 대한 전처리 작업으로 정규화를 수행한 셈이다.

Note 현업에서도 신경망(딥러닝)에 전처리를 활발히 사용한다. 전처리를 통해 식별 능력을 개선하고 학습 속도를 높이는 등의 사례가 많이 제시되고 있다. 앞의 예에서는 각 픽셀의 값을 255로 나누는 단순한 정규화를 수행했지만, 현업에서는 데이터 전체의 분포를 고려해 전처리하는 경우가 많다.

예를 들어 데이터 전체 평균과 표준편차를 이용하여 데이터들이 0을 중심으로 분포하도록 이동하거나 데이터의 확산 범위를 제한하는 정규화를 수행한다. 그 외에도 전체 데이터를 균일하게 분포시키는 데이터 백색화 등도 있다.

3.6.3 배치 처리

구현 진도를 더 나가기 전에 이번 절에서는 입력 데이터와 가중치 매개변수의 '형상'에 주의해서 조금 전의 구현을 다시 살펴보겠다. 우선 파이썬 인터프리터에서 앞서 구현한 신경망 각 층의 가중치 형상을 출력해보겠다.

```
In [151]: x,_ = get_data()
```

```
In [152]: network = init_network()
```

```
In [153]: w1, w2, w3 = network['w1'], network['w2'], network['w3']
```

```
In [154]: x.shape  
Out[154]: (10000, 784)
```

```
In [155]: x[0].shape  
Out[155]: (784,)
```

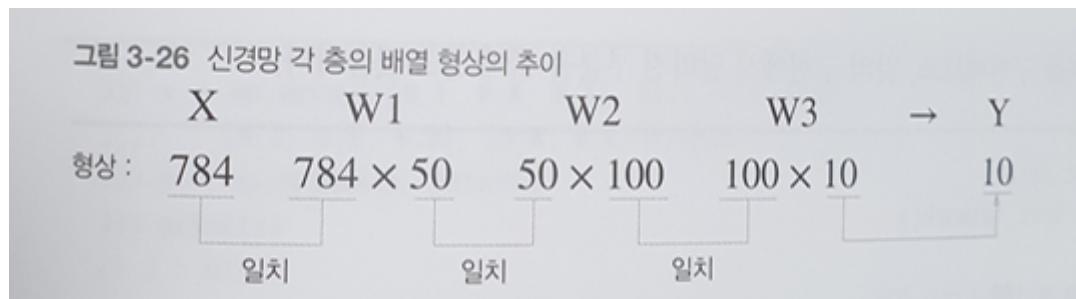
```
In [156]: w1.shape  
Out[156]: (784, 50)
```

```
In [157]: w2.shape  
Out[157]: (50, 100)
```

```
In [158]: w3.shape  
Out[158]: (100, 10)
```

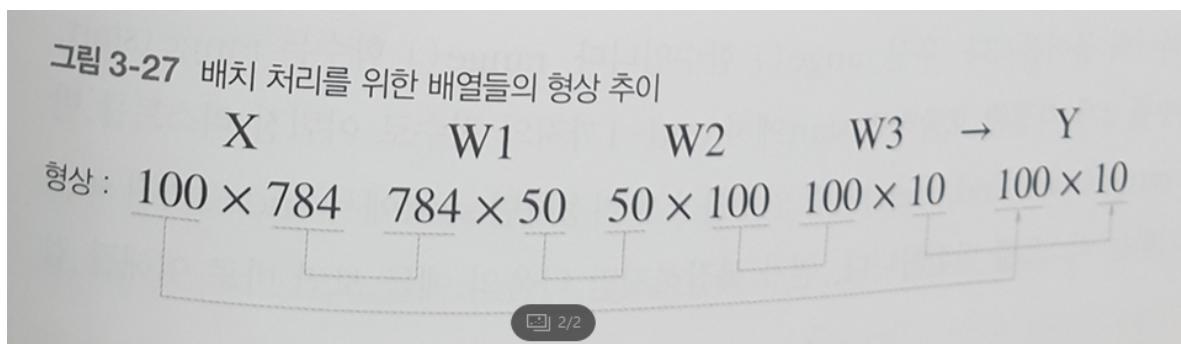
이 결과에서 다차원 배열에 대응하는 차원의 원소 수가 일치함을 확인할 수 있다(편향은 생략했다).

그림으로는 다음처럼 된다. 다차원 배열에 대응하는 차원의 원소 수가 일치하고 있다. 그리고 최종 결과로는 원소가 10개인 1차원 배열 y 가 출력되는 점도 확인하자.



위 그림을 전체적으로 보면 원소 784개로 구성된 1차원 배열(원래는 28×28 인 2차원 배열)이 입력되어 마지막에는 원소가 10개인 1차원 배열이 출력되는 흐름이다. 이는 이미지 데이터를 1장만 입력했을 때의 처리 흐름이다.

그렇다면 이미지 여러장을 한꺼번에 입력하는 경우를 생각해보자. 가령 이미지 100개를 묶어, predict() 함수에 한 번에 넘기는 것이다. x 의 형상을 100×784 로 바꿔서 100장 분량의 데이터를 하나의 입력 데이터로 표현하면 된다. 그림으로는 다음처럼 된다.



2/2

위 그림과 같이 입력 데이터의 형상은 100×784 , 출력 데이터의 형상은 100×10 이 된다. 이는 100장 분량 입력 데이터의 결과가 한 번에 출력됨을 나타낸다. 가령 $x[0]$ 과 $y[0]$ 에는 0번째 이미지와 그 추론 결과가, $x[1]$ 과 $y[1]$ 에는 1번째의 이미지와 그 결과가 저장되는 식이다.

이처럼 하나로 묶은 입력 데이터를 **배치(batch)**라고 한다. 배치가 곧 묶음이라는 의미다. 이미지가 지폐처럼 다발로 묶여 있다고 생각하면 된다.

Note 배치 처리는 컴퓨터로 계산할 때 큰 이점을 준다. 이미지 1장당 처리 시간을 대폭 줄여주는 것이다. 크게 두 가지 이유가 있다.

하나는 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화되어 있기 때문이다. 그리고 커다란 신경망에서는 데이터 전송이 병목으로 작용하는 경우가 자주 있는데, 배치 처리를 함으로써 버스에 주는 부하를 줄인다는 것이 두번째 이유다.

(정확하는 느린 I/O를 통해 데이터를 읽는 횟수가 줄어, 빠른 CPU나 GPU로 순수 계산을 수행하는 비율이 높아진다.)

즉, 배치 처리를 수행함으로써 큰 배열로 이뤄진 계산을 하는데, 컴퓨터에서는 큰 배열을 한꺼번에 계산하는 것이 분할된 작은 배열을 여러 번 계산하는 것보다 빠르다.

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

batch_size = 100 # 배치 크기 이 부분이 달름
accuracy_cnt = 0

for i in range(0, len(x), batch_size):    #for문 추가
    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)))
```

굵은 부분을 하나씩 풀어보자. 우선 range() 함수이다. range() 함수는 range(start, end) 처럼 인수를 2개 지정해 호출하면 start에서 end-1까지의 정수로 이뤄진 리스트를 반환한다. 또 range(start, end, step) 처럼 인수를 3개 지정하면 start에서 end-1까지 step 간격으로 증가하는 리스트를 반환한다.

```
In [159]: list(range(0,10))
Out[159]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

OI range() 함수가 반환하는 리스트를 바탕으로 x[i:i+batch_size]에서 입력 데이터를 묶는다. x[i:i+batch_size]은 입력 데이터의 i번째부터 i+batch_size번째 까지의 데이터를 묶는다는 의미다. 이 예에서는 batch_size가 100이므로 x[0:100], x[100:200], .. 와 같이 앞에서부터 100장씩 묶어 꺼낸다.

그리고 앞에서도 나온 argmax()는 최댓값의 인덱스를 가져온다. 다마 여기서는 axis = 1이라는 인수를 추가한 것에 주의하자. 이는 100x10의 배열 중 1번째 차원을 구성하는 각 원소에서(1번째 차원을 음으로) 최댓값의 인덱스를 찾도록 한 것이다.(인덱스가 0부터 시작하니 0번째 차원이 가장 처음 차원이다.)

```
In [160]: 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]]))

In [161]: y = np.argmax(x, axis =1)

In [162]: print(y)
[1 2 1 0]

In [163]: z = np.argmax(x)

In [164]: print(z)
1
```

마지막으로 배치 단위로 분류한 결과를 실제 답과 비교한다. 이를 위해 == 연산자를 사용해 넘파이 배열끼리 비교하여 T/F로 구성된 bool 배열을 만들고, 이 결과 배열에서 T가 몇 개인지 센다. 이 처리 과정은 다음 예에서 확인해본다.

```
In [165]: y = np.array([1,2,1,0])
In [166]: t = np.array([1,2,0,0])
In [167]: print(y==t)
[ True  True False  True]
In [168]: np.sum(y==t)
Out[168]: 3
```

이상으로 배치 처리 구현에 대한 설명을 마친다. 데이터를 배치로 처리함으로써 효율적이고 빠르게 처리할 수 있었다. 다음 장에서 진행할 신경망 학습에서도 이미지 데이터를 적절히 묶어서 학습하는데, 그때도 이번 장에서 구현한 배치 처리와 같은 방식으로 구현한다.

3.7 정리

이번 장에서는 신경망의 순전파를 살펴봤다. 이번 장에서 설명한 신경망은 각 층의 뉴런들이 다음 층의 뉴런으로 신호를 전달한다는 점에서 앞 장의 퍼셉트론과 같다. 하지만 다른 뉴런으로 갈 때 신호를 변화시키는 활성화 함수에 큰 차이가 있었다. 신경망에서는 매끄럽게 변화하는 시그모이드 함수를, 퍼셉트론에서는 갑자기 변화하는 계단 함수를 활성화 함수로 사용했다. 이 차이가 신경망 학습에 중요하다.

이번 장에서 배운 것

- 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU 함수 같이 매끄럽게 변화하는 함수를 이용 한다.
- 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 소프트맥스 함수를 사용한다.
- 분류에서는 출력층의 뉴런 수를 분류하려는 클래스 수와 같게 설정한다.
- 입력 데이터를 묶은 것을 배치라 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 빠르게 얻을 수 있다.