

이번 장에서는 퍼셉트론 perceptron 알고리즘을 설명합니다. 퍼셉트론은 프랑크 로젠블라트 Frank Rosenblatt가 1957년에 고안한 알고리즘입니다. 고대 화석 같은 이 알고리즘을 지금 시점에 왜 배우는가 하면, 퍼셉트론이 신경망(딥러닝)의 기원이 되는 알고리즘이 되는 때문입니다. 그래서 퍼셉트론의 구조를 배우는 것은 신경망과 딥러닝으로 나아가는 데 중요한 아이디어를 배우는 일도 됩니다.

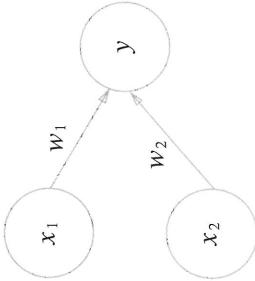
이번 장에서는 퍼셉트론을 설명하고 퍼셉트론을 써서 간단한 문제를 풀어집니다. 여성의 첫 목적인 만큼 가볍고 즐거운 여행이 될 겁니다.

2.1 퍼셉트론이란?

퍼셉트론*은 다수의 신호를 입력으로 받아 하나의 신호를 출력합니다. 여기서 말하는 신호란 전류나 강물처럼 흐름이 있는 것을 상상하면 좋습니다. 전류가 전선을 타고 흐르는 전자를 내보내듯, 퍼셉트론 신호도 흐름을 만들고 정보를 앞으로 전달합니다. 다만, 실제 전류와 달리 퍼셉트론 신호는 ‘흐른다/안 흐른다(1이나 0)’의 두 가지 값이 가능할 수 있습니다. 이 책에서는 1을 ‘신호가 흐른다’, 0을 ‘신호가 흐르지 않는다’라는 의미로 쓰겠습니다.

* 이번 장에서 기술하는 퍼셉트론은 정확히는 인공 뉴런입니다. 단, 기본적인 처리는 거의 비슷하나 이 책에서는 단순히 ‘퍼셉트론’이라 하겠습니다.

그림 2-1 입력이 2개인 퍼셉트론



[그림 2-1]은 입력으로 2개의 신호를 받은 퍼셉트론의 예입니다. x_1 과 x_2 는 입력 신호, y 는 출력 신호, w_1 과 w_2 는 기중치를 뜻합니다(w 는 weight의 머리글자죠). 그림의 원을 뉴런 혹은 노드라고 부릅니다. 입력 신호가 뉴런에 보내질 때는 각각 고유한 기중치가 곱해집니다($w_1x_1 + w_2x_2$). 뉴런에서 보내온 신호의 총합이 정해진 한계를 넘어설 때만 1을 출력합니다(이를 '뉴런이 활성화된다'라 표현하기도 합니다). 이 책에서는 그 한계를 임계값이라 하며, θ 기호(theta, 세타)로 나타냅니다.

퍼셉트론의 동작 원리는 이게 다입니다! 이상을 수식으로 나타내면 [식 2.1]이 됩니다.

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

[식 2.1]

퍼셉트론은 복수의 입력 신호 각각에 고유한 기중치를 부여합니다. 기중치는 각 신호가 결과에 주는 영향력을 조절하는 요소로 작용합니다. 즉, 기중치가 클수록 해당 신호가 그만큼 더 중요함을 뜻합니다.

NOTE 기중치는 전류에서 말하는 저항에 해당합니다. 저항은 전류의 흐름을 억제하는 매개변수로, 저항이 낮을수록 큰 전류가 흐릅니다. 한편 퍼셉트론의 기중치는 그 값이 클수록 강한 신호를 흘려보냅니다. 이처럼 서로 작용하는 방향은 반대지만, 신호가 얼마나 잘(혹은 어렵게) 흐르는지를 통제한다는 점에서 저항과 기중치는 같은 기능을 합니다.

2.2 단순한 논리 회로

2.2.1 AND 게이트

그럼 퍼셉트론을 활용한 간단한 문제를 살펴보죠. 논리 회로를 알아보는 첫걸음으로 AND 게이트를 살펴봅시다. AND 게이트는 입력이 둘이고 출력은 하나입니다. [그림 2-2]와 같은 입력 신호와 출력 신호의 대응 표를 진리표라고 합니다. 이 그림은 AND 게이트의 진리표로, 두 입력이 모두 1일 때만 1을 출력하고, 그 외에는 0을 출력합니다.

그림 2-2 AND 게이트의 진리표

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

이 AND 게이트를 퍼셉트론으로 표현하고 싶습니다. 이를 위해 할 일은 [그림 2-2]의 진리표대로 작동하도록 하는 w_1 , w_2 , θ 의 값을 정하는 것입니다. 그럼 어떤 값으로 설정하면 [그림 2-2]의 조건을 충족하는 퍼셉트론이 만들어질까요?

사실 [그림 2-2]를 만족하는 매개변수 조합은 무한히 많습니다. 가령 $(w_1, w_2, \theta) \mapsto (0.5, 0.5, 0.7)$ 일 때, 또 $(0.5, 0.5, 0.8)$ 이나 $(1.0, 1.0, 1.0)$ 때 모두 AND 게이트의 조건을 만족합니다. 매개변수를 이렇게 설정하면 x_1 과 x_2 모두가 1일 때만 가중 선호의 총합이 주어진 임계값을 초과하게 됩니다.

2.2.2 NAND 게이트와 OR 게이트

이어서 NAND 게이트를 살펴봅시다. NAND는 Not AND를 의미하며, 그 동작은 AND 게이트의 출력을 뒤집은 것이 됩니다. 진리표로 나타내면 [그림 2-3]처럼 x_1 과 $x_2 \not\geq 1$ 모두 1일 때만

0을 출력하고, 그 외에는 1을 출력합니다. 그럼 매개변수 값들을 어떻게 조합하면 NAND 게이트가 만들어질까요?

그림 2-3 NAND 게이트의 진리표

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

NAND 게이트를 표현하려면 예를 들어 $(w_1, w_2, \theta) = (-0.5, -0.5, -0.7)$ 조합이 있습니다 (다른 조합도 무한히 있지요). 사실 AND 게이트를 구현하는 매개변수의 부호를 모두 반전하기만 하면 NAND 게이트가 됩니다.

같은 흐름에서 [그림 2-4]의 OR 게이트도 생각해봅시다. OR 게이트는 입력 신호 중 하나 이상이 1이면 출력이 1이 되는 논리 회로입니다. 이 OR 게이트의 매개변수는 어떻게 설정하면 될까요? 생각해보세요!

그림 2-4 OR 게이트의 진리표

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

NOTE 여기서 페센트론의 매개변수 값을 정하는 것은 컴퓨터가 아니라 우리 인간입니다. 인간이 직접 진리표라는 ‘학습 데이터’를 보면서 매개변수의 값을 생각합니다. 기계학습 문제는 이 매개변수의 값을 정하는 작업을 컴퓨터가 자동으로 하도록 합니다. 학습이란 적절한 매개변수 값을 정하는 작업이며, 사람은 페센트론의 구조(모델)를 고민하고 컴퓨터에 학습할 데이터를 주는 일을 합니다.

이상과 같이 퍼셉트론으로 AND, NAND, OR 논리 회로를 표현할 수 있음을 알았습니다. 여기서 중요한 점은 퍼셉트론의 구조는 AND, NAND, OR 게이트 모두에서 똑같다는 것입니다. 세 가지 게이트에서 다른 것은 매개변수(기중치와 임계값)의 값뿐입니다. 즉, 마치 팔색조 배우가 다양한 인물을 연기하는 것처럼 똑같은 구조의 퍼셉트론이 매개변수의 값만 적절히 조정하여 AND, NAND, OR로 변신하는 것입니다.

2.3 퍼셉트론 구현하기

2.3.1 간단한 구현부터

이제 논리 회로를 파이썬으로 구현해봅시다. 다음은 x_1 과 x_2 를 인수로 받는 AND라는 함수입니다.

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.7
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

매개변수 w_1 , w_2 , θ 는 함수 안에서 초기화하고, 가중치를 곱한 입력의 총합이 임계값을 넘으면 1을 반환하고 그 외에는 0을 반환합니다. 이 함수의 출력이 [그림 2-2]와 같은지 확인해봅시다.

```
AND(0, 0) # 0을 출력
AND(1, 0) # 0을 출력
AND(0, 1) # 0을 출력
AND(1, 1) # 1을 출력
```

가장 대로 잘 작동하는군요! 이상으로 AND 게이트를 구현했습니다. NAND 게이트와 OR 게이트도 같은 식으로 구현할 수 있지만, 그 전에 이 구현을 조금만 손보고 싶습니다.

2.3.2 기중치와 편향: 도입

앞에서 구현한 AND 계이트는 직관적이고 알기 쉽지만, 앞으로를 생각해서 다른 방식으로 수 정하고자 합니다. 그 전에 [식 2.1]의 θ 를 $-b$ 로 치환하면 퍼셉트론의 동작이 [식 2.2]처럼 됩니다.

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

[식 2.2]

[식 2.1]과 [식 2.2]는 기호 표기만 바꿨을 뿐, 그 의미는 같습니다. 여기에서 b 를 편향(bias)이라고 하며 w_1 과 w_2 는 그대로 기중치(weight)입니다. [식 2.2] 관점에서 해석해보자면, 퍼셉트론은 입력 신호에 기중치를 곱한 값과 편향을 합하여, 그 값이 0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력합니다. 그럼 넘파이를 사용해서 [식 2.2] 방식으로 구현해봅시다. 여기에서는 파이썬 인터프리터로 순서대로 결과를 확인하면서 진행해보겠습니다.

```
>>> import numpy as np
>>> x = np.array([0, 1])          # 입력
>>> w = np.array([0.5, 0.5])      # 가중치
>>> b = -0.7                      # 편향
>>> w*x
array([ 0. ,  0.5])
>>> np.sum(w*x)
0.5
>>> np.sum(w*x) + b
-0.1999999999999996 # 대략 -0.2 (부동소수점 수에 의한 연산 오차)
```

넘파이 배열끼리의 곱셈은 두 배열의 원소 수가 같다면 각 원소끼리 곱합니다. 그래서 이 예의 $w*x$ 에서는 인덱스가 같은 원소끼리 곱합니다($[0, 1] * [0.5, 0.5] \Rightarrow [0, 0.5]$). 또, $np.sum()$ 메서드는 입력한 배열에 담긴 모든 원소의 총합을 계산합니다. 이 가중치에 편향을 더하면 [식 2.2]의 계산이 완료됩니다.

2.3.3 기중치와 편향 구현하기

'기중치와 편향을 도입한 AND 계이트는 다음과 같이 구현할 수 있습니다.'

```

def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1

```

여기에서 $-\theta_7$ 편향 b 로 치환되었습니다(2.3.1절에서 구현한 AND의 theta가 $-b$ 가 되었습 니다). 그리고 편향은 가중치 w_1, w_2 와 기능이 다르다는 사실에 주의합시다. 구체적으로 말하면 w_1 과 w_2 는 각 입력 신호가 결과에 주는 영향력(중요도)을 조절하는 매개변수고, 편향은 뉴런이 얼마나 쉽게 활성화(결과로 1을 출력)하느냐를 조정하는 매개변수입니다. 예를 들어 b 가 -0.1 이면 각 입력 신호에 가중치를 곱한 값들의 합이 0.1을 초과할 때만 뉴런이 활성화합니다. 반면 b 가 -20.0 이면 각 입력 신호에 가중치를 곱한 값들의 합이 20.0을 넘지 않으면 뉴런은 활성화하지 않습니다. 이처럼 편향의 값은 뉴런이 얼마나 쉽게 활성화되는지를 결정합니다. 한편 w_1 과 w_2 는 ‘가중치’로, b 는 ‘편향’으로 서로 구별하기도 합니다만, 이 책에서는 문맥에 따라 셋 모두를 ‘가중치’라고 할 때도 있습니다.

NOTE 편향이라는 용어는 한쪽으로 치우쳐 균형을 깬다라는 의미를 담고 있습니다. 실제로 [식 2.2]는 두 입력이 모두 0이어도 결과로 (0|0)인 편향 값을 출력합니다:

이어서 NAND 게이트와 OR 게이트를 구현해봅시다.

```

def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5]) # AND와는 가중치(w와 b)만 다르다!
    b = 0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1

def OR(x1, x2):

```

```

x = np.array([x1, x2])
w = np.array([0.5, 0.5])    # AND와는 가중치(w와 b)만 다르다!
b = -0.2

tmp = np.sum(w*x) + b
if tmp <= 0:
    return 0
else:
    return 1

```

앞 절에서 AND, NAND, OR는 모두 같은 구조의 퍼셉트론이고, 차이는 가중치 매개변수의 값뿐이라 합니다. 실제로 파이썬으로 작성한 NAND와 OR 케이트의 코드에서도 AND와 다른 곳은 가중치와 편향 값을 설정하는 부분뿐입니다.

2.4 퍼셉트론의 한계

지금까지 살펴본 것처럼 퍼셉트론을 이용하면 AND, NAND, OR의 3가지 논리 회로를 구현할 수 있습니다. 계속해서 XOR 케이트도 생각해보죠.

2.4.1 도전! XOR 케이트

XOR 케이트는 **비타직 논리함수**라는 논리 회로입니다. [그림 2-5]와 같o| x_1 과 x_2 중 한쪽이 1일 때만 1을 출력합니다 ('비타직'이란 자기 외에는 거부한다는 의미죠). 자, 이 XOR 케이트를 퍼셉트론으로 구현하려면 가중치 매개변수 값을 어떻게 설정하면 될까요?

그림 2-5 XOR 케이트의 진리표

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

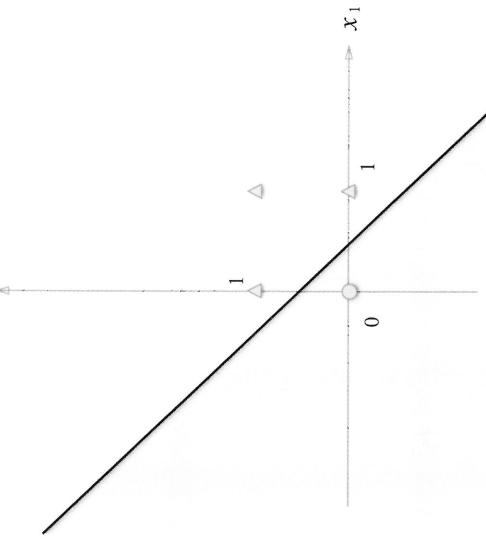
사실 지금까지 본 퍼셉트론으로는 이 XOR 게이트를 구현할 수 없습니다. 왜 AND와 OR는 되고 XOR는 안 될까요? 그림으로 그려가며 시각적으로 설명해보겠습니다.

우선 OR 게이트의 동작을 시각적으로 생각해보죠. OR 게이트는, 예를 들어 기종치 매개변수 가 $(b, w_1, w_2) = (-0.5, 1.0, 1.0)$ 일 때 [그림 2-4]의 진리표를 만족합니다. 이때의 퍼셉트론은 [식 2.3]으로 표현됩니다.

$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leq 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases} \quad [\text{식 2.3}]$$

[식 2.3]의 퍼셉트론은 직선으로 나뉜 두 영역을 만듭니다. 직선으로 나뉜 한쪽 영역은 1을 출력하고 다른 한쪽은 0을 출력합니다. 이를 그려보면 [그림 2-6]처럼 됩니다.

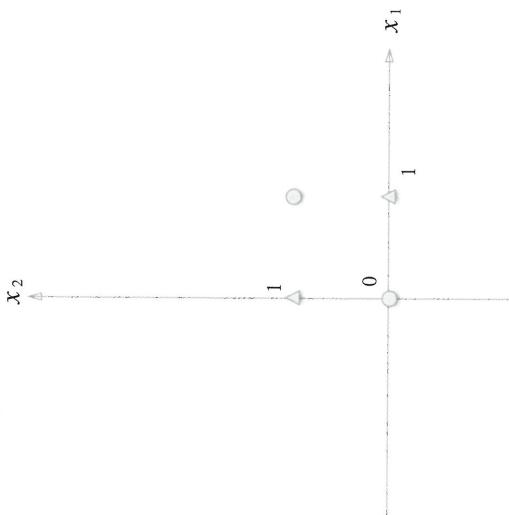
그림 2-6 퍼셉트론의 시각화 : 화색 영역은 0을 출력하는 영역이며, 전체 영역은 OR 게이트의 성질을 만족합니다.



OR 게이트는 $(x_1, x_2) = (0, 0)$ 일 때 0을 출력하고 $(0, 1), (1, 0), (1, 1)$ 일 때는 1을 출력합니다. 그림에서는 0을 원(\circ), 1을 삼각형(\triangle)으로 표시했습니다. OR 게이트를 만들려면 [그림 2-6]의 ○과 \triangle 을 직선으로 나눠야 합니다. 실제로 이 그림의 직선은 네 점을 제대로 나누고 있습니다.

그림 XOR 게이트의 경우는 어떨까요? OR 게이트 때처럼 직선 하나로 ○과 △을 나누는 영역을 만들어낼 수 있을까요?

그림 2-7 ○과 △은 XOR 게이트의 출력을 나타낸다. 직선 하나로 ○과 △을 나누는 영역을 만들 수 있을까?

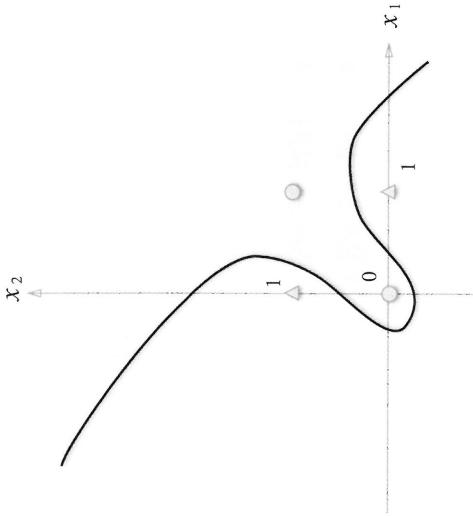


[그림 2-7]의 ○과 △을 직선 하나로 나누는 방법은 아무리 생각해도 떠오르지 않습니다. 사실 직선 하나로 나누기란 불가능합니다.

2.4.2 선형과 비선형

직선 하나로는 [그림 2-7]의 ○과 △을 나눌 수 없습니다. 하지만 '직선'이라는 제약을 없앤다면 가능하죠. 예를 들어 [그림 2-8]처럼 나눌 수 있습니다.

그림 2-8 곡선이라면 ○과 △을 나눌 수 있다.



퍼셉트론은 직선 하나로 나눈 영역만 표현할 수 있다는 한계가 있습니다. [그림 2-8] 같은 곡선은 표현할 수 없다는 것이죠. 몇몇여서 [그림 2-8]과 같은 곡선의 영역을 비선형 영역, 직선의 영역을 선형 영역이라고 합니다. 선형, 비선형이라는 말은 기계학습 분야에서 자주 쓰이는 용어로, [그림 2-6]과 [그림 2-8] 같은 이미지를 떠올리시면 됩니다.

2.5 다층 퍼셉트론이 출동한다면

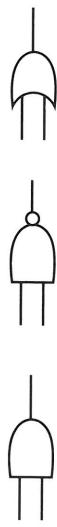
안타깝게도 퍼셉트론으로는 XOR 게이트를 표현할 수 없었습니다. 그렇다고 슬퍼할 필요는 없습니다. 사실 퍼셉트론의 아름다움은 ‘총을 쏘아’ 다중 퍼셉트론 multi-layer perceptron을 만들 수 있다는 테 있습니다. 이번 절에서는 총을 하나 더 쏘아서 XOR를 표현해볼 것입니다. ‘총을 쏘는다’는 말의 뜻은 잠시 뒤에 살펴보기로 하고, 우선은 XOR 게이트 문제를 다른 관점에서 생각해보기로 합시다.

2.5.1 기존 게이트 조합하기

XOR 게이트를 만드는 방법은 다양합니다. 그중 하나는 앞서 만든 AND, NAND, OR 게이트를 조합하는 방법입니다. 여기에서는 AND, NAND, OR 게이트를 [그림 2-9]와 같은 기호로

표기합니다. 참고로 [그림 2-9]의 NAND 게이트 출력부에 있는 ○ 기호는 출력을 반전한다는 뜻입니다.

그림 2-9 AND, NAND, OR 게이트 기호



AND



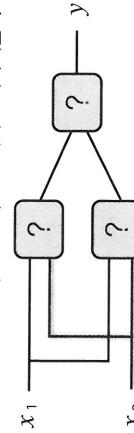
NAND



OR

그럼 XOR 게이트를 만들려면 AND, NAND, OR를 어떻게 조합하면 될까요? 각자 생각해봅시다. 헌트! [그림 2-10]의 ‘?’에 세 가지 게이트를 하나씩 대입하면 XOR를 완성할 수 있습니다.

그림 2-10 AND, NAND, OR 게이트 하나씩을 ‘?’에 대입해 XOR를 완성하라!



NOTE 앞 절에서 말한 패센트론의 한계는 정확히 말하면 “단층 패센트론single-layer perceptron으로는 XOR 게이트를 표현할 수 없다” 또는 “단층 패센트론으로는 비선형 영역을 분리할 수 없다”가 됩니다. 앞으로는 패센트론을 조합하여, 즉 층을 쌓아서 XOR 게이트를 구현하는 모습을 보게 됩니다.

[그림 2-11]과 같은 조합이라면 XOR 게이트를 구현할 수 있습니다. x_1 과 x_2 가 입력 신호, y 가 출력 신호입니다. x_1 과 x_2 는 NAND와 OR 게이트의 입력이 되고, NAND와 OR의 출력이 AND 게이트의 입력으로 들어집니다.

그림 2-11 AND, NAND, OR 게이트를 조합해 구현한 XOR 게이트

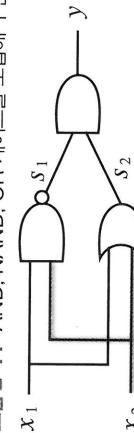


그림 [그림 2-11]의 조합이 정답 XOR를 구현하는지 살펴봅시다. NAND의 출력을 s_1 , OR의 출력을 s_2 로 해서 진리표를 만들면 [그림 2-12]처럼 됩니다. x_1, x_2, y 에 주목하면 분명히 XOR의 출력과 같습니다.

그림 2-12 XOR 게이트의 진리표

x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

2.5.2 XOR 게이트 구현하기

이어서 [그림 2-11]처럼 조합된 XOR 게이트를 파이썬으로 구현해보겠습니다. 지금까지 정의한 함수 AND, NAND, OR를 사용하면 다음과 같이 (쉽게!) 구현할 수 있습니다.

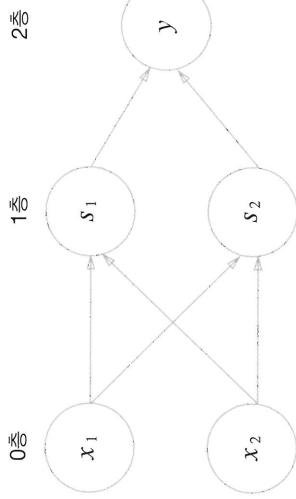
```
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```

○ XOR 합수는 기대한 대로의 결과를 출력합니다.

```
XOR(0, 0) # 0을 출력
XOR(1, 0) # 1을 출력
XOR(0, 1) # 1을 출력
XOR(1, 1) # 0을 출력
```

이로써 XOR 게이트를 완성했습니다. 지금 구현한 XOR를 누군가 이용한 퍼셉트론으로 표현하면 [그림 2-13]처럼 됩니다.

그림 2-13 XOR의 퍼셉트론



XOR는 [그림 2-13]과 같은 다층 구조의 네트워크입니다. 이 챕에서는 원쪽부터 차례로 0층, 1층, 2층이라고 부르겠습니다.

그런데 [그림 2-13]의 퍼셉트론은 지금까지 본 AND, OR 퍼셉트론(그림 2-1)과 형태가 다릅니다. 실제로 AND, OR가 단층 퍼셉트론인 데 반해, XOR는 2층 퍼셉트론입니다. 이처럼 층이 여러 개인 퍼셉트론을 다층 퍼셉트론이라 합니다.

WARNING [그림 2-13]의 퍼셉트론은 모두 3층으로 구성됩니다만, 기중치를 갖는 층은 사실 2개(0층과 1층 시0!, 1층과 2층 시0!)뿐이니 2층 퍼셉트론이라 부르기로 합시다. 문헌에 따라서는 구성 층의 수를 기준으로 3층 퍼셉트론이라 하는 경우도 있습니다.

[그림 2-13]과 같은 2층 퍼셉트론에서는 0층에서 1층으로 신호가 전달되고, 이어서 1층에서 2층으로 신호가 전달됩니다. 이 동작을 더 자세히 서술하면 다음과 같습니다.

1. 0층의 두 뉴런이 입력 신호를 받아 1층의 뉴런으로 신호를 보낸다.
2. 1층의 뉴런이 2층의 뉴런으로 신호를 보내고, 2층의 뉴런은 y 를 출력한다.

덧붙여서, 이 2층 퍼셉트론의 동작을 공장의 조립라인에 비유할 수 있습니다. 1단(1층째) 작업자는 흘러오는 ‘부품’을 다듬어 일이 완료되면 2단(2층째) 작업자에게 전네줍니다. 2단의 작업자는 1단 작업자로부터 전달받은 ‘부품’을 다듬어 완성품으로 만들어 출하(출력)합니다. 이처럼 XOR 게이트 퍼셉트론에서는 작업자들 사이에서 부품을 전달하는 일이 이루어집니다.

이상으로 2층 구조를 사용해 퍼셉트론으로 XOR 게이트를 구현할 수 있게 되었습니다. 다시 말해 **단층 퍼셉트론으로는 표현하지 못한 것을 층을 하나 늘려 구현할 수 있었습니다.** 이처럼 퍼셉트론을 쌓아(깊게 하여) 더 다양한 것을 표현할 수 있답니다.

2.6 NAND에서 컴퓨터까지

다중 퍼셉트론은 지금까지 보이온 회로로보다 복잡한 회로를 만들 수 있습니다. 예를 들면, 텃셈을 처리하는 가산기도 만들 수 있습니다. 2진수를 10진수로 변환하는 인코더, 어떤 조건을 충족하면 1을 출력하는 회로(페리티 검사 회로)도 퍼셉트론으로 표현할 수 있습니다. 사실은 페셀트론을 이용하면 '컴퓨터'마저 표현할 수 있습니다!

컴퓨터는 정보를 처리하는 기계죠. 컴퓨터에 무언가를 입력하면 정해진 방법으로 처리하고 그 결과를 출력합니다. 정해진 방법으로 처리한다는 것은 컴퓨터도 마치 퍼셉트론처럼 입력과 출력으로 구성된 특정 규칙대로 계산을 수행한다는 뜻입니다.

컴퓨터 내부에서 이뤄지는 처리가 매우 복잡할 거 같지만, 사실은 (놀랍게도) NAND 게이트의 조합만으로 컴퓨터가 수행하는 일을 재현할 수 있습니다. NAND 게이트만으로 컴퓨터를 만들 수 있다? 이 말은 곧 퍼셉트론으로도 컴퓨터를 표현할 수 있다는 놀라운 사실로 이어집니다. 지금까지 살펴본 것처럼 NAND 게이트는 퍼셉트론으로 만들 수 있기 때문이죠.

NOTE 'NAND 게이트의 조합만으로 컴퓨터를 만든다'라는 말이 믿어지지 않을지도 모르겠네요. 이것 이 어떻게 가능한지 궁금한 분께는 「The Elements of Computing Systems: Building a Modern Computer from First Principles」(The MIT Press, 2005)를 알아보시길 권합니다. 이 책은 컴퓨터를 깊이 이해하고자 "NAND에서 테트리스까지!"라는 구호 아래, 실제로 NAND로 테트리스가 작동하는 컴퓨터를 만듭니다. 이 책을 읽으면 NAND라는 단순한 소자만으로 컴퓨터와 같은 복잡한 시스템이 만들어진다는 것을 실감할 수 있을 겁니다.

이처럼 다중 퍼셉트론은 컴퓨터도 만들 정도로 복잡한 표현을 해냅니다. 대전하죠! 그럼 어떤 구조의 퍼셉트론이면 컴퓨터를 표현할 수 있을까요? 층을 얼마나 깊게 하면 컴퓨터가 만들어질까요?

그 답은 “이론상 2층 퍼셉트론이면 컴퓨터를 만들 수 있다”입니다. 말도 안 되는 소리 같지만, 2층 퍼셉트론, 정확히는 비선형인 시그모이드 함수를 활성화 함수로 이용하면 임의의 함수를 표현할 수 있다는 사실이 증명되었습니다(3장 참고). 그러나 2층 퍼셉트론 구조에서 가중치를 적절히 설정하여 컴퓨터를 만들기란 너무 어렵습니다. 실제로도 NAND 등의 저수준 소자에서 시작하여 컴퓨터를 만드는 데 필요한 부품(모듈)을 단계적으로 만들어가는 쪽이 자연스러운 방법입니다. 즉, 처음에는 AND와 OR 게이트, 그다음에는 반기선기와 전기산기, 그다음에는

산술 논리 연산 장치(ALU), 그다음에는 CPU라는 식이죠. 그래서 퍼셉트론으로 표현하는 컴퓨터도 여러 층을 다시 층으로 핵친 구조로 만드는 방향이 자연스러운 흐름입니다. 이 책에서는 컴퓨터를 만들지 않습니다. 그래도 퍼셉트론은 층을 거듭 쌓으면 비선형적인 표현도 가능하고, 이런 상 컴퓨터가 수행하는 처리도 모두 표현할 수 있다는 점을 기억해주세요.

2.7 정리

이번 장에서는 퍼셉트론을 배웠습니다. 퍼셉트론은 간단한 알고리즘이라 그 구조를 쉽게 이해할 수 있습니다. 퍼셉트론은 다음 장에서 배울 신경망의 기초가 됩니다. 그러나 이번 장에서 배운 내용은 아주 중요합니다.

이번 장에서 배운 내용

- 퍼셉트론은 입출력을 갖춘 알고리즈다. 입력을 주면 정해진 규칙에 따른 값을 출력한다.
- 퍼셉트론에서는 '기준치'와 '편향'을 매개변수로 설정한다.
- 퍼셉트론으로 AND, OR 게이트 등의 논리 회로를 표현할 수 있다.
- XOR 게이트는 단종 퍼셉트론으로는 표현할 수 없다.
- 2층 퍼셉트론을 이용하면 XOR 게이트를 표현할 수 있다.
- 단종 퍼셉트론은 직선형 영역만 표현할 수 있고, 다중 퍼셉트론은 비선형 영역도 표현할 수 있다.
- 다중 퍼셉트론은 (이론상) 컴퓨터를 표현할 수 있다.

앞 장 배운 퍼셉트론 팔련에서는 좋은 소식과 나쁜 소식이 있었습니다. 좋은 소식은 퍼셉트론으로 복잡한 힘수도 표현할 수 있다는 것입니다. 그 예로 컴퓨터가 수행하는 복잡한 처리도 퍼셉트론으로 (이론상) 표현할 수 있음을 앞 장에서 설명했습니다. 나쁜 소식은 기증치를 설정하는 작업(원하는 결과를 출력하도록 가중치 값을 적절히 정하는 작업)은 여전히 사람이 수동으로 한다는 것입니다. 앞 장에서는 AND, OR 게이트의 진리표를 보면서 우리 인간이 적절한 기증치 값을 정했습니다.

신경망은 이 나쁜 소식을 해결해줍니다. 무슨 말인고 하니, 기증치 매개변수의 적절한 값을 데이터로부터 자동으로 학습하는 능력이 이제부터 살펴볼 신경망의 중요한 성질입니다. 이번 장에서는 신경망의 개요를 설명하고, 신경망이 입력 테이터가 무엇인지 식별하는 처리 과정을 자세히 알아봅니다. 이십지만 테이터에서 기증치 매개변수 값을 학습하는 방법은 다음 장까지 기다려셔야 합니다.

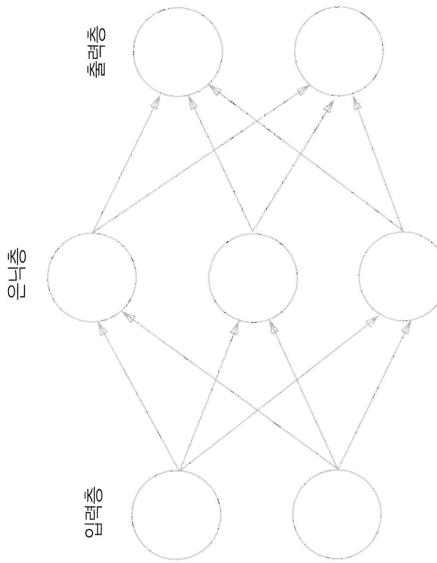
3.1 퍼셉트론에서 신경망으로

신경망은 앞 장에서 설명한 퍼셉트론과 공통점이 많습니다. 이번 절에서는 퍼셉트론과 다른 점을 중심으로 신경망의 구조를 설명합니다.

3.1.1 신경망의 예

신경망을 그림으로 나타내면 [그림 3-1]처럼 됩니다. 여기에서 가장 왼쪽 줄을 입력층, 맨 오른쪽 줄을 출력층, 중간 줄을 은닉층이라고 합니다. 은닉층의 뉴런은 (입력층이나 출력층과 달리) 사람 눈에는 보이지 않습니다. 그래서 ‘은닉’인 것이죠. 또한, 이 책에서는 입력층에서 출력층 방향으로 차례로 0층, 1층, 2층이라 하겠습니다(층 번호를 0부터 시작하는 이유는 파이썬 배열의 인덱스도 0부터 시작하여, 나중에 구현할 때 짹짓기 편하기 때문입니다). [그림 3-1]에서는 0층이 입력층, 1층이 은닉층, 2층이 출력층이 됩니다.

그림 3-1 신경망의 예



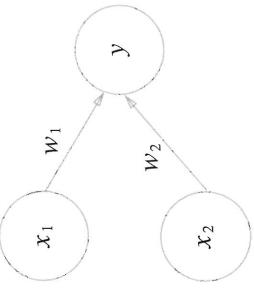
WARNING [그림 3-1]의 신경망은 모두 3층으로 구성됩니다만, 기중치를 갖는 총은 2개뿐이기 때문에 ‘2층 신경망’이라고 합니다. 문헌에 따라서는 신경망을 구성하는 층수를 기준으로 [그림 3-1]을 ‘3층 신경망’이라고 하는 경우도 있으니 주의해야 합니다. 이 책에서는 실제로 기중치를 갖는 층의 개수(입력층, 은닉층, 출력층의 합계에서 1을 뺀 값)를 기준으로 하겠습니다.

[그림 3-1]은 앞 장에서 본 퍼셉트론과 특별히 달라 보이지 않습니다. 실제로 뉴런이 연결되는 방식은 앞 장의 퍼셉트론에서 달라진 것이 없답니다. 자, 그럼 신경망에서는 신호를 어떻게 전달할까요?

3.1.2 퍼셉트론 복습

신경망의 신호 전달 방법을 보기 전에 퍼셉트론을 살짝 복습해보죠. 먼저 [그림 3-2]와 같은 구조의 네트워크를 생각해봅시다.

그림 3-2 퍼셉트론 복습



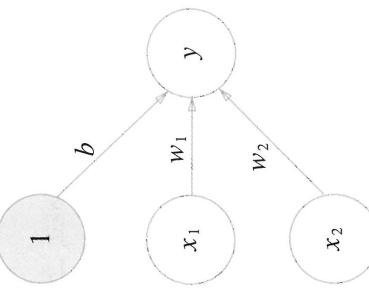
[그림 3-2]는 x_1 과 x_2 라는 두 신호를 입력받아 y 를 출력하는 퍼셉트론입니다. 이 퍼셉트론을 수식으로 나타내면 [식 3.1]이 됩니다.

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \quad [\text{식 3.1}]$$

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

그런데 [그림 3-2]의 네트워크에는 편향 b 가 보이지 않습니다. 여기에 편향을 명시한다면 [그림 3-3]과 같이 나타낼 수 있습니다.

그림 3-3 편향을 명시한 퍼셉트론



[그림 3-3]에서는 가중치가 b 이고 입력이 1인 뉴런이 추가되었습니다. 이 퍼셉트론의 동작은 $x_1, x_2, 1$ 이라는 3개의 신호가 뉴런에 입력되어, 각 신호에 가중치를 곱한 후, 다음 뉴런에 전달 됩니다. 다음 뉴런에서는 이 신호들의 값을 더하여, 그 합이 0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력합니다. 참고로, 편향의 입력 신호는 항상 1이기 때문에 그림에서는 헤딩 뉴런을 회색으로 차워 다른 뉴런과 구별했습니다.

그럼 [식 3.1]을 터 간결한 형태로 다시 작성해보죠. 이를 위해서 조건 분기의 동작(0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력)을 하나의 함수로 나타냅니다. 이 함수를 $h(x)$ 라 하면 [식 3.1]을 다음과 같이 [식 3.2]와 [식 3.3]으로 표현할 수 있습니다.

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

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

[식 3.2]는 입력 신호의 총합이 $h(x)$ 라는 함수를 거쳐 변환되어, 그 변환된 값이 y 의 출력이 됨을 보여줍니다. 그리고 [식 3.3]의 $h(x)$ 함수는 입력이 0을 넘으면 1을 돌려주고 그렇지 않으면 0을 돌려줍니다. 결과적으로 [식 3.1]이 하는 일과 [식 3.2]와 [식 3.3]이 하는 일은 같습니다.

3.1.3 활성화 함수의 등장

조금 전 $h(x)$ 라는 함수가 등장했는데, 이처럼 입력 신호의 총합을 출력 신호로 변환하는 함수를 일반적으로 **활성화 함수 activation function**라 합니다. ‘활성화’라는 이름이 말해주듯 활성화 함수는 입력 신호의 총합이 활성화를 일으키는지를 정하는 역할을 합니다.

그럼 [식 3.2]를 다시 써봅시다. [식 3.2]는 가중치가 곱해진 입력 신호의 총합을 계산하고, 그 합을 활성화 함수에 입력해 결과를 내는 2단계로 처리됩니다. 그래서 이 식은 다음과 같은 2개의 식으로 나눌 수 있습니다.

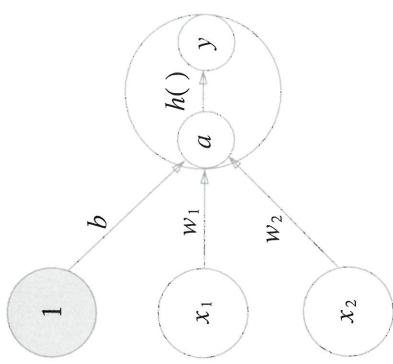
$$a = b + w_1x_1 + w_2x_2 \quad [\text{식 3.4}]$$

$$y = h(a) \quad [\text{식 3.5}]$$

[식 3.4]는 기중치가 달린 입력 신호와 편향의 총합을 계산하고, 이를 a 라 합니다. 그리고 [식 3.5]는 a 를 함수 $h()$ 에 넣어 y 를 출력하는 흐름입니다.

지금까지와 같이 뉴런을 큰 원(○)으로 그려보면 [식 3.4]와 [식 3.5]는 [그림 3-4]처럼 나타낼 수 있습니다.

그림 3-4 활성화 함수의 처리 과정



보시다시피 [그림 3-4]에서는 기존 뉴런의 일을 키우고, 그 안에 활성화 함수의 처리 과정을 명시적으로 그려 넣었습니다. 즉, 기중치 신호를 조합한 결과가 a 라는 노드가 되고, 활성화 함수 $h()$ 를 통과하여 y 라는 노드로 변환되는 과정이 분명하게 나타나 있습니다. 참고로 이 책에서는 **뉴런과 노드**라는 용어를 같은 의미로 사용합니다. 방금 a 와 y 의 원을 노드라고 했는데, 이는 지금까지 뉴런이라고 한 것과 같은 의미입니다.

뉴런을 그릴 때 보통은 지금까지와 마찬가지로 [그림 3-5]의 왼쪽처럼 뉴런을 하나의 원으로 그립니다. 그리고 신경망의 동작을 더 명확히 드러내고자 할 때는 오른쪽 그림처럼 활성화 처리 과정을 명시하기도 합니다.

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



그럼 계속해서 활성화 함수의 더 친해져보기로 하죠. 이 활성화 함수가 퍼셉트론에서 신경망으로 가기 위한 길잡이랍니다.

WARNING_ 이 책에서는 퍼셉트론이라는 말이 기린다는 알고리즘을 엄밀히 통일하지는 않았습니다. 일반적으로 단순 퍼셉트론은 단층 네트워크에서 계단 함수(임계값을 경계로 출력이 바꾸는 함수)를 활성화 함수로 사용한 모델을 기리키고 **다중 퍼셉트론은 신경망(여러 층으로 구성되고 시그모이드 함수 등의 매끈한 활성화 함수를 사용하는 네트워크)을 기리킵니다.**

3.2 활성화 함수

[식 3.3]과 같은 활성화 함수는 입계값을 경계로 출력이 바뀌는데, 이런 함수를 **계단 함수** function라 합니다. 그래서 “퍼셉트론에서는 활성화 함수로 계단 함수를 이용한다”라 할 수 있습니다. 즉, 활성화 함수로 쓸 수 있는 여러 후보 중에서 퍼셉트론은 계단 함수를 채용하고 있습니다. 그렇다면 계단 함수 이외의 함수를 사용하면 어떻게 될까요? 실은 활성화 함수를 계단 함수에서 다른 함수로 변경하는 것이 신경망의 세계로 나아가는 열쇠입니다! 그럼 어서 신경망에서 이용하는 활성화 함수를 소개하겠습니다.

3.2.1 시그모이드 함수

다음은 신경망에서 자주 이용하는 활성화 함수인 시그모이드 함수 sigmoid function를 나타낸 식입니다.

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

[식 3.6]에서 $\exp(-x)$ 은 e^{-x} 를 뜻하며, e 는 자연상수로 2.7182...의 값을 갖는 실수입니다. [식 3.6]으로 나타나는 시그모이드 함수는 얼핏 복잡해 보이지만 이 역시 단순한 ‘함수’일 뿐입니다. 함수는 입력을 주면 출력을 돌려주는 변환기죠. 예를 들어 시그모이드 함수에 1.0과 2.0을 입력하면 $h(1.0) = 0.731\dots$, $h(2.0) = 0.880\dots$ 처럼 특정 값을 출력합니다.

신경망에서는 활성화 함수로 시그모이드 함수를 이용하여 신호를 변환하고, 그 변환된 신호를 다음 뉴런에 전달합니다. 사실 앞장에서 본 패셉트론과 앞으로 불 신경망의 주된 차이는 이 활성화 함수뿐입니다. 그 외에 뉴런이 여러 층으로 이어지는 구조와 신호를 전달하는 방법은 기본적으로 앞에서 살펴본 패셉트론과 같습니다. 그러면 활성화 함수로 이용되는 시그모이드 함수를 계단 함수와 비교하면서 자세히 살펴보겠습니다.

3.2.2 계단 함수 구현하기

이번 절과 다음 절에서는 파이썬으로 계단 함수를 그려보겠습니다(함수의 형태를 눈으로 확인해보면 그 함수를 이해하는 데 큰 도움이 됩니다). 계단 함수는 [식 3.3]과 같이 입력이 0을 넘으면 1을 출력하고, 그 외에는 0을 출력하는 함수입니다. 다음은 이러한 계단 함수를 단순하게 구현한 것입니다.

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

이 구현은 단순하고 쉽지만, 인수 x 는 실수(부동소수점)만 받아들입니다. 즉, `step_function(3.0)`은 되지만 넘파이 배열을 인수로 넣을 수는 없습니다. 가령 `step_function(np.array([1.0, 2.0]))`은 안 됩니다. 우리는 앞으로를 위해 넘파이 배열도 지원하도록 수정하고 싶습니다. 그러기 위해서는 가령 다음과 같은 구현을 생각할 수 있겠죠.

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

마지막 줄이라 엉성해 보이겠지만, 이는 넘파이의 편리한 트릭을 사용한 덕분이죠. 어떤 트릭을 썼는지는 다음 파이썬 인터프리터의 예를 보면서 설명해보죠. 다음 예에서는 넘파이 배열을 준비하고 그 넘파이 배열에 부등호 연산을 수행합니다.

```
>>> 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 생성됩니다. 이 예에서는 배열 x의 원소 각각이 0보다 크면 True로, 0 이하면 False로 변환한 새로운 배열 y가 생성됩니다.

이 y는 bool 배열입니다. 그런데 우리가 원하는 계단 함수는 0이나 1의 'int형'을 출력하는 함수죠. 그래서 배열 y의 원소를 bool에서 int형으로 바꿔줍니다.

```
>>> y = y.astype(np.int)
>>> y
array([0, 1, 1])
```

이처럼 넘파이 배열의 자료형을 변환할 때는 astype() 메서드를 이용합니다. 원하는 자료형 (이 예에서는 np.int)을 인수로 지정하면 되죠. 그리고 파이썬에서는 bool을 int로 변환하면 True는 1로, False는 0으로 변환됩니다. 이상이 계단 함수 구현에서 사용한 넘파이의 '트랙'이었습니다.

3.2.3 계단 함수의 그래프

이제 앞에서 정의한 계단 함수를 그래프로 그려봅시다. 이를 위해 matplotlib 라이브러리를 사용합니다.

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

def step_function(x):
```

```

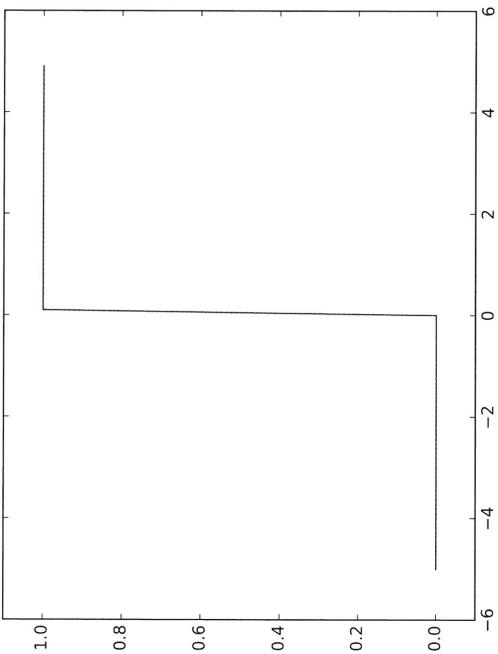
return np.array(x > 0, dtype=np.int)

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 -5.0 에서 5.0 전까지 0.1 간격의 넘파이 배열을 생성합니다. 즉, $[-5.0, -4.9, \dots, 4.9]$ 를 생성합니다. `step_function()`은 인수로 받은 넘파이 배열의 원소 각각을 인수로 계단 함수 실행해, 그 결과를 다시 배열로 만들어 돌려줍니다. 이 `x, y` 배열을 그래프로 그리면(`plot`) 결과는 [그림 3-6]처럼 됩니다.

그림 3-6 계단 함수의 그래프



[그림 3-6]에서 보듯 계단 함수는 0을 경계로 출력이 0에서 1(또는 1에서 0)로 바뀝니다. 이제 ‘계단’ 함수로 불리는 이유를 아셨겠죠? 바로 이 그림처럼 값이 바뀌는 형태가 계단처럼 생겼기 때문이랍니다.

3.2.4 시그모이드 함수 구현하기

이어서 시그모이드 함수를 구현합시다. [식 3.6]의 시그모이드 함수는 파이|써으로 다음과 같이 작성할 수 있습니다.

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

여기서 `np.exp(-x)`는 `exp(-x)` 수식에 해당합니다. 이 구현에서 특별히 어려운 건 없고, 인수 `x`가 넘파이 배열이어도 올바른 결과가 나오다는 정도만 기억해둡시다. 실제로 넘파이 배열을 제대로 처리하는지 실험해보죠.

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

이 함수가 넘파이 배열도 훌륭히 처리해줄 수 있는 버밀은 넘파이의 브로드캐스트에 있습니다 (“1.5.5 브로드캐스트” 참고). 브로드캐스트 기능이란 넘파이 배열과 스칼라값의 연산을 넘파이 배열의 원소 각각과 스칼라값의 연산으로 바꿔 수행하는 것입니다. 복습 겸 구체적인 예를 하나 보시죠.

```
>>> 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])
```

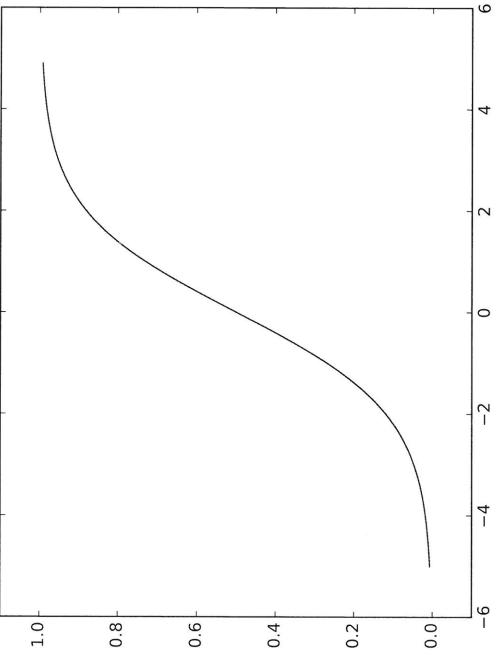
이 예에서는 스칼라값 1.0과 넘파이 배열 사이에서 수치 연산(+와 /)을 해보았습니다. 결과적으로 스칼라값과 넘파이 배열의 각 원소 사이에서 연산이 이루어지고, 연산 결과가 넘파이 배열로 출력되었습니다. 앞에서 구현한 `sigmoid` 함수에서도 `np.exp(-x)`가 넘파이 배열을 반환하기 때문에 `1 / (1 + np.exp(-x))`도 넘파이 배열의 각 원소에 연산을 수행한 결과를 내어줍니다.

그럼, 시그모이드 함수를 그래프로 그려볼까요? 그레프를 그리는 코드는 앞 절의 계단 함수 그리기 코드와 거의 같습니다. 유일하게 다른 부분은 y 를 출력하는 함수를 sigmoid 함수로 변경 한 곳입니다.

```
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 그레프를 보실 수 있습니다.

그림 3-7 시그모이드 함수의 그래프*

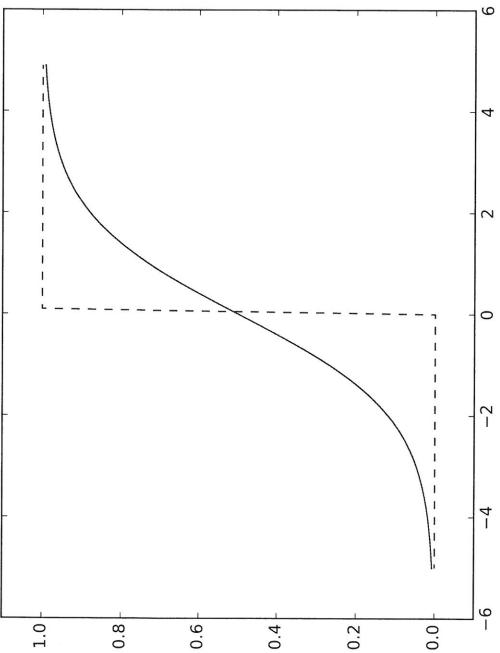


* 옮긴이] 시그모이드(sigmoid)란 'S자 모양'이라는 뜻입니다. 계단 함수처럼 그 모양을 따 이름 지은 것이죠. 그래서 시그모이드 함수'보다는 S자 모양 함수란 이름이 그 성질을 더 적절적으로 내비치지만, 이 책에서는 시그모이드란 이름으로 옮겼습니다. 첫 번째 이유는 실제 학습장에서 많이 쓰이는 이름이고 두 번째는 소스 코드와의 일관성 때문입니다. 소스 코드에서 이 기능을 담당하는 함수 이름이 'sigmoid'이고 첫 번째에 자주 등장합니다. 시그모이드 = S자 모양이란 점만 확실히 기억해주세요.

3.2.5 시그모이드 합수와 계단 합수 비교

시그모이드 합수와 계단 합수를 비교해봅시다. 이 두 합수를 [그림 3-8]에 함께 그려봤습니다. 무엇이 다르고, 또 공통되는 성질이라 할 만한 것은 무엇인가요? 몸금이 생각해보세요.

그림 3-8 계단 합수(점선)와 시그모이드 합수(실선)



[그림 3-8]을 보고 가장 먼저 느껴지는 점은 ‘매끄러움’의 차이일 것입니다. 시그모이드 합수는 부드러운 곡선이며 입력에 따라 출력이 연속적으로 변화합니다. 한편, 계단 합수는 0을 경계로 출력이 갑자기 바뀌어버립니다. 시그모이드 합수의 이 매끈함이 신경망 학습에서 아주 중요한 역할을 하게 됩니다.

(역시 매끈함과 관련되지만) 계단 합수가 0과 1 중 하나의 값만 돌려주는 반면 시그모이드 합수는 실수($0.731, \dots, 0.880, \dots$ 등)를 돌려준다는 점도 다릅니다. 다시 말해 퍼셉트론에서는 뉴런 사이에 0 혹은 1이 흘렀다면, 신경망에서는 연속적인 실수가 흘립니다.

비유하자면, 계단 합수는 ‘시시오도시’*이고 시그모이드 합수는 ‘물레방아’와 비슷하죠. 계단 합수는 시시오도시처럼 물을 쏟아내거나 쏟아내지 않는(0 또는 1) 두 가지 움직임을 보여주며, 시그모이드 합수는 물레방아처럼 흘러온 물의 양에 비례해 흘리는 물의 양을 조절합니다.

* 옮긴이: 대부분 물레방아라고도 부르는, 일본 전통 정원에서 흔히 볼 수 있는 정식입니다. 물이 떨어지는 곳에 헛恸을 지른 다니무 통을 시 소처럼 비스듬히 설치해두면, 통 안에 물이 차운아내고 원래대로 돌아가는 동작을 반복합니다. 이때 바닥을 때리는 소리로 운치를 더합니다.

두 합수의 공통점도 살펴볼까요? 두 합수는 메끄러움이라는 점에서는 다르지만, [그림 3-8]을 큰 관점에서 보면 둘은 같은 모양을 하고 있습니다. 둘 다 입력이 작을 때의 출력은 0에 가깝고 (혹은 0이고), 입력이 커지면 출력이 1에 가까워지는(혹은 1이 되는) 구조인 것이죠. 즉, 계단 합수와 시그모이드 합수는 입력이 중요하면 큰 값을 출력하고 입력이 중요하지 않으면 작은 값을 출력합니다. 그리고 입력이 아무리 작거나 커도 출력은 0에서 1 사이에는 것도 둘의 공통점입니다.

3.2.6 비선형 합수

계단 합수와 시그모이드 합수의 공통점은 그 밖에도 있습니다. 중요한 공통점으로, 둘 모두는 비선형 합수입니다. 시그모이드 합수는 곡선, 계단 합수는 계단처럼 구부러진 직선으로 나타나며, 동시에 비선형 합수로 분류됩니다.

NOTE 활성화 합수를 설명할 때 비선형 합수와 선형 합수라는 용어가 자주 등장합니다. 합수란 어떤 값을 입력하면 그에 따른 값을 돌려주는 변환기입니다. 이 변환기에 무언가 입력했을 때 출력이 입력의 상수배만큼 변하는 합수를 **선형 합수**라고 합니다. 수식으로는 $f(x) = ax + b$ 이고, 이 때 a 와 b 는 상수입니다. 그래서 선형 합수는 곱은 1개의 직선이 됩니다. 한편 **비선형 합수**는 문자 그대로 '선형이 아닌' 합수입니다. 즉, 직선 1개로는 그릴 수 없는 합수를 말합니다.

신경망에서는 활성화 합수로 비선형 합수를 사용해야 합니다. 달리 말하면 선형 합수를 사용해서는 안 됩니다. 왜 선형 합수는 안 되는 걸까요? 그 이유는 바로 선형 합수를 이용하면 신경망의 총을 깊게 하는 의미가 없어지기 때문입니다.

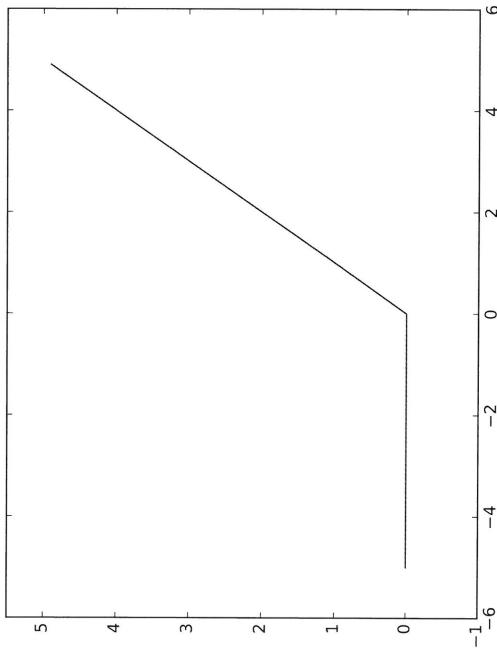
선형 합수의 문제는 총을 아무리 깊게 헤도 '온낙총'이 없는 네트워크'로도 똑같은 기능을 할 수 있다는 데 있습니다. 구체적으로 (약간 직감적으로) 설명해주는 간단한 예를 생각해봅시다. 선형 합수인 $h(x) = cx$ 를 활성화 합수로 사용한 3층 네트워크를 떠올려보세요. 이를 식으로 나타내면 $y(x) = h(h(h(x)))$ 가 됩니다. 이 계산은 $y(x) = c * c * c * x^3$ 처럼 꼽셈을 세 번 수행하지만, 실은 $y(x) = ax$ 와 똑같은 식입니다. $a = c^3$ 이라고만 하면 끝이죠. 즉, 온낙총이 없는 네트워크로 표현할 수 있습니다. 이 예처럼 선형 합수를 이용해서는 여러 층으로 구성하는 이점을 살릴 수 없습니다. 그래서 층을 쌓는 혜택을 얻고 싶다면 활성화 합수로는 반드시 비선형 합수를 사용해야 합니다.

3.2.7 ReLU 험수

지금까지 활성화 험수로서 계단 험수와 시그모이드 험수를 소개했습니다. 시그모이드 험수는 신경망 분야에서 오래전부터 이용해왔으나, 최근에는 ReLU^{Rectified Linear Unit, 레풀 험수}를 주로 이용합니다.

ReLU는 입력이 0을 넘으면 그 입력을 그대로 출력하고, 0 이하이면 0을 출력하는 험수입니다 (그림 3-9).

그림 3-9 ReLU 험수의 그래프*



수식으로는 [식 3.7]처럼 쓸 수 있습니다.

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

[식 3.7]

그래프와 수식에서 보듯 ReLU는 간단한 험수입니다. 그래서 다음과 같이 쉽게 구현해 쓸 수 있습니다.

* 옮긴이_ ReLU에서 Rectified란 정류된이란 뜻입니다. 정류(整流)는 전기회로 쪽 용어로, 예를 들어 반파정류회로(half-wave rectification circuit)는 +/-가 반복되는 교류에서 - 흐름을 차단하는 회로입니다. [그림 3-9]와 비교하면 $x > 0$ 이하일 때를 차단하여 아무 값도 출력하지 않는(0을 출력하는) 것(조. 그래서 ReLU 험수를 정류된 선형 험수' 정도로 옮길 수 있겠으나, 간단한 개념을 어려운 용어로 바꾼다는 느낌이라 그만두었습니다.

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

여기에서는 넘파이의 maximum 함수를 사용했습니다. maximum은 두 입력 중 큰 값을 선택해 반환하는 함수입니다.

이번 장에서는 앞으로 시그모이드 함수를 활성화 함수로 사용합니다만, 이 책 후반부는 주로 ReLU 함수를 사용합니다.

3.3 다차원 배열의 계산

넘파이의 다차원 배열을 사용한 계산법을 숙달하면 신경망을 효율적으로 구현할 수 있습니다. 그래서 이번 절에서는 넘파이의 다차원 배열 계산에 대해서 설명한 뒤 신경망을 구현해보겠습니다.

3.3.1 다차원 배열

다차원 배열도 그 기본은 ‘숫자의 집합’입니다. 숫자가 한 줄로 늘어선 것이나 직사각형으로 늘어놓은 것, 3차원으로 늘어놓은 것이나 (더 일반화한) N차원으로 나열하는 것을 통틀어 다차원 배열이라고 합니다. 그럼 넘파이를 사용해서 다차원 배열을 작성해보겠습니다. 우선은 지금까지 보아온 1차원 배열입니다.

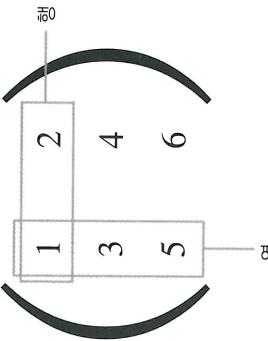
```
>>> import numpy as np  
>>> A = np.array([1, 2, 3, 4])  
>>> print(A)  
[1 2 3 4]  
>>> np.ndim(A)  
1  
>>> A.shape  
(4, )  
>>> A.shape[0]  
4
```

이와 같이 배열의 차원 수는 np.ndim() 함수로 확인할 수 있습니다. 또, 배열의 형상은 인스턴스 변수인 shape으로 알 수 있습니다. 이 예에서 A는 1차원 배열이고 원소 4개로 구성되어 있네요. 한 가지, A.shape이 튜플을 반환하는 것에 주의하세요. 이는 1차원 배열이라도 다차원 배열일 때와 통일된 형태로 결과를 반환하기 위함입니다. 예를 들어 2차원 배열일 때는 (4, 3), 3차원 배열일 때는 (4, 3, 2) 같은 튜플을 반환합니다. 그래서 1차원 배열일 때도 결과를 튜플로 반환하는 것입니다. 자, 이어서 2차원 배열을 작성해보죠.

```
>>> 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)
```

여기에서는 '3×2 배열'인 B를 작성했습니다. 3×2 배열은 처음 차원에는 원소가 3개, 다음 차원에는 원소가 2개 있다는 의미입니다. 이때 처음 차원은 0번째 차원, 다음 차원은 1번쩨 차원에 대응합니다(파이썬의 인덱스는 0부터 시작합니다). 2차원 배열은 특히 행렬(matrix)이라고 부르고 [그림 3-10]과 같이 배열의 가로 방향을 행(row, 세로 방향을 열(column))이라고 합니다.

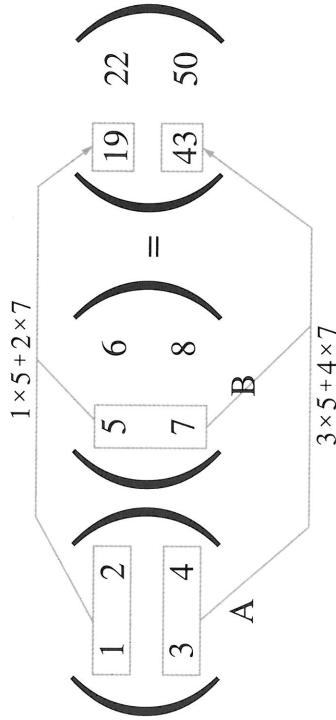
그림 3-10 2차원 배열(행렬)의 행(가로)과 열(세로)



3.3.2 행렬의 곱

이어서 행렬(2차원 배열)의 곱을 구하는 방법을 알아보겠습니다. 예를 들어 2×2 행렬의 곱은 [그림 3-11]처럼 계산합니다.

그림 3-11 행렬의 곱 계산 방법



그림에서처럼 행렬 곱은 왼쪽 행렬의 행(가로)과 오른쪽 행렬의 열(세로)을 원소별로 곱하고 그 값을 더해서 계산합니다. 그리고 그 계산 결과가 새로운 다차원 배열의 원소가 됩니다. 예를 들어 **A**의 1행과 **B**의 1열을 곱한 값은 결과 행렬의 1행 1번째 원소가 되고, **A**의 2행과 **B**의 1열을 곱한 결과는 2행 1번째 원소가 됩니다. 참고로 이 책에서는 '수식에서의 행렬'을 글씨로 표기합니다. 예를 들어 행렬은 **A**처럼 표기하여 원소가 하나인 스칼라값(예컨대 a 와 b)이나 파이썬 코드에서의 변수명과 구별합니다. 이 계산을 파이썬으로 구현하면 다음과 같습니다.

```
>>> 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]])
```

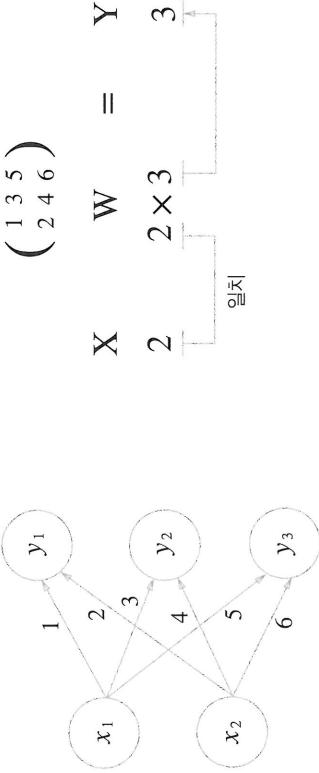
○ 코드에서 A와 B는 2×2 행렬이며,* 이 두 행렬의 곱은 넘파이 함수 np.dot()으로 계산합니다

* 여기에서의 A와 B는 파이썬 코드의 변수명이라 굳게 표기하지 않습니다. 오직 '수식'일 때만 굳게 표기하여 코드 설명과 구분했습니다.

3.3.3 신경망에서의 행렬 곱

그럼 넘파이 행렬을 써서 신경망을 구현해보겠습니다. 이번 예제에서는 [그림 3-14]의 간단한 신경망을 가져보죠. 이 신경망은 편향과 활성화 함수를 생략하고 기중치만 갖습니다.

그림 3-14 행렬의 곱으로 신경망의 계산을 수행합니다.



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

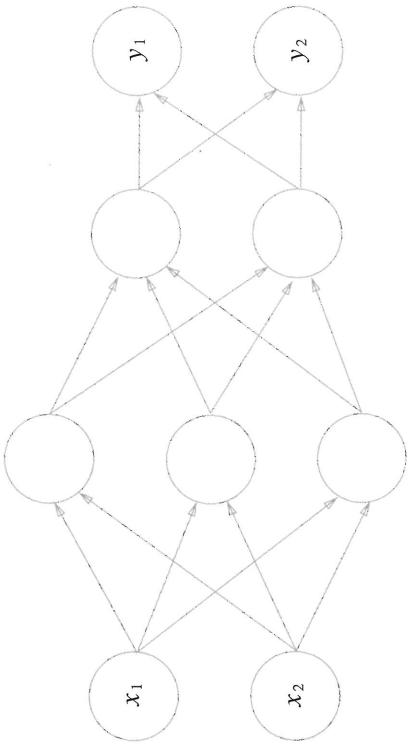
```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5 11 17]
```

다차원 배열의 스칼라곱을 구해주는 `np.dot` 함수를 사용하면 이처럼 단번에 결과 Y 를 계산할 수 있습니다. Y 의 원소가 100개든 1,000개든 한 번의 연산으로 계산할 수 있습니다! 만약 `np.dot`을 사용하지 않으면 Y 의 원소를 하나씩 따져봐야 합니다(또는 `for` 문을 사용해서 계산해야 하는데, 굉장히 귀찮겠지요). 그래서 행렬의 곱으로 한꺼번에 계산해주는 기능은 신경망을 구현할 때 매우 중요하다고 말할 수 있습니다.

3.4 3층 신경망 구현하기

이제 더 그럴싸한 신경망을 구현해보죠. 이번에는 [그림 3-15]의 3층 신경망에서 수행되는, 입력부터 출력까지의 처리(순방향 처리)를 구현하겠습니다. 이를 위해 앞에서 설명한 템파이의 다차원 배열을 사용합니다. 템파이 배열을 잘 쓰면 아주 적은 코드만으로 신경망의 순방향 처리를 완성할 수 있습니다.

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.



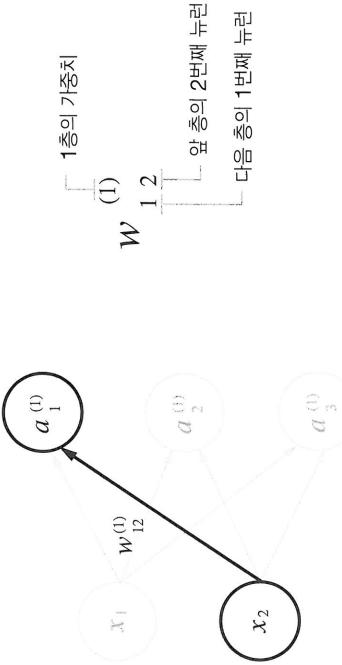
3.4.1 표기법 설명

이번 절에서는 신경망에서의 처리를 설명하며 $w_{12}^{(1)}$ 과 $a_1^{(1)}$ 같은 표기법을 선보입니다. 조금 복잡해 보일 수 있는데, 이번 절에서만 사용하는 표기이나 가볍게 건너뛰고 싶어도 문제는 없습니다.

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

그럼 하나씩 정의해봅시다. [그림 3-16]을 보세요. [그림 3-16]은 입력층의 뉴런 x_2 에서 다음 층의 뉴런 $a_1^{(1)}$ 으로 향하는 선 위에 가중치를 표시하고 있습니다.

그림 3-16 중요한 표 |



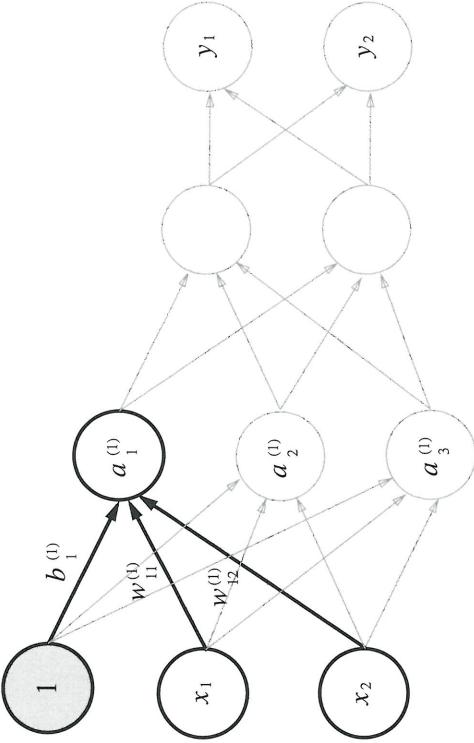
[그림 3-16]과 같이 기중치와 은닉층 뉴런의 오른쪽 위에는 ' (1) '이 붙어 있습니다. ' (1) '은 1층의 기중치, 1층의 뉴런임을 뜻하는 번호입니다. 또, 기중치의 오른쪽 아래의 두 숫자는 차례로 다음 층 뉴런과 앞 층 뉴런의 인덱스 번호입니다. 가령 $w_{12}^{(1)}$ 은 앞 층의 2번째 뉴런(x_2)에서 다음 층의 1번째 뉴런($a_1^{(1)}$)으로 향할 때의 기중치라는 뜻입니다. 기중치 오른쪽 아래의 인덱스 번호는 '다음 층 번호, 앞 층 번호' 순으로 적습니다.*

3.4.2 각 층의 신호 전달 구현하기

이번 절에서는 입력층에서 '1층의 첫 번째 뉴런'으로 가는 신호를 살펴보겠습니다. [그림 3-17]과 같은 상황이죠.

* 옮김10_ 이 순서를 반대로 표기하는 경우도 많으니 다른 자료를 볼 때는 순서를 한 번 확인해보세요.

그림 3-17 입력층에서 1층으로 신호 전달



[그림 3-17]과 같이 편향을 끼치는 뉴런인 ①이 추가되었습니다. 편향은 오른쪽 아래 인덱스가 하나밖에 없다는 것에 주의하세요. 이는 앞 층의 편향 뉴런(뉴런 ①)이 하나뿐이기 때문입니다.

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

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

여기에서 행렬의 곱을 이용하면 1층의 ‘가중치 부분’을 다음 식처럼 간소화할 수 있습니다.

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

이때 행렬 $\mathbf{A}^{(1)}$, \mathbf{X} , $\mathbf{B}^{(1)}$, $\mathbf{W}^{(1)}$ 은 각각 다음과 같습니다.

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

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

그럼 범파이의 다차원 배열을 사용해서 [식 3.9]를 구현합시다(입력 신호, 가중치, 편향은 적당한 값으로 설정하겠습니다).

```

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,)

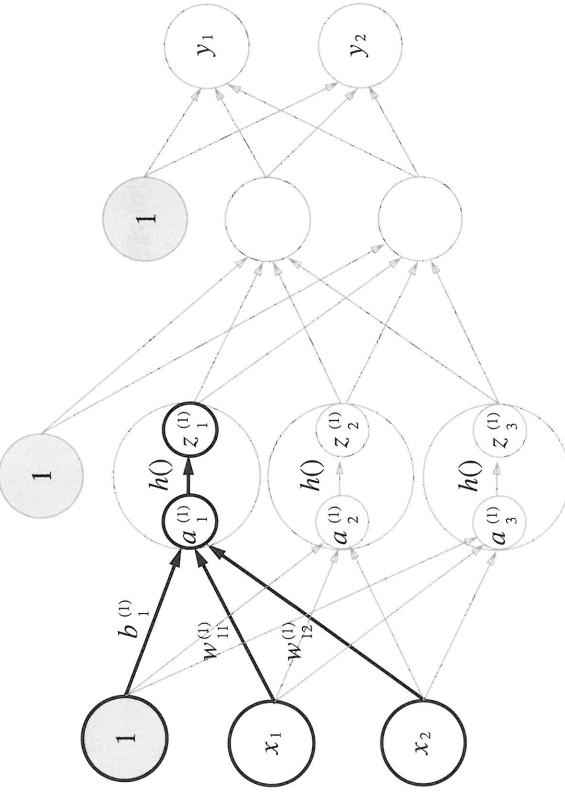
A1 = np.dot(X, W1) + B1

```

이 계산은 앞 절에서 한 계산과 같습니다. $W1$ 은 2×3 행렬, X 는 원소가 2개인 1차원 배열입니다. 여기에서도 역시 $W1$ 과 X 의 대응하는 차원의 원소 수가 일치하고 있군요.

이어서 1층의 활성화 함수에서의 처리를 살펴보겠습니다. 이 활성화 함수의 처리를 그림으로 나타내면 [그림 3-18]처럼 됩니다.

그림 3-18 입력층에서 1층으로의 신호 전달



[그림 3-18]과 같이 은닉층에서의 가중치 합(가중 신호와 편향의 총합)을 a 로 표기하고 활성화 함수 $h()$ 로 변환된 신호를 z 로 표기합니다. 여기에서는 활성화 함수로 시그모이드 함수를 사용하기로 합니다. 이를 파이썬으로 구현하면 다음과 같습니다.

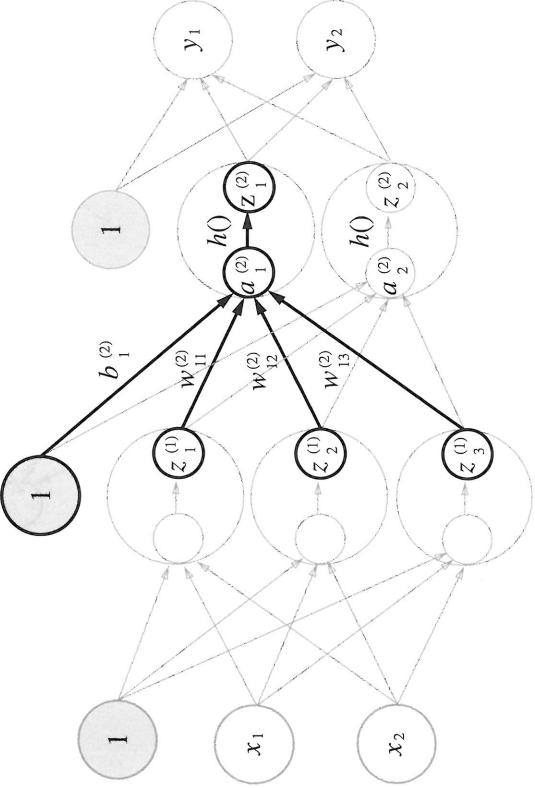
```
Z1 = sigmoid(A1)
```

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

o sigmoid() 함수는 앞에서 정의한 함수입니다. 이 함수는 넘파이 배열을 받아 같은 수의 원소로 구성된 넘파이 배열을 반환합니다.

이어서 1층에서 2층으로 가는 과정(그림 3-19)과 그 구현을 살펴보죠.

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])
```

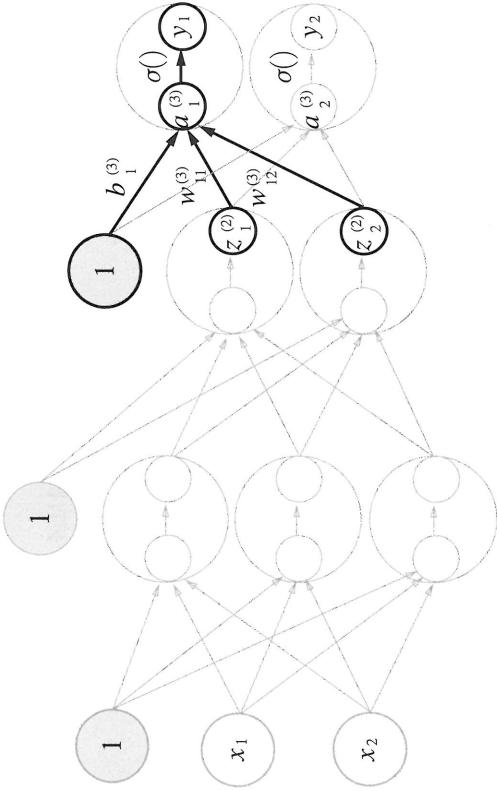
```
print(Z1.shape) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)
```

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

이 구현은 1층의 출력 Z_1 이 2층의 입력이 된다는 점을 제외하면 조금 전의 구현과 똑같습니다. 이처럼 넘파이 배열을 사용하면서 층 사이의 신호 전달을 쉽게 구현할 수 있습니다.

마지막으로 2층에서 출력층으로의 신호 전달입니다(그림 3-20). 출력층의 구현도 그동안의 구현과 거의 같습니다. 딱 하나, 활성화 함수만 지금까지의 은닉층과 다릅니다.

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) # 혹은 Y = A3
```

여기에서는 항등 함수인 `identity_function()`을 정의하고, 이를 출력층의 활성화 함수로 이용했습니다. 항등 함수는 입력을 그대로 출력하는 함수입니다. 그래서 이 예에서는 `identity_function()`을 굳이 정의할 필요는 없지만, 그동안의 흐름과 통일하기 위해 이렇게 구현했습니다. 또한 [그림 3-20]에서는 출력층의 활성화 함수를 $\sigma()$ 로 표시하여 은닉층의 활성화 함수 $h()$ 와는 다른점을 명시했습니다(σ 는 '시그마'라고 읽습니다).

NOTE 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 맞게 정합니다. 예를 들어 회귀에는 항등 함수를 2클래스 분류에는 시그모이드 함수를, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적입니다. 출력층의 활성화 함수에 대해서는 다음에 다시 자세히 설명합니다.

3.4.3 구현 정리

이로써 3층 신경망에 대한 설명은 끝입니다. 그럼 지금까지의 구현을 정리해보도록 하죠. 신경망 구현의 관례에 따라 가중치만 W1과 같이 대문자로 쓰고, 그 외 편향과 중간 결과 등은 모두 소문자로 썼습니다.

```
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) # [ 0.31682708  0.69627909]
```

여기에서는 `init_network()`와 `forward()`라는 험수를 정의했습니다. `init_network()` 험수는 가중치와 편향을 초기화하고 이를을 딕셔너리 변수인 `network`에 저장합니다. 이 딕셔너리 변수 `network`에는 각 층에 필요한 매개변수(가중치와 편향)를 저장합니다. 그리고 `forward()` 험수는 입력 신호를 출력으로 변환하는 처리 과정을 모두 구현하고 있습니다.

험수 이름을 `forward`라 한 것은 신호가 순방향(입력에서 출력 방향)으로 전달됨(순전파)을 알리기 위함입니다. 앞으로 신경망 학습을 다를 때 역방향(backward, 출력에서 입력 방향) 처리에 대해서도 살펴볼 예정입니다.

이로써 신경망의 순방향 구현은 끝입니다. 보신 것처럼 넘파이의 다차원 배열을 잘 사용하면 신경망을 효율적으로 구현할 수 있습니다!

3.5 출력층 설계하기

신경망은 분류와 회귀 모두에 이용할 수 있습니다. 다만 둘 중 어떤 문제나에 따라 출력층에서 사용하는 활성화 험수가 달라집니다. 일반적으로 회귀는 일등 험수를, 분류에는 소프트맥스 험수를 사용합니다.

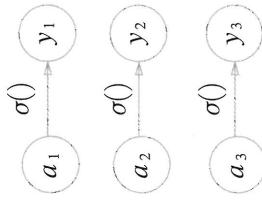
NOTE 기계학습 문제는 **분류(classification)**와 **회귀(regression)**로 나릅니다. 분류는 데이터가 어느 클래스에 속 하느냐는 문제입니다. 사진 속 인물의 성별을 분류하는 문제기 예기에 속합니다. 한편, 회귀는 입력 데이터에서 (연속적인) 수치를 예측하는 문제입니다. 사진 속 인물의 몸무게(57.4kg?)를 예측하는 문제가 회구입니다.*

* 옮긴이. 분류와 달리 회구라는 이름은 전문적이지 않죠. 그 이유를 알려면 이름의 기원을 찾아보아 합니다. 19세기 후반 영국의 우생학자 프란시스 퀸턴 경은 사람과 완두콩 등을 대상으로 그 키(kgi)를 측정했습니다. 관찰 결과 키가 큰 부모의 자식은 부모보다 적고 적은 부모의 자식은 부모보다 큰 쪽 평균으로 회구(regression)하는 경향이 있음을 알았습니다. 그 사이에는 선형 관계가 있어 부모의 키로부터 자식의 키를 예측할 수 있고, 그 예측 결과가 연속적인 수치인 것이죠.

3.5.1 항등 함수와 소프트맥스 함수 구현하기

항등 함수 identity function는 입력을 그대로 출력합니다. 입력과 출력이 항상 같다는 뜻의 항등입니다. 그래서 출력층에서 항등 합수를 사용하면 입력 신호가 그대로 출력 신호가 됩니다. 항등 합수의 처리는 신경망 그림으로는 [그림 3-21]처럼 되겠죠. 항등 합수에 의한 변환은 은닉층에서의 활성화 합수와 마찬가지로 확실표로 그립니다.

그림 3-21 항등 합수



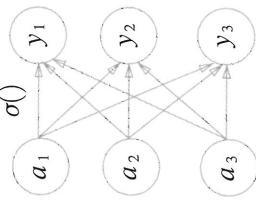
한편, 분류에서 사용하는 소프트맥스 합수 softmax function의 식은 다음과 같습니다.

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

$\exp(x)$ 는 e^x 을 뜻하는 지수 합수 exponential function입니다 (e 는 자연상수). n 은 출력층의 뉴런 수, y_k 는 그중 k 번째 출력임을 뜻합니다. [식 3.10]과 같이 소프트맥스 합수의 분자는 입력 신호 a_k 의 지수 합수, 분모는 모든 입력 신호의 지수 합수의 합으로 구성됩니다.

이 소프트맥스 합수를 그림으로 나타내면 [그림 3-22]처럼 됩니다. 그림과 같이 소프트맥스의 출력은 모든 입력 신호로부터 확실표를 받습니다. [식 3.10]의 분모에서 보듯, 출력층의 각 뉴런이 모든 입력 신호에서 영향을 받기 때문입니다.

그림 3-22 소프트맥스 험수



그럼, 이상의 소프트맥스 험수를 구현해봅시다. 여기에서는 파이썬 인터프리터를 사용하여 결과를 하나씩 확인하기며 진행하겠습니다.

```
>>> a = np.array([0.3, 2.9, 4.0])
>>>
>>> exp_a = np.exp(a) # 지수 험수
>>> print(exp_a)
[ 1.34985881 18.17414537 54.59815003]
>>>
>>> sum_exp_a = np.sum(exp_a) # 지수 험수의 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 험수를 그대로 파이썬으로 표현한 것이라 막히 설명할 것은 없네요. 이제 이 논리 흐름을 파이썬 험수로 정의하여, 앞으로 필요할 때 사용할 수 있도록 해놓겠습니다.

```
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 [식 3.10]을 제대로 표현하고 있지만, 컴퓨터로 계산할 때는 결함이 있습니다. 바로 오버플로 문제입니다. 소프트맥스 합수는 지수 합수를 사용하는데, 지수 합수란 것이 쉽게 아주 큰 값을 내뱉죠. 가령 e^{10} 은 20,000이 넘고, e^{100} 은 0이 40개 넘는 큰 값이 되고, e^{1000} 은 무한대를 뜻하는 inf가 되어 돌아옵니다. 그리고 이런 큰 값끼리 나눗셈을 하면 결과 수치가 '불안정'해집니다.

WARNING__ 컴퓨터는 수number를 4비트나 8비트와 같이 크기가 유한한 데이터로 다릅니다. 다시 말해 표현할 수 있는 수의 범위가 한정되어 너무 큰 값은 표현할 수 없다는 문제가 발생합니다. 이것을 오버플로 overflow라 하며, 컴퓨터로 수치를 계산할 때 주의할 점입니다.

이 문제를 해결하도록 소프트맥스 합수 구현을 개선해봅시다. 다음은 개선한 수식입니다.

$$\begin{aligned} y_k &= \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} = \frac{C \exp(a_k)}{C \sum_{i=1}^n \exp(a_i)} \\ &= \frac{\exp(a_k + \log C)}{\sum_{i=1}^n \exp(a_i + \log C)} \\ &= \frac{\exp(a_k + C')}{\sum_{i=1}^n \exp(a_i + C')} \end{aligned} \quad [\text{식 3.11}]$$

[식 3.11]의 첫개 과정을 살펴보죠. 첫 번째 변형에서는 C라는 임의의 정수를 분자와 분모 양쪽에 곱했습니다(양쪽에 같은 수를 곱했으니 결국 똑같은 계산입니다). 그다음으로 C를 지수 합수 exp() 안으로 옮겨 logC로 만듭니다. 마지막으로 logC를 C'라는 새로운 기호로 바꿉니다.

[식 3.11]이 말하는 것은 소프트맥스의 지수 합수를 계산할 때 어떤 정수를 더해도 (혹은 빼도) 결과는 바뀌지 않는다는 것입니다. 여기서 C'에 어떤 값을 대입해도 상관없지만, 오버플로 를 막을 목적으로는 입력 신호 중 최댓값을 이용하는 것이 일반적입니다. 구체적인 예를 하나 보시죠.

```

>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # 소프트맥스 합수의 계산
array([ nan,    nan,    nan])      # 제대로 계산되지 않는다.
>>>
>>> c = np.max(a)
>>> 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 nan이 출력됩니다 (nan은 not a number의 약자입니다). 하지만 입력 신호 중 최댓값(이 예에서는 c)을 빼주면 올바르게 계산할 수 있습니다. 이를 바탕으로 소프트맥스 합수를 다시 구현하면 다음과 같습니다.

```

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

    return y

```

3.5.3 소프트맥스 합수의 특징

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

```

>>> 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 합수의 출력은 0에서 1.0 사이의 실수입니다. 또, 소프트맥스 합수 출력의 총합은 1입니다. 출력 충합이 1이 된다는 점은 소프트맥스 합수의 중요한 성질입니다.

이 성질 덕분에 소프트맥스 험수의 출력을 ‘확률’로 해석할 수 있습니다.

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 해석할 수 있습니다. 그리고 이 결과 확률들로부터 “2번째 원소의 확률이 가장 높으니, 답은 2번째 클래스다”라고 할 수 있습니다. 혹은 “74%의 확률로 2번째 클래스, 25%의 확률로 1번째 클래스, 1%의 확률로 0번째 클래스다”와 같이 확률적인 결론도 낼 수 있죠. 즉, 소프트맥스 험수를 이용함으로써 문제를 확률적(통계적)으로 대응할 수 있게 되는 것 이죠.

여기서 주의점으로, 소프트맥스 험수를 적용해도 각 원소의 대소 관계는 변하지 않습니다. 이는 지수 험수 $y = \exp(x)$ 가 단조 증가 함수이기 때문이죠.* 실제로 앞의 예에서는 a의 원소들 사이의 대소 관계가 y의 원소들 사이의 대소 관계로 그대로 이어집니다. 예를 들어 a에서 가장 큰 원소는 2번째 원소이고, y에서 가장 큰 원소도 2번째 원소입니다.

신경망을 이용한 분류에서는 일반적으로 가장 큰 출력을 내는 뉴런에 해당하는 클래스로만 인식합니다. 그리고 소프트맥스 험수를 적용해도 출력이 가장 큰 뉴런의 위치는 달라지지 않습니다. 결과적으로 신경망으로 분류할 때는 출력층의 소프트맥스 험수를 생략해도 됩니다. 현실에서도 지수 험수 계산에 드는 자원 낭비를 줄이고자 출력층의 소프트맥스 험수는 생략하는 것이 일반적입니다.

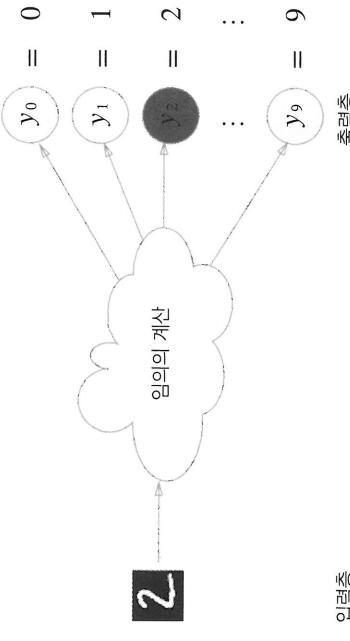
NOTE 기계학습의 문제 풀이는 **학습과 추론**의 두 단계를 가진 이류입니다. 학습 단계에서 모델을 학습하고(직업 훈련을 받고), 추론 단계에서 앞서 학습한 모델로 미지의 데이터에 대해서 추론(분류)을 수행합니다(현장에 나가 진짜 일을 합니다!). 방금 설명한 대로, 추론 단계에서는 출력층의 소프트맥스 험수를 생략하는 것이 일반적입니다. 한편 신경망을 학습시킬 때는 출력층에서 소프트맥스 험수를 사용합니다(4장 참고).

3.5.4 출력층의 뉴런 수 정하기

출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정해야 합니다. 분류에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적입니다. 예를 들어 입력 이미지를 숫자 0부터 9 중 하나로 분류하는 문제라면 [그림 3-23]처럼 출력층의 뉴런을 10개로 설정합니다.

* 옮긴이] 단조 증가 험수란 정의역 원소 a, b 가 $a \leq b$ 일 때, $f(a) \leq f(b)$ 가 성립하는 험수입니다.

그림 3-23 출력층의 뉴런은 각 숫자에 대응한다.



[그림 3-23]의 예에서 출력층 뉴런은 위에서부터 차례로 숫자 0, 1, ..., 9에 대응하며, 뉴런의 회색 높도가 해당 뉴런의 출력 값의 크기를 의미합니다. 이 예에서는 색이 가장 짙은 y_2 뉴런이 가장 큰 값을 출력하는 것이죠. 그래서 이 신경망이 선택한 클래스는 y_2 , 즉 입력 이미지를 숫자 '2'로 판단했음을 의미합니다.

3.6 손글씨 숫자 인식

신경망의 구조를 배웠으니 실전 예에 적용해보죠. 바로 손글씨 숫자 분류입니다. 이번 절에서 는 이미 학습된 매개변수를 사용하여 학습 과정은 생략하고, 추론 과정만 구현할 겁니다. 이 추론 과정을 신경망의 순전파 forward propagation라고도 합니다.

NOTE 기계학습과 미친다!로 신경망도 두 단계를 거친 문제를 해결합니다. 먼저 훈련 데이터(학습 데이터)를 사용해 가중치 매개변수를 학습하고, 추론 단계에서는 앞서 학습한 매개변수를 사용하여 입력 데이터를 분류합니다.

3.6.1 MNIST 데이터셋

이번 예에서 사용하는 데이터셋은 MNIST라는 손글씨 숫자 이미지 집합입니다. MNIST는 기계학습 분야에서 아주 유명한 데이터셋으로, 간단한 실험부터 논문으로 발표되는 연구까지 다

양한 곳에서 이용하고 있습니다. 이미지 인식이나 기계학습 논문들을 읽다 보면 실험용 데이터로 자주 등장하는 것을 확인할 수 있을 겁니다.

MNIST 데이터셋은 0부터 9까지의 숫자 이미지로 구성됩니다(그림 3-24). 훈련 이미지가 60,000장, 시험 이미지가 10,000장 준비되어 있습니다. 일반적으로 이 훈련 이미지들을 사용하여 모델을 학습하고, 학습한 모델로 시험 이미지를 얼마나 정확하게 분류하는지를 평가합니다.

그림 3-24 MNIST 이미지 데이터셋의 예

7	2	1	0	4	1	4	0	5	9
0	6	9	0	1	5	9	7	8	4

MNIST의 이미지 데이터는 28×28 크기의 회색조 이미지(1채널)이며, 각 픽셀은 0에서 255 까지의 값을 취합니다. 각 이미지에는 또한 '7', '2', '1과 같이 그 이미지가 실제로 의미하는 숫자가 레이블로 붙어 있습니다.

이 책에서는 MNIST 데이터셋을 내려받아 이미지를 넘파이 배열로 변환해주는 파이썬 스크립트를 제공합니다(깃허브 저장소의 dataset/mnist.py 파일). mnist.py를 임포트해 사용하면서 작업 디렉터리를 ch01, ch02, ch03, ..., ch08 중 하나로 옮겨주세요(이유는 다음 페이지에서 설명합니다). mnist.py 파일에 정의된 load_mnist() 함수를 이용하면 MNIST 데이터를 다음과 같이 아주 쉽게 가져올 수 있습니다.

```
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 임포트합니다. 그런 다음 load_mnist 험수로 MNIST 데이터셋을 읽습니다. load_mnist가 MNIST 데이터를 받아와야 하니 최초 실행 시에는 인터넷에 연결된 상태여야 합니다. 두 번째부터는 로컬에 저장된 파일(pickle 파일)을 읽기 때문에 순식간에 끝납니다.

WARNING_ mnist.py 파일은 이 책 예제 소스의 dataset 디렉터리에 있고, 이 파일을 이용하는 다른 예제들은 각각 ch01, ch02, ch03, ..., ch08 디렉터리에서만 수행한다고 가정합니다. 즉, 각 예제에서 mnist.py 파일을 찾으려면 부모 디렉터리로부터 시작해야 해서 sys.path.append(os.getcwd()) 문장을 추가한 것입니다.

load_mnist 험수는 읽은 MNIST 데이터를 “**(훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)**” 형식으로 반환합니다. 인수로는 normalize, flatten, one_hot_label 세 가지를 설정할 수 있습니다. 세 인수 모두 bool 값입니다. 첫 번째 인수인 normalize는 입력 이미지의 픽셀 값을 0.0~1.0 사이의 값으로 정규화할지를 정합니다. False로 설정하면 입력 이미지의 픽셀은 원래 값 그대로 0~255 사이의 값을 유지합니다. 두 번째 인수인 flatten은 입력 이미지를 평탄하게, 즉 1차원 배열로 만들지를 정합니다. False로 설정하면 입력 이미지를 $1 \times 28 \times 28$ 의 3차원 배열로, True로 설정하면 784개의 원소로 이루어진 1차원 배열로 저장합니다. 세 번째 인수인 one_hot_label은 레이블을 원-핫 인코딩(one-hot encoding) 형태로 저장할지를 정합니다. 원-핫 인코딩이란, 예를 들어 [0,0,1,0,0,0,0,0,0]처럼 정답을 뜻하는 원소만 1이고(hot하고) 나머지는 모두 0인 배열입니다. one_hot_label이 False면 7이나 2와 같이 숫자 형태의 레이블을 저장하고, True일 때는 레이블을 원-핫 인코딩하여 저장합니다.

NOTE_ 파일에는 pickle이라는 편리한 기능이 있습니다. 이는 프로그램 실행 중에 특정 객체를 파일로 저장하는 기능입니다. 저장해둔 pickle 파일을 로드하면 실행 당시의 객체를 즉시 복원할 수 있습니다. MNIST 데이터셋을 읽는 load_mnist() 험수에서도 (2번째 이후의 읽기 시) pickle을 이용합니다. pickle 덕분에 MNIST 데이터를 순식간에 준비할 수 있습니다.

그럼 데이터도 확인할겸 MNIST 이미지를 화면으로 불러보도록 하겠습니다. 이미지 표시에는 PIL Python Image Library 모듈을 사용합니다. 다음 코드(ch03/mnist_show.py)를 실행하면 첫 번째 훈련 이미지가 모니터 화면에 표시됩니다(그림 3-25).

```

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=True로 설정해 놓어 둘인 이미지는 1차원 넘파이 배열로 저장되어 있다는 것입니다. 그래서 이미지를 표시할 때는 원래 형상인 28×28 크기로 다시 변형해야 합니다. reshape() 메서드에 원하는 형상을 인수로 지정하면 넘파이 배열의 형상을 바꿀 수 있습니다. 또한, 넘파이로 저장된 이미지 데이터를 PIL용 데잍 객체로 변환해야 하며, 이 변환은 Image.fromarray()가 수행합니다.

3.6.2 신경망의 추론 처리

드디어 이 MNIST 테이터셋을 가지고 추론을 수행하는 신경망을 구현할 차례입니다. 이 신경망은 입력층 뉴런을 784개, 출력층 뉴런을 10개로 구성합니다. 입력층 뉴런이 784개인 이유는 이미지 크기가 $28 \times 28 = 784$ 이기 때문이고, 출력층 뉴런이 10개인 이유는 이 문제가 0에서 9까지의 숫자를 구분하는 문제이기 때문입니다. 한편, 은닉층은 총 두 개로, 첫 번째 은닉층에는 50개의 뉴런을, 두 번째 은닉층에는 100개의 뉴런을 배치할 것입니다. 여기서 50과 100은 임의로 정한 값입니다.

이제 순서대로 작업을 처리해줄 세 험수인 get_data(), init_network(), predict()를 정의하겠습니다(ch03/neuralnet_mnist.py).

```
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.pkl에 저장된 ‘학습된 가중치 매개변수’를 얻습니다. 이 파일에는 가중치와 편향 매개변수가 닥셔너리 변수로 저장되어 있습니다.

나머지 두 험수는 지금까지 보아온 구현과 거의 같으니 설명은 생략하겠습니다. 그럼 이 세 험수를 사용해 신경망에 의한 추론을 수행해보고, 정확도(accuracy(분류가 얼마나 올바른가))도 평가해봅시다.

```
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 생성합니다. 이어서 for 문을 돌며 x에 저장된 이미지 데이터를 1장씩 꺼내 predict() 함수로 분류합니다. predict() 함수는 각 레이어의 확률을 범위인 배열로 반환합니다. 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 해석합니다. 그런 다음 np.argmax() 함수로 이 배열에서 값이 가장 큰(확률이 가장 높은) 원소의 인덱스를 구합니다. 이것이 바로 예측 결과죠. 마지막으로, 신경망이 예측한 답변과 정답 레이블을 비교하여 맞힌 숫자(accuracy_cnt)를 세고, 이를 전체 이미지 숫자로 나눠 정확도를 구합니다.

이 코드를 실행하면 “Accuracy:0.9352”라고 출력합니다. 올바르게 분류한 비율이 93.52%라는 뜻이죠. 이번 장의 목표는 학습된 신경망을 들려보는 것까지라 정확도에 대해서는 고민하지 않겠지만, 한 가지만 미리 말씀드립니다. 다음 장부터는 신경망 구조와 학습 방법을 궁리하여 이 정확도를 더 높여갈 것입니다. 마지막에는 99% 이상까지 도달할 예정입니다!

또한, 이 예에서는 load_mnist 함수의 인수인 normalize를 True로 설정했습니다. normalize를 True로 설정하면 0~255 범위인 각 픽셀의 값을 0.0~1.0 범위로 변환합니다(단순히 픽셀의 값을 255로 나눕니다). 이처럼 데이터를 특정 범위로 변환하는 처리를 정규화(normalization)라고, 신경망의 입력 데이터에 특정 변환을 가하는 것을 전처리(pre-processing)합니다. 여기에서는 입력 이미지 데이터에 대한 전처리 작업으로 정규화를 수행한 셈입니다.

NOTE 현업에서도 신경망(딥러닝)에 전처리를 활용해 사용합니다. 전처리를 통해 식별 능력을 개선하고 학습 속도를 높이는 등의 시례가 많이 제시되고 있습니다. 앞의 예에서는 각 픽셀의 값을 255로 나누는 단순한 정규화를 수행했지만, 현업에서는 데이터 전체의 분포를 고려해 전처리하는 경우가 많습니다. 예를 들어 데이터 전체 평균과 표준편차를 이용하여 데이터들이 0을 중심으로 분포하도록 이동하거나 테이터의 확산 범위를 제한하는 정규화를 수행합니다. 그 외에도 전체 데이터를 균일하게 분포시킬 때 데이터 **백색화**(whitening) 등도 있습니다.

3.6.3 배치 처리

구현 정도를 더 나가기 전에, 이번 절에서는 입력 데이터와 가중치 매개변수의 ‘형상’에 주의해서 조금 전의 구현을 다시 살펴보겠습니다.

우선 파이썬 인터프리터에서 앞서 구현한 신경망 각층의 가중치 형상을 출력해보죠.

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']
>>>
>>> x.shape
(100000, 784)
>>> x[0].shape
(784, )
>>> W1.shape
(784, 50)
>>> W2.shape
(50, 100)
>>> W3.shape
(100, 10)
```

이 결과에서 다차원 배열의 대응하는 차원의 원소 수가 일치합을 확인할 수 있습니다(편향은 생략했습니다). 그럼으로는 [그림 3-26]처럼 됩니다. 확실히, 다차원 배열의 대응하는 차원의 원소 수가 일치하고 있군요. 그리고 최종 결과로는 원소가 10개인 1차원 배열 y가 출력되는 점도 확인합시다.

그림 3-26 신경망 각 층의 배열 형상의 추이



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

그렇다면 이미지 여러장을 한꺼번에 입력하는 경우를 생각해봅시다. 가령 이미지 100개를 뮤어 predict() 함수에 한 번에 넘기는 것이죠. x의 형상을 100×784 로 바꿔서 100장 분량의 데이터를 하나의 입력 데이터로 표현하면 됩니다. 그림으로는 [그림 3-27]처럼 됩니다.

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 됩니다. 이는 100장 분량 입력 데이터의 결과가 한 번에 출력됨을 나타냅니다. 가령 x[0]와 y[0]에는 0번째 이미지와 그 추론 결과가, x[1]과 y[1]에는 1번째 이미지와 그 결과가 저장되는 식입니다.

이처럼 하나로 묶은 입력 데이터를 배치[batch]라 합니다. 배치가 곧 묶음이란 의미죠. 이미지가 저께처럼 다발로 묶여 있다고 생각하면 됩니다.

NOTE 배치 처리는 컴퓨터로 계산할 때 큰 이점을 줍니다. 이미지 1장당 처리 시간을大幅 줄여주는 것이죠. 크게 두 가지 이유가 있는데, 하나는 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화되어 있기 때문입니다. 그리고 카드란 신경망에서는 데이터 전송이 병목으로 작용하는 경우가 자주 있는데, 배치 처리를 함으로써 버스에 주는 부하를 줄인다는 것이 두 번째 이유입니다(정확히는 느린 I/O를 통해 데이터를 읽는 횟수가 줄어, 빠른 CPU나 GPU로 순수 계산을 수행하는 비율이 높아집니다). 즉, 배치 처리를 수행함으로써 큰 배열로 이루어진 계산을 하게 되는데, 컴퓨터에서는 큰 배열을 한꺼번에 계산하는 것보다 분할된 작은 배열을 여러 번 계산하는 것보다 빠릅니다.

이제 배치 처리를 구현해보죠. 앞의 구현에서 달라진 부분을 굵게 강조했습니다.

```
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)))
```

굵은 부분을 하나씩 풀어봅시다. 우선 range() 함수입니다. range() 함수는 range(start, end)처럼 인수를 2개 지정해 호출하면 start에서 end-1까지의 정수로 이루어진 리스트를 반환합니다. 또 range(start, end, step)처럼 인수를 3개 지정하면 start에서 end-1까지 step 간격으로 증가하는 리스트를 반환합니다. 뭔가 복잡하지만 다음의 예를 보면 바로 이해될 겁니다.

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

이 range() 함수가 반환하는 리스트를 바탕으로 x[i:i+batch_size]에서 입력 데이터를 뮤슈 니다. x[i:i+batch_size]은 입력 데이터의 i번째부터 i+batch_size번째까지의 데이터를 뮤는다는 의미죠. 이 예에서는 batch_size가 100이므로 x[0:100], x[100:200], ...와 같이 앞에 서부터 100장씩 뮤어 꺼내게 됩니다.

그리고 앞에서도 나온 argmax()는 최댓값의 인덱스를 가져옵니다. 다만 여기에서는 axis=1이라는 인수를 추가한 것에 주의합시다. 이는 100×10 의 배열 중 1번째 차원을 구성하는 각 원소에서 (1번째 차원을 축으로) 최댓값의 인덱스를 찾도록 한 것입니다(인덱스가 0부터 시작하니 0번째 차원이 가장 처음 차원입니다). 이 역시 예를 보면 쉽게 이해될 겁니다.

```
>>> 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]])  
>>> y = np.argmax(x, axis=1)  
>>> print(y)  
[1 2 1 0]
```

마지막으로 배치 단위로 분류한 결과를 실제 답과 비교합니다. 이를 위해 == 연산자를 사용해 넘파이 배열끼리 비교하여 True/False로 구성된 bool 배열을 만들고, 이 결과 배열에서 True가 몇 개인지 셉니다. 이 처리 과정은 다음 예제에서 확인해보죠.

```
>>> y = np.array([1, 2, 1, 0])  
>>> t = np.array([1, 2, 0, 0])  
>>> print(y==t)  
[True True False True]  
>>> np.sum(y==t)  
3
```

이상으로 배치 처리 구현에 대한 설명을 마칩니다. 데이터를 베치로 처리함으로써 효율적이고 빠르게 처리할 수 있었습니다. 다음 장에서 진행할 신경망 학습에서도 이미지 데이터를 적절히 뮤어서 학습하는데, 그때도 이번 장에서 구현한 배치 처리와 같은 방식으로 구현하게 됩니다.

3.7 정리

이번 장에서는 신경망의 순전파를 살펴봤습니다. 이번 장에서 설명한 신경망은 각 층의 뉴런들이 다음 층의 뉴런으로 신호를 전달한다는 점에서 앞 장의 퍼셉트론과 같습니다. 하지만 다음 뉴런으로 갈 때 신호를 변화시키는 활성화 함수에 큰 차이가 있습니다. 신경망에서는 매우 어렵게 변화하는 시그모이드 함수를, 퍼셉트론에서는 잡자기 변화하는 계단 함수를 활성화 함수로 사용했습니다. 이 차이가 신경망 학습에 중요하죠. 이에 대해서는 다음 장에서 설명하겠습니다.

이번 장에서 배운 내용

- 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU 함수 같은 매끄럽게 변화하는 함수를 이용한다.
- 넘파이의 다자원 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 무은 것을 배치라 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 빨리 얻을 수 있다.