

# 머신 러닝

2023 Fall

강미선

- 생물 신경망에서 인공 신경망 태동

- 수영 선수의 지식이 팔에 저장되는지 뇌에 저장되는지가 논란거리인 시대가 있었으나, 1900년대 뇌 과학과 신경 과학이 비약적으로 발전되면서 뇌에 저장된다는 사실 밝혀짐
- 1900년대 중반에 뇌의 정보처리 과정을 수학적으로 모델링하려는 연구 그룹 등장
- 인공 신경망 ANN(artificial neural network)의 태동 또는 인공지능의 태동

- 컴퓨터의 등장으로 실제 구현 시도

- 퍼셉트론은 가장 성공한 인공 신경망 모델
- 퍼셉트론은 발전을 거듭해 현재 딥러닝으로 이어짐
- 이 장은 퍼셉트론과 다층 퍼셉트론을 다룸(이들은 얇은 신경망).

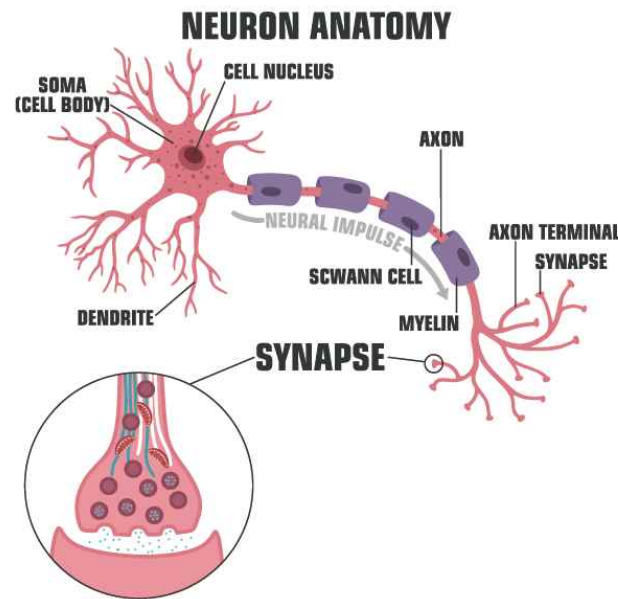
## 4.1 인공 신경망의 태동

- 생물 신경망을 간략히 소개한 후, 인공 신경망 설명
- 인공 신경망은 생물 신경망에서 영감을 얻었지만 실제 구현은  
다름
  - 컴퓨터의 작동 원리가 생물의 작동 원리와 근본적으로 다르기 때문

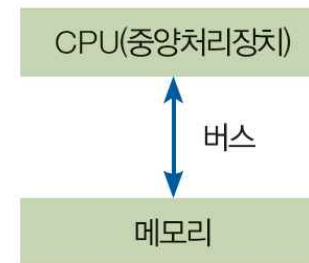
## 4.1.1 생물 신경망

### • 사람 뇌와 컴퓨터

- 뉴런<sub>neuron</sub>은 뇌의 정보처리 단위로서 연산을 수행하는 세포체<sub>soma</sub> 또는 cell body, 처리한 정보를 다른 뉴런에 전달하는 축삭<sub>axon</sub>, 다른 뉴런으로부터 정보를 받는 수상돌기<sub>dendrite</sub>로 구성
- 사람 뇌는  $10^{11}$ 개 가량의 뉴런, 뉴런마다 1000개 가량의 연결 → 고도의 병렬 처리기
- 반면에 폰 노이만 컴퓨터는 아주 빠른 순차 명령어 처리기



(a) 뇌의 정보처리 단위인 뉴런



(b) 폰 노이만 컴퓨터의 구조

그림 4-1 생물의 정보처리와 컴퓨터의 정보처리의 차이

## 4.1.2 인공 신경망의 발상과 전개

### • 간략한 인공 신경망 역사

- 1943년 맥컬록과 피츠의 계산 모형
- 1949년 헤브의 학습 알고리즘
- 1958년 로젠블랫의 퍼셉트론
- 위드로와 호프의 아달린과 마달린
- 1960년대의 퍼셉트론에 대한 과대한 기대
- 1969년 민스키와 페퍼트의 『Perceptrons』는 퍼셉트론의 한계 지적. XOR 문제도 해결 못하는 선형 분류기에 불과함 입증 → 신경망 연구 퇴조. 인공지능 겨울에 일조
- 1986년 루멜하트의 『Parallel Distributed Processing』은 은닉층을 가진 다층 퍼셉트론 제안 → 신경망 부활
- 1990년대 SVM에 밀리는 형국
- 2000년대 딥러닝으로 인해 신경망이 인공지능의 주류로 자리매김

## 4.1.2 인공 신경망의 발상과 전개

### **NOTE** 디지털 컴퓨터와 역사를 같이 하는 인공 신경망

1940년대에는 컴퓨터과학이라는 학문 분야가 없었다. 당시의 '컴퓨터'는 기계가 아니라 계산을 전문으로 하는 전문 직업인을 지칭했다. 1946년에 세계 최초의 전자식 컴퓨터인 에니악이 탄생했고, 1950년대에 상업용 컴퓨터가 출시된다. 이때쯤 인공 신경망이 등장하니 컴퓨터와 인공 신경망은 얼추 역사를 같이하는 셈이다. 1950년대 들어서야 대학에 컴퓨터과학이라는 학문 분야가 태동하였으며, 1960년대에 제대로 된 교육과정으로 자리를 잡는다. 컴퓨터의 속도와 메모리 용량이 발전함에 따라 신경망의 성능 또한 크게 발전한다. 현재는 고도의 병렬 처리 기인 GPU의 도움으로 딥러닝이 꽃을 피우고 있다.

### • 퍼셉트론

- 현재 기준으로 매우 낮은 기술이지만 신경망 공부에서 중요
  - 퍼셉트론은 다층 퍼셉트론과 딥러닝의 핵심 구성 요소이기 때문
- 퍼셉트론은 단순한 모델이기 때문에 기계 학습의 용어와 원리를 설명하는데 적합

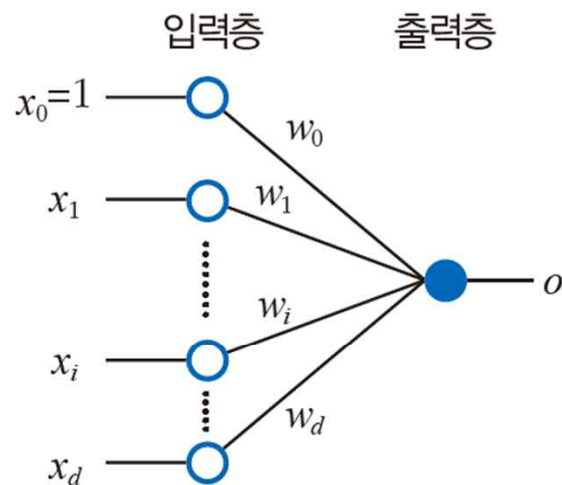
## 4.2.1 퍼셉트론의 구조와 연산

### • 퍼셉트론의 구조

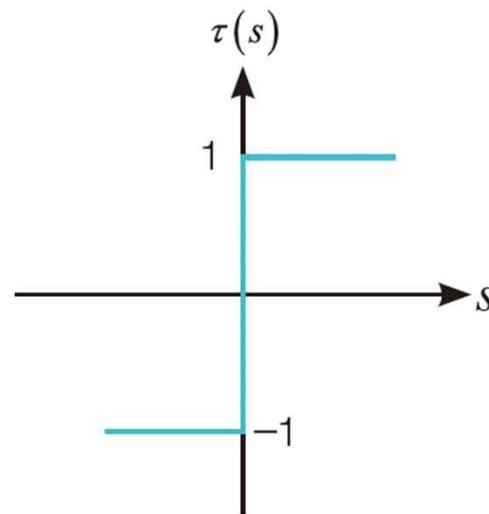
- 입력층과 출력층으로 구성(출력층은 한 개의 노드)
- 입력층은  $d+1$ 개의 노드( $d$ 는 특징 벡터의 차원). 예) iris는  $d=4$ , digit는  $d=64$

특징 벡터:  $\mathbf{x}=(x_1, x_2, \dots, x_d)$  (4.1)

- $i$ 번째 입력 노드와 출력 노드는 가중치  $w_i$ 를 가진 에지로 연결됨



(a) 퍼셉트론의 구조



(b) 계단 함수를 활성화 함수로 사용

그림 4-2 퍼셉트론의 구조와 동작



## 4.2.1 퍼셉트론의 구조와 연산

### • 퍼셉트론의 연산

- $i$ 번째 에지는  $x_i$ 와  $w_i$ 를 곱해 출력 노드로 전달
- 0번째 입력 노드  $x_0$ 은 1인 바이어스 노드
- 출력 노드는  $d+1$ 개의 곱셈 결과를 모두 더한  $s$ 를 계산하고 활성화

함수 activation function 적용

$$\left. \begin{aligned} o &= \tau(s) = \tau\left(\sum_{i=1}^d w_i x_i + w_0\right) \\ \text{이때 } \tau(s) &= \begin{cases} +1, s > 0 \\ -1, s \leq 0 \end{cases} \end{aligned} \right\} \quad (4.2)$$

- 활성화 함수
  - 뉴런을 활성화하는 과정을 모방
  - 퍼셉트론은 활성화 함수로 계단 함수 사용( $s$ 가 0보다 크면 1, 그렇지 않으면 -1 출력)
  - 따라서 퍼셉트론은 특징 벡터를 1 또는 -1로 변환하는 장치, 즉 이진 분류기

## 4.2.2 퍼셉트론으로 인식하기

- 퍼셉트론은 이진 분류기 binary classifier

- 이진 분류 문제의 예)

- 생산 라인에서 나오는 제품을 우량과 불량으로 구분
    - 병원에 온 사람을 정상인과 환자로 구분
    - 내가 등장하는 사진을 구별해 냄

- [예제 4-1] 퍼셉트론의 인식 능력

- 제품을 크기와 색상을 나타내는 두 개의 특징으로 표현한다고 가정( $d=2$ ). 특징 값은 0 또는 1이라 가정

$$\mathbf{x}=(\text{크기}, \text{색상})=(x_1, x_2)$$

- 생산 라인에서 제품 4개를 수집하여 관찰한 결과 다음 데이터를 확보([그림 4-3(a)])

$$x_1=(0,0), \quad x_2=(0,1), \quad x_3=(1,0), \quad x_4=(1,1) \quad \leftarrow \text{특징 벡터}$$

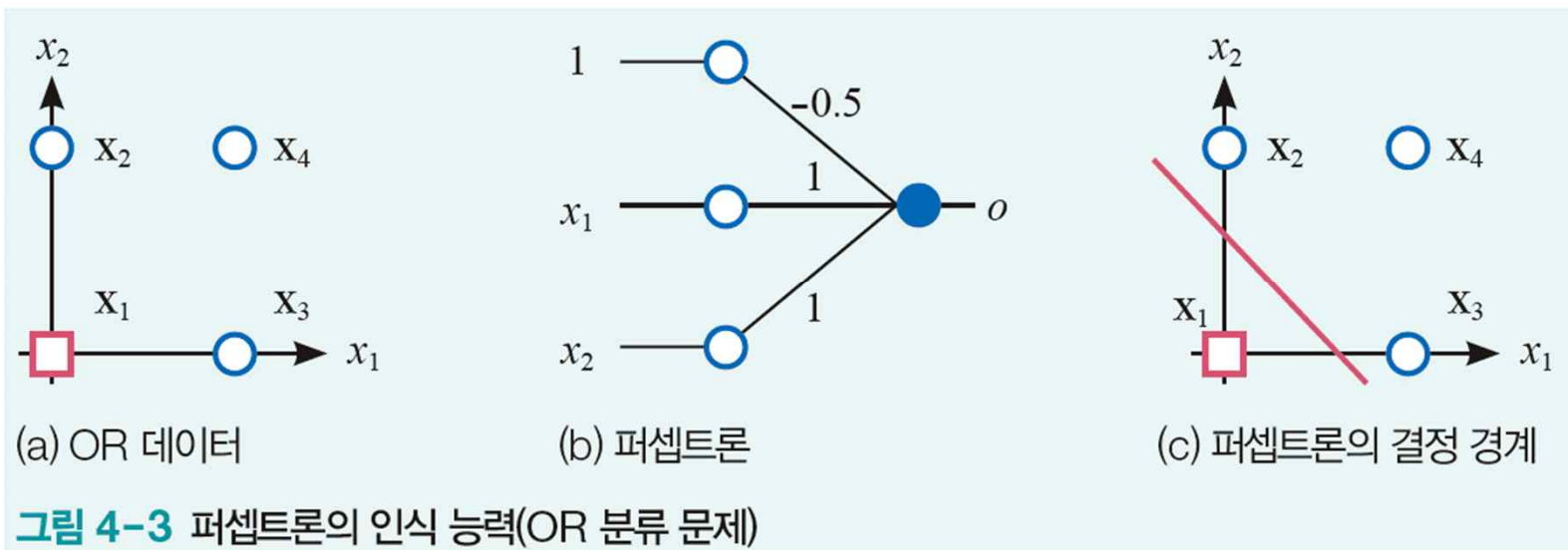
$$y_1=-1, \quad y_2=1, \quad y_3=1, \quad y_4=1 \quad \leftarrow \text{레이블 (} y=1 \text{은 불량, } y=-1 \text{은 우량)}$$

## 4.2.2 퍼셉트론으로 인식하기

- [그림 4-3(b)]는 데이터를 인식하는 퍼셉트론

- $x_2=(0,1)$  샘플을 예측해보면,

$$o = \tau(w_1x_1 + w_2x_2 + w_0) = \tau(1*0 + 1*1 - 0.5) = \tau(0.5) = 1$$



**NOTE** 논리 연산을 본뜬 OR 데이터와 XOR 데이터

[그림 4-3(a)]의 데이터를 분류하는 문제를 OR 분류라고 한다. 1을 참, 0을 거짓으로 간주하면 이진 논리의 OR 연산에 해당하기 때문이다. 단순하여 설명하기 쉽기 때문에 초심자에게 신경망의 동작을 설명할 때 종종 사용한다. 레이블을  $y_1=-1, y_2=-1, y_3=-1, y_4=1$ 로 설정하면 AND 데이터가 되고,  $y_1=-1, y_2=1, y_3=1, y_4=-1$ 로 설정하면 XOR 데이터가 된다.

## 4.2.2 퍼셉트론으로 인식하기

- 퍼셉트론을 수학적으로 해석하면([그림 4-3(c)])

- 가중치  $w_0 = -0.5$ ,  $w_1 = 1$ ,  $w_2 = 1$ 을 식 (4.2)에 대입하면,

$$o = w_1 * x_1 + w_2 * x_2 + w_0 = x_1 + x_2 - 0.5$$

- $o = 0$ 으로 설정하면 직선의 방정식을 얻음 → 특징 공간을 분할하는 결정 경계([그림 4-3(c)])

$$x_1 + x_2 - 0.5 = 0 \text{ 또는 } x_2 = -x_1 + 0.5$$

- 퍼셉트론은 특징 공간을 두 부분 공간으로 분할하는 이진 분류기

- 퍼셉트론은 선형으로 국한됨

## 4.2.3 행렬 표기

- 식 (4.2)를 반복문으로 구현하는 사례

[C 코드]

```
s=0;
for(i=0; i<=d; i++)
    s=s+x[i]*w[i];
```

- 행렬 연산의 필요성

- 대부분 기계 학습 책은 행렬 표기 사용. 파이썬 언어는 주로 행렬 연산 사용하여 코딩
- 파이썬은 다음 코드에서  $x$ 와  $w$ 가 벡터라는 사실을 알고 있어 벡터 곱셈을 해줌

[파이썬 코드]

```
s=sum(x*w) # 요소별로 곱한 후 요소를 합산
```

## 4.2.3 행렬 표기

- 퍼셉트론의 동작을 행렬을 이용해 다시 쓰면,
  - $w$ 와  $x$ 는  $1 \times d$  행렬( $x$ 의 전치 행렬  $x^T$ 는  $d \times 1$  행렬)
  - $w x^T$ 는  $1 \times 1$  행렬로서 스칼라

$$o = \tau \left( \sum_{i=1}^d w_i x_i + b \right) = \tau \left( \underbrace{\mathbf{w} \mathbf{x}^T}_{\text{행렬 표기}} + w_0 \right) \quad (4.3)$$

- 보다 간결한 표현(바이어스를 첫번째 요소로 추가)
  - $w$ 를  $1 \times (d+1)$  행렬  $(w_0, w_1, \dots, w_d)$ ,  $x$ 를  $1 \times (d+1)$  행렬  $(x_0, x_1, \dots, x_d)$ 로 확장하면

$$o = \tau \left( \mathbf{w} \mathbf{x}^T \right) \quad (4.4)$$

## 4.2.3 행렬 표기

- 파이썬 코딩

- [예제 4-1]에 있는 x2 처리

[파이썬 코드 1]

```
import numpy as np                                # numpy 라이브러리를 불러옴
x2=np.array([1,0,1])                              # [그림 4-3(a)]의 샘플 x2
w=np.array([-0.5,1.0,1.0])                        # [그림 4-3(b)]의 퍼셉트론 가중치 벡터
s=sum(x2*w)                                       # 요소별로 곱한 후 합산
```

- [예제 4-1]의 네 개 샘플 x1~x4를 한꺼번에 처리([그림 4-4]는 처리

[파이썬 코드 2]

```
import numpy as np                                # numpy 라이브러리를 불러옴
x=np.array([[1,0,0],[1,0,1],[1,1,0],[1,1,1]])    # [그림 4-3(a)]의 4개 샘플
w=np.array([-0.5,1.0,1.0])                        # [그림 4-3(b)]의 퍼셉트론 가중치 벡터
s=np.sum(x*w,axis=1)                             # 샘플 각각에 대해 요소별로 곱한 후 요소를 합산
```

## 4.2.3 행렬 표기

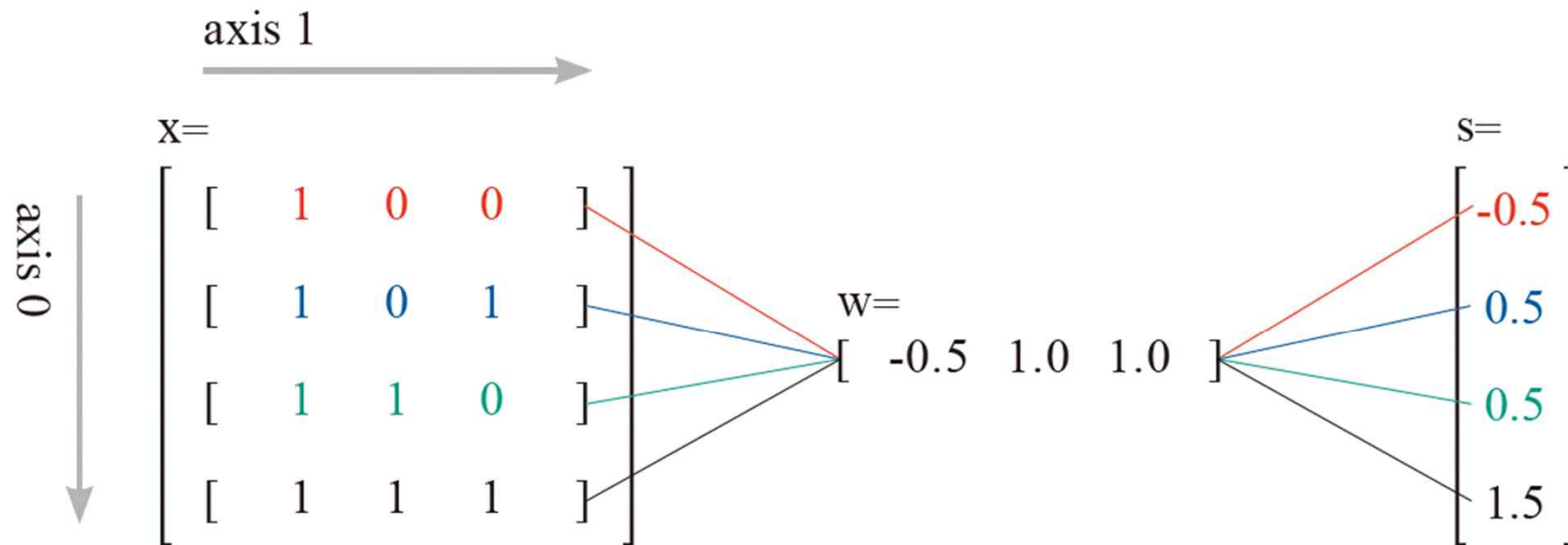


그림 4-4 [파이썬 코드 2]를 계산하는 과정

**TIP** Numpy User Guide와 Numpy Reference Guide는 각각 <https://numpy.org/doc/stable/numpy-user.pdf>와 <https://numpy.org/doc/stable/numpy-ref.pdf>에서 pdf 문서로 다운로드할 수 있다. User Guide는 200쪽 정도이고 Reference Guide는 1,800쪽 정도의 방대한 문서이지만 큰 부담을 느낄 필요는 없다. 기본 개념을 익힌 다음 필요한 곳을 참조하면 된다.



## 4.3 사람의 학습과 신경망의 학습

- 퍼셉트론의 학습

- [그림 4-3]에서는 데이터와 함께 데이터를 인식하는 퍼셉트론(가중치)이 주어짐. 즉 데이터로 학습을 마친 퍼셉트론이 주어짐
- 실제 상황에서는 데이터만 주어지므로, 학습 알고리즘으로 가중치( $w_0, w_2, \dots, w_d$ )를 알아내야 함
  - OR 데이터의 경우 2차원 특징 벡터이고 샘플이 4개 뿐이라 연필을 가지고 쉽게 가중치를 알아낼 수 있음
  - sklearn의 필기 숫자 데이터는 64차원 특징 벡터이고 1,797개 샘플을 가지므로 학습 알고리즘 없이 불가능

## 4.3.1 사람의 학습 알고리즘

- 사람이 수영을 학습하는 과정을 알고리즘 형식으로 기술하면,

### [알고리즘 4-1] 사람의 수영 학습

```
01. 적절한 동작을 취한다.  
02. while (true)  
03.     동작에 따라 수영을 하고 평가한다.  
04.     if (만족스러움) break           # break문으로 루프 탈출  
05.     더 나은 방향으로 동작을 수정한다.  
06. 동작을 기억한다.
```

- 좀 더 수학적으로 기술하면,

### [알고리즘 4-2] 사람의 수영 학습(수학적 기술)

```
01. 초기 동작 벡터  $w$ =(팔 돌리는 속도, 팔꿈치 각도, 팔과 귀의 거리)를 초기화한다.  
02. while (true)  
03.      $w$ 에 따라 수영을 하고 동작  $w$ 의 점수  $J(w)$ 를 계산한다.  
04.     if ( $J(w)$ 가 만족스러움) break  
05.     더 나은 방향  $\Delta w$ 를 계산한다.  
06.      $w=w+\Delta w$   
07.  $w$ 를 기억한다.
```

## 4.3.2 신경망의 학습 알고리즘

- 조금씩 개선해 나가는 절차는 사람의 학습과 같음
  - 신경망은 철저히 수학에 의존한다는 점에서 사람과 다름
    - 신경망의 학습은 최적화 문제. 최적화 대상은 신경망의 가중치(매개변수)

### [알고리즘 4-3] 신경망의 학습

입력: 훈련 데이터

출력: 최적의 매개변수 값

```
01. 난수로 매개변수 벡터  $\mathbf{w}$ 를 초기화한다. #  $\mathbf{w}$ 는 신경망의 가중치([그림 4-2(a)]의  
     $(w_0, w_1, \dots, w_d)$ )  
02. while (true)  
03.      $\mathbf{w}$ 에 따라 데이터를 인식하고 손실 함수  $J(\mathbf{w})$ 를 계산한다.  
04.     if ( $J(\mathbf{w})$ 가 만족스러움) break  
05.     손실 함수 값을 낮추는 방향  $\Delta\mathbf{w}$ 를 계산한다.  
06.      $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$   
07.  $\mathbf{w}$ 를 저장한다.
```

- 구현에 필요한 사항
  - 1행의 초기화는 어떻게? 4행의 멈춤 조건은 어떻게?
  - 3행의 손실 함수 정의(보통 오류의 양을 사용)
  - 5행에서  $\Delta\mathbf{w}$ 를 어떻게 구하나?(미분을 사용)

## 4.4 퍼셉트론 학습 알고리즘

- 학습 알고리즘 고안

- 손실 함수  $J$ 를 설계([알고리즘 4-3]의 03행)
- 손실 함수의 값을 낮추는 방향을 찾는 방법([알고리즘 4-3]의 05행)

## 4.4.1 손실 함수 설계

### • 손실 함수 $J$ 가 만족해야 할 조건

- ①  $\mathbf{w}$ 가 훈련 집합에 있는 샘플을 모두 맞히면, 즉 정확률이 100%이면  $J(\mathbf{w})$ 는 0이다.
- ②  $\mathbf{w}$ 가 틀리는 샘플이 많을수록  $J(\mathbf{w})$ 의 값이 크다.

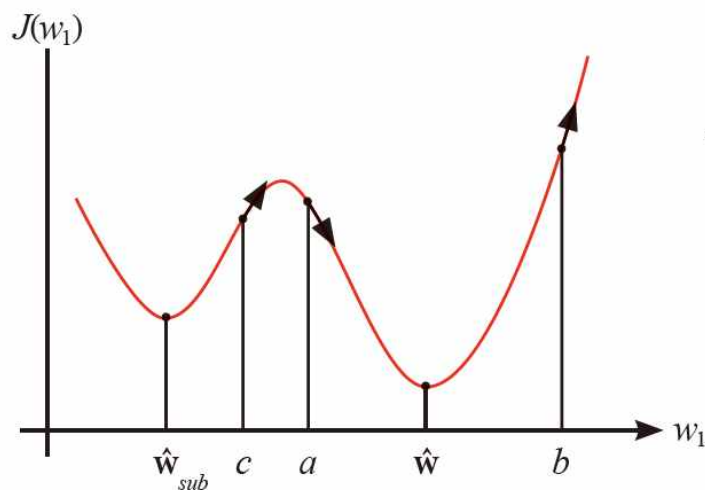
- 조건을 만족하는 함수는 여럿. 식 (4.5)는 그 중 하나로서 직관적으로 이해하기 쉬움
  - $I$ 는  $\mathbf{w}$ 가 틀리는 샘플
  - $x$ 가 +1 부류라면(즉  $y=1$ 이라면), 퍼셉트론의 출력  $\mathbf{w}\mathbf{x}^T$ 는 음수.  $-y(\mathbf{w}\mathbf{x}^T)$ 는 양수
  - $x$ 가 -1 부류라면(즉  $y=-1$ 이라면), 퍼셉트론의 출력  $\mathbf{w}\mathbf{x}^T$ 는 양수.  $-y(\mathbf{w}\mathbf{x}^T)$ 는 양수
  - 결국 틀린 샘플  $x$ 는 손실 함수의 값을 증가시킴(틀린 샘플이 많을수록 손실 함수 값은 커짐)

$$J(\mathbf{w}) = \sum_{\mathbf{x} \in I} -y(\mathbf{w}\mathbf{x}^T) \quad (4.5)$$

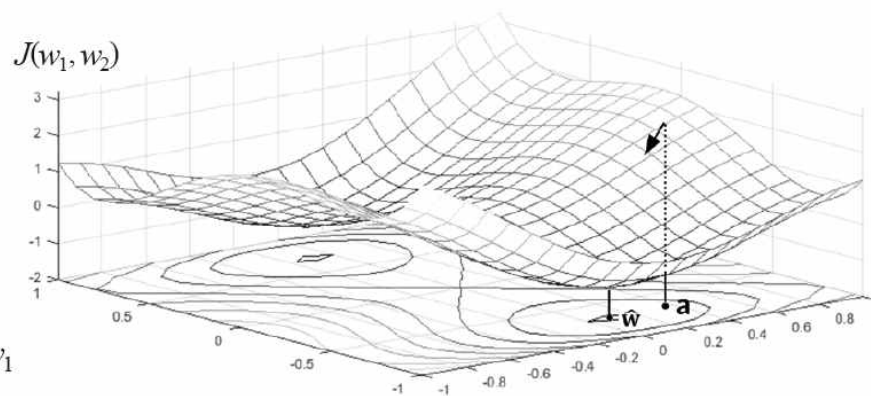
## 4.4.2 학습 알고리즘 설계

### • 경사 하강법 $\text{gradient descent}$ 의 원리([그림 4-5]는 가상의 손실 함수)

- 편의상 매개변수가 한 개인 경우와 두 개인 경우를 가지고 설명
- 학습 알고리즘은  $J$ 의 최저점  $\hat{\mathbf{w}}$ 을 찾아야 함



(a) 매개변수가 1개인 경우:  $\mathbf{w}=(w_1)$



(b) 매개변수가 2개인 경우:  $\mathbf{w}=(w_1, w_2)$

그림 4-5 경사 하강법

## 4.4.2 학습 알고리즘 설계

- 학습 규칙 유도([그림 4-5(a)]는 매개변수가 하나인 경우)

- 경사 하강법은 미분을 이용하여 최적해를 찾는 기법
- 미분값  $\frac{\partial J}{\partial w_1}$ 의 반대 방향이 최적해에 접근하는 방향이므로 현재  $w_1$ 에  $-\frac{\partial J}{\partial w_1}$ 를 더하면 최적해에 가까워짐  $\rightarrow$  식 (4.6)의 학습 규칙
- 방향은 알지만 얼마나 가야하는지에 대한 정보가 없기 때문에 학습률  $\rho$ 를 곱하여 조금씩 이동( $\rho$ 는 하이퍼 매개변수로서 보통 0.001이나 0.0001처럼 작은 값 사용)

$$w_1 = w_1 + \rho \left( -\frac{\partial J}{\partial w_1} \right) \quad (4.6)$$

- 매개변수가 여럿인 경우

- 편미분으로 구한 그레디언트를 사용 (매개변수 별로 독립적 미분)

$$\left. \begin{aligned} \mathbf{w} &= \mathbf{w} + \rho(-\nabla \mathbf{w}) \\ \nabla \mathbf{w} &= \left( \frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \dots, \frac{\partial J}{\partial w_d} \right) \end{aligned} \right\} \quad (4.7)$$

## 4.4.2 학습 알고리즘 설계

- 퍼셉트론에 식 (4.7) 적용

- 식 (4.5)를 매개변수  $w_i$ 로 편미분하면,

$$\frac{\partial J}{\partial w_i} = \sum_{\mathbf{x} \in I} \frac{\partial (-y(w_0 + w_1 x_1 + \dots + w_i x_i + \dots + w_d x_d))}{\partial w_i} = \sum_{\mathbf{x} \in I} -y x_i$$


- 정리하면,

$$\frac{\partial J}{\partial w_i} = \sum_{\mathbf{x} \in I} -y x_i, \quad i = 0, 1, 2, \dots, d \quad (4.8)$$

- 식 (4.8)을 식 (4.7)에 대입하면,

$$\text{퍼셉트론 학습 규칙: } w_i = w_i + \rho \sum_{\mathbf{x} \in I} y x_i, \quad i = 0, 1, 2, \dots, d \quad (4.9)$$

- 식 (4.9)를 행렬 형태로 쓰면, 퍼셉트론 학습 알고리즘이 사용하는 핵심 공식


$$\text{퍼셉트론의 학습 규칙(행렬 표기): } \mathbf{w} = \mathbf{w} + \rho \sum_{\mathbf{x} \in I} y \mathbf{x} \quad (4.10)$$



## 4.4.2 학습 알고리즘 설계

### • 퍼셉트론의 학습 알고리즘

- 식 (4.10)을 [알고리즘 4-3]에 적용하면 [알고리즘 4-4]
- 03~08행을 수행하여 훈련 집합에 있는 샘플 전체를 한번 처리하는 일을 세대<sub>epoch</sub>라 부름

#### [알고리즘 4-4] 퍼셉트론의 학습

입력: 훈련 집합( $n$ 은 샘플의 개수)

출력: 최적의 매개변수  $\hat{\mathbf{w}}$

```
01. 난수로 매개변수 벡터  $\mathbf{w}$ 를 초기화한다. #  $\mathbf{w}$ 는 퍼셉트론 가중치
02. while (true)
03.    $I=[]$                                 # 공집합
04.   for  $j=1$  to  $n$ 
05.      $o=\tau(\mathbf{w}\mathbf{x}_j^T)$                 # 식 (4.4)를 적용해 인식 수행
06.     if( $o \neq y_j$ )  $I=I \cup \mathbf{x}_j$       # 틀린 샘플을  $I$ 에 추가
07.     if( $I=\text{공집합}$ ) break            # 모든 샘플을 맞히면(손실 함수가 0) 루프를 빠져 나감
08.      $\mathbf{w}=\mathbf{w}+\rho \sum_{\mathbf{x} \in I} y\mathbf{x}$       # 식 (4.10)을 적용해 매개변수 갱신
09.    $\hat{\mathbf{w}}=\mathbf{w}$ 
```

- [알고리즘 4-4]는 데이터가 선형 분리 불가능한 경우 무한 루프
- 여러 세대에 걸쳐  $I$ 의 크기가 줄지 않으면 수렴했다고 판단하고 루프를 빠져나오게 수정

## 4.4.2 학습 알고리즘 설계

- [그림 4-6]은 퍼셉트론의 학습 알고리즘을 개념적으로 설명
  - 알고리즘은 전방 계산(05행) → 오차 계산(06행) → 후방 가중치 갱신을 반복(08행)
  - 딥러닝을 포함하여 신경망 학습 알고리즘은 모두 이 절차를 따름(딥러닝은 많은 층을 거쳐 전방 계산과 후방 가중치 갱신을 수행)

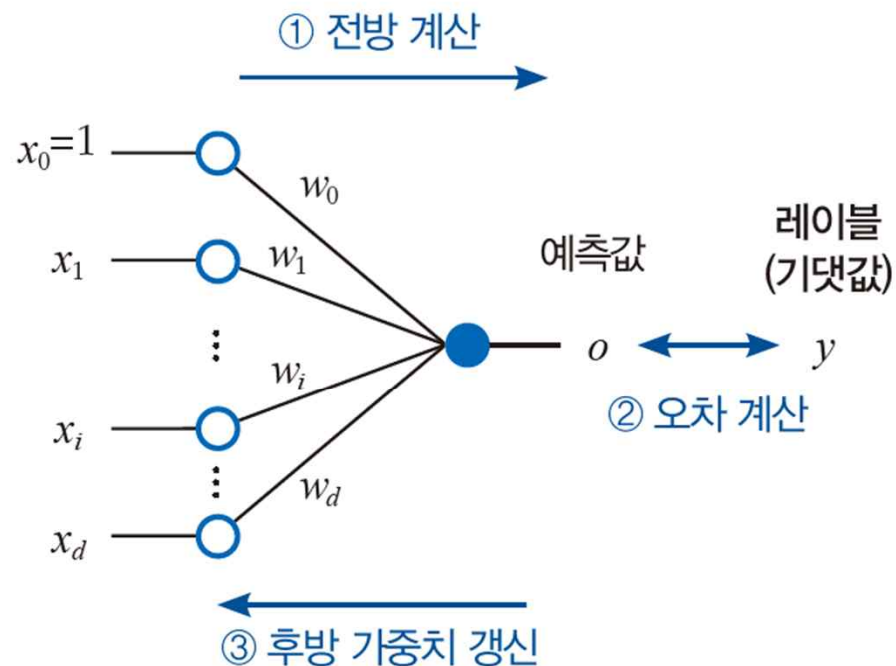


그림 4-6 퍼셉트론 학습 알고리즘의 절차

## 4.4.2 학습 알고리즘 설계

### NOTE 수학은 인공지능을 공부하는 데 등대일까? 횡방꾼일까?

수학은 인공지능의 주춧돌이다. 따라서 피하려야 피할 수가 없다. 수학이 부담스럽다면 나에게 필요한 최소한의 수학이 무엇인지 곰곰이 생각한 다음 딱 그만큼만 공부하는 것도 좋다.

인공지능이 주로 사용하는 수학은 선형 대수와 미분, 확률이다. 신경망 이론은 선형 대수와 미분을 주로 사용한다. 선형 대수는 신경망의 입력과 출력, 가중치, 중간 계산 결과 등을 표현하고 이들이 수행하는 연산을 표현하는 데 주로 사용한다. 미분은 손실 함수를 최적화하는 데 주로 사용한다. 부록 A에서는 이 책을 이해하는 데 필요한 최소한의 선형 대수를 설명한다. 예제를 동원해 최대한 쉽게 설명하니 차분하게 처음부터 끝까지 공부할 것을 권한다. 부록 A를 통해 수학에 대한 두려움을 벗고 즐거운 마음으로 인공지능을 공부할 수 있기를 바란다. 수학은 인공지능을 이해하고 만드는 데 도움이 되는 친절한 길잡이다.

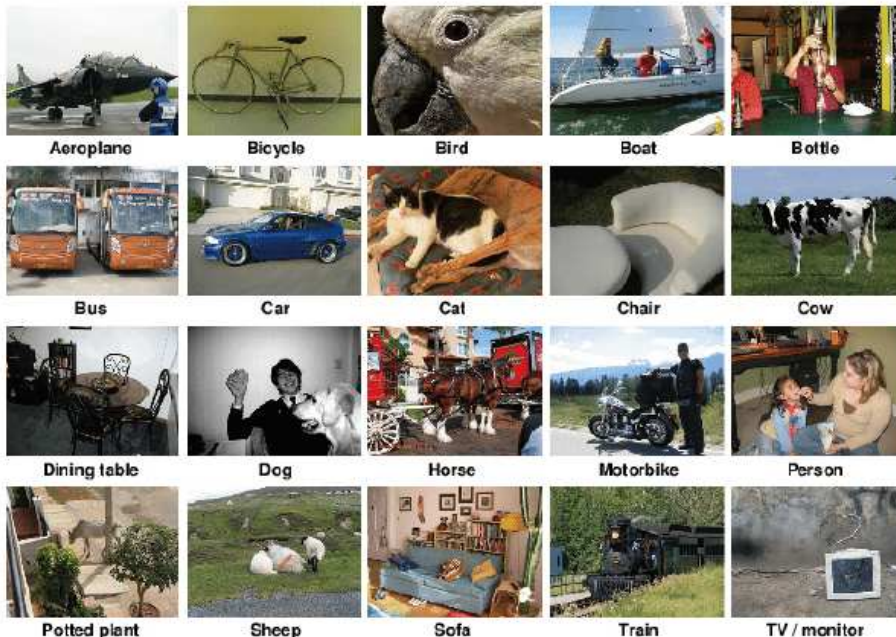
그리고 이 장에서 다루는 수학을 부분적으로만 이해했다고 해서 이 책을 더 이상 따라갈 수 없는 것은 아니다. 할 수 있는 만큼 이해한 다음 앞으로 전진하자. 한참 가다가 뒤돌아보면 수학을 가리고 있던 안개가 확 걷힐 수 있다. 이 절은 이 책에서 수학을 가장 많이 다루고 있다. 이후에는 크게 줄어든다.



## 4.5.1 현대적인 기계학습의 복잡도

### • [그림 4-7]의 세 가지 문제

- 분류 **classification**: 지정된 몇 가지 부류로 구분하는 문제
- 물체 검출 **object detection**: 물체 위치를 바운딩 박스로 찾아내는 문제(사진 촬영에서 얼굴 초점 응용)
- 물체 분할 **object segmentation**: 화소 수준으로 물체를 찾아내는 문제(추적 대상 물체를 형광색으로 칠하는 응용)



(b) 검출 문제



(c) 분할 문제

그림 4-7 세 종류의 문제와 데이터셋

## 4.5.1 현대적인 기계학습의 복잡도

- 적절한 손실 함수 필요

- 분류: 참 값과 예측 값의 차이를 손실 함수에 반영
- 물체 검출: 참 바운딩 박스와 예측한 바운딩 박스의 겹침 정도를 손실 함수에 반영
- 물체 분할: 화소 별로 레이블이 지정한 물체에 해당하는지 따지는 손실 함수

- 실용적인 신경망의 매개변수 개수는 방대

- 예) 딥러닝 모델 ResNet-50은 50개의 층을 가지며 매개변수는 2천300만개 이상
- 다행히 1~2차원에서 개발한 공식이 고차원에 그대로 적용
- 단지 과잉 적합을 방지하기 위해 더 많은 데이터 사용 또는 더 정교한 규제 기법 적용. 속도 향상을 위한 GPU 사용

### • 경사 하강법

- 자연과학과 공학에서 오랫동안 사용해온 최적화 방법([그림 4-5])
- 예) 항공공학자
  - 날개를 설계할 때 두께, 폭, 길이, 곡률 등을 매개변수로 설정한 다음 유체 역학 이론을 기반으로 손실 함수 정의
  - 손실 함수는 매개변수 변화에 따른 연료 소비량을 측정
  - 경사 하강법으로 최적해를 구한 다음 날개 제작

### • 기계 학습의 경사 하강법

- 여러 측면에서 표준 경사 하강법과 다름
  - 잡음이 섞인 데이터가 개입
  - 매개변수가 방대
  - 일반화 능력 필요

← 이런 특성은 최적해를 찾는 일을 어렵게 만듦. 정확률이 등락을 거듭하며 수렴하지 않는다거나 훈련 집합에 대해 높은 성능을 얻었는데 테스트 집합에 대해 형편 없는 성능 등의 문제

## 4.5.2 스토캐스틱 경사 하강법

- 기계 학습은 스토캐스틱 경사 하강법으로 확장하여 사용
  - 배치 모드 vs. 패턴 모드
    - [알고리즘 4-4]는 배치 모드: 틀린 샘플을 모은 다음 한꺼번에 매개변수 갱신
    - 패턴 모드는 패턴 별로 매개변수 갱신(세대를 시작할 때 샘플을 뒤섞어 랜덤 샘플링 효과 제공)
  - 딥러닝은 주로 미니 배치 사용(패턴 모드와 배치 모드의 중간)
    - 훈련 집합을 일정한 크기의 부분 집합으로 나눈 다음 부분 집합 별로 처리
    - 부분 집합으로 나눌 때 랜덤 샘플링을 적용하기 때문에 스토캐스틱 경사 하강법(SGD, stochastic gradient descent)이라 부름

## 4.6 퍼셉트론 프로그래밍

- 퍼셉트론 프로그래밍을 해보면 현대적인 신경망을 프로그래밍하는데 크게 도움
- 이 절에서는 sklearn 라이브러리를 활용해서 퍼셉트론 프로그래밍

**TIP** sklearn 라이브러리에서 제공하는 퍼셉트론은 [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Perceptron.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html)에서 확인할 수 있다.



## 4.6.1 OR 데이터 인식

### • [프로그램 4-1]

프로그램 4-1

OR 데이터에 퍼셉트론 적용

```
01  from sklearn.linear_model import Perceptron
02
03  # 훈련 집합 구축
04  X=[[0,0],[0,1],[1,0],[1,1]]
05  y=[-1,1,1,1]
06
07  # fit 함수로 Perceptron 학습
08  p=Perceptron()
09  p.fit(X,y)
10
11  print("학습된 퍼셉트론의 매개변수: ",p.coef_,p.intercept_)
12  print("훈련집합에 대한 예측: ",p.predict(X))
13  print("정확률 측정: ",p.score(X,y)*100,"%")
```

학습된 퍼셉트론의 매개변수:  $\begin{bmatrix} 2. & 2. \end{bmatrix}$   $[-1.]$

훈련집합에 대한 예측:  $[-1 \ 1 \ 1 \ 1]$

정확률 측정: 100.0%

## 4.6.2 필기 숫자 데이터 인식

- [프로그램 4-2]는 sklearn이 제공하는 숫자 데이터에 퍼셉트론 적용
  - SVM을 사용한 [프로그램 3-5]와 비슷함(음영 표시한 행만 달라짐)

프로그램 4-2

필기 숫자 데이터에 퍼셉트론 적용

```
01 from sklearn import datasets
02 from sklearn.linear_model import Perceptron
03 from sklearn.model_selection import train_test_split
04 import numpy as np
05
06 # 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
07 digit=datasets.load_digits()
08 x_train,x_test,y_train,y_test=train_test_split(
09     (digit.data,digit.target,train_size=0.6)
10
11 # fit 함수로 Perceptron 학습
12 p=Perceptron(max_iter=100,eta0=0.001,verbose=0)
13 p.fit(x_train,y_train) # digit 데이터로 모델링
14 res=p.predict(x_test) # 테스트 집합으로 예측
15
```

데이터 읽기

모델 객체 생성

모델 학습

학습된 모델로 예측

## 4.6.2 필기 숫자 데이터 인식

- 실행 결과 93.88% 정확률

- [프로그램 3-6]의 SVM 모델의 98.74%보다 열등(퍼셉트론은 선형 분류기이기 때문)

```
16 # 혼동 행렬
17 conf=np.zeros((10,10))
18 for i in range(len(res)):
19     conf[res[i]][y_test[i]]+=1
20 print(conf)
21
22 # 정확률 계산
23 no_correct=0
24 for i in range(10):
25     no_correct+=conf[i][i]
26 accuracy=no_correct/len(res)
27 print("테스트 집합에 대한 정확률은 ", accuracy*100, "%입니다.")
```

성능 측정

```
[[66.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0. 65.  0.  0.  0.  0.  1.  0.  6.  2.]
 [ 0.  2. 58.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0. 62.  0.  0.  0.  0.  1.  0.]
 [ 0.  1.  0.  0. 60.  0.  0.  2.  1.  0.]
 [ 0.  0.  0.  0.  0. 71.  0.  0.  1.  1.]
 [ 0.  0.  0.  0.  0.  0. 80.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. 75.  0.  0.]
 [ 0.  3.  0.  2.  0.  0.  1.  1. 66.  0.]
 [ 0.  1.  0.  1.  0.  9.  0.  3.  2. 72.]]
```

테스트 집합에 대한 정확률은 93.88038942976355%입니다.

## 4.6.3 기계 학습의 디자인 패턴

- 물건에 깃든 패턴

- 예) 달구지, 마차, 자동차, 자율 주행차 모두 둥근 모양의 바퀴 사용
- 바퀴의 재발명 reinventing the wheel 을 피하려면 패턴을 잘 활용해야 함

- 컴퓨터 프로그램에 깃든 패턴

- 『Design Patterns: Elements of Reusable Object-oriented Software』은 소프트웨어의 디자인 패턴의 중요성을 강조
- 예)  $a=[12,3,2,5,20,17]$  리스트에서 최소값 찾기, 최대값 찾기, 평균 구하기는 모두 리스트를 순차적으로 훑는 연산 필요 → Iterator라는 디자인 패턴

## 4.6.3 기계 학습의 디자인 패턴

- 기계 학습에 깃든 패턴

- 모델만 달라지고 나머지는 같은 경우가 많음. 예) [프로그램 3-5]와 [프로그램 4-2]
- 기계 학습의 디자인 패턴

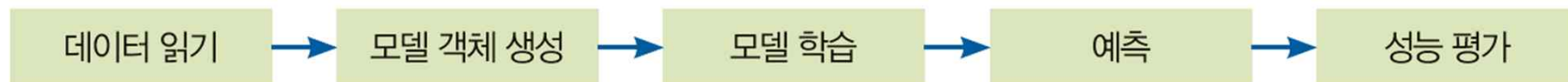


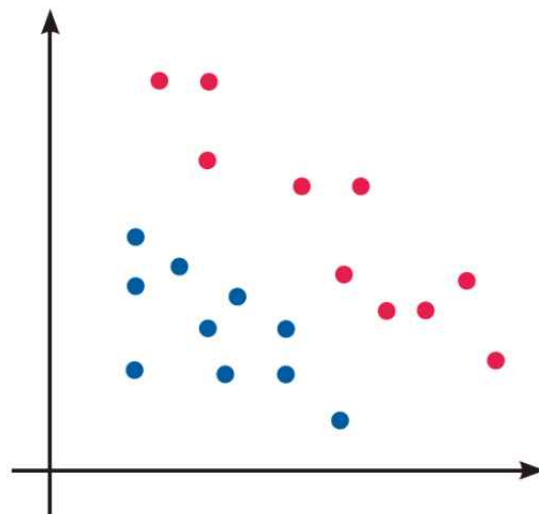
그림 4-8 sklearn의 디자인 패턴

- tensorflow를 이용한 딥러닝 프로그래밍에도 디자인 패턴 풍부
- 디자인 패턴의 계층 구조
  - 하위 작업에도 디자인 패턴 존재. 예) 혼동 행렬 구하기
- 디자인 패턴은 구체적인 코딩 기술이기도 하지만 패턴을 잘 이용하여 효율적으로 프로그래밍하겠다는 마음가짐이기도 함

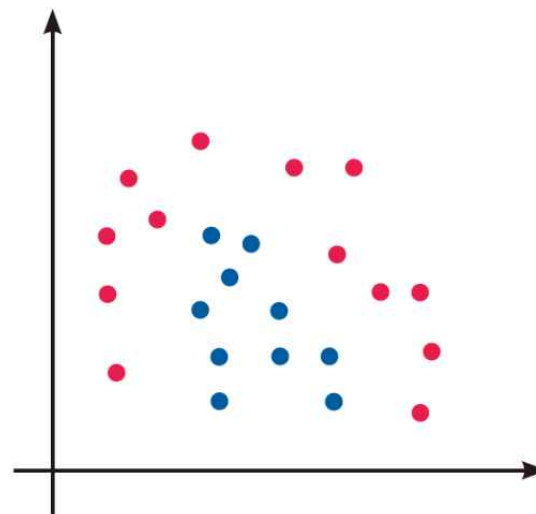
## 4.7 다층 퍼셉트론

- 퍼셉트론은 선형이라는 한계

- [그림 4-9]의 선형 분리 불가능한 데이터에서는 높은 오류율([프로그램 4-2]의 6.12% 오류율의 원인)



(a) 선형 분리 가능



(b) 선형 분리 불가능

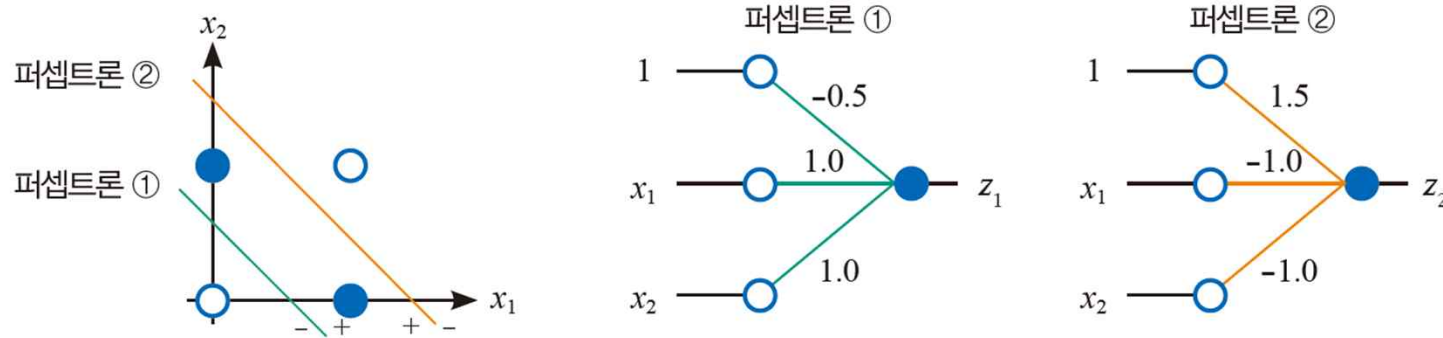
그림 4-9 데이터의 선형 분리 가능성

- XOR에서는 25% 오류율([그림 4-10(a)])

- 이 절에서는 은닉층을 추가한 다층 퍼셉트론으로 비선형으로 확장

## 4.7.1 특징 공간 변환

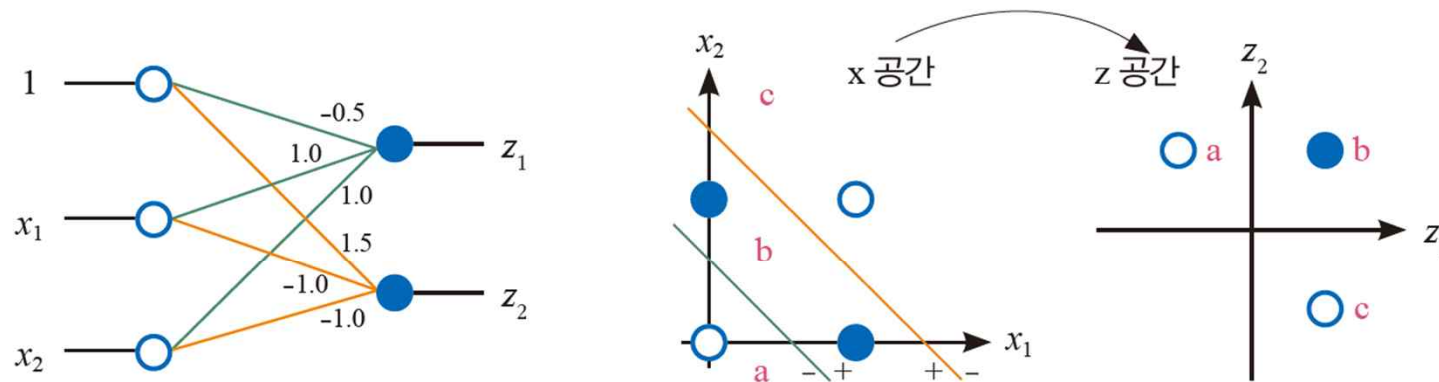
- 퍼셉트론 두 개로 특징 공간을 세 개의 부분 공간으로 나눌 수 있음



(a) 퍼셉트론 2개를 이용해 부분공간 3개로 분할 (b) 2개의 퍼셉트론

그림 4-10 퍼셉트론 2개로 XOR 데이터를 인식하는 길을 찾음

- 두 퍼셉트론을 병렬로 결합하면  $(x_1, x_2)$  공간을  $(z_1, z_2)$  공간으로 변환



(a) 공간 변환하는 퍼셉트론 2개의 병렬 결합

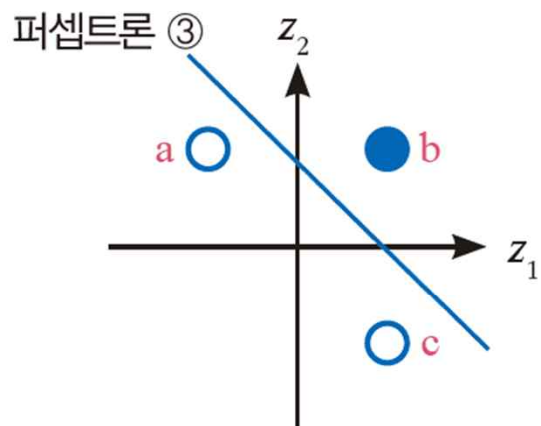
(b) 특징 공간 변환(세 부분공간을 세 점으로 매핑)

그림 4-11 퍼셉트론 2개를 병렬 결합해 특징 공간을 변환

## 4.7.1 특징 공간 변환

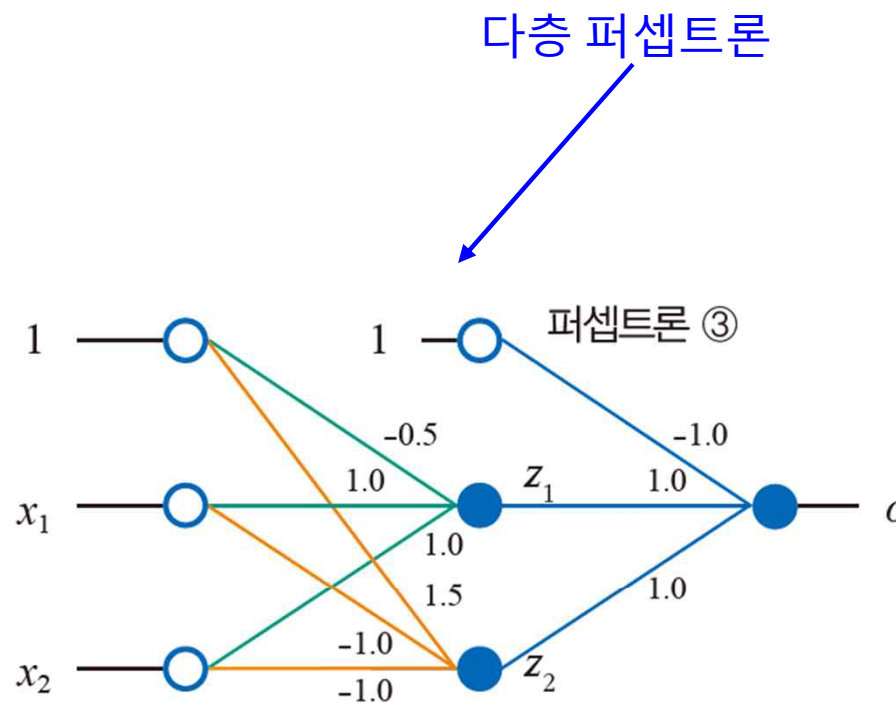
- 새로운 공간  $(z_1, z_2)$ 의 흥미로운 특성

- 선형 분리 불가능하던 네 점이 선형 분리 가능해짐
- 퍼셉트론을 하나 더 쓰면 XOR 문제를 푸는 신경망 완성



(a) 공간 분할하는 세 번째 퍼셉트론

그림 4-12 다층 퍼셉트론



(b) 세 번째 퍼셉트론을 순차적으로 덧붙임



## 4.7.1 특징 공간 변환

- [예제 4-2] XOR를 푸는 다층 퍼셉트론

[그림 4-12(b)]의 다층 퍼셉트론에 XOR 데이터의 샘플 4개를 차례로 입력하고 제대로 인식하는지 확인해보자. 아래 표는 샘플 4개를 인식하는 과정을 보여준다. 100% 옳게 분류하는 것을 확인할 수 있다.

원래 특징 공간( $x_1, x_2$ )	은닉 특징 공간( $z_1, z_2$ )	출력 $o$	레이블 $y$
(0,0)	(-1,1)	-1	-1
(0,1)	(1,1)	1	1
(1,0)	(1,1)	1	1
(1,1)	(1,-1)	-1	-1

## 4.7.1 특징 공간 변환

- 신경망을 공간 변환기로 볼 수 있음

- 원래 특징 공간을 임시 공간으로 변환하고, 임시 공간을 레이블 공간으로 변환하는 두 단계 처리
- 임시 공간에 해당하는 층을 은닉층<sub>hidden layer</sub>이라 부름. 임시 공간을 은닉 공간<sub>hidden space</sub> 또는 잠복 공간<sub>latent space</sub>이라 부름
- 새로운 공간은 이전 공간보다 분류에 더 유리하도록 학습됨
- 자연 영상이나 자연어처럼 복잡한 데이터는 대여섯 은닉층 또는 수십~수백 개의 은닉층 → 이런 깊은 신경망의 학습을 딥러닝이라 부름

## 4.7.2 다층 퍼셉트론의 구조

- 입력층, 은닉층, 출력층으로 구성된 다층 퍼셉트론
  - 층을 연결하는 가중치 뭉치가 두 개 있어 3층이 아니라 2층 신경망으로 간주함
  - 데이터에 따라 입력층과 출력층의 노드 개수 확정
    - 예) iris에서는 5개의 입력 노드와 3개의 출력 노드
  - 은닉층의 노드 개수는 하이퍼 매개변수
    - 은닉 노드가 많으면 신경망 용량이 커지지만 과잉 적합 가능성 높아짐

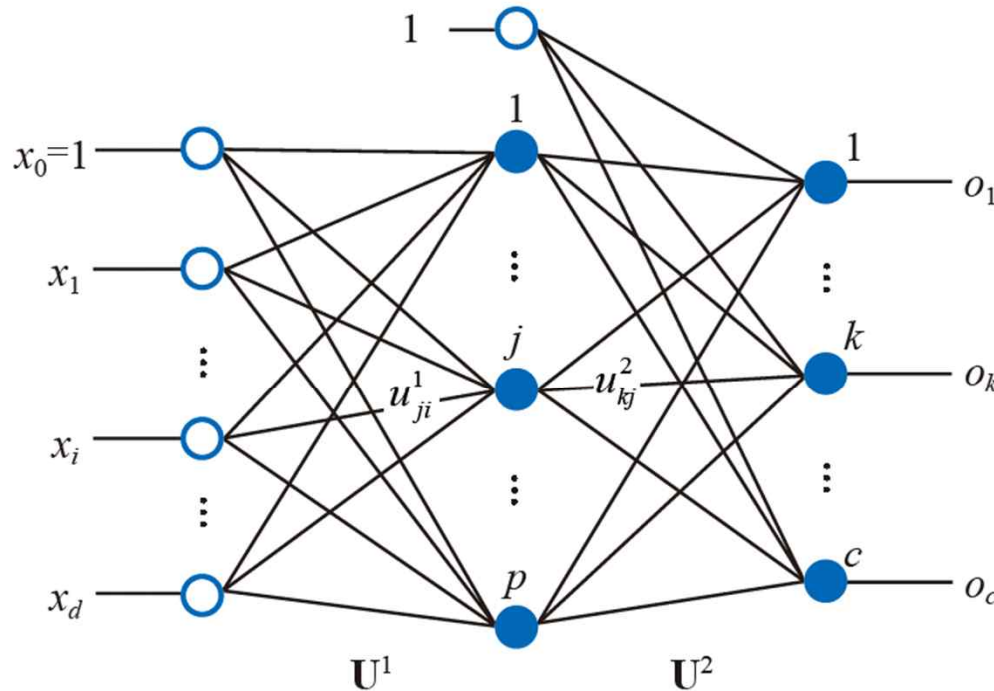


그림 4-13 다층 퍼셉트론의 구조

## 4.7.2 다층 퍼셉트론의 구조

- 가중치 행렬  $\mathbf{U}^1$ 과  $\mathbf{U}^2$

- $u_{ji}^1$ 은  $i$ 번째 입력 노드와  $j$ 번째 은닉 노드를 연결하는 가중치
- $u_{kj}^2$ 는  $j$ 번째 은닉 노드와  $k$ 번째 출력 노드를 연결하는 가중치

$$\mathbf{U}^1 = \begin{pmatrix} u_{10}^1 & u_{11}^1 & \cdots & u_{1d}^1 \\ u_{20}^1 & u_{21}^1 & \cdots & u_{2d}^1 \\ \vdots & \vdots & \ddots & \vdots \\ u_{p0}^1 & u_{p1}^1 & \cdots & u_{pd}^1 \end{pmatrix}, \quad \mathbf{U}^2 = \begin{pmatrix} u_{10}^2 & u_{11}^2 & \cdots & u_{1p}^2 \\ u_{20}^2 & u_{21}^2 & \cdots & u_{2p}^2 \\ \vdots & \vdots & \ddots & \vdots \\ u_{c0}^2 & u_{c1}^2 & \cdots & u_{cp}^2 \end{pmatrix} \quad (4.11)$$

### 4.7.3 다층 퍼셉트론의 동작

- **j번째 은닉 노드의 동작([그림 4-13])**

- $u_j^1$ 은  $U^1$ 의 j번째 행
- 은닉층 때문에 첨자가 많아져 복잡해 보이지만 본질적으로 퍼셉트론의 식 (4.2)와 동일

j번째 은닉 노드의 연산:

$$z_j = \tau_1(s_j), j=1,2,\dots,p \quad (4.12)$$

이때  $s_j = \mathbf{u}_j^1 \mathbf{x}^T$ 이고  $\mathbf{u}_j^1 = (u_{j0}^1, u_{j1}^1, \dots, u_{jd}^1)$ ,  $\mathbf{x} = (1, x_1, x_2, \dots, x_d)$

- **k번째 출력 노드의 동작**

- $u_k^2$ 는  $U^2$ 의 k번째 행

k번째 은닉 노드의 연산:

$$o_k = \tau_2(s_k), k=1,2,\dots,c \quad (4.13)$$

이때  $s_k = \mathbf{u}_k^2 \mathbf{z}^T$ 이고  $\mathbf{u}_k^2 = (u_{k0}^2, u_{k1}^2, \dots, u_{kp}^2)$ ,  $\mathbf{z} = (1, z_1, z_2, \dots, z_p)$

### 4.7.3 다층 퍼셉트론의 동작

- 전체 과정을 행렬로 표현하면,
  - 활성화 함수  $\tau_1$ 은 주로 ReLU,  $\tau_2$ 는 주로 softmax 사용

$$\mathbf{o} = \tau_2 \left( \mathbf{U}^2 \tau_1 \left( \mathbf{U}^1 \mathbf{x}^T \right) \right) \quad (4.14)$$

- 훈련 집합에 있는 샘플 전체를 한꺼번에 처리하는 식으로 쓰면,

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} = \begin{pmatrix} 1 & x_1^1 & \cdots & x_d^1 \\ 1 & x_1^2 & \cdots & x_d^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^n & \cdots & x_d^n \end{pmatrix} \quad (4.15)$$

$$\mathbf{O} = \tau_2 \left( \mathbf{U}^2 \tau_1 \left( \mathbf{U}^1 \mathbf{X}^T \right) \right) \quad (4.16)$$

## 4.7.3 다층 퍼셉트론의 동작

- [예제 4-3] 다층 퍼셉트론의 동작

- [그림 4-12(b)] 다층 퍼셉트론의 가중치 행렬( $d=2, p=2, c=1$ )

$$\mathbf{U}^1 = \begin{pmatrix} -0.5 & 1.0 & 1.0 \\ 1.5 & -1.0 & -1.0 \end{pmatrix}, \mathbf{U}^2 = (-1 \quad 1.0 \quad 1.0)$$

- 샘플을 하나씩 처리하는 식 (4.14) 적용
  - $\mathbf{x}=(0,1)$  샘플을 처리(계단 함수 사용, 바이어스 항을 추가해  $(1,0,1)$  형식으로 입력)

$$\begin{aligned} \mathbf{O} &= \tau_2 \left( (-1 \quad 1.0 \quad 1.0) \tau_1 \left( \begin{pmatrix} -0.5 & 1.0 & 1.0 \\ 1.5 & -1.0 & -1.0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right) \right) \\ &= \tau_2 \left( (-1 \quad 1.0 \quad 1.0) \tau_1 \left( \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \right) \right) \\ &= \tau_2 \left( (-1 \quad 1.0 \quad 1.0) \begin{pmatrix} 1 \\ 1.0 \\ 1.0 \end{pmatrix} \right) \\ &= \tau_2 \left( (1) \right) = 1 \end{aligned}$$

← 레이블과 같으므로 맞음

## 4.7.3 다층 퍼셉트론의 동작

- [예제 4-3] 다층 퍼셉트론의 동작(...앞에서 계속)

- 데이터를 한꺼번에 처리하는 식 (4.16) 적용

$$\mathbf{X} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\begin{aligned} \mathbf{O} &= \tau_2 \left( (-1 \ 1.0 \ 1.0) \tau_1 \left( \begin{pmatrix} -0.5 & 1.0 & 1.0 \\ 1.5 & -1.0 & -1.0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \right) \right) \\ &= \tau_2 \left( (-1 \ 1.0 \ 1.0) \tau_1 \left( \begin{pmatrix} -0.5 & 0.5 & 0.5 & 1.5 \\ 1.5 & 0.5 & 0.5 & -0.5 \end{pmatrix} \right) \right) \\ &= \tau_2 \left( (-1 \ 1.0 \ 1.0) \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & -1.0 \end{pmatrix} \right) \\ &= \tau_2 \left( (-1 \ 1 \ 1 \ -1) \right) = (-1 \ 1 \ 1 \ -1) \end{aligned}$$

네 개 샘플의 레이블과 같으므로 모두 맞음



## 4.7.3 다층 퍼셉트론의 동작

### • 출력 벡터 $O$ 의 모양

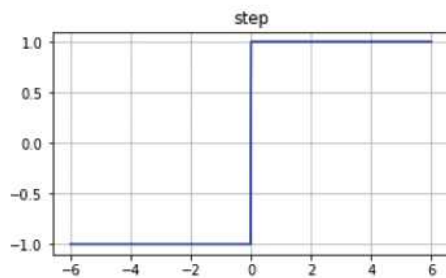
- [예제 4-3]은 출력 노드가 하나이고 샘플이 4개이므로  $O$ 는  $1 \times 4$  행렬
- 일반적으로 출력 노드가  $c$ 이고( $c$ 는 부류 개수) 샘플이  $n$ 개 이면  $O$ 는  $c \times n$  행렬
- 열은 샘플을 나타내며, 값이 가장 큰 인덱스를 최종 분류 결과로 출력
- 예) 숫자 인식

$$O = \begin{pmatrix} 0.1 & 0 & 0.9 & 0 & 0.78 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0.001 & 0 & 0 & 0.01 & 0 \\ 0.899 & 0 & 0 & 0 & 0.02 \\ 0 & 0 & 0 & 0.99 & 0 \\ 0 & 0.22 & 0.1 & 0 & 0.2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0.78 & 0 & 0 & 0 \end{pmatrix} \rightarrow \text{최종 분류 결과} = (4 \ 9 \ 0 \ 5 \ \dots \ 0)$$

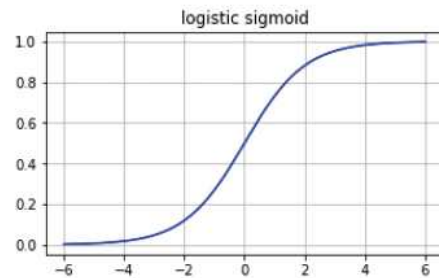
## 4.7.4 활성 함수

### • 다양한 활성 함수

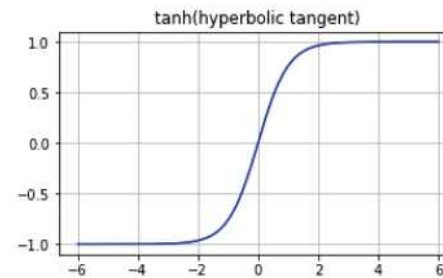
- 퍼셉트론은 계단 함수를 사용하는데, 예측 결과를 0~1 사이의 확률로 표현해야 하는 경우에 계단 함수는 부적절함
- 따라서 다층 퍼셉트론은 시그모이드, 딥러닝은 ReLU와 softmax를 주로 사용



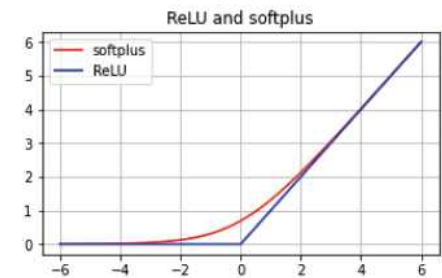
(a) 계단 함수



(b) 로지스틱 시그모이드



(c) 하이퍼볼릭 탄젠트 시그모이드



(d) 소프트플러스와 ReLU

그림 4-14 신경망이 사용하는 다양한 활성 함수

## 4.7.4 활성화 함수

표 4-1 신경망에서 사용하는 여러 가지 활성화 함수

### • 다양한 활성화 함수

함수 이름	함수	1차 도함수	범위
계단	$\tau(s) = \begin{cases} 1 & s \geq 0 \\ -1 & s < 0 \end{cases}$	$\tau'(s) = \begin{cases} 0 & s \neq 0 \\ \text{불가} & s = 0 \end{cases}$	-1과 1
로지스틱 시그모이드	$\tau(s) = \frac{1}{1 + e^{-as}}$	$\tau'(s) = a\tau(s)(1 - \tau(s))$	(0,1)
하이퍼볼릭 탄젠트 (tanh) 시그모이드	$\tau(s) = \frac{2}{1 + e^{-as}} - 1$	$\tau'(s) = \frac{a}{2}(1 - \tau(s)^2)$	(-1,1)
소프트플러스	$\tau(s) = \log_e(1 + e^s)$	$\tau'(s) = \frac{1}{1 + e^{-s}}$	(0, ∞)
렉티파이어(ReLU)	$\tau(s) = \max(0, s)$	$\tau'(s) = \begin{cases} 0 & s < 0 \\ 1 & s > 0 \\ \text{불가} & s = 0 \end{cases}$	(0, ∞)

### • 출력층에서 주로 사용하는 소프트맥스 함수(확률로 간주할 수 있음)

$$o_k = \frac{e^{s_k}}{\sum_{i=1,c} e^{s_i}} \quad (4.17) \quad \leftarrow o_1 + o_2 + \dots + o_c = 1.0 \text{을 만족함}$$

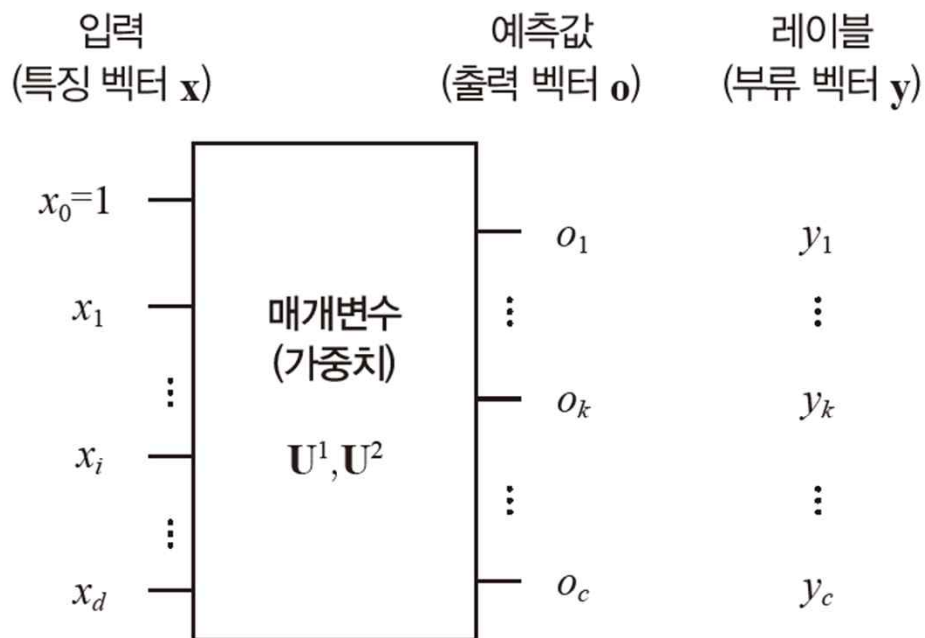
## 4.8 오류 역전파 알고리즘

- 다층 퍼셉트론은 오류 역전파 알고리즘으로 학습함
  - 은닉층이 있고 활성화 함수가 시그모이드이므로 퍼셉트론 학습 알고리즘보다 복잡하지만 기본 원리는 같음

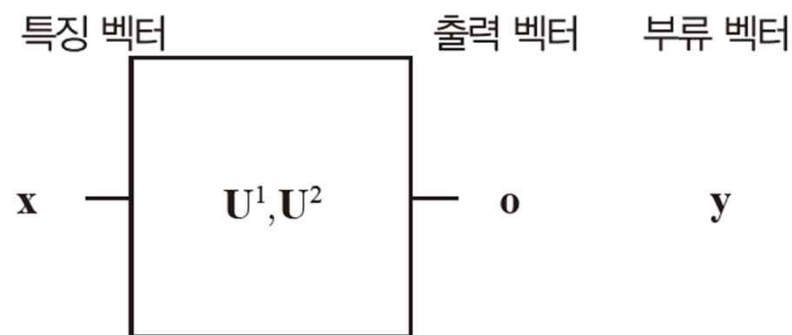
## 4.8.1 손실 함수 설계

### • 다층 퍼셉트론의 블록 다이어그램

- 신경망의 출력 벡터(예측값)  $\mathbf{o}$ 가 부류 벡터(참값)  $\mathbf{y}$ 와 같을수록 매개변수  $\mathbf{U}^1$ 과  $\mathbf{U}^2$ 는 데이터를 잘 인식. 다르면  $\mathbf{o}$ 와  $\mathbf{y}$ 가 가까워지도록  $\mathbf{U}^1$ 과  $\mathbf{U}^2$ 를 갱신해야 함. 오차가 클수록 갱신하는 양이 큼



(a) 입력과 출력, 기댓값의 관계



(b) 벡터 표현

그림 4-15 다층 퍼셉트론의 블록 다이어그램

## 4.8.1 손실 함수 설계

- 샘플 하나의 오차를 측정하는 손실 함수

- 보통  $y$ 는 원핫 코드로 표현. 예) 숫자 0이면  $y=(1,0,0,...,0)$ , 2면  $y=(0,0,1,0,...,0)$

$$J(\mathbf{U}^1, \mathbf{U}^2) = \sqrt{(y_1 - o_1)^2 + (y_2 - o_2)^2 + \dots + (y_c - o_c)^2} = \|\mathbf{y} - \mathbf{o}\| \quad (4.18)$$

- 식 (4.18)을 확장

- 계산 편의를 위해 제곱을 하여 제곱근 제거
- 미니 배치 단위로 처리(샘플의 오차를 평균)

평균제곱오차(MSE, mean squared error)

$$\begin{aligned} J(\mathbf{U}^1, \mathbf{U}^2) &= \frac{1}{|M|} \sum_{\mathbf{x} \in M} \|\mathbf{y} - \mathbf{o}\|^2 \\ &= \frac{1}{|M|} \sum_{\mathbf{x} \in M} \left\| \mathbf{y} - \tau_2 \left( \mathbf{U}^2 \tau_1 \left( \mathbf{U}^1 \mathbf{x}^T \right) \right) \right\|^2 \end{aligned} \quad (4.19)$$

## 4.8.2 학습 알고리즘

### • 가중치 갱신 규칙

- 퍼셉트론과 비슷
- $\frac{\partial J}{\partial u_{ji}^1}$ 와  $\frac{\partial J}{\partial u_{kj}^2}$ 를 구하는 과정은 생략

$$\begin{aligned} \mathbf{U}^2 &= \mathbf{U}^2 + \rho(-\nabla \mathbf{U}^2) & \mathbf{U}^1 &= \mathbf{U}^1 + \rho(-\nabla \mathbf{U}^1) \\ \nabla \mathbf{U}^2 &= \begin{pmatrix} \frac{\partial J}{\partial u_{10}^2} & \frac{\partial J}{\partial u_{11}^2} & \cdots & \frac{\partial J}{\partial u_{1p}^2} \\ \frac{\partial J}{\partial u_{20}^2} & \frac{\partial J}{\partial u_{21}^2} & \cdots & \frac{\partial J}{\partial u_{2p}^2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial u_{c0}^2} & \frac{\partial J}{\partial u_{c1}^2} & \cdots & \frac{\partial J}{\partial u_{cp}^2} \end{pmatrix} & \nabla \mathbf{U}^1 &= \begin{pmatrix} \frac{\partial J}{\partial u_{10}^1} & \frac{\partial J}{\partial u_{11}^1} & \cdots & \frac{\partial J}{\partial u_{1d}^1} \\ \frac{\partial J}{\partial u_{20}^1} & \frac{\partial J}{\partial u_{21}^1} & \cdots & \frac{\partial J}{\partial u_{2d}^1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial u_{p0}^1} & \frac{\partial J}{\partial u_{p1}^1} & \cdots & \frac{\partial J}{\partial u_{pd}^1} \end{pmatrix} \end{aligned} \quad (4.20)$$

**TIP** 미분값을 구하는 수식 유도는 이 책의 범위를 넘으므로 생략한다. 관심있는 독자는 『기계 학습(오일석, 한빛아카데미)』의 3.4절을 참조한다.

## 4.8.2 학습 알고리즘

### • 오류 역전파 error-backpropagation 학습 알고리즘

- 출력층에서 시작하여 역방향으로 오류를 전파한다는 뜻에서 오류 역전파라 부름

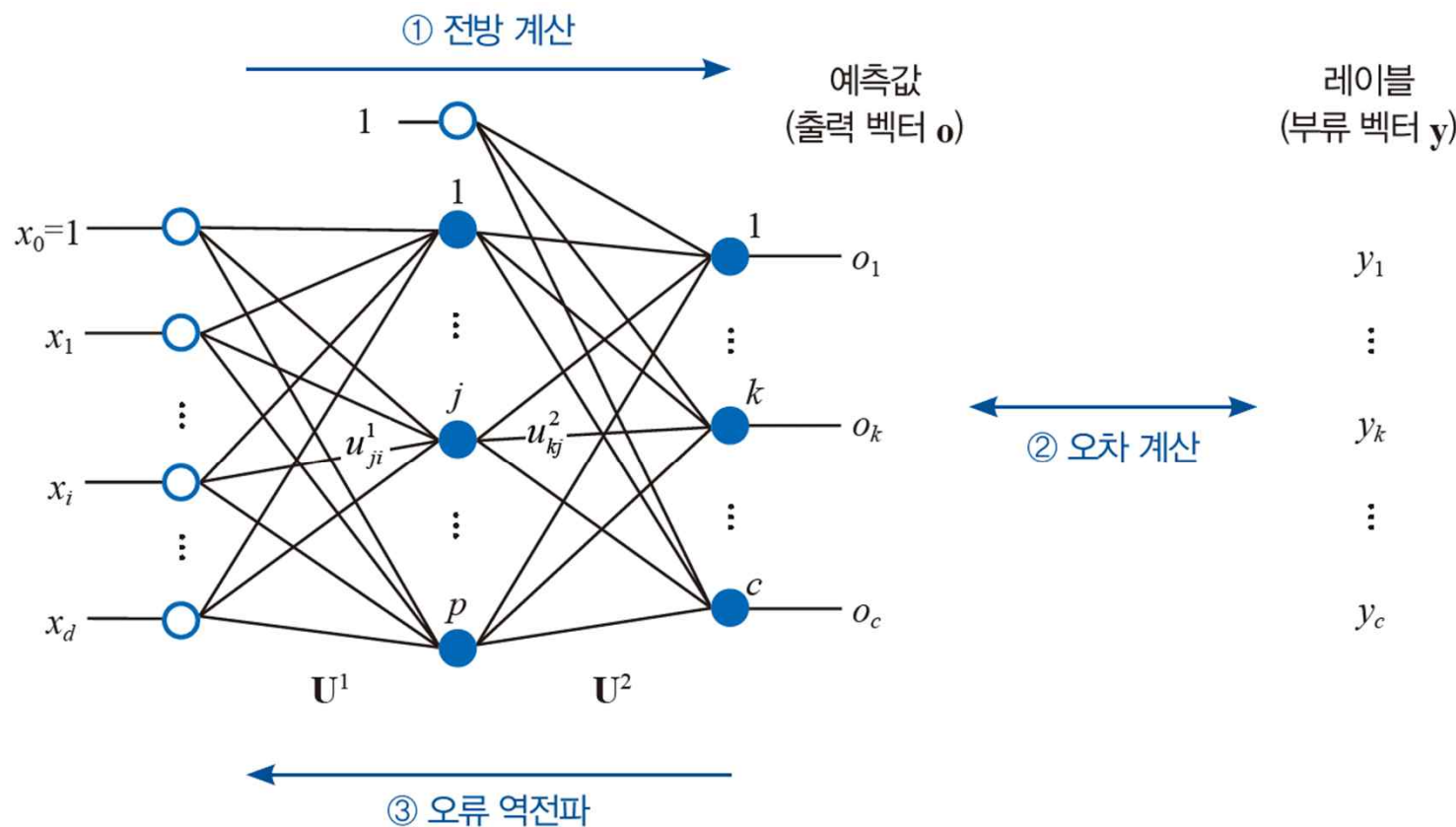


그림 4-16 오류 역전파 학습 알고리즘



## 4.9 다층 퍼셉트론 프로그래밍

- 다층 퍼셉트론

- 1980~1990년대 실용적인 시스템을 구현하는데 널리 사용
- 현재도 사용됨

## 4.9.1 sklearn의 필기 숫자 데이터셋

### • [프로그램 4-3]

프로그램 4-3

sklearn 필기 숫자 데이터에 다층 퍼셉트론 적용

```
01 from sklearn import datasets
02 from sklearn.neural_network import MLPClassifier
03 from sklearn.model_selection import train_test_split
04 import numpy as np
05
06 # 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
07 digit=datasets.load_digits()
08 x_train,x_test,y_train,y_test=train_test_split(digit.data,digit.target,train_
size=0.6)
09
10 # MLP 분류기 모델을 학습
11 mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001,batch_
size=32,max_iter=300,solver='sgd',verbose=True)
12 mlp.fit(x_train,y_train)
13
14 res=mlp.predict(x_test) # 테스트 집합으로 예측
15
16 # 혼동 행렬
17 conf=np.zeros((10,10))
18 for i in range(len(res)):
```

## 4.9.1 sklearn의 필기 숫자 데이터셋

```
19     conf[res[i]][y_test[i]]+=1
20     print(conf)
21
22     # 정확률 계산
23     no_correct=0
24     for i in range(10):
25         no_correct+=conf[i][i]
26     accuracy=no_correct/len(res)
27     print("테스트 집합에 대한 정확률은 ", accuracy*100, "%입니다.")
```

### NOTE 디자인 패턴 다시 생각하기

[프로그램 4-3]은 [프로그램 4-2]와 [프로그램 3-5]의 판박이다. 세 프로그램은 회색 음영으로 표시한 02행과 11행만 다르고 나머지는 똑같다. 기계 학습 모델이 퍼셉트론에서 다층 퍼셉트론으로 바뀌었을 뿐, 데이터 읽기, 모델 학습과 예측, 성능 측정 부분은 그대로이다. 4.6.3항의 디자인 패턴에 대한 설명과 [프로그램 4-2]에 표시한 디자인 패턴을 다시 살펴보기 바란다. 프로그램을 설명하는 본문에도 디자인 패턴을 적용할 수 있다. [프로그램 4-3]을 설명하는 부분은 일부러 [프로그램 4-2]의 설명을 최대한 그대로 사용했다.

## 4.9.1 sklearn의 필기 숫자 데이터셋

### • 실행 결과

- 정확률 97.2%로서 [프로그램 3-6]의 SVM(98.7%)보다 열등하고 [프로그램 4-2]의 퍼셉트론(93.8%)보다 우수

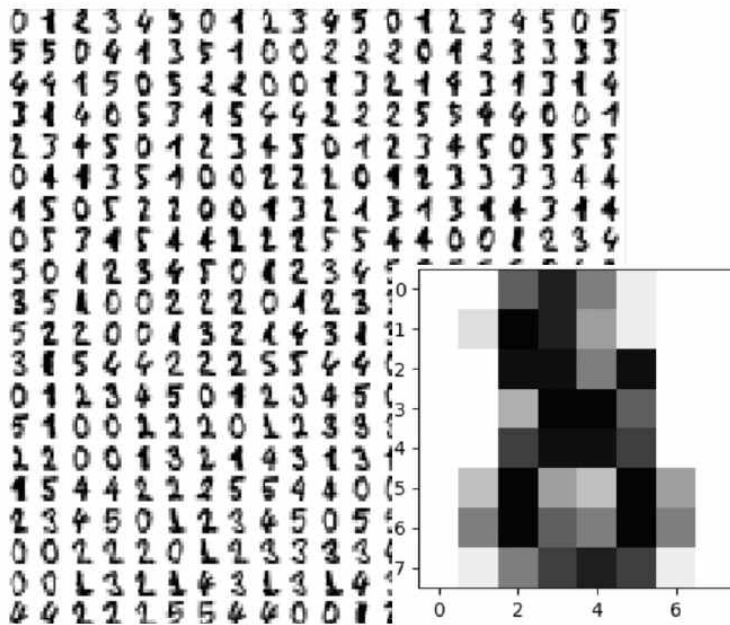
```
Iteration 1, loss = 1.93427517
Iteration 2, loss = 0.32451464
Iteration 3, loss = 0.20142691
Iteration 4, loss = 0.14313824
...
Iteration 40, loss = 0.01163957
Iteration 41, loss = 0.01127886
...
Iteration 80, loss = 0.00538125
Iteration 81, loss = 0.00538663
...
Iteration 107, loss = 0.00405861
Iteration 108, loss = 0.00402524
Iteration 109, loss = 0.00397349
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
```

```
[[73.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. 69.  1.  0.  1.  0.  1.  0.  1.  0.]
 [ 0.  0. 70.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. 67.  0.  0.  0.  0.  1.  3.]
 [ 1.  0.  0.  0. 77.  0.  1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0. 64.  1.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0. 67.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  1.  0. 83.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.  1. 58.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0. 71.]]
```

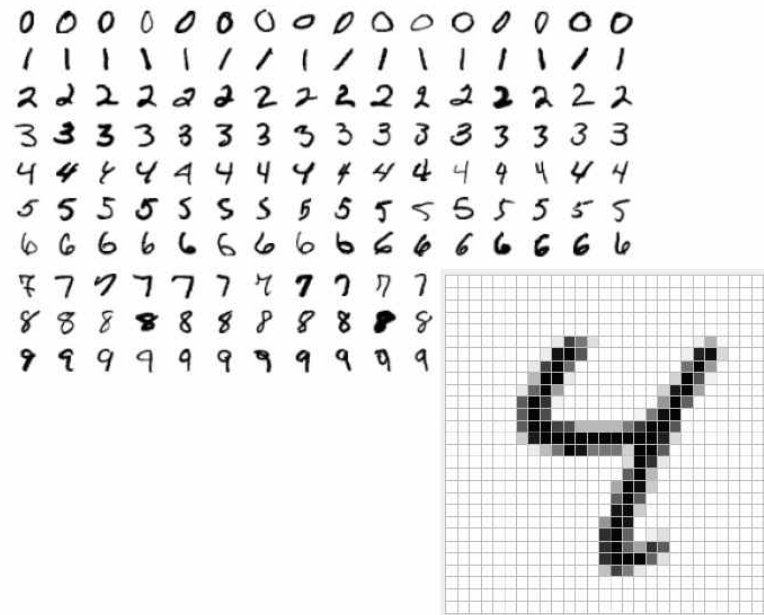
테스트 집합에 대한 정확률은 97.2183588317107%입니다.

## 4.9.2 MNIST 데이터셋으로 확장하기

- MNIST 필기 숫자 데이터셋([그림 3-6(b)])
  - 훈련 집합 60000자 + 테스트 집합 10000자
  - 샘플은 28\*28 맵으로 표현



(a) sklearn에서 제공하는 데이터셋



(b) MNIST 데이터셋

그림 3-6 필기 숫자 데이터셋



## 4.9.2 MNIST 데이터셋으로 확장하기

- [프로그램 4-4]

- [프로그램 4-3]과 매우 유사

09~10행: 앞 6만 개를 훈련,  
뒤 1만 자를 테스트로 분할

프로그램 4-4 MNIST 데이터셋을 다층 퍼셉트론으로 인식

```
01 from sklearn.datasets import fetch_openml
02 from sklearn.neural_network import MLPClassifier
03 import matplotlib.pyplot as plt
04 import numpy as np
05
06 # MNIST 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
07 mnist=fetch_openml('mnist_784')
08 mnist.data=mnist.data/255.0
09 x_train=mnist.data[:60000]; x_test=mnist.data[60000:]
10 y_train=np.int16(mnist.target[:60000]); y_test=np.int16(mnist.target[60000:])
11
12 # MLP 분류기 모델을 학습
13 mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001,batch_
    size=512,max_iter=300,solver='adam',verbose=True)
14 mlp.fit(x_train,y_train)
15
16 # 테스트 집합으로 예측
17 res=mlp.predict(x_test)
18
19 # 혼동 행렬
20 conf=np.zeros((10,10),dtype=np.int16)
21 for i in range(len(res)):
22     conf[res[i]][y_test[i]]+=1
23 print(conf)
24
25 # 정확률 계산
26 no_correct=0
27 for i in range(10):
28     no_correct+=conf[i][i]
29 accuracy=no_correct/len(res)
30 print("테스트 집합에 대한 정확률은", accuracy*100, "%입니다.")
```

8행: [0,255] 범위를 [0,1] 범위로 변환

## 4.9.2 MNIST 데이터셋으로 확장하기

- 실행 결과 97.78% 정확률을 얻음

```
Iteration 1, loss = 0.61803286
Iteration 2, loss = 0.26101832
Iteration 3, loss = 0.20624874
Iteration 4, loss = 0.17280300
...
Iteration 82, loss = 0.00074115
Iteration 83, loss = 0.00072676
Iteration 84, loss = 0.00067264
Iteration 85, loss = 0.00065032
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
```

```
[[ 970    0    4    0    0    1    4    0    4    1]
 [   0 1124    2    0    0    0    2    4    1    3]
 [   1    3 1002    6    3    0    1   10    4    0]
 [   0    1    6  988    2   10    1    1    6    5]
 [   1    0    2    2  964    2    4    3    7    9]
 [   1    1    0    3    0  863    3    0    4    3]
 [   2    2    2    0    2    7  942    0    0    1]
 [   1    1    5    5    3    2    0 1003    3    6]
 [   3    3    8    3    1    4    1    3  942    1]
 [   1    0    1    3    7    3    0    4    3  980]]
```

테스트 집합에 대한 정확률은 97.78%입니다.

## 4.10 하이퍼 매개변수 최적화

- 하이퍼 매개변수

- 모델의 구조와 모델의 학습 과정을 제어하는 역할. 예) 은닉 노드 개수, 식 (4.20)의 학습률, 미니 배치 크기 등
- [프로그램 4-3]의 경우 은닉 노드는 100개, 학습률은 0.001 ← 어떻게 결정했나?
- 이 절에서는 체계적으로 최적의 하이퍼 매개변수 값을 결정하는 방법을 공부



## 4.10.1 하이퍼 매개변수 살피기

- [프로그램 4-3] 11행의 MLPClassifier 함수는 6개의 매개변수를 가짐

```
mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001, batch_size=32,max_iter=300,solver='sgd',verbose=True)
```

- 앞의 5개는 하이퍼 매개변수
  - hidden\_layer\_sizes=(100): 100개 노드를 가진 은닉층 한 개를 둠(100개와 80개 노드를 가진 은닉층 두 개를 설정하려면 hidden\_layer\_sizes=(100,80)으로 함)
  - learning\_rate\_init=0.001: 식 (4.20)의 학습률  $\rho$ 를 0.001로 설정
  - batch\_size=32: 미니 배치 크기를 32로 설정
  - max\_iter=300: 최대 세대 수를 300으로 설정
  - solver='sgd': 최적화 알고리즘으로 스토캐스틱 경사 알고리즘을 사용
- MLPClassifier 함수의 매개변수는 6개가 전부인가?

## 4.10.1 하이퍼 매개변수 살피기

### • 함수의 API

- 예) MLPClassifier는 23개의 매개변수를 가짐
  - 파란색은 11행이 사용한 것들. 나머지는 기본값을 사용
    - 예) activation='relu'가 기본값이므로 활성 함수로 ReLU 사용
    - 예) shuffle=True가 기본값이므로 세대를 시작할 때마다 훈련 집합의 샘플 순서를 섞음

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100, ),
activation='relu', solver='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_
iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_
start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False,
validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_
no_change=10, max_fun=15000)
```

- 11행에서 max\_iter=300으로 설정했는데, 109 세대에서 학습을 멈춤
  - tol=0.0001과 n\_iter\_no\_change=10 때문(10세대 동안 손실 함수 감소량이 0.0001 이하이면 멈추라는 뜻)

## 4.10.1 하이퍼 매개변수 살피기

- 하이퍼 매개변수 설정 가이드라인
  - 논문이나 공식 문서를 따라 함
  - 라이브러리 함수가 제공하는 기본값 사용
  - 중요한 하이퍼 매개변수를 골라 최적화

- [프로그램 4-5]는 은닉 노드 개수를 최적화

프로그램 4-5

validation\_curve 함수로 최적의 은닉 노드 개수 찾기

```
01 from sklearn import datasets
02 from sklearn.neural_network import MLPClassifier
03 from sklearn.model_selection import train_test_split, validation_curve
04 import numpy as np
05 import matplotlib.pyplot as plt
06 import time
07
08 # 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
09 digit=datasets.load_digits()
10 x_train,x_test,y_train,y_test=train_test_split(digit.data,digit.target,train_size=0.6)
11
```

훈련:테스트를 6:4로 분할

## 4.10.2 단일 하이퍼 매개변수 최적화: validation\_curve 함수 이용

```
12 # 다층 퍼셉트론을 교차 검증으로 성능 평가 (소요 시간 측정 포함)
13 start=time.time() # 시작 시각
14 mlp=MLPClassifier(learning_rate_init=0.001,batch_size=32,max_iter=300,solver='sgd')
15 prange=range(50,1001,50)
16 train_score,test_score=validation_curve(mlp,x_train,y_train,param_name="hidden_
    layer_sizes",param_range=prange,cv=10,scoring="accuracy",n_jobs=4)
17 end=time.time() # 끝난 시각
18 print("하이퍼 매개변수 최적화에 걸린 시간은",end-start,"초입니다.")
19
20 # 교차 검증 결과의 평균과 분산 구하기
21 train_mean = np.mean(train_score,axis=1)
22 train_std = np.std(train_score,axis=1)
23 test_mean = np.mean(test_score,axis=1)
24 test_std = np.std(test_score,axis=1)
25
26 # 성능 그래프 그리기
27 plt.plot(prange,train_mean,label="Train score",color="r")
28 plt.plot(prange,test_mean,label="Test score",color="b")
29 plt.fill_between(prange,train_mean-train_std,train_mean+train_std,alpha=0.2,color="r")
30 plt.fill_between(prange,test_mean-test_std,test_mean+test_std,alpha=0.2,color="b")
31 plt.legend(loc="best")
32 plt.title("Validation Curve with MLP")
33 plt.xlabel("Number of hidden nodes"); plt.ylabel("Accuracy")
34 plt.ylim(0.9,1.01)
35 plt.grid(axis='both')
36 plt.show()
```

50에서 시작하여 50씩 증가시키면서 1000까지 조사

10-겹 교차 검증으로 성능 측정

코어 4개를 사용하여 병렬 처리

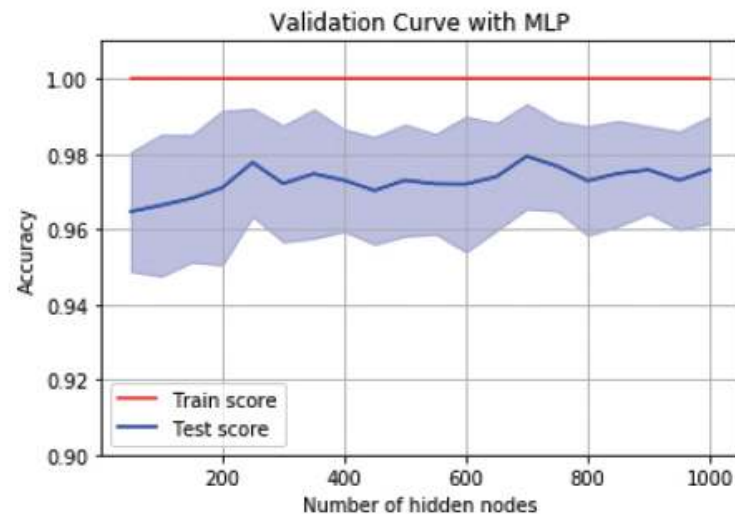


## 4.10.2 단일 하이퍼 매개변수 최적화: validation\_curve 함수 이용

```
37
38 best_number_nodes=prange[np.argmax(test_mean)]    # 최적의 은닉 노드 개수
39 print("\n최적의 은닉층의 노드 개수는",best_number_nodes,"개입니다.\n")
40
41 # 최적의 은닉 노드 개수로 모델링
42 mlp_test=MLPClassifier(hidden_layer_sizes=(best_number_nodes),learning_rate_
    init=0.001,batch_size=32,max_iter=300,solver='sgd')
43 mlp_test.fit(x_train,y_train)
44
45 # 테스트 집합으로 예측
46 res=mlp_test.predict(x_test)
47
48 # 혼동 행렬
49 conf=np.zeros((10,10))
50 for i in range(len(res)):
51     conf[res[i]][y_test[i]]+=1
52 print(conf)
53
54 # 정확률 계산
55 no_correct=0
56 for i in range(10):
57     no_correct+=conf[i][i]
58 accuracy=no_correct/len(res)
59 print("테스트 집합에 대한 정확률은", accuracy*100, "%입니다.")
```

### • 실행 결과

하이퍼 매개변수 최적화에 걸린 시간은 433.8679416179657초입니다.



최적의 은닉층의 노드 개수는 700개입니다.

```
[[65. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  
 [0. 71. 0. 0. 0. 1. 0. 0. 3. 1.]  
 [0. 0. 67. 0. 0. 1. 0. 0. 0. 0.]  
 [0. 0. 0. 64. 0. 0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 78. 0. 0. 0. 0. 0.]  
 [2. 0. 0. 0. 0. 80. 0. 0. 2. 0.]  
 [0. 0. 0. 0. 0. 1. 78. 0. 0. 0.]  
 [0. 0. 0. 0. 0. 0. 0. 64. 0. 0.]  
 [0. 0. 0. 1. 0. 0. 0. 0. 74. 1.]  
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 64.]]
```

테스트 집합에 대한 정확률은 98.0528511821975%입니다.

### NOTE 병렬 처리를 이용해 학습 시간 단축하기

앞으로는 데이터가 크거나 [프로그램 4-5]처럼 하이퍼 매개변수 최적화를 위해 학습을 여러 번 수행하는 등의 요인으로 시간이 많이 걸리는 프로그래밍을 주로 한다. 따라서 컴퓨터의 병렬 처리 기능을 활용해 실행 시간을 줄이는 데 신경을 써야 한다.

현대 컴퓨터는 병렬 처리를 하는 능력을 갖추고 있다. GPU 보드를 별도로 구입해 PC에 장착하면 수십 배 빠르게 프로그램을 실행할 수 있다. 그런데 GPU를 장착하지 않고도 병렬 처리를 활용할 수 있다. 노트북과 스마트폰을 포함한 오늘날의 컴퓨터는 코어라는 단위 프로세서를 여러 개 가진 멀티 코어 마이크로프로세서를 CPU로 사용한다. 컴퓨터에 쿼드 코어 마이크로프로세서가 장착되어 있다면 4개의 작업을 병렬로 실행할 수 있다. 16행에 있는 validation\_curve 함수의 마지막 매개변수인 n\_jobs=4는 작업 4개를 동시에 병렬로 실행하라는 설정이다. 일례로 노트북에서 n\_jobs=1, n\_jobs=2, n\_jobs=4로 설정해 비교한 결과 각각 836초, 582초, 436초가 걸렸다. 하나씩 작업했을 때에 비해 4개를 병렬로 실행함으로써 속도를 두 배 높였음을 알 수 있다.