



# 딥러닝 시작



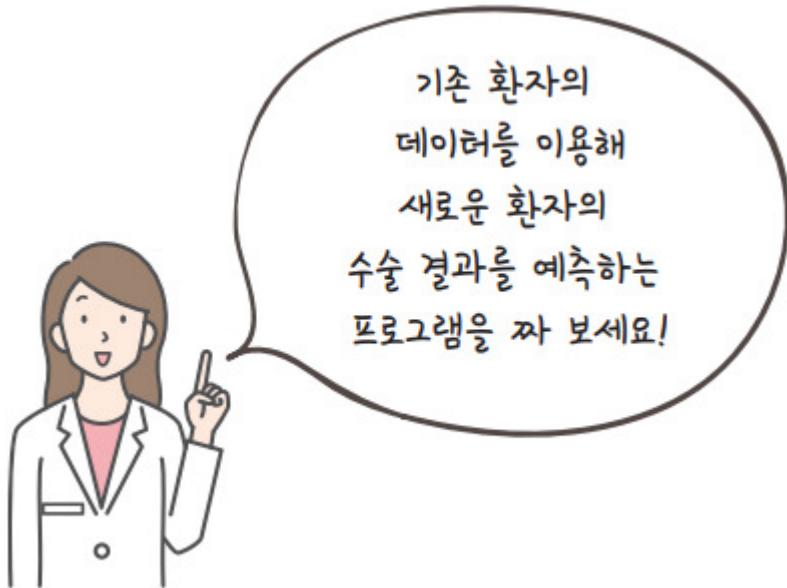
# 1 미지의 일을 예측하는 원리

---



# 1 미지의 일을 예측하는 원리

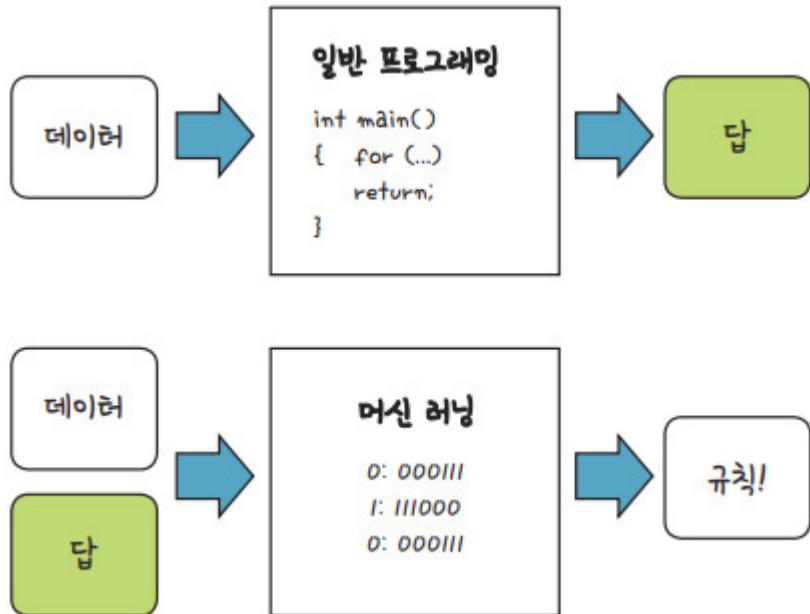
## ● 미지의 일을 예측하는 원리





# 1 미지의 일을 예측하는 원리

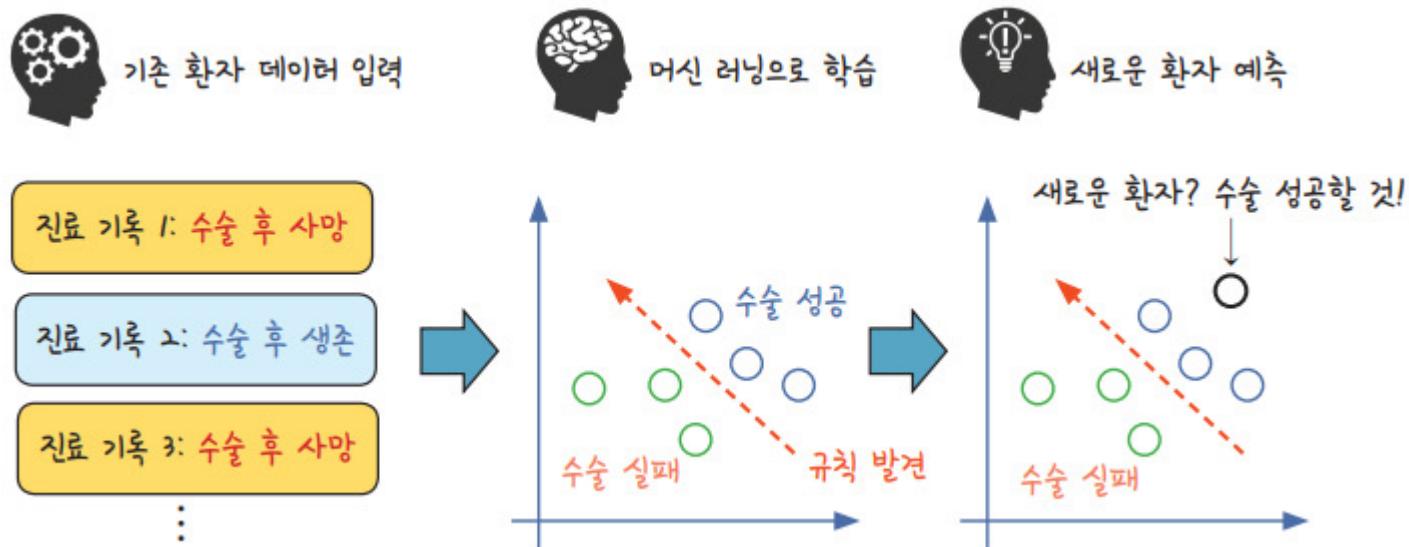
▼ 그림 2-1 | 머신 러닝과 일반 프로그래밍 비교





# 1 미지의 일을 예측하는 원리

## ▼ 그림 2-2 | 머신 러닝의 학습 및 예측 과정





# 1 미지의 일을 예측하는 원리

## ● 미지의 일을 예측하는 원리

- 배우려는 것이 바로 이러한 학습과 예측의 구체적인 과정
- 머신 러닝의 예측 성공률은 결국 얼마나 정확한 경계선을 긋느냐에 달려 있음
- 더 정확한 선을 긋기 위한 여러 가지 노력이 계속되어 왔고, 그 결과 퍼셉트론(perceptron), 아달라인(adaline), 선형 회귀(linear regression) 등을 지나 오늘날 딥러닝이 탄생



## 2 딥러닝 코드 실행해 보기

---



## 2 딥러닝 코드 실행해 보기

- 딥러닝 코드 실행해 보기

### 1. 환경 준비

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
```

### 2. 데이터 준비

```
Data_set = np.loadtxt("./data/ThoracicSurgery3.csv", delimiter=',')
X = Data_set[:,0:16]
y = Data_set[:,16]
```

### 3. 구조 결정

```
model = Sequential()
model.add(Dense(30, input_dim=16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```



## 2 딥러닝 코드 실행해 보기

### ● 딥러닝 코드 실행해 보기

#### 4. 모델 실행

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])  
history=model.fit(X, y, epochs=5, batch_size=16)
```

```
Epoch 1/5  
30/30 [=====] - 1s 2ms/step - loss: 1.5790 - accuracy: 0.4957  
Epoch 2/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4954 - accuracy: 0.8511  
Epoch 3/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4470 - accuracy: 0.8511  
Epoch 4/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4369 - accuracy: 0.8511  
Epoch 5/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4368 - accuracy: 0.8511
```



# 2 딥러닝 코드 실행해 보기

## ▼ 1. 환경 준비

[1] `from tensorflow.keras.models import Sequential # 텐서플로의 클래스 API에서 필요한 함수들을 불러옵니다.  
from tensorflow.keras.layers import Dense # 데이터를 다루는데 필요한 라이브러리를 불러옵니다.  
import numpy as np`

## ▼ 2. 데이터 준비

[2] `!git clone https://github.com/taehojo/data.git # 깃허브에 준비된 데이터를 가져옵니다.  
Data_set = np.loadtxt("./data/ThoracicSurgery3.csv", delimiter=',') # 수술 환자 데이터를 불러옵니다.  
X = Data_set[:,0:16] # 환자의 진찰 기록을 X로 지정합니다.  
y = Data_set[:,16] # 수술 후 사망/생존 여부를 y로 지정합니다.`

① 실행 시간

## ▼ 3. 구조 결정

[3] `model = Sequential() # 딥러닝 모델의 구조를 결정합니다.  
model.add(Dense(30, input_dim=16, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))`

## ▼ 4. 모델 실행

[4] `model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) # 딥러닝 모델을 실행합니다.  
history=model.fit(X, y, epochs=5, batch_size=16)`

```
Epoch 1/5  
30/30 [=====] - 1s 2ms/step - loss: 1.5790 - accuracy: 0.4957  
Epoch 2/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4954 - accuracy: 0.8511  
Epoch 3/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4470 - accuracy: 0.8511  
Epoch 4/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4369 - accuracy: 0.8511  
Epoch 5/5  
30/30 [=====] - 0s 2ms/step - loss: 0.4368 - accuracy: 0.8511
```

② 실행 결과



## 2 딥러닝 코드 실행해 보기

### ● 딥러닝 코드 실행해 보기

- 실행 결과는 매번 실행할 때마다 미세하게 달라짐
- 이것은 첫 가중치를 랜덤하게 정하고 실행을 반복하며, 조금씩 가중치를 수정해 가는 딥러닝의 특성 때문



## 3 딥러닝 개괄하기

---



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 폐암 수술 환자의 수술 1년 후 생존율을 예측한 모델

### 1. 환경 준비

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense  
import numpy as np
```

딥러닝을 구동하거나 데이터를 다루는 데 필요한 라이브러리들을 불러옵니다.



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- `from tensorflow.keras.models import Sequential`은 텐서플로(tensorflow)의 케라스(keras)라는 API에 있는 모델(model) 클래스로부터 Sequential() 함수를 불러오라는 의미
- `from tensorflow.keras.layers import Dense`는 케라스 API의 레이어(layers) 클래스에서 Dense()라는 함수를 불러오라는 의미
- 불러온 라이브러리명이 길거나 같은 이름이 이미 있을 경우 다음과 같이 짧게 줄일 수도 있음

```
import (라이브러리명) as (새로운 이름)
```

- `import numpy as np` 명령은 아나콘다에 이미 포함되어 있는 넘파이(numpy) 라이브러리를 np라는 짧은 이름으로 불러와 사용할 수 있게 해 줌



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 폐암 수술 환자의 수술 1년 후 생존율을 예측한 모델

### 2. 데이터 준비

```
Data_set = np.loadtxt("./data/ThoracicSurgery3.csv", delimiter=",")  
X = Data_set[:,0:16]  
y = Data_set[:,16]
```

준비된 수술 환자 정보 데이터를 나의 구글 코랩 계정에 저장합니다. 해당 파일을 불러와 환자 상태의 기록에 해당하는 부분을 X로, 수술 1년 후 사망/생존 여부를 y로 지정합니다.



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- data 폴더 안에 있는 데이터들은 ./data/데이터명 형식으로 불러올 수 있음
- 넘파이 라이브러리를 이용해 data 폴더에 있는 csv 파일을 불러오는 부분은 다음과 같음

```
data_set = np.loadtxt("./data/ThoracicSurgery3.csv", delimiter=",")
```

- 머신 러닝에서 알고리즘이나 좋은 컴퓨터 환경만큼 중요한 것이 바로 좋은 데이터를 준비하는 일
- 데이터를 면밀히 관찰하고 효율적으로 다루는 연습을 하는 것이 중요



## 3 딥러닝 개괄하기

## ▼ 폐암 수술 환자의 의료 기록과 1년 후 사망 여부 데이터

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	2,88	2,16	1	0	0	0	1	1	3	0	0	0	1	0	60	0
2	2	3,4	1,88	0	0	0	0	0	0	1	0	0	0	1	0	51	0
3	2	2,76	2,08	1	0	0	0	1	0	0	0	0	0	1	0	59	0
4	2	3,68	3,04	0	0	0	0	0	0	0	0	0	0	0	0	54	0
5	2	2,44	0,96	2	0	1	0	1	1	0	0	0	0	1	0	73	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
470	2	4,72	3,56	0	0	0	0	0	0	1	0	0	0	1	0	51	0

샘플 수  
(환자 수: 470명)



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 가로줄 한 행이 한 사람의 환자로부터 기록된 정보를 의미
- 총 470행이므로 환자 470명에 대한 정보
- 한 행에는 17개의 숫자가 들어 있음
- 이는 환자마다 17개의 정보를 순서에 맞추어 정리했다는 의미
- 앞의 정보 16개는 종양의 유형, 폐활량, 호흡 곤란 여부, 고통 정도, 기침, 흡연, 천식 여부 등 16가지 환자 상태를 조사해서 기록해 놓은 것
- 마지막 17번째 정보는 수술 1년 후의 생존 결과
- 1은 수술 후 생존했음을, 0은 수술 후 사망했음을 의미



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 이 프로젝트의 목적은 1번째 항목부터 16번째 항목까지 이용해서 17번째 항목, 즉 수술 1년 후의 생존 또는 사망을 맞히는 것
- 1번째 항목부터 16번째 항목까지 속성(attribute)이라 하고, 정답에 해당하는 17번째 항목을 클래스(class)라고 함
- 클래스는 앞서 이야기한 ‘이름표’에 해당
- 딥러닝을 위해서는 속성과 클래스를 서로 다른 데이터셋으로 지정해 주어야 함



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 먼저 속성으로 이루어진 데이터셋을 X라는 이름으로 만들어 줌

```
X = Data_set[:, 0:16]
```



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 다음으로 17번째 줄에 위치한 클래스를 따로 모아 데이터셋  $y$ 로 지정

```
y = Data_set[:, 16]
```

- 보통 집합은 대문자로, 원소는 소문자로 표시
- X에는 여러 개의 속성이 담기기 때문에 대문자 X로, y는 클래스 하나의 원소만 담기기 때문에 소문자 y로 썼음



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 폐암 수술 환자의 수술 1년 후 생존율을 예측한 모델

### 3. 구조 결정

```
model = Sequential()  
model.add(Dense(30, input_dim=16, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))
```

딥러닝 모델의 구조를 결정합니다. 여기에 설정된 대로 딥러닝을 수행합니다.



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

### 3. 구조 결정 어떤 딥러닝 구조를 만들 것인가

- 딥러닝을 실행시키기 위해 텐서플로를 불러왔음
- 텐서플로는 구글에서 만든 딥러닝 전용 라이브러리
- 텐서플로를 이용하면 여러 가지 알고리즘을 활용해 다양한 딥러닝 작업을 할 수 있지만, 사용법이 쉽지 않다는 단점이 있음

▼ 텐서플로(<https://www.tensorflow.org>)





### 3 딥러닝 개괄하기

- 딥러닝 개괄하기

- 이를 해결해 주기 위해 개발된 것이 **케라스(Keras)**

- ▼ 케라스(<https://keras.io>)



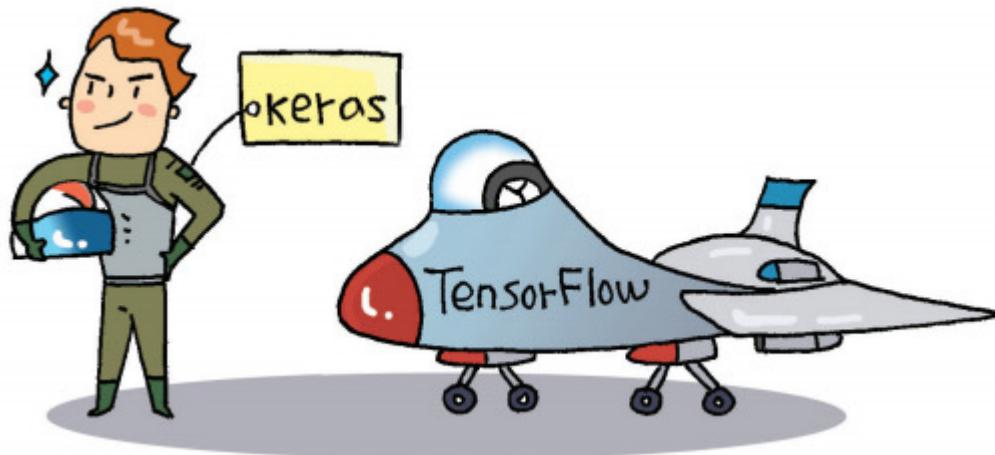


# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 텐서플로가 목적지까지 이동시켜 주는 비행기라면 케라스는 조종사에 해당
- 케라스를 활용하면 딥러닝의 거의 모든 작업을 쉽게 처리할 수 있음

## ▼ 텐서플로와 케라스의 관계





# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 앞의 예제에서 케라스의 활용 방법

```
model = Sequential() ..... ①
model.add(Dense(30, input_dim=16, activation='relu')) ..... ②
model.add(Dense(1, activation='sigmoid')) ..... ③
```



# 3 딥러닝 개괄하기

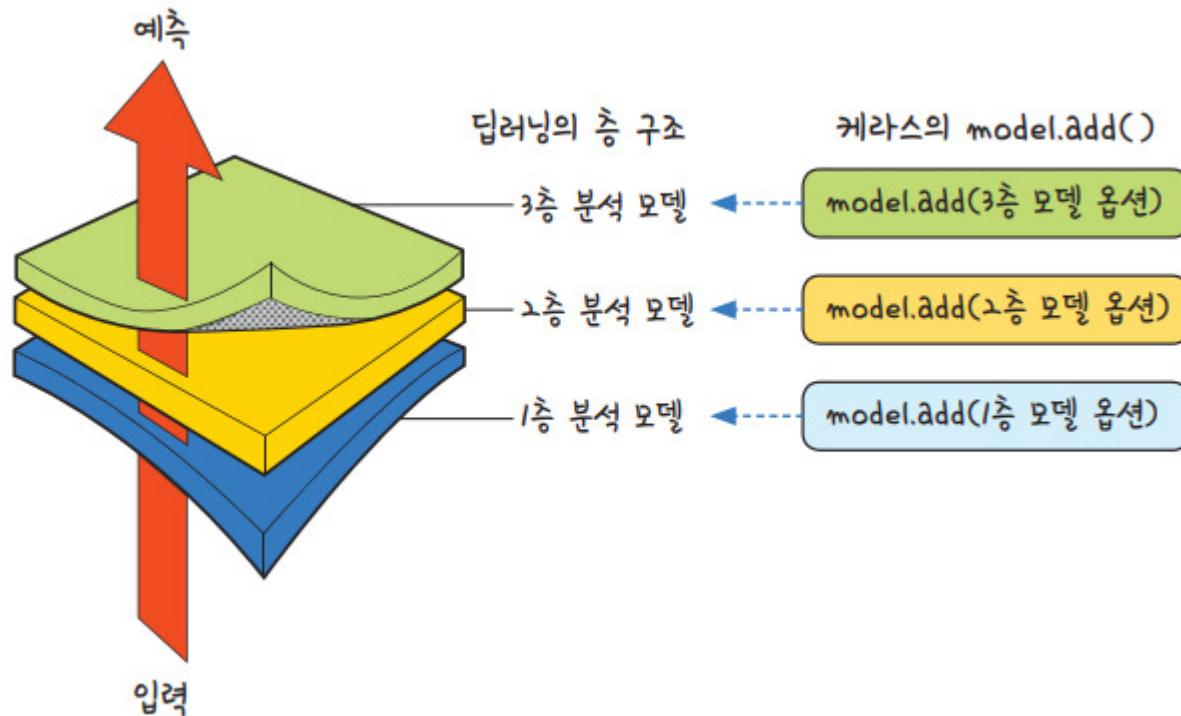
## ● 딥러닝 개괄하기

- ① 먼저 앞서 불러온 Sequential() 함수를 model로 선언
- 케라스의 Sequential() 함수는 딥러닝의 한 층 한 층을 ②model.add()라는 함수를 사용해 간단히 추가시켜 줌
- 여기서는 ②와 ③, 두 개의 층을 쌓았음
- Model.add() 함수를 한 줄 추가하는 것으로 필요한 만큼 내부의 층을 만들 수 있음



# 3 딥러닝 개괄하기

## ▼ 딥러닝의 층 구조와 케라스





# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 각 model.add() 함수 안에는 케라스 API의 layers 클래스에서 불려온 Dense() 함수가 포함되어 있음
- Dense는 ‘밀집한, 빽빽한’이란 뜻으로, 여기서는 각 층의 입력과 출력을 촘촘하게 모두 연결하라는 것



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 폐암 수술 환자의 수술 1년 후 생존율을 예측한 모델

### 4. 모델 실행

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])  
history = model.fit(X, y, epochs=5, batch_size=16)
```

딥러닝 모델을 실행합니다. 앞서 설정된 구조대로 실행하고 결과를 출력합니다.



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- 꼭 알아야 할 두 가지
- 첫째, 좋은 딥러닝 모델을 만들려면 몇 개의 층으로 쌓아 올려야 하는가?
- 둘째, Dense 함수 안에 있는 숫자와 설정의 의미는 무엇이며, 어떻게 정해야 하는가?
- 딥러닝을 설계한다는 것은 결국 몇 개의 층을 어떻게 쌓을지, Dense 외에 어떤 층을 사용할지, 내부의 변수들을 어떻게 정해야 하는지 등에 대해 고민하는 것
- 대개 어떤 데이터를 가지고 무엇을 할 것인지에 따라 딥러닝의 설계가 결정
- 각 설정과 변수의 의미를 알고 이것을 자유롭게 구성할 수 있는지가 딥러닝을 잘 다루는지 여부를 결정하는 것
- Dense() 함수의 내부에 쓰인 각 설정의 의미들은 앞으로 하나씩 배우게 될 것



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

### 4. 모델 실행 만든 딥러닝을 실행시키고 결과 확인

- 만들어 놓은 모델을 실행시키는 부분

```
model.compile(loss='binary_crossentropy', optimizer='adam',  
metrics=['accuracy']) .....❶  
history = model.fit(X, y, epochs=5, batch_size=16) .....❷
```



# 3 딥러닝 개괄하기

## ● 딥러닝 개괄하기

- `model.compile()` 함수는 앞서 만든 `model`의 설정을 그대로 실행하라는 의미
- 함수 내부에 `loss`, `optimizer`, `metrics` 등 키워드들이 들어 있음
- 이것은 앞 단계에서 만들어진 딥러닝 구조를 어떤 방식으로 구동시키고 어떻게 마무리할 것인지에 관련된 옵션들
- 딥러닝은 여러 층이 쌓여 만들어지며, 딥러닝의 기본 방식은 이 층들을 한 번만 통과하는 것이 아니라 위아래로 여러 차례 오가며 최적의 모델을 찾는 것
- 몇 번을 오갈 것인지, 그리고 한 번 오갈 때 몇 개의 데이터를 사용할 것인지 정하는 함수가 `model.fit()` 함수



# 예측 모델의 기본 원리

# 가장 훌륭한 예측선

---

- 1 선형 회귀의 정의
- 2 가장 훌륭한 예측선이란?
- 3 최소 제곱법
- 4 파이썬 코딩으로 확인하는 최소 제곱
- 5 평균 제곱 오차
- 6 파이썬 코딩으로 확인하는 평균 제곱 오차



# 가장 훌륭한 예측선

## ● 가장 훌륭한 예측선

- 딥러닝은 자그마한 통계의 결과들이 무수히 얹히고 쌓여 이루어지는 복잡한 연산의 결정체
- 딥러닝을 이해하려면 딥러닝의 가장 말단에서 이루어지는 기본적인 두 가지 계산 원리인 **선형 회귀**와 **로지스틱 회귀** 알아야 함
- ‘가장 훌륭한 예측선’이라는 표현은 ‘선형 회귀(linear regression) 분석을 이용한 모델’의 의미를 쉽게 풀어서 표현한 것



# 1 선형 회귀의 정의

---



# 1 선형 회귀의 정의

## ● 선형 회귀의 정의

“학생들의 중간고사 성적이 다 다르다.”

네, 다르겠죠.

그런데 위 문장이 나타낼 수 있는 정보는 너무 제한적입니다. 학급의 학생마다 제각각 성적이 다르다는 당연한 사실 외에는 알 수 있는 것이 없습니다. 이번에는 다음 문장을 보겠습니다.

“학생들의 중간고사 성적이 [      ]에 따라 다 다르다.”



# 1 선형 회귀의 정의

## ● 선형 회귀의 정의

- 여기서 [ ]에 들어갈 내용을 ‘정보’라고 함
- 머신 러닝과 딥러닝은 이 정보가 필요함
- 정보를 정확히 준비해 놓기만 하면 성적을 예측하는 방정식을 만들 수도 있음
- 성적을 변하게 하는 ‘정보’ 요소를  $x$ 라고 하고, 이  $x$  값에 따라 변하는 ‘성적’을  $y$ 라고 하자
- 이를 정의하면 ‘ $x$  값이 변함에 따라  $y$  값도 변한다’가 됨
- 이 정의 안에서 독립적으로 변할 수 있는 값  $x$ 를 **독립 변수**라고 함
- 또한, 이 독립 변수에 따라 종속적으로 변하는  $y$ 를 **종속 변수**라고 함



# 1 선형 회귀의 정의

## ● 선형 회귀의 정의

- 독립 변수가  $x$  하나뿐이어서 이것만으로 정확히 설명할 수 없을 때는  $x_1, x_2, x_3$  등  $x$  값을 여러 개 준비해 놓을 수도 있음
- 하나의  $x$  값만으로도  $y$  값을 설명할 수 있다면 단순 선형 회귀(simple linear regression)라고 함
- 또한,  $x$  값이 여러 개 필요하다면 다중 선형 회귀(multiple linear regression)라고 함



## 2 가장 훌륭한 예측선이란?

---



## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 독립 변수가 하나뿐인 단순 선형 회귀의 예
- 성적을 결정하는 여러 요소 중에 ‘공부한 시간’ 한 가지만 놓고 생각해 보자
- 중간고사를 본 4명의 학생에게 각각 공부한 시간을 물어보고 이들의 중간고사 성적을 아래와 같이 정리했다고 하자

### ▼ 공부한 시간과 중간고사 성적 데이터

공부한 시간	2시간	4시간	6시간	8시간
성적	81점	93점	91점	97점



## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 여기서 공부한 시간을  $x$ 라고 하고 성적을  $y$ 라고 할 때, 집합  $x$ 와 집합  $y$ 를 다음과 같이 표현할 수 있음

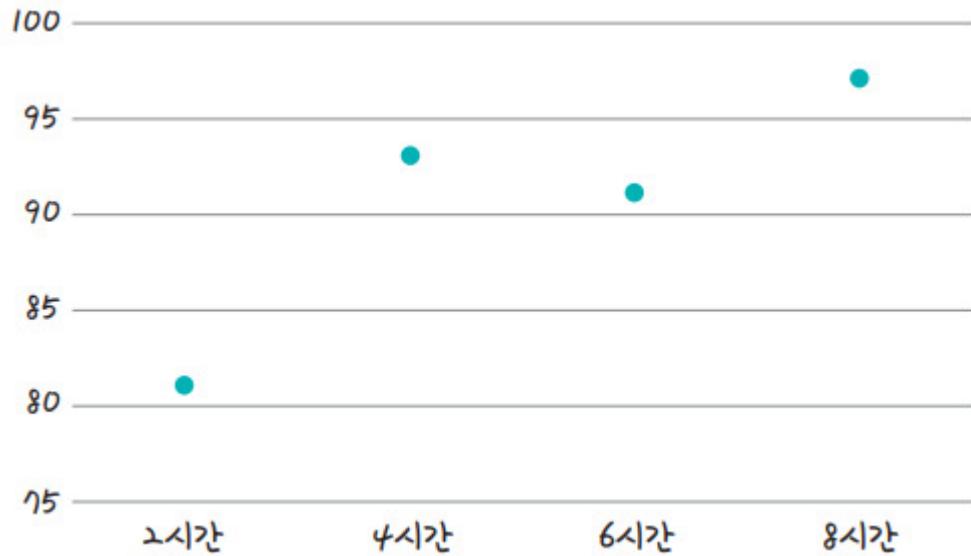
$$X = \{2, 4, 6, 8\}$$

$$Y = \{81, 93, 91, 97\}$$



## 2 가장 훌륭한 예측선이란?

▼ 그림 4-1 | 공부한 시간과 성적을 좌표로 표현





## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 좌표 평면에 나타내 놓고 보니, 왼쪽이 아래로 향하고 오른쪽이 위를 향하는 일종의 ‘선형(선으로 표시될 만한 형태)’을 보임
- 선형 회귀를 공부하는 과정은 이 점들의 특징을 가장 잘 나타내는 선을 그리는 과정과 일치
- 이 데이터에서 주어진 점들의 특징을 담은 선은 직선이므로 곧 일차 함수 그래프
- 일차 함수 그래프는 다음과 같은 식으로 표현할 수 있음

$$y = ax + b$$



## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 여기서  $x$  값은 독립 변수이고  $y$  값은 종속 변수
- 즉,  $x$  값에 따라  $y$  값은 반드시 달라짐
- 다만, 정확하게 계산하려면 상수  $a$ 와  $b$ 의 값을 알아야 함
- 이 직선을 훌륭하게 그으려면 직선의 기울기  $a$  값과  $y$  절편  $b$  값을 정확히 예측해 내야 함
- 지금 주어진 데이터에서의 선형 회귀는 결국 최적의  $a$  값과  $b$  값을 찾아내는 작업이라고 할 수 있음



## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 선을 잘 긋는 것이 어째서 중요할까?
- 잘 그어진 선을 통해 우리는 표 4-1의 공부한 시간과 중간고사 성적 데이터에 들어 있지 않은 여러 가지 내용을 유추할 수 있기 때문임
- 예를 들어 앞의 표에 나와 있지 않은 또 다른 학생의 성적을 예측하고 싶다고 하자
- 이때 정확한 직선을 그어 놓았다면 이 학생이 몇 시간을 공부했는지만 물어보면 됨
- 정확한  $a$ 값과  $b$  값을 따라 움직이는 직선에 학생이 공부한 시간인  $x$  값을 대입하면 예측 성적인  $y$  값을 구할 수 있는 것



## 2 가장 훌륭한 예측선이란?

### ● 가장 훌륭한 예측선이란?

- 딥러닝을 포함한 머신 러닝의 예측은 결국 이러한 기본 접근 방식과 크게 다르지 않음
- 기존 데이터(정보)를 가지고 어떤 선이 그려질지 예측한 후, 아직 답이 나오지 않은 그 무언가를 그 선에 대입해 보는 것
- 선형 회귀의 개념을 이해하는 것은 딥러닝을 이해하는 데 중요한 첫걸음



## 3 최소 제곱법

---



# 3 최소 제곱법

## ● 최소 제곱법

- 우리 목표의 가장 정확한 선을 그는 것
- 더 구체적으로는 정확한 기울기  $a$ 와 정확한  $y$  절편  $b$ 를 알아내면 된다고 했음
- 최소 제곱법**(method of least squares)이라는 공식을 알고 적용한다면, 이를 통해 일차 함수의 기울기  $a$ 와  $y$  절편  $b$ 를 바로 구할 수 있음
- 지금 가진 정보가  $x$  값(입력 값, 여기서는 ‘공부한 시간’)과  $y$  값(출력 값, 여기서는 ‘성적’)일 때 이를 이용해 기울기  $a$ 를 구하는 방법은 다음과 같음

$$a = \frac{(x - x \text{ 평균})(y - y \text{ 평균}) \text{의 합}}{(x - x \text{ 평균})^2 \text{의 합}} \quad (\text{식 4.1})$$



# 3 최소 제곱법

## ● 최소 제곱법

- 쉽게 풀어서 다시 쓰면  $x$ 의 편차(각 값과 평균과의 차이)를 제곱해서 합한 값을 분모로 놓고,  $x$ 와  $y$ 의 편차를 곱해서 합한 값을 분자로 놓으면 기울기가 나온다는 의미
- 실제로 우리가 가진  $y$ (성적) 값과  $x$ (공부한 시간) 값을 이 식에 대입해 보자
- 먼저  $x$  값의 평균과  $y$  값의 평균을 구해 보면 다음과 같음
  - 공부한 시간( $x$ ) 평균:  $(2 + 4 + 6 + 8) \div 4 = 5$
  - 성적( $y$ ) 평균:  $(81 + 93 + 91 + 97) \div 4 = 90.5$



# 3 최소 제곱법

## ● 최소 제곱법

- 기울기  $a$ 는 2.30이 나옴!
- 다음은  $y$  절편인  $b$ 를 구하는 공식

$$b = y \text{의 평균} - (x \text{의 평균} \times \text{기울기 } a) \quad (\text{식 4.2})$$

- 즉,  $y$ 의 평균에서  $x$ 의 평균과 기울기의 곱을 빼면  $b$  값이 나온다는 의미



# 3 최소 제곱법

## ● 최소 제곱법

- 우리는 이미  $y$  평균,  $x$  평균, 그리고 조금 전 구한 기울기  $x$ 까지 이 식을 풀기 위해 필요한 모든 변수를 알고 있음
- 이를 식에 대입해 보자

$$b = 90.5 - (2.3 \times 5)$$

$$= 79$$

- $y$  절편  $b$ 는 79가 나왔음
- 이제 다음과 같이 예측 값을 구하기 위한 직선의 방정식이 완성

$$y = 2.3x + 79$$



# 3 최소 제곱법

## ● 최소 제곱법

- 구한 식에 데이터를 대입해 보자
- $x$ 를 대입했을 때 나오는  $y$  값을 ‘예측 값’이라고 하겠음

## ▼ 최소 제곱법 공식으로 구한 성적 예측 값

공부한 시간	2	4	6	8
성적	81	93	91	97
예측 값	83.6	88.2	92.8	97.4

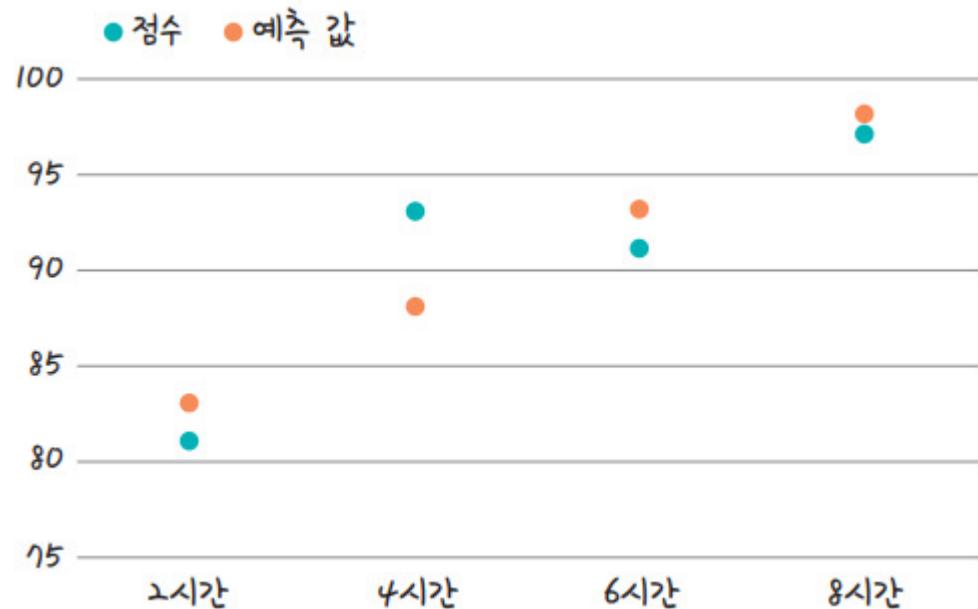


### 3 최소 제곱법

#### ● 최소 제곱법

- 좌표 평면에 이 예측 값을 찍어 보자

▼ 그림 4-2 | 공부한 시간, 성적, 예측 값을 좌표로 표현



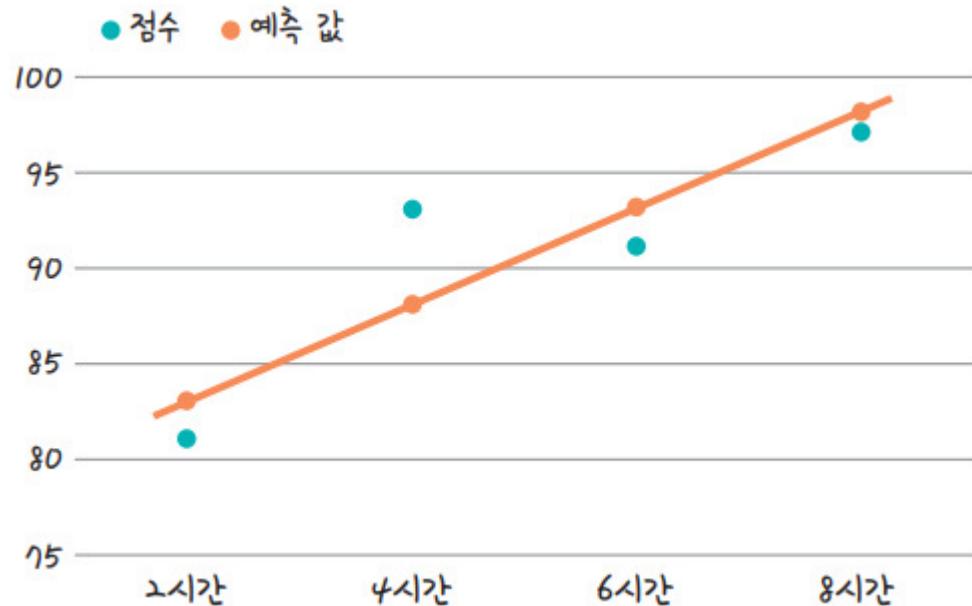


### 3 최소 제곱법

#### ● 최소 제곱법

- 예측한 점들을 연결해 직선을 그으면 그림 4-3과 같음

#### ▼ 그림 4-3 | 오차가 최저가 되는 직선의 완성





# 3 최소 제곱법

## ● 최소 제곱법

- 이것이 바로 오차가 가장 적은 주어진 좌표의 특성을 가장 잘 나타내는 직선
- 우리가 원하는 예측 직선
- 이 직선에 우리는 다른  $x$  값(공부한 시간)을 집어넣어서 ‘공부량에 따른 성적을 예측’ 할 수 있음



## 4 파이썬 코딩으로 확인하는 최소 제곱



# 4 파이썬 코딩으로 확인하는 최소 제곱

- 파이썬 코딩으로 확인하는 최소 제곱

## 실습 | 파이썬 코딩으로 구하는 최소 제곱



```
import numpy as np

# 공부한 시간과 점수를 각각 x, y라는 이름의 넘파이 배열로 만듭니다.
x = np.array([2, 4, 6, 8])
y = np.array([81, 93, 91, 97])

# x의 평균값을 구합니다.
mx = np.mean(x)
```



# 4 파이썬 코딩으로 확인하는 최소 제곱

## ● 파이썬 코딩으로 확인하는 최소 제곱

# y의 평균값을 구합니다.

```
my = np.mean(y)
```

# 출력으로 확인합니다.

```
print("x의 평균값: ", mx)  
print("y의 평균값: ", my)
```

# 기울기 공식의 분모 부분입니다.

```
divisor = sum([(i - mx)**2 for i in x])
```



# 4 파이썬 코딩으로 확인하는 최소 제곱

## ● 파이썬 코딩으로 확인하는 최소 제곱

```
# 기울기 공식의 문자 부분입니다.
```

```
def top(x, mx, y, my):  
    d = 0  
    for i in range(len(x)):  
        d += (x[i] - mx) * (y[i] - my)  
    return d  
  
dividend = top(x, mx, y, my)
```

```
# 출력으로 확인합니다.
```

```
print("분모: ", divisor)  
print("분자: ", dividend)
```



# 4 파이썬 코딩으로 확인하는 최소 제곱

## ● 파이썬 코딩으로 확인하는 최소 제곱

```
# 기울기 a를 구하는 공식입니다.
```

```
a = dividend / divisor
```

```
# y 절편 b를 구하는 공식입니다.
```

```
b = my - (mx*a)
```

```
# 출력으로 확인합니다.
```

```
print("기울기 a = ", a)
```

```
print("y 절편 b = ", b)
```



## 4 파이썬 코딩으로 확인하는 최소 제곱

### ● 파이썬 코딩으로 확인하는 최소 제곱

실행 결과

x의 평균값: 5.0

y의 평균값: 90.5

분모: 20.0

분자: 46.0

기울기  $a = 2.3$

y 절편  $b = 79.0$

- 파이썬으로 최소 제곱법을 구현해 기울기  $a$ 의 값과  $y$  절편  $b$ 의 값이 각각 2.3과 79임을 구할 수 있었음



## 5 평균 제곱 오차

---



# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 최소 제곱법을 이용해 기울기  $a$ 와  $y$  절편을 편리하게 구했지만, 이 공식만으로 앞으로 만나게 될 모든 상황을 해결하기는 어려움
- 여러 개의 입력을 처리하기에는 무리가 있기 때문임
- 예를 들어 앞서 살펴본 예에서는 변수가 ‘공부한 시간’ 하나뿐이지만, 앞에서 살펴본 폐암 수술 환자의 생존율 데이터를 보면 입력 데이터의 종류가 17개나 됨
- 딥러닝은 대부분 입력 값이 여러 개인 상황에서 이를 해결하기 위해 실행되기 때문에 기울기  $a$ 와  $y$  절편  $b$ 를 찾아내는 다른 방법이 필요함
  
- 가장 많이 사용하는 방법은 ‘일단 그리고 조금씩 수정해 나가기’ 방식
- 가설을 하나 세운 후 이 값이 주어진 요건을 충족하는지 판단해서 조금씩 변화를 주고, 이 변화가 긍정적이면 오차가 최소가 될 때까지 이 과정을 계속 반복하는 방법
- 이는 딥러닝을 가능하게 하는 가장 중요한 원리 중 하나



# 5 평균 제곱 오차

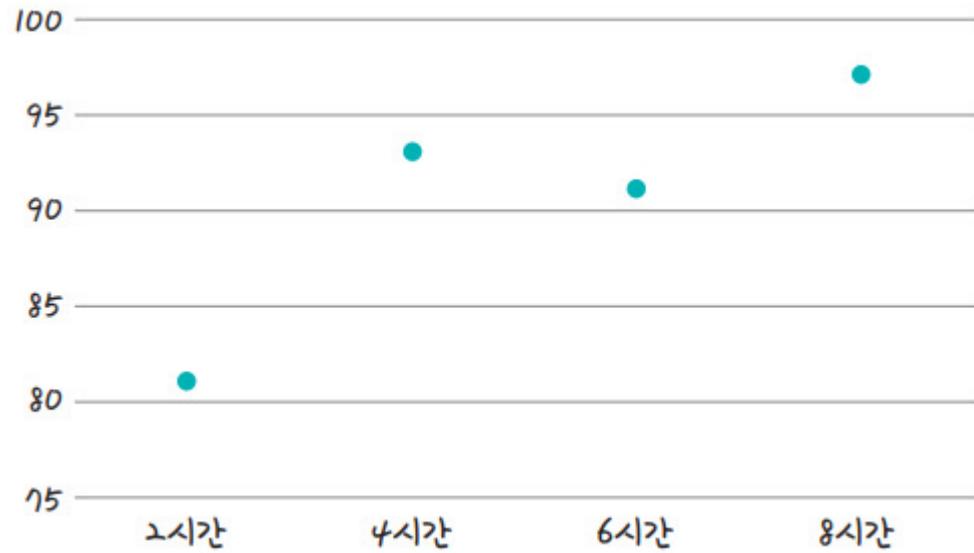
## ● 평균 제곱 오차

- 선을 긋고 나서 수정하는 과정에서 빠지면 안 되는 것이 있음
- 나중에 그린 선이 먼저 그린 선보다 더 좋은지 나쁜지를 판단하는 방법
- 즉, 각 선의 오차를 계산할 수 있어야 하고, 오차가 작은 쪽으로 바꾸는 알고리즘이 필요한 것
  
- 이를 위해 주어진 선의 오차를 평가하는 방법이 필요함
- 오차를 구할 때 가장 많이 사용되는 방법이 **평균 제곱 오차**(Mean Square Error, MSE)
- 지금부터 평균 제곱 오차를 구하는 방법을 알아보자
- 앞서 나온 공부한 시간과 성적의 관계도를 다시 한 번 살펴보자.



## 5 평균 제곱 오차

▼ 그림 4-5 | 공부한 시간과 성적의 관계도





# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 우리는 조금 전 최소 제곱법을 이용해 점들의 특성을 가장 잘 나타내는 최적의 직선이  $y = 2.3x + 79$ 임을 구했지만, 이번에는 최소 제곱법을 사용하지 않고 아무 값이나  $a$ 와  $b$ 에 대입해 보자
- 임의의 값을 대입한 후 오차를 구하고 이 오차를 최소화하는 방식을 사용해서 최종  $a$  값과  $b$  값을 구해 보자

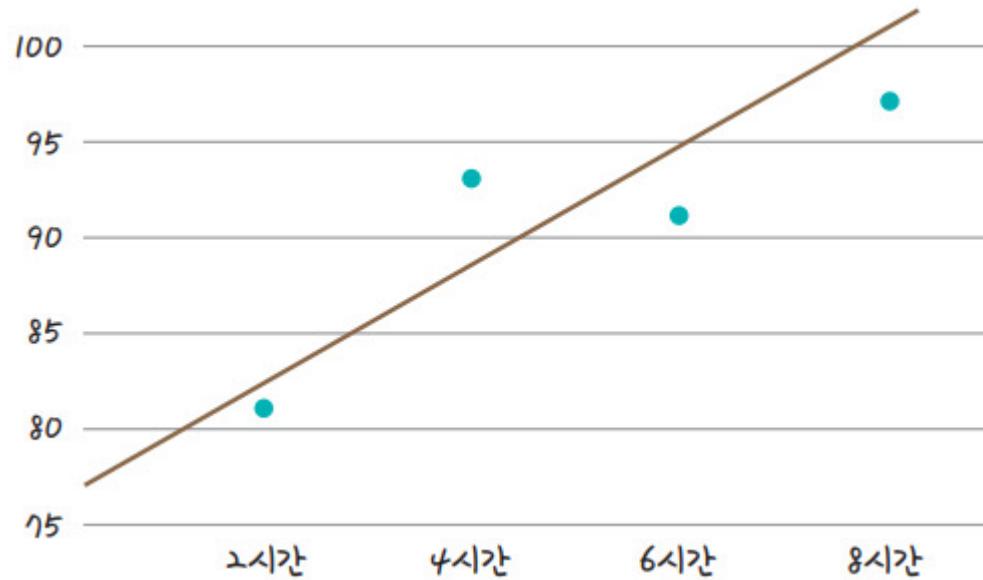


# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 먼저 대강 선을 그어 보기 위해 기울기  $a$ 와  $y$  절편  $b$ 를 임의의 수 3과 76이라고 가정해 보자
- $Y = 3x + 76$ 인 선을 그려 보면 그림 4-6과 같음

## ▼ 그림 4-6 | 임의의 직선 그려 보기



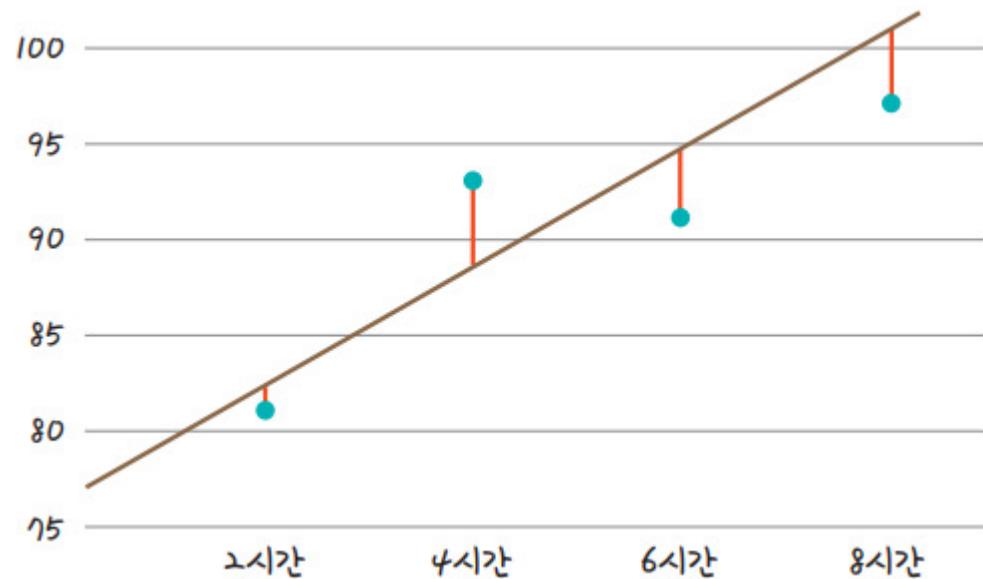


# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 그림 4-6과 같은 임의의 직선이 어느 정도의 오차가 있는지 확인하려면 각 점과 그래프 사이의 거리를 재면 됨

### ▼ 그림 4-7 | 임의의 직선과 실제 값 사이의 거리





# 5 평균 제곱 오차

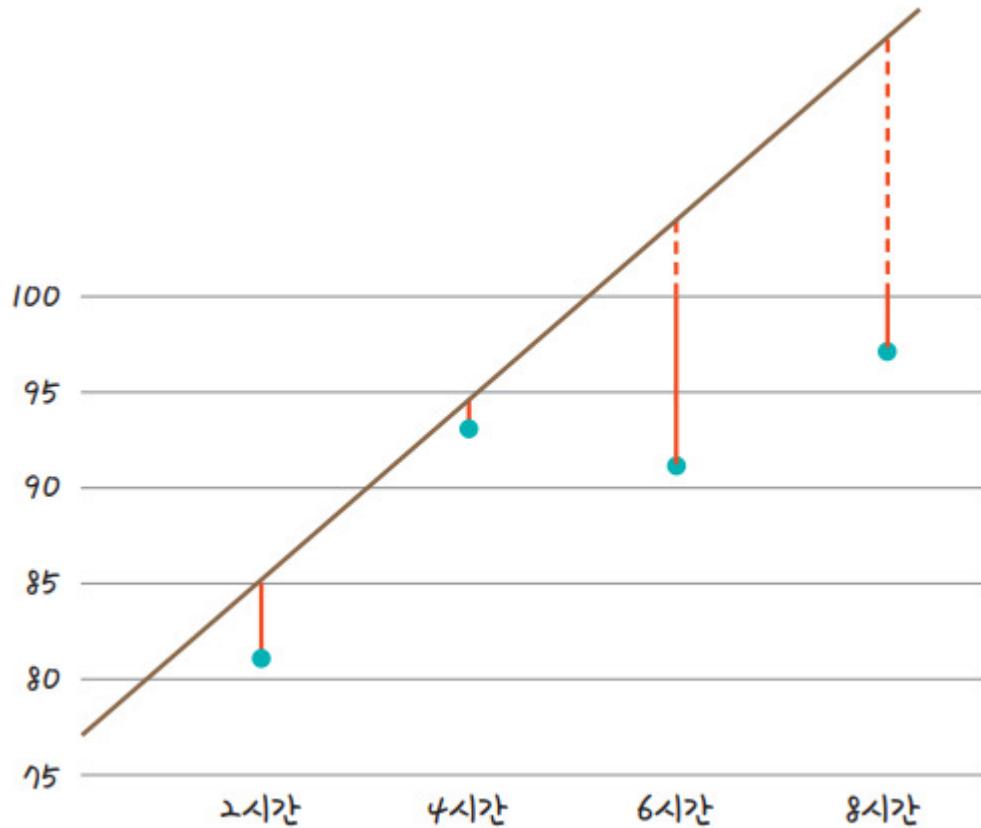
## ● 평균 제곱 오차

- 그림 4-7에서 볼 수 있는 빨간색 선은 직선이 잘 그어졌는지 나타냄
- 이 직선들의 합이 작을수록 잘 그어진 직선이고, 이 직선들의 합이 클수록 잘못 그어진 직선이 됨
- 예를 들어 기울기 값을 각각 다르게 설정한 그림 4-8과 그림 4-9의 그래프를 살펴보자.



## 5 평균 제곱 오차

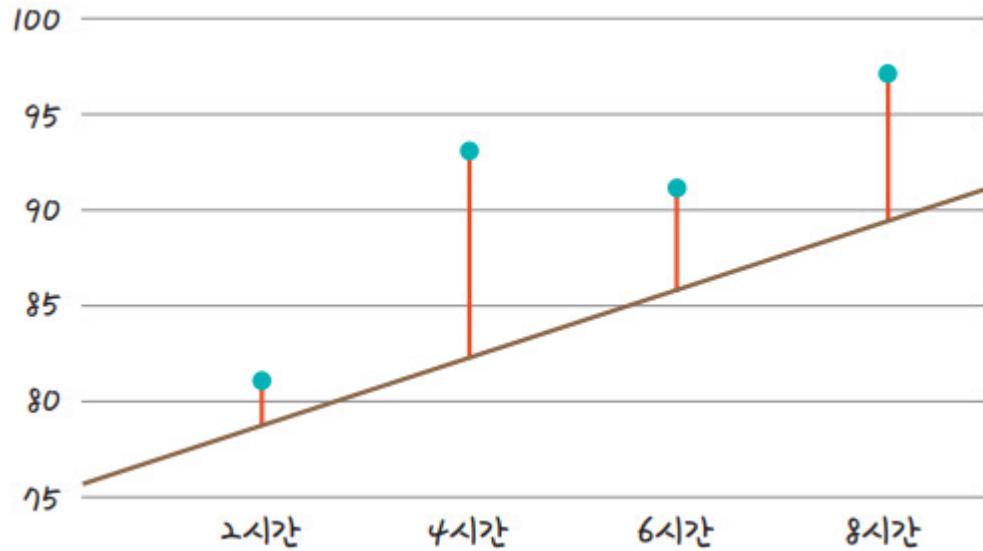
▼ 그림 4-8 | 기울기를 너무 크게 잡았을 때 오차





## 5 평균 제곱 오차

▼ 그림 4-9 | 기울기를 너무 작게 잡았을 때 오차





# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 그래프의 기울기가 잘못되었을 수록 빨간색 선의 거리의 합, 즉 오차의 합도 커짐
- 만일 기울기가 무한대로 커지면 오차도 무한대로 커지는 상관관계가 있는 것을 알 수 있음
- 빨간색 선의 거리의 합을 실제로 계산해 보자
- 거리는 입력 데이터에 나와 있는  $y$ 의 ‘실제 값’과  $x$ 를  $y = 3x + 76$  식에 대입해서 나오는 ‘예측 값’의 차이를 이용해 구할 수 있음
- 예를 들어 2시간을 공부했을 때 실제 나온 점수(81점)와 그래프  $y = 3x + 76$  식에  $x = 2$ 를 대입했을 때(82점)의 차이가 곧 오차
- 오차를 구하는 방정식은 다음과 같음

$$\text{오차} = \text{실제 값} - \text{예측 값}$$



# 5 평균 제곱 오차

## ● 평균 제곱 오차

- 이 식에 주어진 데이터를 대입해 얻을 수 있는 모든 오차 값을 정리하면 아래와 같음

## ▼ 주어진 데이터에서 오차 구하기

공부한 시간(x)	2	4	6	8
성적(실제 값, y)	81	93	91	97
예측 값	82	88	94	100
오차	1	-5	3	3



# 5 평균 제곱 오차

## ● 평균 제곱 오차

$$\text{오차의 합} = \sum_i^n (y_i - \hat{y}_i)^2$$

$$\text{평균 제곱 오차(MSE)} = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$$

- 머신 러닝과 딥러닝을 공부할 때 자주 등장할 중요한 식
- 선형 회귀란 임의의 직선을 그어 이에 대한 평균 제곱 오차를 구하고, 이 값을 가장 작게 만들어 주는 a 값과 b 값을 찾아가는 작업



## 6 파이썬 코딩으로 확인하는 평균 제곱 오차



# 6 파이썬 코딩으로 확인하는 평균 제곱 오차

## ● 파이썬 코딩으로 확인하는 평균 제곱 오차

- \*\* $2$ 는 제곱을 구하라는 것이고, `sum()`은 합을 구하라는 것
- 실제 값과 예측 값을 각각 `mse()` 함수의 `y`와 `y_pred` 자리에 넣어서 평균 제곱을 구함

### 실습 | 파이썬 코딩으로 구하는 평균 제곱 오차



```
import numpy as np

# 가상의 기울기 a와 y 절편 b를 정합니다.
fake_a = 3
fake_b = 76

# 공부 시간 x와 성적 y의 넘파이 배열을 만듭니다.
x = np.array([2, 4, 6, 8])
y = np.array([81, 93, 91, 97])
```



# 6 파이썬 코딩으로 확인하는 평균 제곱 오차

## ● 파이썬 코딩으로 확인하는 평균 제곱 오차

```
# y = ax + b에 가상의 a 값과 b 값을 대입한 결과를 출력하는 함수입니다.
def predict(x):
    return fake_a * x + fake_b

# 예측 값이 들어갈 빈 리스트를 만듭니다.
predict_result = []

# 모든 x 값을 한 번씩 대입해 predict_result 리스트를 완성합니다.
for i in range(len(x)):
    predict_result.append(predict(x[i]))
    print("공부시간=%.f, 실제점수=%.f, 예측점수=%.f" % (x[i], y[i], predict(x[i])))
```



# 6 파이썬 코딩으로 확인하는 평균 제곱 오차

## ● 파이썬 코딩으로 확인하는 평균 제곱 오차

```
# 평균 제곱 오차 함수를 각 y 값에 대입해 최종 값을 구하는 함수입니다.  
n = len(x)  
  
def mse(y, y_pred):  
    return (1/n) * sum((y - y_pred)**2)  
  
# 평균 제곱 오차 값을 출력합니다.  
print("평균 제곱 오차: " + str(mse(y, predict_result)))
```



# 6 파이썬 코딩으로 확인하는 평균 제곱 오차

## ● 파이썬 코딩으로 확인하는 평균 제곱 오차

### 실행 결과

공부시간=2, 실제점수=81, 예측점수=82

공부시간=4, 실제점수=93, 예측점수=88

공부시간=6, 실제점수=91, 예측점수=94

공부시간=8, 실제점수=97, 예측점수=100

평균 제곱 오차: 11.0



# 6 파이썬 코딩으로 확인하는 평균 제곱 오차

## ● 파이썬 코딩으로 확인하는 평균 제곱 오차

- 이를 통해 처음 가정한  $a = 3, b = 76$ 은 오차가 약 11.0이라는 것을 알게 됨
- 이제 남은 것은 이 오차를 줄이면서 새로운 선을 긋는 것
- 이를 위해서는  $a$  값과  $b$  값을 적절히 조절하면서 오차의 변화를 살펴보고, 그 오차가 최소화되는  $a$  값과  $b$  값을 구해야 함

# 선형 회귀 모델: 먼저 굿고 수정하기

---

- 1 경사 하강법의 개요
- 2 파이썬 코딩으로 확인하는 선형 회귀
- 3 다중 선형 회귀의 개요
- 4 파이썬 코딩으로 확인하는 다중 선형 회귀
- 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델



# 선형 회귀 모델: 먼저 굿고 수정하기

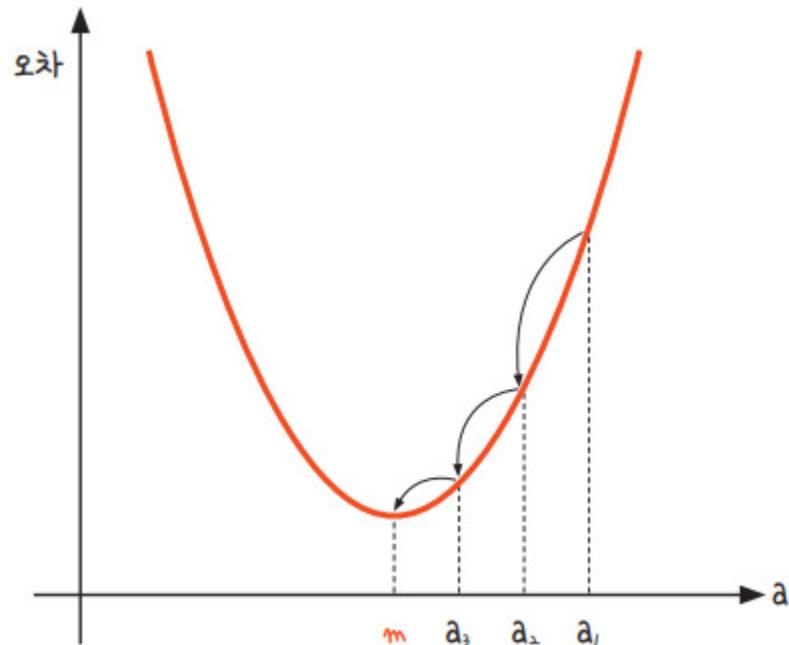
## ● 선형 회귀 모델: 먼저 굿고 수정하기

- 앞에서 기울기  $a$ 를 너무 크게 잡으면 오차가 커지는 것을 확인
- 기울기를 너무 작게 잡아도 오차가 커짐
- 기울기  $a$ 와 오차 사이에는 이런 상관관계가 있음
- 이때 기울기가 무한대로 커지거나 무한대로 작아지면 그래프는  $y$ 축과 나란한 직선이 됨
- 오차도 함께 무한대로 커짐
- 이를 다시 표현하면 기울기  $a$ 와 오차 사이에는 그림 5-1의 빨간색 그래프와 같은 이차 함수의 관계가 있다는 의미



# 선형 회귀 모델: 먼저 굿고 수정하기

▼ 그림 5-1 | 기울기  $a$ 와 오차의 관계: 적절한 기울기를 찾았을 때 오차가 최소화된다





# 선형 회귀 모델: 먼저 굿고 수정하기

## ● 선형 회귀 모델: 먼저 굿고 수정하기

- 이 그래프상에서 오차가 가장 작을 때는 그래프의 가장 아래쪽 볼록한 부분에 이르렀을 때
- 즉, 기울기  $a$ 가  $m$ 의 위치에 있을 때
- 우리는 앞에서 임의의 기울기를 집어넣어 평균 제곱 오차를 구해 보았음
- 그때의 기울기를  $a_1$ 이라고 한다면, 기울기를 적절히 바꾸어  $a_2, a_3$ 으로 이동시키다 결국  $m$ 에 이르게 하면 최적의 기울기를 찾게 되는 것
- 이 작업을 위해  $a_1$  값보다  $a_2$  값이  $m$ 에 더 가깝고,  $a_3$  값이  $a_2$  값보다  $m$ 에 더 가깝다는 것을 컴퓨터가 판단
- 이러한 판단을 하게 하는 방법이 바로 미분 기울기를 이용하는 경사 하강법(gradient decent)



# 1 경사 하강법의 개요

---

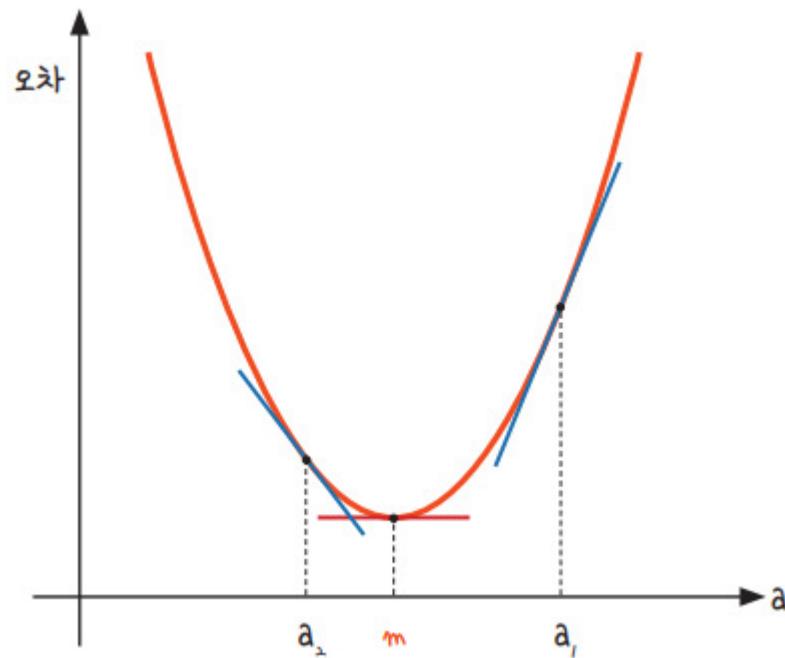


# 1 경사 하강법의 개요

## ● 경사 하강법의 개요

- $y = x^2$  그래프에서  $x$ 에 다음과 같이  $a_1, a_2$  그리고  $m$ 을 대입해 그 자리에서 미분하면 그림 5-2와 같이 각 점에서의 순간 기울기가 그려짐

▼ 그림 5-2 | 순간 기울기가 0인 점이 곧 우리가 찾는 최솟값  $m$ 이다





# 1 경사 하강법의 개요

## ● 경사 하강법의 개요

- 눈여겨보아야 할 것은 우리가 찾는 최솟값  $m$ 에서의 순간 기울기
  - 그래프가 이차 함수 포물선이므로 꼭짓점의 기울기는  $x$ 축과 평행한 선이 됨
  - 즉, 기울기가 0
  - 할 일은 ‘미분 값이 0인 지점’을 찾는 것이 됨
- 
- 이를 위해 다음 과정을 거침

1 |  $a_1$ 에서 미분을 구함

2 | 구한 기울기의 반대 방향(기울기가 +면 음의 방향, -면 양의 방향)으로 얼마간 이동시킨  $a_2$ 에서 미분을 구함(그림 5-3 참조)

3 | 앞에서 구한 미분 값이 0이 아니면 1과 2 과정을 반복

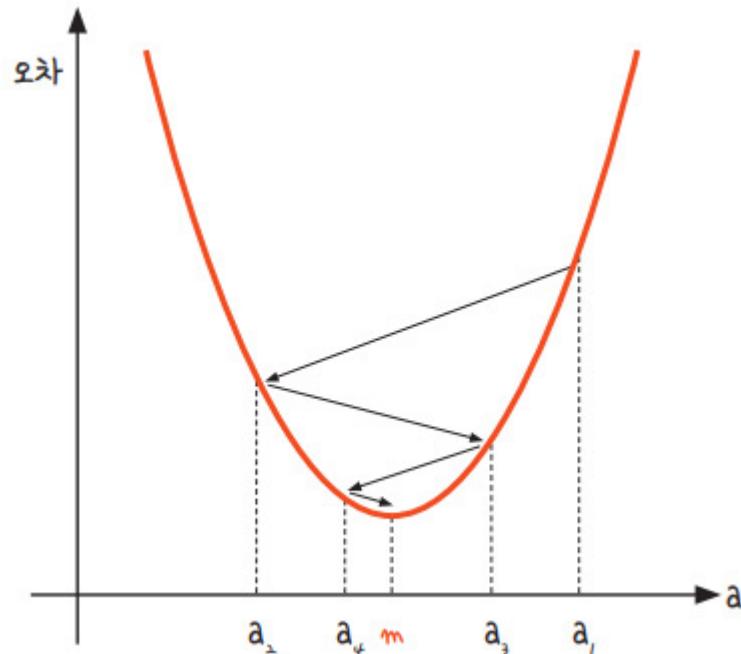


# 1 경사 하강법의 개요

## ● 경사 하강법의 개요

- 그림 5-3과 같이 기울기가 0인 한 점( $m$ )으로 수렴

## ▼ 그림 5-3 | 최솟점 $m$ 을 찾아가는 과정





# 1 경사 하강법의 개요

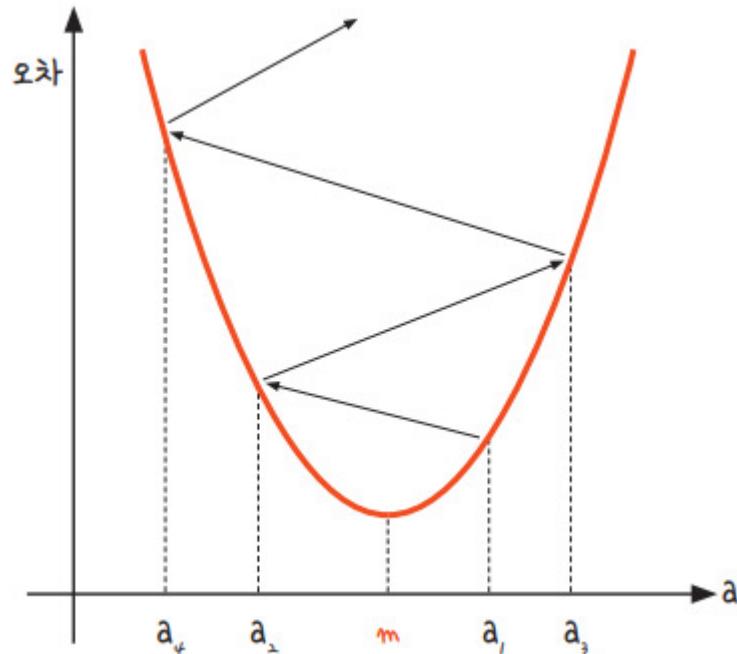
## ● 경사 하강법의 개요

- 경사 하강법은 반복적으로 기울기  $a$ 를 변화시켜서  $m$  값을 찾아내는 방법
- 여기서 학습률(learning rate)이라는 개념을 알 수 있음
- 기울기의 부호를 바꾸어 이동시킬 때 적절한 거리를 찾지 못해 너무 멀리 이동시키면  $a$  값이 한 점으로 모이지 않고 그림 5-4와 같이 위로 치솟아 버림



# 1 경사 하강법의 개요

▼ 그림 5-4 | 학습률을 너무 크게 잡으면 한 점으로 수렴하지 않고 발산





# 1 경사 하강법의 개요

## ● 경사 하강법의 개요

- 어느 만큼 이동시킬지 신중히 결정해야 하는데, 이때 이동 거리를 정해 주는 것이 바로 학습률
- 딥러닝에서 학습률의 값을 적절히 바꾸면서 최적의 학습률을 찾는 것은 중요한 최적화 과정 중 하나
- 다시 말해 경사 하강법은 오차의 변화에 따라 이차 함수 그래프를 만들고 적절한 학습률을 설정해 미분 값이 0인 지점을 구하는 것
- $y$  절편  $b$ 의 값도 이와 같은 성질을 가지고 있음
- $b$  값이 너무 크면 오차도 함께 커지고, 너무 작아도 오차가 커짐
- 최적의  $b$  값을 구할 때 역시 경사 하강법을 사용

## 2 파이썬 코딩으로 확인하는 선형 회귀



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

- 평균 제곱 오차의 식

$$\frac{1}{n} \sum (y_i - \hat{y}_i)^2$$

- 여기서  $\hat{y}_i$ 는  $y = ax + b$ 의 식에  $x_i$ 를 집어넣었을 때 값이므로  $y_i = ax_i + b$ 를 대입하면 다음과 같이 바뀜

$$\frac{1}{n} \sum (y_i - (ax_i + b))^2$$

- 이 값을 미분할 때 우리가 궁금한 것은  $a$ 와  $b$ 라는 것을 기억해야 함
- 식 전체를 미분하는 것이 아니라 필요한 값을 중심으로 미분해야 하기 때문임
- 이렇게 특정한 값, 즉,  $a$ 와  $b$ 를 중심으로 미분할 때 이를  $a$ 와  $b$ 로 ‘편미분한다’고 함

$$a\text{로 편미분한 결과} = \frac{2}{n} \sum -x_i(y_i - (ax_i + b))$$

$$b\text{로 편미분한 결과} = \frac{2}{n} \sum -(y_i - (ax_i + b))$$



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

학습률 0.03은 어떻게 정했나요?

- 여러 학습률을 적용해 보며 최적의 결과를 만드는 학습률을 찾아낸 것
- 최적의 학습률은 데이터와 딥러닝 모델에 따라 다르므로 그때그때 찾아내야 함
- 앞으로 배우게 될 딥러닝 프로젝트에서는 자동으로 최적의 학습률을 찾아 주는 최적화 알고리즘들을 사용



## 2 파이썬 코딩으로 확인하는 선형 회귀

- 파이썬 코딩으로 확인하는 선형 회귀

### 실습 | 선형 회귀 모델 실습



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

# 공부 시간 X와 성적 y의 넘파이 배열을 만듭니다.
x = np.array([2, 4, 6, 8])
y = np.array([81, 93, 91, 97])

# 데이터의 분포를 그래프로 나타냅니다.
plt.scatter(x, y)
plt.show()
```



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

```
# 기울기 a의 값과 절편 b의 값을 초기화합니다.
```

```
a = 0
```

```
b = 0
```

```
# 학습률을 정합니다.
```

```
lr = 0.03
```

```
# 몇 번 반복될지 설정합니다.
```

```
epochs = 2001
```

```
# x 값이 총 몇 개인지 셹니다.
```

```
n = len(x)
```



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

```
# 경사 하강법을 시작합니다.  
  
for i in range(epochs):          # 에포크 수만큼 반복합니다.  
    y_pred = a * x + b           # 예측 값을 구하는 식입니다.  
    error = y - y_pred          # 실제 값과 비교한 오차를 error로 놓습니다.  
  
    a_diff = (2/n) * sum(-x * (error))  # 오차 함수를 a로 편미분한 값입니다.  
    b_diff = (2/n) * sum(-(error))      # 오차 함수를 b로 편미분한 값입니다.  
  
    a = a - lr * a_diff  # 학습률을 곱해 기존의 a 값을 업데이트합니다.  
    b = b - lr * b_diff  # 학습률을 곱해 기존의 b 값을 업데이트합니다.
```



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

```
if i % 100 == 0:      # 100번 반복될 때마다 현재의 a 값, b 값을 출력합니다.  
    print("epoch=% .f, 기울기=% .04f, 절편=% .04f" % (i, a, b))  
  
# 앞서 구한 최종 a 값을 기울기, b 값을 y 절편에 대입해 그래프를 그립니다.  
y_pred = a * x + b  
  
# 그래프를 출력합니다.  
plt.scatter(x, y)  
plt.plot(x, y_pred, 'r')  
plt.show()
```



## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

#### 실행 결과

epoch=0, 기울기=27.8400, 절편=5.4300

epoch=100, 기울기=7.0739, 절편=50.5117

epoch=200, 기울기=4.0960, 절편=68.2822

... (중략) ...

epoch=1900, 기울기=2.3000, 절편=79.0000

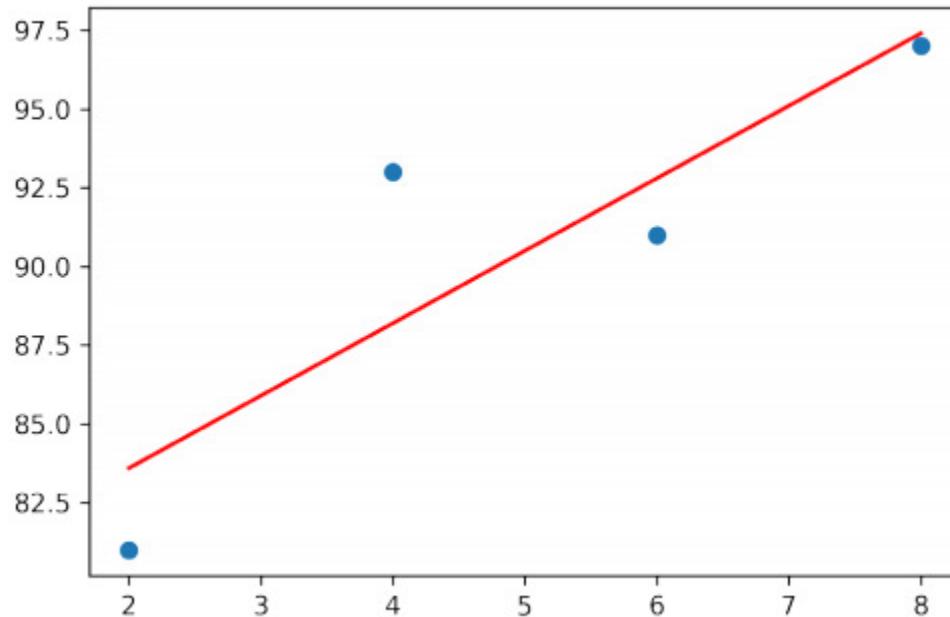
epoch=2000, 기울기=2.3000, 절편=79.0000



## 2 파이썬 코딩으로 확인하는 선형 회귀

- 파이썬 코딩으로 확인하는 선형 회귀

▼ 그림 5-5 | 그래프로 표현한 모습





## 2 파이썬 코딩으로 확인하는 선형 회귀

### ● 파이썬 코딩으로 확인하는 선형 회귀

- 여기서 에포크(epoch)는 입력 값에 대해 몇 번이나 반복해서 실험했는지 나타냄
- 우리가 설정한 실험을 반복하고 100번마다 결과를 내놓음
  
- 기울기  $a$ 의 값이 2.3에 수렴하는 것과  $y$  절편  $b$ 의 값이 79에 수렴하는 과정을 볼 수 있음
- 기울기 2.3과  $y$  절편 79는 앞서 우리가 최소 제곱법을 이용해 미리 확인한 값과 같음
- 이렇게 해서 최소 제곱법을 쓰지 않고 평균 제곱 오차와 경사 하강법을 이용해 원하는 값을 구할 수 있었음
- 이와 똑같은 방식을  $x$ 가 여러 개인 다중 선형 회귀에서도 사용



### 3 다중 선형 회귀의 개요

---



# 3 다중 선형 회귀의 개요

## ● 다중 선형 회귀의 개요

- 앞서 학생들이 공부한 시간에 따른 예측 직선을 그리고자 기울기  $a$ 와  $y$  절편  $b$ 를 구함
- 이 예측 직선을 이용해도 실제 성적 사이에는 약간의 오차가 있었음
- 4시간 공부한 친구는 88점을 예측했는데 이보다 좋은 93점을 받았고, 6시간 공부한 친구는 93점을 받을 것으로 예측했지만 91점을 받았음
- 이러한 차이가 생기는 이유는 공부한 시간 이외의 다른 요소가 성적에 영향을 끼쳤기 때문임
- 더 정확한 예측을 하려면 추가 정보를 입력해야 하며, 정보를 추가해 새로운 예측 값을 구하려면 변수 개수를 늘려 **다중 선형 회귀**를 만들어 주어야 함



### 3 다중 선형 회귀의 개요

#### ● 다중 선형 회귀의 개요

- 예를 들어 일주일 동안 받는 과외 수업 횟수를 조사해서 이를 기록해 보았음

▼ 표 5-1 | 공부한 시간, 과외 수업 횟수에 따른 성적 데이터

공부한 시간( $x_1$ )	2	4	6	8
과외 수업 횟수( $x_2$ )	0	4	2	3
성적(y)	81	93	91	97



# 3 다중 선형 회귀의 개요

## ● 다중 선형 회귀의 개요

- 그럼 지금부터 독립 변수  $x_1$ 과  $x_2$ 가 두 개 생긴 것
- 이를 사용해 종속 변수  $y$ 를 만들 경우 기울기를 두 개 구해야 하므로 다음과 같은 식이 나옴

$$y = a_1x_1 + a_2x_2 + b$$

- 두 기울기  $a_1$ 과  $a_2$ 는 각각 어떻게 구할 수 있을까?
- 앞서 배운 경사 하강법을 그대로 적용



## 4 파이썬 코딩으로 확인하는 다중 선형 회귀

---



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 지금까지 코드를 정리하면 다음과 같음

### 실습 | 파이썬 코딩으로 확인하는 다중 선형 회귀



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

# 공부 시간 x1과 과외 시간 x2, 성적 y의 넘파이 배열을 만듭니다.
x1 = np.array([2, 4, 6, 8])
x2 = np.array([0, 4, 2, 3])
y = np.array([81, 93, 91, 97])
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

```
# 데이터의 분포를 그래프로 나타냅니다.  
fig = plt.figure()  
ax = fig.add_subplot(111, projection='3d')  
ax.scatter3D(x1, x2, y);  
plt.show()
```

```
# 기울기 a의 값과 절편 b의 값을 초기화합니다.  
a1 = 0  
a2 = 0  
b = 0
```

```
# 학습률을 정합니다.  
lr = 0.01
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

```
# 몇 번 반복될지 설정합니다.  
epochs = 2001  
  
# x 값이 총 몇 개인지 셹니다. x1과 x2의 수가 같으므로 x1만 세겠습니다.  
n = len(x1)  
  
# 경사 하강법을 시작합니다.  
for i in range(epochs):          # 에포크 수만큼 반복합니다.  
    y_pred = a1 * x1 + a2 * x2 + b # 예측 값을 구하는 식을 세웁니다.  
    error = y - y_pred           # 실제 값과 비교한 오차를 error로 놓습니다.
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

```
a1_diff = (2/n) * sum(-x1 * (error)) # 오차 함수를 a1로 편미분한 값입니다.  
a2_diff = (2/n) * sum(-x2 * (error)) # 오차 함수를 a2로 편미분한 값입니다.  
b_diff = (2/n) * sum(-(error)) # 오차 함수를 b로 편미분한 값입니다.  
  
a1 = a1 - lr * a1_diff # 학습률을 곱해 기존의 a1 값을 업데이트합니다.  
a2 = a2 - lr * a2_diff # 학습률을 곱해 기존의 a2 값을 업데이트합니다.  
b = b - lr * b_diff # 학습률을 곱해 기존의 b 값을 업데이트합니다.  
  
if i % 100 == 0: # 100번 반복될 때마다 현재의 a1, a2, b의 값을 출력합니다.  
    print("epoch=%f, 기울기1=%f, 기울기2=%f, 절편=%f" % (i, a1,  
a2, b))
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

```
# 실제 점수와 예측된 점수를 출력합니다.
```

```
print("실제 점수: ", y)
```

```
print("예측 점수: ", y_pred)
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

실행 결과

... (전략) ...

epoch=1700, 기울기1=1.5496, 기울기2=2.3028, 절편=77.5168

epoch=1800, 기울기1=1.5361, 기울기2=2.2982, 절편=77.6095

epoch=1900, 기울기1=1.5263, 기울기2=2.2948, 절편=77.6769

epoch=2000, 기울기1=1.5191, 기울기2=2.2923, 절편=77.7260

실제 점수: [81 93 91 97]

예측 점수: [80.76387645 92.97153922 91.42520875 96.7558749]



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- x 값이 두 개이므로 다음과 같이 공부 시간  $x_1$ , 과외 시간  $x_2$ , 성적 y의 넘파이 배열을 만듦

```
x1 = np.array([2, 4, 6, 8])
x2 = np.array([0, 4, 2, 3])
y = np.array([81, 93, 91, 97])
```

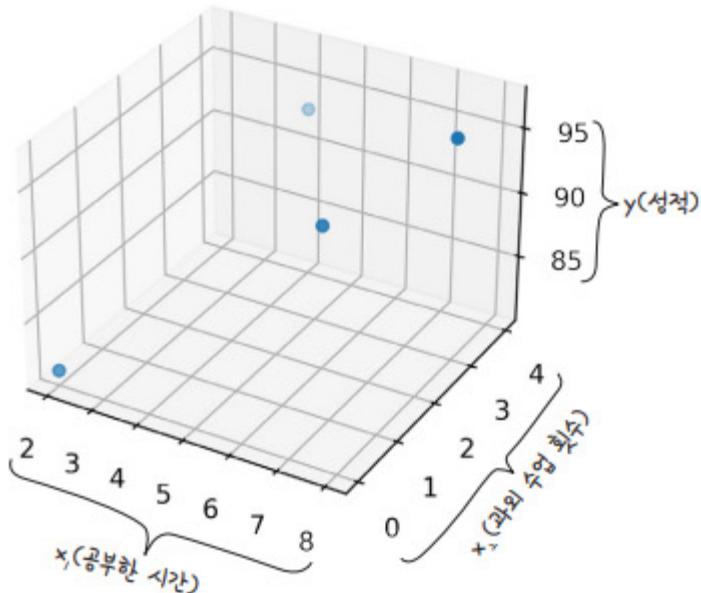
- 데이터의 분포를 그래프로 표현해 보면 다음과 같음

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter3D(x1, x2, y);
plt.show()
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

▼ 그림 5-6 | 축이 하나 더 늘어 3D로 배치된 모습





# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

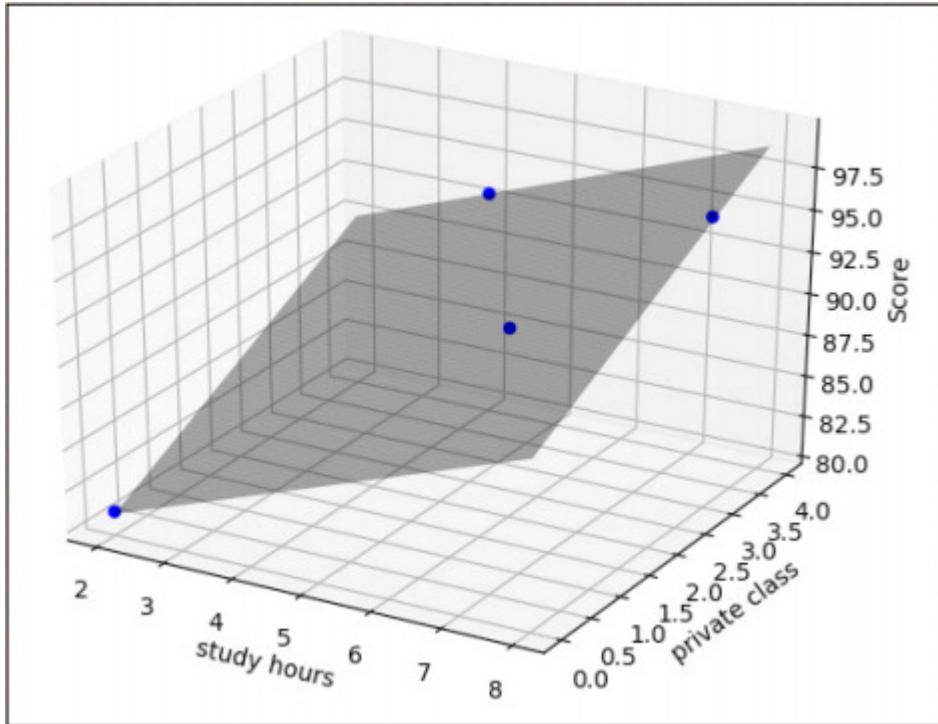
## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 앞서  $x$ 와  $y$  두 개의 축이던 것과는 달리  $x_1, x_2, y$  이렇게 세 개의 축이 필요함
- 새로운 변수가 추가되면 차원이 하나씩 추가되면서 계산은 더욱 복잡해지는 것을 알 수 있음
  
- 선형 회귀는 선을 긋는 작업이라고 했음
- 그러면 다중 선형 회귀는 어떨까?
- 최적의 결과를 찾은 후 이를 그래프로 표현하면 그림 5-7과 같이 평면으로 표시



## 4 파이썬 코딩으로 확인하는 다중 선형 회귀

▼ 그림 5-7 | 다중 선형 회귀





# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

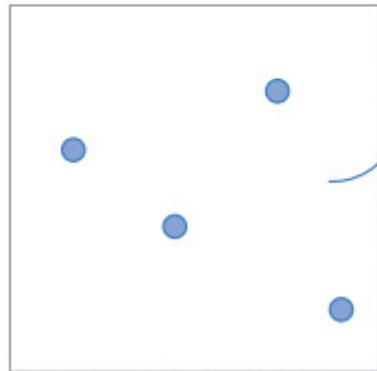
## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 직선상에서 예측하던 것이 평면으로 범위가 넓어지므로 계산이 복잡해지고 더 많은 데이터를 필요로 하게 됨

단순 선형 회귀



다중 선형 회귀



빈 공간이 많아진다.  
→ 계산이 복잡하다.  
→ 더 많은 데이터가 필요하다.



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 코드의 형태는 크게 다르지 않음
- 다만  $x$ 가 두 개가 되었으므로  $x_1, x_2$  두 변수를 만들고, 기울기도  $a_1$ 과  $a_2$  이렇게 두 개를 만듦
- 앞서 했던 방법대로 경사 하강법 적용
- 예측 값을 구하는 식을 다음과 같이 세움

```
y_pred = a1 * x1 + a2 * x2 + b # 기울기와 절편 자리에 a1, a2, b를 각각 대입합니다.  
error = y - y_pred # 실제 값과 비교한 오차를 error로 놓습니다.
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 오차 함수를  $a_1, a_2, b$ 로 각각 편미분한 값을  $a1\_diff, a2\_diff, b\_diff$ 라고 할 때 이를 구하는 식은 다음과 같음

```
n = len(x1)                                # 변수의 총 개수입니다.  
a1_diff = (2/n) * sum(-x1 * (error))    # 오차 함수를 a1로 편미분한 값입니다.  
a2_diff = (2/n) * sum(-x2 * (error))    # 오차 함수를 a2로 편미분한 값입니다.  
b_diff = (2/n) * sum(-(error))          # 오차 함수를 b로 편미분한 값입니다.
```

- 학습률을 곱해 기존의 기울기와 절편을 업데이트한 값을 구함

```
a1 = a1 - lr * a1_diff      # 학습률을 곱해 기존의 a1 값을 업데이트합니다.  
a2 = a2 - lr * a2_diff      # 학습률을 곱해 기존의 a2 값을 업데이트합니다.  
b = b - lr * b_diff        # 학습률을 곱해 기존의 b 값을 업데이트합니다.
```



# 4 파이썬 코딩으로 확인하는 다중 선형 회귀

## ● 파이썬 코딩으로 확인하는 다중 선형 회귀

- 이제 실제 점수와 예측된 점수를 출력해서 예측이 잘되는지 확인

```
print("실제 점수: ", y)
print("예측 점수: ", y_pred)
```

- 2,000번 반복했을 때 최적의 기울기  $a_1$ 과  $a_2$  및 절편을 찾아가며 실제 점수에 가까운 예측 값을 만들어 내고 있음



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

- 머신 러닝의 기본인 선형 회귀에 대해 배우고 있음
- 딥러닝을 실행하기 위해 텐서플로라는 라이브러리의 클래스 API를 불러와 사용할 것
- 지금까지 배운 선형 회귀의 개념과 딥러닝 라이브러리들이 어떻게 연결되는지 살펴볼 필요가 있음
- 이를 통해 텐서플로 및 클래스의 사용법을 익히는 것은 물론이고 딥러닝 자체에 대한 학습도 한걸음 더 나가게 될 것



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

- 선형 회귀는 현상을 분석하는 방법의 하나
- 머신 러닝은 이러한 분석 방법을 이용해 예측 모델을 만드는 것
- 두 분야에서 사용하는 용어가 약간 다름
- 예를 들어 함수  $y = ax + b$ 는 공부한 시간과 성적의 관계를 유추하기 위해 필요했던 식
- 이렇게 문제를 해결하기 위해 가정하는 식을 머신 러닝에서는 가설 함수(hypothesis)라고 하며  $H(x)$ 라고 표기
- 또 기울기  $a$ 는 변수  $x$ 에 어느 정도의 가중치를 곱하는지 결정하므로, 가중치(weight)라고 하며,  $w$ 로 표시
- 절편  $b$ 는 데이터의 특성에 따라 따로 부여되는 값이므로 편향(bias)이라고 하며,  $b$ 로 표시
- 우리가 앞서 배운  $y = ax + b$ 는 머신 러닝에서 다음과 같이 표기

$$y = ax + b \rightarrow H(x) = wx + b$$



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

- 또한, 평균 제곱 오차처럼 실제 값과 예측 값 사이의 오차에 대한 식을 손실 함수(loss function)라고 함

평균 제곱 오차 → 손실 함수(loss function)

- 최적의 기울기와 절편을 찾기 위해 앞서 경사 하강법을 배웠음
- 딥러닝에서는 이를 옵티마이저(optimizer)라고 함
- 우리가 사용했던 경사 하강법은 딥러닝에서 사용하는 여러 옵티마이저 중 하나였음

경사 하강법 → 옵티마이저(optimizer)



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

#### 실습 | 텐서플로에서 실행하는 선형 회귀



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

# 텐서플로의 케라스 API에서 필요한 함수들을 불러옵니다.
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

x = np.array([2, 4, 6, 8])
y = np.array([81, 93, 91, 97])

model = Sequential()
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

```
# 출력 값, 입력 변수, 분석 방법에 맞게끔 모델을 설정합니다.  
model.add(Dense(1, input_dim=1, activation='linear'))  
  
# 오차 수정을 위해 경사 하강법(sgd)을, 오차의 정도를 판단하기 위해  
# 평균 제곱 오차(mse)를 사용합니다.  
model.compile(optimizer='sgd', loss='mse')  
  
# 오차를 최소화하는 과정을 2000번 반복합니다.  
model.fit(x, y, epochs=2000)  
  
plt.scatter(x, y)  
plt.plot(x, model.predict(x), 'r')      # 예측 결과를 그래프로 나타냅니다.  
plt.show()
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

# 임의의 시간을 집어넣어 점수를 예측하는 모델을 테스트해 보겠습니다.

```
hour = 7
```

```
prediction = model.predict([hour])
```

```
print("%.f시간을 공부할 경우의 예상 점수는 %.02f점입니다." % (hour, prediction))
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

#### 실행 결과

Epoch 1/2000

1/1 [=====] - 1s 114ms/step - loss: 9241.3984

... (중략) ...

Epoch 2000/2000

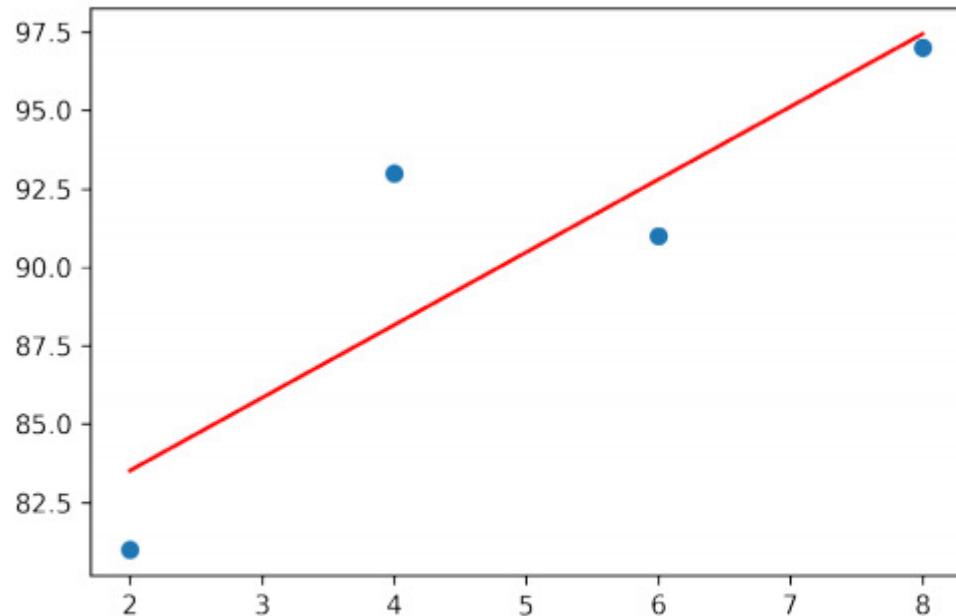
1/1 [=====] - 0s 2ms/step - loss: 8.3022



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

▼ 그림 5-8 | 텐서플로로 실행한 선형 회귀 분석 결과



7시간을 공부할 경우의 예상 점수는 95.12점입니다.



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

- 손실 함수, 옵티마이저라는 용어를 사용해 설명
- 먼저 텐서플로에 포함된 캐라스 API 중 필요한 함수들을 다음과 같이 불러옴

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense
```

- Sequential() 함수와 Dense() 함수는 2장에서 이미 소개한 바 있음
- 이 함수를 불러와 선형 회귀를 실행하는 코드는 다음과 같음

```
model.add(Dense(1, input_dim=1, activation='linear')) .....❶  
model.compile(optimizer='sgd', loss='mse') .....❷  
model.fit(x, y, epochs=2000) .....❸
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

- 단 세 줄의 코드에 앞서 공부한 모든 것이 담겨 있음
- ① 먼저 가설 함수는  $H(x) = wx + b$
- 이때 출력되는 값(=성적)이 하나씩이므로 Dense() 함수의 첫 번째 인자에 1이라고 설정
- 입력될 변수(=학습 시간)도 하나뿐이므로 input\_dim 역시 1이라고 설정
- 입력된 값을 다음 층으로 넘길 때 각 값을 어떻게 처리할지를 결정하는 함수를 활성화 함수라고 함
- activation은 활성화 함수를 정하는 옵션
- 여기에서는 선형 회귀를 다루고 있으므로 'linear'라고 적어 주면 됨
- 딥러닝 목적에 따라 다른 활성화 함수를 넣을 수 있는데, 예를 들어 앞으로 배울 시그모이드 함수가 필요하다면 'sigmoid'라고 넣어 주는 식



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 선형 회귀 모델

- ② 앞서 배운 경사 하강법을 실행하려면 옵티마이저에 sgd라고 설정
- 손실 함수는 평균 제곱 오차를 사용할 것이므로 mse라고 설정
- ③ 끝으로 앞서 따로 적어 주었던 epochs 숫자를 model.fit() 함수에 적음
- 학습 시간( $x$ )이 입력되었을 때의 예측 점수는 model.predict( $x$ )로 알 수 있음
- 예측 점수로 그래프를 그려 보면 다음과 같음

```
plt.scatter(x, y)
plt.plot(x, model.predict(x), 'r')    # 예측 결과를 그래프로 나타냅니다.
plt.show()
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

- 모든 코드를 정리하면 다음과 같음

#### 실습 | 텐서플로에서 실행하는 다중 선형 회귀



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

# 텐서플로의 케라스 API에서 필요한 함수들을 불러옵니다.
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

```
x = np.array([[2, 0], [4, 4], [6, 2], [8, 3]])  
y = np.array([81, 93, 91, 97])
```

```
model = Sequential()
```

# 입력 변수가 두 개(학습 시간, 과외 시간)이므로 input\_dim에 2를 입력합니다.

```
model.add(Dense(1, input_dim=2, activation='linear'))  
model.compile(optimizer='sgd', loss='mse')
```

```
model.fit(x, y, epochs=2000)
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

```
# 임의의 학습 시간과 과외 시간을 집어넣어 점수를 예측하는 모델을 테스트해 보겠습니다.  
hour = 7  
private_class = 4  
prediction = model.predict([[hour, private_class]])  
  
print("%.f시간을 공부하고 %.f시간의 과외를 받을 경우, 예상 점수는 %.02f점입니다."  
% (hour, private_class, prediction))
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

실행 결과

Epoch 1/2000

1/1 [=====] - 0s 119ms/step - loss: 8184.9204  
... (중략) ...

Epoch 2000/2000

1/1 [=====] - 0s 2ms/step - loss: 0.0743

7시간을 공부하고 4시간의 과외를 받을 경우, 예상 점수는 97.53점입니다.

- 다중 선형 회귀 모델을 통해 임의의 학습 시간과 과외 시간을 입력했을 때의 예상 점수를 확인할 수 있었음



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

- 앞서 구한 선형 회귀 결과와 같은 그래프를 구했음
- 임의의 시간을 넣었을 때 예상되는 점수를 보여 줌
- 마찬가지로 다중 선형 회귀 역시 텐서플로를 이용해서 실행해 보자
- 앞서 실행했던 내용과 거의 유사함
- 다만, 입력해야 하는 변수가 한 개에서 두 개로 늘었음
- 이 부분을 적용하려면 input\_dim 부분을 2로 변경해 줌

```
model.add(Dense(1, input_dim=2, activation='linear'))
```



## 5 텐서플로에서 실행하는 선형 회귀, 다중 선형 회귀 모델

### ● 텐서플로에서 실행하는 다중 선형 회귀 모델

- 변수가 두 개이므로, 모델의 테스트를 위해서도 변수를 두 개 입력해야 함
- 임의의 학습 시간과 과외 시간을 입력했을 때의 점수는 다음과 같이 설정해서 구함

```
hour = 7
private_class = 4
prediction = model.predict([[hour, private_class]])

print("%.f시간을 공부하고 %.f시간의 과외를 받을 경우, 예상 점수는 %.02f점입니다."
% (hour, private_class, prediction))
```

# 로지스틱 회귀 모델: 참 거짓 판단하기

---

- 1 로지스틱 회귀의 정의
- 2 시그모이드 함수
- 3 오차 공식
- 4 로그 함수
- 5 텐서플로에서 실행하는 로지스틱 회귀 모델



# 로지스틱 회귀 모델: 참 거짓 판단하기

## ● 로지스틱 회귀 모델: 참 거짓 판단하기

- 때로 할 말이 많아도 ‘예’ 혹은 ‘아니요’로만 대답해야 할 때가 있고, 이와 같은 상황이 딥러닝에서도 끊임없이 일어남
- 전달받은 정보를 놓고 참과 거짓 중 하나를 판단해 다음 단계로 넘기는 장치들이 딥러닝 내부에서 쉬지 않고 작동하는 것
- 딥러닝을 수행한다는 것은 겉으로 드러나지 않는 ‘미니 판단 장치’들을 이용해서 복잡한 연산을 해낸 끝에 최적의 예측 값을 내놓는 작업이라고 할 수 있음



# 로지스틱 회귀 모델: 참 거짓 판단하기

- 로지스틱 회귀 모델: 참 거짓 판단하기



- 참과 거짓 중 하나를 내놓는 과정은 로지스틱 회귀(logistic regression)의 원리를 거쳐 이루어짐
- 회귀 분석의 또 다른 토대를 이루는 로지스틱 회귀에 대해 알아보자



# 1 로지스틱 회귀의 정의

---



# 1 로지스틱 회귀의 정의

## ● 로지스틱 회귀의 정의

- 앞에서 공부한 시간과 성적 사이의 관계를 좌표에 나타냈을 때, 좌표의 형태가 직선으로 해결되는 선형 회귀를 사용하기에 적절했음을 보았음
- 직선으로 해결하기에 적절하지 않은 경우도 있음
- 점수가 아니라 오직 합격과 불합격만 발표되는 시험이 있다고 하자
- 공부한 시간에 따른 합격 여부를 조사해 보니 표 6-1과 같았음

▼ 표 6-1 | 공부한 시간에 따른 합격 여부

공부한 시간	2	4	6	8	10	12	14
합격 여부	불합격	불합격	불합격	합격	합격	합격	합격

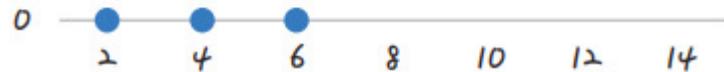


# 1 로지스틱 회귀의 정의

## ● 로지스틱 회귀의 정의

● 합격을 1, 불합격을 0이라고 하고, 이를 좌표 평면에 표현하면 그림 6-1과 같음

### ▼ 그림 6-1 | 합격과 불합격만 있을 때의 좌표 표현



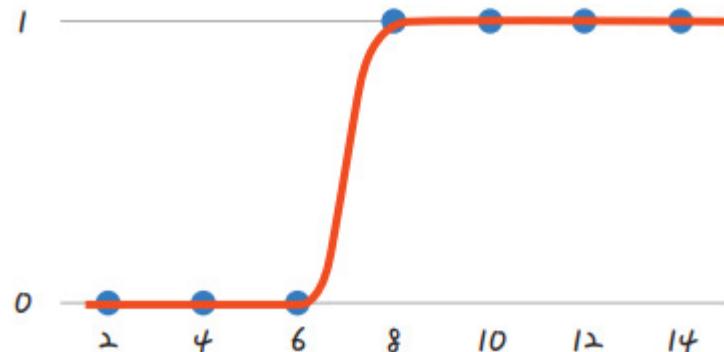


# 1 로지스틱 회귀의 정의

## ● 로지스틱 회귀의 정의

- 선을 그어 이 점의 특성을 잘 나타내는 일차 방정식을 만들 수 있을까?
- 이 점들은 1과 0 사이의 값이 없으므로 직선으로 그리기가 어려움
- 점들의 특성을 정확하게 담아내려면 직선이 아니라 다음과 같이 S자 형태여야 함

## ▼ 그림 6-2 | 각 점의 특성을 담은 선을 그었을 때





# 1 로지스틱 회귀의 정의

## ● 로지스틱 회귀의 정의

- 로지스틱 회귀는 선형 회귀와 마찬가지로 적절한 선을 그려 가는 과정
- 다만, 직선이 아니라 참(1)과 거짓(0) 사이를 구분하는 S자 형태의 선을 그어 주는 작업



## 2 시그모이드 함수

---



## 2 시그모이드 함수

### ● 시그모이드 함수

- 이러한 S자 형태로 그래프가 그려지는 함수가 시그모이드 함수(sigmoid function)
- 시그모이드 함수를 이용해 로지스틱 회귀를 풀어 나가는 공식은 다음과 같음

$$y = \frac{1}{1 + e^{-(ax+b)}}$$

- 이 식을 통해 알 수 있는 것은 우리가 구해야 하는 값이 여기서도 결국  $ax + b$ 라는 것

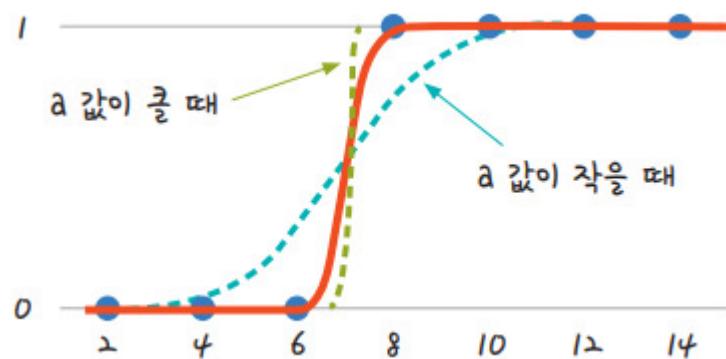


## 2 시그모이드 함수

### ● 시그모이드 함수

- 여기서  $a$ 와  $b$ 는 무슨 의미를 가지고 있을까?
- 먼저  $a$ 는 그래프의 경사도를 결정
- 그림 6-3과 같이  $a$  값이 커지면 경사가 커지고  $a$  값이 작아지면 경사가 작아짐

### ▼ 그림 6-3 | $a$ 값이 클 때와 작을 때의 그래프 변화



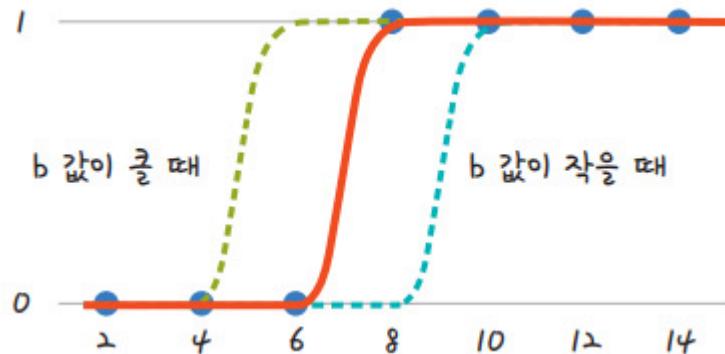


## 2 시그모이드 함수

### ● 시그모이드 함수

- b는 그래프의 좌우 이동을 의미
- 그림 6-4와 같이 b 값이 크고 작아짐에 따라 그래프가 이동

### ▼ 그림 6-4 | b 값이 클 때와 작을 때의 그래프 변화





## 2 시그모이드 함수

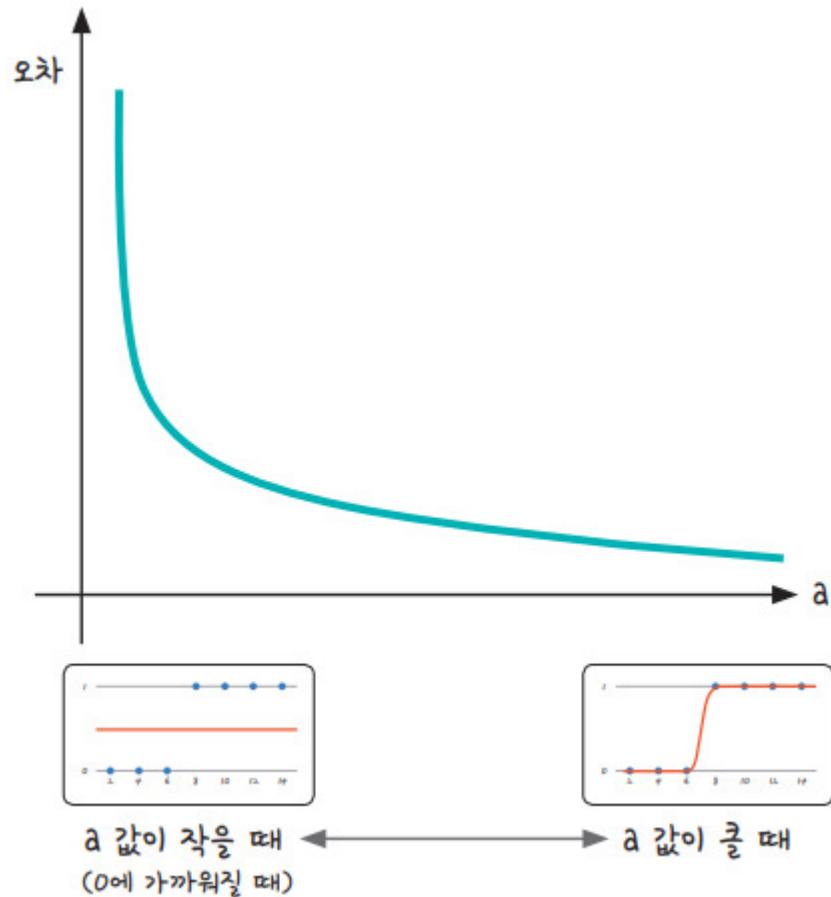
### ● 시그모이드 함수

- a 값과 b 값에 따라 오차가 변함
- a 값에 따라 변화하는 오차를 그래프로 나타내면 그림 6-5와 같음



## 2 시그모이드 함수

- ▼ 그림 6-5 |  $a$ 와 오차의 관계:  $a$ 가 작아질수록 오차는 무한대로 커지지만,  $a$ 가 커진다고 해서 오차가 없어지지는 않는다





## 2 시그모이드 함수

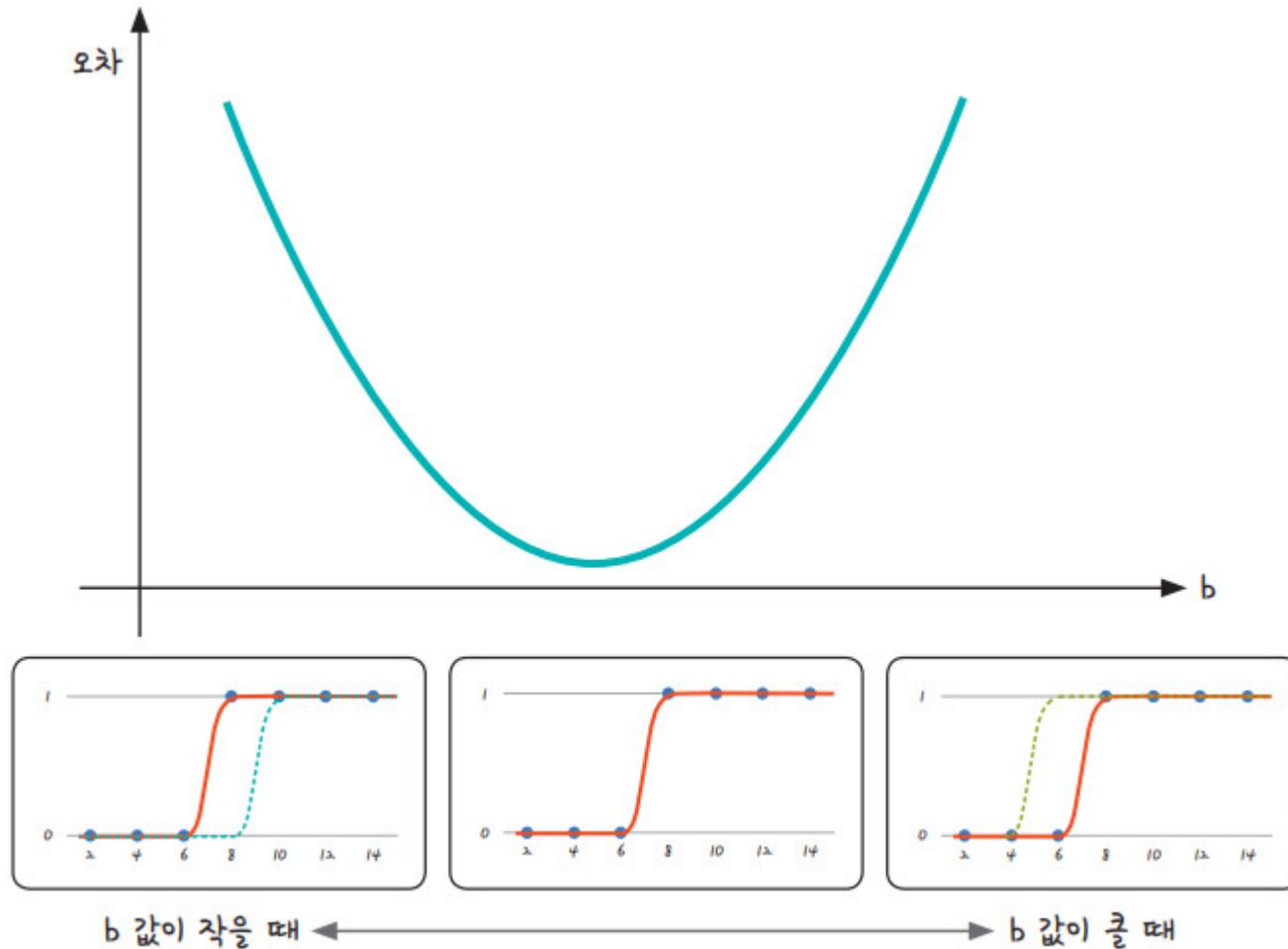
### ● 시그모이드 함수

- $a$  값이 작아지면 오차는 무한대로 커짐
- $a$  값이 커진다고 해서 오차가 무한대로 커지지는 않음



## 2 시그모이드 함수

▼ 그림 6-6 | b와 오차의 관계: b 값이 너무 작아지거나 커지면 오차도 이에 따라 커진다





## 2 시그모이드 함수

### ● 시그모이드 함수

- b 값이 너무 크거나 작을 경우 오차는 그림 6-6과 같이 이차 함수 그래프와 유사한 형태로 나타남



## 3 오차 공식

---



# 3 오차 공식

## ● 오차 공식

- 즉, 주어진 과제는 또다시  $a$  값과  $b$  값을 구하는 것
- 시그모이드 함수에서  $a$  값과  $b$  값을 어떻게 구해야 할까?
- 답은 역시 경사 하강법
- 경사 하강법은 먼저 오차를 구한 후 오차가 작은 쪽으로 이동시키는 방법이라고 했음
- 이번에도 예측 값과 실제 값의 차이, 즉 오차를 구하는 공식이 필요함
- 앞서 배웠던 평균 제곱 오차를 사용하면 될까?
- 안타깝게도 평균 제곱 오차를 사용할 수 없음

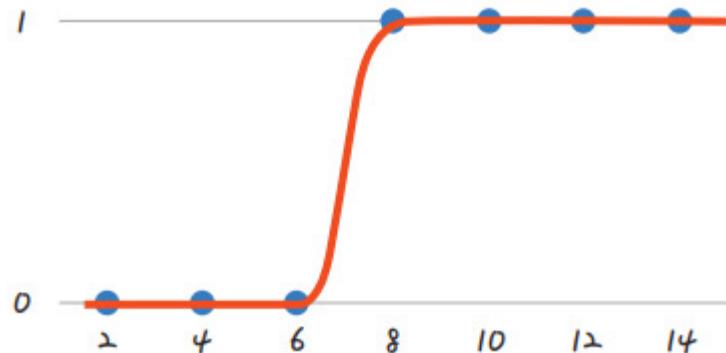


### 3 오차 공식

#### ● 오차 공식

- 오차 공식을 도출하기 위해 시그모이드 함수 그래프의 특징을 다시 한 번 살펴보자

#### ▼ 그림 6-7 | 시그모이드 함수 그래프



- 시그모이드 함수의 특징은  $y$  값이 0과 1 사이라는 것
- 실제 값이 1일 때 예측 값이 0에 가까워지면 오차가 커짐
- 반대로 실제 값이 0일 때 예측 값이 1에 가까워지는 경우에도 오차는 커짐
- 이를 공식으로 만들 수 있게 하는 함수가 바로 로그 함수



## 4 텐서플로에서 실행하는 로지스틱 회귀 모델



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

- 텐서플로에서 실행하는 로지스틱 회귀 모델

## 실습 | 텐서플로에서 실행하는 다중 선형 회귀



```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

x = np.array([2, 4, 6, 8, 10, 12, 14])
y = np.array([0, 0, 0, 1, 1, 1, 1])

model = Sequential()
model.add(Dense(1, input_dim=1, activation='sigmoid'))
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

```
# 교차 엔트로피 오차 함수를 이용하기 위해 'binary_crossentropy'로 설정합니다.  
model.compile(optimizer='sgd', loss='binary_crossentropy')  
model.fit(x, y, epochs=5000)  
  
# 그래프로 확인해 봅니다.  
plt.scatter(x, y)  
plt.plot(x, model.predict(x), 'r')  
plt.show()  
  
# 임의의 학습 시간을 집어넣어 합격 예상 확률을 예측해 보겠습니다.  
hour = 7  
prediction = model.predict([hour])  
print("%.f시간을 공부할 경우, 합격 예상 확률은 %.01f%%입니다." % (hour,  
prediction * 100))
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

실행 결과

Epoch 1/5000

1/1 [=====] - 0s 183ms/step - loss: 0.5741

... (중략) ...

Epoch 5000/5000

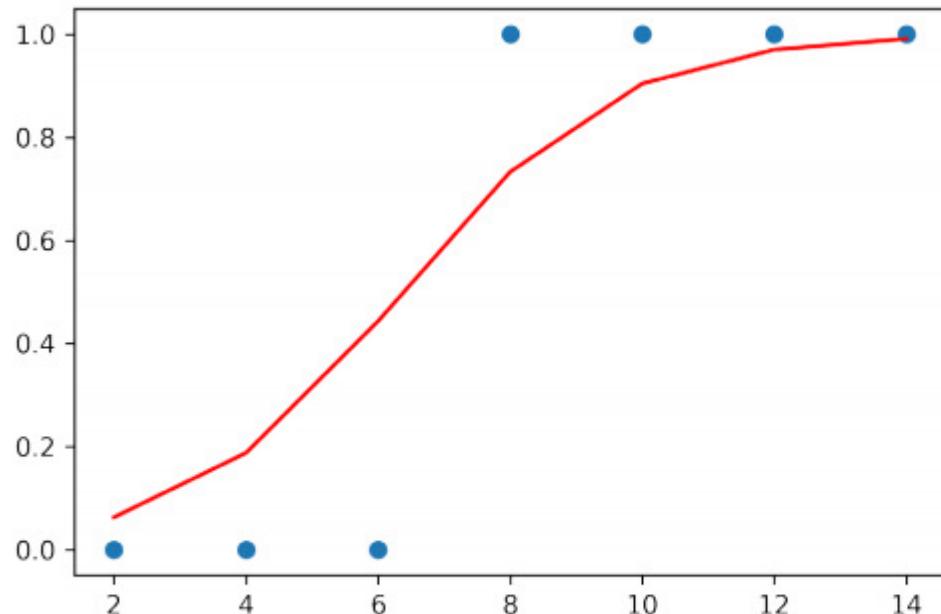
1/1 [=====] - 0s 1ms/step - loss: 0.1882



## 4 텐서플로에서 실행하는 로지스틱 회귀 모델

- 텐서플로에서 실행하는 로지스틱 회귀 모델

▼ 그림 6-9 | 실행 결과



7시간을 공부할 경우, 합격 예상 확률은 59.7%입니다.



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- 출력되는 그래프는 학습이 진행됨에 따라 점점 시그모이드 함수 그래프의 형태를 취해 가는 것을 보여 줌
- 학습된 모델의 테스트를 위해 여러 가지 임의의 시간을 집어넣고 테스트해 보면, 학습 시간이 7보다 클 경우에는 합격 확률이 50%가 넘는다는 것을 알 수 있음
- 데이터양이나 학습 시간 등 환경에 따라 예측 정확도는 더욱 향상될 수 있음



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- 텐서플로에서 실행하는 방법은 앞서 선형 회귀 모델을 만들 때와 유사함
- 다른 점은 오차를 계산하기 위한 손실 함수가 평균 제곱 오차 함수에서 크로스 엔트로피 오차로 바뀐다는 것
- 먼저 표 6-1에서 본 합격 여부 데이터를 다음과 같이 만들어 주자

```
x = np.array([2, 4, 6, 8, 10, 12, 14])
y = np.array([0, 0, 0, 1, 1, 1, 1])
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- 이제 모델을 준비
- 먼저 시그모이드 함수를 사용하게 되므로 activation 부분을 sigmoid로 바꾸어 줌

```
model.add(Dense(1, input_dim=1, activation='sigmoid'))
```

- 손실 함수로 교차 엔트로피 오차 함수를 이용하기 위해 loss를 binary\_crossentropy로 설정

```
model.compile(optimizer='sgd', loss='binary_crossentropy')
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- model.predict(x) 함수를 이용해 학습 시간 x가 입력되었을 때의 결과를 그래프로 그려 보면 다음과 같음

```
plt.scatter(x, y)
plt.plot(x, model.predict(x), 'r')
plt.show()
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- 임의의 학습 시간에 따른 합격 확률을 보여 주는 부분을 다음과 같이 설정

```
hour = 7  
prediction = model.predict([hour])  
print("%.f시간을 공부할 경우, 합격 예상 확률은 %.01f%%입니다." % (hour,  
prediction * 100))
```



# 4 텐서플로에서 실행하는 로지스틱 회귀 모델

## ● 텐서플로에서 실행하는 로지스틱 회귀 모델

- 지금까지 선형 회귀와 로지스틱 회귀를 사용한 모델링에 관해 알아보았음
- 이 두 가지가 딥러닝의 기본 요소가 된다는 것은 이미 설명한 바 있음
- 이러한 통계 모델링은 어떻게 해서 딥러닝과 연관을 갖게 될까?
- 로지스틱 회귀 모델의 전신인 퍼셉트론과 퍼셉트론의 한계를 극복하며 탄생한 신경망에 대해 상세히 알아보며 딥러닝 학습 속도를 더해 보자



# 딥러닝의 시작, 신경망

# 퍼셉트론과 인공지능의 시작

---

1 인공지능의 시작을 알린 퍼셉트론

2 퍼셉트론의 과제

3 XOR 문제



# 1 인공지능의 시작을 알린 퍼셉트론

---



# 1 인공지능의 시작을 알린 퍼셉트론

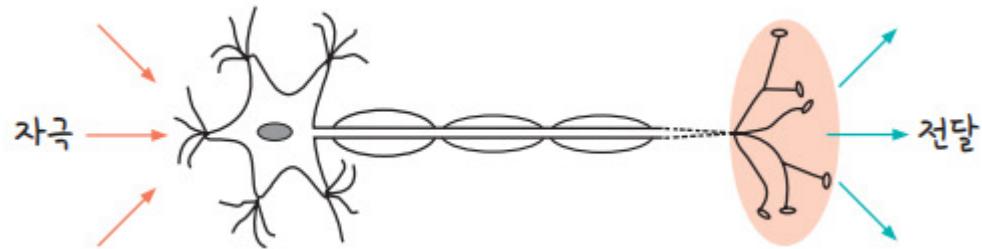
## ● 인공지능의 시작을 알린 퍼셉트론

- 인간의 뇌는 치밀하게 연결된 뉴런 약 1,000억 개로 이루어져 있음
- 뉴런과 뉴런 사이에는 시냅스라는 연결 부위가 있는데, 신경 말단에서 자극을 받으면 시냅스에서 화학 물질이 나와 전위 변화를 일으키고, 전위가 임계 값을 넘으면 다음 뉴런으로 신호를 전달하고, 임계 값에 미치지 못하면 아무것도 하지 않음
- 이 메커니즘은 앞서 배운 로지스틱 회귀와 많이 닮았음
- 이 간단한 회로는 입력 값을 놓고 활성화 함수에 의해 일정한 수준을 넘으면 참을, 그렇지 않으면 거짓을 내보내는 일을 하는데 뉴런과 유사함



# 1 인공지능의 시작을 알린 퍼셉트론

▼ 그림 7-1 | 뉴런의 신호 전달





# 1 인공지능의 시작을 알린 퍼셉트론

## ● 인공지능의 시작을 알린 퍼셉트론

- 우리 몸 안에 있는 수많은 뉴런은 서로 긴밀히 연결되어 신경 말단부터 뇌까지 곳곳에서 자신의 역할을 수행
- 이처럼 복잡하고 어려운 조합의 결과가 바로 우리의 ‘생각’
- 뉴런과 비슷한 메커니즘을 사용하면 인공적으로 ‘생각’하는 그 무언가를 만들 수 있지 않을까?
  
- 이러한 상상과 함께 출발한 연구가 바로 인공 신경망(artificial neural network, 이하 줄여서 ‘신경망’이라고 함) 연구
- 맨 처음 시작은 ‘켜고 끄는 기능이 있는 신경’을 그물망 형태로 연결하면 사람의 뇌처럼 동작할 수 있다는 가능성을 처음으로 주장한 맥컬럭–월터 피츠(McCulloch–Walter Pitts)의 1943년 논문
- 그 후 1957년, 미국의 신경 생물학자 프랑크 로젠블랫(Frank Rosenblatt)이 이 개념을 실제 장치로 만들어 선보임
- 이것의 이름이 퍼셉트론(perceptron)



# 1 인공지능의 시작을 알린 퍼셉트론

## ● 인공지능의 시작을 알린 퍼셉트론

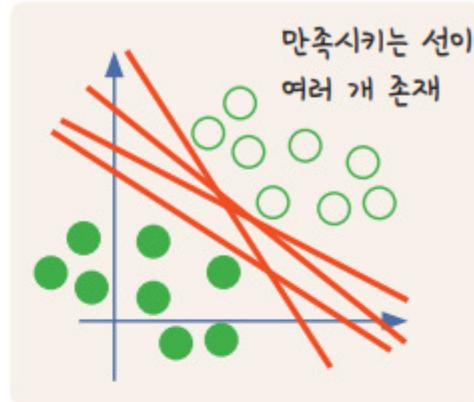
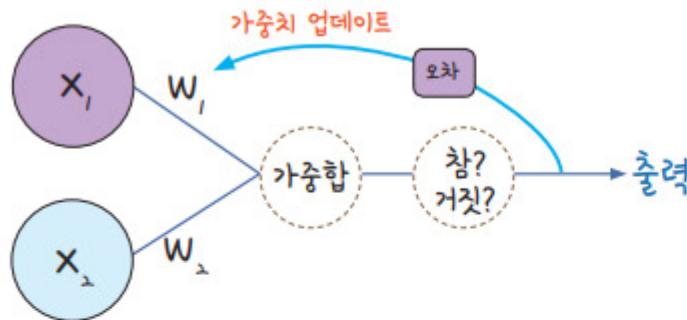
- 퍼셉트론은 입력 값을 여러 개 받아 출력을 만드는데, 이때 입력 값에 가중치를 조절할 수 있게 만들어 최초로 ‘학습’을 하게 했음(그림 7-2 ① 참조)
- 3년 후, 여기에 앞에서 다룬 경사 하강법을 도입해 최적의 경계선을 그릴 수 있게 한 아달라인(Adaline)이 개발(그림 7-2 ② 참조)
- 특히 아달라인은 이후 서포트 벡터 머신(Support Vector Machine) 등 머신 러닝의 중요한 알고리즘들로 발전해 가는데, 이 중 시그모이드를 활성화 함수로 사용한 것이 바로 앞서 배웠던 로지스틱 회귀



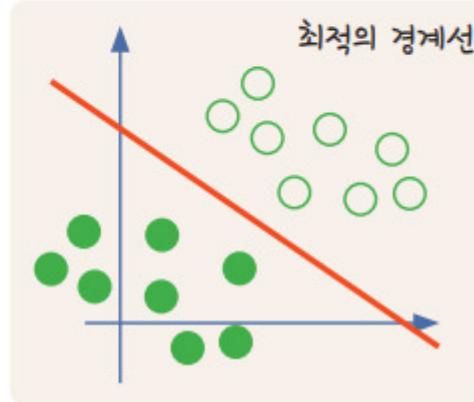
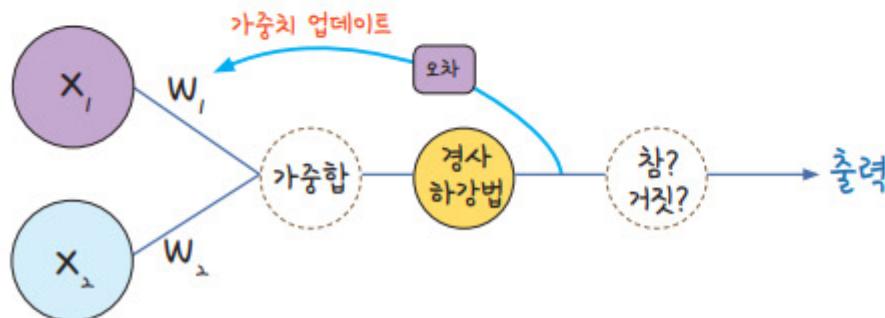
# 1 인공지능의 시작을 알린 퍼셉트론

▼ 그림 7-2 | 퍼셉트론, 아달라인 그리고 로지스틱 회귀 모델

## ① 퍼셉트론



## ② 아달라인



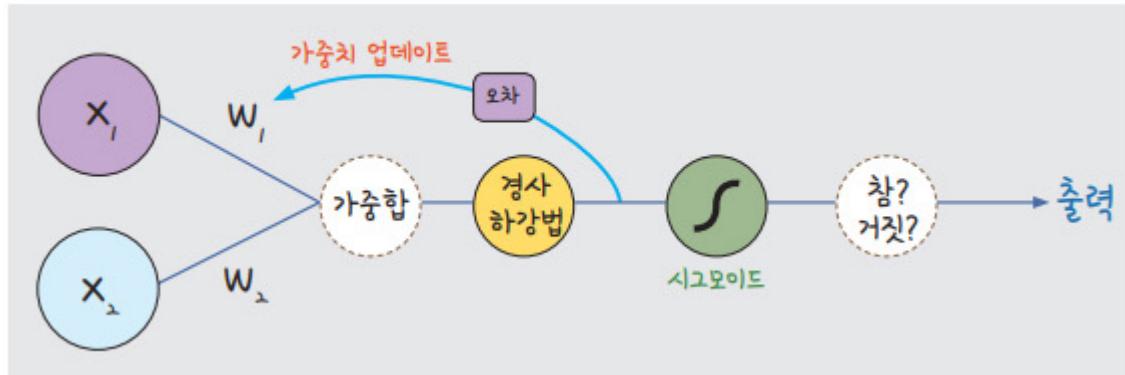


# 1 인공지능의 시작을 알린 퍼셉트론

## ● 인공지능의 시작을 알린 퍼셉트론

- 가중합(weighted sum)이란 입력 값과 가중치를 모두 곱한 후 바이어스를 더한 값을 의미

(비교) 로지스틱 회귀 모델





## 2 퍼셉트론의 과제

---



## 2 퍼셉트론의 과제

### ● 퍼셉트론의 과제

- 퍼셉트론이 완성되고 아달라인에 의해 보완되며 드디어 현실 세계의 다양한 문제를 해결하는 인공지능이 개발될 것으로 기대했지만, 퍼셉트론의 한계가 보고
- 퍼셉트론의 한계가 무엇이었는지 알고 이를 극복하는 과정을 이해하는 것도 매우 중요함
- 이것을 해결한 것이 바로 딥러닝이기 때문임
- 지금부터는 퍼셉트론의 한계와 이를 해결하는 과정을 보며 신경망의 기본 개념을 확립해 보자

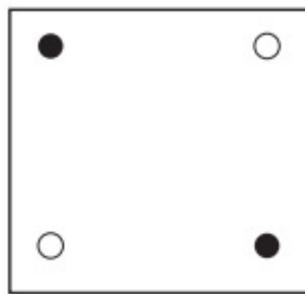


## 2 퍼셉트론의 과제

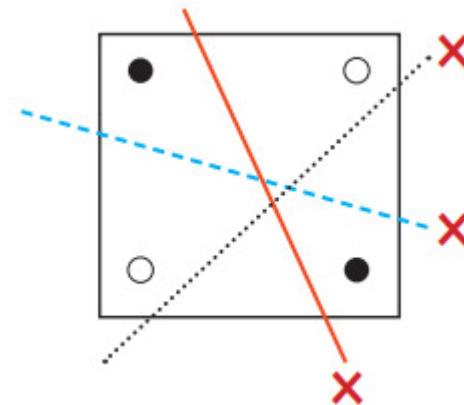
### ● 퍼셉트론의 과제

- 사각형 종이에 검은색 점 두 개와 흰색 점 두 개가 놓여 있음
- 이 네 점 사이에 직선을 하나 긋는다고 하자
- 이때 직선의 한쪽 편에는 검은색 점만 있고, 다른 한쪽에는 흰색 점만 있게끔 선을 그을 수 있을까?

▼ 그림 7-3 | 사각형 종이에  
놓인 검은색 점 두 개와 흰색 점  
두 개



▼ 그림 7-4 | 선으로는 같은 색끼리 나눌 수 없다:  
퍼셉트론의 한계





## 2 퍼셉트론의 과제

### ● 퍼셉트론의 과제

- 선을 여러 개 아무리 그어 보아도 하나의 직선으로는 흰색 점과 검은색 점을 구분할 수 없음
- 퍼셉트론이나 아달라인은 모두 2차원 평면상에 직선을 긋는 것만 가능함
- 이 예시는 경우에 따라 선을 아무리 그어도 해결되지 않는 상황이 있다는 것을 말해 줌



## 3 XOR 문제

---



# 3 XOR 문제

## ● XOR 문제

- 퍼셉트론의 한계를 설명할 때 등장하는 XOR(exclusive OR) 문제
- XOR 문제는 논리 회로에 등장하는 개념
- 컴퓨터는 두 가지의 디지털 값, 즉 0과 1을 입력해 하나의 값을 출력하는 회로가 모여 만들어지는데, 이 회로를 ‘게이트(gate)’라고 함
- 그림 7-5는 AND 게이트, OR 게이트, XOR 게이트에 대한 값을 정리한 것
- AND 게이트는  $x_1$ 과  $x_2$  둘 다 1일 때 결괏값이 1로 출력
- OR 게이트는 둘 중 하나라도 1이면 결괏값이 1로 출력
- XOR 게이트는 둘 중 하나만 1일 때 1이 출력



# 3 XOR 문제

▼ 그림 7-5 | AND, OR, XOR 게이트에 대한 진리표

AND  
(논리곱)  
두 개 모두 1일 때 1

$x_1$	$x_2$	결과값
0	0	0
0	1	0
1	0	0
1	1	1

OR  
(논리합)  
두 개 중 한 개라도 1이면 1

$x_1$	$x_2$	결과값
0	0	0
0	1	1
1	0	1
1	1	1

XOR  
(배타적 논리합)  
하나만 1이어야 1

$x_1$	$x_2$	결과값
0	0	0
0	1	1
1	0	1
1	1	0

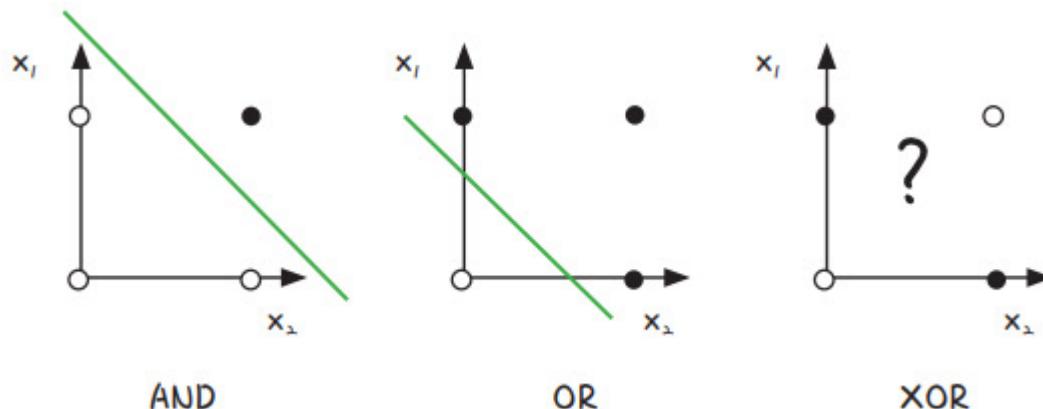


# 3 XOR 문제

## ● XOR 문제

- 그림 7-5를 각각 그래프로 좌표 평면에 나타내 보자
- 결과값이 0이면 흰색 점으로, 1이면 검은색 점으로 나타낸 후 조금 전처럼 직선을 그어 위 조건을 만족할 수 있는지 보자

▼ 그림 7-6 | AND, OR, XOR 진리표대로 좌표 평면에 표현한 후 선을 그어 색이 같은 점끼리 나누기(XOR는 불가능)





# 3 XOR 문제

## ● XOR 문제

- AND와 OR 게이트는 직선을 그어 결괏값이 1인 값(검은색 점)을 구별할 수 있음
- XOR의 경우 선을 그어 구분할 수 없음
- 이는 인공지능 분야의 선구자였던 MIT의 마빈 민스키(Marvin Minsky) 교수가 1969년에 발표한 “퍼셉트론즈(Perceptrons)”라는 논문에 나오는 내용
- ‘뉴런 → 신경망 → 지능’이라는 도식에 따라 ‘퍼셉트론 → 인공 신경망 → 인공지능’이 가능하리라 꿈꾸었던 당시 사람들은 이것이 생각처럼 쉽지 않다는 사실을 깨닫게 됨
- 알고 보니 간단한 XOR 문제조차 해결할 수 없었던 것
- 이 논문 이후 인공지능 연구가 한동안 침체기를 겪게 됨
- 이 문제는 두 가지 방법이 순차적으로 개발되면서 해결
- 하나는 다층 퍼셉트론(multilayer perceptron), 그리고 또 하나는 오차 역전파(back propagation)

# 다층 퍼셉트론

---

- 1 다층 퍼셉트론의 등장
- 2 다층 퍼셉트론의 설계
- 3 XOR 문제의 해결
- 4 코딩으로 XOR 문제 해결하기



# 1 다층 퍼셉트론의 등장

---

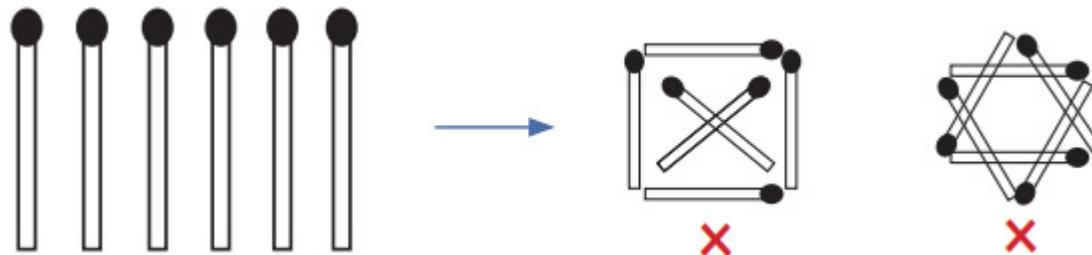


# 1 다층 퍼셉트론의 등장

## ● 다층 퍼셉트론의 등장

- 앞서 종이 위에 각각 엇갈려 놓인 검은색 점 두 개와 흰색 점 두 개를 하나의 선으로는 구별할 수 없다는 것을 살펴보았음
- 언뜻 보기에도 해답이 없어 보이는 이 문제를 해결하려면 새로운 접근이 필요함
- 예 : ‘성냥개비 여섯 개로 정삼각형 네 개를 만들 수 있는가’라는 문제를 기억하나요?

### ▼ 그림 8-1 | 성냥개비 여섯 개로 정삼각형 네 개를?



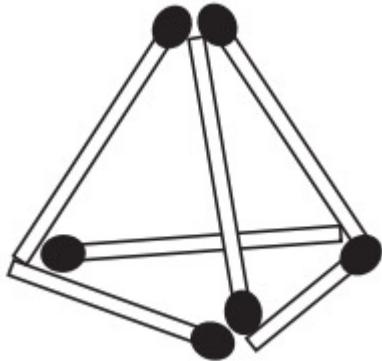


# 1 다층 퍼셉트론의 등장

## ● 다층 퍼셉트론의 등장

- 골똘히 연구해도 답을 찾지 못했던 이 문제는 2차원 평면에서만 해결하려는 고정 관념을 깨고 피라미드 모양으로 성냥개비를 쌓아 올리니 해결

## ▼ 그림 8-2 | 차원을 달리하니 쉽게 완성!



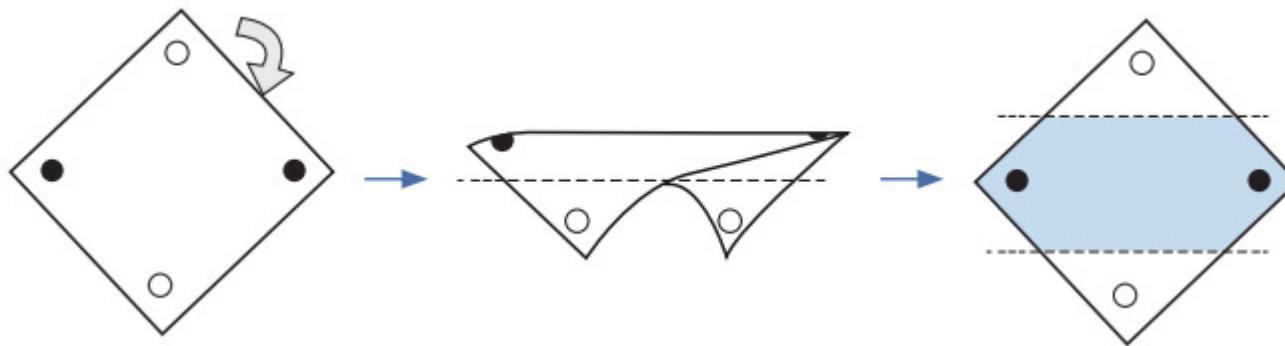


# 1 다층 퍼셉트론의 등장

## ● 다층 퍼셉트론의 등장

- 인공지능 학자들은 인공 신경망을 개발하기 위해 반드시 XOR 문제를 극복해야만 했음
- 이 문제를 해결하는 데도 고정 관념을 깨는 기발한 아이디어가 필요했음
- 이러한 노력은 결국 그림 8-3과 같은 아이디어를 낳았음

▼ 그림 8-3 | XOR 문제의 해결은 평면을 휘어 주는 것!





# 1 다층 퍼셉트론의 등장

## ● 다층 퍼셉트론의 등장

- 즉, 종이를 휘어 주어 선 두 개를 동시에 굿는 방법
- 이것을 XOR 문제에 적용하면 ‘퍼셉트론 두 개를 한 번에 계산’하면 된다는 결론에 이를
- 이를 위해 퍼셉트론 두 개를 각각 처리하는 은닉층(hidden layer)을 만듦
- 은닉층을 만드는 것이 어떻게 XOR 문제를 해결하는지는 그림 8-4에 소개되어 있음



# 1 다층 퍼셉트론의 등장

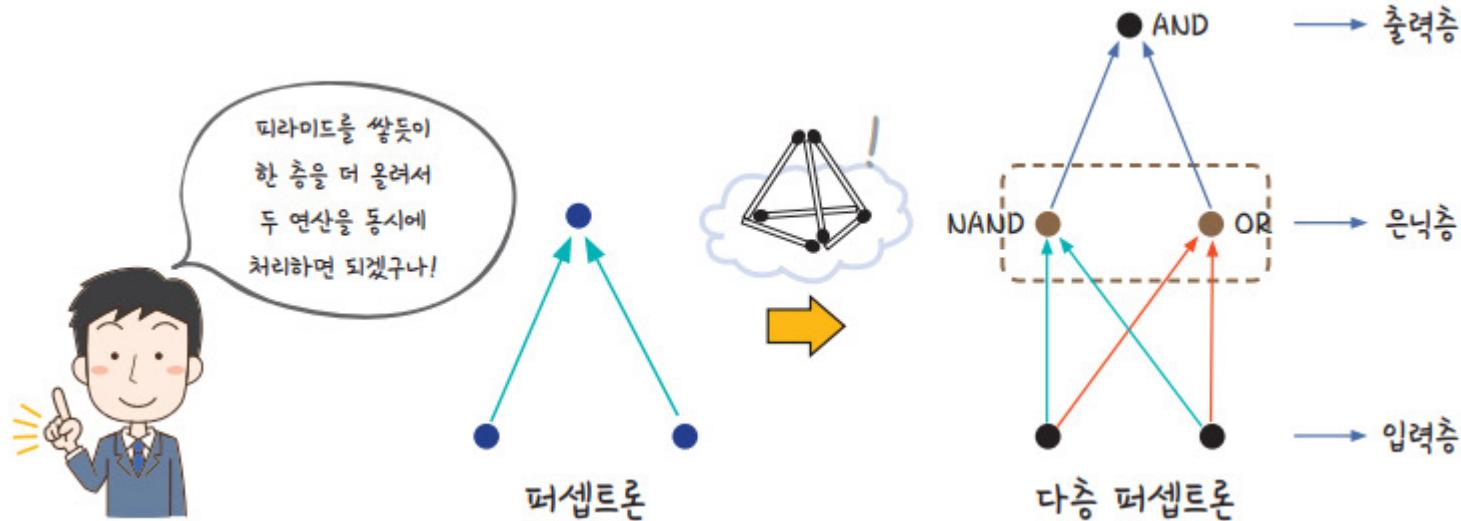
▼ 그림 8-4 | 은닉층이 XOR 문제를 해결하는 원리

XOR (배라적 논리합) 하나만 1이어야 /		② NAND (부정 논리곱) 하나라도 0이면 /		③ OR (논리합) 두 개 중 한 개라도 1이면 /		④ AND (논리곱) 두 개 모두 1일 때 /									
$x_1$	$x_2$	결과값		$x_1$	$x_2$	결과값 1		$x_1$	$x_2$	결과값 2		$x_1$	$x_2$	결과값	
0	0	0		0	0	1		0	0	0		1	0	1	
0	1	1		0	1	1		0	1	1		1	1	1	
1	0	1		1	0	1		1	0	1		1	1	1	
1	1	0		1	1	0		1	1	1		0	1	0	

①



# 1 다층 퍼셉트론의 등장





# 1 다층 퍼셉트론의 등장

## ● 다층 퍼셉트론의 등장

- 먼저 ①  $x_1$ 과  $x_2$ 를 두 연산으로 각각 보냄
- ② 첫 번째 연산에서는 NAND 처리를 함
- ③ 이와 동시에 두 번째 연산에서 OR 처리를 함
- ② 와 ③을 통해 구한 결괏값 1과 결괏값 2를 가지고 ④ AND 처리를 하면 우리가 구하고자 하는 출력 값을 만들 수 있음



## 2 다층 퍼셉트론의 설계

---

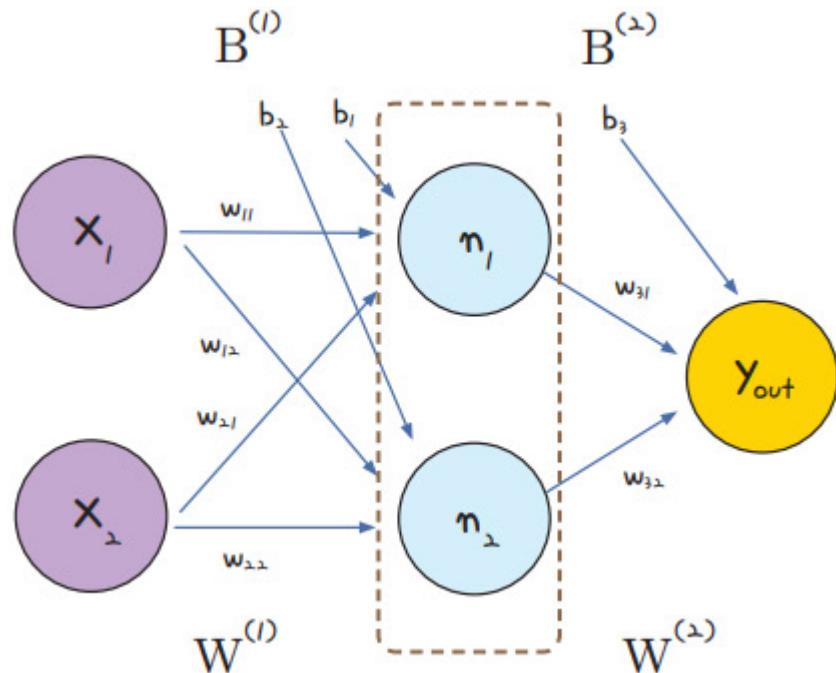


## 2 다층 퍼셉트론의 설계

### ● 다층 퍼셉트론의 설계

- 다층 퍼셉트론이 입력층과 출력층 사이에 숨어 있는 은닉층을 만드는 것을 그림으로 나타내면 그림 8-5와 같음

### ▼ 그림 8-5 | 다층 퍼셉트론의 내부





## 2 다층 퍼셉트론의 설계

### ● 다층 퍼셉트론의 설계

- 가운데 점선으로 표시된 부분이 은닉층
- $x_1$ 과  $x_2$ 는 입력 값인데, 각 값에 가중치( $w$ )를 곱하고 바이어스( $b$ )를 더해 은닉층으로 전송
- 이 값들이 모이는 은닉층의 중간 정거장을 노드(node)라고 하며, 여기서는  $n_1$ 과  $n_2$ 로 표시
- 은닉층에 취합된 값들은 활성화 함수를 통해 다음으로 보내는데, 만약 앞서 배운 시그모이드 함수( $\sigma(x)$ )를 활성화 함수로 사용한다면  $n_1$ 과  $n_2$ 에서 계산되는 값은 각각 다음과 같음

$$n_1 = \sigma(x_1 w_{11} + x_2 w_{21} + b_1)$$

$$n_2 = \sigma(x_1 w_{12} + x_2 w_{22} + b_2)$$

- 두식의 결괏값이 출력층의 방향으로 보내어지고, 출력층으로 전달된 값은 마찬가지로 활성화 함수를 사용해  $y$  예측 값을 정하게 됨
- 이 값을  $y_{\text{out}}$ 이라고 할 때 이를 식으로 표현하면 다음과 같음

$$y_{\text{out}} = \sigma(n_1 w_{31} + n_2 w_{32} + b_3)$$



## 2 다층 퍼셉트론의 설계

### ● 다층 퍼셉트론의 설계

- 이제 각각의 가중치(w)와 바이어스(b) 값을 정할 차례
- 2차원 배열로 늘어놓으면 다음과 같이 표시할 수 있음
- 은닉층을 포함해 가중치 여섯 개와 바이어스 세 개가 필요함

$$W^{(1)} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \quad B^{(1)} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$
$$W^{(2)} = \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix} \quad B^{(2)} = [b_3]$$



## 3 XOR 문제의 해결

---



# 3 XOR 문제의 해결

## ● XOR 문제의 해결

- 앞서 어떤 가중치와 바이어스가 필요한지 알아보았음
- 이를 만족하는 가중치와 바이어스의 조합은 무수히 많음
- 지금은 먼저 다음과 같이 각 변수 값을 정하고 이를 이용해 XOR 문제를 해결하는 과정을 알아보자

$$W^{(1)} = \begin{bmatrix} -2 & 2 \\ -2 & 2 \end{bmatrix} \quad B^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad B^{(2)} = [-1]$$

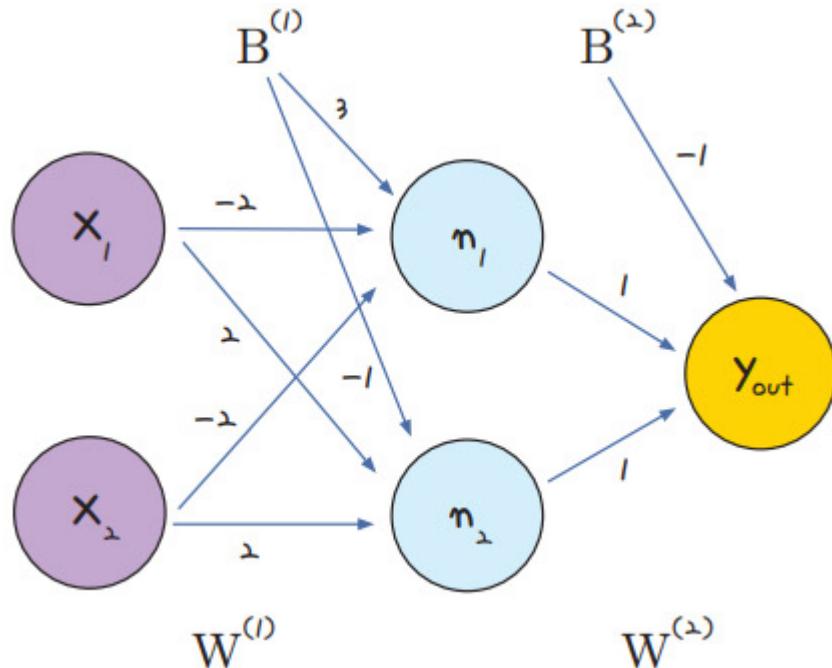


# 3 XOR 문제의 해결

## ● XOR 문제의 해결

- 가중치와 바이어스 조합을 대입하면 그림 8-6과 같음

### ▼ 그림 8-6 | 다층 퍼셉트론의 내부에 변수 채우기





# 3 XOR 문제의 해결

## ● XOR 문제의 해결

- 이제  $x_1$  값과  $x_2$  값을 각각 입력해 우리가 원하는  $y$  값이 나오는지 점검해 보자

▼ 표 8-1 | XOR 다층 문제 해결

$x_1$	$x_2$	$n_1$	$n_2$	$y_{out}$	우리가 원하는 값
0	0	$\sigma(0 * (-2) + 0 * (-2) + 3) \approx 1$	$\sigma(0 * 2 + 0 * 2 - 1) \approx 0$	$\sigma(1 * 1 + 0 * 1 - 1) \approx 0$	0
0	1	$\sigma(0 * (-2) + 1 * (-2) + 3) \approx 1$	$\sigma(0 * 2 + 1 * 2 - 1) \approx 1$	$\sigma(1 * 1 + 1 * 1 - 1) \approx 1$	1
1	0	$\sigma(1 * (-2) + 0 * (-2) + 3) \approx 1$	$\sigma(1 * 2 + 0 * 2 - 1) \approx 1$	$\sigma(1 * 1 + 1 * 1 - 1) \approx 1$	1
1	1	$\sigma(1 * (-2) + 1 * (-2) + 3) \approx 0$	$\sigma(1 * 2 + 1 * 2 - 1) \approx 1$	$\sigma(0 * 1 + 1 * 1 - 1) \approx 0$	0

- ≈ 기호는 ‘거의 같다’를 의미
- 표 8-1에서 볼 수 있듯이  $n_1$ ,  $n_2$ ,  $y$ 를 구하는 공식에 차례로 대입하니 우리가 원하는 결과를 구할 수 있었음
- 숨어 있는 노드 두 개를 둔 다층 퍼셉트론을 통해 XOR 문제가 해결된 것



## 4 코딩으로 XOR 문제 해결하기

---



# 4 코딩으로 XOR 문제 해결하기

- 코딩으로 XOR 문제 해결하기

## 실습 | 다층 퍼셉트론으로 XOR 문제 해결하기



```
import numpy as np

# 가중치와 바이어스
w11 = np.array([-2, -2])
w12 = np.array([2, 2])
w2 = np.array([1, 1])
b1 = 3
b2 = -1
b3 = -1
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

```
# 퍼셉트론
def MLP(x, w, b):
    y = np.sum(w * x) + b
    if y <= 0:
        return 0
    else:
        return 1

# NAND 게이트
def NAND(x1, x2):
    return MLP(np.array([x1, x2]), w11, b1)
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

```
# OR 게이트
def OR(x1, x2):
    return MLP(np.array([x1, x2]), w12, b2)

# AND 게이트
def AND(x1, x2):
    return MLP(np.array([x1, x2]), w2, b3)

# XOR 게이트
def XOR(x1, x2):
    return AND(NAND(x1, x2), OR(x1, x2))
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

```
# x1 값, x2 값을 번갈아 대입하며 최종 값 출력
for x in [(0, 0), (1, 0), (0, 1), (1, 1)]:
    y = XOR(x[0], x[1])
    print("입력 값: " + str(x) + " 출력 값: " + str(y))
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

### 실행 결과

입력 값: (0, 0) 출력 값: 0

입력 값: (1, 0) 출력 값: 1

입력 값: (0, 1) 출력 값: 1

입력 값: (1, 1) 출력 값: 0



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

- 이제 주어진 가중치와 바이어스를 이용해 XOR 문제를 해결하는 파이썬 코드를 작성
- 정해진 가중치와 바이어스를 넘파이 라이브러리를 사용

```
import numpy as np

w11 = np.array([-2, -2])
w12 = np.array([2, 2])
w2 = np.array([1, 1])
b1 = 3
b2 = -1
b3 = -1
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

- 퍼셉트론 함수를 만들어 줌
- 0과 1 중에서 값을 출력하게 설정

```
def MLP(x, w, b):  
    y = np.sum(w * x) + b  
    if y <= 0:  
        return 0  
    else:  
        return 1
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

- 각 게이트의 정의에 따라 NAND 게이트, OR 게이트, AND 게이트, XOR 게이트 함수를 만들어 줌

```
# NAND 게이트
def NAND(x1, x2):
    return MLP(np.array([x1, x2]), w11, b1)

# OR 게이트
def OR(x1, x2):
    return MLP(np.array([x1, x2]), w12, b2)

# AND 게이트
def AND(x1, x2):
    return MLP(np.array([x1, x2]), w2, b3)
```



# 4 코딩으로 XOR 문제 해결하기

- 코딩으로 XOR 문제 해결하기

```
# XOR 게이트
def XOR(x1, x2):
    return AND(NAND(x1, x2), OR(x1, x2))
```



# 4 코딩으로 XOR 문제 해결하기

## ● 코딩으로 XOR 문제 해결하기

- 이제  $x_1$  값과  $x_2$  값을 번갈아 대입해 가며 최종 값을 출력해 보자

```
for x in [(0, 0), (1, 0), (0, 1), (1, 1)]:  
    y = XOR(x[0], x[1])  
    print("입력 값: " + str(x) + " 출력 값: " + str(y))
```



# 4 코딩으로 XOR 문제 해결하기

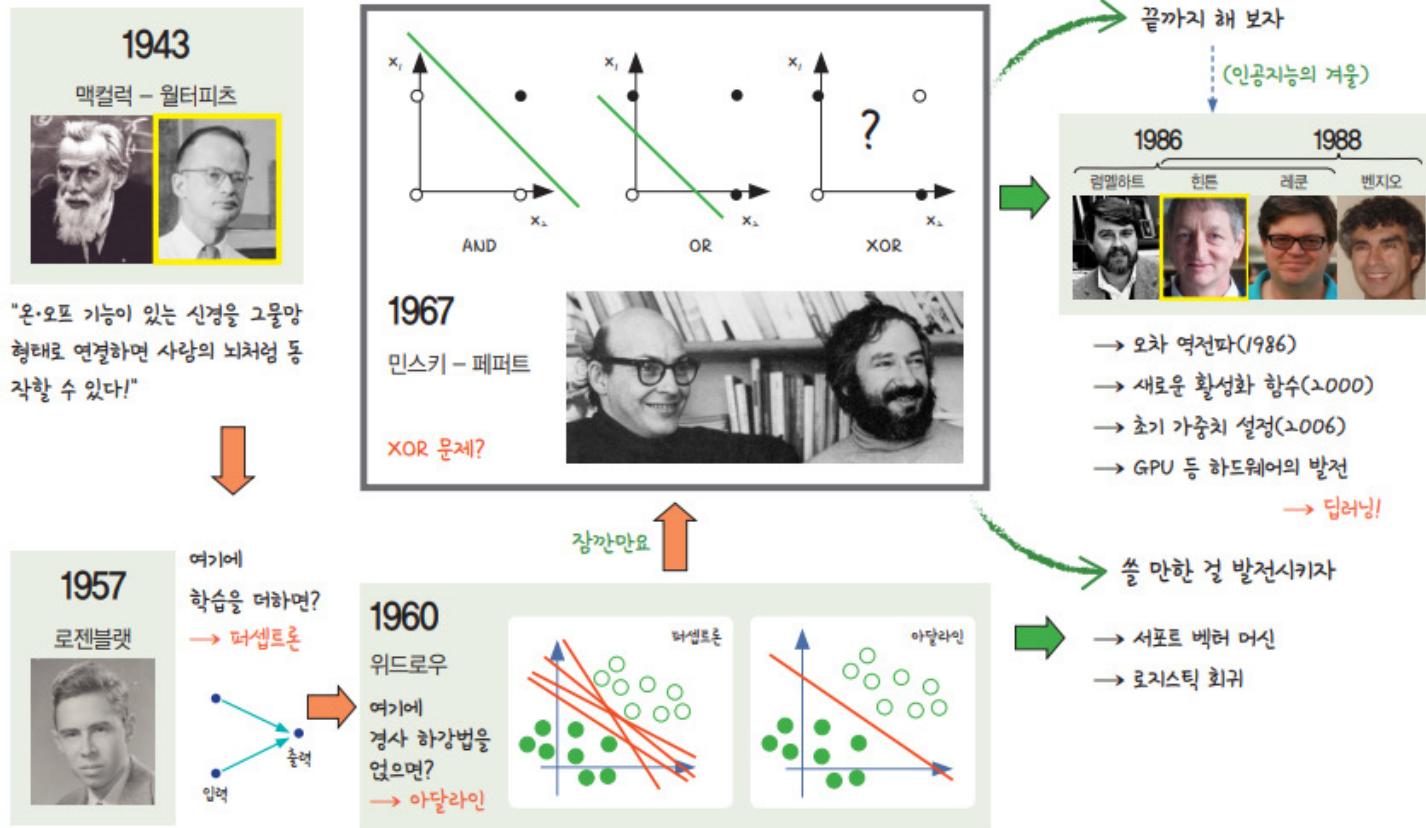
## ● 코딩으로 XOR 문제 해결하기

- XOR 문제의 정답이 도출
- 이렇게 퍼셉트론 하나로 해결되지 않던 문제를 은닉층을 만들어 해결했지만, 퍼셉트론의 문제가 완전히 해결된 것은 아니었음
- 다층 퍼셉트론을 사용할 경우 XOR 문제는 해결되었지만, 은닉층에 들어 있는 가중치를 데이터를 통해 학습하는 방법이 아직 없었기 때문에, 다층 퍼셉트론의 적절한 학습 방법을 찾기까지 그 후로 약 20여 년의 시간이 더 필요했음
- 이 기간을 지나며 데이터 과학자들은 두 부류로 나뉨
- 하나는 최적화된 예측선을 잘 그려 주던 아달라인을 발전시켜 SVM이나 로지스틱 회귀 모델을 만든 그룹
- 또 하나의 그룹은 여러 어려움 속에서도 끝까지 다층 퍼셉트론의 학습 방법을 찾던 그룹
- 이 두 번째 그룹에 속해 있던 제프리 힌튼(Geoffrey Hinton) 교수가 바로 딥러닝의 아버지로 칭송 받는 사람
- 힌튼 교수는 여러 가지 혁신적인 아이디어로 인공지능의 겨울을 극복해 냈음
- 첫 번째 아이디어는 1986년에 발표한 오차 역전파



# 4 코딩으로 XOR 문제 해결하기

## ▼ 그림 8-7 | 한눈에 보는 인공지능의 역사: 퍼셉트론에서 딥러닝까지



# 오차 역전파에서 딥러닝으로

---

- 1 딥러닝의 태동, 오차 역전파
- 2 활성화 함수와 고급 경사 하강법
- 3 속도와 정확도 문제를 해결하는 고급 경사 하강법



# 오차 역전파에서 딥러닝으로

## ● 오차 역전파에서 딥러닝으로

- 해결되지 않던 XOR 문제를 다층 퍼셉트론으로 해결했지만, 은닉층에 포함된 가중치를 업데이트할 방법이 없었음
- 이는 오차 역전파라는 방법을 만나고 나서야 해결
- 오차 역전파는 후에 지금 우리가 아는 딥러닝의 탄생으로 이어짐
  
- 이 수업에서는 오차 역전파의 개념을 이해하고 넘어가는 것
- 오차 역전파의 개념만 알아도 앞으로 배울 과정을 마치는 데는 아무런 문제가 없음
- 개념을 숙지하는 것을 목표로 학습할 분들은 이 장의 내용만 익히면 됨
- 반면, 딥러닝 알고리즘을 더 깊이 이해하고 싶다거나, 텐서플로 같은 자동화 라이브러리에 맡기지 않고 직접 코딩을 해야 한다면 오차 역전파의 계산법을 숙지하기 바람



# 1 딥러닝의 태동, 오차 역전파

---



# 1 딥러닝의 태동, 오차 역전파

## ● 딥러닝의 태동, 오차 역전파

- 앞서 XOR 문제를 해결했지만, 입력 값과 출력 값을 알고 있는 상태에서 가중치( $w$ )와 바이어스( $b$ )를 미리 알아본 후 이를 집어넣었음
- 이것은 ‘모델링’이라고 할 수 없음
- 둘째 마당의 회귀 모델에서 살펴본 바와 같이 우리가 원하는 것은 데이터를 통해 스스로 가중치를 조절하는 ‘학습’의 실현
- 오차 역전파 방법은 어떻게 해서 숨겨진 은닉층의 가중치를 업데이트할 수 있었을까?
- 이를 설명하기 위해 다시 경사 하강법을 알아보자.



# 1 딥러닝의 태동, 오차 역전파

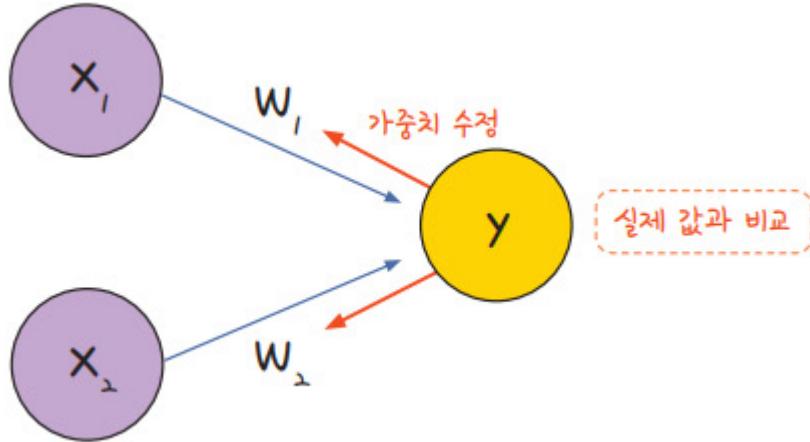
## ● 딥러닝의 태동, 오차 역전파

- 경사 하강법은 임의의 가중치를 선언하고 결괏값을 이용해 오차를 구한 후 이 오차가 최소인 지점으로 계속해서 조금씩 이동시키는 것
- 이 오차가 최소인 지점은 미분했을 때 기울기가 0이 되는 지점이라고 했음
- 지금 이야기하고 있는 경사 하강법은 ‘단일 퍼셉트론’, 즉 입력층과 출력층만 존재할 때 가능
- 은닉층이 생기면서 우리는 두 번의 경사 하강법을 실행해야 함
- 즉, 그림 9-1과 같이 가중치를 한 번 수정하면 되던 작업이 그림 9-2와 같이 가중치를 두 번 수정해야 하는 것

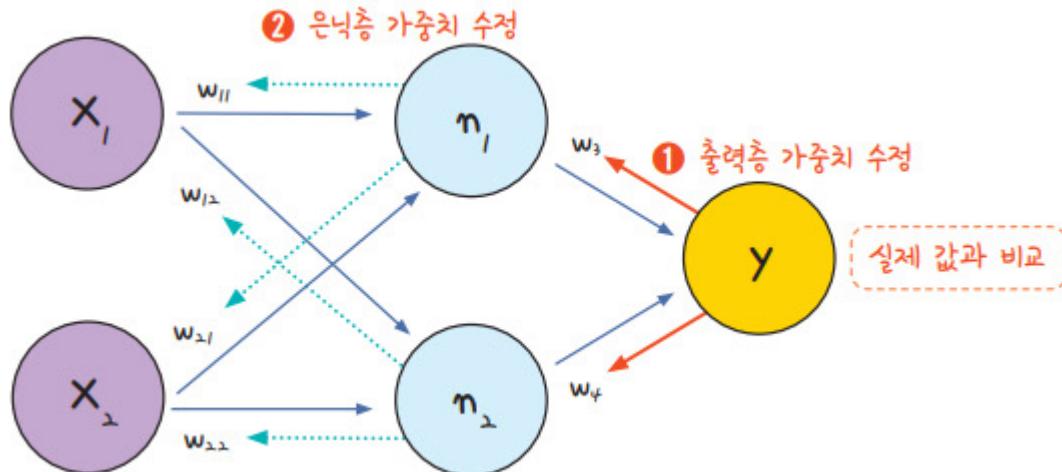


# 1 딥러닝의 태동, 오차 역전파

## ▼ 그림 9-1 | 단일 퍼셉트론에서 오차 수정



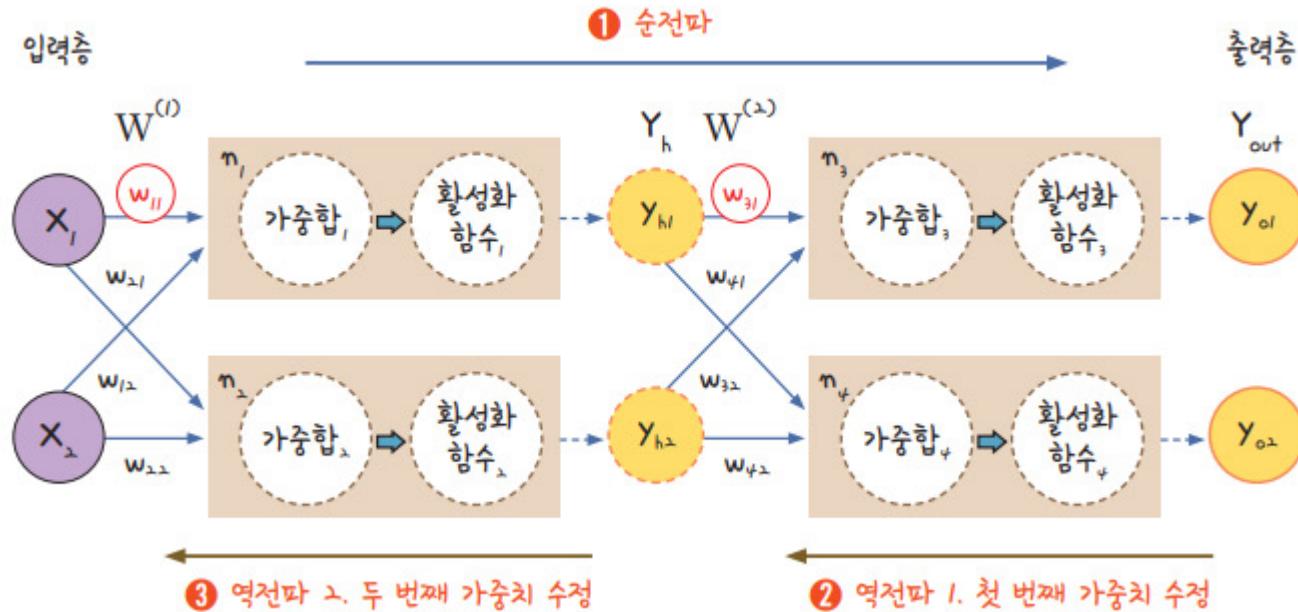
## ▼ 그림 9-2 | 다층 퍼셉트론에서 오차 수정





# 1 딥러닝의 태동, 오차 역전파

### ▼ 그림 9-3 | 오차 역전파의 개념





# 1 딥러닝의 태동, 오차 역전파

## ● 딥러닝의 태동, 오차 역전파

- 먼저 ①처럼 한 번의 순전파가 일어남
- 이 과정을 통해 각 가중치의 초기값이 정하고, 이 초기값의 가중치로 만들어진 값과 실제 값을 비교해 출력층의 오차를 계산
- 목표는 이때 계산된 출력층의 오차를 최소화시키는 것
- 이를 위해 ②첫 번째 가중치를 수정하는 과정과 ③두 번째 가중치를 수정하는 과정이 이어짐



# 1 딥러닝의 태동, 오차 역전파

## ● 딥러닝의 태동, 오차 역전파

- 예를 들어 첫 번째 가중치 중 하나인  $w_{31}$ 을 업데이트한다고 하자
- 경사 하강법에서 배운 대로  $w_{31}$ 을 업데이트하기 위해서는 오차 공식을 구하고  $w_{31}$ 에 대해 편미분해야 함
- 합성 함수의 미분이므로 체인 룰을 적용해 편미분을 구하고 이를 이용해  $w_{31}$ 을 업데이트
- 이제 두 번째 가중치를 업데이트할 차례
- 예를 들어  $w_{11}$ 을 업데이트한다고 하면 마찬가지로 오차 공식을 구하고  $w_{11}$ 에 대해 편미분하면 됨
- 여기서 문제가 발생, 앞에서 출력층의 결과와 실제 값을 비교해 오차를 얻었지만, 은닉층은 겉으로 드러나지 않으므로 그 값을 알 수 없음
- 오차를 구할 만한 적절한 출력 값도 없다는 것



# 1 딥러닝의 태동, 오차 역전파

## ● 딥러닝의 태동, 오차 역전파

- 이 문제는 다시 출력층의 오차를 이용하는 것으로 해결
- $w_{31}$ 의 경우  $y_{o1}$ 의 오차만 필요했었지만,  $w_{11}$ 은  $y_{o1}$ 과  $y_{o2}$ 가 모두 관여되어 있음
- 오차 두 개를 모두 계산해 이를 편미분하게 되고, 계산식은 조금 더 복잡해짐
- 이 과정을 마치면 첫 번째 가중치 업데이트 공식과 두 번째 가중치 업데이트 공식이 다음과 같이 정리

$$\text{첫 번째 가중치 업데이트 공식} = \frac{(y_{o1} - y_{\text{실제 값}}) \cdot y_{o1}(1-y_{o1}) \cdot y_{h1}}{w_{11}}$$

$$\text{두 번째 가중치 업데이트 공식} = \frac{(\delta y_{o1} \cdot w_{31} + \delta y_{o2} \cdot w_{41})y_{h1}(1-y_{h1}) \cdot x_1}{w_{11}}$$



# 1 딥러닝의 태동, 오차 역전파

## ● 딥러닝의 태동, 오차 역전파

- 이 공식 중 밑줄 친 부분을 잘 보면 두 식 모두 ‘out(1-out)’ 형태를 취하고 있고, 이를 델타식이라고 함
- 은닉층의 숫자가 늘어도 이러한 형태가 계속해서 나타나게 되므로, 이를 이용해 깊은 층의 계산도 할 수 있게 됨
- 이렇게 깊은 층을 통해 학습할 수 있는 계기가 마련되면서 드디어 딥러닝이 태동하게 된 것
- 우리는 텐서플로를 이용해 이 과정을 진행



## 2 활성화 함수와 고급 경사 하강법

---

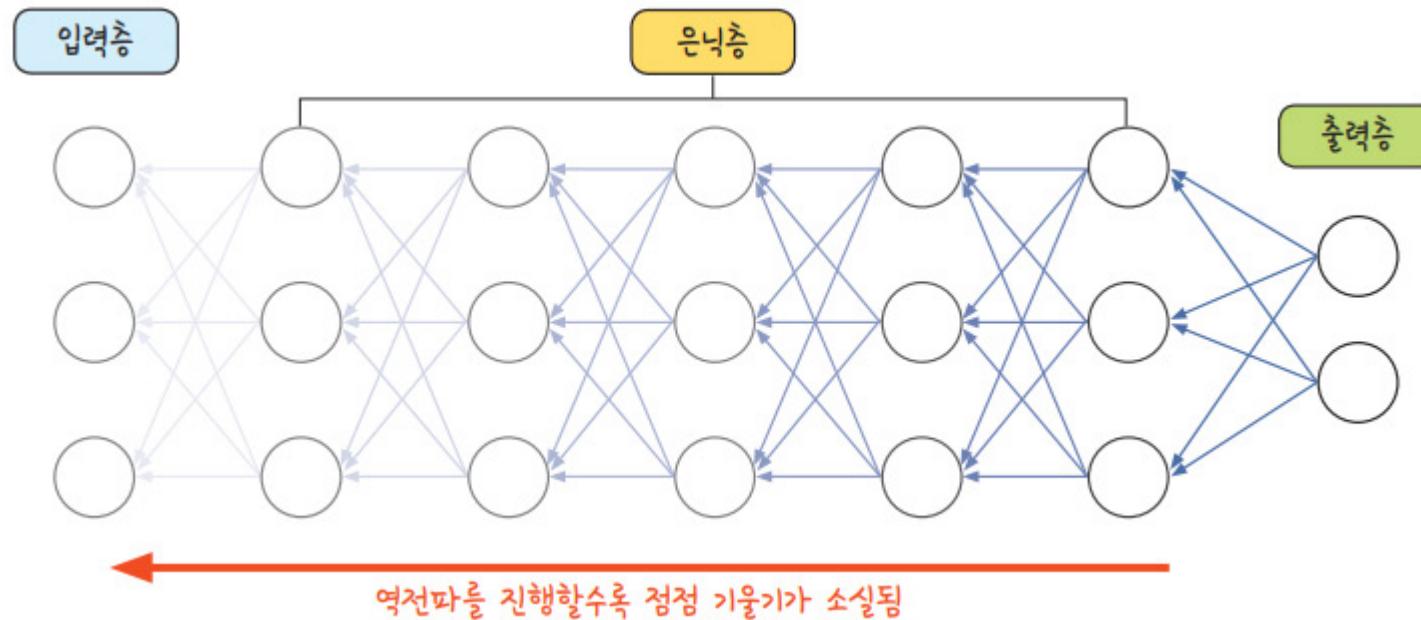


## 2 활성화 함수와 고급 경사 하강법

### ● 활성화 함수와 고급 경사 하강법

- 앞의 델타식을 이용해 깊은 신경망의 계산이 가능해졌음
- 이제 수많은 층을 연결해 학습하면 여러 난제를 해결하는 인공지능이 완성될 것 같아 보임
- 하지만, 아직 한 가지 문제가 더 남아 있음

### ▼ 그림 9-4 | 기울기 소실 문제 발생



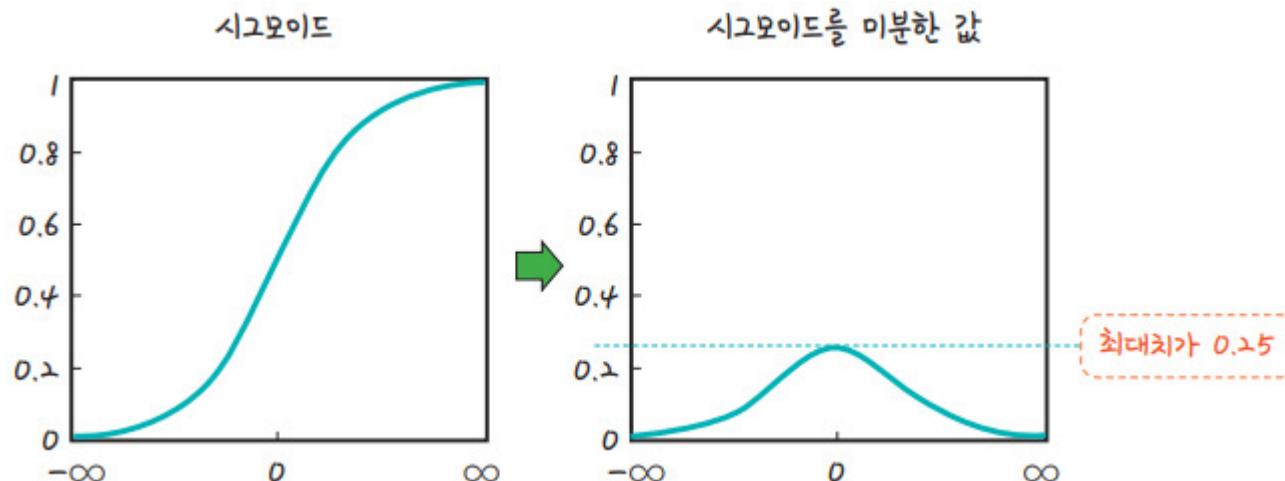


## 2 활성화 함수와 고급 경사 하강법

### ● 활성화 함수와 고급 경사 하강법

- 그림 9-4와 같이 깊은 층을 만들어 보니 출력층에서 시작된 가중치 업데이트가 처음 층까지 전달되지 않는 현상이 생기는 문제가 발견
- 이는 활성화 함수로 사용된 시그모이드 함수의 특성 때문임
- 그림 9-5와 같이 시그모이드 함수를 미분하면 최대치는 0.25
- 1보다 작으므로 계속 곱하다 보면 0에 가까워짐
- 여러 층을 거칠수록 기울기가 사라져 가중치를 수정하기 어려워지는 것

#### ▼ 그림 9-5 | 시그모이드의 미분



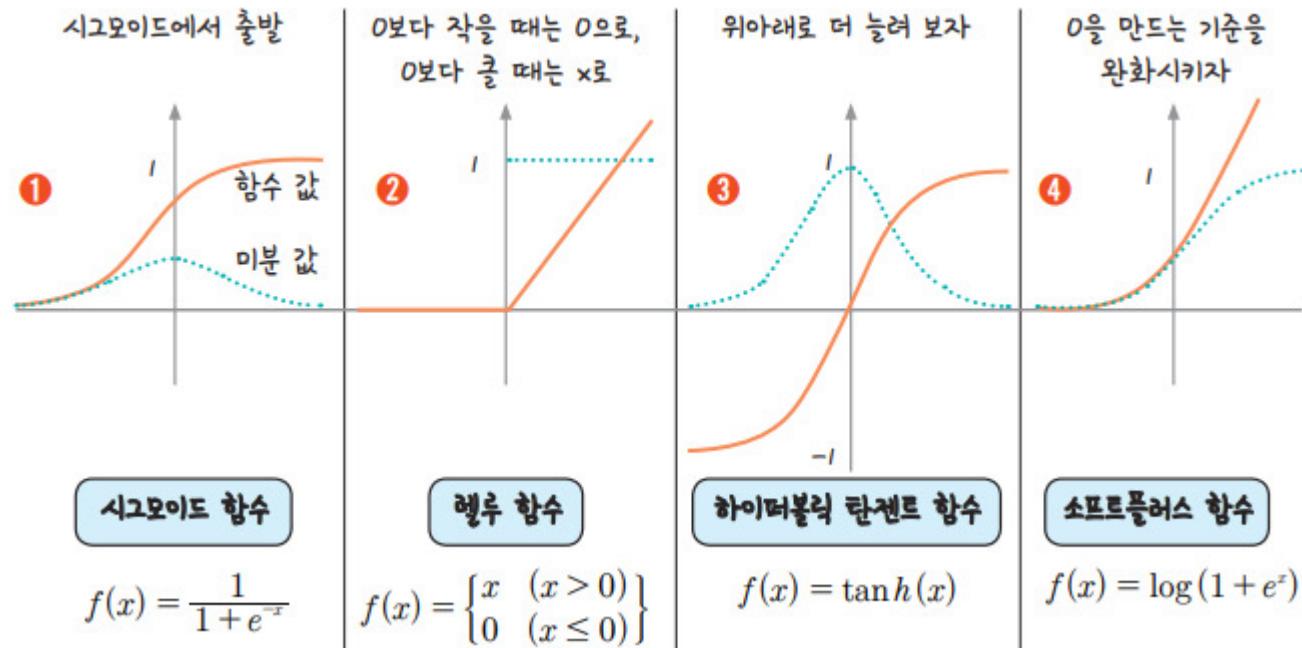


## 2 활성화 함수와 고급 경사 하강법

### ● 활성화 함수와 고급 경사 하강법

- 이를 해결하고자 제프리 힌튼 교수는 렐루(ReLU)라는 새로운 활성화 함수를 제안

### ▼ 그림 9-6 | 여러 활성화 함수의 도입





## 2 활성화 함수와 고급 경사 하강법

### ● 활성화 함수와 고급 경사 하강법

- 그림 9-6 ②에 있는 렐루는  $x$ 가 0보다 작을 때 모든 값을 0으로 처리하고, 0보다 큰 값은  $x$ 를 그대로 사용하는 방법으로, 파란색 점선이 미분을 한 결과인데, 그림에서 보이듯  $x$ 가 0보다 크기만 하면 미분 값은 1이 됨
- 활성화 함수로 렐루를 쓰면 여러 번 오차 역전파가 진행되어도 맨 처음 층까지 값이 남아 있게 됨
- 학습은 결국 오차를 최소화하는 가중치를 찾는 과정
- 출력층에서 알아낸 오차가 역전파를 통해 입력층까지 거슬러 올라가면서 잘못된 가중치들을 수정할 수 있게 되자, 더 깊은 층을 쌓아 올리는 것이 가능해졌음
- 활성화 함수는 그 이후로도 여러 데이터 과학자에 의해 연구되어 ③ 하이퍼볼릭 탄젠트(hyperbolic tangent) 함수나 ④ 소프트플러스(softplus) 함수 등 좀 더 나은 활성화 함수를 만들기 위한 노력이 이어지고 있음



## 3 속도와 정확도 문제를 해결하는 고급 경사 하강법



### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

#### ● 속도와 정확도 문제를 해결하는 고급 경사 하강법

- 앞에서 가중치를 업데이트하는 방법으로 경사 하강법을 배웠음
- 경사 하강법은 정확하게 가중치를 찾아가지만, 계산량이 매우 많다는 단점이 있음
- 이러한 점을 보완한 고급 경사 하강법이 등장하면서 딥러닝의 발전 속도는 더 빨라졌음



### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

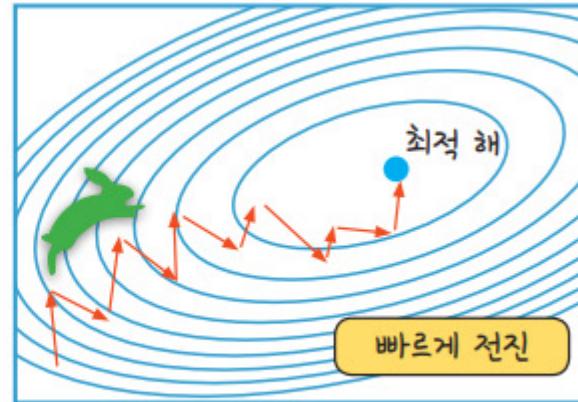
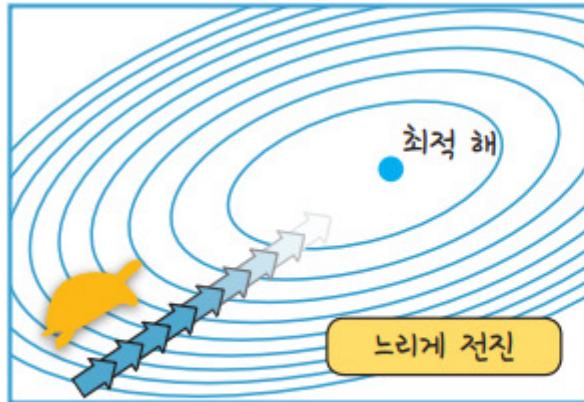
#### ● 속도와 정확도 문제를 해결하는 고급 경사 하강법

- 확률적 경사 하강법
- 경사 하강법은 한 번 업데이트할 때마다 전체 데이터를 미분하므로 속도가 느릴 뿐 아니라, 최적 해를 찾기 전에 최적화 과정이 멈출 수도 있음
- 확률적 경사 하강법(Stochastic Gradient Descent, SGD)은 경사 하강법의 이러한 단점을 보완한 방법
- 전체 데이터를 사용하는 것이 아니라, 랜덤하게 추출한 일부 데이터만 사용하기 때문에 빠르고 더 자주 업데이트할 수 있다는 장점이 있음
- 그림 9-7은 경사 하강법과 확률적 경사 하강법의 차이를 보여 줌
- 랜덤한 일부 데이터를 사용하는 만큼 확률적 경사 하강법은 중간 결과의 진폭이 크고 불안정해 보일 수도 있음
- 속도가 확연히 빠르면서도 최적 해에 근사한 값을 찾아낸다는 장점 덕분에 경사 하강법의 대안으로 사용되고 있음



### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

▼ 그림 9-7 | 경사 하강법과 확률적 경사 하강법의 비교





### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

#### ● 속도와 정확도 문제를 해결하는 고급 경사 하강법

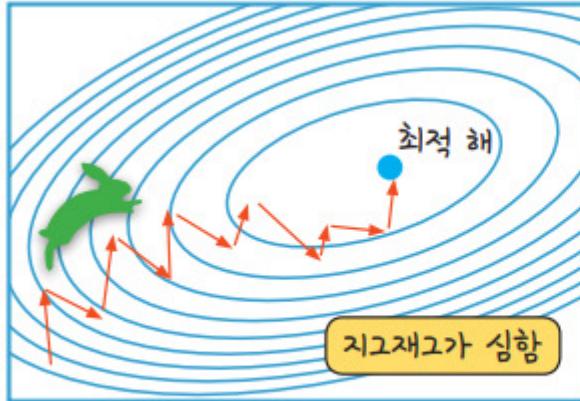
- 모멘텀

- 모멘텀(momentum)이란 단어는 ‘관성, 탄력, 가속도’라는 뜻
- 모멘텀 확률적 경사 하강법(모멘텀 SGD)이란 경사 하강법에 탄력을 더해 주는 것
- 다시 말해 경사 하강법과 마찬가지로 매번 기울기를 구하지만, 이를 통해 오차를 수정하기 전 바로 앞 수정 값과 방향(+, -)을 참고해 같은 방향으로 일정한 비율만 수정되게 하는 방법
- 수정 방향이 양수(+) 방향으로 한 번, 음수(-) 방향으로 한 번 지그재그로 일어나는 현상이 줄어들고, 이전 이동 값을 고려해 일정 비율만큼 다음 값을 결정하므로 관성 효과를 낼 수 있음

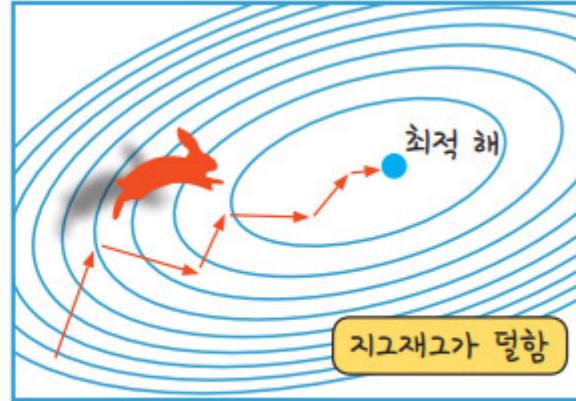


### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

▼ 그림 9-8 | 모멘텀을 적용했을 때



확률적 경사 하강법



모멘텀을 적용한 확률적 경사 하강법



### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

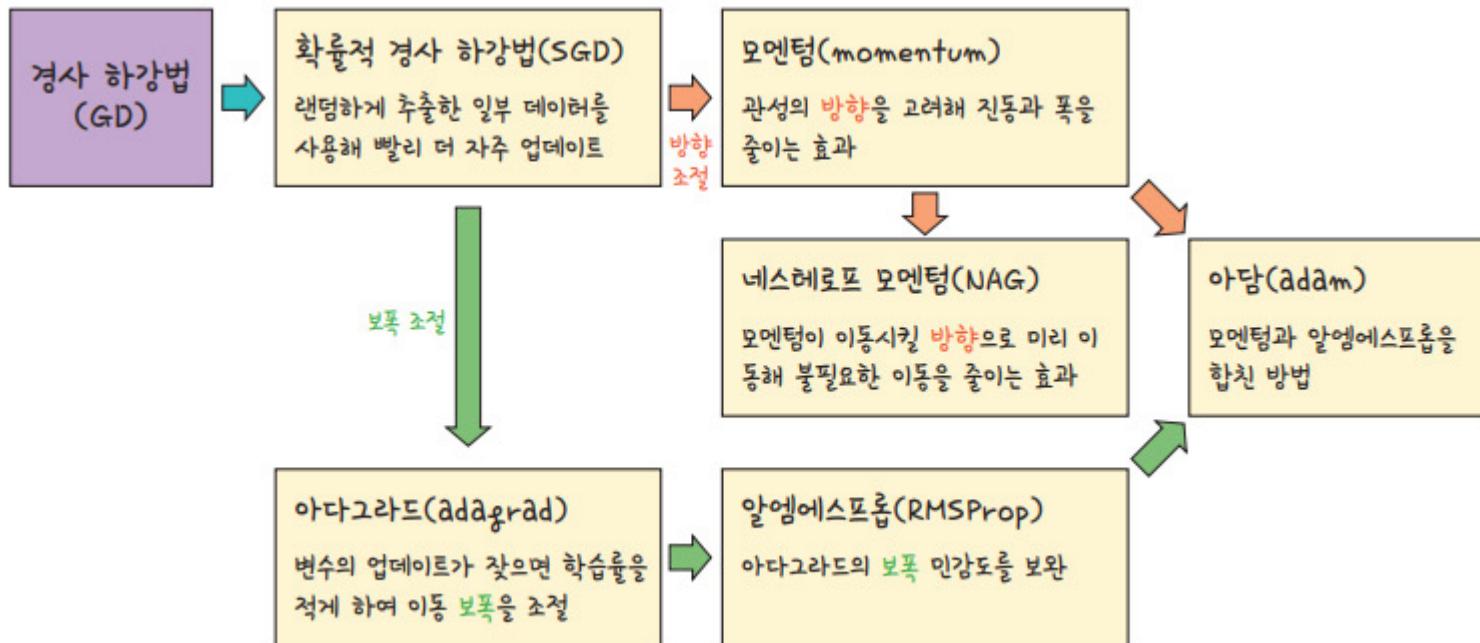
#### ● 속도와 정확도 문제를 해결하는 고급 경사 하강법

- 이 밖에도 딥러닝의 학습을 더 빠르고 정확하게 만들기 위한 노력이 계속되었음
- 지금은 정확도와 속도를 모두 향상시킨 아담(adam)이라는 고급 경사 하강법이 가장 많이 쓰이고 있음
- 그림 9-9는 경사 하강법이 어떻게 해서 아담에 이르게 되었는지 보여 줌



### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

#### ▼ 그림 9-9 | 딥러닝에 사용되는 고급 경사 하강법의 변천





### 3 속도와 정확도 문제를 해결하는 고급 경사 하강법

#### ● 속도와 정확도 문제를 해결하는 고급 경사 하강법

- 이렇게 오차를 최소화하는 경사 하강법들을 딥러닝에서는 ‘옵티마이저’라고 한다.
- 앞에 소개한 고급 경사 하강법들은 텐서플로에 포함되어 있는 optimizers라는 객체에 이름을 적어 주는 것만으로 손쉽게 실행할 수 있음
- 또 앞에서 소개된 시그모이드, 렐루 등 활성화 함수도 activation이라는 객체에 이름을 적어 주는 것으로 손쉽게 실행할 수 있음
- 지금까지 텐서플로에서 사용되는 대부분의 개념과 용어를 배웠음
- 이제 텐서플로를 이용한 모델링을 할 준비가 되었음



# 딥러닝 기본기 다지기

# 딥러닝 모델 설계하기

---

- 1 모델의 정의
- 2 입력층, 은닉층, 출력층
- 3 모델 컴파일
- 4 모델 실행하기



# 딥러닝 모델 설계하기

## ● 딥러닝 모델 설계하기

- 앞에서 소개한 ‘폐암 수술 환자의 생존율 예측하기’ 예제중 딥러닝 모델 부분을 이제 설명할 수 있음
- 앞서 배운 내용이 이 짧은 코드 안에 모두 들어 있음



# 1 모델의 정의

---



# 1 모델의 정의

## ● 모델의 정의

- ‘폐암 수술 환자의 생존율 예측하기’의 딥러닝 코드를 다시 한 번 옮겨 보면 다음과 같음

```
# 텐서플로 라이브러리 안에 있는 클래스 API에서 필요한 함수들을 불러옵니다.
```

```
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense
```

```
# 데이터를 다루는 데 필요한 라이브러리를 불러옵니다.
```

```
import numpy as np
```



# 1 모델의 정의

## ● 모델의 정의

```
# 준비된 수술 환자 데이터를 불러옵니다.  
Data_set = np.loadtxt("./data/ThoracicSurgery3.csv", delimiter=',')  
  
X = Data_set[:,0:16] # 환자의 진찰 기록을 X로 지정합니다.  
y = Data_set[:,16]    # 수술 1년 후 사망/생존 여부를 y로 지정합니다.  
  
# 딥러닝 모델의 구조를 결정합니다.  
model = Sequential()  
model.add(Dense(30, input_dim=16, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))
```



# 1 모델의 정의

## ● 모델의 정의

```
# 딥러닝 모델을 실행합니다.  
model.compile(loss='binary_crossentropy', optimizer='adam',  
metrics=['accuracy'])  
history = model.fit(X, y, epochs=5, batch_size=16)
```



# 1 모델의 정의

## ● 모델의 정의

- 이 코드에서 데이터를 불러오고 다루는 부분은 이미 살펴보았으므로, 지금은 실제로 딥러닝이 수행되는 부분을 더 자세히 알아보도록 함
- 딥러닝의 모델을 설정하고 구동하는 부분은 모두 `model`이라는 함수를 선언하며 시작
- 먼저 `model = Sequential()`로 시작되는 부분은 딥러닝의 구조를 짜고 층을 설정하는 부분
- 이어서 나오는 `model.compile()` 부분은 앞에서 정한 모델을 컴퓨터가 알아들을 수 있게끔 컴파일하는 부분
- `model.fit()`으로 시작하는 부분은 모델을 실제로 수행하는 부분



## 2 입력층, 은닉층, 출력층

---



## 2 입력층, 은닉층, 출력층

### ● 입력층, 은닉층, 출력층

- 먼저 딥러닝의 구조를 짜고 층을 설정하는 부분을 살펴보면 다음과 같음

```
model = Sequential()  
model.add(Dense(30, input_dim=16, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))
```



## 2 입력층, 은닉층, 출력층

### ● 입력층, 은닉층, 출력층

- 앞에서 딥러닝이란 입력층과 출력층 사이에 은닉층들을 차곡차곡 추가하면서 학습시키는 것임을 배웠음
- 이 층들이 케라스에서는 Sequential( ) 함수를 통해 쉽게 구현
- Sequential() 함수를 model로 선언해 놓고 model.add()라는 라인을 추가하면 새로운 층이 만들어짐
- 코드에는 model.add( )로 시작되는 라인이 두 개 있으므로 층을 두 개 가진 모델을 만든 것
- 맨 마지막 층은 결과를 출력하는 ‘출력층’이 됨
- 나머지는 모두 ‘은닉층’의 역할을 함
- 지금 만들어진 이 층 두 개는 각각 은닉층과 출력층
- 각각의 층은 Dense라는 함수를 통해 구체적으로 그 구조가 결정



## 2 입력층, 은닉층, 출력층

### ● 입력층, 은닉층, 출력층

- 이제 model.add(Dense(30, input\_dim=16)) 부분을 더 살펴보자
- model.add() 함수를 통해 새로운 층을 만들고 나면 Dense() 함수의 첫 번째 인자에 몇 개의 노드를 이 층에 만들 것인지 숫자를 적어 줌
- 노드란 앞서 소개된 ‘가중합’에 해당하는 것으로 이전 층에서 전달된 변수와 가중치, 바이어스가 하나로 모이게 되는 곳
- 하나의 층에 여러 개의 노드를 적절히 만들어 주어야 하는데, 30이라고 되어 있는 것은 이 층에 노드를 30개 만들겠다는 것



## 2 입력층, 은닉층, 출력층

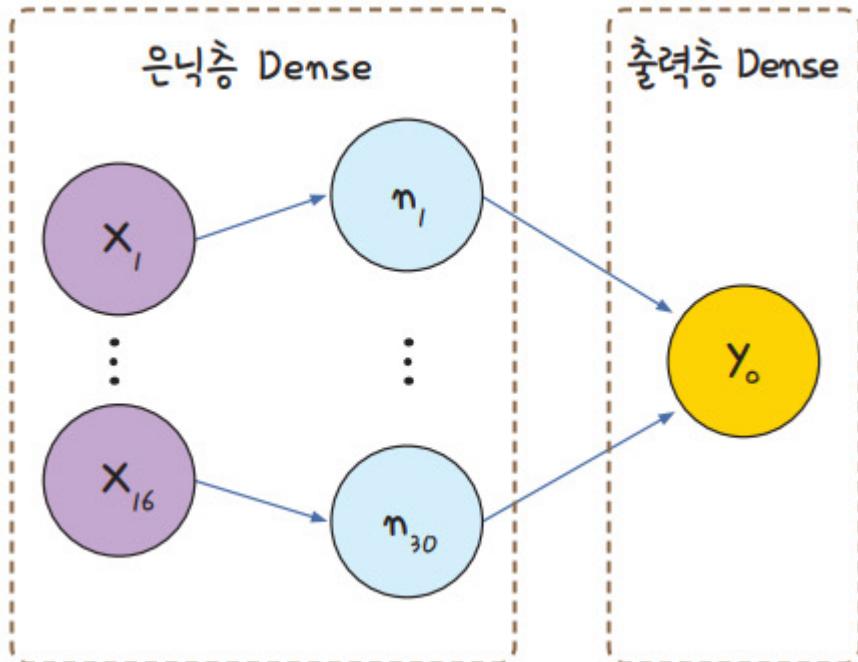
### ● 입력층, 은닉층, 출력층

- 이어서 `input_dim`이라는 변수가 나옴
- 이는 입력 데이터에서 몇 개의 값을 가져올지 정하는 것
- keras는 입력층을 따로 만드는 것이 아니라, 첫 번째 은닉층에 `input_dim`을 적어 줌으로써 첫 번째 Dense가 은닉층 + 입력층의 역할을 겸함
- 현재 다루고 있는 폐암 수술 환자의 생존 여부 데이터에는 입력 값이 16개 있음
- 데이터에서 값을 16개 받아 은닉층의 노드 30개로 보낸다는 의미



## 2 입력층, 은닉층, 출력층

▼ 그림 10-1 | 첫 번째 Dense는 입력층과 첫 번째 은닉층을, 두 번째 Dense는 출력층을 의미





## 2 입력층, 은닉층, 출력층

### ● 입력층, 은닉층, 출력층

- 두 번째 나오는 `model.add(Dense(1, activation='sigmoid'))`를 보겠음
- 마지막 층이므로 이 층이 곧 출력층이 됨
- 출력 값을 하나로 정해서 보여 주어야 하므로 출력층의 노드 수는 한 개
- 이 노드에서 입력받은 값은 활성화 함수를 거쳐 최종 출력 값으로 나와야 함
- 여기서는 활성화 함수로 시그모이드(sigmoid) 함수를 사용



## 3 모델 컴파일

---



# 3 모델 컴파일

## ● 모델 컴파일

- 다음으로 model.compile 부분

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[  
    'accuracy'])
```



# 3 모델 컴파일

## ● 모델 컴파일

- model.compile 부분은 앞서 지정한 모델이 효과적으로 구현될 수 있게 여러 가지 환경을 설정해 주면서 컴파일하는 부분
- 먼저 어떤 오차 함수를 사용할지 정해야 함
- 앞에서 손실 함수에는 두 가지 종류가 있음을 배웠음
- 바로 선형 회귀에서 사용한 평균 제곱 오차와 로지스틱 회귀에서 사용한 교차 엔트로피 오차
- 폐암 수술 환자의 생존율 예측은 생존과 사망, 둘 중 하나를 예측하므로 교차 엔트로피 오차 함수를 적용하기 위해 binary\_crossentropy를 선택



# 3 모델 컴파일

## ● 모델 컴파일

- 손실 함수는 최적의 가중치를 학습하기 위해 필수적인 부분
- 올바른 손실 함수를 통해 계산된 오차는 옵티마이저를 적절히 활용하도록 만들어 줌
- 케라스는 쉽게 사용 가능한 여러 가지 손실 함수를 준비해 놓고 있음
- 크게 평균 제곱 오차 계열과 교차 엔트로피 계열 오차로 나누는데, 이를 표 10-1에 정리해 놓았음
- 선형 회귀 모델은 평균 제곱 계열 중 하나를, 이항 분류를 위해서는 binary\_crossentropy를, 그리고 다항 분류에서는 categorical\_crossentropy를 사용한다는 것을 기억하자



# 3 모델 컴파일

## ▼ 표 10-1 | 대표적인 오차 함수

\* 실제 값을  $y_t$ , 예측 값을  $y_o$ 라고 가정할 때

평균 제곱 계열 (선형 회귀 모델)	mean_squared_error	평균 제곱 오차 계산: $\text{mean}(\text{square}(y_t - y_o))$
	mean_absolute_error	평균 절대 오차(실제 값과 예측 값 차이의 절댓값 평균) 계산: $\text{mean}(\text{abs}(y_t - y_o))$
	mean_absolute_percentage_error	평균 절대 백분율 오차(절댓값 오차를 절댓값으로 나눈 후 평균) 계산: $\text{mean}(\text{abs}(y_t - y_o) / \text{abs}(y_t))$ (단, 분모 $\neq 0$ )
	mean_squared_logarithmic_error	평균 제곱 로그 오차(실제 값과 예측 값에 로그를 적용한 값의 차이를 제곱한 값의 평균) 계산: $\text{mean}(\text{square}((\log(y_o) + 1) - (\log(y_t) + 1)))$
교차 엔트로피 계열 (다항 분류, 이항 분류)	categorical_crossentropy	범주형 교차 엔트로피(다항 분류, 여럿 중 하나를 예측할 때)
	binary_crossentropy	이항 교차 엔트로피(이항 분류, 둘 중 하나를 예측할 때)



# 3 모델 컴파일

## ● 모델 컴파일

- 이어서 옵티마이저를 선택할 차례
- 앞 장에서 현재 가장 많이 쓰는 옵티마이저는 adam이라고 했음
- Optimizer란에 adam을 적어 주는 것으로 실행할 준비가 됨(**예** optimizer='adam')
- metrics() 함수는 모델이 컴파일될 때 모델 수행의 결과를 나타내게끔 설정하는 부분
- accuracy라고 설정한 것은 학습셋에 대한 정확도에 기반해 결과를 출력하라는 의미(**예** metrics=['accuracy'])
- accuracy 외에 학습셋에 대한 손실 값을 나타내는 loss, 테스트셋을 적용할 경우 테스트셋에 대한 정확도를 나타내는 val\_acc, 테스트셋에 대한 손실 값을 나타내는 val\_loss 등을 사용할 수 있음



## 4 모델 실행하기

---



# 4 모델 실행하기

## ● 모델 실행하기

- 모델을 정의하고 컴파일하고 나면 이제 실행시킬 차례
- 앞서 컴파일 단계에서 정해진 환경을 주어진 데이터를 불러 실행시킬 때 사용되는 함수는 다음과 같이 model.fit() 부분

```
history = model.fit(X, y, epochs=5, batch_size=16)
```



# 4 모델 실행하기

## ● 모델 실행하기

- 이 부분을 설명하기에 앞서 용어를 다시 한 번 정리해 보자
- 주어진 폐암 수술 환자의 수술 후 생존 여부 데이터는 총 470명의 환자에게서 16개의 정보를 정리한 것
- 이때 각 정보를 ‘속성’이라고 함
- 생존 여부를 클래스, 가로 한 줄에 해당하는 각 환자의 정보를 각각 ‘샘플’이라고 함
- 주어진 데이터에는 총 470개의 샘플이 각각 16개씩의 속성을 가지고 있는 것이라고 앞서 설명한 바 있음
- 이 용어는 샘플을 instance 또는 example이라고도 하며, 속성 대신 피처(feature) 또는 특성이라고도 함
- ‘속성’과 ‘샘플’로 통일해서 사용



## 4 모델 실행하기

▼ 그림 10-2 | 폐암 환자 생존율 예측 데이터의 샘플, 속성, 클래스 구분

	속성				클래스	
샘플	정보 1	정보 2	...	정보 16	생존 여부	
샘플	1번째 환자	2	2.88	...	60	0
	2번째 환자	2	3.4	...	51	0
	3번째 환자	2	2.76	...	59	0
	...	...	...	...	...	...
	470번째 환자	2	4.72	...	51	0



# 4 모델 실행하기

## ● 모델 실행하기

- 학습 프로세스가 모든 샘플에 대해 한 번 실행되는 것을 1 epoch('에포크'라고 읽음)라고 함
- 코드에서 epochs=5로 지정한 것은 각 샘플이 처음부터 끝까지 다섯 번 재사용될 때까지 실행을 반복하라는 의미
- batch\_size는 샘플을 한 번에 몇 개씩 처리할지 정하는 부분으로 batch\_size=16은 전체 470개의 샘플을 16개씩 끊어서 집어넣으라는 의미
- batch\_size가 너무 크면 학습 속도가 느려지고, 너무 작으면 각 실행 값의 편차가 생겨서 전체 결괏값이 불안정해질 수 있음
- 자신의 컴퓨터 메모리가 감당할 만큼의 batch\_size를 찾아 설정해 주는 것이 좋음



# 딥러닝 기본기 다지기

# 데이터 다루기

---

- 1 딥러닝과 데이터
- 2 피마 인디언 데이터 분석하기
- 3 판다스를 활용한 데이터 조사
- 4 중요한 데이터 추출하기
- 5 피마 인디언의 당뇨병 예측 실행



# 1 딥러닝과 데이터

---



# 1 딥러닝과 데이터

## ● 딥러닝과 데이터

- 데이터양이 많다고 해서 무조건 좋은 결과를 얻을 수 있는 것은 아님
- 데이터양도 중요하지만, 그 안에 ‘필요한’ 데이터가 얼마나 있는가도 중요하기 때문임
- 준비된 데이터가 우리가 사용하려는 머신 러닝과 딥러닝에 얼마나 효율적으로 사용되게끔 가공되었는지 역시 중요함
- 머신 러닝 프로젝트의 성공과 실패는 얼마나 좋은 데이터를 가지고 시작하느냐에 영향을 많이 받음
- 여기서 좋은 데이터란 한쪽으로 치우치지 않고, 불필요한 정보가 대량으로 포함되어 있지 않으며, 왜곡되지 않은 데이터를 의미
- 이러한 데이터를 만들기 위해 머신 러닝, 딥러닝 개발자들은 데이터를 직접 들여다보고 분석할 수 있어야 함
- 이루고 싶은 목적에 맞추어 가능한 한 많은 정보를 모았다면 이를 머신 러닝과 딥러닝에서 사용 할 수 있게 잘 정제된 데이터 형식으로 바꾸어야 함
- 이 작업은 모든 머신 러닝, 딥러닝 프로젝트의 첫 단추이자 가장 중요한 작업



# 1 딥러닝과 데이터

## ● 딥러닝과 데이터

- 데이터 분석에 가장 많이 사용하는 파이썬 라이브러리인 판다스(pandas)와 맷플롯립(matplotlib) 등을 사용해 앞으로 다룰 데이터가 어떤 내용을 담고 있는지 확인하면서 딥러닝의 핵심 기술들을 하나씩 구현해 보자



## 2 피마 인디언 데이터 분석하기

---



## 2 피마 인디언 데이터 분석하기

▼ 그림 11-1 | 피마 인디언 옛 모습





## 2 피마 인디언 데이터 분석하기

### ● 피마 인디언 데이터 분석하기

- 비만은 유전일까?
- 아니면 식습관 조절에 실패한 자신의 탓일까?
- 비만이 유전 및 환경, 모두의 탓이라는 것을 증명하는 좋은 사례가 바로 미국 남서부에 살고 있는 피마 인디언의 사례
- 피마 인디언은 1950년대까지만 해도 비만인 사람이 단 1명도 없는 민족이었음
- 지금은 전체 부족의 60%가 당뇨, 80%가 비만으로 고통받고 있음
- 이는 생존하기 위해 영양분을 체내에 저장하는 뛰어난 능력을 물려받은 인디언들이 미국의 기름진 패스트푸드 문화를 만나면서 벌어진 일
- 피마 인디언을 대상으로 당뇨병 여부를 측정한 데이터는 data 폴더에서 찾을 수 있음(data/pima-indians-diabetes3.csv)



## 2 피마 인디언 데이터 분석하기

### ● 피마 인디언 데이터 분석하기

- 데이터 파일을 열어 보면 모두 768명의 인디언으로부터 여덟 개의 정보와 한 개의 클래스를 추출한 데이터임을 알 수 있음

### ▼ 그림 11-2 | 피마 인디언 데이터의 샘플, 속성, 클래스 구분

	정보 1	정보 2	정보 3	...	정보 8	당뇨병 여부
샘플	1번째 인디언	6	148	72	...	50
	2번째 인디언	1	85	66	...	31
	3번째 인디언	8	183	64	...	32
	...	...	...	...	...	...
	768번째 인디언	1	93	70	...	23



## 2 피마 인디언 데이터 분석하기

- 샘플 수: 768
- 속성: 8
  - 정보 1(pregnant): 과거 임신 횟수
  - 정보 2(plasma): 포도당 부하 검사 2시간 후 공복 혈당 농도(mm Hg)
  - 정보 3(pressure): 혓장기 혈압(mm Hg)
  - 정보 4(thickness): 삼두근 피부 주름 두께(mm)
  - 정보 5(insulin): 혈청 인슐린(2-hour, mu U/ml)
  - 정보 6(BMI): 체질량 지수(BMI, weight in kg/(height in m)<sup>2</sup>)
  - 정보 7(pedigree): 당뇨병 가족력
  - 정보 8(age): 나이
- 클래스: 당뇨(1), 당뇨 아님(0)



## 2 피마 인디언 데이터 분석하기

### ● 피마 인디언 데이터 분석하기

- 데이터의 각 정보가 의미하는 의학, 생리학 배경지식을 모두 알 필요는 없지만, 딥러닝을 구동하려면 반드시 속성과 클래스를 먼저 구분해야 함
- 또한, 모델의 정확도를 향상시키기 위해서는 데이터를 추가하거나 재가공해야 할 수도 있음
- 데이터의 내용과 구조를 파악하는 것이 중요함



## 3 판다스를 활용한 데이터 조사

---



### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 데이터를 잘 파악하는 것이 딥러닝을 다루는 기술의 1단계
- 데이터의 크기가 커지고 정보량이 많아지면 데이터를 불러오고 내용을 파악할 수 있는 효과적인 방법이 필요함
- 이때 가장 유용한 방법이 데이터를 시각화해서 눈으로 직접 확인해 보는 것
- 데이터를 다룰 때는 데이터를 다루기 위해 만들어진 라이브러리를 사용하는 것을 권장
- 판다스 라이브러리 사용



### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 이 실습에는 판다스(pandas)와 시본(seaborn) 라이브러리가 필요함
- 코랩은 기본으로 제공하지만, 주피터 노트북을 이용해 실습 중이라면 다음 명령으로 두 라이브러리를 설치

```
!pip install pandas
```

```
!pip install seaborn
```



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

```
# 필요한 라이브러리를 불러옵니다.
```

```
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns
```

```
# 피마 인디언 당뇨병 데이터셋을 불러옵니다.
```

```
df = pd.read_csv('./data/pima-indians-diabetes3.csv')
```



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 판다스 라이브러리의 `read_csv()` 함수로 csv 파일을 불러와 df라는 이름의 데이터 프레임으로 저장
- csv란 comma separated values의 약어로, 쉼표(,)로 구분된 데이터들의 모음이란 뜻
- csv 파일에는 데이터를 설명하는 한 줄이 파일 맨 처음에 나옴
- 이를 헤더(header)라고 함



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 이제 불러온 데이터의 내용을 간단히 확인하고자 head() 함수를 이용해 데이터의 첫 다섯 줄을 불러오자

```
df.head(5)
```



### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 다음과 같이 출력
- 파이썬에서는 숫자를 0부터 세기 때문에 맨 첫 번째 행이 1이 아닌 0

pregnant	plasma	pressure	thickness	insulin	bmi	pedigree	age	diabetes
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33



### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 이제 정상과 당뇨 환자가 각각 몇 명씩인지 조사해 보자
- 불러온 데이터 프레임의 특정 칼럼을 불러오려면 df[“칼럼명”]이라고 입력하면 됨
- value\_counts() 함수를 이용하면 각 컬럼의 값이 몇 개씩 있는지 알려 줌

```
df["diabetes"].value_counts()
```



### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 다음과 같은 정보가 화면에 출력
- 정상인 500명과 당뇨병 환자 268명을 포함, 총 768개의 샘플이 준비되어 있는 것을 알 수 있음

실행 결과

```
0    500  
1    268
```

```
Name: diabetes, dtype: int64
```



### 3 판다스를 활용한 데이터 조사

- 판다스를 활용한 데이터 조사

- 정보별 특징을 좀 더 자세히 알고 싶으면 `describe()` 함수를 이용

```
df.describe()
```



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 다음과 같은 내용이 출력
- 정보별 샘플 수(count), 평균(mean), 표준편차(std), 최솟값(min), 백분위 수로 25%, 50%, 75%에 해당하는 값 그리고 최댓값(max)이 정리되어 보임

	pregnant	plasma	pressure	thickness	insulin	bmi	pedigree	age	diabetes
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000



### 3 판다스를 활용한 데이터 조사

- 판다스를 활용한 데이터 조사

- 각 항목이 어느 정도의 상관관계를 가지고 있는지 알고 싶다면 다음과 같이 입력

```
df.corr()
```



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 다음과 같이 출력

	pregnant	plasma	pressure	thickness	insulin	bmi	pedigree	age	diabetes
pregnant	1.000000	0.129459	0.141282	-0.081672	-0.073535	0.017683	-0.033523	0.544341	0.221898
plasma	0.129459	1.000000	0.152590	0.057328	0.331357	0.221071	0.137337	0.263514	0.466581
pressure	0.141282	0.152590	1.000000	0.207371	0.088933	0.281805	0.041265	0.239528	0.065068
thickness	-0.081672	0.057328	0.207371	1.000000	0.436783	0.392573	0.183928	-0.113970	0.074752
insulin	-0.073535	0.331357	0.088933	0.436783	1.000000	0.197859	0.185071	-0.042163	0.130548
bmi	0.017683	0.221071	0.281805	0.392573	0.197859	1.000000	0.140647	0.036242	0.292695
pedigree	-0.033523	0.137337	0.041265	0.183928	0.185071	0.140647	1.000000	0.033561	0.173844
age	0.544341	0.263514	0.239528	-0.113970	-0.042163	0.036242	0.033561	1.000000	0.238356
diabetes	0.221898	0.466581	0.065068	0.074752	0.130548	0.292695	0.173844	0.238356	1.000000



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 상관관계를 그래프로 표현
- 맷플롯립(matplotlib)은 파이썬에서 그래프를 그릴 때 가장 많이 사용되는 라이브러리
- 이를 기반으로 조금 더 정교한 그래프를 그리게 해 주는 시본(seaborn) 라이브러리까지 사용해서 각 정보 간 상관관계 가시화
- 먼저 그래프의 색상과 크기를 정함

```
colormap = plt.cm.gist_heat      # 그래프의 색상 구성을 정합니다.  
plt.figure(figsize=(12,12))       # 그래프의 크기를 정합니다.
```



# 3 판다스를 활용한 데이터 조사

## ● 판다스를 활용한 데이터 조사

- 시본 라이브러리 중 각 항목 간 상관관계를 나타내는 heatmap() 함수를 통해 그래프를 표시해 보자
- heatmap() 함수는 두 항목씩 짹을 지은 후 각각 어떤 패턴으로 변화하는지 관찰하는 함수
- 두 항목이 전혀 다른 패턴으로 변화하면 0을, 서로 비슷한 패턴으로 변할수록 1에 가까운 값을 출력

```
sns.heatmap(df.corr(), linewidths=0.1, vmax=0.5, cmap=cmap,
            linecolor='white', annot=True)
plt.show()
```



### 3 판다스를 활용한 데이터 조사

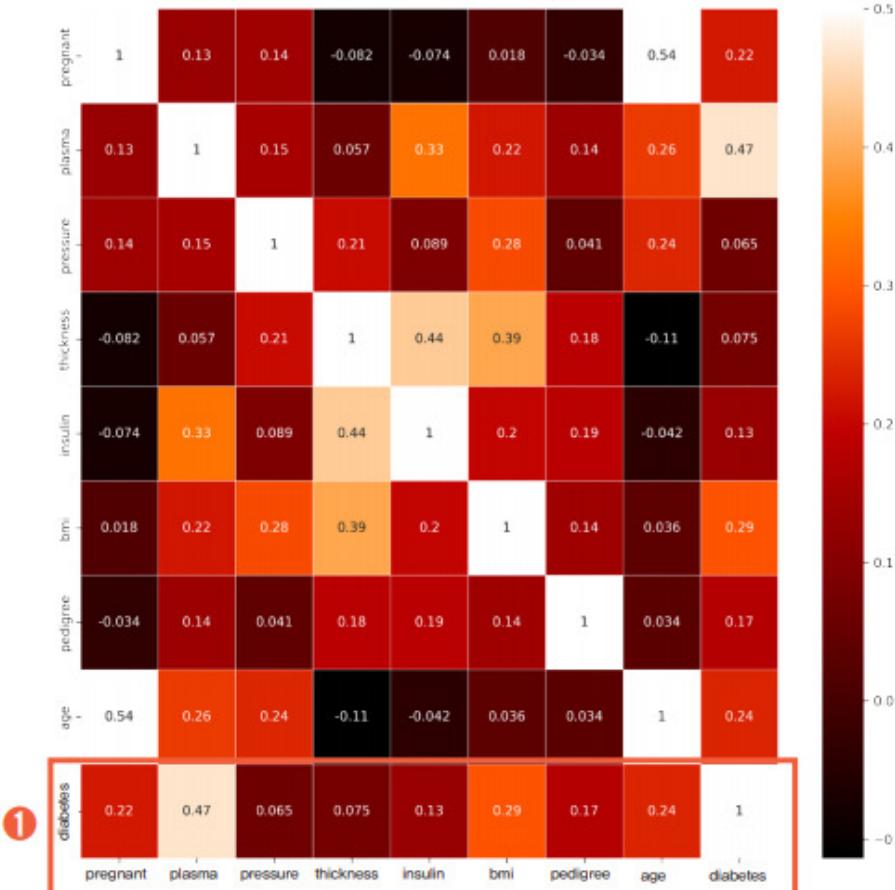
#### ● 판다스를 활용한 데이터 조사

- vmax는 색상의 밝기를 조절하는 인자
- cmap은 미리 정해진 맷플롯립 색상의 설정 값을 불러옴
- 색상 설정 값은 <https://matplotlib.org/users/colormaps.html>에서 확인할 수 있음



### 3 판다스를 활용한 데이터 조사

▼ 그림 11-3 | 정보 간 상관관계 그래프





### 3 판다스를 활용한 데이터 조사

#### ● 판다스를 활용한 데이터 조사

- 그림 11-3에서 가장 눈여겨보아야 할 부분은 당뇨병 발병 여부를 가리키는 ❶ diabetes 항목
- diabetes 항목을 보면 pregnant부터 age까지 상관도가 숫자로 표시되어 있고, 숫자가 높을수록 밝은 색상으로 채워져 있음



## 4 중요한 데이터 추출하기

---



# 4 중요한 데이터 추출하기

## ● 중요한 데이터 추출하기

- 앞서 그림 11-3을 살펴보면 plasma 항목(공복 혈당 농도)과 BMI(체질량 지수)가 예측하고자 하는 diabetes 항목과 상관관계가 높다는 것을 알 수 있음
- 즉, 이 항목들이 예측 모델을 만드는데 중요한 역할을 할 것으로 기대할 수 있음
- 이 두 항목만 따로 떼어 내어 당뇨의 발병 여부와 어떤 관계가 있는지 알아보자



# 4 중요한 데이터 추출하기

## ● 중요한 데이터 추출하기

- 먼저 plasma를 기준으로 각각 정상과 당뇨 여부가 어떻게 분포되는지 살펴보자
- 다음과 같이 히스토그램을 그려 주는 맷플롯립 라이브러리의 hist() 함수를 이용

```
plt.hist(x=[df.plasma[df.diabetes==0], df.plasma[df.diabetes==1]], bins=30,  
histtype='barstacked', label=['normal', 'diabetes'])  
plt.legend()
```



# 4 중요한 데이터 추출하기

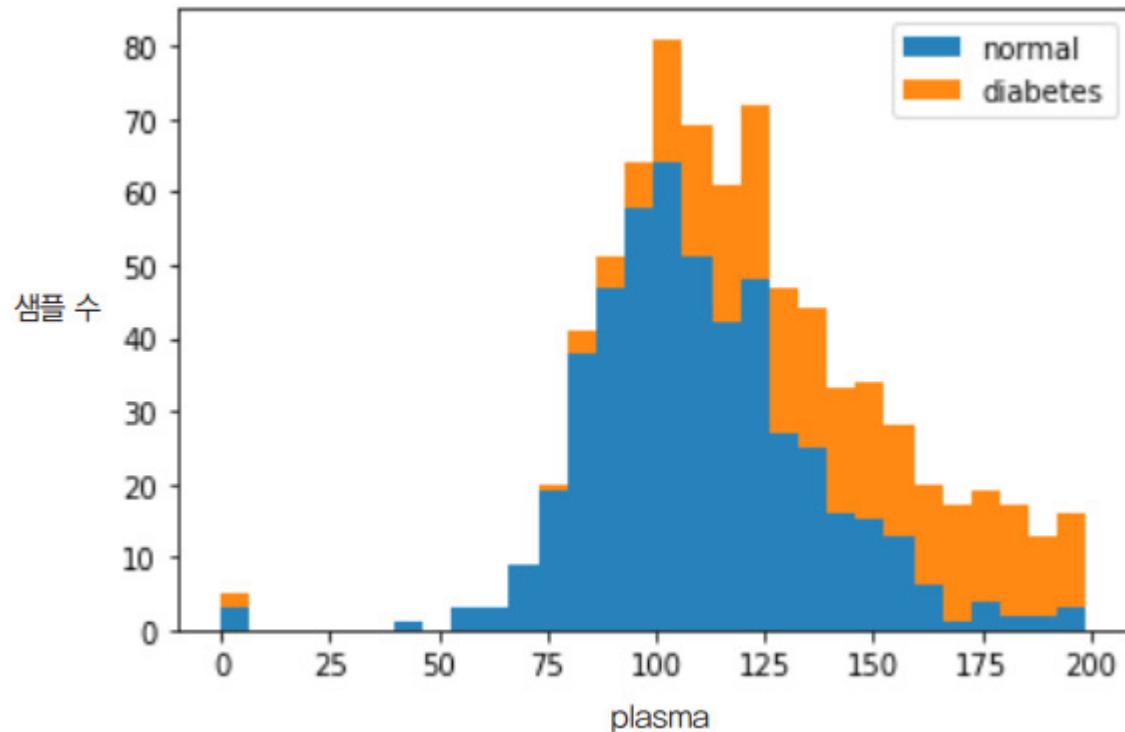
## ● 중요한 데이터 추출하기

- 가져오게 될 칼럼을 hist() 함수 안에 x축으로 지정
- 여기서는 df 안의 plasma 칼럼 중 diabetes 값이 0인 것과 1인 것을 구분해 불러오게 했음
- bins는 x축을 몇 개의 막대로 쪼개어 보여 줄 것인지 정하는 변수
- barstacked 옵션은 여러 데이터가 쌓여 있는 형태의 막대바를 생성하는 옵션
- 불러온 데이터의 이름을 각각 normal(정상)과 diabetes(당뇨)로 정함
- 이를 실행시키면 그림 11-4와 같은 그래프가 형성



## 4 중요한 데이터 추출하기

▼ 그림 11-4 | plasma를 기준으로 정상과 당뇨 여부 표시





# 4 중요한 데이터 추출하기

## ● 중요한 데이터 추출하기

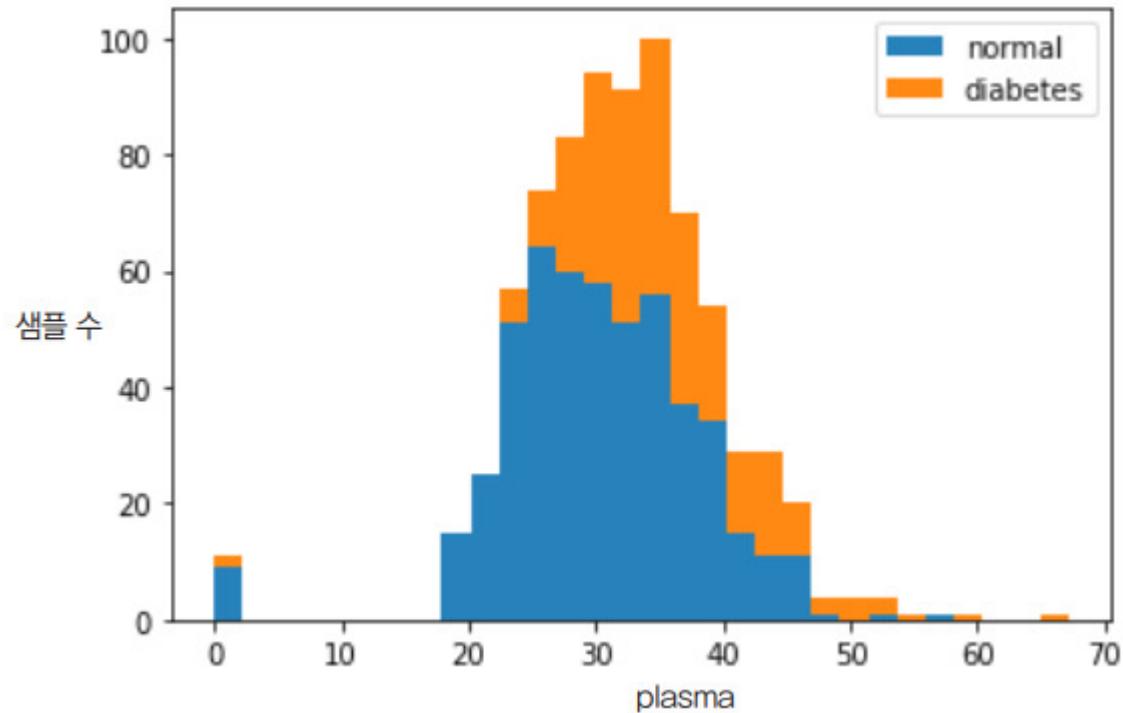
- plasma 수치가 높아질수록 당뇨인 경우가 많음을 알 수 있음
- 마찬가지 방법으로, 이번에는 BMI를 기준으로 각각 정상과 당뇨가 어느 정도 비율로 분포하는지 살펴보자

```
plt.hist(x=[df.bmi[df.diabetes==0], df.bmi[df.diabetes==1]], bins=30,  
histtype='barstacked', label=['normal', 'diabetes'])  
plt.legend()
```



## 4 중요한 데이터 추출하기

▼ 그림 11-5 | BMI를 기준으로 정상과 당뇨 여부 표시





# 4 중요한 데이터 추출하기

## ● 중요한 데이터 추출하기

- BMI가 높아질 경우 당뇨의 발병률도 함께 증가하는 추세를 볼 수 있음
- 이렇게 결과에 미치는 영향이 큰 항목을 발견하는 것이 데이터 전처리 과정 중 하나
- 이 밖에도 데이터에 빠진 값이 있다면 평균이나 중앙값으로 대체하거나, 흐름에서 크게 벗어나는 이상치를 제거하는 과정 등이 데이터 전처리에 포함될 수 있음
- SVM이나 랜덤 포레스트처럼 일반적인 머신 러닝에서는 데이터 전처리 과정이 성능 향상에 중요한 역할을 함



## 5 피마 인디언의 당뇨병 예측 실행



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

- 텐서플로의 케라스를 이용해서 예측 실행
- 판다스 라이브러리를 사용하기 때문에 iloc[] 함수를 사용해 X와 y를 각각 저장
- iloc는 데이터 프레임에서 대괄호 안에 정한 범위만큼 가져와 저장하게 함

```
# 피마 인디언 당뇨병 데이터셋을 불러옵니다.
```

```
df = pd.read_csv('./data/pima-indians-diabetes3.csv')
```

```
X = df.iloc[:,0:8] # 세부 정보를 X로 지정합니다.
```

```
y = df.iloc[:,8] # 당뇨병 여부를 y로 지정합니다.
```



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

- 모델 구조를 설정

```
model = Sequential()  
model.add(Dense(12, input_dim=8, activation='relu', name='Dense_1'))  
model.add(Dense(8, activation='relu', name='Dense_2'))  
model.add(Dense(1, activation='sigmoid', name='Dense_3'))  
model.summary()
```

- 이전과 달라진 점은 은닉층이 하나 더 추가되었다는 것
- 층과 층의 연결을 한눈에 볼 수 있게 해 주는 model.summary() 부분이 추가
- model.summary()의 실행 결과는 다음과 같음



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

실행 결과

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
Dense_1 (Dense)	(None, 12)	108
Dense_2 (Dense)	(None, 8)	104
Dense_3 (Dense)	(None, 1)	9
<hr/>		

Total params: 221

Trainable params: 221

Non-trainable params: 0

} 4



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

- ① Layer 부분은 층의 이름과 유형을 나타냄
- 각 층의 이름은 자동으로 정해지는데, 따로 이름을 만들려면 Dense() 함수 안에 name='층 이름'을 추가해 주면 됨
- 입력층과 첫 번째 은닉층을 연결해 주는 Dense\_1층, 첫 번째 은닉층과 두 번째 은닉층을 연결하는 Dense\_2층, 그리고 두 번째 은닉층과 출력층을 연결하는 Dense\_3층이 만들어졌음을 알 수 있음
- ② Output Shape 부분은 각 층에 몇 개의 출력이 발생하는지 나타냄
- 쉼표(,)를 사이에 두고 괄호의 앞은 행(샘플)의 수, 뒤는 열(속성)의 수를 의미
- 행의 수는 batch\_size에 정한 만큼 입력되므로 딥러닝 모델에서는 이를 특별히 세지 않음
- 괄호의 앞은 None으로 표시
- 여덟 개의 입력이 첫 번째 은닉층을 지나며 12개가 되고, 두 번째 은닉층을 지나며 여덟 개가 되었다가 출력층에서는 한 개의 출력을 만든다는 것을 알 수 있음



# 5 피마 인디언의 당뇨병 예측 실행

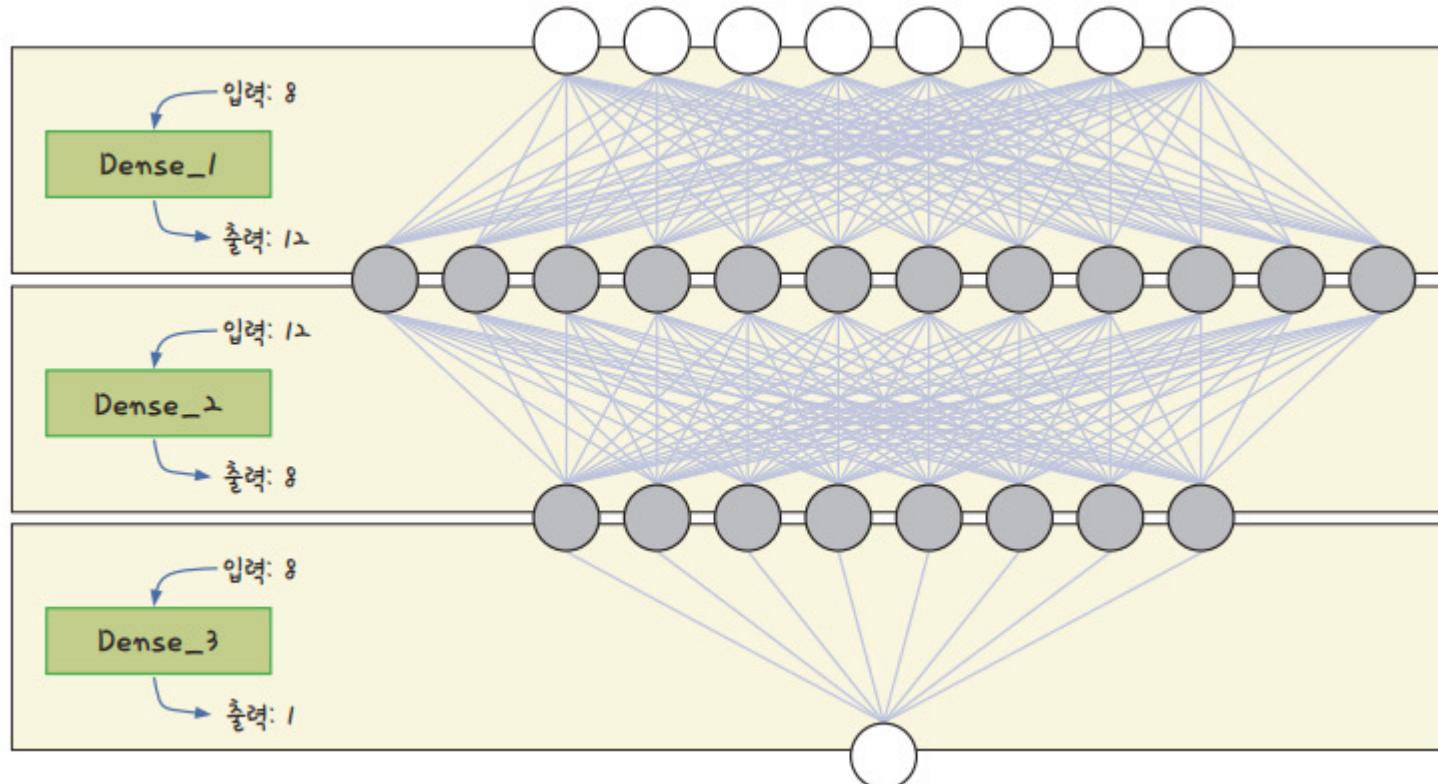
## ● 피마 인디언의 당뇨병 예측 실행

- ③ Param 부분은 파라미터 수, 즉 총 가중치와 바이어스 수의 합을 나타냄
- 예를 들어 첫 번째 층의 경우 입력 값 8개가 층 안에서 12개의 노드로 분산되므로 가중치가  $8 \times 12 = 96$ 개가 되고, 각 노드에 바이어스가 한 개씩 있으니 전체 파라미터 수는  $96 + 12 = 108$ 이 됨
- ④ 부분은 전체 파라미터를 합산한 값
- Trainable params는 학습을 진행하면서 업데이트가 된 파라미터들이고, Non-trainable params는 업데이트가 되지 않은 파라미터 수를 나타냄



# 5 피마 인디언의 당뇨병 예측 실행

▼ 그림 11-6 | 피마 인디언 당뇨병 예측 모델의 구조





# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

은닉층의 개수를 왜 두 개로 했나요? 그리고 노드의 수는 왜 각각 12개와 8개로 했나요?

- 입력과 출력의 수는 정해져 있지만, 은닉층은 몇 층으로 할지, 은닉층 안의 노드는 몇 개로 할지에 대한 정답은 없음
- 자신의 프로젝트에 따라 설정해야 함
- 여러 번 반복하면서 최적 값을 찾아내는 것이 좋으며, 여기서는 임의의 수로 12와 8을 설정했고 설명의 편의성을 위해 두 개의 은닉층을 만들었음
- 직접 노드의 수와 은닉층의 개수를 바꾸어 보면서 더 좋은 정확도가 나오는지 확인 가능



# 5 피마 인디언의 당뇨병 예측 실행

- 피마 인디언의 당뇨병 예측 실행

## 실습 | 피마 인디언의 당뇨병 예측하기



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# pandas 라이브러리를 불러옵니다.
import pandas as pd

# 피마 인디언 당뇨병 데이터셋을 불러옵니다.
df = pd.read_csv('./data/pima-indians-diabetes3.csv')
```



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

```
X = df.iloc[:,0:8]    # 세부 정보를 X로 지정합니다.  
y = df.iloc[:,8]      # 당뇨병 여부를 y로 지정합니다.  
  
# 모델을 설정합니다.  
model = Sequential()  
model.add(Dense(12, input_dim=8, activation='relu', name='Dense_1'))  
model.add(Dense(8, activation='relu', name='Dense_2'))  
model.add(Dense(1, activation='sigmoid', name='Dense_3'))  
model.summary()
```



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

```
# 모델을 컴파일합니다.  
  
model.compile(loss='binary_crossentropy', optimizer='adam',  
metrics=[ 'accuracy' ])  
  
# 모델을 실행합니다.  
  
history = model.fit(X, y, epochs=100, batch_size=5)
```



# 5 피마 인디언의 당뇨병 예측 실행

## ● 피마 인디언의 당뇨병 예측 실행

### 실행 결과

Epoch 1/100

154/154 [=====] - 2s 2ms/step - loss: 1.7955 -

accuracy: 0.5430

... (중략) ...

Epoch 100/100

154/154 [=====] - 0s 2ms/step - loss: 0.5705 -

accuracy: 0.7161

- 100번을 반복한 현재, 약 71.61%의 정확도를 보이고 있음



# 딥러닝 기본기 다지기

# 다중 분류 문제 해결하기

---

- 1 다중 분류 문제
- 2 상관도 그래프
- 3 원-핫 인코딩
- 4 소프트맥스
- 5 아이리스 품종 예측의 실행



# 1 다중 분류 문제

---



# 1 다중 분류 문제

## ● 다중 분류 문제

- 아이리스는 그 꽃봉오리가 마치 먹물을 머금은 붓과 같다 하여 우리나라에서는 ‘붓꽃’이라고 부르는 아름다운 꽃
- 아이리스는 꽃잎의 모양과 길이에 따라 여러 가지 품종으로 나뉨
- 사진을 보면 품종마다 비슷해 보이는데요. 과연 딥러닝을 사용해서 이들을 구별해 낼 수 있을까?

## ▼ 그림 12-1 | 아이리스의 품종



Iris-virginica



Iris-setosa



Iris-versicolor



# 1 다중 분류 문제

## ● 다중 분류 문제

- 아이리스 품종 예측 데이터는 예제 파일의 data 폴더에서 찾을 수 있음(data/iris3.csv)
- 데이터의 구조는 다음과 같음

▼ 그림 12-2 | 아이리스 데이터의 샘플, 속성, 클래스 구분

	정보 1	정보 2	정보 3	정보 4	품종
샘플	1번째 아이리스	5.1	3.5	4.0	0.2
	2번째 아이리스	4.9	3.0	1.4	0.2
	3번째 아이리스	4.7	3.2	1.3	0.3
	...	...	...	...	...
	150번째 아이리스	5.9	3.0	5.1	1.8
					Iris-virginica



# 1 다중 분류 문제

- 샘플 수: 150
- 속성 수: 4
  - 정보 1: 꽃받침 길이(sepal length, 단위: cm)
  - 정보 2: 꽃받침 너비(sepal width, 단위: cm)
  - 정보 3: 꽃잎 길이(petal length, 단위: cm)
  - 정보 4: 꽃잎 너비(petal width, 단위: cm)
- 클래스: Iris-setosa, Iris-versicolor, Iris-virginica



# 1 다중 분류 문제

## ● 다중 분류 문제

- 속성을 보니 우리가 앞서 다루었던 것과 중요한 차이가 있음
- 바로 클래스가 두 개가 아니라 세 개
- 즉, 참(1)과 거짓(0)으로 해결하는 것이 아니라, 여러 개 중에 어떤 것이 답인지 예측하는 문제
- 이렇게 여러 개의 답 중 하나를 고르는 분류 문제를 **다중 분류**(multi classification)라고 함
- 다중 분류 문제는 둘 중에 하나를 고르는 이항 분류(binary classification)와는 접근 방식이 조금 다름
- 아이리스 품종을 예측하는 실습을 통해 다중 분류 문제 해결



## 2 상관도 그래프

---



## 2 상관도 그래프

### ● 상관도 그래프

- 먼저 데이터의 일부를 불러와 내용을 보자

```
import pandas as pd

# 아이리스 데이터를 불러옵니다.
df = pd.read_csv('./data/iris3.csv')

df.head() # 첫 다섯 줄을 봅니다.
```



## 2 상관도 그래프

### ● 상관도 그래프

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa



## 2 상관도 그래프

### ● 상관도 그래프

- 이번에는 시본(seaborn) 라이브러리에 있는 pairplot() 함수를 써서 전체 상관도를 볼 수 있는 그래프를 출력해 보자

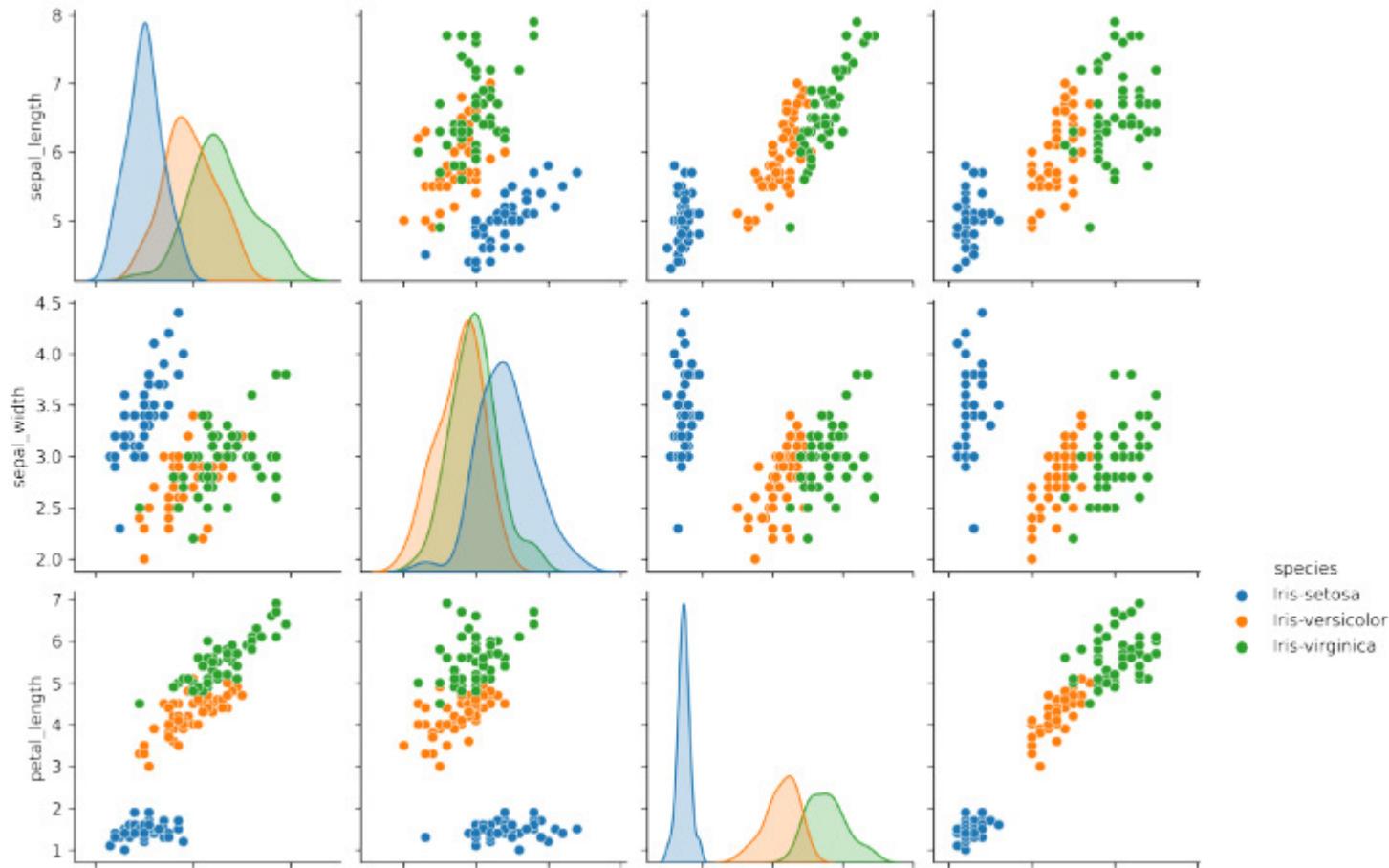
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.pairplot(df, hue='species');
plt.show()
```



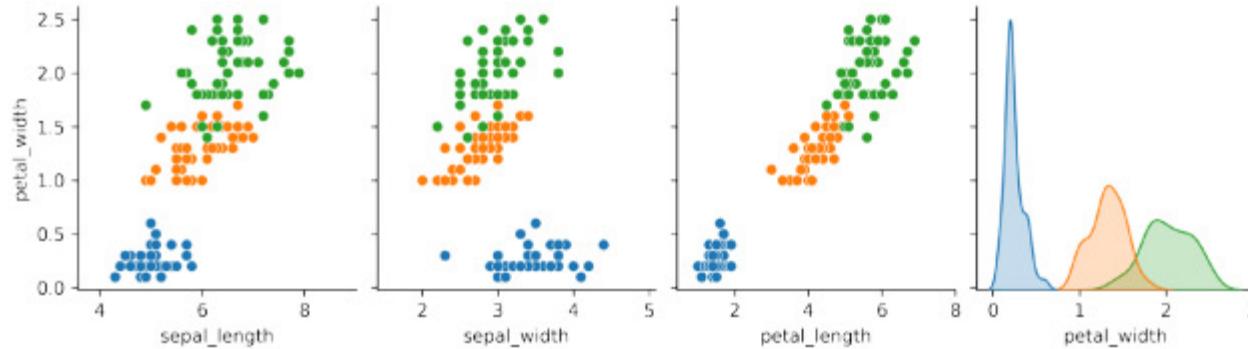
## 2 상관도 그래프

▼ 그림 12-3 | pairplot 함수로 데이터 한번에 보기





## 2 상관도 그래프





## 2 상관도 그래프

### ● 상관도 그래프

- 아피 그림을 상관도 그래프라고 함
- 이를 통해 각 속성별 데이터 분포와 속성 간의 관계를 한눈에 볼 수 있음
- pairplot() 함수 설정 중 hue 옵션은 주어진 데이터 중 어떤 카테고리를 중심으로 그래프를 그릴지 정해 주게 되는데, 우리는 품종(❶ species)에 따라 보여지게끔 지정
- 그래프 각각의 가로축과 세로축은 서로 다른 속성을 나타내며, 이러한 속성에 따라 품종이 어떻게 분포되는지 알 수 있음
- 가운데 대각선 위치에 있는 그림은 가로축과 세로축이 같으므로 단순히 해당 속성에 따라 각 품종들이 어떻게 분포하는지 보여 줌
- 이러한 분석을 통해 사진상으로 비슷해 보이던 꽃잎과 꽃받침의 크기와 너비가 품종별로 어떤 차이가 있는지 알 수 있음



## 3 원-핫 인코딩

---



# 3 원-핫 인코딩

## ● 원-핫 인코딩

- 케라스를 이용해 아이리스의 품종 예측
- Iris-setosa, Iris-virginica 등 데이터 안에 문자열이 포함되어 있음
- 불러온 데이터 프레임을 X와 y로 나누겠음

```
X = df.iloc[:,0:4]  
y = df.iloc[:,4]
```



# 3 원-핫 인코딩

## ● 원-핫 인코딩

- X와 y의 첫 다섯 줄을 출력해 보자

```
print(X[0:5])  
print(y[0:5])
```



# 3 원-핫 인코딩

## ● 원-핫 인코딩

실행 결과

```
  sepal_length  sepal_width  petal_length  petal_width
0          5.1         3.5          1.4         0.2
1          4.9         3.0          1.4         0.2
2          4.7         3.2          1.3         0.2
3          4.6         3.1          1.5         0.2
4          5.0         3.6          1.4         0.2
0    Iris-setosa
1    Iris-setosa
2    Iris-setosa
3    Iris-setosa
4    Iris-setosa
Name: species, dtype: object
```



# 3 원-핫 인코딩

## ● 원-핫 인코딩

- 우리가 저장한  $y$ 의 값이 숫자가 아닌 문자
- 딥러닝에서는 계산을 위해 문자를 모두 숫자형으로 바꾸어 주어야 함
- 이를 위해서는 다음과 같이 처리
- 먼저 아이리스 꽃의 종류는 ①처럼 세 종류
- ②처럼 각각의 이름으로 세 개의 열을 만든 후 ③처럼 자신의 이름이 일치하는 경우 1로, 나머지는 0으로 바꾸어 줌



### 3 원-핫 인코딩

▼ 그림 12-4 | 원-핫 인코딩

species	setosa	versicolor	virginica
setosa	1	0	0
versicolor	0	1	0
virginica	0	0	1
versicolor	0	1	0
...	...	...	...



# 3 원-핫 인코딩

## ● 원-핫 인코딩

- 여러 개의 값으로 된 문자열을 0과 1로만 이루어진 형태로 만들어 주는 과정을 **원-핫 인코딩**(one-hot encoding)이라고 함
- 원-핫 인코딩은 판다스가 제공하는 `get_dummies()` 함수를 사용하면 간단하게 해낼 수 있음

```
# 원-핫 인코딩 처리를 합니다.  
y = pd.get_dummies(y)  
  
# 원-핫 인코딩 결과를 확인합니다.  
print(y[0:5])
```



# 3 원-핫 인코딩

## ● 원-핫 인코딩

실행 결과

	Iris-setosa	Iris-versicolor	Iris-virginica
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0



## 4 소프트맥스

---



# 4 소프트맥스

## ● 소프트맥스

- 모델을 만들어 줄 차례
- 다음 코드를 보면서 이전에 실행했던 피마 인디언의 당뇨병 예측과 무엇이 달라졌는지 찾아보기 바람

```
# 모델 설정
```

```
model = Sequential()  
  
model.add(Dense(12, input_dim=4, activation='relu'))  
model.add(Dense(8, activation='relu'))  
model.add(Dense(3, activation='softmax'))  
  
model.summary()
```

```
# 모델 컴파일
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam',  
metrics=['accuracy'])
```



# 4 소프트맥스

## ● 소프트맥스

- 세 가지가 달라졌음
- 첫째 출력층의 노드 수가 3으로 바뀜
- 활성화 함수가 softmax로 바뀜
- 마지막으로 컴파일 부분에서 손실 함수 부분이 categorical\_crossentropy로 바뀜



# 4 소프트맥스

## ● 소프트맥스

- 먼저 출력 부분에 대해 알아보자
- 이전까지 우리는 출력이 0~1 중 하나의 값으로 나왔음
- 예를 들어 당뇨인지 아닌지에 대한 예측 값이 시그모이드 함수를 거치며 0~1 사이의 값 중 하나로 변환되어 0.5 이상이면 당뇨로, 이하이면 정상으로 판단
- 이항 분류의 경우 출력 값이 하나면 됨
- 이번 예제에서는 예측해야 할 값이 세 가지로 늘었음
- 즉, 각 샘플마다 이것이 setosa일 확률, versicolor일 확률, 그리고 virginica일 확률을 따로따로 구해야 한다는 것
- 예를 들어 예측 결과는 그림 12-5와 같은 형태로 나타남



## 4 소프트맥스

▼ 그림 12-5 | 소프트맥스

샘플	setosa일 확률	versicolor일 확률	virginica일 확률
1번 샘플	0.2	0.7	0.1
2번 샘플	0.8	0.1	0.1
3번 샘플	0.2	0.2	0.6

예측 실행  
→



# 4 소프트맥스

## ● 소프트맥스

- 이렇게 세 가지의 확률을 모두 구해야 하므로 시그모이드 함수가 아닌 다른 함수가 필요함
- 이때 사용되는 함수가 바로 소프트맥스 함수
- 소프트맥스 함수는 그림 12-5와 같이 각 항목당 예측 확률을 0과 1 사이의 값으로 나타내 주는데, 이때 각 샘플당 예측 확률의 총합이 1인 형태로 바꾸어 주게 됨(예를 들어 1번 샘플의 경우  $0.2 + 0.7 + 0.1 = 1$ 이 됨)
- activation란에 ‘softmax’라고 적어 주는 것으로 소프트맥스 함수를 바로 적용 할수 있음
- 마찬가지로 손실 함수도 이전과는 달라져야 함
- 이항 분류에서 binary\_crossentropy를 썼다면, 다항 분류에서는 categorical\_crossentropy를 쓰면 됨



## 5 아이리스 품종 예측의 실행

---



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

- 이제 모든 소스 코드를 모아 보면 다음과 같음

### 실습 | 아이리스 품종 예측하기



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

```
# 아이리스 데이터를 불러옵니다.  
df = pd.read_csv('./data/iris3.csv')  
  
# 속성을 X, 클래스를 y로 저장합니다.  
X = df.iloc[:,0:4]  
y = df.iloc[:,4]  
  
# 원-핫 인코딩 처리를 합니다.  
y = pd.get_dummies(y)
```



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

```
# 모델 설정
```

```
model = Sequential()  
  
model.add(Dense(12, input_dim=4, activation='relu'))  
model.add(Dense(8, activation='relu'))  
model.add(Dense(3, activation='softmax'))  
  
model.summary()
```

```
# 모델 컴파일
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam',  
metrics=['accuracy'])
```

```
# 모델 실행
```

```
history = model.fit(X, y, epochs=50, batch_size=5)
```



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

실행 결과

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
dense (Dense)	(None, 12)	60
<hr/>		
dense_1 (Dense)	(None, 8)	104
<hr/>		
dense_2 (Dense)	(None, 3)	27
<hr/>		



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

Total params: 191

Trainable params: 191

Non-trainable params: 0

---

Epoch 1/30

30/30 [=====] - 2s 2ms/step - loss: 1.4888 - accuracy: 0.3333

... (중략) ...

Epoch 30/30

30/30 [=====] - 0s 2ms/step - loss: 0.2746 - accuracy: 0.9667



# 5 아이리스 품종 예측의 실행

## ● 아이리스 품종 예측의 실행

- model.summary()를 사용해 두 개의 은닉층에 각각 12개와 여덟 개의 노드가 만들어졌고, 출력은 세 개임을 확인할 수 있음
- 결과는 30번 반복했을 때 정확도가 96.0% 나왔음
- 꽃의 너비와 길이를 담은 150개의 데이터 중 144개의 꽃 종류를 정확히 맞추었다는 의미
- 앞으로는 이렇게 측정된 정확도를 어떻게 신뢰할 수 있는지, 예측 결과의 신뢰도를 높이는 방법에 대해 알아보도록 함



# 딥러닝 기본기 다지기

# 모델 성능 검증하기

---

1 데이터의 확인과 예측 실행

2 과적합 이해하기

3 학습셋과 테스트셋

4 모델 저장과 재사용

5 k겹 교차 검증



# 모델 성능 검증하기

## ● 모델 성능 검증하기

- 1986년 제프리 힌튼 교수가 오차 역전파를 발표한 직후, 존스 홉킨스의 세즈노프스키(Sejnowski) 교수는 오차 역전파가 은닉층의 가중치를 실제로 업데이트시키는 것을 확인하고 싶었음
- 그는 광석과 일반 암석에 수중 음파 탐지기를 쓴 후 결과를 모아 데이터셋을 준비했고, 음파 탐지기의 수신 결과만 보고 광석인지 일반 암석인지를 구분하는 모델을 만들었음
- 그가 측정한 결과의 정확도는?
- 앞으로 세즈노프스키 교수가 했던 초음파 광물 예측 실험을 텐서플로로 재현해 보고, 이렇게 구해진 실험 정확도를 평가하는 방법과 성능을 향상시키는 중요한 머신 러닝 기법들에 대해 알아보자



# 모델 성능 검증하기

- 모델 성능 검증하기





# 1 데이터의 확인과 예측 실행

---



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

- 먼저 데이터를 불러와 첫 다섯 줄을 확인해 보자

```
# pandas 라이브러리를 불러옵니다.  
import pandas as pd  
  
# 광물 데이터를 불러옵니다.  
df = pd.read_csv('./data/sonar3.csv', header=None)  
  
df.head() # 첫 다섯 줄을 봅니다.
```



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

실행 결과														
	0	1	2	3	4	5	...	55	56	57	58	59	60	
0	0.0200	0.0371	0.0428	0.0207	0.0954	0.0986	...	0.0167	0.0180	0.0084	0.0090	0.0032	0	
1	0.0453	0.0523	0.0843	0.0689	0.1183	0.2583	...	0.0191	0.0140	0.0049	0.0052	0.0044	0	
2	0.0262	0.0582	0.1099	0.1083	0.0974	0.2280	...	0.0244	0.0316	0.0164	0.0095	0.0078	0	
3	0.0100	0.0171	0.0623	0.0205	0.0205	0.0368	...	0.0073	0.0050	0.0044	0.0040	0.0117	0	
4	0.0762	0.0666	0.0481	0.0394	0.0590	0.0649	...	0.0015	0.0072	0.0048	0.0107	0.0094	0	

5 rows × 61 columns



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

- 전체가 61개의 열로 되어 있고, 마지막 열이 광물의 종류를 나타냄
- 일반 암석일 경우 0, 광석일 경우 1로 표시되어 있음
- 첫 번째 열부터 60번째 열까지는 음파 주파수의 에너지를 0에서 1 사이의 숫자로 표시하고 있음
- 이제 일반 암석과 광석이 각각 몇 개나 포함되어 있는지 알아보자

```
df[60].value_counts()
```



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

실행 결과

1 111

0 97

Name: 60, dtype: int64



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

- 광석이 111개, 일반 암석이 97개, 총 208개의 샘플이 준비되어 있는 것을 알 수 있음
- 이제 다음과 같이 1~60번째 열을 X 변수에 저장하고 광물의 종류는 y로 저장

```
X = df.iloc[:, 0:60]  
y = df.iloc[:, 60]
```



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

- 이후 앞서 했던 그대로 딥러닝을 실행

### 실습 | 초음파 광물 예측하기: 데이터 확인과 실행



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 모델을 설정합니다.
model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
```



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

```
model.add(Dense(10, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))  
  
# 모델을 컴파일합니다.  
model.compile(loss='binary_crossentropy', optimizer='adam',  
metrics=['accuracy'])  
  
# 모델을 실행합니다.  
history = model.fit(X, y, epochs=200, batch_size=10)
```



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

### 실행 결과

Epoch 1/200

21/21 [=====] - 4s 4ms/step - loss: 0.6951 -

accuracy: 0.5000

... (중략) ...

Epoch 200/200

21/21 [=====] - 0s 2ms/step - loss: 0.0327 -

accuracy: 1.0000



# 1 데이터의 확인과 예측 실행

## ● 데이터의 확인과 예측 실행

- 200번 반복되었을 때의 결과를 보니 정확도가 100%
- 정말로 어떤 광물이든 100%의 확률로 판별해 내는 모델이 만들어진 것일까?



## 2 과적합 이해하기

---



## 2 과적합 이해하기

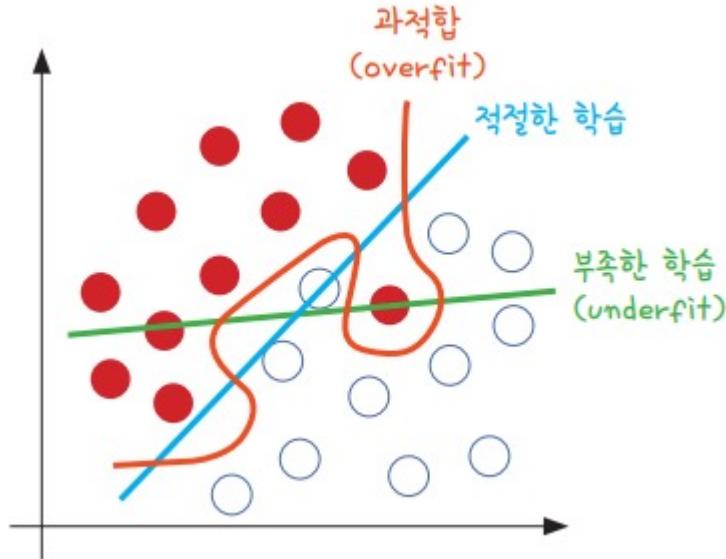
### ● 과적합 이해하기

- 이제 과적합 문제가 무엇인지 알아보고 이를 어떻게 해결하는지 살펴보자
- 과적합(overfitting)이란 모델이 학습 데이터셋 안에서는 일정 수준 이상의 예측 정확도를 보이지만, 새로운 데이터에 적용하면 잘 맞지 않는 것을 의미
- 그림 13-1의 그래프에서 빨간색 선을 보면 주어진 샘플에 정확히 맞게끔 그어져 있음
- 이 선은 너무 주어진 샘플에만 최적화되어 있음
- 지금 그어진 선을 새로운 데이터에 적용하면 정확한 분류가 어려워진다는 의미
- 과적합은 층이 너무 많거나 변수가 복잡해서 발생하기도 하고 테스트셋과 학습셋이 중복될 때 생기기도 함
- 특히 딥러닝은 학습 단계에서 입력층, 은닉층, 출력층의 노드들에 상당히 많은 변수가 투입
- 딥러닝을 진행하는 동안 과적합에 빠지지 않게 늘 주의해야 함



## 2 과적합 이해하기

▼ 그림 13-1 | 과적합이 일어난 경우(빨간색)와 학습이 제대로 이루어지지 않은 경우(초록색)





## 3 학습셋과 테스트셋

---



### 3 학습셋과 테스트셋

#### ● 학습셋과 테스트셋

- 그렇다면 과적합을 방지하려면 어떻게 해야 할까?
- 먼저 학습을 하는 데이터셋과 이를 테스트할 데이터셋을 완전히 구분한 후 학습과 동시에 테스트를 병행하며 진행하는 것이 한 방법
- 예를 들어 데이터셋이 총 100개의 샘플로 이루어져 있다면 다음과 같이 두 개의 셋으로 나눔

#### ▼ 그림 13-2 | 학습셋과 테스트셋의 구분

70개의 샘플은 학습셋으로

30개의 샘플은 테스트셋으로



# 3 학습셋과 테스트셋

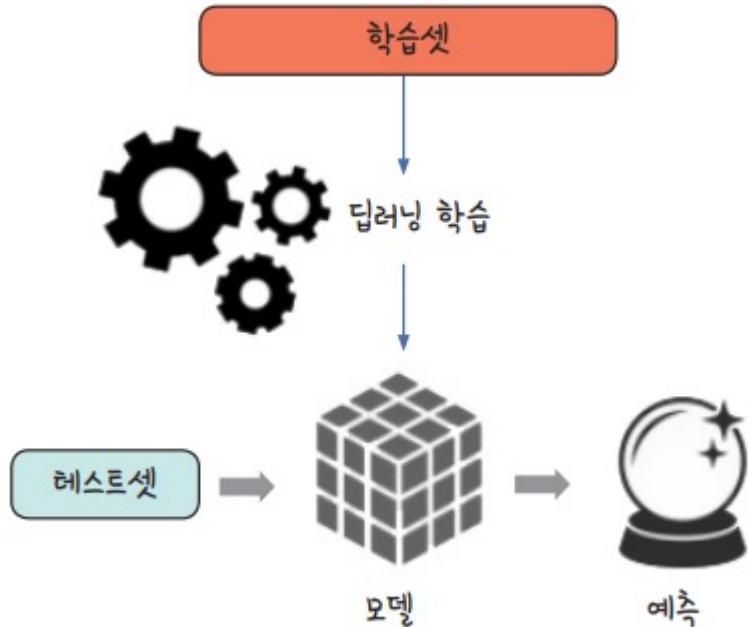
## ● 학습셋과 테스트셋

- 신경망을 만들어 70개의 샘플로 학습을 진행한 후 이 학습의 결과를 저장
- 이렇게 저장된 파일을 ‘모델’이라고 함
- 모델은 다른 셋에 적용할 경우 학습 단계에서 각인되었던 그대로 다시 수행
- 나머지 30개의 샘플로 실험해서 정확도를 살펴보면 학습이 얼마나 잘되었는지 알 수 있는 것
- 딥러닝 같은 알고리즘을 충분히 조절해 가장 나은 모델이 만들어지면, 이를 실생활에 대입해 활용하는 것이 바로 머신 러닝의 개발 순서



### 3 학습셋과 테스트셋

▼ 그림 13-3 | 학습셋과 테스트셋





# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 지금까지는 테스트셋을 만들지 않고 학습해 왔음
- 매번 정확도(accuracy)를 계산할 수 있었음
- 어째서 가능했을까?
- 지금까지 학습 데이터를 이용해 정확도를 측정한 것은 데이터에 들어 있는 모든 샘플을 그대로 테스트에 활용한 결과
- 이를 통해 학습이 진행되는 상황을 파악할 수는 있지만, 새로운 데이터에 적용했을 때 어느 정도의 성능이 나올지는 알 수 없음
- 머신 러닝의 최종 목적은 과거의 데이터를 토대로 새로운 데이터를 예측하는 것
- 즉, 새로운 데이터에 사용할 모델을 만드는 것이 최종 목적이므로 테스트셋을 만들어 정확한 평가를 병행하는 것이 매우 중요



# 3 학습셋과 테스트셋

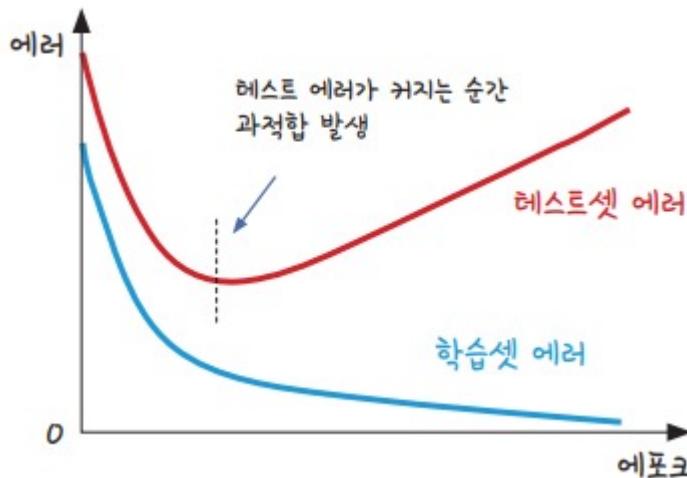
## ● 학습셋과 테스트셋

- 학습셋만 가지고 평가할 때, 층을 더하거나 에포크(epoch) 값을 높여 실행 횟수를 늘리면 정확도가 계속해서 올라갈 수 있음
- 학습 데이터셋만으로 평가한 예측 성공률이 테스트셋에서도 그대로 나타나지는 않음
- 즉, 학습이 깊어져 학습셋 내부에서 성공률은 높아져도 테스트셋에서는 효과가 없다면 과적합이 일어나고 있는 것
- 이를 그래프로 표현하면 그림 13-4와 같음



### 3 학습셋과 테스트셋

▼ 그림 13-4 | 학습이 계속되면 학습셋에서의 에러는 계속해서 작아지지만, 테스트셋에서는 과적합이 발생!





# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 학습을 진행해도 테스트 결과가 더 이상 좋아지지 않는 지점에서 학습을 멈추어야 함
- 이때 학습 정도가 가장 적절한 것으로 볼 수 있음
- 음파 광물 예측 데이터를 만든 세즈노프스키 교수가 실험 결과를 발표한 논문의 일부

▼ 그림 13-5 | 학습셋과 테스트셋 정확도 측정의 예(RP Gorman et.al., 1998)

TABLE 2  
Aspect-Angle Dependent Series

Number of Hidden Units	Average Performance on Training Sets (%)	Standard Deviation on Training Sets (%)	Average Performance on Testing Sets (%)	Standard Deviation on Testing Sets (%)
0	79.3	3.4	73.1	4.8
2	96.2	2.2	85.7	6.3
3	98.1	1.5	87.6	3.0
6	99.4	0.9	89.3	2.4
12	99.8	0.6	90.4	1.8
24	100.0	0.0	89.2	1.4

Summary of the results of the aspect-angle dependent series of experiments with training and testing sets selected to include all target aspect angles. The standard deviation shown is across networks with different initial conditions.



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 여기서 눈여겨보아야 할 부분은 은닉층(Number of Hidden Units) 개수가 올라감에 따라 학습셋의 예측률(Average Performance on Training Sets)과 테스트셋의 예측률(Average Performance on Testing Sets)의 변화
- 이 부분만 따로 뽑아 정리하면 표 13-1과 같음

▼ 표 13-1 | 은닉층 개수의 변화에 따른 학습셋 및 테스트셋의 예측률

은닉층 개수의 변화	학습셋의 예측률	테스트셋의 예측률
0	79.3	73.1
2	96.2	85.7
3	98.1	87.6
6	99.4	89.3
12	99.8	90.4
24	100	89.2



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 은닉층이 늘어날수록 학습셋의 예측률이 점점 올라가다가 결국 24개의 층에 이르면 100% 예측률을 보임
- 조금 전 실행했던 것과 같은 결과!
- 이 모델을 토대로 테스트한 결과는 어떤가?
- 테스트셋 예측률은 은닉층의 개수가 12개일 때 90.4%로 최고를 이루다 24개째에서는 다시 89.2%로 떨어지고 맘
- 즉, 식이 복잡해지고 학습량이 늘어날수록 학습 데이터를 통한 예측률은 계속해서 올라가지만, 적절하게 조절하지 않을 경우 테스트셋을 이용한 예측률은 오히려 떨어지는 것을 확인할 수 있음
- 예제에 주어진 데이터를 학습셋과 테스트셋으로 나누는 예제를 만들어 보자



### 3 학습셋과 테스트셋

#### ● 학습셋과 테스트셋

- 사이킷런(scikit-learn) 라이브러리 필요
- 사이킷런은 파이썬으로 머신 러닝을 실행할 때 필요한 전반적인 것들이 담긴 머신 러닝의 필수 라이브러리



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 저장된 X 데이터와 y 데이터에서 각각 정해진 비율(%)만큼 학습셋과 테스트셋으로 분리시키는 함수가 사이킷런의 `train_test_split()` 함수
- 다음과 같이 학습셋과 테스트셋을 만들 수 있음
- 학습셋을 70%, 테스트셋을 30%로 설정했을 때의 예

```
from sklearn.model_selection import train_test_split

# 학습셋과 테스트셋을 구분합니다.

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
shuffle=True)
```



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- `test_size`는 테스트셋의 비율을 나타냄
- 0.3은 전체 데이터의 30%를 테스트셋으로 사용하라는 것으로, 나머지 70%를 학습셋으로 사용하게 됨
- 이렇게 나누어진 학습셋과 테스트셋은 각각 `X_train`, `X_test`, `y_train`, `y_test`로 저장
- 모델은 앞서 만든 구조를 그대로 유지하고 대신 모델에 테스트 함수를 추가
- 만들어진 모델을 테스트셋에 적용하려면 `model.evaluate()` 함수를 사용하면 됨

```
score = model.evaluate(X_test, y_test)
print('Test accuracy:', score[1])
```



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- model.evaluate() 함수는 loss와 accuracy, 두 가지를 계산해 출력
- 이를 score로 저장하고 accuracy를 출력하도록 범위를 정했음
- 함수 내부에는 테스트셋 정보를 집어넣음



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 이제 전체 코드를 실행해 보자

### 실습 | 초음파 광물 예측하기: 학습셋과 테스트셋 구분



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split

import pandas as pd

# 광물 데이터를 불러옵니다.
df = pd.read_csv('./data/sonar3.csv', header=None)
```



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

```
# 음파 관련 속성을 X로, 광물의 종류를 y로 저장합니다.
```

```
X = df.iloc[:,0:60]
```

```
y = df.iloc[:,60]
```

```
# 학습셋과 테스트셋을 구분합니다.
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,  
shuffle=True)
```

```
# 모델을 설정합니다.
```

```
model = Sequential()
```

```
model.add(Dense(24, input_dim=60, activation='relu'))
```

```
model.add(Dense(10, activation='relu'))
```

```
model.add(Dense(1, activation='sigmoid'))
```



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

# 모델을 컴파일합니다.

```
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

# 모델을 실행합니다.

```
history = model.fit(X_train, y_train, epochs=200, batch_size=10)
```

# 모델을 테스트셋에 적용해 정확도를 구합니다.

```
score = model.evaluate(X_test, y_test)
print('Test accuracy:', score[1])
```



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

실행 결과

Epoch 1/200

15/15 [=====] - 0s 3ms/step - loss: 0.7158 -

accuracy: 0.4828

... (중략) ...

Epoch 200/200

15/15 [=====] - 0s 2ms/step - loss: 0.0590 -

accuracy: 0.9931

①

2/2 [=====] - 0s 2ms/step - loss: 0.4214 -

accuracy: 0.8413

Test accuracy: 0.841269850730896

②



### 3 학습셋과 테스트셋

#### ● 학습셋과 테스트셋

- 두 가지를 눈여겨보아야 함
- 첫째는 학습셋( $X_{\text{train}}$ 과  $y_{\text{train}}$ )을 이용해 200번의 학습을 진행했을 때 정확도가 99.31%라는 것(①)
- 따로 저장해 둔 테스트셋( $X_{\text{test}}$ 와  $y_{\text{test}}$ )에 이 모델을 적용하면 84.12%의 정확도를 보여 줌(②)



# 3 학습셋과 테스트셋

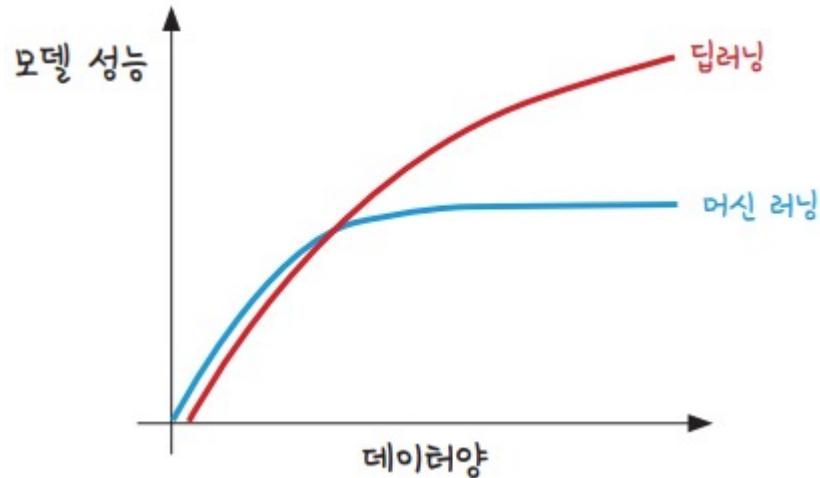
## ● 학습셋과 테스트셋

- 딥러닝, 머신 러닝의 목표는 학습셋에서만 잘 작동하는 모델을 만드는 것이 아님
- 새로운 데이터에 대해 높은 정확도를 안정되게 보여 주는 모델을 만드는 것이 목표
- 어떻게 하면 이러한 모델을 만들 수 있을까?
- 모델 성능의 향상을 위한 방법에는 크게 데이터를 보강하는 방법과 알고리즘을 최적화하는 방법이 있음
- 데이터를 이용해 성능을 향상시키려면 우선 충분한 데이터를 가져와 추가하면 됨
- 많이 알려진 다음 그래프는 특히 딥러닝의 경우 샘플 수가 많을수록 성능이 좋아짐을 보여 줌



### 3 학습셋과 테스트셋

▼ 그림 13-6 | 데이터의 증가와 딥러닝, 머신 러닝 성능의 상관관계





# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 데이터를 추가하는 것 자체가 어렵거나 데이터 추가만으로는 성능에 한계가 있을 수 있음
- 가지고 있는 데이터를 적절히 보완해 주는 방법을 사용
- 예를 들어 사진의 경우 사진 크기를 확대/축소한 것을 더해 보거나 위아래로 조금씩 움직여 넣어 보는 것
- 테이블형 데이터의 경우 너무 크거나 낮은 이상치가 모델에 영향을 줄 수 없도록 크기를 적절히 조절할 수 있음
- 시그모이드 함수를 사용해 전체를 0~1 사이의 값으로 변환하는 것이 좋은 예
- 또 교차 검증 방법을 사용해 가지고 있는 데이터를 충분히 이용하는 방법도 있음



# 3 학습셋과 테스트셋

## ● 학습셋과 테스트셋

- 다음으로 알고리즘을 이용해 성능을 향상하는 방법은 먼저 다른 구조로 모델을 바꾸어 가며 최적의 구조를 찾는 것
- 예를 들어 은닉층의 개수라든지, 그 안에 들어갈 노드의 수, 최적화 함수의 종류를 바꾸어 보는 것
- 앞서 이야기한 바 있지만, 딥러닝 설정에 정답은 없음
- 자신의 데이터에 꼭 맞는 구조를 계속해서 테스트해 보며 찾는 것이 중요
- 데이터에 따라서는 딥러닝이 아닌 랜덤 포레스트, XGBoost, SVM 등 다른 알고리즘이 더 좋은 성과를 보일 때도 있음
- 일반적인 머신 러닝과 딥러닝을 합해서 더 좋은 결과를 만드는 것도 가능
- 많은 경험을 통해 최적의 성능을 보이는 모델을 만드는 것이 중요



## 4 모델 저장과 재사용

---



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

- 학습이 끝난 후 지금 만든 모델을 저장하면 언제든 이를 불러와 다시 사용할 수 있음
- 학습 결과를 저장하려면 `model.save()` 함수를 이용해 모델 이름을 적어 저장

```
# 모델 이름과 저장할 위치를 함께 지정합니다.
```

```
model.save('./data/model/my_model.hdf5')
```



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

- hdf5 파일 포맷은 주로 과학 기술 데이터 작업에서 사용되는데, 크고 복잡한 데이터를 저장하는 데 사용
- 이를 다시 불러오려면 케라스 API의 `load_model` 함수를 사용
- 앞서 `Sequential` 함수를 불러온 모델 클래스 안에 함께 들어 있으므로, `Sequential` 뒤에 `load_model`을 추가해 다시 불러옴

```
from tensorflow.keras.models import Sequential, load_model
```

①



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

- 테스트를 위해 조금 전 만든 모델을 메모리에서 삭제

```
del model
```



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

- load\_model() 함수를 사용해서 조금 전 저장한 모델을 불러옴

```
# 모델이 저장된 위치와 이름까지 적어 줍니다.
```

```
model = load_model('./data/model/my_model.hdf5')
```



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

- 불러온 모델을 테스트셋에 적용해 정확도를 구함

```
score = model.evaluate(X_test, y_test)  
print('Test accuracy:', score[1])
```



# 4 모델 저장과 재사용

## ● 모델 저장과 재사용

실행 결과

```
2/2 [=====] - 0s 2ms/step - loss: 0.4214 -  
accuracy: 0.8413  
Test accuracy: 0.841269850730896
```

- 앞에서 실행한 것과 같은 값이 나오는 것을 확인할 수 있음



## 5 k겹 교차 검증

---



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 앞서 데이터가 충분히 많아야 모델 성능도 향상된다고 했음
- 이는 학습과 테스트를 위한 데이터를 충분히 확보할수록 세상에 나왔을 때 더 잘 작동하기 때문임
- 실제 프로젝트에서는 데이터를 확보하는 것이 쉽지 않거나 많은 비용이 발생하는 경우도 있음
- 가지고 있는 데이터를 활용하는 것이 중요
- 특히 학습셋을 70%, 테스트셋을 30%로 설정할 경우 30%의 테스트셋은 학습에 이용할 수 없다는 단점이 있음
- 이를 해결하기 위해 고안된 방법이 k겹 교차 검증(k-fold cross validation)



# 5 k겹 교차 검증

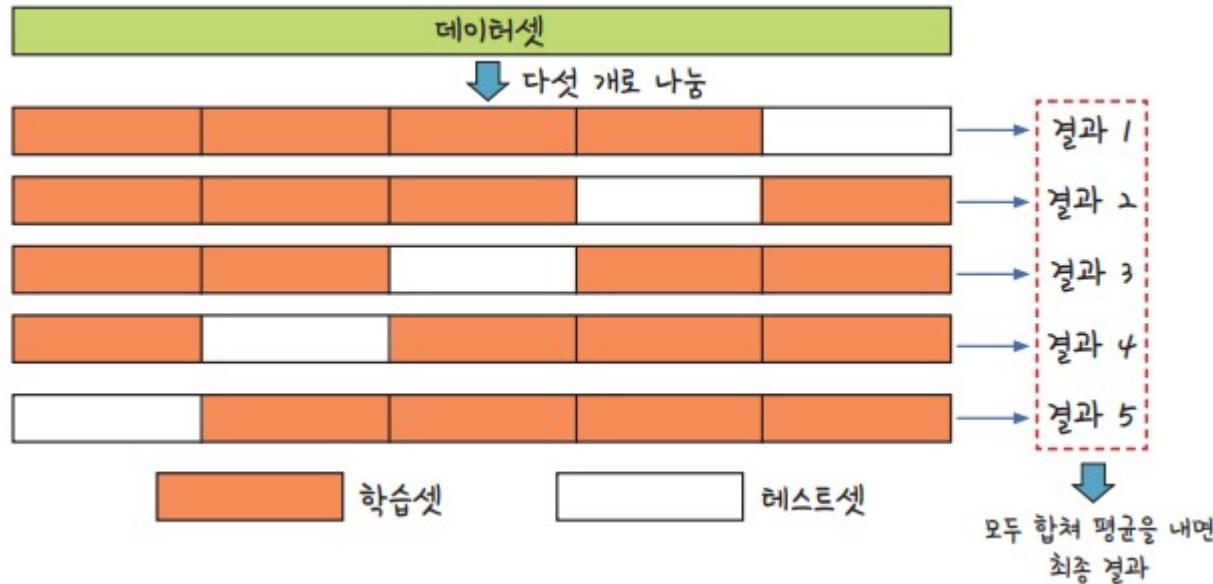
## ● k겹 교차 검증

- k겹 교차 검증이란 데이터셋을 여러 개로 나누어 하나씩 테스트셋으로 사용하고 나머지를 모두 합해서 학습셋으로 사용하는 방법
- 이렇게 하면 가지고 있는 데이터의 100%를 학습셋으로 사용할 수 있고, 또 동시에 테스트셋으로도 사용할 수 있음
- 예를 들어 5겹 교차 검증(5-fold cross validation)의 예가 그림 13-7에 설명되어 있음



# 5 k겹 교차 검증

▼ 그림 13-7 | 5겹 교차 검증의 도식





# 5 k겹 교차 검증

## ● k겹 교차 검증

- k데이터셋을 다섯 개로 나눈 후 그중 네 개를 학습셋으로, 나머지 하나를 테스트셋으로 만들어 다섯 번의 학습을 순차적으로 실시하는 것이 5겹 교차 검증
- 이제 초음파 광물 예측 예제를 통해 5겹 교차 검증을 실시해 보자



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 데이터를 원하는 수만큼 나누어 각각 학습셋과 테스트셋으로 사용되게 하는 함수는 사이킷런 라이브러리의 KFold() 함수
- 실습 코드에서 KFold()를 활용하는 부분만 뽑아 보면 다음과 같음

```
from sklearn.model_selection import KFold
k = 5 .....①
kfold = KFold(n_splits=k, shuffle=True) .....②
acc_score = [] .....③

for train_index, test_index in kfold.split(X): .....④
    X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 먼저 몇 개의 파일로 나눌 것인지 정해 ① k 변수에 넣음
- 사이킷런의 ② KFold() 함수를 불러옴
- 샘플이 어느 한쪽에 치우치지 않도록 shuffle 옵션을 True로 설정해 줌
- 정확도가 채워질 ③ acc\_score라는 이름의 빈 리스트를 준비
- ④ split()에 의해 k개의 학습셋, 테스트셋으로 분리되며 for 문에 의해 k번 반복



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 반복되는 각 학습마다 정확도를 구해 다음과 같이 acc\_score 리스트를 채움

```
accuracy = model.evaluate(X_test, y_test)[1] # 정확도를 구합니다.  
acc_score.append(accuracy) # acc_score 리스트에 저장합니다.
```

- k번의 학습이 끝나면 각 정확도들을 취합해 모델 성능을 평가



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 모든 코드를 모아 실행하면 다음과 같음

### 실습 | 초음파 광물 예측하기: k겹 교차 검증



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

import pandas as pd
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

# 광물 데이터를 불러옵니다.

```
df = pd.read_csv('./data/sonar3.csv', header=None)
```

# 음파 관련 속성을 X로, 광물의 종류를 y로 저장합니다.

```
X = df.iloc[:,0:60]
```

```
y = df.iloc[:,60]
```

# 몇 겹으로 나눌 것인지 정합니다.

```
k = 5
```

# KFold 함수를 불러옵니다. 분할하기 전에 샘플이 치우치지 않도록 섞어 줍니다.

```
kfold = KFold(n_splits=k, shuffle=True)
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

```
# 정확도가 채워질 빈 리스트를 준비합니다.  
acc_score = []  
  
def model_fn():  
    model = Sequential() # 딥러닝 모델의 구조를 시작합니다.  
    model.add(Dense(24, input_dim=60, activation='relu'))  
    model.add(Dense(10, activation='relu'))  
    model.add(Dense(1, activation='sigmoid'))  
    return model  
  
# k겹 교차 검증을 이용해 k번의 학습을 실행합니다.  
# for 문에 의해 k번 반복합니다.
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

```
# split()에 의해 k개의 학습셋, 테스트셋으로 분리됩니다.  
  
for train_index, test_index in kfold.split(X):  
    X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]  
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]  
  
    model = model_fn()  
    model.compile(loss='binary_crossentropy', optimizer='adam',  
metrics=['accuracy'])  
    history = model.fit(X_train, y_train, epochs=200, batch_size=10,  
verbose=0)  
  
    accuracy = model.evaluate(X_test, y_test)[1] # 정확도를 구합니다.  
    acc_score.append(accuracy) # 정확도 리스트에 저장합니다.
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

```
# k번 실시된 정확도의 평균을 구합니다.
```

```
avg_acc_score = sum(acc_score) / k
```

```
# 결과를 출력합니다.
```

```
print('정확도: ', acc_score)
```

```
print('정확도 평균: ', avg_acc_score)
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

실행 결과

```
2/2 [=====] - 0s 2ms/step - loss: 1.0080 -  
accuracy: 0.7381
```

```
2/2 [=====] - 0s 2ms/step - loss: 0.7071 -  
accuracy: 0.8095
```

```
2/2 [=====] - 0s 2ms/step - loss: 0.3312 -  
accuracy: 0.8810
```

```
2/2 [=====] - 0s 2ms/step - loss: 0.4377 -  
accuracy: 0.9024
```

```
2/2 [=====] - 0s 3ms/step - loss: 0.6416 - accuracy: 0.7317
```

정확도: [0.738095223903656, 0.8095238208770752, 0.8809523582458496,  
0.9024389982223511, 0.7317073345184326]

정확도 평균: 0.8125435471534729



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 다섯 번의 정확도를 구했음
- 학습이 진행되는 과정이 길어서 model.fit 부분에 verbose=0 옵션을 주어 학습 과정의 출력력을 생략



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 텐서플로 함수가 for 문에 포함되는 경우 다음과 같이 WARNING 메시지가 나오는 경우가 있음
- 텐서플로 구동에는 문제가 없으므로 그냥 진행하면 됨

```
WARNING:tensorflow:5 out of the last 9 calls to <function Model.make_
test_function.<locals>.test_function at 0x000001EDE50D60D0> triggered
tf.function retracing...
```



# 5 k겹 교차 검증

## ● k겹 교차 검증

- 이렇게 해서 가지고 있는 데이터를 모두 사용해 학습과 테스트를 진행
- 다음에는 학습 과정을 시각화해 보는 방법과 학습을 몇 번 반복할지(epochs) 스스로 판단하게 하는 방법 등을 알아보며 모델 성능을 향상시켜 보겠음