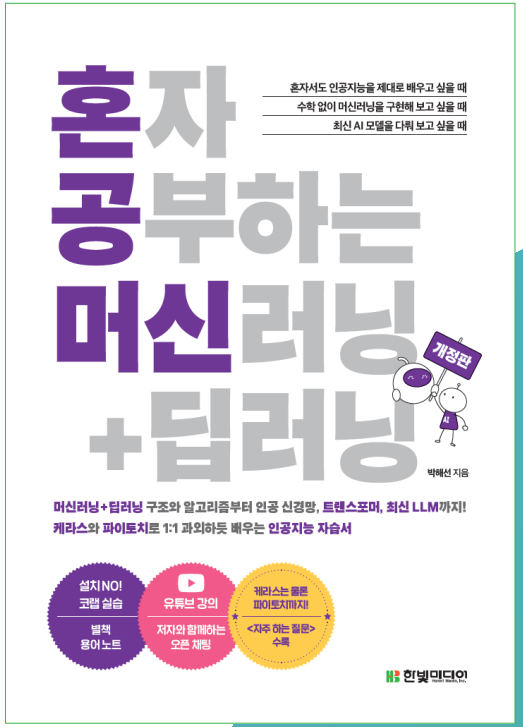
CHAPTER 07 딥러닝을 시작합니다

혼자 공부하는 머신러닝 + 딥러닝 (개정판)



한국공학대학교 게임공학과
이재영

시작하기 전에

지은이 / 박해선

기계공학을 전공했으나 졸업 후엔 줄곧 코드를 읽고 쓰는 일을 했다. 머신러닝과 딥러닝에 관한 책을 집필하고 번역하면서 소프트웨어와 과학의 경계를 흥미롭게 탐험하고 있다.

『핸즈온 머신러닝 2판』(한빛미디어, 2020)을 포함해서 여러 권의 머신러닝, 딥러닝 책을 우리말로 옮겼고 『Do it! 딥러닝 입문』(이지스퍼블리싱, 2019)을 집필했다.

- 교재의 모든 코드는 웹 브라우저에서 파이썬 코드를 실행할 수 있는 구글 코랩(Colab)을 사용하여 작성했습니다.
- 사용할 실습 환경은 네트워크에 연결된 컴퓨터와 구글 계정입니다.

학습 로드맵

머신러닝편 **01~06장**

딥러닝만 먼저 배우고 싶다면 01~04장을 읽은 후 07장으로 건너뛰어도 좋습니다.

딥러닝편 **07~10장**

07장을 읽은 후 08장과 09장은 순서대로 읽지 않아도 괜찮습니다. 10장을 읽기 전에 07장과 09장을 읽는 것이 좋습니다.

난이도

START

01 나의 첫 머신러닝

02 데이터 다루기

03 회귀 알고리즘과 모델 규제

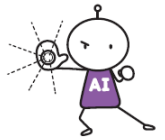
2번 보기

04 다양한 분류 알고리즘

05 트리 알고리즘

06 비지도 학습

07 딥러닝을 시작합니다



08 이미지를 위한 인공 신경망

09 텍스트를 위한 인공 신경망

10 언어 모델을 위한 신경망

GOAL

이 책의 학습 목표

- CHAPTER 01: 나의 첫 머신러닝
  - 인공지능, 머신러닝, 딥러닝의 차이점을 이해합니다.
  - 구글 코랩 사용법을 배웁니다.
  - 첫 번째 머신러닝 프로그램을 만들고 머신러닝의 기본 작동 원리를 이해합니다.
- CHAPTER 02: 데이터 다루기
  - 머신러닝 알고리즘에 주입할 데이터를 준비하는 방법을 배웁니다.
  - 데이터 형태가 알고리즘에 미치는 영향을 이해합니다.
- CHAPTER 03: 회귀 알고리즘과 모델 규제
  - 지도 학습 알고리즘의 한 종류인 회귀 알고리즘에 대해 배웁니다.
  - 다양한 선형 회귀 알고리즘의 장단점을 이해합니다.
- CHAPTER 04: 다양한 분류 알고리즘
  - 로지스틱 회귀, 확률적 경사 하강법과 같은 분류 알고리즘을 배웁니다.
  - 이진 분류와 다중 분류의 차이를 이해하고 클래스별 확률을 예측합니다.
- CHAPTER 05: 트리 알고리즘
  - 성능이 좋고 이해하기 쉬운 트리 알고리즘에 대해 배웁니다.
  - 알고리즘의 성능을 최대화하기 위한 하이퍼파라미터 튜닝을 실습합니다.
  - 여러 트리를 합쳐 일반화 성능을 높일 수 있는 앙상블 모델을 배웁니다.

이 책의 학습 목표

- **CHAPTER 06: 비지도 학습**
  - 타깃이 없는 데이터를 사용하는 비지도 학습과 대표적인 알고리즘을 소개합니다.
  - 대표적인 군집 알고리즘인 k-평균과 DBSCAN을 배웁니다.
  - 대표적인 차원 축소 알고리즘인 주성분 분석(PCA)을 배웁니다.
- **CHAPTER 07: 딥러닝을 시작합니다**
  - 딥러닝의 핵심 알고리즘인 인공 신경망을 배웁니다.
  - 대표적인 인공 신경망 라이브러리인 텐서플로와 케라스를 소개합니다.
  - 인공 신경망 모델의 훈련을 돕는 도구를 익힙니다.
- **CHAPTER 08: 이미지를 위한 인공 신경망**
  - 이미지 분류 문제에 뛰어난 성능을 발휘하는 합성곱 신경망의 개념과 구성 요소에 대해 배웁니다.
  - 케라스 API로 합성곱 신경망을 만들어 패션 MNIST 데이터에서 성능을 평가해 봅니다.
  - 합성곱 층의 필터와 활성화 출력을 시각화하여 합성곱 신경망이 학습한 내용을 고찰해 봅니다.
- **CHAPTER 09: 텍스트를 위한 인공 신경망**
  - 텍스트와 시계열 데이터 같은 순차 데이터에 잘 맞는 순환 신경망의 개념과 구성 요소에 대해 배웁니다.
  - 케라스 API로 기본적인 순환 신경망에서 고급 순환 신경망을 만들어 영화 감상평을 분류하는 작업에 적용해 봅니다.
  - 순환 신경망에서 발생하는 문제점과 이를 극복하기 위한 해결책을 살펴봅니다.
- **CHAPTER 10: 언어 모델을 위한 신경망**
  - 어텐션 메커니즘과 트랜스포머에 대해 배웁니다.
  - 트랜스포머로 상품 설명 요약하기를 학습합니다.
  - 대규모 언어 모델로 텍스트 생성하기를 익힙니다.

Contents

- **CHAPTER 07: 딥러닝을 시작합니다**

CHAPTER 07 딥러닝을 시작합니다

패션 럭키백을 판매합니다!

학습목표

- 딥러닝의 핵심 알고리즘인 인공 신경망을 배웁니다.
- 대표적인 인공 신경망 라이브러리인 텐서플로와 케라스를 소개합니다.
- 인공 신경망 모델의 훈련을 돕는 도구를 익힙니다.

SECTION 7-1 인공 신경망(1)

○ 패션 MNIST

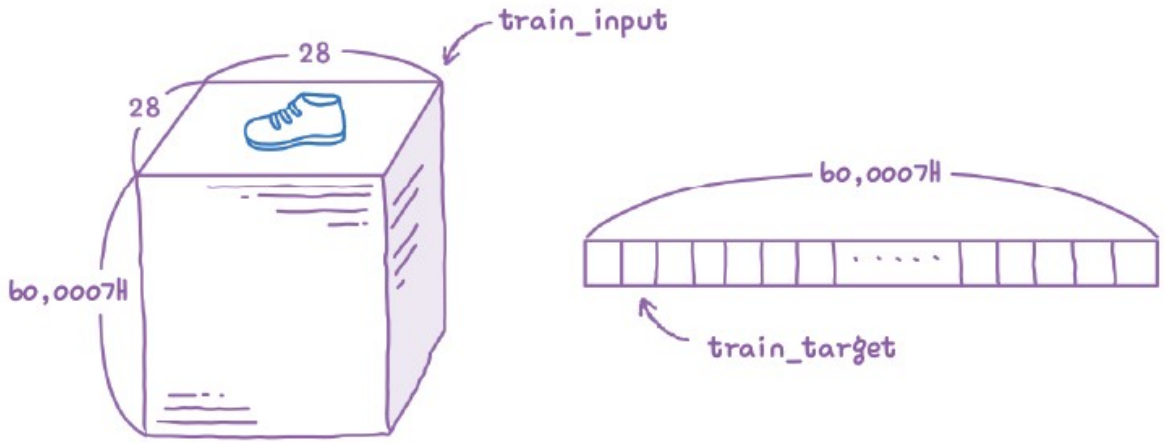
- 패션 MNIST 데이터셋은 10종류의 패션 아이템으로 구성
- 케라스 Keras 패키지를 임포트하고 패션 MNIST 데이터를 다운로드

```
import keras
(train_input, train_target), (test_input, test_target) =\
    keras.datasets.fashion_mnist.load_data()
```

- 전달받은 데이터의 크기를 확인

```
print(train_input.shape, train_target.shape)
```

→ (60000, 28, 28) (60000,)



SECTION 7-1 인공 신경망(2)

○ 패션 MNIST

- **테스트 세트의 크기 확인**

```
print(test_input.shape, test_target.shape)
```

→ (10000, 28, 28) (10000,)

- **훈련 데이터에서 몇 개의 샘플을 그림으로 출력**

```
import matplotlib.pyplot as plt
fig, axs = plt.subplots(1, 10, figsize=(10,10))
for i in range(10):
    axs[i].imshow(train_input[i], cmap='gray_r')
    axs[i].axis('off')
plt.show()
```

→

- **슬라이싱 연산자를 사용해서 처음 10개 샘플의 타깃값을 확인**

```
print([train_target[i] for i in range(10)])
```

→ [9, 0, 0, 3, 0, 2, 7, 2, 5, 5]

SECTION 7-1 인공 신경망(3)

○ 패션 MNIST

- 패션 MNIST의 타깃은 0~9까지의 숫자 레이블로 구성

레이블	0	1	2	3	4	5	6	7	8	9
패션 아이템	티셔츠	바지	스웨터	드레스	코트	샌달	셔츠	스니커즈	가방	앵클부츠

- 넘파이 unique( ) 함수로 레이블 당 샘플 개수 확인
  - 0~9까지 레이블마다 6,000개의 샘플

```
import numpy as np
print(np.unique(train_target, return_counts=True))
```

→ (array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8),
array([6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000]))

SECTION 7-1 인공 신경망(4)

○ 로지스틱 회귀로 패션 아이템 분류하기

- 이 훈련 샘플은 60,000개나 되기 때문에 전체 데이터를 한꺼번에 사용하여 모델을 훈련하는 것보다 샘플을 하나씩 꺼내서 모델을 훈련하는 방법이 더 효율적

- 확률적 경사 하강법
  - 확률적 경사 하강법은 여러 특성 중 기울기가 가장 가파른 방향을 따라 이동
  - 만약 특성마다 값의 범위가 많이 다르면 올바르게 손실 함수의 경사를 내려올 수 없음
  - 패션 MNIST의 경우 각 픽셀은 0~255 사이의 정숫값을 가지며, 이런 이미지의 경우 보통 255로 나누어 0~1 사이의 값으로 정규화
  - 이는 표준화는 아니지만 양수 값으로 이루어진 이미지를 전처리할 때 널리 사용하는 방법

SECTION 7-1 인공 신경망(5)

○ 로지스틱 회귀로 패션 아이템 분류하기

- reshape( ) 메서드를 사용해 2차원 배열인 각 샘플을 1차원 배열로 전개

```
train_scaled = train_input / 255.0
train_scaled = train_scaled.reshape(-1, 28*28)
```

- 변환된 train_scaled의 크기 확인

```
print(train_scaled.shape)
```

→ (60000, 784)

- SGDClassifier 클래스와 cross_validate 함수를 사용해 이 데이터에서 교차 검증으로 성능 확인

```
from sklearn.model_selection import cross_validate
from sklearn.linear_model import SGDClassifier

sc = SGDClassifier(loss='log_loss', max_iter=5, random_state=42)
scores = cross_validate(sc, train_scaled, train_target, n_jobs=-1)
print(np.mean(scores['test_score']))
```

→ 0.8194166666666666

- 위에서는 SGDClassifier의 반복 횟수(max_iter)를 5번으로 지정
  - 반복 횟수를 늘려도 성능이 크게 향상되지는 않음
  - 100이나 20 등의 여러 숫자를 넣어서 테스트한 결과

```
0.8311666666666667      # max_iter=10
0.8437333333333334      # max_iter=20
```

SECTION 7-1 인공 신경망(6)

○ 로지스틱 회귀로 패션 아이템 분류하기

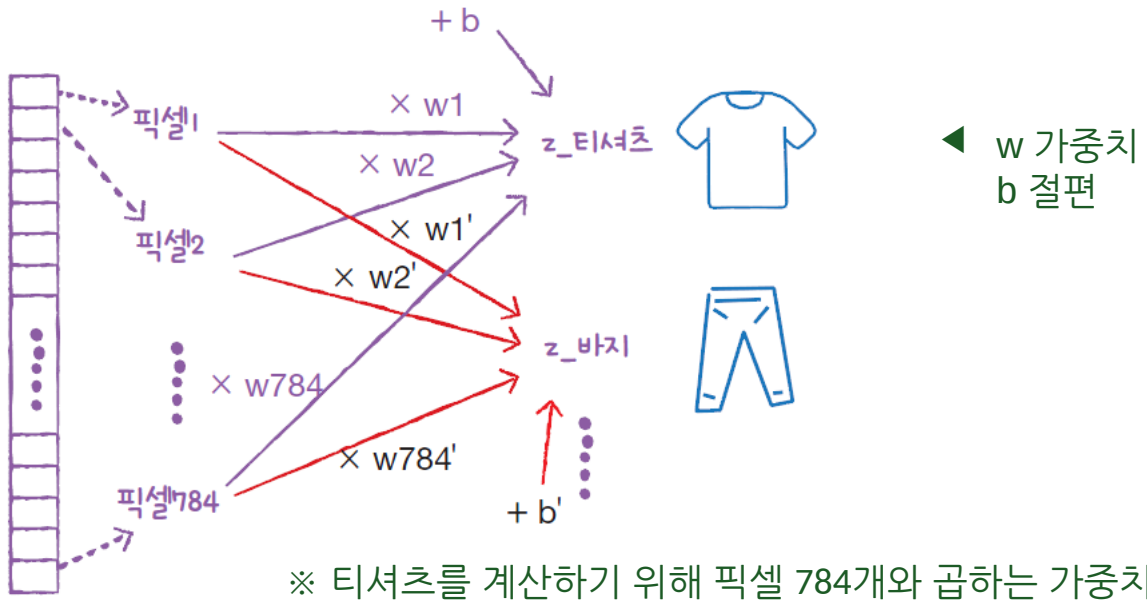
- **로지스틱 회귀 공식을 패션 MNIST 데이터에 맞게 변형**

$$z = a \times (Weight) + b \times (Length) + c \times (Diagonal) + d \times (Height) + e \times (Width) + f$$

$$z\_티셔츠 = w1 \times (픽셀1) + w2 \times (픽셀2) + \ldots + w784 \times (픽셀784) + b$$

$$z\_바지 = w1' \times (픽셀1) + w2' \times (픽셀2) + \ldots + w784' \times (픽셀784) + b'$$

- **SGDClassifier 모델은 패션 MNIST 데이터의 클래스를 가능한 잘 구분할 수 있도록 이 10개의 방정식에 대한 모델 파라미터 (가중치와 절편)를 탐색**



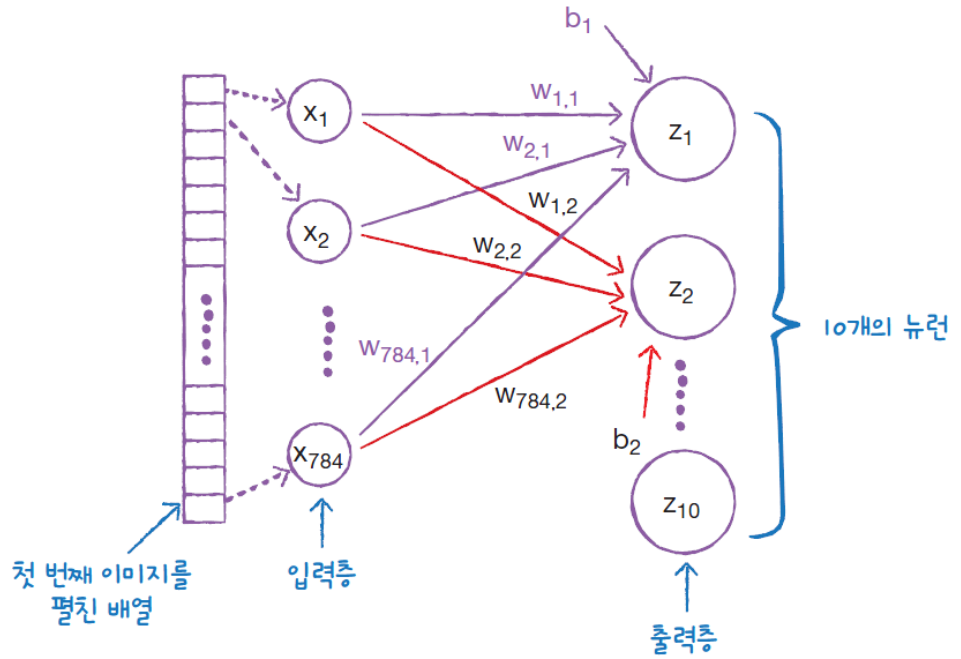
◀ w 가중치
b 절편

※ 티셔츠를 계산하기 위해 픽셀 784개와 곱하는 가중치 784개(w1~w784)와 절편(b)이
바지를 계산하기 위해 픽셀 784개와 곱하는 가중치 784개(w1'~w784'), 절편(b')과 다름

SECTION 7-1 인공 신경망(7)

○ 인공 신경망

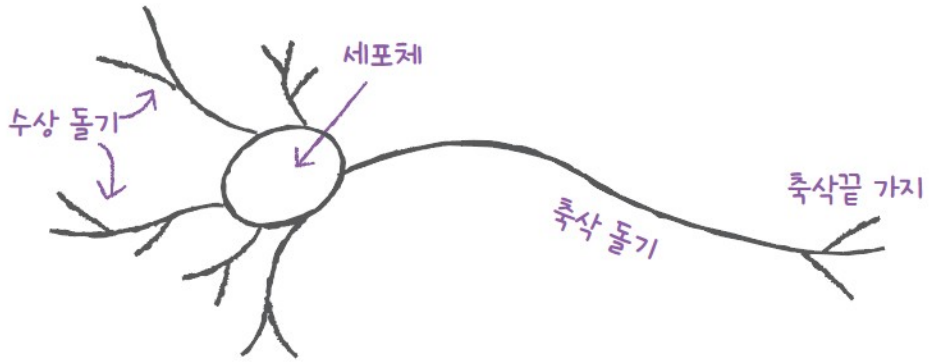
- 가장 기본적인 인공 신경망은 확률적 경사 하강법을 사용하는 로지스틱 회귀와 같음
- 출력층(output layer): 클래스가 총 10개이므로 z10까지 계산. z1~z10을 계산하고 이를 바탕으로 클래스를 예측하기 때문에 신경망의 최종 값을 만든다는 의미에서 출력층이라 부름
- 뉴런(neuron): 인공 신경망에서 z 값을 계산하는 단위. 뉴런이란 표현 대신에 유닛(unit)이라고 부르기도 함
- 입력층(input layer): x1~x784까지를 입력층이라고 함. 입력층은 픽셀값 자체이고 특별한 계산을 수행하지 않음



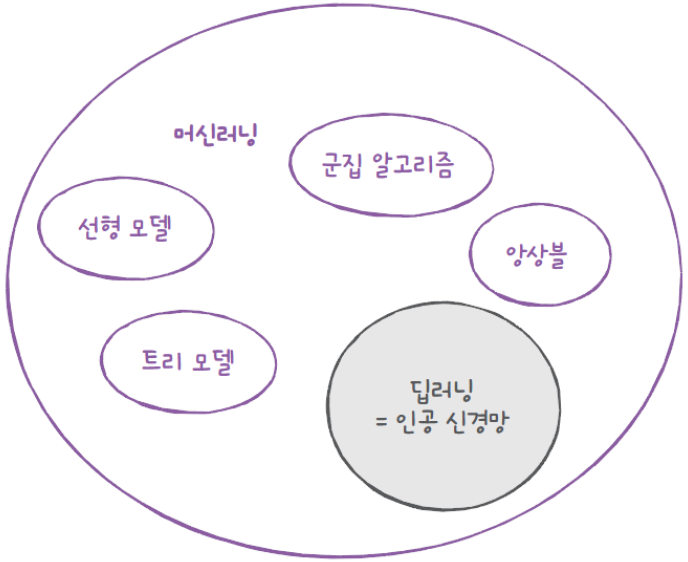
SECTION 7-1 인공 신경망(8)

○ 인공 신경망

- 매컬러-피츠 뉴런



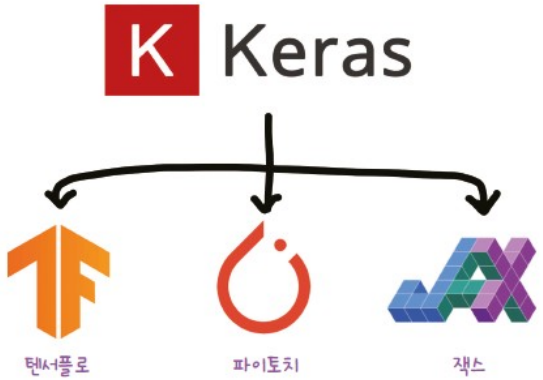
- 생물학적 뉴런과 인공 신경망은 같지 않음
- 인공 신경망은 기존의 머신러닝 알고리즘이 잘 해결하지 못했던 문제에서 높은 성능을 발휘하는 새로운 종류의 머신러닝 알고리즘



SECTION 7-1 인공 신경망(9)

○ 텐서플로(TensorFlow)와 케라스(Keras)

- 텐서플로: 구글이 2015년 11월 오픈소스로 공개한 딥러닝 라이브러리
- 케라스: 텐서플로의 고수준 API. 케라스는 2015년 3월 프랑소와 숄레가 만든 딥러닝 라이브러리
- 딥러닝 라이브러리는 그래픽 처리 장치인 GPU를 사용하여 인공 신경망을 훈련. GPU는 벡터와 행렬 연산에 매우 최적화되어 있기 때문에 곱셈과 덧셈이 많이 수행되는 인공 신경망에 큰 도움
- 멀티-백엔드 케라스
- 프랑소와가 구글에 합류한 뒤 텐서플로 라이브러리에 케라스 API가 내장. 텐서플로 2.0부터는 케라스 API를 남기고 나머지 고수준 API를 모두 정리했고, 케라스는 텐서플로의 핵심 API가 되었음
- 케라스 3.0부터 다시 멀티-백엔드 정책으로 바뀜
  - 케라스는 텐서플로, 파이토치, 잭스(JAX)를 백엔드로 사용



SECTION 7-1 인공 신경망(10)

○ 케라스 임포트

```
import keras
```

- 케라스가 사용중인 백엔드 확인
  - keras.config.backend( ) 함수

```
keras.config.backend()
```

→ 'tensorflow'

- 케라스의 백엔드 변경
  - 환경 변수 “KERAS_BACKEND”
  - 벡엔드를 바꾸려면 케라스 패키지를 임포트하기 전에 "KERAS_BACKEND" 환경 변수를 설정
    - 코랩의 경우에는 런타임을 종료하고 다시 시작

```
import os
os.environ["KERAS_BACKEND"] = "torch" # 또는 "jax"
```

SECTION 7-1 인공 신경망(11)

○ 인공 신경망으로 모델 만들기

- 로지스틱 회귀에서는 교차 검증을 사용해 모델을 평가
- 인공 신경망에서는 교차 검증을 잘 사용하지 않고 검증 세트를 별도로 덜어내어 사용
  - ① 딥러닝 분야의 데이터셋은 충분히 크기 때문에 검증 점수가 안정적
  - ② 교차 검증을 수행하기에는 훈련 시간이 너무 오래 걸림
- 패션 MNIST 데이터셋에서 검증 세트 나누기
  - 사이킷런의 train_test_split ( ) 함수

```
from sklearn.model_selection import train_test_split
train_scaled, val_scaled, train_target, val_target = train_test_split(
        train_scaled, train_target, test_size=0.2, random_state=42)
```

- 훈련 세트와 검증 세트의 크기 확인

```
print(train_scaled.shape, train_target.shape)
```

→ (48000, 784) (48000,)

```
print(val_scaled.shape, val_target.shape)
```

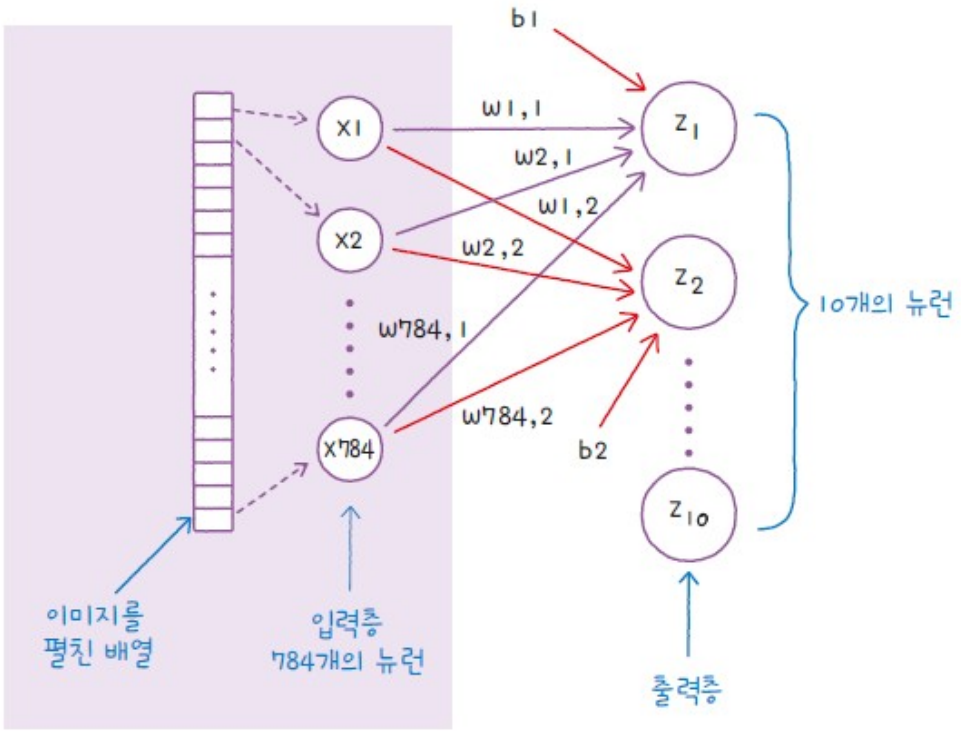
→ (12000, 784) (12000,)

SECTION 7-1 인공 신경망(12)

○ 인공 신경망 모델의 왼쪽 층 만들기

- 케라스에서 입력층을 정의 - Input( ) 함수
  - Input( ) 함수의 shape 매개변수에 입력의 크기를 지정

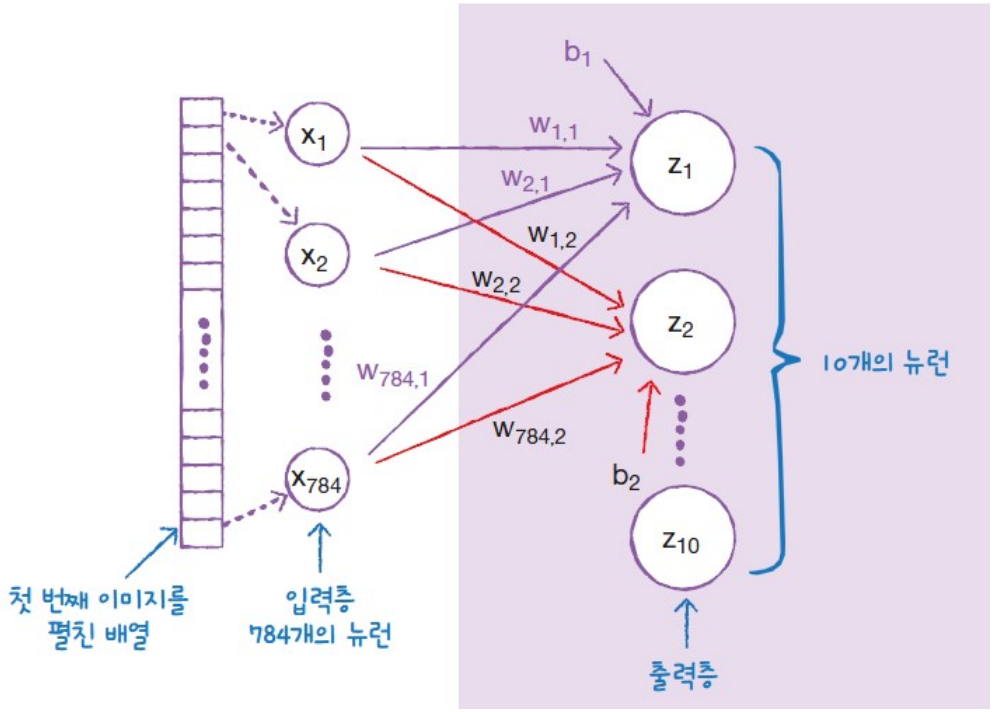
```
inputs = keras.layers.Input(shape=(784,))
```

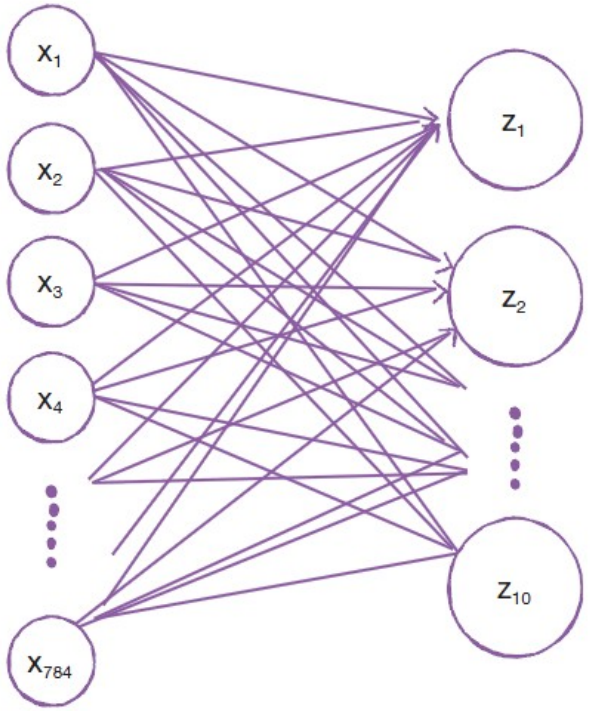


SECTION 7-1 인공 신경망(13)

○ 인공 신경망 모델의 오른쪽 층 만들기

- 10개의 패션 아이템을 분류하기 위해 10개의 뉴런으로 구성
- 밀집층(dense layer), 완전 연결층(fully connected layer)





SECTION 7-1 인공 신경망(14)

- 밀집층 만들기
  - 케라스의 Dense 클래스 사용
  - 매개변수 - 뉴런 개수, 뉴런의 출력에 적용할 함수

```
dense = keras.layers.Dense(10, activation='softmax')
```

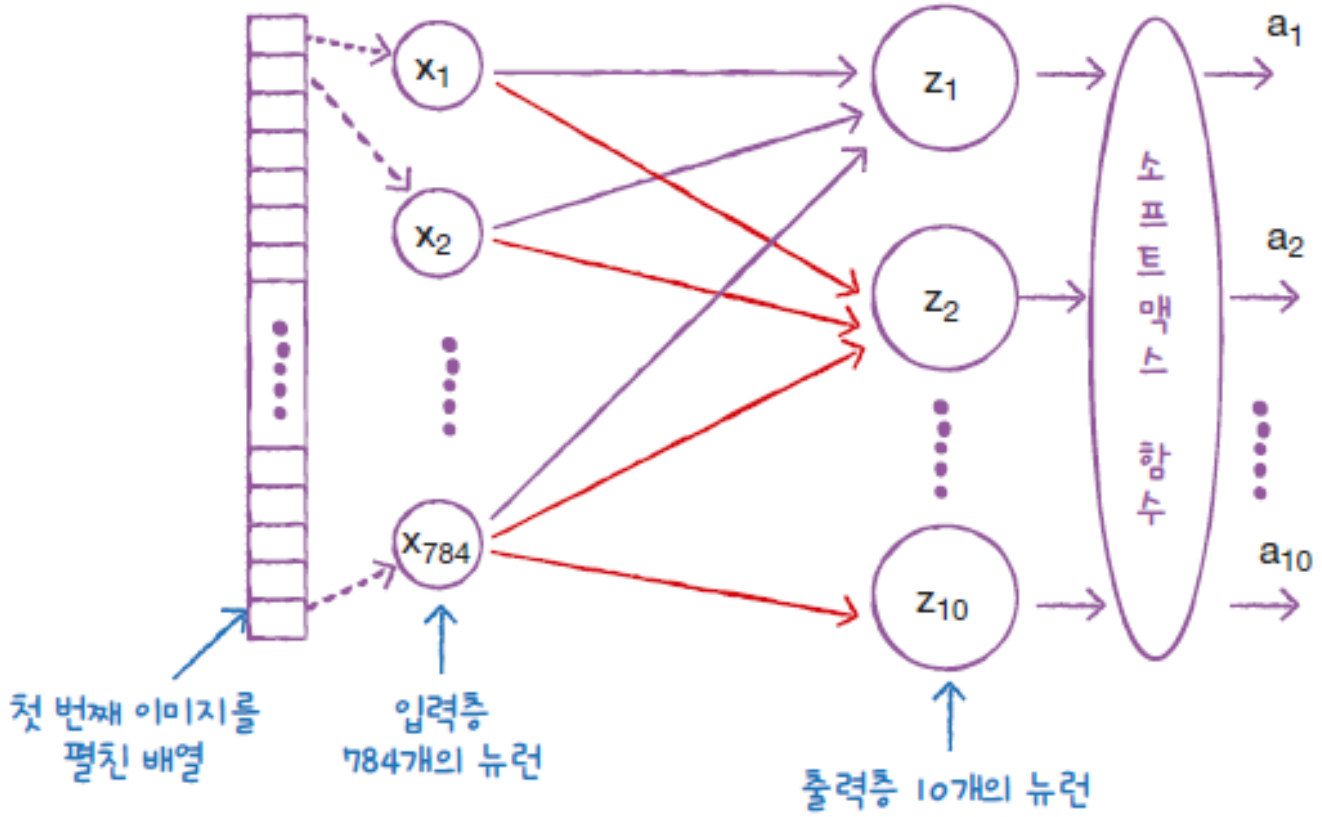
뉴런 개수 — 뉴런의 출력에 적용할 함수

- 입력층과 밀집층을 가진 신경망 모델 만들기
  - 케라스의 Sequential 클래스 사용
  - 앞에서 만든 입력층 객체 inputs와 밀집층의 객체 dense를 리스트로 묶어 전달

```
model = keras.Sequential([inputs, dense])
```

- 활성화 함수(activation function)
  - 소프트맥스와 같이 뉴런의 선형 방정식 계산 결과에 적용되는 함수

SECTION 7-1 인공 신경망(15)



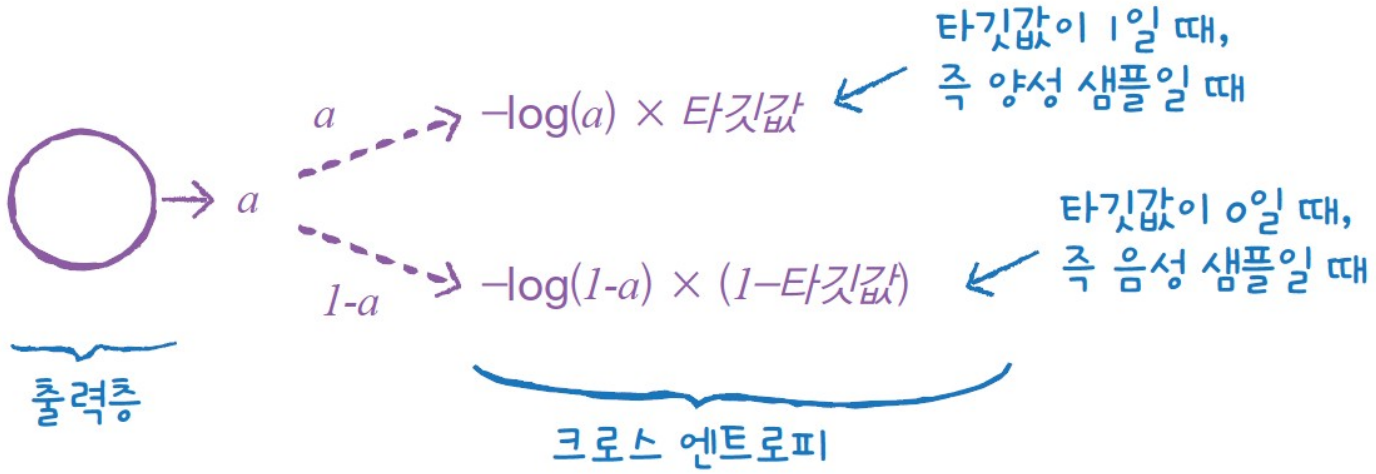
SECTION 7-1 인공 신경망(16)

○ 인공 신경망으로 패션 아이템 분류하기

- **손실 함수의 종류와 훈련 과정에서 계산하고 싶은 측정값 지정**

```
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

  - **이진 분류: loss = 'binary_crossentropy''**
  - **다중 분류: loss = ''categorical_crossentropy'**

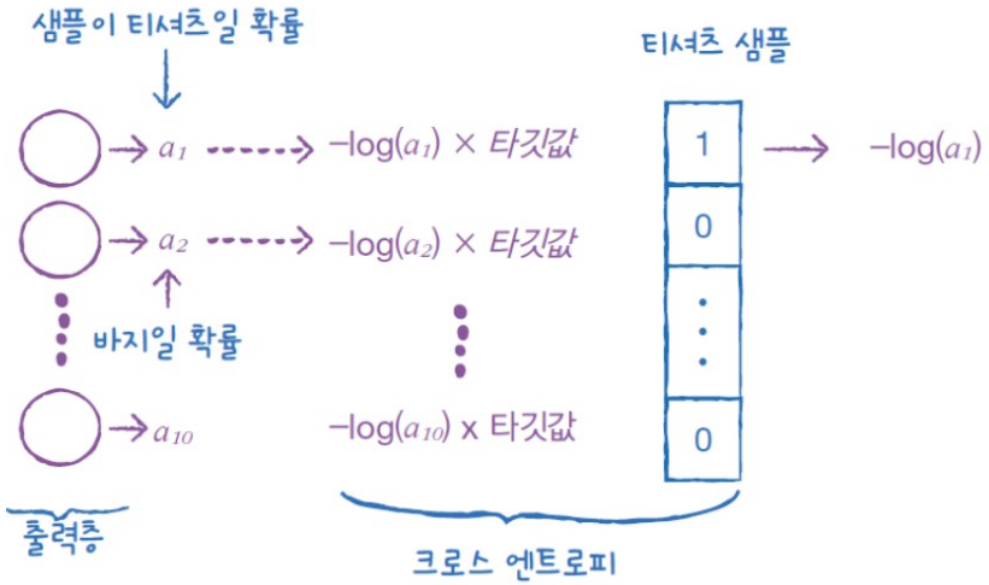


- 이진 분류에서는 출력층의 뉴런이 하나
- 이 뉴런이 출력하는 확률값 a(시그모이드 함수의 출력값)를 사용해 양성 클래스와 음성 클래스에 대한 크로스 엔트로피를 계산
- 이 계산은 4장의 로지스틱 손실 함수와 동일

SECTION 7-1 인공 신경망(17)

○ 인공 신경망으로 패션 아이템 분류하기

- 패션 MNIST 데이터셋과 같은 다중 분류의 손실 함수 계산
  - 출력층은 10개의 뉴런이 있고 10개의 클래스에 대한 확률을 출력
  - 첫 번째 뉴런은 티셔츠일 확률이고 두 번째 뉴런은 바지일 확률을 출력
  - 이진 분류와 달리 각 클래스에 대한 확률이 모두 출력되기 때문에 타깃에 해당하는 확률만 남겨 놓기 위해서 나머지 확률에는 모두0 을 곱함



샘플이 티셔츠일 경우

1) 첫 번째 뉴런의 활성화 함수 출력인 a1에 크로스 엔트로피 손실 함수를 적용하고 나머지 활성화 함수 출력 a2~a10까지는 모두 0 티셔츠 샘플의 타깃값은 첫 번째 원소만 1이고 나머지는 모두 0인 배열로 만들기

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

2) 배열과 출력층의 활성화 값의 배열과 곱하기

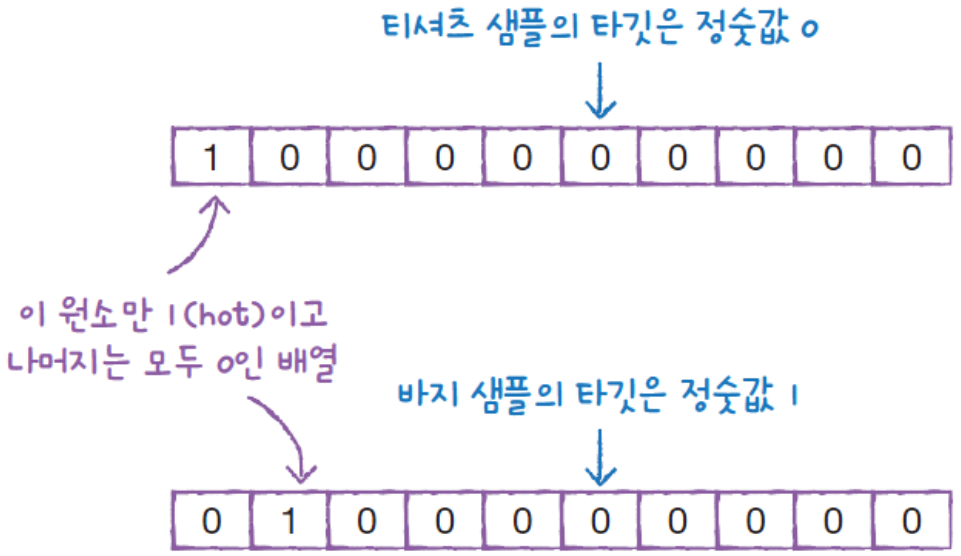
$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}] \times [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

3) 길이가 같은 넘파이 배열의 곱셈은 원소별 곱셈으로 수행 다른 원소는 모두 0이 되고 a1만 남음

SECTION 7-1 인공 신경망(18)

○ 인공 신경망으로 패션 아이템 분류하기

- 원-핫 인코딩(one-hot encoding): 타깃값을 해당 클래스만 1이고 나머지는 모두 0인 배열로 만드는 것
- 다중 분류에서 크로스 엔트로피 손실 함수를 사용하려면 0, 1, 2와 같이 정수로 된 타깃값을 원-핫 인코딩으로 변환해야 함



SECTION 7-1 인공 신경망(19)

○ 인공 신경망으로 패션 아이템 분류하기

- 패션 MNIST 데이터의 타깃값 출력

```
print(train_target[:10])
```

→ [7 3 5 8 6 9 3 3 9 9]

- 케라스에서는 정수로 된 타깃값을 원-핫 인코딩으로 바꾸지 않고 그냥 사용
- 'sparse_categorical_crossentropy': 정수로된 타깃값을 사용해 크로스 엔트로피 손실을 계산하는 것
- 타깃값을 원-핫 인코딩으로 준비했다면 compile( ) 메서드에
  손실 함수를 loss='categorical_crossentropy'로 지정
- compile( ) 메서드의 두 번째 매개변수 metrics
  - 케라스는 모델이 훈련할 때 기본으로 에포크마다 손실 값을 출력
  - 손실과 정확도를 함께 출력
    - metrics 매개변수에 정확도 지표 'accuracy'를 지정
  - metrics 매개변수에는 여러 개의 지표를 지정할 수 있으며, 하나의 지표를 지정할 때도 꼭 리스트로 전달

SECTION 7-1 인공 신경망(20)

○ 인공 신경망으로 패션 아이템 분류하기

- 모델 훈련
  - 처음 두 매개 변수에 입력(train_scaled)과 타깃(train_traget)을 지정
  - 반복할 에포크 횟수를 epochs 매개변수로 지정
  - 사이킷런의 로지스틱 모델과 동일하게 5번 반복

```
model.fit(train_scaled, train_target, epochs=5)
```

```
Epoch 1/5
1500/1500 ---------- 5s 2ms/step - accuracy: 0.7370 - loss: 0.7853
Epoch 2/5
1500/1500 ---------- 2s 2ms/step - accuracy: 0.8346 - loss: 0.4845
Epoch 3/5
1500/1500 ---------- 3s 2ms/step - accuracy: 0.8452 - loss: 0.4564
Epoch 4/5
1500/1500 ---------- 3s 2ms/step - accuracy: 0.8504 - loss: 0.4425
Epoch 5/5
1500/1500 ---------- 4s 2ms/step - accuracy: 0.8537 - loss: 0.4337
<keras.src.callbacks.history.History at 0x7b3eb6dacb80>
```

▲ 에포크마다 걸린 시간과 손실(loss), 정확도(accuracy)를 출력
- 5번 반복에 정확도가 85% 이상

SECTION 7-1 인공 신경망(21)

○ 인공 신경망으로 패션 아이템 분류하기

- **evaluate ( ) 메서드로 검증 세트(val_scaled, val_target)에서 모델의 성능 확인**

```
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8462 - loss: 0.4364
[0.44444453716278076, 0.8458333611488342]
```

▲ 검증 세트의 점수는 훈련 세트 점수보다 조금 낮은 것이 일반적
평가 결과는 훈련 세트의 점수보다 조금 낮은 84% 정도의 정확도

SECTION 7-1 인공 신경망(22)

○ 인공 신경망 모델로 성능 향상(문제해결 과정)

- 학습

  - 28 × 28 크기의 흑백 이미지로 저장된 패션 아이템 데이터셋인 패션 MNIST 데이터셋 사용
  - 로지스틱 손실 함수를 사용한 SGDClassifier 모델을 만들어 교차 검증 점수 확인
  - 텐서플로와 케라스 API
  - 케라스를 사용해 간단한 인공 신경망 모델을 만들어 패션 아이템을 분류
    - 이 간단한 인공 신경망은 경사 하강법을 사용한 로지스틱 회귀 모델과 거의 비슷
  - 로지스틱 손실 함수와 크로스 엔트로피 손실 함수 복습
  - 신경망에서 손실 함수 계산
  - 원-핫 인코딩

SECTION 7-1 인공 신경망(23)

○ 사이킷런 SGDClassifier와 케라스 Sequential 클래스 사용법 차이

사이킷런 모델

손실 함수, 반복 횟수

```
모델 → sc = SGDClassifier(loss='log', max_iter=5)
훈련 → sc.fit(train_scaled, train_target)
평가 → sc.score(val_scaled, val_target)
```

케라스 모델

층 생성

```
dense = keras.layers.Dense(10, activation='softmax', input_shape=(784,))
모델 → model = keras.Sequential(dense)
손실 함수
model.compile(loss='sparse_categorical_crossentropy', metrics='accuracy')
훈련 → model.fit(train_scaled, train_target, epochs=5)
평가 → model.evaluate(val_scaled, val_target)
반복 횟수
```

SECTION 7-1 마무리(1)

○ 키워드로 끝나는 핵심 포인트

- 인공 신경망은 생물학적 뉴런에서 영감을 받아 만든 머신러닝 알고리즘
  - 이름이 신경망이지만 실제 우리 뇌를 모델링한 것은 아님. 신경망은 기존의 머신러닝 알고리즘으로 다루기 어려웠던 이미지, 음성, 텍스트 분야에서 뛰어난 성능을 발휘하면서 크게 주목
  - 인공 신경망 알고리즘을 종종 딥러닝이라고도 함
- 케라스는 프랑소와 숄레가 만든 딥러닝 라이브러리로 매우 인기가 많음
  - 케라스는 사용하기 편한 고수준 API를 제공하고, 실제 연산은 텐서플로와 같은 백엔드가 담당
  - 케라스 3.0부터는 텐서플로, 파이토치, 잭스를 백엔드로 사용할 수 있음
  - 코랩에 설치된 케라스는 기본적으로 텐서플로를 백엔드로 사용
- 밀집층은 가장 간단한 인공 신경망의 층
  - 인공 신경망에는 여러 종류의 층이 있음
  - 밀집층에서는 뉴런들이 모두 연결되어 있기 때문에 완전 연결층이라고도 부름
  - 특별히 출력층에 밀집층을 사용할 때는 분류하려는 클래스와 동일한 개수의 뉴런을 사용
- 원-핫 인코딩은 정숫값을 배열에서 해당 정수 위치의 원소만 1이고 나머지는 모두 0으로 변환
  - 이런 변환이 필요한 이유는 다중 분류에서 출력층에서 만든 확률과 크로스 엔트로피 손실을 계산하기 위해서임
  - 케라스에서는 'sparse_categorical_entropy' 손실을 지정하면 이런 변환을 수행할 필요가 없음

SECTION 7-1 마무리(2)

○ 핵심 패키지와 함수

- Keras

- Input - 입력층을 구성하기 위한 함수
  - shape 매개변수에 입력의 크기를 튜플로 지정

- Dense - 신경망에서 가장 기본 층인 밀집층을 만드는 클래스
  - 첫 번째 매개변수에는 뉴런의 개수를 지정
  - activation 매개변수에는 사용할 활성화 함수를 지정
    - 대표적으로 'sigmoid', 'softmax' 함수 - 아무것도 지정하지 않으면 활성화 함수를 사용하지 않음
- Sequential - 케라스에서 신경망 모델을 만드는 클래스.
  - 이 클래스의 객체를 생성할 때 신경망 모델에 추가할 층을 파이썬 리스트로 전달

SECTION 7-1 마무리(3)

○ 핵심 패키지와 함수

- compile( ) - 모델 객체를 만든 후 훈련하기 전에 사용할 손실 함수와 측정 지표 등을 지정
  - loss 매개변수에 손실 함수를 지정
  - 이진 분류일 경우 'binary_crossentropy', 다중 분류일 경우 'categorical_crossentropy'를 지정
  - 클래스 레이블이 정수일 경우 'sparse_categorical_crossentropy'로 지정
  - 회귀 모델일 경우 'mean_square_error' 등으로 지정
  - metrics 매개변수에 훈련 과정에서 측정하고 싶은 지표를 리스트로 전달
- fit( ) - 모델을 훈련하는 메서드
  - 첫 번째와 두 번째 매개변수에 입력과 타깃 데이터를 전달
  - epochs 매개변수에 전체 데이터에 대해 반복할 에포크 횟수를 지정
- evaluate( ) - 모델 성능을 평가하는 메서드
  - 첫 번째와 두 번째 매개변수에 입력과 타깃 데이터를 전달
  - compile( ) 메서드에서 loss 매개변수에 지정한 손실 함수의 값과 metrics 매개변수에서 지정한 측정 지표를 출력

SECTION 7-1 확인 문제

1 어떤 인공 신경망의 입력 특성이 100개이고 밀집층에 있는 뉴런 개수가 10개일 때 필요한 모델 파라미터의 개수는 몇 개인가?

① 1,000개
② 1,001개
③ 1,010개
④ 1,100개

2 정수 레이블을 타깃으로 가지는 다중 분류 문제일 때 케라스 모델의 compile( ) 메서드에 지정할 손실 함수로 적절한 것은 무엇인가?

① 'sparse_categorical_crossentropy'
② 'categorical_crossentropy'
③ 'binary_crossentropy'
④ 'mean_square_error'

SECTION 7-2 심층 신경망(1)

○ 2개의 층

- **케라스 API를 사용해서 패션 MNIST 데이터셋 호출**

```
import keras
(train_input, train_target), (test_input, test_target) =\
    keras.datasets.fashion_mnist.load_data()
```

- **이미지의 픽셀값을 0~255 범위에서 0~1 사이로 변환하고, 28 × 28 크기의 2차원 배열을 784 크기의 1차원 배열로 펼침. 마지막으로 사이킷런의 train_test_split ( ) 함수로 훈련 세트와 검증 세트로 나누기**

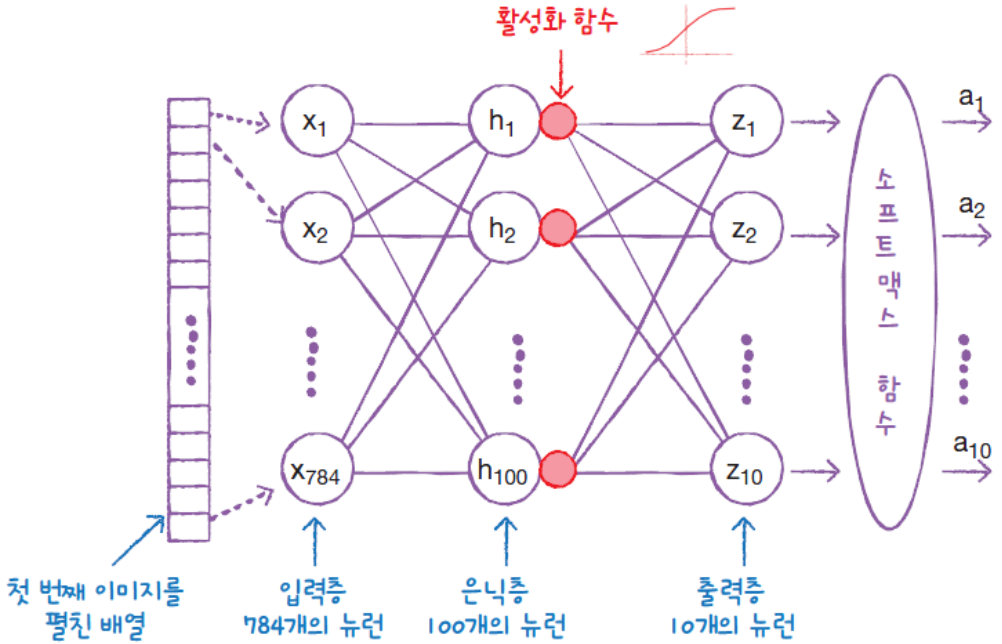
```
from sklearn.model_selection import train_test_split

train_scaled = train_input / 255.0
train_scaled = train_scaled.reshape(-1, 28*28)
train_scaled, val_scaled, train_target, val_target = train_test_split(
    train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 7-2 심층 신경망(2)

○ 2개의 층

- 인공 신경망 모델에 층을 2개 추가하기
- 은닉층(hidden layer): 입력층과 출력층 사이에 있는 모든 층
  - 은닉층의 활성화 함수: 신경망 층의 선형 방정식의 계산 값에 적용하는 함수
  - 이전 절에서 출력층에 적용했던 소프트맥스 함수도 활성화 함수. 출력층에 적용하는 활성화 함수는 종류가 제한
  - 이진 분류일 경우 시그모이드 함수를 사용하고 다중 분류일 경우 소프트맥스 함수를 사용
  - 이에 비해 은닉층의 활성화 함수는 비교적 자유로움 - 대표적으로 시그모이드 함수와 렐루 ReLU 함수 등을 사용



SECTION 7-2 심층 신경망(3)

○ 2개의 층

- 은닉층에 왜 활성화 함수를 적용할까?

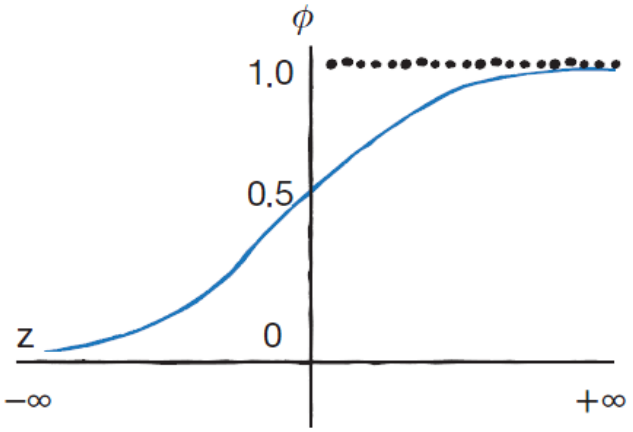
$$a \times 4 + 2 = b$$
$$b \times 3 - 5 = c$$
$$\longrightarrow \quad a \times 12 + 1 = c$$

$$a \times 4 + 2 = b$$
$$\log(b) = k$$
$$k \times 3 - 5 = c$$

- 많이 사용하는 활성화 함수 중 하나는 4장에서 배웠던 시그모이드 함수

$$\phi = \frac{1}{1+e^{-z}}$$

시그모이드 함수



시그모이드 그래프

SECTION 7-2 심층 신경망(4)

○ 2개의 층

- **케라스의 Dense 클래스로 시그모이드 활성화 함수를 사용한 은닉층과 소프트맥스 함수를 사용한 출력층만들기**
  - **입력층은 Input( ) 함수로 구성**
  - **신경망의 첫 번째 층은 input_shape 매개변수로 반드시 입력의 크기를 지정**

```
inputs = keras.layers.Input(shape=(784,))
dense1 = keras.layers.Dense(100, activation='sigmoid')
dense2 = keras.layers.Dense(10, activation='softmax')
```

- 은닉층의 뉴런 개수를 정하는데는 특별한 기준이 없으나, 적어도 출력층의 뉴런보다는 많게 만들어야 함
- 클래스 10 개에 대한 확률을 예측해야 하는데 이전 은닉층의 뉴런이 10개보다 적다면 부족한 정보가 전달
- dense2는 출력층. 10개의 클래스를 분류하므로 10개의 뉴런을 두었고 활성화 함수는 소프트맥스 함수로 지정

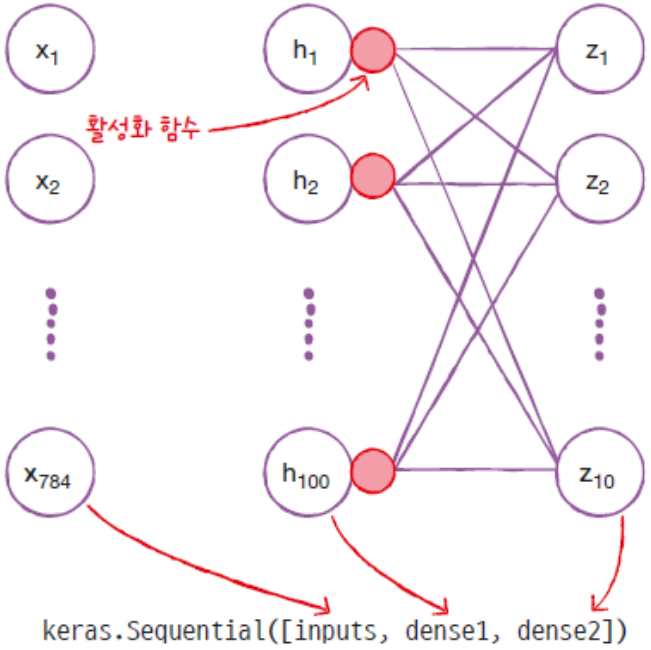
SECTION 7-2 심층 신경망(5)

○ 심층 신경망(deep neural network, DNN) 만들기

- inputs와 dense1, dense2 객체를 Sequential 클래스에 추가하여 심층 신경망 만들기

```
model = keras.Sequential([inputs, dense1, dense2])
```

- Sequential 클래스의 객체를 만들 때 여러 개의 층을 추가하려면 이와 같이 리스트에 계속 필요한 층을 추가
  - 주의 - 입력층을 맨 앞에 두고 출력층을 가장 마지막에 두어야 함
  - 이 리스트는 가장 처음 등장하는 은닉층에서 마지막 출력층의 순서로 나열



SECTION 7-2 심층 신경망(6)

○ 심층 신경망(deep neural network, DNN) 만들기

- **summary( ) 메서드: 층에 대한 정보**

```
model.summary()
```

→

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	78,500
dense_1 (Dense)	(None, 10)	1,010

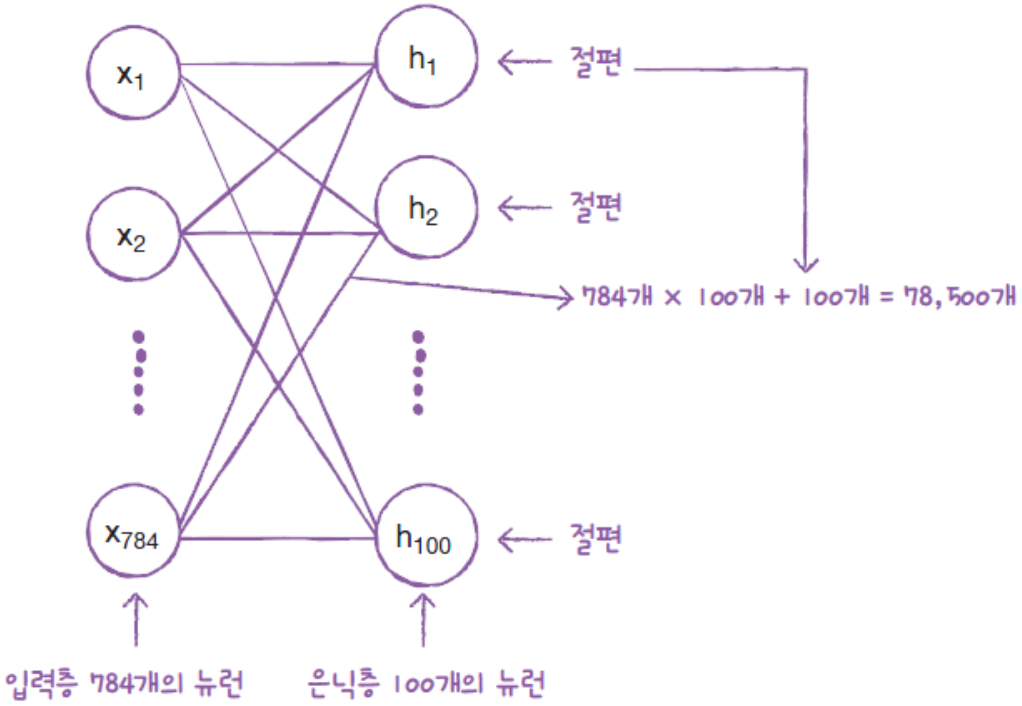
```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

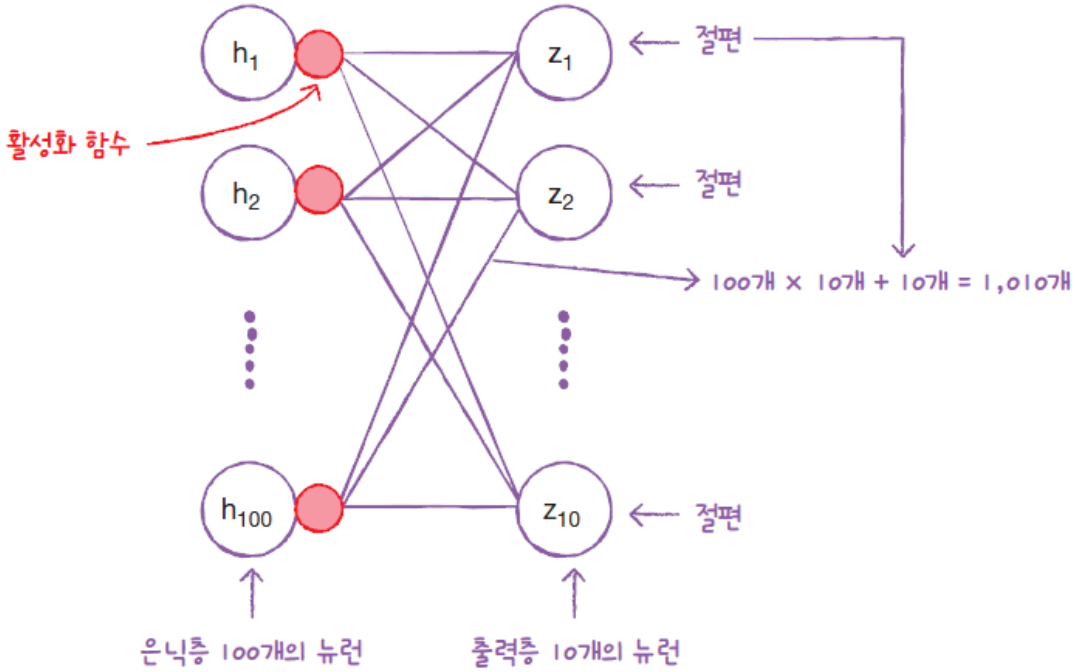
SECTION 7-2 심층 신경망(7)

○ 심층 신경망(deep neural network, DNN) 만들기

- summary( ) 메서드: 층에 대한 정보
  - 맨 첫 줄 - 모델의 이름
  - 그다음 입력층을 제외하고 이 모델에 들어 있는 층이 순서대로 나열
  - 층마다 층 이름, 클래스, 출력 크기, 모델 파라미터 개수가 출력
    - 층을 만들 때 name 매개변수로 이름을 지정 - 층 이름을 지정하지 않으면 케라스가 자동으로 'dense'로 지정됨
  - 출력 크기 (None, 100)
    - 첫 번째 차원은 샘플의 개수 - 샘플 개수가 아직 정의되어 있지 않기 때문에 None
    - 두 번째 차원 100 - 은닉층의 뉴런 개수를 100개로 두었으니 100개의 출력
  - 모델 파라미터 개수
    - 이 층은 Dense 층이므로 입력 픽셀 784개와 100개의 모든 조합에 대한 가중치가 있음
    - 뉴런마다 1개의 절편

SECTION 7-2 심층 신경망(8)





SECTION 7-2 심층 신경망(9)

○ 층을 추가하는 다른 방법

- **모델을 훈련하기 전에 Seqeuntial 클래스에 층을 추가하는 다른 방법**
  - **Sequential 클래스의 생성자 안에서 바로 Dense 클래스의 객체 만들기**

```
model = keras.Sequential([
    keras.layers.Input(shape=(784,)),
    keras.layers.Dense(100, activation='sigmoid', name='은닉층'),
    keras.layers.Dense(10, activation='softmax', name='출력층')
    ], name='패션 MNIST 모델')
```

- Sequential 클래스의 name 매개변수로 모델의 이름을 지정
- Dense 층의 name 매개변수에 층의 이름을 '은닉층'과 '출력층'으로 각각 지정

SECTION 7-2 심층 신경망(10)

○ 층을 추가하는 다른 방법

- **summary( ) 메서드의 출력에 이름이 잘 반영되는지 확인**

```
model.summary()
```

→

```
Model: "패션 MNIST 모델"
```

Layer (type)	Output Shape	Param #
은닉층 (Dense)	(None, 100)	78,500
출력층 (Dense)	(None, 10)	1,010

```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

- 2개의 Dense 층이 이전과 동일하게 추가되었고 파라미터 개수도 같음 - 바뀐 것은 모델 이름과 층 이름
- 여러 모델과 많은 층을 사용할 때 name 매개변수를 사용하면 구분하기 쉬움
- 이 방법이 편리하지만 아주 많은 층을 추가하려면 Sequential 클래스 생성자가 매우 길어짐

SECTION 7-2 심층 신경망(11)

○ 층을 추가하는 다른 방법

- **다른 방법 2 - Sequential 클래스의 객체를 만들고 이 객체의 add( ) 메서드를 호출하여 층을 추가**

```
model = keras.Sequential()
model.add(keras.layers.Input(shape=(784,)))
model.add(keras.layers.Dense(100, activation='sigmoid'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

- **summary( ) 메서드의 결과에서 층과 파라미터 개수는 당연히 동일**

```
model.summary()
```

→

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 100)	78,500
dense_3 (Dense)	(None, 10)	1,010

```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

SECTION 7-2 심층 신경망(12)

○ 모델 훈련

- 5번의 에포크 동안 훈련

```
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(train_scaled, train_target, epochs=5)
```

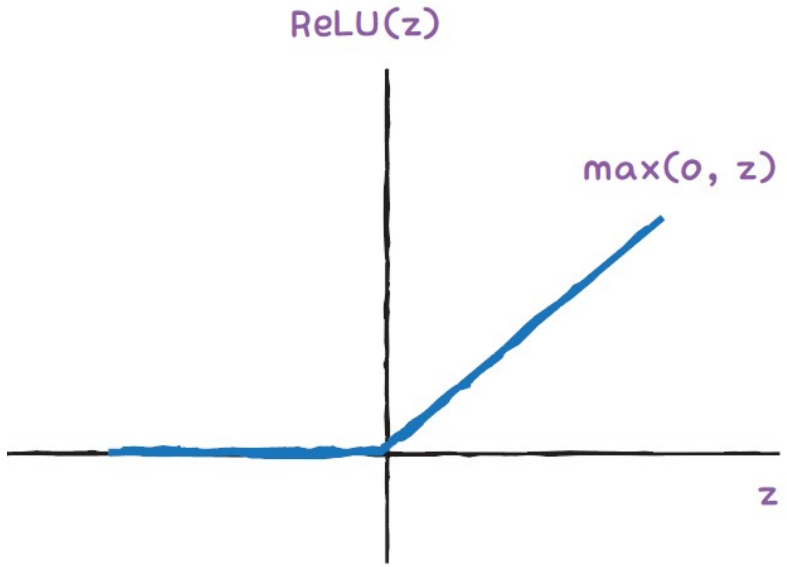
→

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7525 - loss: 0.7720
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - accuracy: 0.8463 - loss: 0.4270
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8604 - loss: 0.3857
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8696 - loss: 0.3600
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8759 - loss: 0.3410
<keras.src.callbacks.history.History at 0x795663038ee0>
```

SECTION 7-2 심층 신경망(13)

○ 렐루(ReLU) 함수

- 초창기 인공 신경망의 은닉층에 많이 사용된 활성화 함수는 시그모이드 함수
  - 함수의 오른쪽과 왼쪽 끝으로 갈수록 그래프가 누워있기 때문에 올바른출력을 만드는데 신속하게 대응하지 못하는 단점
- 특히 층이 많은 심층 신경망일수록 그 효과가 누적되어 학습을 더 어렵게 함
- 이를 개선하기 위해 다른 종류의 활성화 함수가 바로 렐루 함수
  -렐루 함수는 입력이 양수일 경우 마치 활성화 함수가 없는 것처럼 그냥 입력을 통과시키고 음수일 경우에는 0으로 만듦
- 렐루 함수는 z가 0보다 크면 z를 출력하고 z가 0보다 작으면 0을 출력



SECTION 7-2 심층 신경망(14)

○ 렐루(ReLU) 함수

- Flatten 클래스는 배치 차원을 제외하고 나머지 입력 차원을 모두 일렬로 펼치는 역할
  -입력에 곱해지는 가중치나 절편이 없음
  -Flatten 클래스는 학습하는 층이 아님
- Flatten 클래스를 입력층과 은닉층 사이에 추가

```
model = keras.Sequential()
model.add(keras.layers.Input(shape=(28,28)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

  - Input( ) 함수의 shape 매개변수에 입력의 크기를 (784,)가 아니라 원본 이미지 크기인 (28,28)로 지정
  - 다음에 등장하는 Flatten 층이 1차원으로 펼쳐 줄 것이므로 입력층에는 2차원 입력을 받겠다는 의미
  - 첫 번째 Dense 층의 활성화 함수를 'relu'로 바꾼 것에 주의
    - 이 신경망을 깊이가 3인 신경망이라고 부르지는 않음
      - Flatten 클래스는 학습하는 층이 아님

SECTION 7-2 심층 신경망(15)

○ 렐루(ReLU) 함수

- **Flatten 클래스를 추가한 모델의 summary( ) 메서드 호출**

```
model.summary()
```

→

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 100)	78,500
dense_5 (Dense)	(None, 10)	1,010

```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

SECTION 7-2 심층 신경망(16)

○ 렐루(ReLU) 함수

- 훈련 데이터를 다시 준비해서 모델을 훈련 - reshape( ) 메서드를 적용 않음

```
(train_input, train_target), (test_input, test_target) =\
    keras.datasets.fashion_mnist.load_data()
train_scaled = train_input / 255.0
train_scaled, val_scaled, train_target, val_target = train_test_split(
    train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 7-2 심층 신경망(17)

○ 렐루(ReLU) 함수

- **모델을 컴파일하고 훈련**

```
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(train_scaled, train_target, epochs=5)
```

→

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7637 - loss: 0.6723
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8515 - loss: 0.4054
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8676 - loss: 0.3595
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8786 - loss: 0.3344
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8858 - loss: 0.3177
<keras.src.callbacks.history.History at 0x79565c031ae0>
```

◀ 시그모이드 함수를 사용했을 때와 비교하면 성능이 조금 향상

SECTION 7-2 심층 신경망(18)

○ 렐루(ReLU) 함수

- **검증 세트 성능 확인**

```
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8671 - loss: 0.3837
[0.3847014605998993, 0.8665000200271606]
```

▲ 은닉층을 추가하지 않은 경우보다 몇 퍼센트 성능이 향상

SECTION 7-2 심층 신경망(19)

○ 옵티마이저(optimizer)

- 옵티마이저는 케라스에서 제공하는 다양한 종류의 경사 하강법 알고리즘
- 하이퍼파라미터는 모델이 학습하지 않아 사람이 지정해 주어야 하는 파라미터
  - 추가할 은닉층의 개수, 뉴런 개수, 활성화 함수, 층의 종류, 배치 사이즈 매개변수, 에포크 매개변수 등
- RMSprop의 학습률도 조정할 하이퍼파라미터 중 하나
- 가장 기본적인 옵티마이저는 확률적 경사 하강법인 SGD
  - compile( ) 메서드의 optimizer 매개변수를 'sgd'로 지정

```
model.compile( optimizer='sgd', loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])
```

- 옵티마이저는 keras.optimizers 패키지 아래 SGD 클래스로 구현(아래 코드와 위의 코드는 동일함)

```
sgd = keras.optimizers.SGD()
model.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

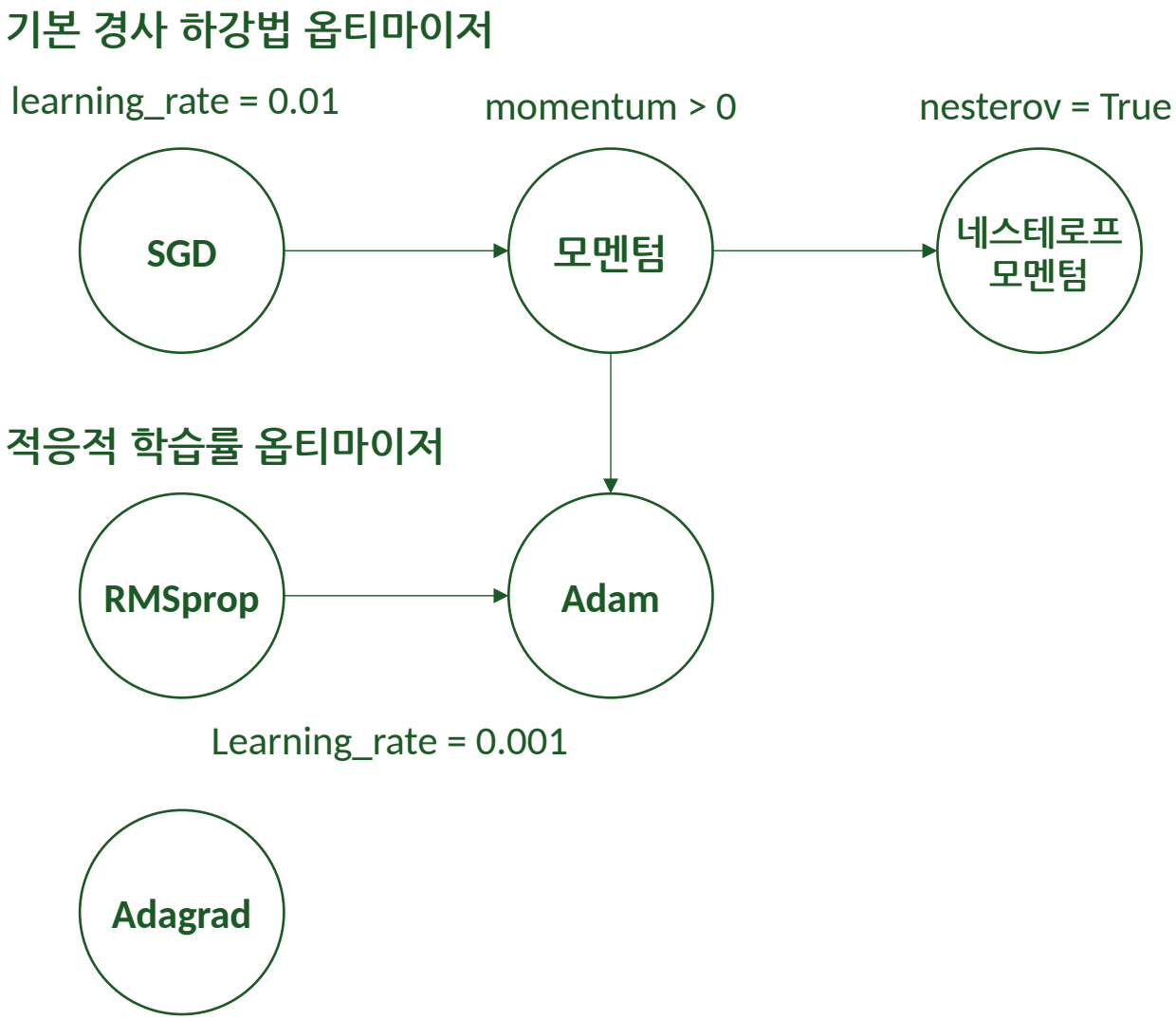
- SGD 클래스의 학습률 기본값이 0.01일 때 이를 변경하려면 원하는 학습률을 learning_rate 매개변수에 지정하여 사용

```
sgd = keras.optimizers.SGD(learning_rate=0.1)
```

SECTION 7-2 심층 신경망(20)

○ 옵티마이저(optimizer)

- 다양한 옵티마이저



SECTION 7-2 심층 신경망(21)

○ 옵티마이저(optimizer)

- 다양한 옵티마이저

- SGD 클래스의 momentum 매개변수의 기본값은 0. 이를 0보다 큰 값으로 지정하면 마치 이전의 그레이디언트를 가속도처럼 사용하는 모멘텀 최적화(momentum optimization)를 사용
  - 보통 momentum 매개변수는 0.9 이상을 지정
- SGD 클래스의 nesterov 매개변수를 기본값 False에서 True로 바꾸면 네스테로프 모멘텀 최적화(또는 네스테로프 가속 경사)를 사용

```
sgd = keras.optimizers.SGD(momentum=0.9, nesterov=True)
```

- 네스테로프 모멘텀은 모멘텀 최적화를 2번 반복하여 구현
  - 대부분의 경우 네스테로프 모멘텀 최적화가 기본 확률적 경사 하강법보다 더 나은 성능을 제공
- 적응적 학습률(adaptive learning rate): 모델이 최적점에 가까이 갈수록 학습률을 낮출 수 있음
  - 대표적인 옵티마이저는 Adagrad와 RMSprop
- Adam: 모멘텀 최적화와 RMSprop의 장점을 접목

SECTION 7-2 심층 신경망(22)

○ 옵티마이저(optimizer)

- **Adam 클래스의 매개변수 기본값을 사용해 패션 MNIST 모델 훈련을 위해 모델을 다시 생성**

```
model = keras.Sequential()
model.add(keras.layers.Input(shape=(28,28)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

- **compile( ) 메서드의 optimizer를 'adam'으로 설정하고 5번의 에포크 동안 훈련**

```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.fit(train_scaled, train_target, epochs=5)
```

→

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.7691 - loss: 0.6706
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8515 - loss: 0.4134
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8691 - loss: 0.3618
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - accuracy: 0.8793 - loss: 0.3302
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8873 - loss: 0.3088
<keras.src.callbacks.history.History at 0x7956692ffdc0>
```

SECTION 7-2 심층 신경망(23)

○ 옵티마이저(optimizer)

- **검증 세트에서의 성능 확인**

```
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8762 - loss: 0.3506
[0.35239022970199585, 0.8725833296775818]
```

SECTION 7-2 심층 신경망(24)

○ 케라스 API를 활용한 심층 신경망(문제해결 과정)

- 학습
  - 여러 개의 층을 추가하여 다층 인공 신경망을 만들기 – 특별히 이런 인공 신경망을 심층 신경망이라고 함
  - 케라스 API를 사용하여 층을 추가하는 여러 가지 방법
  - 케라스 모델의 정보를 요약해 주는 summary( ) 메서드 사용출
  - 력값의 의미를 이해하고 모델 파라미터 개수를 계산
  - 모델 파라미터 개수를 계산하는 과정은 모델을 올바르게 이해하고 있는지 확인하는 좋은 방법 중 하나
  - 은닉층에 적용한 시그모이드 활성화 함수 대신에 새로운 렐루 활성화 함수와 이를 적용해 약간의 성능을 향상
  - 다양한 고급 경사 하강법 옵티마이저들을 적용하는 방법 - 케라스 API를 사용하면 이런 작업이 어렵지 않고 직관적으로 구성할 수 있음

SECTION 7-2 마무리(1)

○ 키워드로 끝나는 핵심 포인트

- 심층 신경망은 2개 이상의 층을 포함한 신경망
  - 종종 다층 인공 신경망, 심층 신경망, 딥 러닝을 같은 의미로 사용
- 렐루 함수는 이미지 분류 모델의 은닉층에 많이 사용하는 활성화 함수
  - 시그모이드 함수는 층이 많을수록 활성화 함수의 양쪽 끝에서 변화가 작기 때문에 학습이 어려워지나, 렐루 함수는 이런 문제가 없으며 계산도 간단
- 옵티마이저는 신경망의 가중치와 절편을 학습하기 위한 알고리즘 또는 방법
  - 케라스에는 다양한 경사 하강법 알고리즘이 구현
  - 대표적으로 SGD, 네스테로프 모멘텀, RMSprop, Adam 등

SECTION 7-2 마무리(2)

○ 핵심 패키지와 함수

- Keras

- add( ): 케라스 모델에 층을 추가하는 메서드
- summary( ): 케라스 모델의 정보를 출력하는 메서드
- SGD: 기본 경사 하강법 옵티마이저 클래스
- Adagrad: Adagrad 옵티마이저 클래스
- RMSprop: RMSprop 옵티마이저 클래스
- Adam: Adam 옵티마이저 클래스

SECTION 7-2 확인 문제(1)

1 다음 중 모델의 add( ) 메서드 사용법이 올바른 것은 어떤 것인가?

① model.add(keras.layers.Dense)

② model.add(keras.layers.Dense(10, activation='relu'))

③ model.add(keras.layers.Dense, 10, activation='relu')

④ model.add(keras.layers.Dense) (10, activation='relu')

2 크기가 300 × 300인 입력을 케라스 층으로 펼치려고 할 때 다음 중 어떤 층을 사용해야 하나?

① Plate

② Flatten

③ Normalize

④ Dense

SECTION 7-2 확인 문제(2)

3. 다음 중에서 이미지 분류를 위한 심층 신경망에 널리 사용되는 케라스의 활성화 함수는?

① linear
② sigmoid
③ relu
④ tanh

4. 다음 중 적응적 학습률을 사용하지 않는 옵티마이저는?

① SGD
② Adagrad
③ RMSprop
④ Adam

SECTION 7-2 파이토치 버전 살펴보기(1)

○ 파이토치로 신경망 모델 만들기

- 파이토치 역시 코랩에 기본적으로 설치
- torchvision을 사용해서 패션 MNIST 데이터셋 가져오기

```
from torchvision.datasets import FashionMNIST

fm_train = FashionMNIST(root='.', train=True, download=True)
fm_test = FashionMNIST(root='.', train=False, download=True)
```

- FashionMNIST 클래스를 호출하여 훈련 데이터와 테스트 데이터를 위한 fm_train과 fm_test 객체를 만듦
- 세 개의 주요 매개변수
  - root 매개변수 - 다운로드된 데이터를 저장될 위치를 지정
  - train 매개변수 - 훈련 데이터를 다운로드할지, 테스트 데이터를 다운로드할지를 결정
  - download 매개변수를 True로 지정하면 원격에 저장된 데이터를 다운로드하여 로컬에 저장
- 실제 데이터는 fm_train과 fm_test 객체의 data 속성에 파이토치 텐서(PyTorch Tensor)로 저장됨

SECTION 7-2 파이토치 버전 살펴보기(2)

- 파이썬의 type( ) 함수로 data 속성 확인

```
type(fm_train.data)
```

→ torch.Tensor

- 텐서(Tensor) - 파이토치의 기본 데이터 구조
  - 파이 배열과 비슷한 인터페이스

    예) 훈련 데이터와 테스트 데이터의 크기는 텐서 객체의 shape 속성을 사용하여 확인

```
print(fm_train.data.shape, fm_test.data.shape)
```

→

torch.Size([60000, 28, 28]) torch.Size([10000, 28, 28])

- 타깃 데이터는 fm_train와 fm_test 객체의 targets 속성에 저장됨
  - 타깃의 크기 확인

```
print(fm_train.targets.shape, fm_test.targets.shape)
```

→

torch.Size([60000]) torch.Size([10000])

SECTION 7-2 파이토치 버전 살펴보기(3)

- fm_train 객체의 속성을 사용해 모델 훈련에 사용할 입력과 타깃 데이터를 준비

```
train_input = fm_train.data
train_target = fm_train.targets
```

- 파이토치 텐서는 넘파이 배열처럼 브로드캐스팅을 지원하기 때문에 간편하게 입력을 정규화 가능

```
train_scaled = train_input / 255.0
```

- 사이킷런의 train_test_split ( ) 함수를 사용해 훈련 세트를 다시 훈련 세트와 검증 세트로 나눌 수 있음

```
from sklearn.model_selection import train_test_split

train_scaled, val_scaled, train_target, val_target = train_test_split(
                train_scaled, train_target, test_size=0.2, random_state=42)
```

- 훈련 세트와 검증 세트의 크기 확인

```
print(train_scaled.shape, val_scaled.shape)
```

→ torch.Size([48000, 28, 28]) torch.Size([12000, 28, 28])

SECTION 7-2 파이토치 버전 살펴보기(4)

○ 모델 만들기

- 파이토치의 층은 torch.nn 모듈 아래에 위치
  - Sequential 클래스를 호출할 때 필요한 층을 차례로 나열
    - 여기에서는 본문과 동일하게 100개의 뉴런을 가진 은닉층과 10개의 뉴런을 가진 출력층을 구성
  - 시작할 때 Flatten 층을 추가하고 두 개의 밀집층 사이에 렐루 함수를 삽입

```
import torch.nn as nn

model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

  - 그다음 Flatten, Linear, ReLU, Linear 클래스를 사용하여 각각의 층을 생성한 후, 이 객체들을 Sequential 클래스에 매개변수로 전달하여 모델을 구성
  - Linear 층에서 입력 크기를 지정하는 방법
    - Flatten 층이 입력을 1차원으로 펼치기 때문에 첫 번째 Linear 층은 784개의 입력을 받음
    - 두 번째 Linear 층은 첫 번째 Linear 층의 출력 크기인 100개의 입력을 받도록 지정

SECTION 7-2 파이토치 버전 살펴보기(5)

○ 케라스 모델과 파이토치 모델의 주요 차이점

1. 파이토치에서는 모델의 입력 크기를 사전에 지정할 필요가 없음
   - 따라서 케라스의 Input( )과 같은 별도의 입력 정의 함수가 없음
2. 케라스의 Dense 층과 동일한 역할을 하는 것이 파이토치의 Linear 층
   - Linear 층을 사용할 때는 입력 크기와 출력 크기(뉴런 개수)를 매개변수로 전달
3. 파이토치에서는 활성화 함수를 별도의 층으로 추가해야 합니다. 렐루 함수의 경우 ReLU 층을 사용
4. 출력층에 해당하는 두 번째 Linear 층 다음에는 활성화 함수가 없음
   - 케라스에서는 다중 분류 문제를 해결하기 위해 마지막 층에 소프트맥스 함수를 포함했지만,
   파이토치에서는 이를 생략

SECTION 7-2 파이토치 버전 살펴보기(6)

- 파이토치는 케라스의 summary( ) 메서드와 같이 전체 구조를 확인하는 도구를 제공하지 않음
  - 대신 torchinfo 패키지를 설치하면 비슷한 결과를 획득
  - 이 패키지는 코랩에 설치되어 있지 않으므로 다음 명령을 실행하여 먼저 torchinfo를 설치

```
!pip install torchinfo
```

- torchinfo에서 summary ( ) 함수를 임포트하고 모델과 함께 호출
  - 선택적으로 input_size 매개변수에 입력 크기를 지정할 수 있음
  - 입력 크기를 지정하면 입력 데이터가 각 층을 통과할 때 크기가 어떻게 변하는지 확인할 수 있어 매우 유용
  - 여기서는 배치 크기를 32로 가정하여 입력 크기를 (32, 28, 28)로 설정
    - 한 번에 32개의 샘플이 모델에 입력되는 형태

```
from torchinfo import summary
summary(model, input_size=(32, 28, 28))
```

→

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
Sequential                               [32, 10]                  --
├─Flatten: 1-1                           [32, 784]                 --
├─Linear: 1-2                            [32, 100]                 78,500
├─ReLU: 1-3                              [32, 100]                 --
├─Linear: 1-4                            [32, 10]                  1,010
==========================================================================
Total params: 79,510
```

SECTION 7-2 파이토치 버전 살펴보기(7)

○ 훈련 수행

- 훈련을 시작하기 전에 코랩에서 GPU 런타임을 사용하고 있다면 앞서 만든 모델을 GPU에 적재
  - 케라스의 경우 GPU가 감지되면 자동으로 모델을 GPU에서 훈련하지만, 파이토치는 명시적으로 GPU로 모델을 이동
- 일반적으로 다음과 같이 파이토치 디바이스 객체와 to( ) 메서드를 사용해 GPU 사용 여부를 설정

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

SECTION 7-2 파이토치 버전 살펴보기(8)

- 손실 함수와 옵티마이저 준비
  - 케라스에서는 다중 분류를 위한 크로스 엔트로피 손실 함수로 CrossEntropyLoss 클래스를 제공
  - Adam을 비롯한 다양한 옵티마이저 클래스는 torch.optim 패키지 아래에 포함
- 손실 함수와 옵티마이저 클래스에 대한 객체 만들기

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

  - 앞서 모델의 마지막 층 다음에 소프트맥스 활성화 함수를 추가하지 않은 이유는 CrossEntropyLoss 클래스에 소프트맥스 함수가 이미 포함되어 있기 때문임
  - 파이토치의 CrossEntropyLoss 클래스는 소프트맥스 함수 계산과 크로스 엔트로피 계산을 하나의 연산으로 합쳐 효율적으로 계산할 수 있도록 설계
  - 따라서 다중 분류 문제를 다룰 때, 파이토치 모델의 마지막에 소프트맥스 함수를 추가할 필요가 없음

SECTION 7-2 파이토치 버전 살펴보기(9)

✚ 여기서 잠깐 **Adam 클래스에 입력으로 전달하는 model.parameters()는?**

- 옵티마이저를 만들 때 훈련 과정에서 최적화시킬 파이토치 텐서를 전달해야 함
- 모델 객체의 parameters( ) 메서드를 호출하면 훈련 가능한 모든 모델 파라미터를 전달
- 이 메서드는 제너레이터 객체를 반환하기 때문에 다음처럼 for 문으로 모델 파라미터 목록을 확인

```
for params in model.parameters():
print(params.shape)
```

→

```
torch.Size([100, 784])
torch.Size([100])
torch.Size([10, 100])
torch.Size([10])
```

위에서부터 순서대로 첫 번째 Linear 층의 가중치와 절편, 두 번째 Linear 층의 가중치와 절편

SECTION 7-2 파이토치 버전 살펴보기(10)

- 파이토치에는 케라스와 같은 fit ( ) 메서드가 없음
  - 에포크와 배치 경사 하강법을 위한 for 문을 직접 구현

```
for 에포크 반복
    에포크 손실 초기화
    for 배치 반복
        배치 입력과 타깃 준비
        옵티마이저 그레디언트 초기화
        모델에 입력 전달
        모델 출력과 타깃으로 손실 계산
        손실 역전파
        손실 파라미터 업데이트
        에포크  손실 기록
    에포크 손실 출력
```

- 두 개의 for 문을 중첩하여 에포크와 미니 배치를 반복
  - 여기에서는 본문에서와 동일하게 에포크 횟수를 5로 지정하고, 32개의 샘플을 하나의 배치로 사용
  - 훈련 세트의 샘플 개수가 48,000개이므로 각 에포크에서 1,500번이 반복
- 에포크마다 배치를 반복하기 전에 훈련 손실을 기록할 변수를 초기화
- 배치 반복이 모두 끝난 후 마지막에 훈련 손실을 출력
- 배치 반복 루프
  - 전체 훈련 세트 중에서 32개씩 선택해 배치 입력과 타깃을 준비
  - 옵티마이저의 그레이디언트를 초기화
  - 준비한 배치를 모델에 전달하여 출력을 생성
    - 정방향 계산(forward pass) 또는 순전파(forward propagation)
  - 모델이 계산한 출력과 타깃을 손실 함수에 전달하여 손실을 계산
  - 손실을 모델의 출력층에서부터 입력층 방향으로 거꾸로 전달하여 각 층의 모델 파라미터에 대한 그레이디언트를 계산 - 역전파(backpropagation)
  - 옵티마이저 객체는 계산된 그레이디언트를 사용해 손실 함수가 감소되는 방향으로 모델 파라미터를 업데이트
- 마지막으로 손실값을 기록하고 for 반복문을 종료

SECTION 7-2 파이토치 버전 살펴보기(11)

- 예제 코드와 출력 결과

```
epochs = 5
batches = int(len(train_scaled)/32)
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for i in range(batches):
        inputs = train_scaled[i*32:(i+1)*32].to(device)
        targets = train_target[i*32:(i+1)*32].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    print(f"에포크:{epoch + 1}, 손실:{train_loss/batches:.4f}")
```

→

```
에포크:1, 손실:0.5428
에포크:2, 손실:0.4004
에포크:3, 손실:0.3594
에포크:4, 손실:0.3320
에포크:5, 손실:0.3119
```

SECTION 7-2 파이토치 버전 살펴보기(12)

○ 모델의 성능 평가

- model.train ( )과 유사하게 평가할 때는 model.eval ( ) 메서드를 호출하여 모델을 평가함을 알림
- 그다음 파이썬 with 문으로 torch.no_grad( )를 호출하여 그레이디언트 계산을 하지 않는다고 알림
  - 이를 사용하면 메모리와 계산량이 줄어 들기 때문에 모델을 훈련하지 않는 경우에는 꼭 호출
- with 문 블록 안에서 검증 세트와 타깃을 GPU에 적재하고, 모델의 출력을 계산해 타깃과 비교
- outputs는 검증 세트의 샘플 12,000개에 대해 타깃 클래스마다 출력한 값
  - 따라서 이 텐서의 크기는 (12000, 10)
- 예측 클래스 - 각 샘플마다 가장 큰 값의 인덱스를 추출
  - torch.argmax( ) 함수를 사용해 두 번째 축을 따라 가장 큰 값의 인덱스를 predicts에 저장
- predicts와 val_target을 비교하여 올바르게 예측한 개수를 헤아려 corrects 변수에 저장
  - 이 값을 검증 세트의 샘플 개수로 나누면 검증 정확도가 됨

SECTION 7-2 파이토치 버전 살펴보기(13)

- 모델의 성능 평가 코드와 출력 결과

```
model.eval()
with torch.no_grad():
    val_scaled = val_scaled.to(device)
    val_target = val_target.to(device)
    outputs = model(val_scaled)
    predicts = torch.argmax(outputs, 1)
    corrects = (predicts == val_target).sum().item()

accuracy = corrects / len(val_target)
print(f"검증 정확도: {accuracy:.4f}")
```

→ 검증 정확도: 0.8719

SECTION 7-3 신경망 모델 훈련(1)

○ 손실 곡선

- 7-2에서 fit ( ) 메서드로 모델을 훈련하면 에포크 횟수, 손실, 정확도 등 훈련 과정이 상세하게 출력됨
  - 이 출력의 마지막에 다음과 같은 메시지를 포함

```
<keras.src.callbacks.history.History at 0x7956692ffdc0>
```

- 케라스의 fit ( ) 메서드는 History 클래스 객체를 반환
  - Hisotry 객체에는 훈련 과정에서 계산한 지표, 즉 손실과 정확도 값이 저장
  - 이 값을 사용하면 그래프를 그릴 수 있음
- 패션 MNIST 데이터셋을 적재하고 훈련 세트와 검증 세트로 나누기

```
from tensorflow import keras
from sklearn.model_selection import train_test_split
(train_input, train_target), (test_input, test_target) =\
        keras.datasets.fashion_mnist.load_data()
train_scaled = train_input / 255.0
train_scaled, val_scaled, train_target, val_target = train_test_split(
        train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 7-3 신경망 모델 훈련(2)

○ 손실 곡선

- if 구문으로 model_fn ( ) 함수에 (a_layer 매개변수로) 케라스 층을 추가하면 은닉층 뒤에 또 하나의 층을 추가

```
def model_fn(a_layer=None):
    model = keras.Sequential()
    model.add(keras.layers.Input(shape=(28,28)))
    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(100, activation='relu'))
    if a_layer:
        model.add(a_layer)
    model.add(keras.layers.Dense(10, activation='softmax'))
    return model
```

SECTION 7-3 신경망 모델 훈련(3)

○ 손실 곡선

- a_layer 매개변수로 층을 추가하지 않고 단순하게 model_fn ( ) 함수를 호출하고 모델 구조를 출력
  - 이전 절과 동일한 모델이라는 것을 확인

```
model = model_fn()
model.summary()
```

→

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 100)	78,500
dense_1 (Dense)	(None, 10)	1,010

```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

SECTION 7-3 신경망 모델 훈련(4)

○ 손실 곡선

- **이전 절과 동일하게 모델을 훈련하지만 fit ( ) 메서드의 결과를 history 변수에 담기**

```
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=5, verbose=0)
```

- **훈련 측정값이 담겨 있는 history 딕셔너리 확인**

```
print(history.history.keys())
```

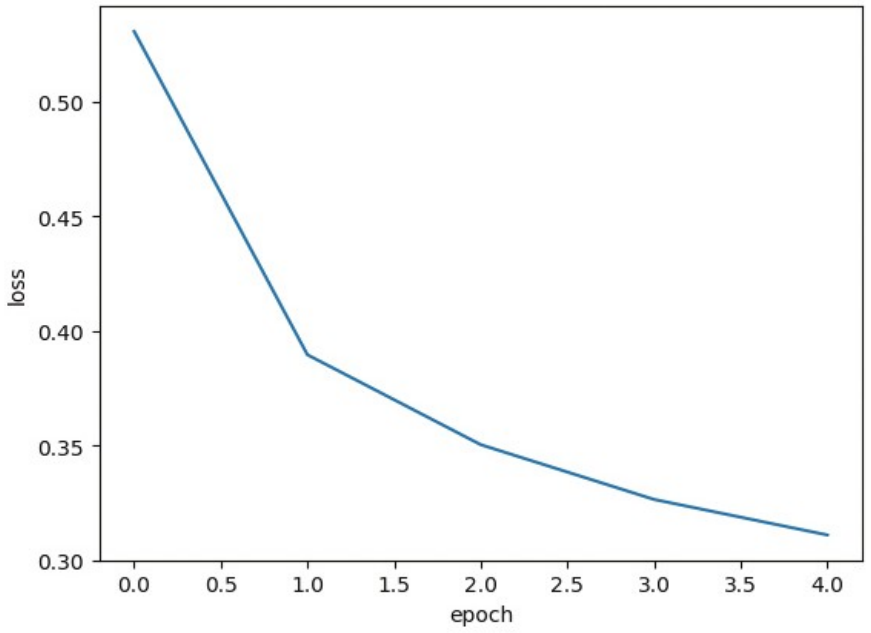
→ dict_keys(['accuracy', 'loss'])

- **맷플롯립을 사용해 history 속성에 포함된 손실 그래프 그리기**

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

→



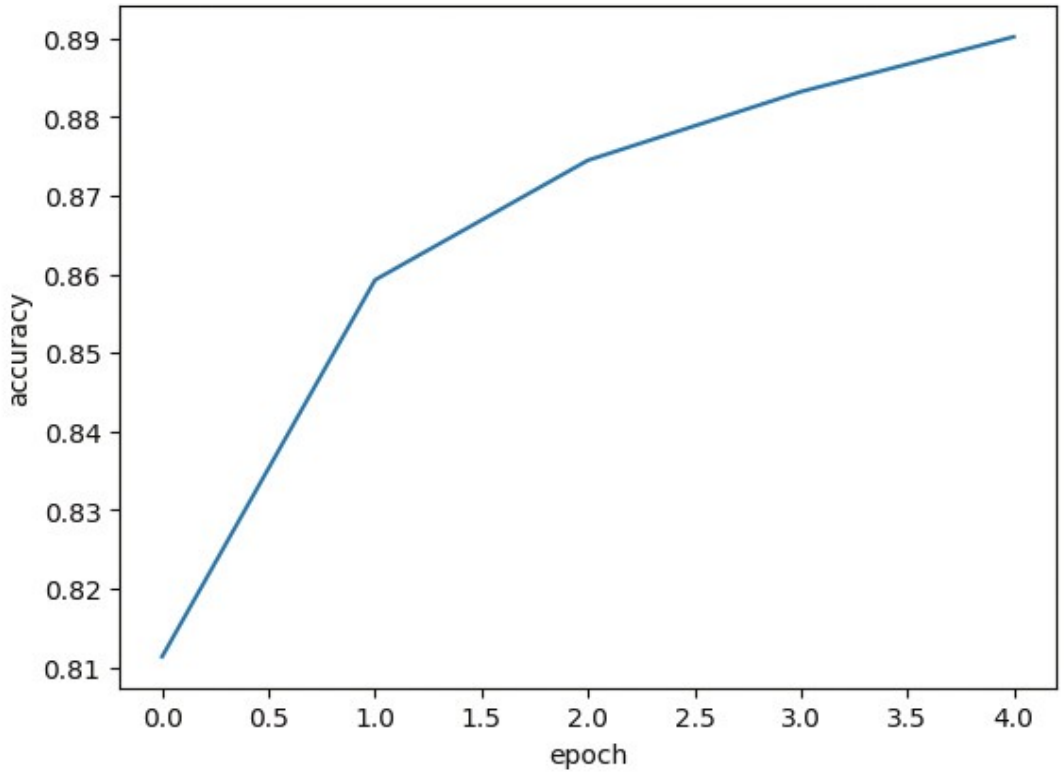
SECTION 7-3 신경망 모델 훈련(5)

○ 손실 곡선

- 맷플롯립을 사용해 history 속성의 정확도 그래프 그리기

```
plt.plot(history.history['accuracy'])
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

→

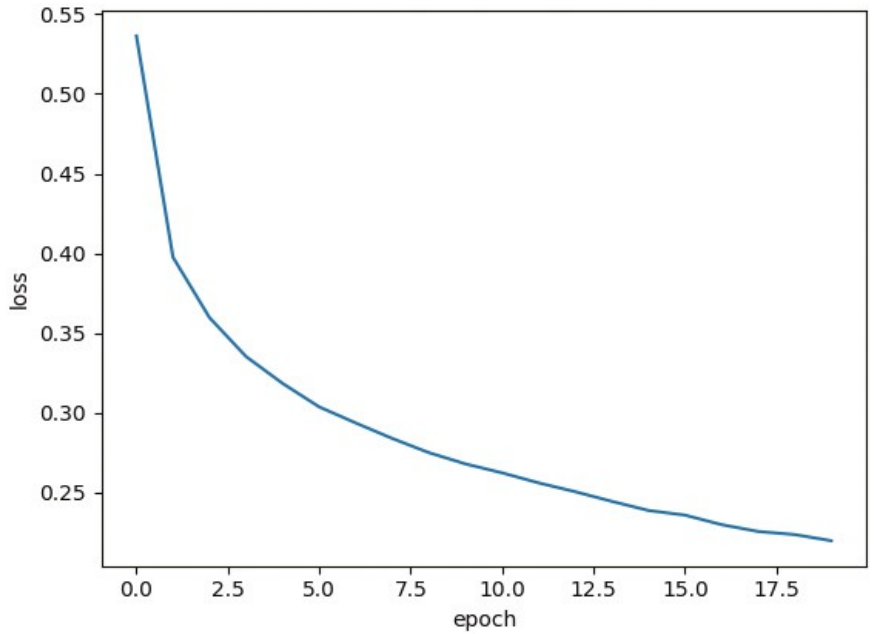


SECTION 7-3 신경망 모델 훈련(6)

○ 손실 곡선

- **에포크 횟수를 20으로 늘려서 모델을 훈련하고 손실 그래프 그리기**

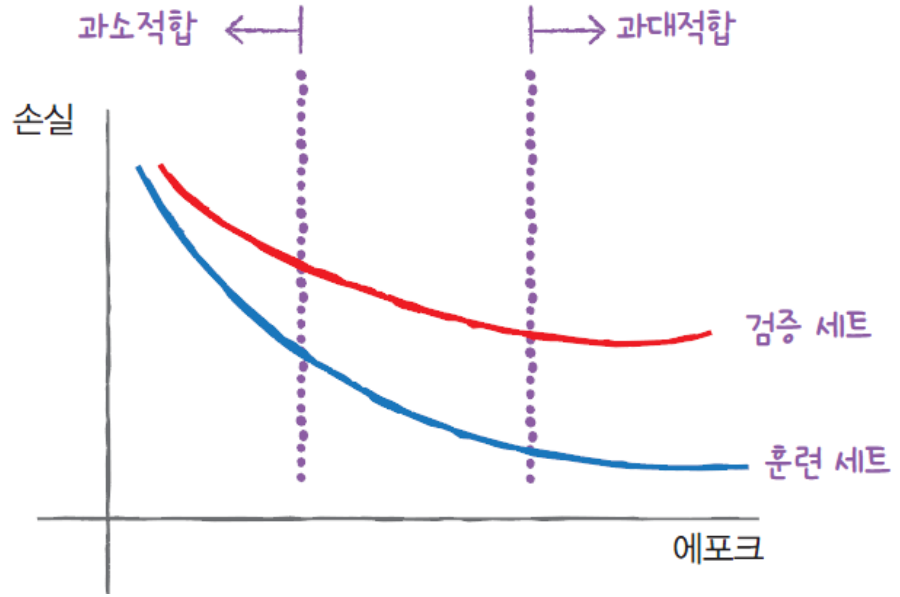
```
model = model_fn()
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=20, verbose=0)
plt.plot(history.history['loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```



SECTION 7-3 신경망 모델 훈련(7)

○ 검증 손실

- 검증 세트에 손실을 사용하여 에포크에 따른 과대/과소적합 파악



✚ 여기서 잠깐 손실을 사용하는 것과 정확도를 사용하는 것은 어떤 차이가 있나?

- 인공 신경망 모델이 최적화하는 대상은 정확도가 아니라 손실 함수
- 이따금 손실 감소에 비례하여 정확도가 높아지지 않는 경우도 있으므로, 모델이 잘 훈련되었는지 판단하려면 정확도보다는 손실 함수의 값을 확인하는 것이 더 좋음

SECTION 7-3 신경망 모델 훈련(8)

○ 검증 손실

- 검증 세트에 손실을 사용하여 에포크에 따른 과대/과소적합 파악
- 에포크마다 검증 손실을 계산하기 위해 케라스 모델의 fit ( ) 메서드에 검증 데이터를 전달
  - validation_data 매개변수에 검증에 사용할 입력과 타깃값을 튜플로 만들어 전달

```
model = model_fn()
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=20, verbose=0,
                    validation_data=(val_scaled, val_target))
```

- 반환된 history.history 딕셔너리의 키 확인

```
print(history.history.keys())
```

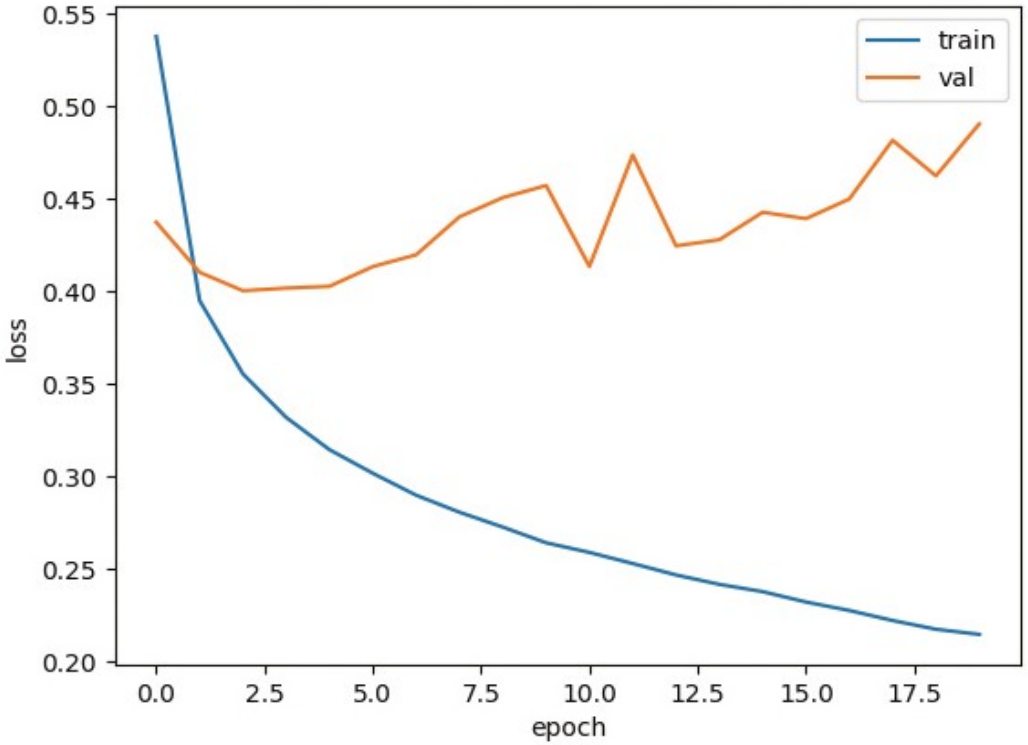
→ dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])

SECTION 7-3 신경망 모델 훈련(9)

○ 검증 손실

- **과대/과소적합 문제를 조사하기 위해 훈련 손실과 검증 손실을 한 그래프에 그리기**

```
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



▲ 초기에 검증 손실이 감소하다가 다섯 번째 에포크 만에 다시 상승하기 시작
- 훈련 손실은 꾸준히 감소하기 때문에 전형적인 과대적합 모델이 만들어짐
- 검증 손실이 상승하는 시점을 가능한 뒤로 늦추면 검증 세트에 대한 손실이 줄어들 뿐만 아니라 검증 세트에 대한 정확도도 증가

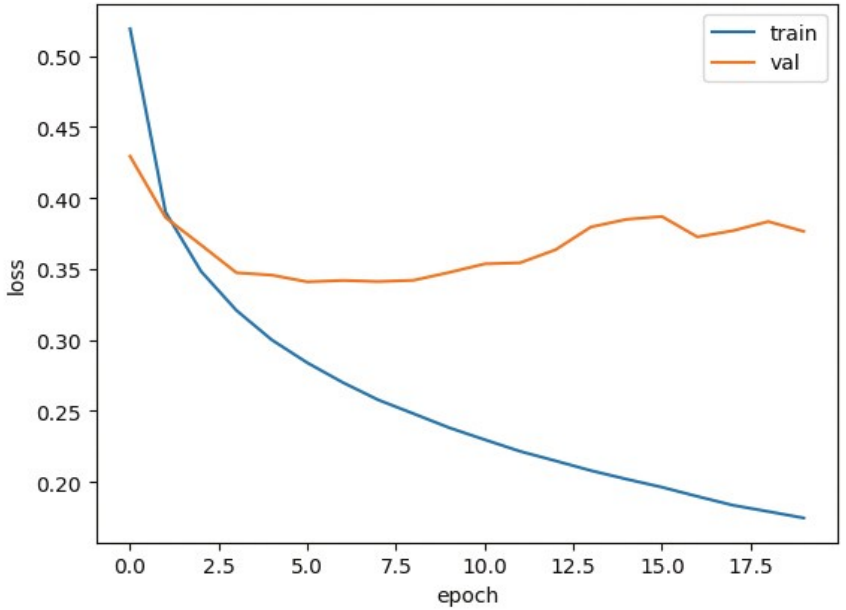
SECTION 7-3 신경망 모델 훈련(10)

○ 검증 손실

- 옵티마이저 하이퍼파라미터를 조정하여 과대적합을 완화
- Adam은 적응적 학습률을 사용하기 때문에 에포크가 진행되면서 학습률의 크기 조정 가능
- Adam 옵티마이저를 적용해 보고 훈련 손실과 검증 손실 그래프 그리기

```
model = model_fn()
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=20, verbose=0,
        validation_data=(val_scaled, val_target))
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

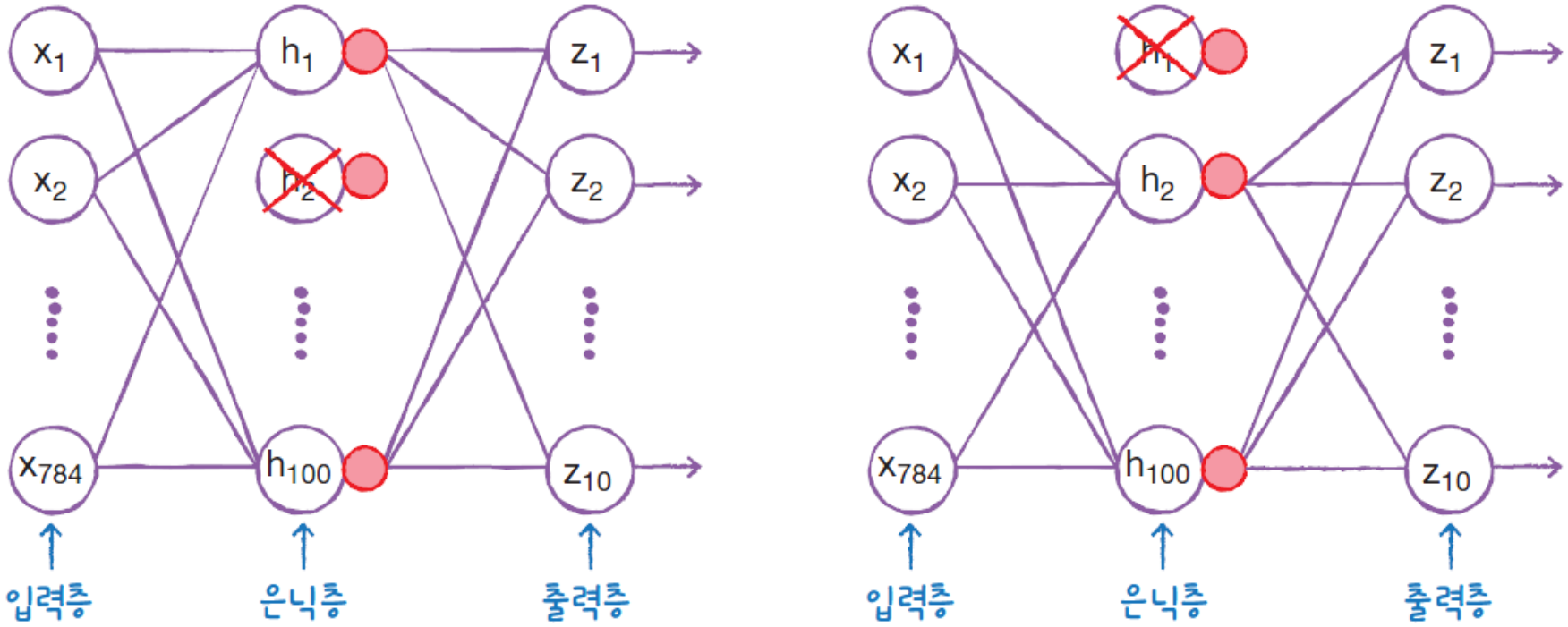
→



SECTION 7-3 신경망 모델 훈련(11)

○ 드롭아웃(dropout)

- 훈련 과정에서 층에 있는 일부 뉴런을 랜덤하게 꺼서(즉 뉴런의 출력을 0으로 만들어) 과대적합을 규제



SECTION 7-3 신경망 모델 훈련(12)

○ 드롭아웃(dropout)

- 정의한 model_fn ( ) 함수에 드롭아웃 객체를 전달하여 층을 추가(30% 정도를 드롭아웃)
- summary( ) 메서드를 사용해 드롭아웃 층이 잘 추가되었는지 확인

```
model = model_fn(keras.layers.Dropout(0.3))
model.summary()
```

→

```
Model: "sequential_4"
```

Layer (type)	Output Shape	Param #
flatten_4 (Flatten)	(None, 784)	0
dense_8 (Dense)	(None, 100)	78,500
dropout (Dropout)	(None, 100)	0
dense_9 (Dense)	(None, 10)	1,010

```
Total params: 79,510 (310.59 KB)
Trainable params: 79,510 (310.59 KB)
Non-trainable params: 0 (0.00 B)
```

▲ 은닉층 뒤에 추가된 드롭아웃 층(Dropout)은 훈련되는 모델 파라미터가 없으며,
입력과 출력의 크기가 같음
일부 뉴런의 출력을 0으로 만들지만 전체 출력 배열의 크기를 바꾸지는 않음

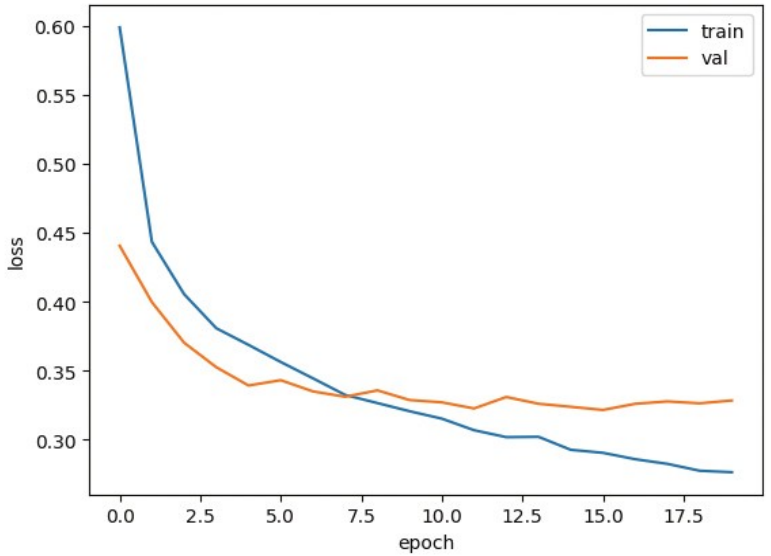
SECTION 7-3 신경망 모델 훈련(13)

○ 드롭아웃(dropout)

- **훈련 손실과 검증 손실의 그래프 그리기**

  - 평가와 예측에 모델을 사용할 때는 드롭아웃이 적용되지 않음

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=20, verbose=0,
        validation_data=(val_scaled, val_target))
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



SECTION 7-3 신경망 모델 훈련(14)

○ 모델 저장과 복원

- 에포크 횟수를 11로 다시 지정하고 모델을 훈련

```
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_scaled, train_target, epochs=11, verbose=0,
              validation_data=(val_scaled, val_target))
```

- save( ) 메서드

  - .keras 확장자를 가진 파일에 필요한 정보를 모두 압축하여 저장

```
model.save('model-whole.keras')
```

- save_weights ( ) 메서드
  - 훈련된 모델의 파라미터만 저장
  - 이 메서드는 파라미터를 HDF5 포맷으로 저장하며 파일의 확장자는 weights.h5로 끝나야 함

```
model.save('model-whole.h5')
```

- 파일 저장 확인

```
!ls -al model*
```

→

```
-rw-r--r-- 1 root root 971928 Jan 10 09:56 model.weights.h5
-rw-r--r-- 1 root root 974764 Jan 10 09:56 model-whole.keras
```

SECTION 7-3 신경망 모델 훈련(15)

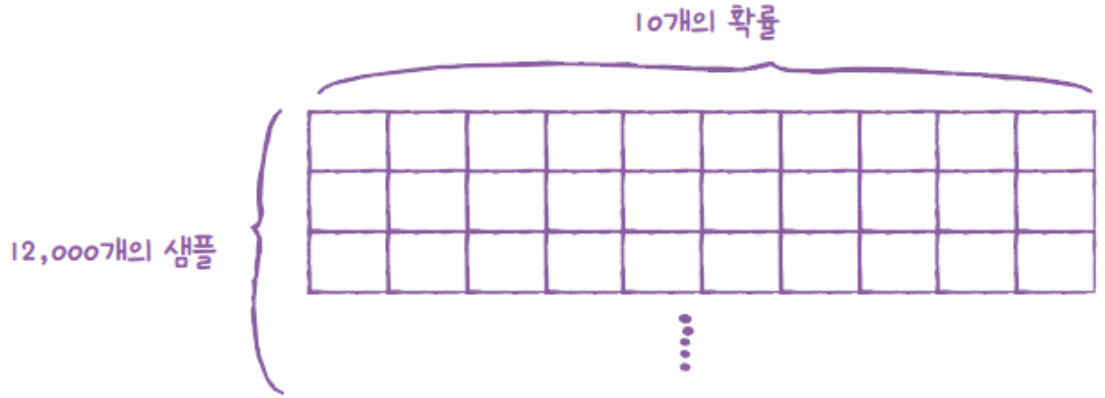
○ 모델 저장과 복원

- 훈련을 하지 않은 새로운 모델을 만들고 model.weights.h5 파일에서 훈련된 모델 파라미터를 읽어서 사용하기

  - 훈련하지 않은 새로운 모델을 만들고 이전에 저장했던 모델 파라미터를 적재
    -save_weights( )와 쌍을 이루는 load_weights( ) 메서드

```
model = model_fn(keras.layers.Dropout(0.3))
model.load_weights('model.weights.h5')
```

  - 모델의 검증 정확도를 확인
    -예측을 수행하는 predict ( ) 메서드는 사이킷런과 달리 샘플마다 10개의 클래스에 대한 확률을 반환
    - 패션 MNIST 데이터셋에서 덜어낸 검증 세트의 샘플 개수는 12,000개이기 때문에 predict ( ) 메서드는 (12000, 10) 크기의 배열을 반환



SECTION 7-3 신경망 모델 훈련(16)

○ 모델 저장과 복원

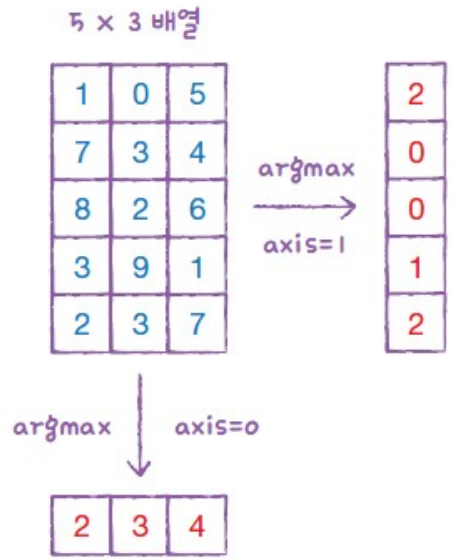
- 훈련을 하지 않은 새로운 모델을 만들고 model-weights.h5 파일에서 훈련된 모델 파라미터를 읽어서 사용하기
  - 10개 확률 중에 가장 큰 값의 인덱스를 골라 타깃 레이블과 비교하여 정확도를 계산

```
import numpy as np

val_labels = np.argmax(model.predict(val_scaled), axis=-1)
print(np.mean(val_labels == val_target))
```

→ 375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
0.8788333333333334

- 모델의 predict ( ) 메서드 결과에서 가장 큰 값을 고르기 위해 넘파이 argmax( ) 함수를 사용
  - 배열에서 가장 큰 값의 인덱스를 반환
- axis=1이면 열을 따라 각 행의 최댓값의 인덱스를 선택하고,
  axis=0이면 행을 따라 각 열의 최댓값의 인덱스를 선택
  그다음 라인은 argmax( )로 고른 인덱스(val_labels)와
  타깃(val_target)을 비교
  두 배열에서 각 위치의 값이 같으면 1이되고 다르면 0
  이를 평균하면 정확도



SECTION 7-3 신경망 모델 훈련(17)

○ 모델 저장과 복원

- 모델 전체를 파일에서 읽은 다음 검증 세트의 정확도를 출력
  - load_model( ) 함수 사용

```
model = keras.models.load_model('model-whole.keras')
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8799 - loss: 0.3312
[0.3367460072040558, 0.8788333535194397]]
```

  - 같은 모델을 저장하고 다시 불러들였기 때문에 위와 동일한 정확도를 획득

SECTION 7-3 신경망 모델 훈련(18)

○ 콜백(callback)

- 훈련 과정 중간에 어떤 작업을 수행할 수 있게 하는 객체로 keras.callbacks 패키지 아래에 있는 클래스들
  fit( ) 메서드의 callbacks 매개변수에 리스트로 전달하여 사용
- ModelCheckpoint 콜백은 기본적으로 최상의 검증 점수를 만드는 모델을 저장
- save_best_only=True 매개변수를 지정하여 가장 낮은 검증 손실을 만드는 모델을 저장
- 저장될 파일 이름을 'best-model.keras'로 지정하여 콜백을 적용

```
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-model.keras',
                                                save_best_only=True)
model.fit(train_scaled, train_target, epochs=20, verbose=0,
      validation_data=(val_scaled, val_target),
      callbacks=[checkpoint_cb])
```

SECTION 7-3 신경망 모델 훈련(19)

○ 콜백(callback)

- 모델이 훈련한 후에 best-model.keras에 최상의 검증 점수를 낸 모델이 저장됨
- 이 모델을 load_model( ) 함수로 다시 읽어서 예측 수행

```
model = keras.models.load_model('best-model.keras')
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy:
0.8886 - loss: 0.3160
[0.3214062154293603, 0.8859166502952576]
```

- 조기 종료(early stopping): 과대적합이 시작되기 전에 훈련을 미리 중지하는 것
  - EarlyStopping 콜백의 patience 매개변수는 검증 점수가 향상되지 않더라도 참을 에포크 횟수로 지정

SECTION 7-3 신경망 모델 훈련(20)

○ 콜백(callback)

- EarlyStopping 콜백을 ModelCheckpoint 콜백과 함께 사용하면 가장 낮은 검증 손실의 모델을 파일에 저장하고 검증 손실이 다시 상승할 때 훈련을 중지할 수 있음
- 훈련을 중지한 다음 현재 모델의 파라미터를 최상의 파라미터로 되돌림

```
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-model.keras',
                         save_best_only=True)
early_stopping_cb = keras.callbacks.EarlyStopping(patience=2,
                         restore_best_weights=True)
history = model.fit(train_scaled, train_target, epochs=20, verbose=0,
                        validation_data=(val_scaled, val_target),
                        callbacks=[checkpoint_cb, early_stopping_cb])
```

- 훈련 후 몇 번째 에포크에서 훈련이 중지되었는지 early_stopping_cb 객체의 stopped_epoch 속성에서 확인

```
print(early_stopping_cb.stopped_epoch)
```

→ 14

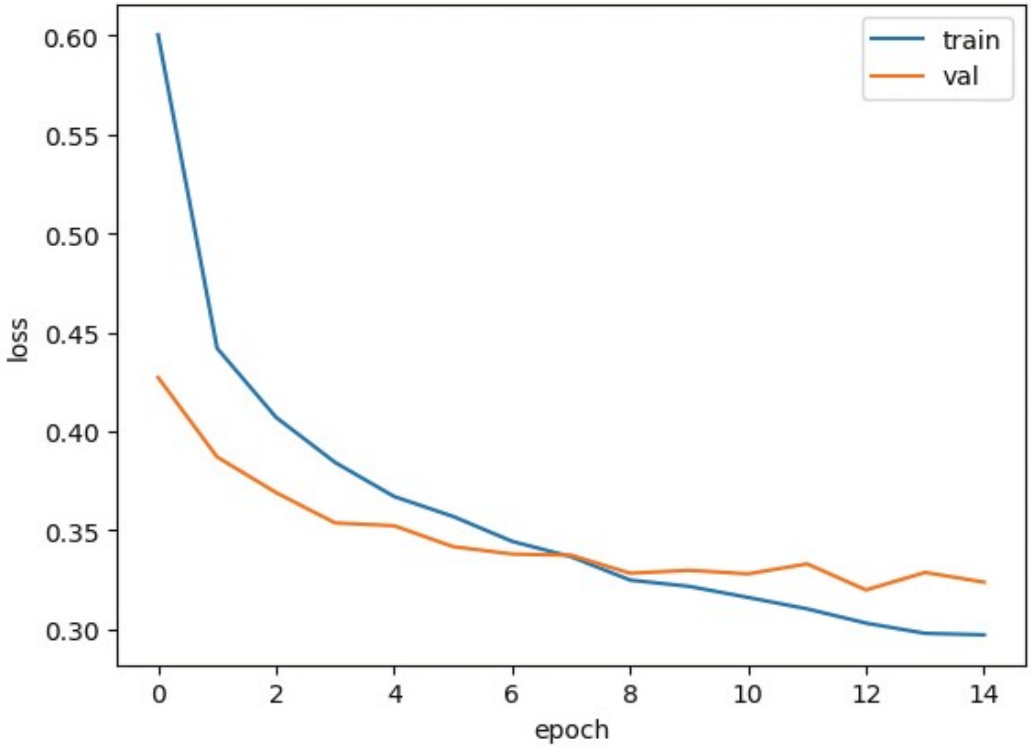
SECTION 7-3 신경망 모델 훈련(21)

○ 콜백(callback)

- 에포크 횟수가 0부터 시작하기 때문에 14는 열다섯 번째 에포크에서 훈련이 중지되었다는 것을 의미
- patience를 2로 지정했으므로 최상의 모델은 열세 번째 에포크
- 훈련 손실과 검증 손실을 출력해서 확인

```
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

→



SECTION 7-3 신경망 모델 훈련(22)

○ 콜백(callback)

- **조기 종료로 얻은 모델을 사용해 검증 세트에 대한 성능 확인**

```
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy:
0.8876 - loss: 0.3189
[0.3199739754199817, 0.8846666812896729]
```

SECTION 7-3 신경망 모델 훈련(23)

○ 최상의 신경망 모델 얻기(문제해결 과정)

- 문제
  - 패션 럭키백을 위한 분류 모델 만들기
- 학습
  - 인공 신경망 모델을 훈련하기 위한 다양한 도구 학습
  - fit ( ) 메서드의 반환값을 사용해 훈련 세트와 검증 세트에 대한 손실을 그래프로 그리기
    - 이를 위해 fit ( ) 메서드는 훈련 세트뿐만 아니라 검증 세트를 전달할 수 있는 매개변수를 제공
  - 과대적합을 막기 위해 신경망에서 즐겨 사용하는 대표적인 규제 방법인 드롭아웃
  - 드롭아웃은 일부 뉴런의 출력을 랜덤하게 꺼서 일부 뉴런에 의존하는 것을 막고 마치 많은 신경망을 앙상블 하는 효과
  - 케라스에서는 드롭아웃을 층으로 제공하기 때문에 밀집층을 추가하듯이 간편하게 모델의 원하는 곳에 드롭아웃을 추가할 수 있음
  - 케라스는 훈련된 모델의 파라미터를 저장하고 다시 불러오는 메서드를 제공
  - 또한 모델 전체를 파일에 저장하고 파일에서 모델을 만들 수도 있음
  - 과대적합 되기 전의 에포크를 수동으로 찾아 모델을 다시 훈련하는 대신 콜백을 사용하면 자동으로 최상의 모델을 유지할 수 있음

SECTION 7-3 마무리(1)

○ 키워드로 끝나는 핵심 포인트

- 드롭아웃은 은닉층에 있는 뉴런의 출력을 랜덤하게 꺼서 과대적합을 막는 기법
  - 드롭아웃은 훈련 중에 적용되며 평가나 예측에서는 적용하지 않고, 텐서플로는 이를 자동으로 처리
- 콜백은 케라스 모델을 훈련하는 도중에 어떤 작업을 수행할 수 있도록 도와주는 도구
  - 대표적으로 최상의 모델을 자동으로 저장해 주거나 검증 점수가 더 이상 향상되지 않으면 일찍 종료할 수 있음
- 조기 종료는 검증 점수가 더 이상 감소하지 않고 상승하여 과대적합이 일어나면 훈련을 계속 진행하지 않고 멈추는 기법
  - 계산 비용과 시간을 절약

SECTION 7-3 마무리(2)

○ 핵심 패키지와 함수

- Keras
  - Dropout: 드롭아웃 층
  - save_weights( ): 모든 층의 가중치와 절편을 파일에 저장
    - 파일 이름은 반드시 '.weights.h5'로 끝나야 함
  - load_weights( ): 모든 층의 가중치와 절편을 파일에 읽기
  - save( ): 모델 구조와 모든 가중치와 절편을 파일에 저장
    - 기본적으로 케라스 3.x 포맷으로 저장
    - 파일 이름은 '.keras'로 끝나야 함
  - load_model ( ): model.save( )로 저장된 모델을 로드
  - ModelCheckpoint: 케라스 모델과 가중치를 일정 간격으로 저장
  - EarlyStopping: 관심 지표가 더이상 향상하지 않으면 훈련을 중지
- NumPy
  - argmax: 배열에서 축을 따라 최댓값의 인덱스를 반환

SECTION 7-3 확인 문제(1)

1 케라스 모델의 fit ( ) 메서드에 검증 세트를 올바르게 전달하는 코드는?

① model.fit (..., val_input=val_input, val_target=val_target)
② model.fit (..., validation_input=val_input, validation_target=val_target)
③ model.fit (..., val_data=(val_input, val_target))
④ model.fit (..., validation_data=(val_input, val_target))

2 이전 층의 뉴런 출력 중 70%만 사용하기 위해 드롭아웃 층을 추가하려고 할 때 다음 중 옳게 설정한 것은?

① Dropout(0.7)
② Dropout(0.3)
③ Dropout(1/0.7)
④ Dropout(1/0.3)

SECTION 7-3 확인 문제(2)

3. 케라스 모델의 가중치만 저장하는 메서드는?

① save( )
② load_model( )
③ save_weights( )
④ load_weights( )

4. 케라스의 조기 종료 콜백을 사용하려고 할 때, 3번의 에포크 동안 손실이 감소되지 않으면 종료하고 최상의 모델 가중치를 복원하도록 올바르게 설정한 것은?

① EarlyStopping(monitor='loss', patience=3)
② EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
③ EarlyStopping(monitor='accuracy', patience=3)
④ EarlyStopping (monitor='val_accuracy', patience=3, restore_best_weights=True)

SECTION 7-3 파이토치 버전 살펴보기(1)

○ 파이토치로 신경망 모델 만들기

- 학습 목표
  - 훈련 세트 손실과 검증 세트 손실을 기록하고, 그래프로 시각화
  - 검증 손실이 일정 에포크 동안 향상되지 않으면 자동으로 훈련을 종료하는 '조기 종료' 기법 구현
- 패션 MNIST 데이터를 로드하고, 훈련 세트와 검증 세트를 준비

```
from torchvision.datasets import FashionMNIST

fm_train = FashionMNIST(root='.', train=True, download=True)
fm_test = FashionMNIST(root='.', train=False, download=True)

train_input = fm_train.data
train_target = fm_train.targets
train_scaled = train_input / 255.0
from sklearn.model_selection import train_test_split

train_scaled, val_scaled, train_target, val_target = train_test_split(
    train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 7-3 파이토치 버전 살펴보기(2)

○ 파이토치로 신경망 모델 만들기

- 07-3절 본문에서 만든 모델과 동일하게 두 개의 밀집층 사이에 드롭아웃을 추가
- 파이토치의 드롭아웃 비율은 0.3으로 지정하고 만든 모델을 GPU에 적재

```
import torch.nn as nn

model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(100, 10)
)

import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

SECTION 7-3 파이토치 버전 살펴보기(3)

- 손실 함수와 옵티마이저

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

SECTION 7-3 파이토치 버전 살펴보기(4)

- 모델 훈련 코드
  - 변수 준비

```
train_hist = []
val_hist = []
patience = 2
best_loss = -1
early_stopping_counter = 0
```

    - 훈련 손실과 검증 손실을 기록하기 위해 train_hist와 val_hist 리스트를 생성
    - 조기 종료를 구현하기 위해서는 세 개의 추가 변수가 필요
      - patience - 검증 손실이 향상될 때까지 에포크 횟수를 설정
      - best_loss - 최상의 손실을 기록
      - early_stopping_counter - 연속적으로 검증 손실이 향상되지 않은 에포크 횟수를 기록
    - early_stopping_counter가 patience보다 크거나 같으면 더 이상 모델을 훈련하지 않고 종료

SECTION 7-3 파이토치 버전 살펴보기(5)

- 훈련 반복문 (7-2절과 동일함)

```
epochs = 20
batches = int(len(train_scaled)/32)
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for i in range(batches):
        inputs = train_scaled[i*32:(i+1)*32].to(device)
        targets = train_target[i*32:(i+1)*32].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
```

SECTION 7-3 파이토치 버전 살펴보기(6)

- 검증 세트에 대한 손실을 계산하는 코드 추가
  - 이 코드 블록은 첫 번째 for 반복문 안에 속하므로 들여쓰기에 유의

```
model.eval()
val_loss = 0
with torch.no_grad():
    val_scaled = val_scaled.to(device)
    val_target = val_target.to(device)
    outputs = model(val_scaled)
    loss = criterion(outputs, val_target)
    val_loss = loss.item()
```

  - 손실 함수 객체인 criterion에 모델 출력과 타깃을 전달해 손실을 계산하고 val_loss 변수에 저장

SECTION 7-3 파이토치 버전 살펴보기(7)

- train_hist와 val_hist 리스트에 훈련 손실과 검증 손실을 추가

```
train_hist.append(train_loss/batches)
val_hist.append(val_loss)
print(f"에포크:{epoch+1},",
    f"훈련 손실:{train_loss/batches:.4f}, 검증 손실:{val_loss:.4f}")
```

SECTION 7-3 파이토치 버전 살펴보기(8)

- 조기 종료를 위한 코드
  - 첫 번째 에포크이거나 (best_loss == -1) 검증 손실이 이전에 기록된 최상의 손실보다 작으면(val_loss < best_loss) 현재 검증 손실을 최상의 손실로 저장(best_loss = val_loss)
  - 조기 종료 카운터를 0으로 설정
    - 이 카운터를 0으로 설정하면 검증 손실이 더 좋아지지 않더라도 patience에 지정된 횟수만큼 참고 기다림
  - 현재 검증 손실이 최상이므로 torch.save ( ) 함수를 사용해 모델을 저장
    - torch.save(model, 'some_model.pt')와 같이 이 함수에 모델 객체와 파일 이름을 지정하면 모델 구조와 모델 파라미터가 모두 저장
    - 일반적으로 권장되는 방식을 따라서 모델의 state_dict ( ) 메서드로 모델 파라미터와 훈련하는 동안 기록된 다른 값(10장에서 배울 층 정규화의 파라미터 등)을 반환
    - 딕셔너리를 best_model.pt 파일에 기록
    - 만약 검증 손실이 더 나아지지 않았다면 조기 종료 카운터를 1 증가
      - 이 카운터가 patience보다 크다면 조기 종료된다는 메시지와 함께 훈련을 중지

SECTION 7-3 파이토치 버전 살펴보기(9)

- 앞의 코드는 모두 첫 번째 for 반복문 안에 포함
  - 코드를 실행 결과 – 10번째 에포크에서 조기 종료

```
if best_loss == -1 or val_loss < best_loss:
        best_loss = val_loss
        early_stopping_counter = 0
        torch.save(model.state_dict(), 'best_model.pt')
    else:
        early_stopping_counter += 1
    if early_stopping_counter >= patience:
        print(f"{epoch+1}번째 에포크에서 조기 종료되었습니다.")
        Break
```

→

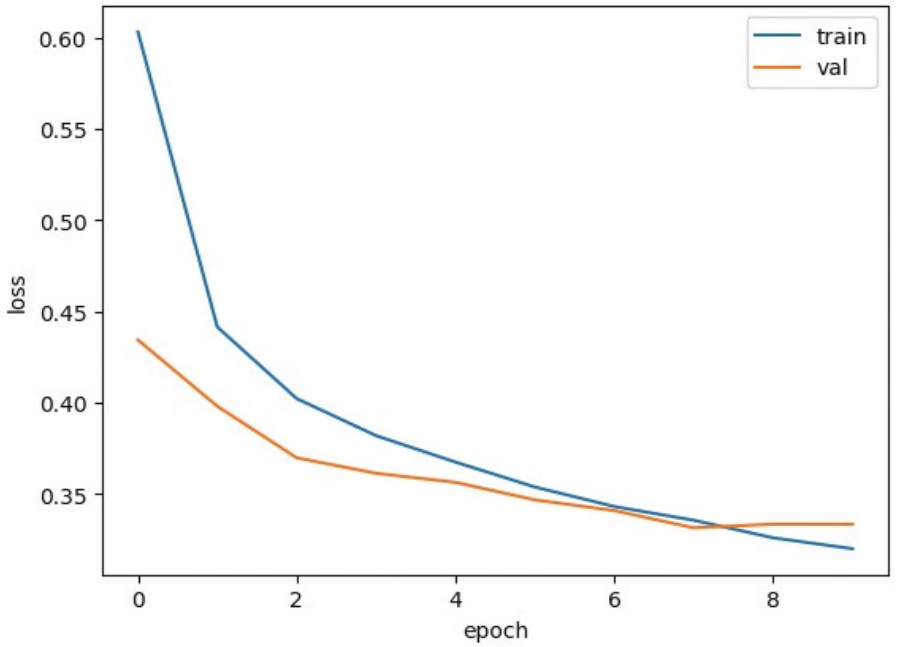
```
에포크:1, 훈련 손실:0.6031, 검증 손실:0.4344
에포크:2, 훈련 손실:0.4415, 검증 손실:0.3981
에포크:3, 훈련 손실:0.4023, 검증 손실:0.3699
에포크:4, 훈련 손실:0.3820, 검증 손실:0.3614
에포크:5, 훈련 손실:0.3675, 검증 손실:0.3564
에포크:6, 훈련 손실:0.3539, 검증 손실:0.3468
에포크:7, 훈련 손실:0.3432, 검증 손실:0.3410
에포크:8, 훈련 손실:0.3357, 검증 손실:0.3315
에포크:9, 훈련 손실:0.3261, 검증 손실:0.3335
에포크:10, 훈련 손실:0.3201, 검증 손실:0.3335
10번째 에포크에서 조기 종료되었습니다.
```

SECTION 7-3 파이토치 버전 살펴보기(10)

- 훈련 손실과 검증 손실의 그래프

```
import matplotlib.pyplot as plt

plt.plot(train_hist, label='train')
plt.plot(val_hist, label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



SECTION 7-3 파이토치 버전 살펴보기(11)

- torch.load( ) 함수로 best_model.pt 파일에 저장한 모델 파라미터를 읽은 다음 load_state_dict ( ) 메서드에 전달하여 최상의 파라미터로 model 객체를 업데이트

```
model.load_state_dict(torch.load('best_model.pt',
weights_only=True))
```

  - torch.load( ) 함수의 weights_only 매개변수 기본값은 False
  - 향후 기본값이 True로 바뀐다는 경고가 발생하기 때문에 앞에서 명시적으로 weights_only=True로 지정

SECTION 7-3 파이토치 버전 살펴보기(12)

- 최상의 모델을 사용해 검증 세트에 대한 성능을 확인
  - 이 코드는 7-2 절의 파이토치 코드와 동일

```
model.eval()
with torch.no_grad():
    val_scaled = val_scaled.to(device)
    val_target = val_target.to(device)
    outputs = model(val_scaled)
    predicts = torch.argmax(outputs, 1)
    corrects = (predicts == val_target).sum().item()

accuracy = corrects / len(val_target)
print(f"검증 정확도: {accuracy:.4f}")
```

→ 검증 정확도: 0.8798

SECTION 7-3 자주하는 질문

- 07-1절에서 사이킷런 SGDClassifier의 fit ( ) 메서드와 케라스 모델의 fit ( ) 메서드가 동일한건가요?
- 07-1절에서 검증 세트를 나누기 전에 특성을 정규화한 것 아닌가요?
- 07-2절에서 summary( ) 메서드의 출력 결과에 모델 이름과 층 이름이 책과 달라요.
- 07-2절에서 경사 하강법으로 훈련되지 않는 파라미터를 가진 층에는 어떤 것이 있나요?
- 07-3절에서 패션 MNIST의 테스트 세트는 왜 사용하지 않나요?
- 07-3절에서 드롭아웃 비율 0.3을 사용했는데요. 이 값은 어떻게 결정된 건가요?