1부 딥러닝의 기초 - 2장 시작하기 전에 : 신경망의 수학적 구성 요소

시작일	@2022년 7월 18일
링크	
종료일	@2022년 7월 19일

▼ 2장 시작하기 전에: 신경망의 수학적 구성 요소

텐서, 텐서 연산, 미분, 경사 하강법(gradient descent)등의 수학 개념과 친숙해져야 함

▼ 2.1 신경망과의 첫 만남

▼ MNIST 손글씨 숫자 분류 학습

▼ MNIST 데이터셋

- 손글씨 숫자 이미지 (28 x 28 픽셀)을 10개의 범주 (0~9)까지로 분류
- 6만 개의 훈련 이미지, 1만 개의 텍스트 이미지
- Numpy 배열 형태로 케라스에 이미 포함됨

▼ 코드

MNIST 데이터셋

- 훈련 세트(training set) - train_images, train_labels
- 테스트 세트(test set) - test_images, test_labels

```
# 케라스에서 MNIST 데이터셋 적재하기
from keras.datasets import mnist
(train_images,. train_labels), (test_images, test_labels) = mnist.load_data()

train_images.shape # (60,000, 28, 28)
len(train_labels) # 60,000
train_labels # array[5,0,4,,, 5,6,8], dtype = unit 8)

test_images.shape # (10,000, 28,28)
len(test_labels) # 10,000
test_labels # array[7,2,1,,,4,5,6], dtype = unit 8)
```

신경망 구조

- 데이터 처리 필터 : `층(layer)`

- 주어진 문제에 더 의미 있는 `표현(representation)`을 입력된 데이터로부터 추출
- `완전 연결(fully connected)된 신경망 층인 Dense층` 2개가 연속
- 마지막 층은 10개의 확률 점수가 들어 있는 배열을 반환하는 `소프트맥스(softmax) 층`
- 각 점수는 현재 숫자 이미지가 10개의 숫자 클래스 중 하나에 속할 확률

```
# 신경망 구조
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation = 'relu', input_shape = (28,28,))
network.add(layers.Dense(10, activation = 'softmax'))
```

신경망 훈련 (컴파일 단계)

- `손실 함수(loss function)`
  - 훈련 데이터, 성능 측정하는 방법
- `옵티마이저(optimizer)`
  - 입력된 데이터, 손실 함수 기반으로 네트워크를 업데이터
- `훈련과 테스트 과정을 모니터링할 지표`
  - ex.여기서는 정확도(정확히 분류된 이미지의 비율)

```
# 컴파일 단계
network.compile(optimizer= 'rmsprop',
                loss= 'categorical_crossentropy',
                metrics = ['accuracy'])
```

이미지 데이터 전처리

- 데이터를 0~1 사이로 스케일을 조정
- [0,255] 사이의 unit8 타입의 (60,000, 28, 28) 크기를 가진 배열로 저장
- 0과 1 사이의 값을 가지는 float32 타입의 (60,000, 28*28 크기인 배열로 바꿈)

```
# 이미지 데이터 준비하기
train_images = train_images.reshape((60000, 28*28))
train_images = train_images.astype('float32') / 255
```

```
test_images = test_images.reshape(10000, 28*28))
test_images = test_images.astype('float32') / 255
```

```
# 레이블 준비하기
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

신경망 훈련 (fit)

- fit 메서드 호출하여 훈련 데이터에 모델을 학습
- 훈련하는 동안 2개의 정보가 출력됨
- 훈련 데이터에 대한 네트워크의 손실, 정확도

```
# fit 메서드를 호출하여 훈련 데이터에 모델을 학습
network.fit(train_images, train_labels, epoch = 5, batch_size= 128)

# 출력 예시
# Epochs 1/5
# 60,000/60,000 [==============] - 1s 22us/step - loss : 0.2571 - acc - 0.9257
# Epochs 2/5
# 60,000/60,000 [==============] - 1s 12us/step - loss : 0.1027 - acc - 0.9695
```

결과 확인

- test_loss, test_acc

```
# 테스트 세트에서도 모델이 잘 작동하는지 확인
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc', test_acc)
```

▼ 2.2 신경망을 위한 데이터 표현

▼ 텐서 (tensor)

- 임의의 차원 개수를 가지는 다차원 넘파이 배열
- 숫자 데이터를 위한 컨테이너(container)
- ex. 2D 텐서: 행렬

▼ 2.2.1 스칼라 (0D 텐서)

- 하나의 숫자만 담고 있는 텐서를 스칼라(scala, 0차원 텐서, 0D테서)

- numpy 에서는 float32, float64 타입의 숫자가 스칼라 텐서
- ndim: 넘파이 배열의 축 개수를 확인
- 스칼라 텐서의 축 개수는 0 (ndim ==0)
- 텐서의 축 개수를 랭크(rank)라고도 부름

```
import numpy as np
x = np.array(12)
x # array(12)
x.ndim # 0
```

▼ 2.2.2 벡터 (1D 텐서)

- 숫자의 배열을 벡터(vector) 또는 1D 텐서라고 부름

```
x = np.array([12,3,6,14,7]) # 5개의 원소 - 5D 벡터 ,not 5D 텐서
x # array([12,3,6,14,7])
x.ndim # 1
```

▼ 2.2.3 행렬 (2D 텐서)

- 벡터의 배열이 행렬(maxtrix) 또는 2D 텐서
- 행렬은 2개의 축을 가짐 (행(row), 열(column))

```
x = np.array([5,78,2,34,0],
             [6,79,3,35,1],
             [7,89,4,36,2])

x.ndim # 2
# 첫 번째 축에 놓여 있는 원소를 행
# 두 번째 축에 놓여 있는 원소를 열
# x의 첫 번째 행은 [4,78,2,34,0]
# x의 첫 번째 열을 [5,6,7]
```

▼ 2.2.4 3D 텐서와 고차원 텐서

- 행렬들을 하나의 새로운 배열로 합치면 숫자가 채워진 직육면체 형태로 해석할 수 있는 3D 텐서
- 3D 텐서를 하나의 배열로 합치면 4D ~ , 딥러닝은 보통 4D까지 다룸
- 동영상 데이터는 5D까지 가기도 함

```
x = np.array([[1,2,3,4,5],
              [6,7,8,9,10],
              [11,12,13,14,15]],
             [[1,2,3,4,5],
              [6,7,8,9,10],
```

```
               [11,12,13,14,15]],
              [[1,2,3,4,5],
               [6,7,8,9,10],
               [11,12,13,14,15]])

x.ndim # 3
```

▼ 2.2.5 핵심 속성

- 텐서는 3개의 속성으로 정의됨
  - ▼ 축의 개수(랭크)
    - 3D 텐서, 3개의 축 / 행렬 , 2개의 축
    - ndim 속성에 저장됨
  - ▼ 크기(shape)
    - 0D : 배열/스칼라 () : 크기 없음
    - 1D: 벡터 (5,)
    - 2D: 행렬의 크기 (3,5)
    - 3D: 3D 텐서의 크기 (3,3,5)
  - ▼ 데이터 타입
    - dtype에 저장됨
    - 텐서에 포함된 데이터의 타입
    - float32, unit8, float64 등
    - 드물게 char 타입 사용/ 가별 길이 문자열은 지원 X
- MINST 속성 확인

```
# MNIST 데이터 불러오기
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# train_images 배열의 ndim 속성으로 축의 개수를 확인
print(train_images.ndim) # 3

# 배열의 크기
print(train_images.shape) # (60,000, 28, 28)

# dtype, 속성 확인
print(train_images.dtypes) # unit8 , 8비트 정슈형 3D 텐서

# 결론
# 28 x 28 크기의 정수 행렬 6만개 있는 배열
# 각 행렬을 하나의 흑백 이지미
# 행렬의 각 원소는 0~255 사이의 값을 가짐
```

```
# 5번째 이미지 출력
digit = train_images[4]

import matplotlib.pyplot as plt
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

▼ 2.2.6 넘파이로 텐서 조작하기

- train_images[i] : 첫 번째 축을 따라 특성 숫자 선택
- 슬라이싱(slicing)
  - 배열에 있는 특정 원소를 선택

```
# 11~101번째 숫자 선택하여 (90,28,28) 크기의 배열 만듦

# method 1
my_slice = train_images[10:100]
print(my_slice.shape) # (90,28,28)

# method 2
my_slice = train_images[10:100, :, :]
my_slice.shape # (90,28,28)

# method 3
my_slice = train_images[10:100, 0:28, 0:28]
my_slice.shape # (90,28,28)

# if, 이미지의 오른쪽 아래 14 x 14 픽셀 선택
my_slice= train_images[:, 14:, 14:]

# 음수 인덱스 ,정중앙에 위치한 7 x 7 픽셀
my_slice = train_images[: 7:-7, 7:-7]
```

▼ 2.2.7 배치 데이터

- 텐서의 첫 번째 축: 샘플 축(sample axis), 샘플 차원(sample dimension)

▼ `배치(Batch)`

- 딥러닝 전체 데이터를 작음 배치(batch)로 나누어 처리
- ex. MNIST, 배치 128로 만든다면
  - 1) batch = train_images[:128]
  - 2) batch = train_images[128:256]
  - n) batch = train_images[128*n : 128 * (n+1)]
- 배치 데이터 첫 번째 축 : `배치 축(batch axis)` / `배치 차원(batch dimension)`

▼ 2.2.8 텐서의 실제 사례

- `벡터 데이터`
  - (samples, features)
  - 2D 텐서
- `시계열 데이터/ 시퀀스(sequence) 데이터`
  - (samples, timesteps, features)
  - 3D 텐서
- `이미지`
  - (samples, height, width, channels)
  - (samples, channels, height, width)
  - 4D 텐서
- `동영상`
  - (samples, frames, height, width, channels)
  - (samples/frames, channels, height, width)
  - 5D 텐서

▼ 2.2.9 벡터 데이터

- (samples, features) , 2D 텐서
- 하나의 데이터 포인트가 벡터로 인코딩
- 배치 데이터는 2D 텐서로 인코딩됨
- 첫 번째 축은 `샘플 축`, 두 번째 축은 `특성 축 (feature axis)`

▼ ex1. 사람의 나이, 우편 번호, 소득으로 구성된 인구 통계 데이터

- 각 사람은 3개의 값을 가진 벡터
- 10만 명이 포함된 전체 데이터셋은 (100,000, 3) 크기의 텐서

▼ ex2. (공통 단어 2만 개로 만든 사전에서)각 단어가 등장한 횟수로 표현된 텍스트 문서 데이터셋

- 각 문서는 2만 개의 원소(사전에 있는 단어마다 하나의 원소에 대응합니다)를 가진 벡터
- 500개의 문서로 이루어진 전체 데이터셋 : (500, 20,000) 크기의 텐서

▼ 2.2.10 시계열 데이터 또는 시퀀스 데이터

- (samples, timesteps, features) , 3D 텐서
- 시간(연속된 순서)가 중요할 때, 시간 축 포함하여 3D 텐서로 저장됨

- 각 샘플은 벡터(2D 텐서)의 시퀀스로 인코딩 → 배치 데이터 3D 텐서로 인코딩

▼ 주식 가격 데이터셋

- 1분마다 현재 주식 가격
- 지난 1분 동안 최고 가격, 최소 가격을 저장
- 1분 마다 데이터 3D 벡터로 인코딩
- 하루 동안(390분)의 거래 (390,3) 크기의 2D 텐서로 인코딩
- 250일치 데이터: (250,390,3) 크기의 3D 텐서
- 1일치 데이터가 하나의 샘플

▼ 트윗 데이터셋

- 각 트윗은 128개의 알파벳으로 구성된 280개의 문자 시퀀스
- 각 문자가 128개의 크기인 이진 벡터로 인코딩 → 해당 문자만 1, 나머지 0
- 각 트윗은 (280,128) 크기의 2D 텐서로 인코딩
- 100만 개의 트윗으로 구성된 데이터셋 (100,000, 280, 128) 크기의 텐서에 저장

▼ 2.2.11 이미지 데이터

- (samples, height, width, channels) /(samples, channels, height, width) /4D 텐서
- 흑백 이미지: MNIST - 하나의 컬러 채널 - 2D 텐서
  - 컬러 채널의 차원 크기 : 1
  - 256 x 256 크기의 흑백 이미지에 대한 128개의 배치 : `(128, 256, 256,1)`
- 컬러이미지 : 높이, 너비, 컬러 채널의 3차원 - 3D 텐서
  - 256 x 256 크기의 컬러 이미지(3) 에 대한 128개의 배치 : `(128, 256, 256, 3)`
- 이미지 저장 방식
  - 채널 마지막 (channel-last) 방식
    - Tensorflow: (samples, height, width, channels = color_depth)
  - 채널 우선 (channel-first) 방식
    - 채널의 깊이를 배치 축 바로 뒤에 놓음
    - Theano : (samples, color_depth, height, width)
    - `흑백: (128,1,256,256)`
    - `컬러: (128,3,256,256)`

▼ 2.2.12 비디오 데이터

- (samples, frames, height, width, channels)/(samples/frames, channels, height, width)/5D 텐서

▼ 하나의 비디오

- 비디오: `(samples, frames, height, width, color_depths)`의 5D 텐서
- 연속된 프레임: `(frames, height, width, color_depth)`의 4D 텐서
- 각 프레임 = 하나의 컬러 이미지 - `(height, width, color_depth)`의 3D 텐서

▼ ex. 60 초짜리 144 x 256 유튜브 비디오 클립 / 초당 4프레임으로 샘플링 / 240프레임

- 이런 비디오 클립 4개를 가진 배치 : (4,240,144,256,3) : 106,168,320개의 값
- dtype: float 32 로 했다면 32 비트로 저장됨: 405MB
- 실생활의 비디오 float 32크기로 저장 X, 훨씬 용량 적고, 압축됨 (MPEG)

▼ 2.3 신경망의 톱니바퀴: 텐서 연산

```
# Dense 층을 쌓아 신경망 만듦
keras.layers.Dense(512, activation = 'relu')

# 2D 텐서를 입력으로 받고
# 입력 텐서의 새로운 표현인
# 또 다른 2D 텐서를 반환하는 함수
# (w: 2D 텐서 , b: 벡터)

output = relu(dot(W, input) + b)
  # 입력 텐서와 텐서 w 사이의 점곱(dot)
  # 점곱의 결과인 2D 텐서와 벡터 b사이의 덧셈(+)
  # 마지막으로 relu(렐루) 연산
  # relu(x) = max(x,0)
```

▼ 2.3.1 원소별 연산

- 원소별 연산(element-wise operation)
  - relu 함수 & 덧셈
  - 텐서에 있는 각 원소에 독립적으로 적용됨

```
#------------------------
# 함수를 통한 연산
#------------------------
# Relu 함수
def naive_relu(x) :
  assert len(x.shape) == 2 # x는 2D 넘파이 배열

  x = x.copy() # 입력 텐서 자체를 바꾸지 않도록 복사
```

```
  for i in range(x.shape[0]):
    for j in range(x.shape([1]):
      x[i.j] = max(x[i,j], 0)
  return x

# Add(덧셈) 함수
def naive_add(x,y):
  asset len(x.shape) == 2 # x는 2D 넘파이 배열
  asset x.shape == y.shape

  x = x.copy() # 입력 텐서 자체를 바꾸지 않도록 복사
  for i in range(x.shape[0]):
    for j in range(x.shape[1]):
      x[i,j] + = y[i,j]
  return x

# 함수 매개변수 수장 가능한 (mutable) 데이터 타입인 경우
  # 참조에 의한 호출(call by reference) 처럼
  # 배열 원본을 변경시키지 않기 위해 복사 필요

#------------------------
# 내장 함수를 통한 연산
#------------------------
import numpy as np
z = x + y # 원소별 덧셈
z = np.maximum(z, 0.) # 원소별 렐루 함수
```

▼ 2.3.2 브로드캐스팅

- 크기가 다른 두 텐서가 더해질 때
  - 작은 텐서가 큰 텐서에 크기에 맞게 `브로드캐스팅(broadcasting)됨`
  - 큰 텐서의 ndim 에 맞춰 작은 테서에 축이 추가됨
  - 작은 텐서가 새 축을 따라 큰 텐서의 크기에 맞도록 반복됨

```
# X shape:  (32,10)
# y shape: (10,)
# X + y ? -> Broadcasting

# Broadcasting 과정
  # y에 비어 있는 첫 번째 축을 추가 y: (1,10)
  # y를 이 축에 32번 반족 : 텐서 y: (32,10)
  # Y[i:, :] == y for i in range(0,32)
  # X + y

def naive_add_matrix_and_vector(x,y):
  assert len(x.shape) == 2 # x는 2D 넘파이 배열
  assert len(y.shape) == 1 # y는 넘파이 벡터
  assert x.shape[1] == y.shape[0]

  x = x.copy() # 입력 텐서 자체를 바꾸지 않도록 복사함
  for i in range(x.shape[0]):
    for j in range(x.shape[1]):
      x[i,j] += y[j]
  return x

# (a,b,,..n, n+1, ,,,m) 크기의 텐서
```

```
# (n, n+1, ,,,m) 크기의 텐서 사이 브로드캐스팅으로 원소별 연산 적용
# 브로드캐스팅은 a 부터 n-1까지 축에 자동으로 일어남
```

```
# 크기가 다른 두 텐서에 브로드캐스팅
  # 원소별 maximum 연산 적용

import numpy as np
x = np.random.random((64,3,32,10)) # x 는 (64,3,32,10) 크기의 랜덤 텐서
y = np.random.random((32,10)) # y는 (32,10) 크기의 랜덤 텐서
z = np.maximum(x,y) # 출력 z 크기는 x 와 동일하게 (64,3,32,10)
```

▼ 2.3.3 텐서 점곱

- 텐서 접곰(tensor product)
  - 원소별 곱셈 X
  - 점곱 연산(dot operation) - 입력 텐서의 원소들 결합

```
# 1. 벡터끼리의 점곱 연산
  # 2개의 벡터 x와 y 의 점곱
  # 두 벡터의 점곱: 스칼라가 됨
  # 원소 개수가 같은 벡터끼리 점곱이 가능
def naive_vector_dot(x,y):
  assert len(x.shape) == 1 # x와 y는 넘파이 벡터
  assert len(y.shape) == 1 # x와 y는 넘파이 벡터
  assert x.shape[0] == y.shape[0]

  z = 0.
  for i in range(x.shape([0]):
    z += x[i] * y[i]
  return z
```

```
# 2. 행렬과 벡터의 접곱 연산
  # x: 행렬, y: 벡터
  # y와 x 의 행 사이에 점곱 : 벡터를 반환
import numpy as np
def naive_matrix_vector_dot(x,y):
  assert len(x.shape) == 2 # x의 넘파이 행렬
  assert len(y.shape) == 1 # y는 넘파이 벡터
  assert x.shape[1] == y.shape[0] # x의 두 번째 차원 = y의 첫 번째 차원

  z = np.zeros(x.shape[0]) # x의 행과 같은 크기의 0 이 채워진 벡터
  for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        z[i] += x[i,j] * y[j]
  return z
```

```
# 3.행렬-벡터 점곱 & 벡터-벡터 점곱

def naive_matrix_vector_doc(x,y):
  z = np.zeros(x.shape[0])
```

```
  for i in range(x.shape[0]):
    z[i] = naive_vector_dot(x[i,:], y)
  return z

# 4. 단순 dot연산 구현
def naive_matrix_dot(x,y):
  assert len(x.shape) == 2 # x,y 는 넘파이 행렬
  assert len(y.shape) == 2 # x,y 는 넘파이 행렬
  assert x.shape[1] == y.shape[0] # x는 2번째 차원이 y의 첫 번째 차원과 같아야
 함

  z = np.zeros((x.shape[0], y.shape[1])) # 0이 채워진 특정 크기의 벡터를 만듦
  for i in range(x.shape[0]): # x의 행을 반복합니다
    for j in range(y.shape[1]): # y의 열을 반복합니다
      row_x = x[i, :]
      column_y = y[:,j]
      z[i,j] = naive_vector_dot(row_x, column_y)
  return z
```

- numpy, keras, theano, tensorflow 곱셈: * 연산자
- tensorflow dot 연산자

```
# tensorflow dot 연산자
tf.matmul(x,y)

# python (v 3.5> ) :
x @ y
```

- numpy, keras dot 연산자

```
# Numpy
import numpy as np
Z = np.dot(x,y)

# Keras
from keras import backend as K
K.dot(x,y)
```

▼ 2.3.4 탠서 크기 변환

- 텐서 크기 변환(tensor reshaping)
  - 텐서의 크기를 변환하는 것은 특정 크기에 맞게 열과 행을 재배열한다는 뜻
  - 크기가 변환된 텐서는 원래 텐서와 원소 개수는 동일
  - 자주 사용하는 크기 변환은 전차(transposition)
    - 행과 열을 바꾸는 것 x[i, :] → x[:, i]가 됨
  - `train_images = train_images.reshape((60000,28 * 28))`

```
x = np.array([[0., 1.],
              [2., 3.],
              [4., 5.]])

print(x.shape) # [3,2]

x = x.reshape((6,1))
x # 하면 6개 행인 array 생성됨

x = x.reshape((2,3))
x
```

```
x = np.zeros((300,20)) # 모두 0으로 채워진 (300,20) 크기의 행렬을 만듦
x = np.transpose(x)
print(x.shape) # (20,300)
```

▼ 2.3.5 텐서 연산의 기하학적 해석

▼ 2.3.6 딥러닝의 기하학적 해석

▼ 2.4 신경망의 엔진: 그래디언트 기반 최적화

`output = relu(dot(W, input) + b)`

- W와 b는 층의 속성처럼 볼 수 있음
- 가중치(weight) 또는 훈련되는 파라미터(trainable parameter)라고 부름
- 가중치에는 훈련 데이터를 신경망에 노출시켜서 학습된 정보가 담겨 있음
- 초기에는 가중치 행렬이 난수로 채워져 있음(무작위 초기화(random initialization) 단계라고 부름
- 피드백 신호에 기초하여 가중치가 점진적으로 조정됨 → 훈련 (training)
- 훈련 반복 루프(training loop) 가 발생
- 신경망에서 사용된 모든 연산은 미분 가능(differentiable)
- 네트워크 가중치에 대한 손실을 그래디언트(gradient)를 계산하는 것이 훨씬 더 좋은 방법

▼ 2.4.1 변화율이란?

- f(x + epsilon_x) = y + epsilon_y

▼ 2.4.2 텐서 연산의 변화율 : 그래디언트

- 그래디언트 : 텐서 연산의 변화율
- 입력 벡터 : x , 행렬: W, 타깃: y, 손실 함수: loss
  - W를 사용하여 타깃의 에측 y_pred 를 계산하고, y_pred와 y사이의 오차를 계산

- `y_pred = dot(W,x)`
- `loss_value = loss(y_pred, y)`

▼ 2.4.3 확률적 경사 하강법

- 미니 배치 확률적 경사 하강법(mini-batch stochastic gradient descent, 미니 배치 SGD)
- 확률적(stochastic) → 각 배치 데이터가 무작위로 선택된다는 의미
- 무작위(random) 하다는 것의 과학적 표현
  - 반복마다 하나의 샘플과 하나의 타깃을 뽑는 것 , 배치 SGD
- `SGD의 변종 /최적화 방법(optimization method)/ 옵티마이저`
  - `SGD, Adagrad, RMSProp`
  - 여러 변동즐에서 사용하는 모멘텀(momentum) 개념 중요
  - 모멘텀은 수렴 속도와 지역 최솟값을 모두 해결함

▼ 2.4.4. 변화율 연결: 역전파 알고리즘

- 텐서 연산 a,b,c / 가중치 행렬 W1, W2, W3 로 구성된 네트워크 f
- f(W1,W2,W3) = a(W1, b(W2, c(W3))
  - 연쇄 법칙(chain rule) 을 신경망의 그래디언트 계산에 적용
  - 역전파(Backpropagation) 알고리즘 (후진 모드 자동 미분)) 가능

▼ 2.5 첫 번째 예제 다시 살펴보기

```
# 입력 데이터
(train_iamges, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28)) # (60,000, 784) 크기 넘파이 배열
train_images = train_images.astype('float32') / 255  # 데이터 타입 : float32

test_images = test_images.reshape((10000, 28 * 28)) # (10,000, 784) 크기 넘파이 배열
test_images = ttest_images.astype('float32') / 255 # 데이터 타입 : float32

# 신경망 모형
network = models.Sequential()
network.add(layers.Dense(512, activation = 'relu', input_shape = (28,28, ))
network.add(layers.Dense(10, activation = 'softmax'))

# 컴파일
network.compile(optimizer = 'rmsprop',
                loss = 'categorical_crossentropy',
                metrics=  ['accuracy'])

# 훈련 방법
  # 네트워크가 128개 샘플씩 미니 배치로 훈련 데이터를 다섯 번 반복
```

```
  # 각 반복을 에포크(epoch)
network.fit(train_iamges, train_labels, epoch = 5, batch_size = 128)
```

▼ 2.6 요약

- 학습(learning)은 훈련 데이터 샘플과 타깃이 주어졌을 때 손실 함수를 최소화하는 모델 파라미터 조합을 찾는 것
- 손실과 옵티마이저!
    - 손실 : 훈련하는 동안 최소화해야 할 양, 해결하려는 문제의 성공을 측정하는데 사용함
    - 옵티마이저: 손실에 대한 그래디언트가 파라미터를 업데이트하는 정확한 방식을 정의함
        - RMSProp, 모멘텀을 사용한 SGD 등