텐서플로 딥러닝

최석재 *lingua@naver.com*

텐서플로 시작

텐서플로 시작

- 텐서플로 *Tensorflow*

  - 텐서플로는 머신러닝, 특히 딥러닝 프로그램을 쉽게 구현할 수 있도록 다양한 기능을 제공해주는 라이브러리
  - 구글이 내부 용도로 개발하였다가 2015년 11월 오픈소스로 공개
  - 텐서플로 자체는 C++로 작성되었으나 Python, Java, Go 등의 언어로 프로그래밍이 가능 (Python을 최우선으로 지원)
  - 딥러닝의 많은 연산량을 지원하기 위하여 CPU와 GPU를 모두 활용할 수 있도록 되어 있음
  - 윈도우, 맥, 리눅스, 안드로이드, iOS, 라즈베리 파이 등 다양한 OS에서 구동 가능

텐서플로 시작

- 케라스 *Keras*

  - 구글의 AI 연구원 프랑소와 숄레가 2007년 개발된 AI 딥러닝 라이브러리인 Theano를 쉽게 사용할 수 있도록 wrapper 프로그램으로 개발을 시작
  - 2015년 구글 텐서플로가 공개된 뒤 케라스는 Theano와 Tensorflow 중 하나를 선택할 수 있는 형태로 발전되었음
  - 2019년 텐서플로 2.0이 발표되면서 케라스는 tensorflow만을 지원하기 시작하였음
  - 2023년 Keras3가 발표되어 PyTorch에서도 사용할 수 있게 설계됨
  - 케라스는 Sequential, FunctionalAPI, Model Subclassing의 3가지 방법으로 딥러닝 모델을 작성할 수 있으며, 이 중 Sequential 모델은 작성하기가 간편하여 처음 딥러닝 모델을 만드는 경우에 적합함

텐서플로 시작

- Keras2 vs Keras3

  - 2023년 12월, Keras3가 발표되었다
  - Keras2는 텐서플로에 포함되어 있기 때문에 tf.keras 와 같이 사용해야 하지만
  - Keras3는 단순히 import keras 하여 사용하면 된다

  - 대부분 사용 방법은 바뀌지 않았으므로
  - 기존에 tf.keras로 사용하던 것을 keras 로 사용하면 된다
  - 그러나 아직 모든 API가 Keras3로 넘어가지 않았고, 일부 사용법이 다를 수도 있다
  - 여기서는 기본적으로 Keras2를 사용하면서 Keras3도 사용한다
  - https://keras.io/api/ 을 참조

텐서플로 시작

- 텐서플로 라이브러리 사용
  - 텐서플로를 임포트하고, 버전을 확인해본다

```
import tensorflow as tf
print(tf.__version__)
```

텐서플로 시작

- 첫번째 텐서플로 프로그램
  - 간단한 텐서플로 프로그램을 작성해본다

```
import tensorflow as tf

w = tf.Variable(2.0)
b = tf.Variable(0.7)

x = 1.5
y = w * x + b

print("y:", y)
```

신경망 네트워크를 통해서 찾을 값을 tensor로 만들어준다

y = ax +b 형태의 간단한 수식을 계산하는 프로그램을 만든다

```
y: tf.Tensor(3.7, shape=(), dtype=float32)
```

shape=() : 스칼라

텐서플로 시작

- 첫번째 텐서플로 프로그램
  - 결과 확인

```
if y == 3.7:
    print("Right")
```

```
Right
```

텐서플로를 이용한 회귀모델

텐서플로를 이용한 회귀모델

※ tensorflow 를 여러 번 사용하면
시스템 자원이 부족하여 더 이상 작동이 안될 수 있다
이러한 경우에는 Jupyter를 완전히 종료하고 다시 들어온다
(distributed_function 에러 또는 graph error 등)

- 텐서플로를 이용한 회귀분석 모델
  - 앞에서 작성한 프로그램을 조금 더 발전시켜 본다
  - 회귀분석에 필요한 가중치와 편향(절편) 값을 찾아야 한다

shape = [1] → [0.8272306]

shape = [2,1] → [[0.02256799]
[0.41108167]]

shape = [2,2] → [[0.5458367 0.8452722]
[0.09414423 0.01103127]]

- import tensorflow as tf
- import numpy as np

- x = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=tf.float32)
- y = tf.constant([3, 5, 5, 6, 7, 7, 8, 9, 9, 10], dtype=tf.float32)

딥러닝에서는 기본적으로 float32 데이터 타입을 사용한다

x, y 값은 변하지 않을 것이므로 tf.constant()로 만들어주고,
w, b 값은 학습 중 갱신되므로 tf.Variable()로 만들어준다

- w = tf.Variable(tf.random.normal(shape=[1]))
- b = tf.Variable(tf.random.normal(shape=[1]))

가중치 w와 편향 b는 하나씩만 필요하므로
shape=[1]

- def compute_loss():
  y_pred = w * x + b
  loss = tf.reduce_mean((y - y_pred)**2)
  return loss

reduce_mean: 입력값의 평균을 구하는 함수
전체 결과는 MSE와 동일

```
# reduce_mean()은 차원을 제거하고 평균을 구한다
x = tf.constant([[1, 3], [2, 6]])       # (2, 2)
print(tf.reduce_mean(x))                # 3
```

텐서플로를 이용한 회귀모델

최적화 함수로는 Adam을 사용
이 최적화 함수로 손실값 계산 함수와 최적화 대상인 w와 b를 넣어 모델을 훈련한다
훈련의 결과로는 loss를 최소화하는 w와 b의 최적값이 산출된다

rmsprop, adam은 기본 학습률(lr)이 0.001이고, sgd와 adagrad는 0.01이다

- 모델 훈련

```
0 loss: 0.3865098

100 loss: 0.1418192

200 loss: 0.1418182

300 loss: 0.14181809

400 loss: 0.14181809

500 loss: 0.14181809

600 loss: 0.14181809

700 loss: 0.14181809

800 loss: 0.14181809

900 loss: 0.14181809

final w: <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([0.7090909], dtype=float32)>
final b: <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([3.0000002], dtype=float32)>
```

- optimizer = tf.optimizers.Adam(learning_rate=0.07)

- for i in range(1000):
  optimizer.minimize(compute_loss, var_list=[w, b])

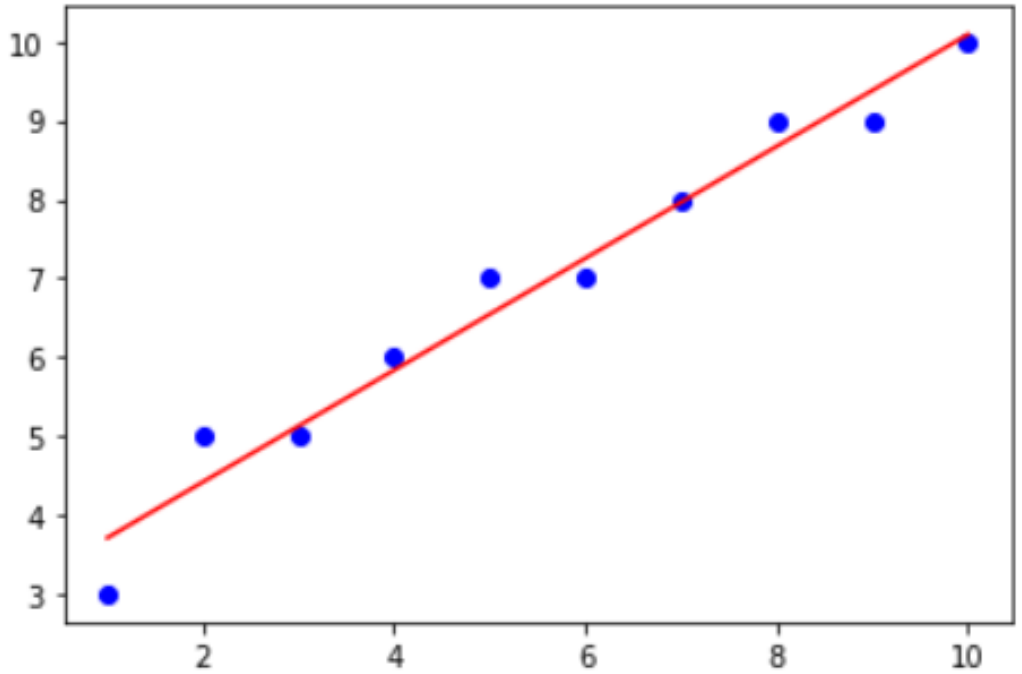
  if i % 100 == 0:
  print(i, "loss:", compute_loss().numpy(), "₩n")

- print("final w:", w)
- print("final b:", b)

※ optimizer.minimize는 손실값을 줄여나가며 손실함수에 사용된 변수 중 지정된 것(w, b)을 찾는다

텐서플로를 이용한 회귀모델

- 그래프로 확인
  - %matplotlib inline
  - import matplotlib.pyplot as plt
  - plt.plot(x, y, 'bo')
  - plt.plot(x, w * x + b, 'red')
  - plt.show()



그래프 결과 모델이 현재 데이터를 통해 w와 b 값을 잘 찾은 것으로 보인다

Keras Layer를 이용한 모델 생성

Keras Layer를 이용한 모델 생성

- Keras Layer를 이용한 모델 생성
  - 앞에서는 선형 회귀식에 필요한 가중치 w와 편향 b를 직접 찾았다
  - 이번에는 Keras의 딥러닝 네트워크를 이용하여 비선형 모델을 생성하는 방법을 알아본다
  - optimizer를 이용하여 예측 결과의 loss를 가장 적게 하는 최적 가중치를 찾는다

```
import tensorflow as tf
import numpy as np

x = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=tf.float32)
y = tf.constant([3, 5, 5, 6, 7, 7, 8, 9, 9, 10], dtype=tf.float32)

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units=10, activation='tanh', input_dim=1))
model.add(tf.keras.layers.Dense(units=1))
```

input_shape=(1,) 와 같이도 표현
은닉1층을 구성할 때 입력층의 형상을 지정

Keras Layer를 이용한 모델 생성

- 신경망 네트워크를 이용한 모델 생성

여기서는 훈련데이터 부족으로 학습률을 지정해주었지만,
일반적으로는 기본값을 사용한다

  - opt = tf.keras.optimizers.Adam(learning_rate=0.07)
  - model.compile(loss='mse', optimizer=opt)
  - model.summary()

.compile()은 모델을 훈련하기 전에 손실함수, 옵티마이저 등을 설정하기 위하여 사용한다
loss 값으로는 보통은 mse 나 mae를,
optimizer로는 adam 또는 rmsprop을 많이 사용한다

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 10)                20
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 11
=================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
_________________________________________________________________
```

Keras Layer를 이용한 모델 생성

- 훈련 결과 최종 손실값

  - model.fit(x, y, epochs=1000)

```
Epoch 994/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.0627
Epoch 995/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.0627
Epoch 996/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.0627
Epoch 997/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.0627
Epoch 998/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.0627
Epoch 999/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.0627
Epoch 1000/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.0626
<keras.src.callbacks.History at 0x7e501ce55510>
```

.fit()은 모델을 학습시키는 데 사용된다
데이터의 입력, 에포크 수 설정, 배치 사이즈 설정 등을 한다

※ 손실값이 매우 작게 나왔다
훈련 데이터가 적기 때문에 많은 epoch를 사용했다

Keras Layer를 이용한 모델 생성

- 훈련데이터에 대한 예측 결과

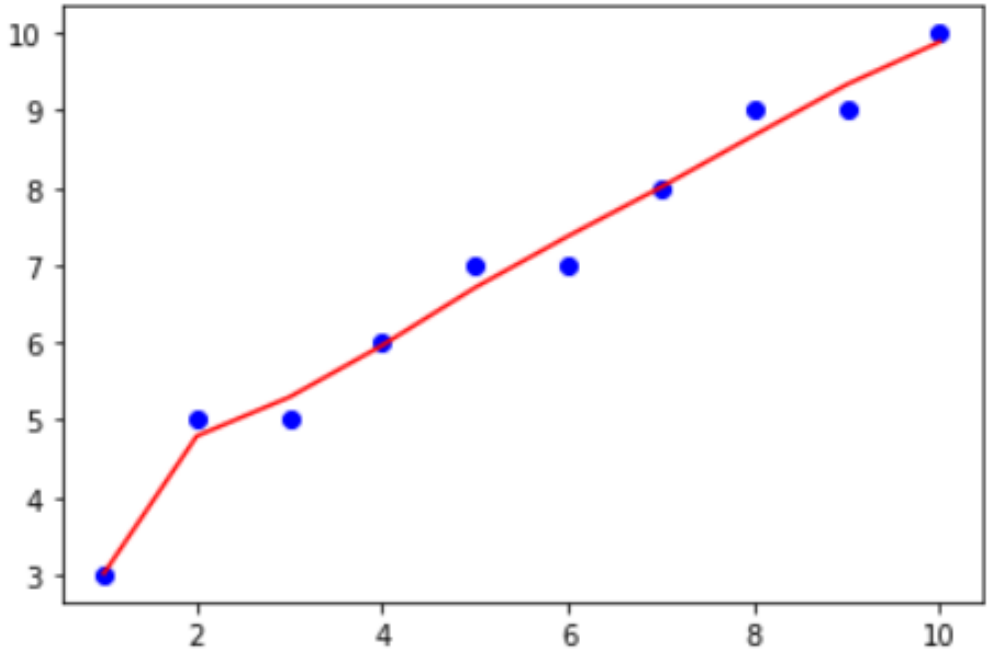
  - model.predict(x)

```
1/1 [==============================] - 0s 109ms/step
array([[3.0081658],
       [4.8011146],
       [5.3583026],
       [5.959837 ],
       [6.62553  ],
       [7.324118 ],
       [8.023972 ],
       [8.697393 ],
       [9.319647 ],
       [9.869003 ]], dtype=float32)
```

여기서는 훈련 데이터를 그대로 예측할 데이터로 넣었다

Keras Layer를 이용한 모델 생성

- 그래프 확인
  - %matplotlib inline
  - import matplotlib.pyplot as plt
  - plt.plot(x, y, 'bo')
  - plt.plot(x, model.predict(x), 'red')
  - plt.show()



더욱 완벽한 비선형 모델 생성!

GradientTape

tf.GradientTape()

tf.GradientTape()

- 일반적으로는 compile()과 fit() 메소드를 이용하여 미분을 계산하지만,
- tf.GradientTape()을 사용하여 이들을 대신할 수도 있다

- tf.GradientTape()을 이용하면 compile()에서 설정하는 손실함수, 옵티마이저 설정,
- fit()에서 설정하는 데이터 입력, 배치 사이즈 등을 수동으로 설정해주어야 한다

- tf.GradientTape()을 사용할 때의 장점은 세부적인 설정이 가능하다는 것
- 기능적인 면에서는 compile()과 fit()이 최적화되어 있고, 다양한 상황에 대응이 가능하다
- 또한 tf.GradientTape()은 중간 과정을 기록하므로 시간과 자원이 더 많이 소요된다

tf.GradientTape()

- import tensorflow as tf
- import numpy as np

- x = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=tf.float32)
- y = tf.constant([3, 5, 5, 6, 7, 7, 8, 9, 9, 10], dtype=tf.float32)

- model = tf.keras.Sequential()
- model.add(tf.keras.layers.Dense(units=10, activation='tanh', input_dim=1))
- model.add(tf.keras.layers.Dense(units=1))

tf.GradientTape()

- 손실함수와 옵티마이저를 인스턴스로 생성한다

손실 함수 인스턴스

- loss_fn = tf.keras.losses.MeanSquaredError()

옵티마이저 인스턴스

- optimizer = tf.keras.optimizers.Adam(learning_rate=0.07)

tf.GradientTape()

모델 훈련 단계와 테스트 단계에서 수행할 함수를 정의한다

```
def train_step(x_batch, y_batch):                                  # 모델 학습 함수
    with tf.GradientTape() as tape:
        predictions = model(x_batch, training=True)                # 예측값 계산
        loss = loss_fn(y_batch, predictions)                       # 예측값에 대한 손실값 계산
    gradients = tape.gradient(loss, model.trainable_variables)     # 가중치와 같은 훈련 가능 변수들의 그라디언트 계산 (미분)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))  # 그라디언트를 이용하여 변수 업데이트
    return loss                                                    # 배치 처리 후의 손실값 출력
```

```
def test_step(x_batch, y_batch):                                   # 테스트 데이터 성능 평가 함수
    predictions = model(x_batch, training=False)                   # 예측값 계산
    loss = loss_fn(y_batch, predictions)                           # 예측값에 대한 손실값 계산
    return loss
```

tf.GradientTape()

- 배치 사이즈를 지정하여 모델을 학습한다
- 원하는 내용을 출력할 수 있다

```
----------
epoch 996
one train data Loss: 0.12812221

----------
epoch 997
one train data Loss: 0.5193209

----------
epoch 998
one train data Loss: 0.73994553

----------
epoch 999
one train data Loss: 0.080098175

----------
epoch 1000
one train data Loss: 0.714967
```

- batch_size = 1 # 앞과 동일하게 배치 사이즈 1로 설정
- for epoch in range(1001): # 시간이 많이 소요되므로 GPU 또는 에포크 수를 줄인다

```
    print( " ₩n----------₩nepoch " , epoch)
    for i in range(0, len(x), batch_size):
        # 2차원 데이터가 되게 하고, 행의 개수는 배치 사이즈에 따라 달라지게 한다
        x_batch = tf.reshape(x[i:i+batch_size], (-1, 1))        # 행의 개수는 현 데이터 상황에 맞게 자동 결정되게 함
        y_batch = tf.reshape(y[i:i+batch_size], (-1, 1))
        train_loss = train_step(x_batch, y_batch)               # 모델 학습
    print("one train data Loss:", train_loss.numpy())           # 마지막 배치의 손실값
```

tf.GradientTape()

- 테스트 데이터에 대한 성능 평가를 수행한다
- 여기서는 훈련 데이터를 그대로 사용한다

- test_loss = test_step(tf.reshape(x, (-1, 1)), tf.reshape(y, (-1, 1)))
- print("Test Loss:", test_loss.numpy())

```
Test Loss: 0.075838186
```

tf.GradientTape()

- 예측을 수행한다

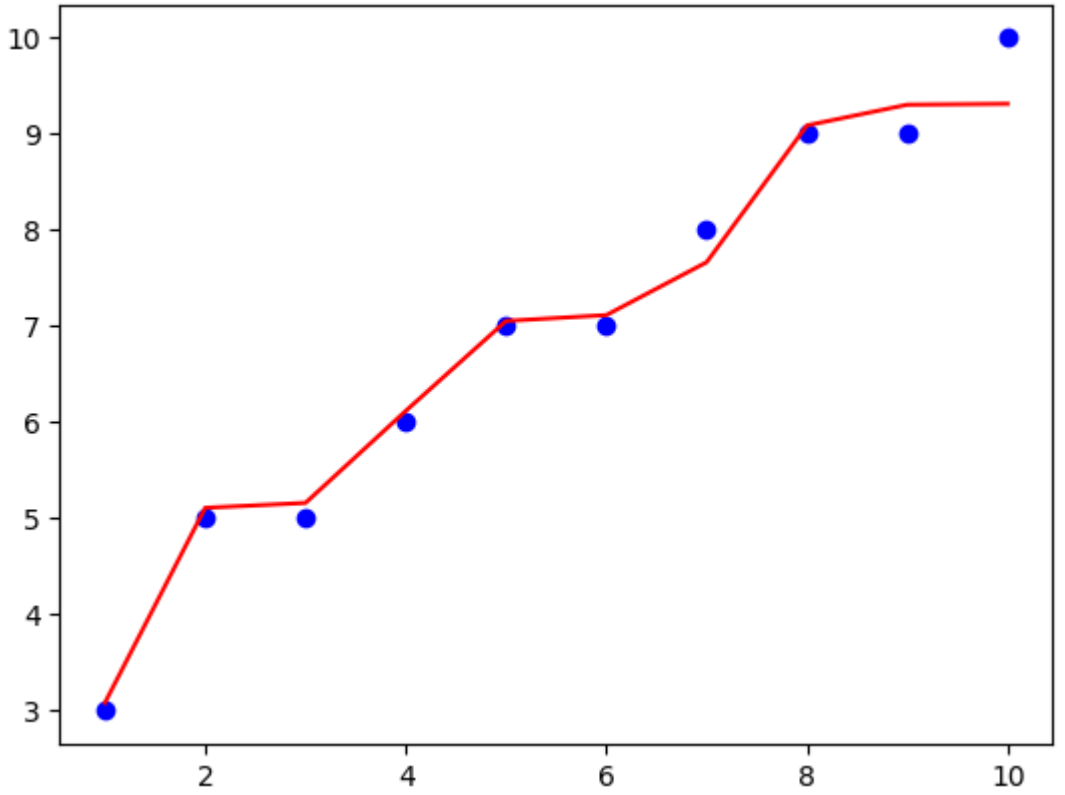
- model.predict(x)

```
1/1 [==============================] - 0s 59ms/step
array([[3.0834398],
       [5.112029 ],
       [5.1631465],
       [6.1207447],
       [7.056369 ],
       [7.1171083],
       [7.6677866],
       [9.090567 ],
       [9.305961 ],
       [9.315455 ]], dtype=float32)
```

tf.GradientTape()

- 성능 시각화

- %matplotlib inline
- import matplotlib.pyplot as plt
- plt.plot(x, y, 'bo')
- plt.plot(x, model.predict(x), 'red')
- plt.show()



Dataset

tf.data.Dataset

- 텐서플로는 기본 데이터 타입으로 텐서플로 텐서를 요구한다
- 일반적으로는 모델 입력에 넘파이 배열을 사용하지만, 내부적으로는 텐서로 변환된다
- 텐서플로 텐서는 텐서플로에서만 잘 동작하며, 넘파이 배열과는 다르게 변경 불가능하다
- 그러나 넘파이 배열은 CPU에서 작동하는 반면, 텐서플로 텐서는 GPU 및 TPU에서도
- 효율적으로 작동한다

- tf.data.Dataset은 텐서를 묶어서 대규모 데이터를 효율적으로 처리할 수 있게 설계되었다
- 대규모 데이터를 읽고, 전처리하며, 배치로 묶는 작업을 용이하게 한다
- 또한, 병렬 처리를 통해 데이터 로딩 속도를 높일 수 있다

Dataset 만들기

- tf.data.Dataset.from_tensor_slices()는 배열, 텐서, 리스트, 파일 경로 리스트를 입력으로 받아
- 개별 요소로 슬라이싱하여 텐서플로의 Dataset 타입으로 만든다

- import tensorflow as tf
- import numpy as np

- data = np.array([1, 2, 3, 4, 5]) # numpy 배열 생성

- dataset = tf.data.Dataset.from_tensor_slices(data) # TensorFlow 데이터셋 생성
- print(type(dataset))

데이터셋의 각 요소 출력
- for element in dataset:
    print(element.numpy())

Dataset 사용예 (1/2)

- 넘파이 배열을 Dataset으로 변환하여 모델에 주입하기까지의 과정을 본다

- import tensorflow as tf
- import numpy as np

예제 데이터 생성
- x_data = np.random.random((100, 32))
- y_data = np.random.randint(0, 10, (100,))

- dataset = tf.data.Dataset.from_tensor_slices((x_data, y_data)) # TensorFlow 데이터셋 생성

사용자 정의 전처리 함수
- def preprocess(x, y):
    x = tf.cast(x, tf.float32) # 독립변수는 float32형으로 타입 변환
    y = tf.cast(y, tf.int64) # 종속변수는 int64형으로 타입 변환. 이 외에 다양한 전처리를 수행할 수 있다
    return x, y

Dataset 사용예 (2/2)

```
dataset = dataset.map(preprocess)                    # 전처리 함수 적용
dataset = dataset.shuffle(buffer_size=100)           # 셔플
dataset = dataset.batch(32)                          # 배치로 묶기
dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
                                                     # 시스템 리소스에 따라 병렬 처리 수준을 자동으로 조절하고(AUTOTUNE),
                                                       prefetch() 함수로 다음 배치를 로드하고 준비한다
# 모델 설계
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(64, activation='relu', input_shape=(32,)),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(dataset, epochs=10)                        # 모델 훈련. tf.data.Dataset 객체를 모델의 fit 메서드에 바로 전달
```