Week3

CH12 - 텐서플로를 사용한 사용자 정의 모델과 훈련

저수준 API - 모델을 세부적으로 완전하게 제어하고 싶을 때

12.1 텐서플로 훑어보기

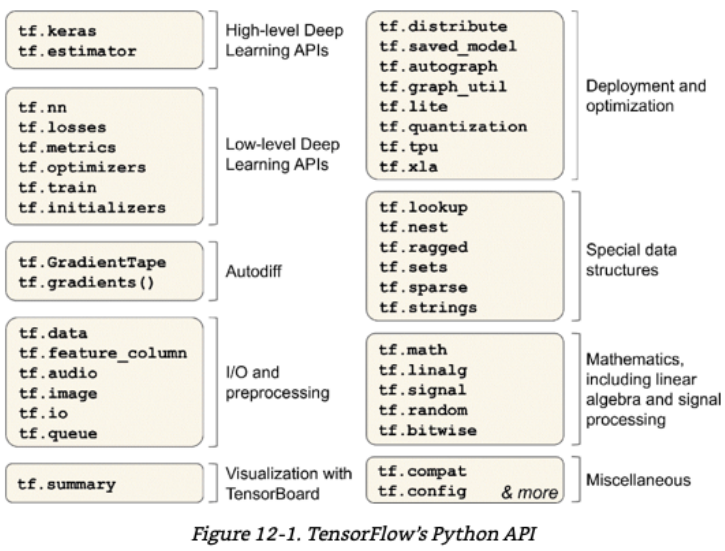
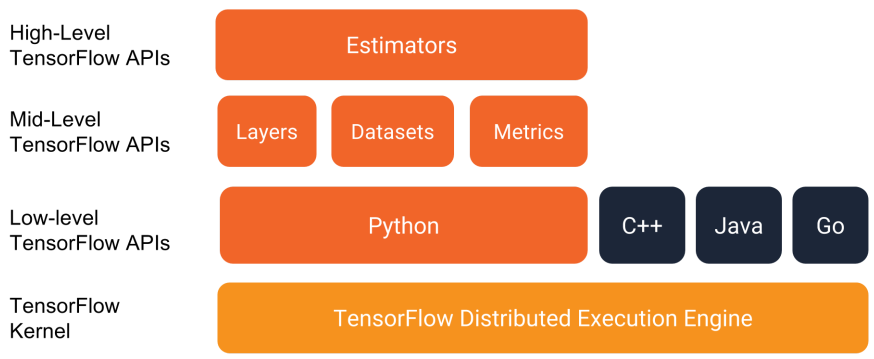


Figure 12-1. TensorFlow's Python API



12.2 넘파이처럼 텐서플로 사용하기

텐서는 넘파이 `ndarray`와 비슷

- 다차원 배열
- 하지만 스칼라 값도 가질 수 있음

12.2.1 텐서와 연산

```
>>> tf.constant([[1.,2.,3.],[4.,5.,6.]]) #행렬
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>

>>> tf.constant(42) #스칼라
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=42>

#크기와 데이터 타입을 가짐
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>

>>> t.shape
TensorShape([2, 3])

>>> t.dtype
tf.float32

#인덱스 참조
>>> t[:, 1:]
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>

>>> t[..., 1, tf.newaxis]
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>

#연산 가능
>>> t + 10 #tf.add(t, 10)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>

>>>tf.square(t)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>

>>>t @ tf.transpose(t) # @는 행렬 곱셈 연산자로 tf.matmul()과 동일
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

✔️ 케라스 저수준 API

`keras.backend` : 자체적인 저수준 API

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10

<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

12.2.2 텐서와 넘파이

텐서 - 넘파이

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>

>>>t.numpy() #np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)

>>>tf.square(a)
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([ 4., 16., 25.])>

>>>np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

12.2.3 타입 변환

타입 변환은 성능을 크게 감소시킬 수 있음

텐서플로는 — 어떠한 타입 변환도 자동으로 수행 X

```
#실수 텐서와 정수 텐서 연산 불가
try:
    tf.constant(2.0) + tf.constant(40)
except tf.errors.InvalidArgumentError as ex:
    print(ex)

#32비트 실수와 64비트 실수 연산 불가
try:
    tf.constant(2.0) + tf.constant(40., dtype=tf.float64)
except tf.errors.InvalidArgumentError as ex:
    print(ex)

>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32) #tf.cast() 사용
<tf.Tensor: shape=(), dtype=float32, numpy=42.0>
```

12.2.4 변수

`tf.Variable` : 변경되어야 하는 것들을 변경

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v.assign(2 * v)
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>

>>> v[0, 1].assign(42)
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  6.],
       [ 8., 10., 12.]], dtype=float32)>

>>> v[:, 2].assign([0., 1.])
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  0.],
       [ 8., 10.,  1.]], dtype=float32)>

>>> v.scatter_nd_update(indices=[[0, 0], [1, 2]],
                        updates=[100., 200.])
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[100.,  42.,   0.],
       [  8.,  10., 200.]], dtype=float32)>
```

12.2.5 다른 데이터 구조

- 희소 텐서
- 텐서 배열
- 레그드 텐서
- 문자열 텐서
- 집합
- 큐

12.3 사용자 정의 모델과 훈련 알고리즘

12.3.1 사용자 정의 손실 함수

비효율적인 잡음 데이터 → **후버 손실 사용**

```
#공식 케라스 API에서느 지원X
#keras.losses.Huber 클래스 이용하면 됨
#없다치고 구현
>>> def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < 1
        squared_loss = tf.square(error) / 2
        linear_loss  = tf.abs(error) - 0.5
        return tf.where(is_small_error, squared_loss, linear_loss)

>>> model.compile(loss=huber_fn, optimizer="nadam", metrics=["mae"])
>>> model.fit(X_train_scaled, y_train, epochs=2,
          validation_data=(X_valid_scaled, y_valid))
Epoch 1/2
```

```
363/363 [==============================] - 3s 4ms/step - loss: 0.6235 - mae: 0.9953 - val_loss: 0.2862 - val_mae: 0.5866
Epoch 2/2
363/363 [==============================] - 2s 4ms/step - loss: 0.2197 - mae: 0.5177 - val_loss: 0.2382 - val_mae: 0.5281
<keras.callbacks.History at 0x7fc42615d910>
```

12.3.2 사용자 정의 요소를 자긴 모델을 저장하고 로드하기

사용자 정의 개체를 포함한 모델 로드 시, 그 이름과 객체를 매핑해 전달해주어야 함

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn}) #매핑

#매개변수를 받을 수 있는 함수 생성
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam", metrics=["mae"])

#threshold 값을 저장 X - 따로 지정
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})

#자신만의 손실함수 만들기
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}

model.compile(loss=HuberLoss(2.), optimizer="nadam", metrics=["mae"])

model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

12.3.3 활성화 함수, 초기화, 규제, 제한을 커스터마이징하기

```
def my_softplus(z): # tf.nn.softplus(z) 값을 반환합니다
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # tf.nn.relu(weights) 값을 반환합니다
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

#사용자 정의 함수 적용
layer = keras.layers.Dense(1, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)

#모델과 함께 저장해야 할 파라미터가 있을 때
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

→ 손실, 층 모델: call( ) 메서드 구현

→ 규제, 초기화, 제한: __call__( ) 메서드 구현

12.3.4 사용자 정의 지표

손실과 지표가 개념적으로 다른 것은 아니지만 차이가 있음

- 손실
  - 미분 가능해야 함
  - 그레디언트가 모든 곳에서 0이 아니어야 함
  - 사람이 쉽게 이해 못해도 괜찮음
- 지표
  - 미분 불가능해도 상관없음
  - 그레디언트가 0이어도 상관없음
  - 사람이 쉽게 이해할 수 있어야 함

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
#모델 전체의 진짜 정밀도를 계산
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: shape=(), dtype=float32, numpy=0.8> #정밀도 80%

>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: shape=(), dtype=float32, numpy=0.5> #정밀도 50%
```

스트리밍 지표: 배치마다 점진적으로 업데이트 되는 지표

```
#현재 정밀도 확인
>>> precision.result()
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>

#진짜 양성과 거짓 양성 변수 확인
>>> precision.variables
[<tf.Variable 'true_positives:0' shape=(1,) dtype=float32, numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' shape=(1,) dtype=float32, numpy=array([4.], dtype=float32)>]

#변수 초기화
precision.reset_states()


#스트리밍 지표 만들기
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # 기본 매개변수 처리 (예를 들면, dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

12.3.5 사용자 정의층

- 텐서플로에 없는 특이한 층이 필요하거나
- 동일한 층 블럭이 반복되는 네트워크를 만드는 경우 → 사용자 정의층 생성

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)
```

```
    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

```
#여러 가지 입력을 받는 층 생성

class MyMultiLayer(keras.layers.Layer):
    def call(self, X): #모든 입력이 포함된 튜플을 매개변수로 전달
        X1, X2 = X
        print("X1.shape: ", X1.shape ," X2.shape: ", X2.shape) # 사용자 정의 층 디버깅
        return X1 + X2, X1 * X2

    def compute_output_shape(self, batch_input_shape): #이 매개변수도 각 입력의 배치 크기를 담은 튜플
        b1, b2 = batch_input_shape
        return [b1, b2]
```

→ 함수형 API나 서브클래싱 API에서만 사용 가능

→ 시퀀셜 API 사용 불가

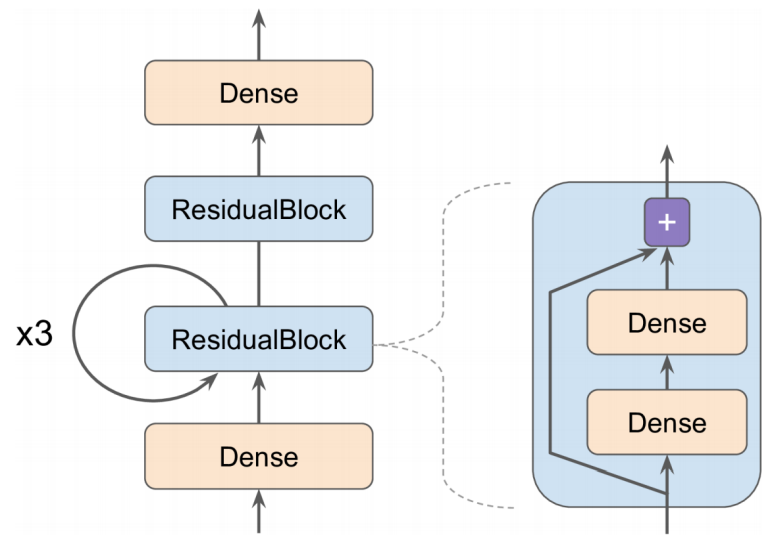
```
#훈련과 테스트에서 다르게 동작하는 층

class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

12.3.6 사용자 정의 모델



스킵 연결이 있는 사용자 정의 잔차 블록 층을 가진 모델 예시

잔차 블록: 두 개의 완전 연결 층과 스킵 연결로 구성

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                          kernel_initializer="he_normal")
                       for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z

#서브클래싱 API를 사용해 모델 정의

class ResidualRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                          kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

거의 모든 모델이 위와 같은 방식으로 만들어짐

12.3.7 모델 구성 요소에 기반한 손실과 지표

모델 내부 상황을 모니터링할 때 유용

재구성 손실: 재구성과 입력 사이 평균 오차

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)
        self.reconstruction_mean = keras.metrics.Mean(name="reconstruction_error")

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        #super().build(batch_input_shape)

    def call(self, inputs, training=None):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        self.recon_loss = 0.05 * tf.reduce_mean(tf.square(reconstruction - inputs))

        if training:
           result = self.reconstruction_mean(recon_loss)
           self.add_metric(result)
        return self.out(Z)

    def train_step(self, data):
        x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x)
            loss = self.compiled_loss(y, y_pred, regularization_losses=[self.recon_loss])

        gradients = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(gradients, self.trainable_variables))

        return {m.name: m.result() for m in self.metrics}
```

12.3.8 자동 미분을 사용하여 그레디언트 계산하기

드물지만 훈련 반복 자체를 제어해야 하는 경우가 있음

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2

w1, w2 = 5, 3
eps = 1e-6
(f(w1 + eps, w2) - f(w1, w2)) / eps

1(f(w1, w2 + eps) - f(w1, w2)) / eps
.
.
.
중략
```

12.3.9 사용자 정의 훈련 반복

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                       kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])

def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]

def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
                          for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - ".format(iteration, total) + metrics,
          end=end)
```

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]

#사용자 정의 훈련 반복 생성

for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        for variable in model.variables:
            if variable.constraint is not None:
                variable.assign(variable.constraint(variable))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

12.4 텐서플로 함수와 그래프

원본 파이썬 함수보다 빠름

→ 파이썬 함수를 텐서플로 함수로 변환 !

12.4.1 오토그래프와 트레이싱

텐서플로의 그래프 생성과정

1. 파이썬 함수의 소스 코드 분석 → 제어문 찾기: **오토그래프**
2. 찾은 제어문을 텐서플로 연산으로 변환해 업그레이드
3. 업그레이드된 함수 호출(매개변수 대신 **심볼릭 텐서** 전달)
   심볼릭 텐서: 실제 값은 없고 이름, 데이터 타입, 크기만 갖음
4. 그래프 모드로 실행
   → **즉시 실행 모드**: 텐서플로 연산이 해당 연산을 의미, 텐서를 출력하기 위해 그래프에 노드 추가
   ⇒ **트레이싱...?!**

12.4.2 텐서플로 함수 사용 방법

- 다른 라이브러리를 호출하면 트레이싱 과정에서 실행된다.
- 다른 파이썬 함수나 텐서플로 함수를 호출할 수 있다.
- 함수에서 텐서플로 변수를 만든다면 처음 호출될 때만 수행되어야 한다.
- 파이썬 함수의 소스 코드는 텐서플로에서 사용 가능해야 한다.
- 텐서플로는 텐서나 데이터 셋을 순회하는 for문만 감지한다.
- 성능면에서는 반복문보다 가능한 벡터화된 구현을 사용하는 것이 좋다.

12.5 연습문제