


TAVE 서기

서기 내용			
서 기 일 자	21.10.01	서기	이아현
주 제	텐서플로를 사용자 정의 모델과 훈련		
시 간	20:30~22:30	장소	Zoom 미팅
스 터 디 인 원	<p>고성호, 권기호, 서가을, 이아현 : 시작</p>  <p>고성호, 권기호, 서가을, 이아현 : 종료</p>		



내용

[목차]

Chapter 12. 텐서플로를 사용한 사용자 정의 모델과 훈련

12.1 텐서플로 훑어보기

12.2 넘파이처럼 텐서플로 사용하기

12.3 사용자 정의 모델과 훈련 알고리즘

12.4 텐서플로 함수와 그래프

12.5 연습문제

배운 내용

[배운내용]

12.1 텐서플로 훑어보기

About TensorFlow

- 수치 계산과 대규모 머신러닝을 위한 오픈소스 라이브러리 -> 데이터 획득, 모델 학습, 예측, 미래 결과 정제와 같은 과정을 쉽게 해줌
- 핵심 구조는 넘파이와 매우 유사, but GPU 지원
- 분산 컴퓨팅 지원
- JIT 컴파일러를 포함
- 계산 그래프는 플랫폼에 독립적인 포맷으로 내보낼 수 있음
- 자동 미분, RMSProp, Nadam 같은 고성능 옵티마이저 제공

파이썬 코드

케라스

데이터 API

저수준 파이썬 API (ops)

C++

Go

...

단일 / 분산 실행 엔진

CPU 커널

GPU 커널

TPU 커널

...

- 텐서플로 파이썬 API

TensorFlow python API

고수준 입려닝 API

tf.keras
tf.estimator

자동 미분

tf.GradientTape
tf.gradients()

선형 대수와 신호 처리를 포함한 수학 연산

tf.math
tf.linalg
tf.signal
tf.random
tf.bitwise

배포와 최적화

tf.distribute
tf.saved_model
tf.autograph
tf.graph_util
tf.lite
tf.quantization
tf.tpu
tf.xla

저수준 입려닝 API

tf.nn
tf.losses
tf.metrics
tf.optimizers
tf.train
tf.initializers

입출력과 전처리

tf.data
tf.feature_column
tf.audio
tf.image
tf.io
tf.queue

특수한 데이터 구조

tf.lookup
tf.nest
tf.ragged
tf.sets
tf.sparse
tf.strings

텐서보드 시각화

tf.summary

그 외

tf.compat
tf.config & more

- TensorBoard : TensorFlow 시각화 도구

- 손실 및 정확도와 같은 측정항목 추적 및 시각화
- 모델 그래프(작업 및 레이어) 시각화
- 시간의 경과에 따라 달라지는 가중치, 편향, 기타 텐서의 히스토그램 확인
- 저차원 공간에 임베딩 투영
- 이미지, 텍스트, 오디오 데이터 표시
- TensorFlow 프로그램 프로파일링
- 그 외 다양한 도구

- TFX(TensorFlow Extended) : 데이터 시각화, 전처리, 모델 분석, 서빙 등이 포함됨

- TensorFlow를 기반으로 하는 Google의 프로덕션 규모 머신러닝 (ML) 플랫폼
- 머신러닝 시스템을 정의, 시작, 모니터링하는 데 공통으로 필요한 구성요소를 통합할 수 있는 구성 프레임워크 및 공유 라이브러리를 제공

12.2 넘파이처럼 텐서플로 사용하기

12.2.1 텐서와 연산

- tf.constant() 함수로 텐서를 만들 수 있음

```
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>
```

- Tf.tensor는 크기와 데이터 타입(dtype)을 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 참조도 넘파이와 매우 비슷하게 작동함

```
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)>
```

- 모든 종류의 텐서 연산이 가능함

```
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)

<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

- 케라스로 구현한 연산으로 텐서플로에서 제공하는 함수의 일부분만 지원함

```
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

- 넘파이 배열에서 텐서플로 연산 적용이 가능함
- 텐서에 넘파이 연산 적용이 가능함

```
a = np.array([2., 4., 5.])
tf.constant(a)

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

t.numpy()

array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)

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 있지만 자동변환 시 사용자가 눈치채지 못할 수 있음=> 텐서플로는 어떤 타입 변환도 자동으로 수행하지 않음

```
[26] tf.constant(2.) + tf.constant(40)
```

```
InvalidArgumentError                                Traceback (most recent call last)
<ipython-input-26-1026a7e18cf4> in <module>()
----> 1 tf.constant(2.) + tf.constant(40)

~
~
~
3 frames
/usr/local/lib/python3.7/dist-packages/six.py in raise_from(value, from_value)

InvalidArgumentError: cannot compute AddV2 as input #1(zero-based) was expected to be a float tensor but is a int32
tensor [Op:AddV2] name: add/
```

- 타입 변환이 필요할 때 : `tf.cast()` 사용

```
t2 = tf.constant(40., dtype=tf.float64)
tf.constant(2.0) + tf.cast(t2, tf.float32)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=42.0>
```

12.2.4 변수

- `tf.Tensor`는 텐서의 내용을 바꿀 수 없음 => `tf.Variable`이 필요함
- `assign()` 메서드를 사용하여 변수값을 바꿀 수 있음

```
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)>
```

12.2.5 다른 데이터 구조

- 희소 텐서 : 대부분 0으로 채워진 텐서를 효율적으로 나타냄
- 텐서 배열 : 텐서의 리스트
- 래그드 텐서 : 리스트의 리스트, `tf.nn.nested` 패키지 사용
- 문자열 텐서 : `tf.string` 타입의 텐서
- 집합 : 일반적인 텐서로 나타냄, `tf.nn.nested` 패키지 연산을 사용
- 큐 : 단계별로 텐서를 저장, 텐서플로는 여러 종류의 큐를 제공함, `tf.queue` 패키지에 포함

12.3 사용자 정의 모델과 훈련 알고리즘

12.3.1 사용자 정의 손실 함수

- MSE는 이상치에 관대해서 훈련이 수렴되기까지 시간이 걸림, 모델이 정밀하게 훈련되지 않음 -> 후버 손실

```
'''
MSE는 이상치에 관대해서 훈련이 수렴되기까지 시간이 걸림. 그리고 정확하지 않음

MSE 대신 후버 Huber 로스를 사용하면 좋음
'''

def huber_fn(y_true, y_pred):
    error = y_true - y_pred # 실제값 - 예측값
    is_small_error = tf.abs(error) < 1 # -1 < error < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5

    # tf.where
    # tf.where(bool type 텐서, True일 때 출력값, False일 때 출력값)
    return tf.where(is_small_error, squared_loss, linear_loss)
```

12.3.2 사용자 정의 요소를 가진 모델을 저장하고 로드하기

- 모델을 로드할 때는 함수 이름과 실제 함수를 매핑한 딕셔너리를 전달

```
# 모델을 로드할때는 함수 이름과 실제 함수를 매핑한 딕셔너리를 전달해야함
model.save("my_model_with_a_custom_loss.h5")

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

model.fit(X_train_scaled, y_train, epochs=2,
          validation_data=(X_valid_scaled, y_valid))
```

- 모델을 저장할 때 threshold값은 저장되지 않음 -> 모델을 로드할 때 이를 지정해야함 -> get_config()메서드를 구현하여 해결할 수 있음
- 모델을 저장할 때 케라스는 손실 객체의 get_config() 메서드를 호출하여 반환된 설정을 HDF5에 JSON 형태로 저장

```

class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        # super?
        # https://velog.io/@gwiko/%ED%81%B4%EB%9E%98%EC%A4-%EC%83%B1%EC%B0-%EB%B0%B9-super-%ED%95%A0%EC%B0%B3-%EC%97%A0%EL
        # **kwargs ?
        # keyword argument의 줄임말
        # https://velog.io/@h8853/Python-args%EC%B0-kwags
        """
        def name2(**kwargs):
            print(kwargs)

        name2(name1="홍길동", name2="이순신")

        >> {'name1': '홍길동', 'name2': '이순신'}
        """
        super().__init__(**kwargs)

    # 손실의 손실을 반환
    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

        return tf.where(is_small_error, squared_loss, linear_loss)

    # 하이퍼파라미터 이름과 값이 매핑된 딕셔너리를 반환
    def get_config(self):
        base_config = super().get_config()
        # print(**base_config, "threshold": self.threshold)

        # {'reduction': 'auto', 'name': None, 'threshold': 2.0}
        # reduction algorithm?
        # http://sanghyukohun.github.io/80/
        # 손실함수의 name과 개별 손실들 모으기 위해 사용할 reduction 알고리즘 (차분)
        return {**base_config, "threshold": self.threshold}

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="selu", kernel_initializer="lecun_normal",
        input_shape=input_shape),
    keras.layers.Dense(1),
])

# 모델을 컴파일 할때, 클래스의 인스턴스 사용
model.compile(loss=HuberLoss(2.), optimizer="nadam", metrics=["mae"])

model.fit(X_train_scaled, y_train, epochs=2,
        validation_data=(X_valid_scaled, y_valid))

# save
# 모델을 저장할때 이것과도 함께 저장됨
model.save("my_model_with_a_custom_loss_class.h5")

Epoch 1/2
363/363 [=====] - 1s 2ms/step - loss: 0.9111 - mae: 1.0419 - val_loss: 0.3128 - val_mae: 0.5701
Epoch 2/2
363/363 [=====] - 1s 2ms/step - loss: 0.2562 - mae: 0.5281 - val_loss: 0.2399 - val_mae: 0.5010
{'reduction': 'auto', 'name': None, 'threshold': 2.0}

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

model.fit(X_train_scaled, y_train, epochs=2,
        validation_data=(X_valid_scaled, y_valid))

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

Epoch 1/2
363/363 [=====] - 1s 2ms/step - loss: 0.2334 - mae: 0.5045 - val_loss: 0.2569 - val_mae: 0.5012
Epoch 2/2
363/363 [=====] - 1s 2ms/step - loss: 0.2275 - mae: 0.4973 - val_loss: 0.2285 - val_mae: 0.4832
2.0

```

12.3.3 활성화 함수, 초기화, 규제, 제한을 커스터마이징하기

- 아래 그림과 같이 사용자 정의 활성화 함수를 만들 수 있음

```

"""
사용자 정의 활성화 함수
"""

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)

```

12.3.4 사용자 정의 지표

- 손실 : 모델을 훈련하기 위해 경사 하강법에서 사용하므로 미분 가능해

야 하고 그레디언트가 0이 아니어야 함

- 지표 : 모델을 평가할 때 사용
- Precision : 전체 정답의 개수 중에 모델이 맞춘 개수
- 앞서 만든 후버 손실 함수는 지표로도 사용해도 잘 동작함

12.3.5 사용자 정의 층

- 층 B층 C층 반복 (ABCBABCABC) -> ABC = D -> (DDD)
- 사용자 정의 층을 만드는 과정

```
# keras.layers.Flatten()나 keras.layers.ReLU 같은 층은 가중치가 없음
# 이런 층들을 lambda로 감쌀 수 있음
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
exponential_layer([-1., 0., 1.])

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=input_shape),
    keras.layers.Dense(1),
    exponential_layer
])
model.compile(loss="mse", optimizer="sgd")
model.fit(X_train_scaled, y_train, epochs=5,
        validation_data=(X_valid_scaled, y_valid))
model.evaluate(X_test_scaled, y_test)

Epoch 1/5
363/363 [=====] - 1s 2ms/step - loss: 1.0631 - val_loss: 0.4457
Epoch 2/5
363/363 [=====] - 1s 1ms/step - loss: 0.4562 - val_loss: 0.3798
Epoch 3/5
363/363 [=====] - 1s 2ms/step - loss: 0.4029 - val_loss: 0.3548
Epoch 4/5
363/363 [=====] - 1s 1ms/step - loss: 0.3851 - val_loss: 0.3464
Epoch 5/5
363/363 [=====] - 1s 1ms/step - loss: 0.3708 - val_loss: 0.3449
162/162 [=====] - 0s 1ms/step - loss: 0.3586
0.3586341142654419
```

- 생성자는 모든 하이퍼파라미터를 매개변수로 받음
- keras Dense 층의 간소화 한 버전 (사용자 커스터마이징)
- build() : 가중치마다 add_weight() 메서드를 호출하여 층의 변수를 만드는 것
- call() : 이 층의 필요한 연산을 수행
- compute_output_shape() : 이 층의 출력 크기를 반환함
- get_config() : keras.activations.serialize()를 사용하여 활성화 함수의 전체 설정을 저장

```

# keras Dense층을 과소파한 버전 (사용자 커스터마이징)
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    # build의 역할
    # 가중치마다 add_weight() 메서드를 호출하여 층의 변수를 만드는 것
    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

    # 이 층에서 필요한 연산을 수행
    def call(self, X):
        # @: st operator
        # 곱셈과 합을 사용함
        """
        import numpy

        x = numpy.ones(5) # array([1., 1., 1.])
        y = numpy.ones(5) # array([1., 1., 1.])
        z = x @ y

        print(z) // 5
        """
        # 여기서 입력 X와 층의 커널의 곱셈을 하기 위해 사용함
        return self.activation(X @ self.kernel + self.bias)

    # 이 층의 출력 크기를 반환함
    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()

        # keras.activations.serialize(): 활성화 함수의 전체 설정을 저장
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    MyDense(80, activation="relu", input_shape=input_shape),
    MyDense(1)
])

model.compile(loss="mse", optimizer="nadam")
model.fit(X_train_scaled, y_train, epochs=2,
        validation_data=(X_valid_scaled, y_valid))
model.evaluate(X_test_scaled, y_test)

model.save("my_model_with_a_custom_layer.h5")

model = keras.models.load_model("my_model_with_a_custom_layer.h5",
                                custom_objects={"MyDense": MyDense})

Epoch 1/2
888/888 [-----] - 1s 2ms/step - loss: 2.2583 - val_loss: 0.9472
Epoch 2/2
888/888 [-----] - 1s 2ms/step - loss: 0.6485 - val_loss: 0.6219

```

- 여러 가지 입력을 받는 층을 만들려면?

- call() : 모든 입력이 포함된 튜플을 매개변수 값을 전달, 여러 출력
을 가진 층을 만들려면 출력의 리스트 반환

- compute_output_shape() : 각 입력의 배치 크기를 담은 튜플이 매개
변수로 전달, 배치 출력 크기의 리스트 반환

```

class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        # call() 메서드는 심볼릭 입력을 받음
        # 이 입력의 크기는 부분적으로만 지정되어 있음
        # 이 시점에서는 배치 크기를 모름
        # 그래서 첫 번째 차원이 None임
        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):
        batch_input_shape1, batch_input_shape2 = batch_input_shape
        return [batch_input_shape1, batch_input_shape2]

inputs1 = keras.layers.Input(shape=[2])
inputs2 = keras.layers.Input(shape=[2])
outputs1, outputs2 = MyMultiLayer()(inputs1, inputs2)

```

- 훈련하는 동안 가우스 잡음을 추가하고 테스트 시에는 아무것도 하지 않는 층을 구현한 코드

```

# 훈련하는 동안 규제 목적으로 가우스 잡음을 추가하고
# 테스트 시에는 아무것도 하지 않는 층
class AddGaussianNoise(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

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

# model
model = keras.models.Sequential([
    AddGaussianNoise(stddev=1.0),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="nadam")
model.fit(X_train_scaled, y_train, epochs=2,
        validation_data=(X_valid_scaled, y_valid))
model.evaluate(X_test_scaled, y_test)

Epoch 1/2
363/363 [=====] - 1s 2ms/step - loss: 2.3857 - val_loss: 7.6082
Epoch 2/2
363/363 [=====] - 1s 2ms/step - loss: 1.0571 - val_loss: 4.4597
162/162 [=====] - 0s 1ms/step - loss: 0.7560

```

12.3.6 사용자 정의 모델

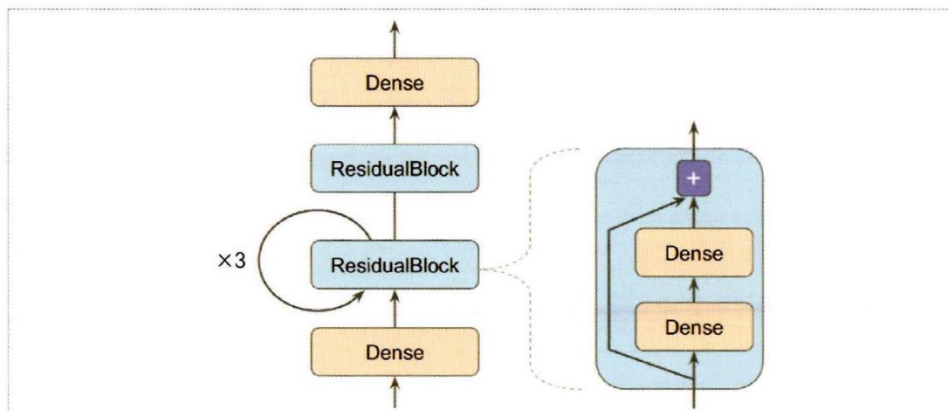


그림 12-3 사용자 정의 모델: 스킵 연결이 있는 사용자 정의 잔차 블록(ResidualBlock) 층을 가진 예제 모델

- 그림 12-3을 구현한 코드
- ResidualBlock class : 동일한 블록을 여러 개 만들기 위해
- ResidualRegressor class : 서브클래싱 API를 사용해 이 모델을 정의

```
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", # (exponential linear unit)
                                           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 # residual

# 서브클래싱 API를 사용해 이 모델을 정의해보자.
class ResidualRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu", # (exponential linear unit)
                                           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)

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = ResidualRegressor(1)
model.compile(loss="mse", optimizer="nadam")

# model.fit 사용 가능
history = model.fit(X_train_scaled, y_train, epochs=5)

# model.evaluate 사용 가능
score = model.evaluate(X_test_scaled, y_test)

# model.predict 사용 가능
y_pred = model.predict(X_new_scaled)

Epoch 1/5
363/363 [=====] - 2s 2ms/step - loss: 9.1325
Epoch 2/5
363/363 [=====] - 1s 2ms/step - loss: 1.0579
Epoch 3/5
363/363 [=====] - 1s 2ms/step - loss: 0.8869
Epoch 4/5
363/363 [=====] - 1s 2ms/step - loss: 0.5832
Epoch 5/5
363/363 [=====] - 1s 2ms/step - loss: 0.6461
```

12.3.7 모델 구성 요소에 기반한 손실과 지표

- 사용자 정의 재구성 손실을 가지는 모델을 만드는 코드

```

# 모델 구성요소?
# e.g.) 가중치, 출력층수

# 다섯개의 은닉층과 출력층으로 구성된 피쿼틀 MLP
class ReconstructingRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(80, activation='selu',
                                           kernel_initializer='lecun_normal')
                        for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)
        self.reconstruct = keras.layers.Dense(8) # TF 0.4 #48858에 대한 대응
        self.reconstruction_mean = keras.metrics.Mean(name='reconstruction_error')
        ...

    [Disclaimer]
    TF 2.2에 있는 이슈(#48858) 때문에 build() 메서드와 함께 add_loss()를 사용할 수 없습니다.
    따라서 다음 코드는 현재 작동합니다. build() 메서드 대신 self.reconstruct 층을 만듭니다.
    이 때문에 이 층의 유닛 개수를 하드코딩해야 합니다(또는 self가 매개변수로 전달해야 합니다).
    ...

# TF 0.4 #48858 때문에 주석 처리
def build(self, batch_input_shape):
    n_inputs = batch_input_shape[-1]
    self.reconstruct = keras.layers.Dense(n_inputs, name='recon')
    super().build(batch_input_shape)

# 입력이 다섯개의 은닉층을 모두 통과함
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)

...
CHECK THIS OUT
...
def train_step(self, data):
    x, y = data

    # tf.GradientTape() ?
    # https://www.tensorflow.org/guide/autodiff?hl=ko
    # 컨텍스트(context) 안에서 실행된 모든 연산을 테이프(tape)에 "기록"합니다.
    with tf.GradientTape() as tape:
        y_pred = self(x)
        # recon_loss
        loss = self.compiled_loss(y, y_pred, regularization_losses=[self.recon_loss])

    # tape.gradient
    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}

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = ReconstructingRegressor(1)
model.compile(loss='mse', optimizer='nadam')
history = model.fit(X_train_scaled, y_train, epochs=2)
y_pred = model.predict(X_test_scaled)

Epoch 1/2
985/985 [-----] - 2s 2ms/step - loss: 0.7885 - reconstruction_error: 0.0000e+00
Epoch 2/2
985/985 [-----] - 1s 2ms/step - loss: 0.4128 - reconstruction_error: 0.0000e+00

```

- build() : 완전 연결 층을 하나 더 추가하여 모델의 입력을 재구성하는 데 사용, 유닛 개수는 입력 개수와 같아야 함
- call() : 입력이 다섯 개의 은닉층에 모두 통과, 결과값을 재구성 층에 전달하여 재구성

12.3.8 자동 미분을 사용하여 그레디언트 계산하기

- tf.GradientTape 블록 : 이 변수와 관련된 모든 연산을 자동으로 기록
- gradient() 메서드가 호출된 후에는 자동으로 테이프가 즉시 지워짐 -> 동시에 두번 호출하면 에러가 남, 만약 한 번 이상 호출해야 한다면 지속

가능한 테이프를 만들고 사용이 끝난 후 테이프를 삭제하여 리소스를 해제해야 함

```
def f(w1, w2):
    #  $3x^2 + 2x + a \rightarrow 6x + 2$ 
    return 3 * w1 ** 2 + 2 * w1 + w2

w1, w2 = 5, 3
eps = 1e-6

# 미분계수 공식
print((f(w1 + eps, w2) - f(w1, w2)) / eps)

print((f(w1, w2 + eps) - f(w1, w2)) / eps)

w1, w2 = tf.Variable(5.), tf.Variable(3.)

# 변수와 관련된 모든 연산을 자동으로 기록
with tf.GradientTape() as tape:
    z = f(w1, w2)

# gradient()가 호출된 후에는 자동으로 테이프가 즉시 지워짐
# 그래서 tape.gradient()를 동시에 두번 호출하면 에러가 남
gradients = tape.gradient(z, [w1, w2])
print(gradients)

# 만약에 gradient를 두번 이상 호출하려면 지속가능한 테이프를 호출하고 사용이 끝나면 삭제해야됨
"""
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)
dz_dw2 = tape.gradient(z, w2) # works now!
del tape
"""
```

```
36.000003007075065
10.00000003174137
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>, <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

- 기본적으로 테이프는 변수가 포함된 연산만 기록 -> 아니면 None
- 필요하면 모든 연산을 기록하도록 강제할 수 있음
- 후진 모드 자동 미분
- 개별 그레디언트를 계산하고 싶으면 테이프의 jacobian() 메서드 호출

```
# 기본적으로 tape는 변수가 포함된 연산만을 기록함
# 그래서 변수가 아닌 다른 객체에 대한 그레디언트를 계산하면 None이 나옴
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])
print(gradients)

# 필요하면 모든 연산을 기록하도록 강제할 수 있음
# 사용자: 입력이 작을때 변수특이 큰 출성과 함수에 대한 규제손실을 구현할때 사용 가능
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])
print(gradients)
```

```
[None, None]
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>, <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

```
# "아~ 이렇게 있구나"
# 후진 모드 자동 미분 (reverse-mode autodiff)이 적합함
# 한번에 경방향 계산과 역방향 계산으로 모든 그레디언트를 동시에 계산할 수 있기 때문.
# 예를들어 모델 파라미터에 대한 각 손실의 그레디언트를 개별적으로 계산하고 싶다면 tape의 jacobian() 메서드를 호출해야함

# jacobian()
# 벡터에 있는 각 손실마다 후진 자동 미분을 수행함
# 이계도함수도 가능 (Hessian)
with tf.GradientTape() as tape:
    z1 = f(w1, w2 + 2.)
    z2 = f(w1, w2 + 5.)
    z3 = f(w1, w2 + 7.)

tape.gradient([z1, z2, z3], [w1, w2])

[<tf.Tensor: shape=(), dtype=float32, numpy=136.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=30.0>]
```

```

: with tf.GradientTape(persistent=True) as tape:
    z1 = f(w1, w2 + 2.)
    z2 = f(w1, w2 + 5.)
    z3 = f(w1, w2 + 7.)

tf.reduce_sum(tf.stack([tape.gradient(z, [w1, w2]) for z in (z1, z2, z3)]), axis=0)
del tape

: with tf.GradientTape(persistent=True) as hessian_tape:
    with tf.GradientTape() as jacobian_tape:
        z = f(w1, w2)
        jacobians = jacobian_tape.gradient(z, [w1, w2])
        Hessians = [hessian_tape.gradient(jacobian, [w1, w2])
                    for jacobian in jacobians]
    del hessian_tape

```

- 신경망의 일부분에 그레이디언트가 역전파 되지 않도록 막아야 할 때 `tf.stop_gradient()` 사용 but 역전파 시에는 그레이디언트를 전파하지 않음

- 그레이디언트를 계산할 때 수치적인 이슈가 발생할 수 있음 -> 수치적으로 안전한 소프트플러스의 도함수를 해석적으로 구할 수 있음 -> `@tf.custom_gradient` 데코레이터를 사용

```

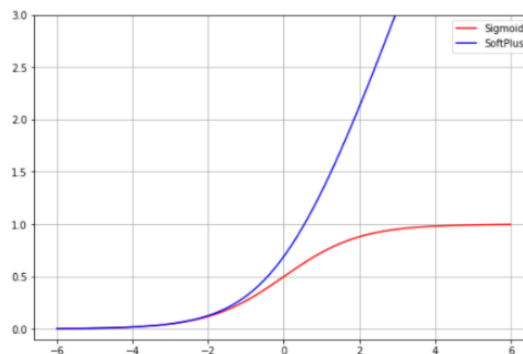
# 신경망의 일부분에 그레이디언트가 역전파되지 않도록 막아야 할 때
def f(w1, w2):
    # stop_gradient
    return 3 * w1 + 2 + tf.stop_gradient(2 * w1 + w2)

with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
print(gradients) # Note: None

[<tf.Tensor: shape=(), dtype=float32, numpy=30.0>, None]

```



```

# 그레이디언트의 수치적인 이슈
# e.g.) 큰 입력에 대해 my_softplus() -> NaN

x = tf.Variable(100.)
with tf.GradientTape() as tape:
    z = my_softplus(x)

tape.gradient(z, [x])

[<tf.Tensor: shape=(), dtype=float32, numpy=nan>]

```

```

@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    # 여전히 지수이기 때문에 값이 클 때 문제가 생길 수 있음
    return tf.math.log(exp + 1), my_softplus_gradients

# 해결방법: tf.where()를 사용해 값이 클 때 입력을 그대로 반환하는 것입니다.
"""
주석 풀어보기
"""
# def my_better_softplus(z):
#     return tf.where(z > 30., z, tf.math.log(tf.exp(z) + 1.))

x = tf.Variable([1000.])
with tf.GradientTape() as tape:
    z = my_better_softplus(x)

print(z, tape.gradient(z, [x]))

tf.Tensor([inf], shape=(1,), dtype=float32) [<tf.Tensor: shape=(1,), dtype=float32, numpy=array([1.], dtype=float32)>]

```

12.3.9 사용자 정의 훈련 반복

- 사용자 훈련 반복을 만들면 길고, 버그가 발생하기 쉽고, 유지 보수하기

어려운 코드가 만들어짐

- 극도의 유연성이 필요한 것이 아니라면 사용자 정의 훈련 반복 대신 fit() 메서드를 사용하는 것이 좋음

12.4 텐서플로 함수와 그래프

- tf.function()과 @tf.function을 사용해 텐서플로 함수로 변환

```
def cube(x):  
    return x ** 3  
  
tf_cube = tf.function(cube)  
  
print(cube(2))  
print(cube(tf.constant(2.0)))  
print(tf_cube(2))  
print(tf_cube(tf.constant(2.0)))  
  
8  
tf.Tensor(8.0, shape=(), dtype=float32)  
tf.Tensor(8, shape=(), dtype=int32)  
tf.Tensor(8.0, shape=(), dtype=float32)
```

파이썬 함수 -> 텐서플로 함수로 변환

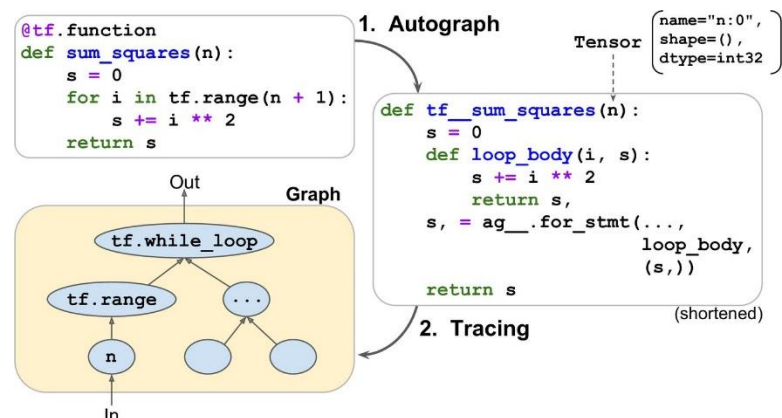
```
@tf.function  
def cube(x):  
    return x ** 3
```

데코레이터 사용

- 사용하지 않는 노드를 제거하고 표현을 단순화하여 계산 그래프 최적화
- 그래프 내의 연산을 효율적으로 실행
- 일반적으로 원본 파이썬 함수보다 훨씬 빠르게 실행됨
- 사용자 정의 손실, 지표, 층 등의 함수는 텐서플로 함수로 변환함
- 호출에 사용되는 입력 크기와 데이터 타입에 맞춰 매번 새로운 그래프

생성

12.4.1 오토그래프와 트레이싱



1. 오토그래프 : 파이썬 함수의 소스 코드를 분석하여 제어문을 찾음
2. 코드 분석 후 오토그래프는 이 함수의 모든 제어문을 텐서플로 연산으로

로 바꾼 업그레이드된 버전 만들

3. 텐서플로가 업그레이드된 함수를 호출, BUT 매개변수 값을 전달하는 대신 심볼릭 텐서 전달

4. 그래프 모드로 함수 실행

5. 트레이싱 : 노드는 연산을 나타내고 화살표는 텐서를 나타냄

12.4.2 텐서플로 함수 사용 방법

- 규칙 1. 텐서플로 구성 요소만 포함 가능

- 규칙 2. 다른 파이썬 함수나 텐서플로 함수를 호출할 수 있음, 이때는 데코레이터 적용할 필요 없음

- 규칙 3. 함수에서 텐서플로 변수를 만든다면 처음 호출될 때만 수행되어야 함, 아니면 예외 발생 => 텐서플로 변수는 함수 밖에서 생성하는 것이 좋음

- 규칙 4. 파이썬 함수의 소스 코드는 텐서플로에서 사용 가능해야 함

- 규칙 5. for문 사용 시, for i in tf.range(x) 사용

- 규칙 6. 반복문보다 가능한 한 벡터화 된 구현

12.5 연습문제

1. 텐서플로 한마디로 정의 / 주요 특징 / 인기 있는 딥러닝 라이브러리

- 텐서플로는 머신러닝을 위한 오픈소스 라이브러리

- 인기 있는 딥러닝 라이브러리로는 파이토치, MXNet, 마이크로소프트 코그니티브 툴킷 등이 있음

- 주요 특징으로 GPU 지원, 분산 컴퓨팅 기능 지원, 그래프 최적화 기능 제공, 다양한 API 제공을 뽑을 수 있음

2. 텐서플로는 넘파이를 그대로 대체 가능? / 둘의 차이점은?

- 유사한 점이 많지만 그대로 대체할 수는 없다.

1. 모든 함수 이름이 같지 않다.

2. 일부 함수는 작동 방식이 완전히 같지 않다.

3. 넘파이 배열은 변경 가능하지만 텐서플로의 텐서는 만든 후 변경하지 못한다

3. tf.range(10)와 tf.constant(np.arange(10))의 결과는 같나요?

- 둘 다 결과값은 같지만 텐서플로는 32비트가 기본이고 넘파이는 64비

트가 기본이기 때문에 dtype이 다르다.

4. 일반 텐서 외에 텐서플로에서 사용할 수 있는 6가지 다른 데이터 구조

- 최소 텐서, 텐서 배열, 래그드 텐서, 문자열 텐서, 집합, 큐

5. keras.losses.Loss 클래스를 상속 하거나 일반 함수를 작성하여 사용자 정의 손실 함수를 정의할 수 있습니다. 언제 사용해야 하나요?

- 사용자 정의 손실 함수에서 일부 매개변수를 사용하고 싶을 때

6. keras.metrics.Metric 클래스를 상속 하거나 함수를 정의하여 지표를 정의할 수 있습니다. 언제 사용해야 하나요?

- 사용자 정의 지표에서 일부 매개변수를 사용하고 싶을 때

7. 언제 사용자 정의 층 또는 사용자 정의 모델을 만들어야 하나요?

- 사용자 정의 층 : 모델을 구축하는데 쓰이는 내부 구성 요소, keras.layers.Layer 클래스 상속
- 사용자 정의 모델 : 층이 조합 된 모델 자체, keras.Model 클래스 상속

8. 사용자 정의 훈련 반복을 만들어야 하는 경우는 어떤 것이 있나요?

- 와이드 & 딥 논문 (10장)
- 신경망의 일부분에 각각 다른 옵티마이저를 사용하는 경우
- 되도록이면 사용자 정의 훈련 반복은 사용하지 않는게 좋음

9. 케라스의 사용자 정의 구성 요소가 임의의 파이썬 코드를 담을 수 있나요? 또는 이 구성 요소를 텐서플로 함수로 바꿀 수 있나요?

- 가능함
 1. 사용자 정의 구성 요소에 tf.py_function()으로 감싸 임의의 파이썬 코드를 넣을 수 있다. *but* 성능 감소, 모델 이식성 제약 (파이썬이 가능한 플랫폼에서만 실행되기 때문)
 2. 사용자 정의 층이나 모델을 만들 때 dynamic = True로 지정
 3. 모델의 compile() 메서드 호출할 때 run_eagerly=True로 지정
- 사용 이유

1. 텐서플로는 사용하지 않는 노드를 제거하고 표현을 단순화 하는 방식으로 계산 그래프 최적화
2. 텐서플로 함수는 원본 파이썬 함수보다 훨씬 빠르게 실행
3. 파이썬 함수를 빠르게 실행하기 위해

10. 텐서플로 함수로 바꿀 수 있는 함수를 만든다면 따라야 할 주요 규칙은?

- 규칙 1. 텐서플로 구성 요소만 포함 가능
- 규칙 2. 다른 파이썬 함수나 텐서플로 함수를 호출할 수 있음, 이때는 데코레이터 적용할 필요 없음
- 규칙 3. 함수에서 텐서플로 변수를 만든다면 처음 호출될 때만 수행되어야 함, 아니면 예외 발생 => 텐서플로 변수는 함수 밖에서 생성하는 것이 좋음
- 규칙 4. 파이썬 함수의 소스 코드는 텐서플로에서 사용 가능해야 함
- 규칙 5. for문 사용 시, for i in tf.range(x) 사용
- 규칙 6. 반복문보다 가능한 한 벡터화 된 구현

11. 동적인 케라스 모델을 만들어야 할 때는 언제인가요? 어떻게 만들 수 있나요? 왜 전체 모델을 동적으로 만들지 않나요?

- 디버깅에 유용함
- 모델을 생성할 때 `dynamic=True`로 지정 or 모델의 `compile()` 메서드를 호출할 때 `run_eagerly=True`를 지정
- 모델을 동적으로 만들면 케라스가 텐서플로의 그래프 장점을 활용하지 못함 -> 훈련과 추론 속도가 느려짐, 모델 이식성에 제약이 생김

12. Layer normalization(층 정규화)를 수행하는 사용자 정의 층을 구현하세요

a. `build()` 메서드에서 두 개의 훈련 가능한 가중치 α 와 β 를 정의합니다. 두 가중치 모두 크기가 `input_shape[-1:]`이고 데이터 타입은 `tf.float32`입니다. α 는 1로 초기화되고 β 는 0으로 초기화되어야 합니다.

```
def build(self, batch_input_shape):
    self.alpha = self.add_weight(
        name="alpha", shape=batch_input_shape[-1:],
        initializer="ones")
    self.beta = self.add_weight(
        name="beta", shape=batch_input_shape[-1:],
        initializer="zeros")
    super().build(batch_input_shape) # 반드시 끝에 와야 합니다
```

b. call() 메서드는 샘플의 특성마다 평균 μ 와 표준편차 σ 를 계산해야 합니다. 이를 위해 전체 샘플의 평균 μ 와 분산 σ^2 을 반환하는 `tf.nn.moments(inputs, axes=-1, keepdims=True)`을 사용할 수 있습니다(분산의 제곱근으로 표준편차를 계산합니다). 그다음 $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$ 를 계산하여 반환합니다. 여기에서 \otimes 는 원소별 곱셈(*)을 나타냅니다. ϵ 은 안전을 위한 항입니다(0으로 나누어지는 것을 막기 위한 작은 상수. 예를 들면 0.001).

```
class LayerNormalization(keras.layers.Layer):
    def __init__(self, eps=0.001, **kwargs):
        super().__init__(**kwargs)
        self.eps = eps

    def build(self, batch_input_shape):
        self.alpha = self.add_weight(
            name="alpha", shape=batch_input_shape[-1:],
            initializer="ones")
        self.beta = self.add_weight(
            name="beta", shape=batch_input_shape[-1:],
            initializer="zeros")
        super().build(batch_input_shape) # 반드시 끝에 와야 합니다

    def call(self, X):
        mean, variance = tf.nn.moments(X, axes=-1, keepdims=True)
        return self.alpha + (X - mean) / (tf.sqrt(variance + self.eps)) + self.beta

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "eps": self.eps}
```

ϵ 하이퍼파라미터(eps)는 필수가 아님. 또한 `tf.sqrt(variance) + self.eps` 보다 `tf.sqrt(variance + self.eps)`를 계산하는 것이 좋음. `sqrt(z)`의 도함수는 $z=0$ 에서 정의되지 않기 때문에 분산 벡터의 한 원소가 0에 가까우면 훈련이 이리저리 널뛰는데, 제곱근 안에 ϵ 를 넣으면 이런 현상을 방지할 수 있습니다.

c. 사용자 정의 층이 ``keras.layers.LayerNormalization`` 층과 동일한(또는 거의 동일한) 출력을 만드는지 확인하세요.

```
X = X_train.astype(np.float32)

custom_layer_norm = LayerNormalization()
keras_layer_norm = keras.layers.LayerNormalization()

tf.reduce_mean(keras.losses.mean_absolute_error(
    keras_layer_norm(X), custom_layer_norm(X)))
```

<tf.Tensor: shape=(), dtype=float32, numpy=3.80914e-08>

네 충분히 가깝네요. 조금 더 확실하게 알파와 베타를 완전히 랜덤하게 지정하고 다시 비교해 보죠:

```
random_alpha = np.random.rand(X.shape[-1])
random_beta = np.random.rand(X.shape[-1])

custom_layer_norm.set_weights([random_alpha, random_beta])
keras_layer_norm.set_weights([random_alpha, random_beta])

tf.reduce_mean(keras.losses.mean_absolute_error(
    keras_layer_norm(X), custom_layer_norm(X)))
```

<tf.Tensor: shape=(), dtype=float32, numpy=1.695759e-08>

13. 사용자 정의 훈련 반복을 사용해 패션 MNIST 데이터셋으로 모델을 훈련해 보세요.

a. 에포크, 반복, 평균 훈련 손실, (반복마다 업데이트되는) 에포크의 평균 정확도는 물론 에포크 끝에서 검증 손실과 정확도를 출력

```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full = X_train_full.astype(np.float32) / 255.
X_valid, X_train = X_train_full[:6000], X_train_full[6000:]
y_valid, y_train = y_train_full[:6000], y_train_full[6000:]
X_test = X_test.astype(np.float32) / 255.

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
62768/29516 [=====] - 0s Ous/step
40980/29516 [=====] - 0s Ous/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
28427392/28421880 [=====] - 0s Ous/step
28436584/28421880 [=====] - 0s Ous/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
18384/5148 [=====] - 0s Ous/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [=====] - 0s Ous/step
4431672/4422102 [=====] - 0s Ous/step

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax"),
])


n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.sparse_categorical_crossentropy
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.SparseCategoricalAccuracy()]


/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/optimizer_v2.py:868: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```


```


with trange(1, n_epochs + 1, desc="All epochs") as epochs:
    for epoch in epochs:
        with trange(1, n_steps + 1, desc="Epoch {}/{}".format(epoch, n_epochs)) as steps:
            for step in steps:
                X_batch, y_batch = random_batch(X_train, 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))
                status = OrderedDict()
                mean_loss(loss)
                status["loss"] = mean_loss.result().numpy()
                for metric in metrics:
                    metric(y_batch, y_pred)
                    status[metric.name] = metric.result().numpy()
                steps.set_postfix(status)
                y_pred = model(X_valid)
                status["val_loss"] = np.mean(loss_fn(y_valid, y_pred))
                status["val_accuracy"] = np.mean(keras.metrics.sparse_categorical_accuracy(
                    tf.constant(y_valid, dtype=np.float32), y_pred))
                steps.set_postfix(status)
            for metric in [mean_loss] + metrics:
                metric.reset_states()


```


All epochs: 100%  5/5 [03:07<00:00, 37.36s/it]

Epoch 1/5: 100%  1718/1718 [00:38<00:00, 47.59it/s, loss=0.511, sparse_categorical_accuracy=0.816]

Epoch 2/5: 100%  1718/1718 [00:37<00:00, 46.25it/s, loss=0.408, sparse_categorical_accuracy=0.854]

Epoch 3/5: 100%  1718/1718 [00:37<00:00, 48.13it/s, loss=0.374, sparse_categorical_accuracy=0.865]

Epoch 4/5: 100%  1718/1718 [00:37<00:00, 46.01it/s, loss=0.368, sparse_categorical_accuracy=0.869]

Epoch 5/5: 100%  1718/1718 [00:37<00:00, 47.02it/s, loss=0.362, sparse_categorical_accuracy=0.869]

b. 상위 층과 하위 층에 학습률이 다른 옵티마이저를 따로 사용해 보세요.

```

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)







[ ] lower_layers = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(100, activation="relu"),
])
upper_layers = keras.models.Sequential([
    keras.layers.Dense(10, activation="softmax"),
])
model = keras.models.Sequential([
    lower_layers, upper_layers
])

[ ] lower_optimizer = keras.optimizers.RMSprop(lr=1e-4)
upper_optimizer = keras.optimizers.Nadam(lr=1e-5)

/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/optimizer_v2.py:356: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")

[ ] n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
loss_fn = keras.losses.sparse_categorical_crossentropy
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.SparseCategoricalAccuracy()]

```

	<pre> with trange(1, n_epochs + 1, desc="All epochs") as epochs: for epoch in epochs: with trange(1, n_steps + 1, desc="Epoch {}/{}".format(epoch, n_epochs)) as steps: for step in steps: X_batch, y_batch = random_batch(X_train, y_train) with tf.GradientTape(persistent=True) 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) for layers, optimizer in ((lower_layers, lower_optimizer), (upper_layers, upper_optimizer)): gradients = tape.gradient(loss, layers.trainable_variables) optimizer.apply_gradients(zip(gradients, layers.trainable_variables)) del tape for variable in model.variables: if variable.constraint is not None: variable.assign(variable.constraint(variable)) status = OrderedDict() mean_loss(loss) status["loss"] = mean_loss.result().numpy() for metric in metrics: metric(y_batch, y_pred) status[metric.name] = metric.result().numpy() steps.set_postfix(status) y_pred = model(X_valid) status["val_loss"] = np.mean(loss_fn(y_valid, y_pred)) status["val_accuracy"] = np.mean(keras.metrics.sparse_categorical_accuracy(tf.constant(y_valid, dtype=np.float64), y_pred)) steps.set_postfix(status) for metric in [mean_loss] + metrics: metric.reset_states() </pre> <p> All epochs: 100%  5/5 [03:00<00:00, 36.04s/it] Epoch 1/5: 100%  1718/1718 [00:36<00:00, 49.65it/s, loss=1.06, sparse_categorical_accuracy=0.679] Epoch 2/5: 100%  1718/1718 [00:35<00:00, 48.54it/s, loss=0.648, sparse_categorical_accuracy=0.783] Epoch 3/5: 100%  1718/1718 [00:35<00:00, 49.57it/s, loss=0.572, sparse_categorical_accuracy=0.802] Epoch 4/5: 100%  1718/1718 [00:36<00:00, 48.74it/s, loss=0.537, sparse_categorical_accuracy=0.811] Epoch 5/5: 100%  1718/1718 [00:36<00:00, 47.57it/s, loss=0.523, sparse_categorical_accuracy=0.814] </p> <p>12번, 13번 문제 참고 url https://colab.research.google.com/github/rickiepark/handson-ml2/blob/master/12_custom_models_and_training_with_tensorflow.ipynb#scrollTo=aHM1uuinnFSY </p>
과제할당	<p>Chapter 13</p> <p>13.1 서가을</p> <p>13.2 이문기</p> <p>13.3 안세윤</p> <p>13.4 권기호</p> <p>13.5 허주희</p> <p>13.6 고성호</p>
특이	<p>없습니다.</p>

사 항	
비 고	<ul style="list-style-type: none"> - 중간고사 끝나고 모각코 or 테버랜드 모임 - 휴동 : 10월 22일, 10월 29일