Ch12_텐서플로를 사용한 사용자 정의 모델과 훈련

▼ ref

텐서 소개 | TensorFlow Core

https://www.tensorflow.org/guide/tensor?hl=ko

[핸즈온 머신러닝] 12장. 텐서플로를 사용한 사용자 정의 모델과 훈련

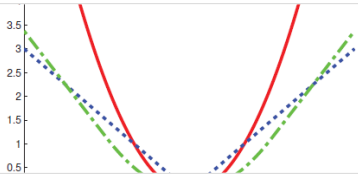
part 2 신경망과 딥러닝 부분을 개인공부를 목적으로 내용 요약 및 정리한 글입니다. - 텐서플로를 사용한 사용자 정의 모델과 훈련

https://ingu627.github.io/hands_on/TDL3/

[핸즈온 머신러닝 2판] 12장 정리

출처핸즈온 머신러닝2핸즈온 머신러닝2 github되도록이면 책의 내용과 코드를 그대로 옮기기 보다는 요약과 보충설명!구글에서 개발, 가장 널리 쓰이는 딥러닝 라이브러리넘파이와 비슷해 보이지만, GPU를 지원다양한 연산 기능 제공커널(kernel)파이썬 API이외에도 C

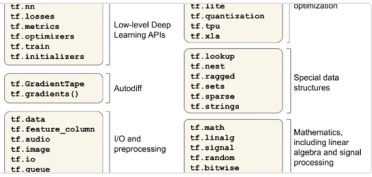
https://velog.io/@juyeonma9/핸즈온-머신러닝-2판-12장-정리



Hands-On Machine Learning Chapter 12 텐서플로를 사용한 사용자 정의 모델과 훈련

12.1 텐서플로 훑어보기 텐서플로는 강력한 수치 계산용 라이브러리. 특히 대규모 머신러닝에 잘 맞도록 튜닝 되어 있다. 또한 계산량이 많은 어떠한 작업에도 사용할 수 있다. 텐서플로의 핵심은 다음과 같다. 핵심구조는 numpy와 비슷하지만 GPU를 지원한다. 여러 장치와 서버에 대해서 분산 컴퓨팅을 지원한다. 일종의

https://velog.io/@wlsn404/Hands-On-Machine-Learning-Chapter-12-텐서플로를-사용한-사용자-정의-모델과-훈련



[인공지능] TensorFlow 기초 문법

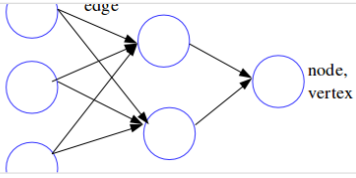
참조 https://www.youtube.com/watch?v=B961QM47g64&t=1037s TensorFlow 란 가장 널리 쓰이는 딥러닝 프레임워크 중 하나 구글이 주도적으로 개발하는 플래폼 파이썬, C++ API를 기본적으로 제공하고, 자바스크립트(JavaScript), 자바(Java), 고(Go), 스위프트(Swift) 등 다양한 프로그래밍 언어를 지원 tf.keras를 중심으로 고수

https://afsdzvcx123.tistory.com/entry/인공지능-TensorFlow-기초-문법

[러닝 텐서플로]Chap03 - 텐서플로의 기본 이해하기

Chap03 - 텐서플로의 기본 이해하기텐서플로의 핵심 구축 및 동작원리를 이해하고, 그래프를 만들고 관리하는 방법과 상수, 플레이스홀더, 변수 등 텐서플로의 '구성 요소'에 대해 알아보자. 3.1 연산 그래프3.1.1 연산 그래프란?그래프는 아래의 그림과 같이 노드(node)나 꼭지점(vertex)로 연결 되어 있는 개체(entity)의 집합을 부르는

https://excelsior-cjh.tistory.com/151



▼ 12.1 **텐서플로 훑어보기**

텐서플로

텐서플로는 강력한 **수치 계산용 라이브러리**로, *대규모 머신러닝*에 잘 맞도록 튜닝되어 있다. **딥러닝 프로그램**을 쉽게 구현할 수 있도록 다양한 기능을 제공해주는 라이브러리이다.

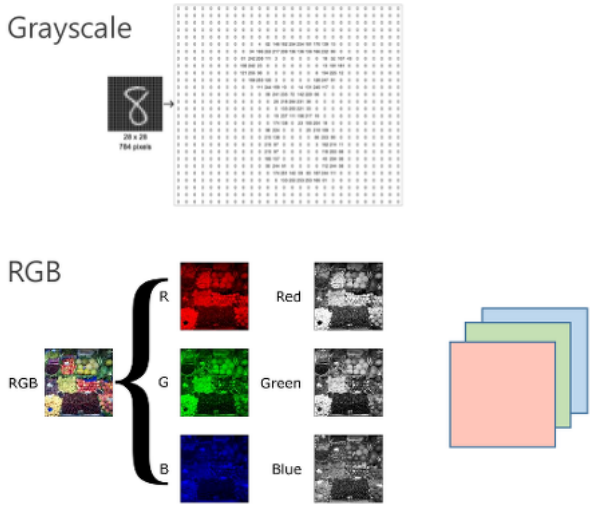
텐서플로가 제공하는 기능은 다음과 같다.

- 핵심 구조는 넘파이와 매우 비슷하지만 **GPU를 지원**한다.
- (여러 장치와 서버에 대해서) **분산 컴퓨팅**을 지원한다.
- 일종의 JIT 컴파일러를 포함한다. 속도를 높이고 메모리 사용량을 줄이기 위해 계산을 최적화한다. 이를 위해 파이썬 함수에서 계산 그래프를 추출한 다음 최적화하고 효율적으로 실행한다.
- 계산 그래프는 플랫폼에 중립적인 포맷으로 내보낼 수 있다.
- 텐서플로는 자동 미분 기능과 RMSProp, Nadam 같은 고성능 옵티마이저를 제공하므로 모든 종류의 손실 함수를 최소화할 수 있다.
- 가장 저수준의 텐서플로 연산은 매우 효율적인 C++ 코드로 구현되어 있다.
- 많은 연산은 **커널**이라 부르는 여러 구현을 가진다.
- 각 커널은 CPU, GPU 또는 TPU와 같은 특정 장치에 맞추어 만들어졌다.
- TPU는 딥러닝 연산을 위해 특별하게 설계된 ASIC 칩이다.
- GPU는 계산을 작은 단위로 나누어 여러 GPU 스레드에서 병렬로 실행하므로 속도를 극적으로 향상한다.

텐서플로의 의미

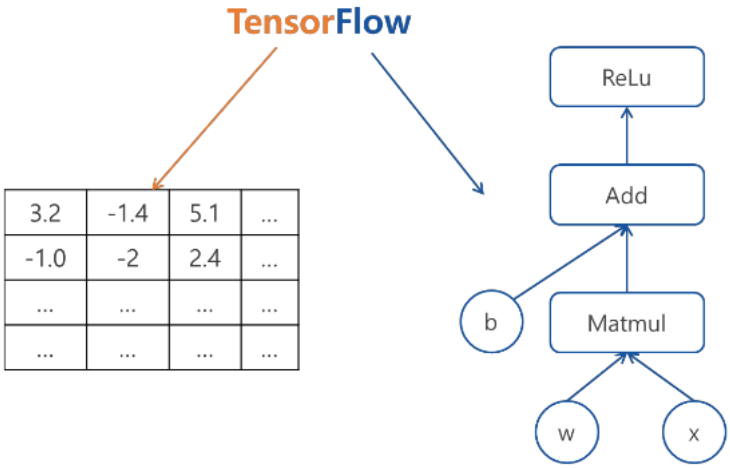
TensorFlow에서 **Tensor(텐서)**란 딥러닝에서 데이터를 표현하는 방식이라고 할 수 있다. 즉, 텐서는 **행렬로 표현할 수 있는 2차원 형태의 배열을 높은 차원으로 확장한 다차원 배열**이다.

예를 들어, 아래의 그림에서 볼 수 있듯이 회색조(grayscale) 이미지는 하나의 채널(channel)에 2차원 행렬(배열)로 나타낼 수 있다. 반면, RGB 이미지는 R(ed), G(reen), B(lue) 각 3개의 채널마다 2차원 행렬(배열)로 표현하는데, 이를 **텐서(3차원의 값을 가지는 배열)**로 표현할 수 있다.



TenorFlow에서 계산은 **데이터 흐름 그래프(dataflow graph)**로 이루어 진다. 즉, 텐서 형태의 데이터들이 딥러닝 모델을 구성하는 연산들의 그래프를 따라 흐르면서 연산이 일어난다.

따라서, 딥러닝에서 데이터를 의미하는 **Tensor** 와 DataFlow Graph를 따라 연산이 수행되는 형태(**Flow**)를 합쳐 **TensorFlow** 란 이름이 나오게 되었다.



텐서플로 구조

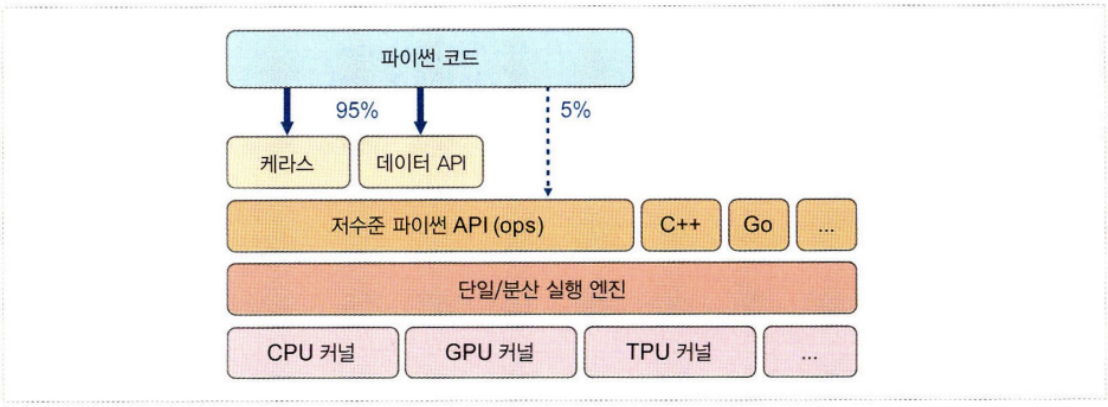


그림 12-2 텐서플로 구조

▼ 12.2 넘파이처럼 텐서플로 사용하기

▼ 12.2.1 텐서와 연산

- `tf.constant()` 함수 : 텐서를 만들 수 있다.

```
import tensorflow as tf

tf.constant([[1.,2.,3.], [4.,5.,6.]]) # 행렬 - 2개의 행과 3개의 열을 가진 실수 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.Tensor`는 크기(shape)와 데이터 타입(Dtype)을 가진다.

```
t = tf.constant([[1.,2.,3.], [4.,5.,6.]])
t.shape # TensorShape([2, 3])
t.dtype # tf.float32
```

• **인덱스 참조**도 넘파이와 매우 비슷하게 작동한다.

```
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)>

#float32는 32bits를 사용
#tf.newaxis : 차원(dimension)을 추가하는 방법
```

- **차원 추가**

[tensorflow] 데이터 사이즈 변경 기초

> 데이터를 모델에 집어넣기 전에 데이터 사이즈를 맞춰줘야 한다. 기본적으로 tensorflow에서 제공하는 MNIST 예제를 통해 데이터 size를 변경하는 법을 알아보자. 1. 데이터 불러오기 데이터 사이즈 변경을 위한 블로그이기 때문에 코드만 간단히 적어보겠다. import tensorflow as tf from tensorflow.keras import

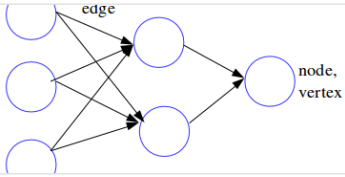
https://han-py.tistory.com/341

- **모든 종류의 텐서 연산이 가능**하다.

[러닝 텐서플로]Chap03 - 텐서플로의 기본 이해하기

Chap03 - 텐서플로의 기본 이해하기텐서플로의 핵심 구축 및 동작원리를 이해하고, 그래프를 만들고 관리하는 방법과 상수, 플레이스홀더, 변수 등 텐서플로의 '구성 요소'에 대해 알아보자. 3.1 연산 그래프3.1.1 연산 그래프란?그래프는 아래의 그림과 같이 노드(node)나 꼭지점(vertex)로 연결 되어 있는 개체(entity)의 집합

https://excelsior-cjh.tistory.com/151



- `t+10`이라고 쓰는 것은 `tf.add(t,10)`을 호출하는 것과 같다.

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

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

t @ tf.transpose(t) # transpose는 행렬 변환
# <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
# array([[14., 32.],
#        [32., 77.]], dtype=float32)>
```

- @연산은 `tf.matmul()` 함수를 호출하는 것과 동일

- 필요한 모든 기본 수학 연산 `tf .add()` , `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`과 넘파이에서 볼 수 있는 부분의 연산 `tf.reshape()` , `tf.squeeze()` , `tf. tile()` 제공
- `tf. reduce_mean()` = `np.mean()`

`tf.reduce_sum()` = `np.sum()`

`tf.reduce_max()` = `np.max()`

`tf.math log()` = `np.log()`

▼ 12.2. 텐서와 넘파이

- **넘파이 배열에 텐서플로 연산을 적용**할 수 있고 **텐서에 넘파이 연산을 적용**할 수도 있다.
- **넘파이는 기본으로 64비트 정밀도**를 사용하지만 **텐서플로는 32비트 정밀도**를 사용한다.
- 넘파이 배열로 텐서를 만들 때 `dtype=tf.float32`로 지정해야 한다.

```
import numpy as np
a = np.array([2., 4., 5.])

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

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

- 텐서플로는 어떤 타입 변환도 자동으로 수행하지 않는다.
- 호환되지 않는 타입의 텐서로 연산을 실행하면 예외가 발생한다.
- 타입 변환이 필요할 때는 `tf.cast()` 함수를 사용한다.

```
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.Variable`는 텐서의 내용을 바꿀 수 있다. : `tf.Variable(value, name, dtype, shape)`

변수의 값을 증가시키거나 원소의 값을 바꾸면 새로운 텐서가 만들어진다.


```
v = tf.Variable([[1.,2.,3.], [4.,5.,6.]])
v
```

```
# <tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
# array([[1., 2., 3.],
#        [4., 5., 6.]], dtype=float32)>
```

- `assign()` 메서드를 사용하여 변숫값을 바꿀 수 있다.
  - `assign_add()` 나 `assign_sub()` 메서드를 사용하면 주어진 값만큼 변수를 증가시키거나 감소시킬 수 있다.
- 원소의 `assign()` 메서드나 `scatter_update()`, `scatter_nd_update()` 메서드를 사용하여 개별 원소를 수정할 수 있다.

```
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** (sparse tensor) `tf.SparseTensor()`

  희소 텐서 작업 | TensorFlow Core

  https://www.tensorflow.org/guide/sparse_tensor?hl=ko

  TensorFlow

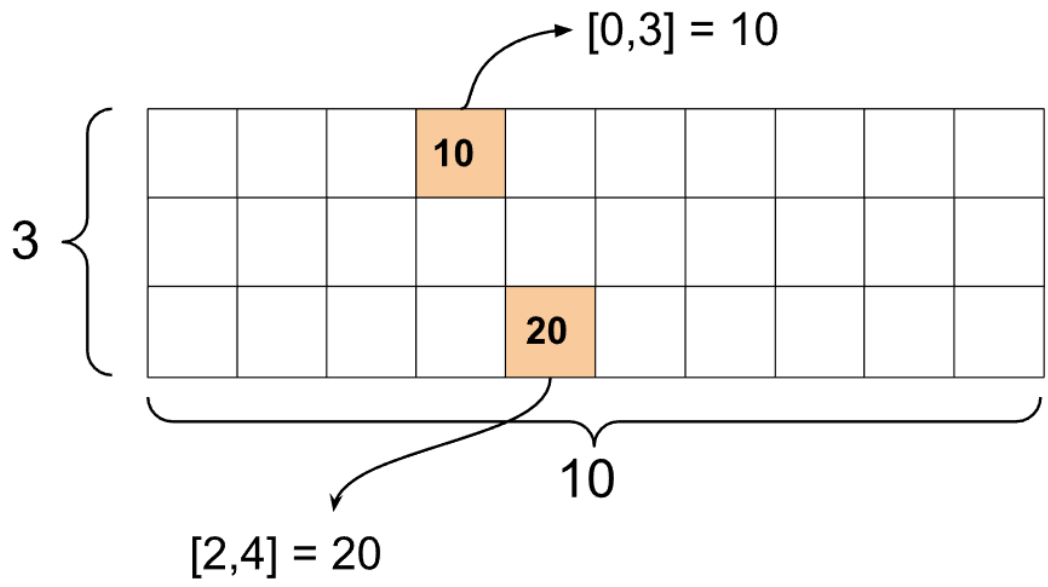
  : 0 값이 많이 포함된 텐서로 작업할 때 공간 및 시간 효율적인 방식으로 저장하는 것이 중요하다. 희소 텐서는 0 값이 많이 포함된 텐서를 효율적으로 저장하고 처리할 수 있도록 한다.

  `values` , `indices` 및 `dense_shape` 를 직접 지정하여 희소 텐서를 구성한다.

  - `values` : 0이 아닌 모든 값을 포함하는 `[N]` 모양의 1D 텐서.
  - `indices` : 0이 아닌 값의 인덱스를 포함하는 `[N, rank]` 모양의 2D 텐서.
  - `dense_shape` : 텐서의 모양을 지정하는 `[rank]` 모양의 1D 텐서.

```
import tensorflow as tf

st1 = tf.SparseTensor(indices=[[0, 3], [2, 4]],
                      values=[10, 20],
                      dense_shape=[3, 10])
```



- **텐서 배열** (tensor array) `tf.TensorArray()`

  : **텐서의 리스트**이다. 기본적으로 고정된 길이를 가지지만 동적으로 바꿀 수 있다.

- **래그드 텐서** (ragged tensor) `tf.RaggedTensor()`

  : **리스트의 리스트**를 나타낸다. 텐서에 포함된 값은 동일한 데이터 타입을 가져야 하지만 리스트의 길이는 다를 수 있다.

- **문자열 텐서** (string tensor)

  : `tf.string` 타입의 텐서

- **집합** (set)

  : 집합은 일반적인 텐서 (또는 희소 텐서)로 나타낸다.

- **큐** (queue) (`tf.queue` 패키지에 포함)

  : 큐는 단계별로 텐서를 저장한다.

▼ 12.3 사용자 정의 모델과 훈련 알고리즘

▼ 12.3.1 사용자 정의 손실 함수

> ## 후버 손실(Huber loss)

: Huber loss는 제곱 오차 손실보다 데이터의 특이치에 덜 민감하게 반응하는 강력한 Regression 또는 Classifcation에 사용되는 손실 함수

- 후버 손실 정의하기

```
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)
```

이 손실을 사용해 케라스 모델의 컴파일 메서드를 호출하고 모델을 훈련할 수 있다.

```
model.compile(loss=huber_fn, optimizer='nadam')
model.fit(X_train, y_train, [...])
```

• 훈련하는 동안 배치마다 **케라스는** `huber_fn()` **함수**를 호출하여 **손실을 계산**하고, 이를 사용해 경**사 하강법을 수행**한다. 또한 에포크 시작부터 전체 손실을 기록하여 **평균 손실을 출력**한다.

▼ 12.3.2 사용자 정의 요소를 가진 모델을 저장하고 로드하기

- 모델을 로드할 때는 함수 이름과 실제 함수를 매핑한 딕셔너리를 전달해야 한다.
- 사용자 정의 객체를 포함한 모델을 로드할 때는 그 이름과 객체를 매핑해야 한다.

```
from tensorflow.keras.models import load_model

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

• 매개변수를 받을 수 있는 함수 만들기

```
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")
```

• 모델을 로드할 때 threshold 값을 지정해야 한다.

```
model = load_model("my_model_with_a_custom_loss_threshold_2.h5",
                   custom_objects={"huber_fn": create_huber(2.0)})
```

• 이 문제는 `keras.losses.Loss` 클래스를 상속하고 `get_config()` 메서드를 구현하여 해결할 수 있다.

```
from tensorflow.keras.losses import Loss

class HuberLoss(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}
```

• 모델을 컴파일 할 때 이 클래스의 인스턴스를 사용할 수 있다.

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

• 이 모델을 저장할 때 임곗값도 함께 저장된다. 모델을 로드할 때 클래스 이름과 클래스 자체를 매핑해주어야 한다.

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

▼ 12.3.3 활성화 함수, 초기화, 규제, 제한을 커스터마이징하기

• 사용자 정의 활성화 함수 `keras.activations.softplus()`

```
ef my_softplus(z):
    return tf.math.log(tf.exp(z) + 1.0)
```

• 사용자 정의 글로럿 초기화 `keras.initializers.glorot_normal()`

```
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)
```

• 사용자 정의 l_1 규제 `keras.regularizers.l1(0.01)`

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

• 양수인 가중치만 남기는 사용자 정의 제한 `keras.constraints.nonneg()`

```
def my_positive_weights(weights):
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

• 만들어진 사용자 정의 함수는 보통의 함수와 동일하게 사용할 수 있다.

```
from tensorflow.keras.layers import Dense

layer = Dense(30 activation=my_softplus,
              kernel_initializer=my_glorot_initializer,
```

```
            kernel_regularizer=my_l1_regularizer,
            kernel_constarint=my_positive_weights)
```

- 이 활성화 함수는 Dense 층의 출력에 적용되고 다음 층에 그 결과가 전달 된다.
- 층의 가중치는 초기화 함수에서 반환된 값으로 초기화된다.
- 훈련 스텝마다 가중치가 규제 함수에 전달되어 규제 손실을 계산하고 전체 손실에 추가되어 훈련을 위한 최종 손실을 만든다.
- 마지막으로 제한 함수가 훈련 스텝마다 호출되어 층의 가중치를 제한한 가중치 값으로 바뀐다.

▼ 12.3.4 사용자 정의 지표

- **손실**은 모델을 훈련하기 위해 경사 하강법에서 사용하므로 미분 가능해야 하고 그레이디언트가 모든 곳에서 0이 아니어야 한다.
- **지표**는 모델을 평가할 때 사용한다. 미분이 가능하지 않거나 모든 곳에서 그레이디언트가 0이어도 괜찮다.

```
model.compile(loss='mse', optimizer='nadam', metrics=[create_huber(2.0)])
```

• 전체 후버 손실과 지금까지 처리한 샘플 수를 기록하는 클래스 생성해보기

```
from tensorflow.keras.metrics import Metric
import tensorflow as tf

class HuberMetric(Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        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 정의 층

• 텐서플로에는 없는 특이한 층을 가진 네트워크를 만들어야 할 때가 있다. 이런 경우 사용자 정의 층을 만든다.

```
from tensorflow.keras.layers import Layer

class MyDense(Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = tf.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)
```

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

    def comput_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': tf.keras.activations.serialize(self.activation)}
```

• 두 개의 입력과 세 개의 출력을 만드는 층 만들어보기

```
class MyMultiLayer(tf.keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1]
훈련과 테스트에서 다르게 동작하는 층
```

- 훈련과 테스트에서 다르게 동작하는 층이 필요하다면 call() 메서드에 training 매개변수를 추가하여 훈련인지 테스트인지를 결정해야 한다.
- 훈련하는 동안 (규제 목적으로) 가우스 잡음을 추가하고 테스트 시에는 아무것도 하지 않는 층을 만들어보기

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

• **사용자 정의 모델**: 아래는 스킵 연결이 있는 사용자 정의 잔차 블록(ResidualBlock) 층을 가진 예제 모델

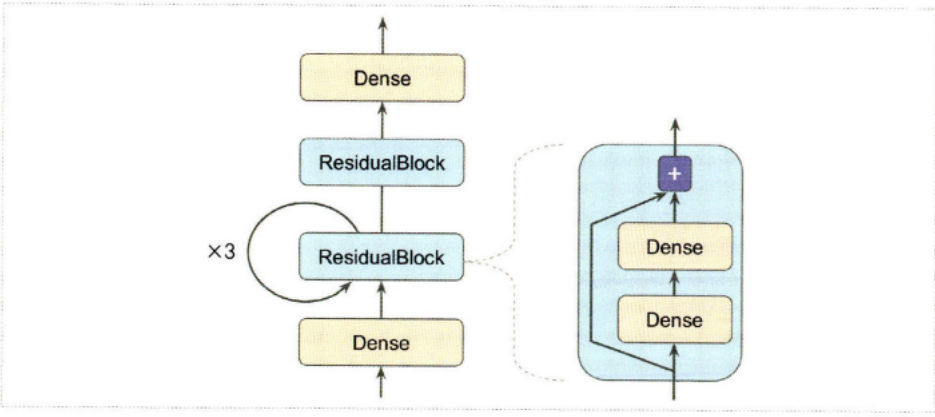


그림 12-3 사용자 정의 모델: 스킵 연결이 있는 사용자 정의 잔차 블록(ResidualBlock) 층을 가진 예제 모델

- 방법 : `keras.Model` 클래스를 상속하여 생성자에서 층과 변수를 만들고 모델이 해야 할 작업을 `call()` 메서드에 구현한다.
- 입력이 첫 번째 완전 연결 층을 통과하여 두 개의 완전 연결 층과 스킵 연결로 구성된 `잔차 블록` (residual block)으로 전달된다.
- 그 다음 동일한 잔차 블록에 세 번 더 통과시킨다.
- 그다음 두 번째 잔차 블록을 지나 마지막 출력이 완전 연결된 출력 층에 전달된다.

```
import tensorflow as tf

# 이 층은 케라스가 알아서 추적해야 할 객체가 담긴 hidden 속성을 감지하고 필요한 변수를 자동으로 이 층의 변수 리스트에 추가한다,
class ResidualBlock(tf.keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [tf.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 이 모델을 정의해보기

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

▼ 12.3.7 모델 구성 요소에 기반한 손실과 지표

- 모델 구성 요소에 기반한 손실을 정의하고 계산하여 `add_loss()` 메서드에 그 결과를 전달한다.

- 사용자 정의 재구성 손실을 가지는 모델을 만들어보기
  - 맨 위의 은닉층에 보조 출력을 가짐. 이 보조 출력에 연결된 손실을 `재구성 손실` (재구성과 입력 사이의 평균 제곱 오차)이라 함
  - 재구성 손실을 주 손실에 더하여 회귀 작업에 직접적으로 도움이 되지 않은 정보일지라도 모델이 은닉층을 통과하면서 가능한 많은 정보를 유지하도록 유도한다.

```
class ReconstructingRegressor(tf.keras.Model):
    def __init__(self, output_dim, **kwargs):
        # 생성자가 다섯 개의 은닉층과 하나의 출력층으로 구성된 심층 신경망을 만든다.
        super().__init__(**kwargs)
        self.hidden = [tf.keras.layers.Dense(30, activation='selu',
                                             kernel_initializer='lecun_normal')
                       for _ in range(5)]
        self.out = tf.keras.layers.Dense(output_dim)

    # 완전 연결 층을 하나 더 추가하여 모델의 입력을 재구성하는 데 사용한다.
    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = tf.keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    # 입력이 다섯 개의 은닉층에 모두 통과한다. 그 다음 결괏값을 재구성 층에 전달하여 재구성을 만든다.
    # 재구성 손실(재구성과 입력 사이의 평균 제곱 오차)을 계산하고 add_loss() 메서드를 사용해 모델의 손실 리스트에 추가한다.
    # 마지막에서 은닉층의 출력을 출력층에 전달하여 얻은 출력값을 반환한다.
    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

▼ 12.3.8 자동 미분을 사용하여 그레이디언트 계산하기

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

w1, w2 = 5, 3; eps = 1e-6
# 각 파라미터가 바뀔 때마다 함수의 출력이 얼마나 변하는지 측정하여 도함수의 근삿값을 계산함
print((f(w1 + eps, w2) - f(w1, w2)) / eps) # 36.000003007075065
print((f(w1, w2 + eps) - f(w1, w2)) / eps) # 10.000000003174137
```

- 자동 미분 사용해보기

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
gradients
# [<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
#  <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

▼ 12.3.9 사용자 정의 훈련 반복

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

• 훈련 세트에서 샘플 배치를 랜덤하게 추출하는 함수 만들기

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

• 현재 스텝 수, 전체 스텝 횟수, 에포크 시작부터 평균 손실, 그 외 다른 지표를 포함하여 훈련 상태를 출력하는 함수 만들기

```
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)
```

• 실제로 적용해보기

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

# 두 개의 반복문을 중첩한 것은 하나는 에포크를 위해서, 다른 하나는 에포크 안의 배치를 위한 것이다.
for epoch in range(1, n_epochs + 1):
    print('에포크 {}/{}'.format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train) # 훈련 세트에서 배치를 랜덤하게 샘플링한다.
        with tf.GradientTape() as tape: # 배치 하나를 위한 예측을 만들고 손실을 계산한다.
            y_pred = model(X_batch, training=True)
            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) # 테이프를 사용해 훈련 가능한 각 변수에 대한 손실의 그레이디언트를 계산한다
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics: # 다음 (현재 에포크에 대한) 평균 손실과 지표를 업데이트하고 상태 막대를 출력한다.
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics) # 다음 (현재 에포크에 대한) 평균 손실과 지표를 업데이트하고
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics: # 평균 손실과 지푯값을 초기화한다.
        metric.reset_states()
```

▼ 12.4 텐서플로 함수와 그래프

해라