Chapter12. 텐서플로를 사용한 사용자 정의 모델과 훈련

- 자신만의 손실 함수, 지표, 층, 모델, 초기화, 규제, 가중치 규제 등을 만들어 세부적으로 제어가 가능
- 그레디언트에 특별한 변환이나 규제 적용 가능
- 네트워크의 부분마다 다른 옵티마이저 사용 가능

1. 텐서플로 훑어보기

- 핵심 구조는 numpy와 매우 비슷하지만 **GPU를 지원**
- (여러 장치와 서버에 대해서) **분산 컴퓨팅**을 지원
- 일종의 JIT(just-in-time) 컴파일러를 포함. 속도를 높이고 메모리 사용량을 줄이기 위해 계산을 최적화. 이를 위해 파이썬 함수에서 **계산 그래프(computation graph)**를 추출한 다음 최적화하고(예를 들어, 사용하지 않는 node를 가지치기) 효율적으로 실행(예를 들어, 독립적인 연산을 자동으로 병렬 실행)
- 계산 그래프는 플랫폼에 중립적인 포맷으로 내보낼 수 있으므로 한 환경(예를 들어, 리눅스에 있는 파이썬)에서 텐서플로 모델을 훈련하고 다른 환경(예를 들어, 안드로이드 장치에 있는 자바)에서 실행할 수 있음
- 텐서플로는 자동 미분(autodiff) 기능과 RMSProp, Nadam 같은 고성능 Optimizer를 제공하므로 모든 종류의 손실 함수를 쉽게 최소화할 수 있음

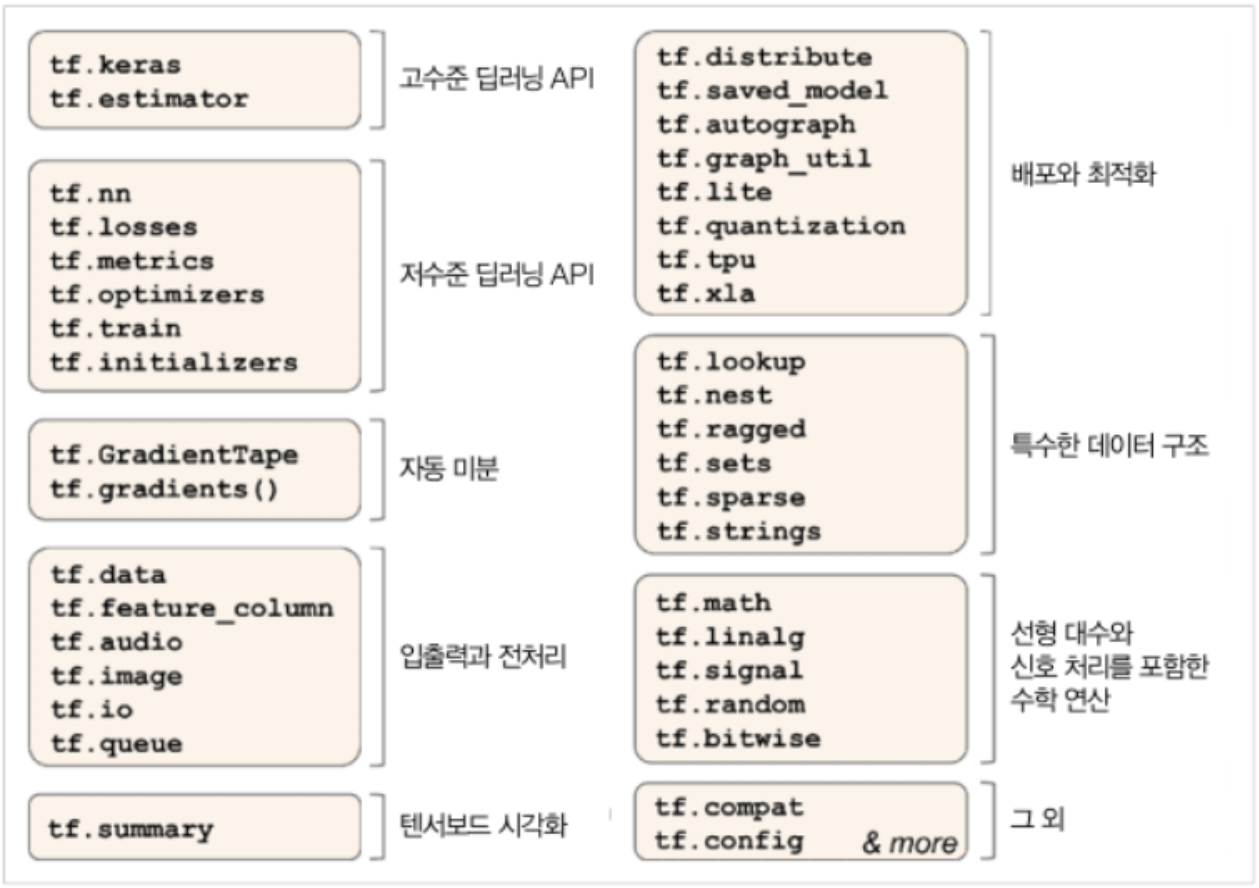

그림 12-1 텐서플로 파이썬 API

- 가장 중요한 tf.keras부터 데이터 적재와 전처리 연산(tf.data, tf.io 등), 이미지 처리 연산(tf.image), 신호 처리 연산(tf.signal)과 그 외 많은 기능을 가지고 있음

- 가장 저수준의 tensorflow operation(op)은 매우 효율적인 C++코드로 구현되어 있음
- 많은 연산은 **커널(kernel)** 이라 부르는 여러 구현을 가짐. 각 커널은 CPU, GPU, TPU(tensor processing unit)와 같은 특정 장치에 맞추어 만들어짐.
  - GPU는 계산을 작은 단위로 나누어 여러 GPU 스레드에서 병렬로 실행하므로 속도를 극적으로 향상
  - TPU는 딥러닝을 위해 특별하게 설계된 ASIC chip으로 GPU보다 빠른 연산이 가능

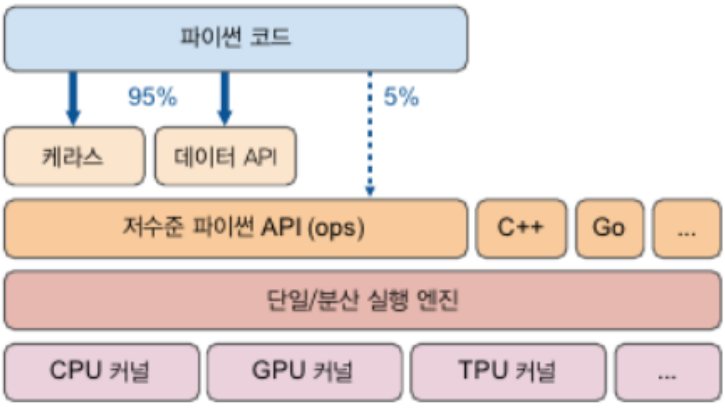

그림 12-2 텐서플로 구조

- 대부분 tensorflow의 고수준 API를 사용하지만, 더 높은 자유도가 필요한 경우 저수준 파이썬 API를 사용하여 텐서를 직접 다룰 수 있음. 파이썬 외에 다른 언어의 API도 제공.
- tensorflow의 실행 엔진은 여러 디바이스와 서버로 이루어진 분산 환경에서도 연산을 효율적으로 실행
- windows, linux, 맥os, (텐서플로 Lite를 사용하여) iOS, 안드로이드 같은 모바일 장치에서도 실행됨
- 파이썬 API, C++, java, Go, swift API 사용 가능
- TensorFlow.js의 자바스크립트 구현도 있음
  - 이를 사용하면 브라우저에서 직접 모델을 사용하는 것도 가능

텐서플로의 광범위한 라이브러리 생태계

- TensorBoard : 시각화
- TFX(TensorFlow Extended) : 구글에서 만든 텐서플로 제품화를 위한 라이브러리 모음
  - 데이터 시각화, 전처리, 모델 분석, serving 등이 포함됨
- TensorFlow Hub : 사전훈련된 신경망을 손쉽게 다운로드하여 재사용 가능

2. numpy 처럼 tensorflow 사용하기

- tensorflow API는 tensorflow를 순환시킴. tensor는 한 연산에서 다른 연산으로 흐름.
- tensor : 일반적으로 다차원 배열이지만, 스칼라 값도 가질 수 있음(numpy와 비슷)
  - 크기와 데이터 type를 가짐

- 대부분의 numpy의 연산이 가능하지만 일부 함수들은 numpy와 이름이 다름
  - np.reduce_mean(), tf.reduce_sum(), tf.reduce_max(), tf.math.log(), tf.transpose()
    - reduce알고리즘을 사용하여 원소가 추가된 순서를 보장하지 않고, 전치행렬의 경우 tensorflow에서는 복사본으로 새로운 텐서가 만들어지지만 numpy 에서는 그렇지 않기 때문
    - 32bit 부동소수는 제한된 정밀도를 가지므로 mean, sum 에서 실행마다 결과가 조금씩 달라질 수 있음
- 사용자 정의 손실 함수, 사용자 정의 지표, 사용자 정의 층 등을 만들 때 tensor가 중요

2.1 Tensor와 연산

In [1]:
```
import tensorflow as tf
```

In [2]:
```
tf.constant([[1,2,3],[4,5,6]]) # 행렬
```

Out[2]:
```
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 2, 3],
       [4, 5, 6]])>
```

In [3]:
```
tf.constant(42) # 스칼라
```

Out[3]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

In [5]:
```
t = tf.constant([[1,2,3],[4,5,6]])
print(t.shape)
print(t.dtype)
```

```
(2, 3)
<dtype: 'int32'>
```

In [14]:
```
print(t[:,1:])
print(t[...,1,tf.newaxis]) # 새로운 차원을 추가
```

```
tf.Tensor(
[[2 3]
 [5 6]], shape=(2, 2), dtype=int32)
tf.Tensor(
[[2]
 [5]], shape=(2, 1), dtype=int32)
```

In [16]:
```
print(t+10) # tf.add(t, 10)
print(tf.square(t))
print(t@tf.transpose(t)) # matrix 곱, tf.matmul()과 동일
```

```
tf.Tensor(
[[11 12 13]
 [14 15 16]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 1  4  9]
 [16 25 36]], shape=(2, 3), dtype=int32)
```

2.2 텐서와 넘파이

- 텐서는 넘파이와 함께 사용할 수 있음
  - 넘파이 배열로 텐서를 만들 수 있고, 그 반대도 가능
  - 넘파이 배열로 텐서플로 연산을 적용할 수 있고, 텐서에 넘파이 연산을 적용할 수도 있음
- 넘파이, 텐서플로 정밀도
  - 넘파이 : 64비트 정밀도
  - 텐서플로 : 32비트 정밀도(일반적인 신경망은 32비트 정밀도로도 충분하고, 더 빠르고 메모리를 적게 사용하기 때문. 따라서 넘파이 배열로 텐서를 만들 때, dtype=tf.float32로 지정해주어야 함

In [2]:
```
import numpy as np
```

In [16]:
```
# numpy를 tensor로, tensor를 numpy로 바꾸기
a = np.array([2.,4.,5.,])
print(tf.constant(a)) # float64가 되기 때문에 float32로 지정해주는 것이 필요
print(tf.constant(a, dtype=tf.float32))
print(t.numpy()) # or np.array(t)
```

```
tf.Tensor([2. 4. 5.], shape=(3,), dtype=float64)
tf.Tensor([2. 4. 5.], shape=(3,), dtype=float32)
[[1 2 3]
 [4 5 6]]
```

In [19]:
```
# 서로의 연산 적용
print(tf.square(a))
print(np.square(t))
```

```
tf.Tensor([ 4. 16. 25.], shape=(3,), dtype=float64)
[[ 1  4  9]
 [16 25 36]]
```

2.3 타입 변환

- 어떤 타입 변환도 자동으로 수행하지 않음
  - 타입 변환은 성능을 크게 감소시킬 수 있음. 또한 타입이 자동으로 변환되면 사용자가 눈치채지 못할 수 있기 때문
  - 호환되지 않는 타입의 텐서플로 연산을 실행하면 예외가 발생
    - 실수 텐서와 정수 텐서를 더할 수 없음
    - 32비트 실수와 64비트 실수를 더할 수 없음
- 타입 변환이 필요할 땐 tf.cast를 사용하면 됨

In [20]:
```
tf.constant(2.)+tf.constant(40) # 실수와 정수의 덧셈
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-20-38d828164b4c> in <module>
----> 1 tf.constant(2.)+tf.constant(40)

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\math_ops.py in
binary_op_wrapper(x, y)
   1122     with ops.name_scope(None, op_name, [x, y]) as name:
   1123       try:
-> 1124         return func(x, y, name=name)
   1125       except (TypeError, ValueError) as e:
   1126         # Even if dispatching the op failed, the RHS may be a tensor awar
e

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\util\dispatch.py i
n wrapper(*args, **kwargs)
    199     """Call target, and fall back on dispatchers if there is a TypeErro
r."""
    200     try:
--> 201       return target(*args, **kwargs)
    202     except (TypeError, ValueError):
    203       # Note: convert_to_eager_tensor currently raises a ValueError, not
 a

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\math_ops.py in
_add_dispatch(x, y, name)
   1444     return gen_math_ops.add(x, y, name=name)
   1445   else:
-> 1446     return gen_math_ops.add_v2(x, y, name=name)
   1447
   1448

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\gen_math_ops.p
y in add_v2(x, y, name)
    484       return _result
    485     except _core._NotOkStatusException as e:
--> 486       _ops.raise_from_not_ok_status(e, name)
    487     except _core._FallbackException:
    488       pass

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\framework\ops.py i
n raise_from_not_ok_status(e, name)
   6841   message = e.message + (" name: " + name if name is not None else "")
   6842   # pylint: disable=protected-access
-> 6843   six.raise_from(core._status_to_exception(e.code, message), None)
   6844   # pylint: enable=protected-access
   6845

C:\ProgramData\Anaconda3\lib\site-packages\six.py in raise_from(value, from_valu
e)

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

In [21]:
```
tf.constant(2.)+tf.constant(40, dtype=tf.float64)
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-21-f443af4ff38a> in <module>
----> 1 tf.constant(2.)+tf.constant(40, dtype=tf.float64)

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\math_ops.py in
binary_op_wrapper(x, y)
   1122     with ops.name_scope(None, op_name, [x, y]) as name:
   1123       try:
-> 1124         return func(x, y, name=name)
   1125       except (TypeError, ValueError) as e:
   1126         # Even if dispatching the op failed, the RHS may be a tensor awar
e

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\util\dispatch.py i
n wrapper(*args, **kwargs)
    199     """Call target, and fall back on dispatchers if there is a TypeErro
r."""
    200     try:
--> 201       return target(*args, **kwargs)
    202     except (TypeError, ValueError):
    203       # Note: convert_to_eager_tensor currently raises a ValueError, not
 a

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\math_ops.py in
_add_dispatch(x, y, name)
   1444     return gen_math_ops.add(x, y, name=name)
   1445   else:
-> 1446     return gen_math_ops.add_v2(x, y, name=name)
   1447
   1448

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\gen_math_ops.p
y in add_v2(x, y, name)
    484       return _result
    485     except _core._NotOkStatusException as e:
--> 486       _ops.raise_from_not_ok_status(e, name)
    487     except _core._FallbackException:
    488       pass

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\framework\ops.py i
n raise_from_not_ok_status(e, name)
   6841   message = e.message + (" name: " + name if name is not None else "")
   6842   # pylint: disable=protected-access
-> 6843   six.raise_from(core._status_to_exception(e.code, message), None)
   6844   # pylint: enable=protected-access
   6845

C:\ProgramData\Anaconda3\lib\site-packages\six.py in raise_from(value, from_valu
e)

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

In [22]:
```
t2 = tf.constant(40., dtype=tf.float64)
tf.constant(4.0)+tf.cast(t2, tf.float32)
```

Out[22]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=44.0>
```

2.4 변수

- tf.constant는 변경이 불가능한 객체.
  - 따라서 수정하고 싶거나 계산 결과는 새로운 constant에 저장해주어야 함.
- tf.Variable 을 이용하면 텐서의 내용을 바꿀 수 있음 -> 원본 객체에 바로 바뀜
  - 가능 메서드
    - assign() 메서드
      - assign_add() / assign_sub()
    - scatter_update() / scatter_nd_update()
  - gradient descent를 이용한 기울기 update와 같은 작업 수행 가능
- keras는 add_weight() 메서드로 변수 생성을 대신 처리해주기 때문에 실전에서 변수를 직접 만드는 일은 매우 적음.
  - 변수, 층이나 모델처럼 모든 추적이 가능한 객체를 관리

In [23]:
```
v = tf.Variable([[1.,2.,3.,],[4.,5.,6.]])
v
```

Out[23]:
```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

In [24]:
```
v.assign(2*v) # 2를 곱한 형태로 바뀜
```

Out[24]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

In [27]:
```
v[0,1].assign(42)
```

Out[27]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

In [28]:
```
v[:,2].assign([0.,1.])
```

Out[28]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  0.],
       [ 8., 10.,  1.]], dtype=float32)>
```

In [29]:
```
v.scatter_nd_update(indices = [[0,0],[1,2]], updates = [100,200])
```

Out[29]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[100.,  42.,   0.],
       [  8.,  10., 200.]], dtype=float32)>
```

2.5 다른 데이터 구조

- 다양한 데이터 구조를 지원한다.
- **sparse tensor(tf.SparseTensor)**
  - 대부분 0으로 채워진 텐서를 효율적으로 나타낸다. tf.sparse 패키지는 희소 텐서를 위한 연산을 제공한다
- **tensor array(tf.TensorArray)**
  - 텐서의 리스트. 즉, 각 원소가 텐서이다. 기본적으로 고정된 길이를 가지지만 동적으로 바꿀 수 있다. 리스트에 포함된 모드 텐서는 크기와 데이터 타입이 동일해야 한다.
- **ragged tensor(tf.RaggedTensor)**
  - 래그드 텐서는 텐서의 리스트의 리스트를 나타낸다. 텐서에 포함된 값은 동일한 데이터 타입을 가져야 하지만 리스트의 길이는 다를 수 있다. tf.ragged 패키지는 래그드 텐서를 위한 연산을 제공한다.
- **string tensor**
  - tf.string 타입의 텐서이다. Unicode가 아니라 바이트 문자열을 나타낸다. 유니코드 문자열(e.g., u"cafe")을 사용해 문자열 텐서를 만들면 자동으로 UTF-8로 인코딩 된다(e.g., b"caf/xc3/xa9"). 또는 유니코드 코드 포인트를 나타내는 tf.int32 텐서를 사용해 유니코드 문자열을 표현할 수 있다(e.g. [99, 97, 102, 233])
  - tf.strings 패키지를 사용하여 바이트 문자열과 문자 코드 문자열 간의 변환을 위한 연산을 제공. 문자열의 크기가 텐서 크기에 나타나지 않음. 유니코드 텐서(i.e., 유니코드 코드 포인트를 가진 tf.int32텐서)로 바꾸면 문자열의 크기가 텐서 크기에 표현됨
- **set**
  - 집합은 일반적인 텐서(or sparse tensor)로 나타낸다. 예를 들면, tf.constant([[1,2],[3,4]])는 두 개의 집합 {1,2}와 {3,4}를 나타낸다. 일반적으로 각 집합은 텐서의 마지막 축에 있는 벡터에 의해 표현된다. tf.sets 패키지의 연산을 사용해 집합을 다룰 수 있다.
- **queue**
  - queue는 단계별로 tensor를 저장한다. tensorflow는 여러 종류의 queue를 제공한다. 간단한 FIFO(first in, first out)queue(FIFOQueue), 어떤 원소에 우선권을 주는 큐(PriorityQueue), 원소를 섞는 queue(RandomShuffleQueue), 패딩을 추가하여 크기가 다른 원소의 배치를 만드는 큐(PaddingFIFOQueue) 등이 있다. 이 클래스들은 tf.queue 패키지에 포함되어 있다.

3. 사용자 정의 모델과 훈련 알고리즘

- 손실, 활성화 함수를 포함한 층, 모델 : call()
- 규제, 초기화, 제한 : __call__()
- tensorflow의 keras API는 현재 층, 모델, 콜백, 규제를 상속하는 방법만 정의하고 있다. 따라서 손실, 지표 metric, 초기화, 제한(constraint) 같은 상속을 통해 다른 요소를 만들려면 다른 kears 구현과 호환되지 않을 것이다.
  - 규제는 tf.constraints 패키지 / 스케줄링 방식은 tf.keras.optimizers.schedules 아래의 LearningRateSchedule 클래스를 상속하면 된다.

3.1 사용자 정의 손실 함수

- 잡음이 너무 많다면
  - MSE : 큰 오차에 너무 과한 벌칙을 가하기 때문에 정확하기 않은 모델이 만들어짐
  - MAE : 데이터에 이상치가 많은 경우 사용하면 좋지만, 큰 오차에 관대해서 훈련이 수렴되기까지 시간이 오래 걸림
  - Huber : 오차의 임계값(전형적으로 1)보다 작으면 MSE, 크면 MAP
    - $L_H(y - f(x), \delta)$
      - $(y - f(x))^2/2 \quad if|y - f(x)| <= \delta$

    - $\delta|y-f(x)|-\delta^2/2 \quad if iy-f(x)|r>\delta$
  - MSE 부분 : 1/2를 곱해줌으로써 기존 MSE에 비해 기울기가 완만한 손실함수, 이상치/잡음의 영향을 줄임
  - MAE 부분 : δ가 크다면, 이상치/잡음에 덜 민감한 것으로 해석, 작을 때에 비해, 더 가파른 그래프
  - 기

In [ ]:
```
# 샘플 하나하나 마다의 loss
# 전체 손실 평균보다 샘플마다의 손실을 담은 텐서를 반환하는 것이 좋음.
# 이렇게 해야 kears가 클래스 가중치나 샘플 가중치를 적용할 수 있음.

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)
```

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

In [ ]:
```
# 함수 이름은 마음대로 정의하면 됨
model = keras.models.load_moel("my_model_with_a_custom_loss.h5",
                               custom_objects={"huber_fn":huber_fn})
```

3.2 사용자 정의 요소를 가진 모델을 저장하고 로드하기

- 사용자 정의 객체를 포함한 모델을 load할 때, 함수 이름과 실제 함수를 매핑한 딕셔너리를 전달해야 함.
- 사용자 정의 요소(매개변수)는 저장되지 않기 때문에 모델을 새로 load할 때, 매개변수 값을 지정해 주어야 함

모델을 load할 때, 사용자 정의 요소가 load 되지 않는 경우

In [ ]:
```
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
```

In [ ]:
```
model.compile(loss=create_huber(2.0), optimizer= "nadam")
```

In [ ]:
```
# model을 저장할 때, threshold값이 저장되지 않았기 때문에, 모델을 load할 때, threshold값을 사
# 함수 이름은 새로 정의한 함수 이름이 아닌 저장한 keras model에 사용했던 함수 이름인 "huber_
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn":create_huber(2.0)})
```

모델을 load할 때, 사용자 정의 요소가 같이 load되는 경우

- init :
  - 부모클래스는 손실함수의 name과 개별 샘플의 손실을 모으기 위해 reduction 알고리즘을 사용
    - default : "sum_over_batch_size"
      - 개별 샘플 손실에 가중치를 곱하여 더하고 배치 크기로 나눔(가중치 합으로 나누지 않기 때문에 가중치 평균이 아님)
      - 가중치의 합으로 나누게 되면 배치마다 가중치 합이 달라져서 같은 가중치라도 반영되는 가중치 크기가 달라지는 문제가 발생하기 때문에 가중치 합으로 나누면 안됨.
      - 샘플 가중치가 없다면 1.0으로 간주
    - others
      - "sum" : 손실에 가중치를 곱하여 더한 값을 반환
      - "none" : 손실에 가중치를 곱한 배열을 반환
- call() : 레이블과 예측을 받고 모든 샘플의 손실을 계산하여 반환
- get_config() : 하이퍼파라미터 이름과 같이 매핑된 딕셔너리를 반환

In [ ]:
```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold = 1.0, **kwargs): # 매개변수에 threshold를 추가해줌
        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): # 설정에 threshold도 반환해줌
        base_config = super.get_config()
        return {**base_config, "threshold":self.threshold}
```

In [ ]:
```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

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

3.3 활성화 함수, 초기화, 규제, 제한을 커스터마이징 하기

사용자 정의 활성화 함수

- keras.activations.sofrplus() / tf.nn.softplus()와 동일

In [33]:
```
def my_softplus(z):
    return tf.math.log(tf.exp(z)+1.0)
```

Out[33]:
```
0.7310381925766541
```

글로럿 초기화

- keras.initializers.glorot_normal()과 동일

In [34]:
```
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)
```

Out[34]: 0.2689618074233459

L1 규제

- keras.regularizers.l1(0.01)과 동일

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

양수인 가중치만 남기는 사용자 정의 제한

- keras.constraints.nonneg() / tf.nn.relu()와 동일

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

사용

In [ ]:
```
layer = keras.layers.Dense(30,
                           activation = my_softplus,
                           kernel_initializer = my_glorot_initializer,
                           kernel_regularizer = my_l1_regularizer,
                           kernel_constraint = my_positive_weights)
```

-> 초기화 함수에 의해 초기화, activation function을 통해 값을 출력하고 다음층에 결과를 전달. 마지막 층까지 계산이 되고, 가중치가 규제 함수에 전달되어 규제 손실을 계산하여 훈련을 위한 최종 손실 생성. 마지막으로 제한 함수가 훈련 스텝마다 호출되어 층의 가중치를 제한한 가중치 값으로 바뀜

사용자 정의 요소를 가지고 있는 경우

In [ ]:
```
# 부모 클래스에 생성자와 get_config() 메서드가 정의되어 있지 않기 때문에 호출할 필요가 없음
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}
```

3.4 사용자 정의 지표

- 사용자 지표 함수와 사용자 손실 함수를 만드는 것은 동일
- 손실 : 모델을 훈련하기 위해 gradient descent에서 사용하므로 미분 가능해야 하고, gradient가 모든 곳에서 0이 아니어야 함. 사람이 이해할 수 없어도 괜찮다.(e.g., cross-entropy-error)
- 지표 : 모델을 평가할 때 사용하므로 미분 가능하지 않아도 괜찮고, gradient가 0이어도 괜찮음. (e.g., accuracy, recall, precision)
  - metrics클래스의 모든 지표는 sample_weight를 지원
  - 배치마다 점진적으로 업데이트 되는 지표 : streaming metric
    - 즉, 현재 배치에서의 지표가 아닌, 지금까지의 전체 지표
    - 한 에포크의 지표를 streaming metric으로 계산
      - 대부분의 문제는 배치마다의 지표를 단순히 저장하고 평균함으로써 수행가능하지만, precision과 같은 것은 단순히 평균 낼 수 없음

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

Streaming Precision

In [40]:
```
precision = tf.keras.metrics.Precision()
precision([0,1,1,1,0,1,0,1],[1,1,0,1,0,1,0,1])
precision([0,1,0,0,1,0,1,1],[1,0,1,1,0,0,0,0])
```

Out[40]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
```

In [44]:
```
# 현재 지표값을 얻는 방법
precision.result()
```

Out[44]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
```

In [45]:
```
# variables 속성을 사용하여 (진짜 양성과 거짓 양성을 기록한) 변수를 확인할 수도 있음
precision.variables
```

Out[45]:
```
[<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)>]
```

In [49]:
```
# result와 variables를 초기화
precision.reset_states()
```

후버손실 지표

- 생성자 : 여러 배치에 걸쳐 지표의 상태를 기록하기 위해 변수 생성(add_weight 메서드를 이용)
  - total : hubo loss의 총합
  - count : 지금까지 처리한 샘플 수
  - tf.Variable으로 직접 수동으로 변수를 만들 수도 있음
- 사용할 때마다 호출됨. 배치의 레이블과 예측을 바탕으로 변수를 업데이트(샘플 가중치가 있다면 이 또한 반영)
- 최종 결과를 계산하고 반환. update_state()->result()를 통해 출력이 반환됨

- get_config() 메서드를 통해 threshold 변수를 모델과 함께 저장
- reset_state : 모든 변수를 0으로 초기화. 이를 통해 함수를 재정의(override)할 수 있음

In [51]:
```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # 기본 매개변수 처리
        self.threshold = threshold
        self.hubor_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer = "zeros") # 후버 손실의 합
        self.count = self.add_weight("count", initializer = "zeros") # 지금까지 처리한 샘플
    def update_state(self, y_true, y_pred, sample_weight = None):
        metric = self.huber_fn(y_true, y_pred) # huber loss 계산
        self.total.assign_add(tf.reduce_sum(metric)) # 손실을 더해줌
        self.count.assign.add(tf.cast(tf.size(y_true), tf.float32)) # 처리한 샘플 수를 더해
    def result(self):
        return self.total/self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold":self.threshold}
```

Out[51]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.0>
```

3.5 사용자 정의 층

- tensorflow에는 특이한 층을 가진 네트워크를 만들 수 있음
- ex) ABC ABC ABC -> DDD (D=ABC)

파이썬 함수를 만든 후, layers.Lambda 층으로 감싸는 방법

- 시퀀셜 API, 함수형 API, 서브클래싱 API에서 보통의 층과 동일하게 사용 가능
- 활성화 함수로 사용 가능

지수 함수

- 지수 함수를 적용하는 layer
- 활성화 함수로도 사용 가능
  - activation = keras.activations.exponential / activation = "exponential" 과 동일
  - 예측값의 스케일이 매우 다를 때 출력층에 사용됨

In [ ]:
```
exponential_layer = tf.keras.layers.Lambda(lambda x: tf.exp(x)) # exponential을 수행하는 층
```

class를 상속 받는 방법 - 상태가 있는 층(즉, 가중치를 가진 층) 생성 가능

- keras.layers.Layer 상속을 통해 구현

- keras.activation.get() / keras.initializers.get() 을 통해 문자열을 적절한 활성화 함수로 변경
- build()
  - 변수를 생성하는 층(update 됨)

    - 층이 처음 사용될 때 호출. 따라서 층의 입력 크기를 알고 있기 때문에, 이를 매개변수로 하여 가중치를 생성.
- call()
    - 층에 필요한 연산을 수행
- compute_output_shape()
    - 층의 출력 크기를 반환.
    - 동적인 층을 제외하고 tf.keras가 자동으로 출력 크기를 추측할 수 있다면 생략 가능. 그렇지 않다면, 필수적이거나 출력 크기가 입력 크기와 동일하다고 가정

In [ ]:
```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation = None, initializer = "glorot_normal", **kwargs):
        super().__init__(**kwargs)
        self.units = units
        # 문자열 / tf function을 통해 활성화 함수 / 초기화 지정
        self.activation = keras.activations.get(activation) # 문자열을 받아서 적절한 활성화
        self.initializer = keras.initializers.get(initializer)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name = "kernel", shape = [batch_input_shape[-1], self.units],
            initializer = self.initializer)
        self.bias = self.add_weight(
            name = "bias", shape=[self.units], initializer = "zeros")
        # 이를 통해 layer가 만들어졌다는 것을 keras가 인식(self.built = True)
        super.build(batch_input_shape) # 반드시 끝에서 호출해야 함.

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

    def compute_output_shape(self, batch_input_shape):
        # tf.shape를 통해 구한 shape를 리스트로 바꾼 후, units 리스트와 이어붙임
        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)}
```

2개의 입력과 3개의 출력

- matrix(5x3)이 2개인 입력 -> (matrix1(5x3),matrix2(5x3))
- matrix(5x3)이 3개인 출력 -> (output1(5x3),output2(5x3),output3(5x3))

In [53]:
```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        # 2개의 입력
        X1, X2 = X
        return [X1+X2, X1*X2, X1/X2]

    def compute_output_shape(self,batch_input_shape):
        # 3개의 출력
        b1,b2 = batch_input_shape
        return [b1,b1,b1]
```

Out[53]:
```
<tensorflow.python.keras.initializers.initializers_v2.GlorotNormal at 0x2749ecf5370>
```

훈련과 테스트에서 다르게 동작하는 층

- Dropout / BatchNormalization 등과 같이 훈련과 테스트에서 다르게 동작하는 층이 필요하다면 call() 메서드에 training 매개변수를 추가하여야 함

훈련하는 동안 가우스 잡음을 추가하고 테스트 시에는 아무것도 하지 않는 층

- keras.layers.GaussianNoise() 와 동일

In [59]:
```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training = None):
        # 업데이트 되어야할 변수가 아닌, 매 training마다 랜덤으로 생성되는 값이기 때문에 bui
        if training:
            noise = tf.random.normal(tf.shape(X), stddev = self.stddev)
            return X+noise
        else:
            return X

    def comput_output_shape(self, batch_output_shape):
        return batch_output_shape
```

Out[59]:
```
[2, 3]
```

3.6 사용자 정의 모델

- SubClassing API : class 상속을 통해 사용자 정의 모델 class를 만드는 것
  - 큰 유연성을 가지지만 모델의 구조가 call 메서드 안에 숨겨져 있기 때문에 keras가 이를 분석하기 힘듦
- 생성자에서 층과 변수를 만들고, 모델이 해야 할 작업을 call() 메서드에 구현
- 구현한 모델을 compile, fit, evaluate, predict 에 사용할 수 있음
- save() 메서드를 사용해 모델을 저장하고, 로드하고 싶다면, get_config() 메서드를 구현해야 한다.
- 또한 save_weights()와 load_weights() 메서드를 사용해 가중치를 저장하고 로드할 수 있다

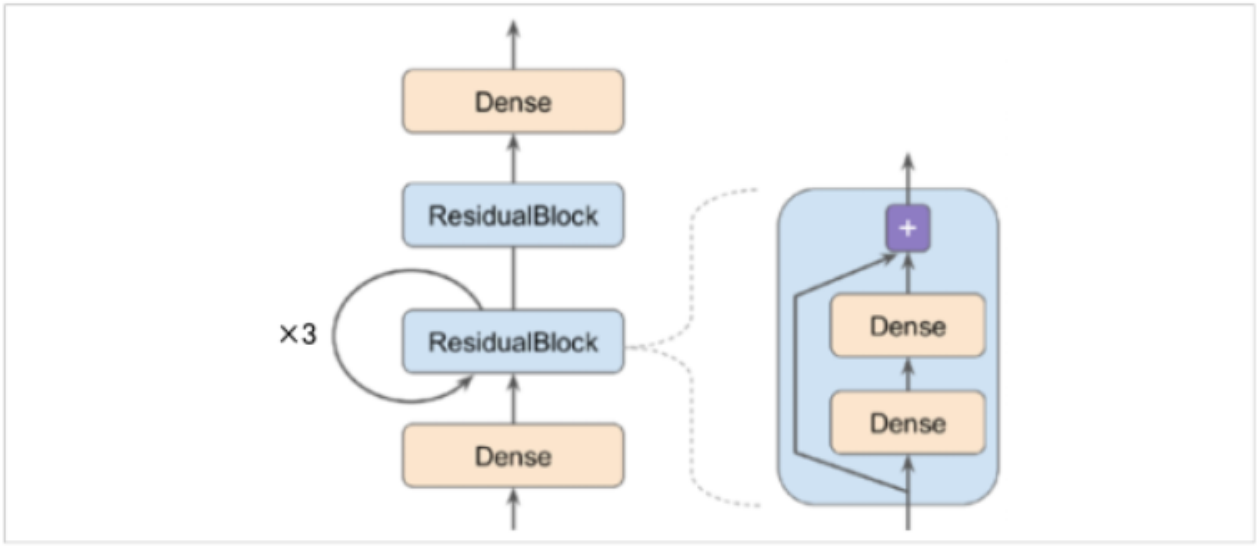

그림 12-3 사용자 정의 모델: 스킵 연결이 있는 사용자 정의 잔차 블록(ResidualBlock) 층을 가진 예제 모델

ResidualBlock layer 생성

- 다른 층을 포함하고 있음

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

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

모델 정의

In [61]:
```
class ResidualRegressor(keras.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): # 1번 하고 그리고 3번 더 반복
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

3.7 모델 구성 요소에 기반한 손실과 지표

- 예측값과 레이블을 이용한 것이 아닌, 은닉층의 가중치나 활성화 함수 등과 같이 모델의 구성 요소에 기반한 손실함수를 정의
- 규제나 모델의 내부 상황을 모니터링할 때 유용
- add_loss() / add_metric() 메서드를 사용해서 손실과 지표를 추가해줌

Reconstruction loss(재구성 손실)을 가지는 모델

- 5개의 은닉층과 출력층으로 구성된 회귀용 MLP 모델
- 맨 위의 은닉층(맨 마지막 은닉층)에 보조 출력을 가짐. 이 보조 출력에 연결된 손실을 **재구성 손실 (reconstruction loss)** 라고 부름
- 재구성과 입력 사이의 MSE
- 모델이 은닉층을 통과하면서 가능한 많은 정보를 유지하도록 유도.
- 이따금 규제손실처럼 작동하여 일반화 성능을 향상시킴

- 생성자에서 layer을 생성
- build 에서 reconstruction layer 생성
  - input layer의 크기를 알아야 reconstruction layer을 생성할 수 있기 때문
- call 메서드에서 hidden layer을 통과하고 이를 reconstruction layer에 전달, reconstruction layer에서 loss를 계산하고 이를 loss 리스트에 추가. 손실이 계산될 때, predicted, target의 손실과 reconstruction loss x 0.05가 더해진 손실이 계산됨
- 마지막 은닉층의 출력을 출력층에 전달하여 얻은 출력값을 반환

In [ ]:
```
class ReconstructingRegressor(keras.Model):
    def __init__(self, ouput_dim, **kwargs):
        # layer을 생성
        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)

    def build(self, batch_input_shape):
        # 입력의 크기를 알아야 reconstruction loss로의 hidden layer을 생성할 수 있기 때문에
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layer.Dense(n_inputs)
        super().build(batch_input_shape) # 반드시 맨 마지막에 호출해야함!!!

    def call(self, inputs):
        Z = inputs
        # layer가 실행되면서 call을 실행시킴
        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) # 모델의 손실 리스트에 추가
        self.add_metric(recon_loss) # 모델의 지표 리스트에 추가
        return self.out(Z)
```

- 사용자 정의 지표 출력 예시
- 전체 손실(recostructx0.05+output loss), reconstruction loss

```
Epoch 1/5
11610/11610 [==============] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [==============] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

3.8 자동 미분을 사용하여 그레디언트 계산하기

In [68]:
```
def f(w1, w2):
    return 3*w1**2+2*w1*w2
```

도함수를 구하는 첫번째 방법

- 파라미터가 바뀔 때마다 함수의 출력이 얼마나 변하는지 측정하여 도함수의 근삿값을 계산하는 방법
- 정확한 값이 아닌 근삿값이고, 파라미터마다 적어도 한번씩 f()를 출력하므로 시간이 많이 걸림

In [69]:
```
w1, w2 = 5,3
eps = 1e-6
print((f(w1+eps,w2)-f(w1,w2))/eps)
print((f(w1,w2+eps)-f(w1,w2))/eps)
```

```
36.000003007075065
10.000000003174137
```

자동미분

- tf.GradientTape를 열면 모든 연산을 자동으로 기록
  - with 문을 빠져나가면 연산을 더이상 기록하지 않고, 기록된 내용은 남아있음(다른 with를 사용했을 때랑 달리)
  - tape.gradient 를 실행하면 자동으로 테이프가 지워짐(즉 두번 호출하면 예외 발생)
  - tf.GradientTape(persistent=True) 를 통해 지속 가능한 tape 생성 가능
  - with tf.GradientTape() as tape
    - 이 때, tape 외에 다른 변수는 불가능
  - with tape.stop_recording() 을 통해 계산을 기록하지 않을 수도 있음
    - 메모리 절약을 위해 블록 안에 최소한만 담는 것이 좋음

- tape는 변수가 포함된 연산만을 기록
  - 변수가 아닌 다른 객체에 대한 z의 그레디언트를 계산하려면 None이 반환
  - 필요시 어떤 tensor라도 감시하여 모든 연산을 기록하도로 강제 -> 어떤 텐서에 대해서도 그레디언트를 계산할 수도 있음

- 후진 모드 자동 미분(reverse-mode autooff) : 한 번의 정방향 계산과 역방향 계산으로 모든 그레디언트를 동시에 계산
  - 벡터의 그레디언트를 계산하면 벡터의 합의 그레디언트를 계산

- 개별 그레디언트를 계산하고 싶다면, jacobian() 메서드를 호출해야 함
- Hessian matrix를 계산할 수도 있음(second-order partial derivative)
  - 메모리, 속도 측면에서 힘들기 때문에 실제로는 잘 사용하지 않음

- 신경망의 일부분에 그레디언트가 역전파 되지 않도록 막고 싶은 경우 : tf.stop_gradient()

- 함수의 gradient를 계산하는 것이 수치적으로 불안정하면 부동소수점 정밀도 오류로 인해 자동 미분이 무한 나누기 무한을 계산하게 됨
  - 지수 함수 등으로 인해 값이 너무 커진 경우(inf)
- 따라서 수치적으로 안전한 softplus의 도함수를 해석적으로 구하고 @tf.custom_gradient decorator를 사용한하여 일반 출력과 도함수를 계산하는 함수를 반환하여 tensorflow가 softmax plus() 함수의 gradient를 계산할 때 안전한 함수를 사용하도록 만듦

In [77]:
```
w1,w2 = tf.Variable(5.), tf.Variable(3.) # 변수를 선언해주어야 함

# tf.GradientTape는 열면 모든 연산을 자동으로 기록
# with 문을 빠져나가면서 연산을 기록하지 않지만 tape에 그 내용이 남아있음
# gradient 를 실행하면 테이프가 즉시 지워짐
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
gradients
```

Out[77]:
```
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

In [78]:
```
# 두번 호출 시 예외 발생
gradients = tape.gradient(z, [w1, w2])
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-78-95adbff8d903> in <module>
----> 1 gradients = tape.gradient(z, [w1, w2])

C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\eager\backprop.py
 in gradient(self, target, sources, output_gradients, unconnected_gradients)
   1012     """
   1013     if self._tape is None:
-> 1014       raise RuntimeError("GradientTape.gradient can only be called once
on "
   1015                          "non-persistent tapes.")
   1016     if self._recording:

RuntimeError: GradientTape.gradient can only be called once on non-persistent tapes.
```

In [79]:
```
# 계속 gradient를 반환할 수 있도록 변경
with tf.GradientTape(persistent=True) as tape:
    z = f(w1,w2)

dz_dw1 = tape.gradient(z,w1)
dz_dw2 = tape.gradient(z,w2)
del tape # 테이프 객체 해제
```

In [80]:
```
print(dz_dw1)
print(dz_dw2)
```

```
tf.Tensor(36.0, shape=(), dtype=float32)
tf.Tensor(10.0, shape=(), dtype=float32)
```

In [81]:
```
# 변수가 아닌 객체의 gradient 값은 None
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

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

```
[None, None]
```

In [83]:
```
# 변수가 아닌 객체의 gradient를 계산할 수 있도록 변경
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1,c2)

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

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

In [85]:
```
# 신경망의 일부분에 gradient가 역전파되지 않도록 막음
def f(w1, w2):
    return 3*w1**2 + tf.stop_gradient(2*w1*w2)

with tf.GradientTape() as tape:
    z = f(w1,w2) # tf.stop_gradient를 쓰지 않았을 때와 동일

gradients = tape.gradient(z, [w1,w2])
print(gradients)
```

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

In [104]:
```
# 함수의 gradient를 계산하는 것이 수치적으로 불안정할때
# 값이 너무 크기 때문에 무한 나누기 무한을 계산하게 됨

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

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

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

Out[104]:
```
[<tf.Tensor: shape=(1,), dtype=float32, numpy=array([nan], dtype=float32)>]
```

In [87]:
```
# 따라서 수치적으로 안전한 softplus의 도함수를 해석적으로 구하고
# @tf.custom_gradient decorator를 사용
# 일반 출력과 도함수를 계산하는 함수를 반환하여
# tensorflow가 softmax plus() 함수의 gradient를 계산할 때 안전한 함수를
# 사용하도록 만듦
@ tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1+1/exp)
    return tf.math.log(exp+1), my_softplus_gradients
```

3.9 사용자 정의 훈련 반복

- 두 개의 optimizer를 사용하는 Wide and Deep Model의 경우 사용
- 의도한 대로 잘 동작하는지 확신을 갖기 위해 사용
- 길고, 버그가 발생하기 쉽고, 유지 보수하기 어려운 코드가 만들어짐
- 훈련과 반복을 직접 다루기 때문에 컴파일할 필요가 없음

In [ ]:
```
l2_reg = keras.regularizers.l2(0.05)
model
```

In [103]:
```
my_softplus(np.array([100.]))
```

Out[103]:
```
<tf.Tensor: shape=(1,), dtype=float64, numpy=array([100.])>
```

In [107]:
```
np.exp(100, dtype="float32")
```

```
<ipython-input-107-b88911773d51>:1: RuntimeWarning: overflow encountered in exp
  np.exp(100, dtype="float32")
```

Out[107]:
```
inf
```

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

샘플 배치를 랜덤하게 추출하는 작은 함수

- 중복을 허용하지 않고 샘플링 하려면 np.random.permutation(np.arange(len(X))으로 랜덤하게 섞인 index를 만든 후 yield 문을 사용하여 미니배치 크기만큼 순서대로 데이터를 반환

In [ ]:
```
# 중복을 허용해서 샘플링하는 방법
def random_batch(X,y,batch_size = 32):
    idx = np.random.randint(len(X), size = batch_size)
    return X[idx], y[idx]
```

In [ ]:
```
# 상태바를 출력하는 함수
def print_status_bar(iteration, total, loss, metrics = None):
    # metrics가 None이면 [], 원소가 존재하는 리스트이면 metrics로 하당됨
    metrics = " - ".join(["{}: {:.4f}".format(m.name,m.result())
                          for m in [loss]+(metrics or [])])
    end = "" if iteration < total else "₩n" # 마지막 iteration이면 ₩n, 그렇지 않으면 ""을 e
    print("₩r{}/{} - ".format(iteration, total)+metrics,
          end=end)
```

\r(캐리지 리턴) : 상태막대가 동일한 줄에 출력된다

- 동일한 기능을 하는 다른 함수/라이브러리
  - print_status_bar()
  - tqdm

In [19]:
```
from tensorflow import keras
```

In [20]:
```
import tensorflow.keras
```

In [22]:
```
# hyperparameter, loss function, metrics 선택
n_epochs = 5
batch_size = 32
learning_rate = 0.01
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=learning_rate)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

In [14]:
```
for epoch in range(1,n_epochs+1): # 에포크 반복
    print("에포크 {}/{}".foramt(epoch, n_epochs))
    for step in range(1, n_steps+1): # 한 에포크 안의 배치를 위한 반복
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            # loss_fn(mean_squared_error)가 샘플마다 하나의 손실을 반환 -> 평균을 구해야함
            # 만약 샘플마다 다른 가중치를 적용하려면 이 단계에서 적용해야 함
            main_loss = tf.reduce_mean(loss_fn(y_batch,y_pred))
            # 리스트의 원소 중 동일한 크기와 타입을 가진 tensor를 더함
            # main 손실에 규제 손실과 같은 다른 손실도 더해줌
            loss = tf.add_n([main_loss]+model.losses)
        gradients = tape.gradient(loss, model.trainable_variables) # loss에 대한 훈련 가능한
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss) # 한 에포크를 기준으로 loss를 평균냄
        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)

    # 한 epoch를 다 돌면 loss와 metrics를 초기화함
    for metric in [mean_loss]+metrics:
        metric.reset_states()
```

Out[14]: 5

만약 kernel_constraint / bias_constraint를 지정하여 모델에 가중치 제한을 추가하면

In [ ]:
```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

12.4 텐서플로 함수와 그래프

- tf.function()을 사용하여 텐서플로 함수로 바꿀 수 있음
  - 원래 파이썬 함수처럼 사용할 수 있고 동일한 결과를 반환하지만 그 형태가 tensor
  - 함수에서 수행되는 계산을 분석하고 동일한 작업을 수행하는 계산 그래프를 생성
  - 원본 파이썬 함수는 파이썬함수.python_function() 속성으로 참조 가능
- 텐서플로 함수의 최적화
  - 사용하지 않는 노드를 제거하고 표현을 단순화(e.g. 1+2 -> 3)하는 등의 방식으로 계산 그래프를 최적화
  - 최적화된 그래프를 적절한 순서에 맞춰 (가능하면 병렬로) 그래프 내의 연산을 효율적으로 실행
  - 일반적으로 텐서플로 함수는 원본 파이썬 함수보다 훨씬 빠르게 실행됨(특히 복잡한 연산을 수행할 때)
- 사용자 정의 함수(사용자 정의 손실, 지표, 층 등)를 작성하고 케라스 모델에 사용할 때, 자동으로 텐서플로 함수로 변환하기 때문에 tf.function()을 사용할 필요 X
  - 만약 바꾸고 싶지 않다면, dynamic=True 로 작성, 혹은 모델의 compile() 메서드를 호출할 때 run_eagerly=True로 지정
- 텐서플로 함수는 입력 크기와 데이터 타입에 맞춰 매번 새로운 그래프를 재사용. polynorphism(다형성, 다양한 매개변수 타입과 크기)를 처리. 이는 매개변수 값으로 텐서를 사용했을 때만 해당

- 예를 들어 tf_cube(tf.constant(10))을 호출하면 []크기의 int32 텐서에 맞는 그래프가 생성. 그다음 tf_cube(tf.constant(20))을 호출하면 동일한 그래프가 재사용. 하지만 tf_cube(tf.constant([10,20])을 호출하면 [2] 크기의 int32 텐서에 맞는 새로운 그래프가 생성
- tf_cube(10), tf_cube(20)은 두개의 그래프

In [ ]:
```
def cube(x):
    return x**3
```

In [30]:
```
cube(2)
```

Out[30]: 8

In [32]:
```
tf_cube = tf.function(cube)
tf_cube
```

Out[32]: <tensorflow.python.eager.def_function.Function at 0x1b383346100>

In [33]:
```
tf_cube(3)
```

Out[33]: <tf.Tensor: shape=(), dtype=int32, numpy=27>

In [35]:
```
tf_cube(tf.constant(2.0))
```

Out[35]: <tf.Tensor: shape=(), dtype=float32, numpy=8.0>

In [ ]:
```
# tf.fuunction을 사용하는 또 다른 방법
@tf.function
def tf_cube(x):
    return x**3
```

In [37]:
```
tf_cube.python_function(2)
```

Out[37]: 8

4.1 오토그래프와 트레이싱

1. autograph : 여러 제어문(e.g. for문, while문, if, break, continue, return 등)을 모두 찾음. 그 이후 모든 제어문을 텐서플로 연산으로 바꾼 업그레이드된 버전을 생성.
2. tracing : symbolic tensor를 전달하여 함수를 호출, graph model로 실행되고 최종적으로 그래프를 생성. 이 때 어떠한 계산도 수행되지 않음.
   - symbolic tensor : 실제 값은 없고 이름, 데이터 타입, 크기만을 가진 tensor
     - function(tf.constant(10)) -> 크기가 []이고, data type이 int32인 symbolic tensor를 사용해 호출
   - graph mode : 텐서플로 연산이 해당 연산을 나타내고 텐서를 출력하기 위해 그래프에 노드를 추가
     - eager execution(즉시 실행)과 반대

4.2 텐서플로 함수 사용방법

1. numpy나 표준 library를 포함해서 다른 라이브러리를 호출하면 tracing 과정에서 실행됨. 이 호출은 그래프에 포함되지 않음. 실제 텐서플로 그래프는 텐서플로 구성요소(텐서, 연산, 변수, 데이터셋 등)만 포함할 수 있음. 따라서 (tracing 과정에서 코드가 실행되는 것을 원하지 않는다면) np.sum() 대신에 tf.reduce_sum()을, sorted() 내장함수 대신에 tf.sort()와 같이 사용
    - np.random.rand()를 반환하는 함수의 경우 f(tf.constant(2.))와 f(tf.constant(3.))가 같은 난수를, f(tf.constant([2.,3.])은 다른 난수를 반환할 것임. 따라서 tf.random.uniform([])으로 바꾸면 **이 연산이 그래프의 일부분**이 되므로 호출할 때마다 난수가 생성될 것
    - 텐서플로에서 지원하지 않는 코드가 부수적인 작업을 하면 함수를 트레이싱할 때만 호출되므로 텐서플로 함수를 호풀할 때 이 코드가 실행되지 않음
    - 어떤 임의의 코드를 tf.py_function()으로 감쌀 수 있음. 하지만 최적화를 수행할 수 없어 성능이 저하됨. 또한 파이썬이 가능한, 그리고 필요한 라이브러리가 설치된 플랫폼에서만 이 그래프가 실행되므로 이식성이 낮아짐
2. 다른 파이썬 함수나 텐서플로 함수를 호출할 수 있음. 하지만 텐서플로가 계산 그래프에 있는 이 함수들의 연산을 감지하므로 모두 텐서플로 함수로 바뀜. 따라서 tf.function 데코레이터를 적용할 필요가 없음
3. 함수에서 텐서플로 변수 (또는 데이터셋이나 큐와 같은 상태가 있는 다른 텐서플로 객체)를 만든다면 처음 호출될 때만 수행되어야 함. 아니면 예외가 발생. 일반적으로 텐서플로 함수밖에서 변수를 생성하는 것이 좋음(e.g. build() 메서드). 변수의 새로운 값을 할당하려면 = 연산자 대신에 assign() 메서드를 사용해야 함
4. 파이썬 함수의 소스 코드는 텐서플로에서 사용 가능하여야 함. 만약 소스 코드를 사용할 수 없다면 (e.g. 소스 코드에 접근할 수 없는 python cell에서 함수를 정의/컴파일된 *.pyc파이썬 파일을 상용 환경에 배포한다면) 그래프 생성 과정이 실패하거나 일부 기능을 사용할 수 없음
5. 텐서플로는 텐서나 데이터셋을 순회하는 for문만 감지할 수 있음. 따라서 for i in range(x) 대신 for i in tf.range(x)를 사용하여야 함. 그렇지 않으면 이 반복문이 그래프에 표현되지 못하고, 트레이싱 단계에서 실행.(신경망의 층을 반복문을 만드는 경우, 일부러 이런 for문을 사용해 그래프를 만들기도 함)
6. 성능면에서는 반복문보다 가능한 한 벡터화된 구현을 사용하는 것이 좋음