02. Low-Level TensorFlow API

In this notebook you will learn how to use TensorFlow's low-level API, then use it to build custom loss functions, as well as custom Keras layers and models.

Imports

In [1]:
```
%matplotlib inline
```

In [2]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import sklearn
import sys
import tensorflow as tf
from tensorflow import keras
import time
```

In [3]:
```
print("python", sys.version)
for module in mpl, np, pd, sklearn, tf, keras:
    print(module.__name__, module.__version__)
```

```
python 3.6.8 |Anaconda, Inc.| (default, Dec 30 2018, 01:22:34)
[GCC 7.3.0]
matplotlib 3.0.2
numpy 1.15.4
pandas 0.24.0
sklearn 0.20.2
tensorflow 2.0.0-dev20190124
tensorflow.python.keras.api._v2.keras 2.2.4-tf
```

In [4]:
```
assert sys.version_info >= (3, 5) # Python ≥3.5 required
assert tf.__version__ >= "2.0"    # TensorFlow ≥2.0 required
```

Tensors and operations

You can browse through the code examples or jump directly to the exercises.

Tensors

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

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

In [6]:
```
t.shape
```

Out[6]:
```
TensorShape([2, 3])
```

In [7]:
```
t.dtype
```

Out[7]:
```
tf.float32
```

Indexing

In [8]:
```
t[:, 1:]
```

Out[8]:
```
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
```

In [9]:
```
t[..., 1, tf.newaxis]
```

Out[9]:
```
<tf.Tensor: id=10, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Ops

In [10]:
```
t + 10
```

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

In [11]:
```
tf.square(t)
```

Out[11]:
```
<tf.Tensor: id=15, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
```

In [12]:
```
t @ tf.transpose(t)
```

Out[12]:
```
<tf.Tensor: id=19, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

To/From NumPy

In [13]:
```
t.numpy()
```

Out[13]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

In [14]:
```
a = np.array([[1., 2., 3.], [4., 5., 6.]])
tf.constant(a)
```

Out[14]:
```
<tf.Tensor: id=22, shape=(2, 3), dtype=float64, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]])>
```

In [15]:
```
t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
np.square(t)
```

Out[15]:
```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

Scalars

In [16]:
```
t = tf.constant(2.718)
t
```

Out[16]:
```
<tf.Tensor: id=26, shape=(), dtype=float32, numpy=2.718>
```

In [17]:
```
t.shape
```

Out[17]:
```
TensorShape([])
```

In [18]:
```
t.numpy()
```

Out[18]:
```
2.718
```

Conflicting Types

In [19]:
```
try:
    tf.constant(1) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
cannot compute Add as input #0(zero-based) was expected to be a float tensor b
ut is a int32 tensor [Op:Add] name: add/
```

In [20]:
```
try:
    tf.constant(1.0, dtype=tf.float64) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
cannot compute Add as input #0(zero-based) was expected to be a float tensor b
ut is a double tensor [Op:Add] name: add/
```

In [21]:
```
t = tf.constant(1.0, dtype=tf.float64)
tf.cast(t, tf.float32) + tf.constant(1.0)
```

Out[21]:
```
<tf.Tensor: id=36, shape=(), dtype=float32, numpy=2.0>
```

Strings

In [22]:
```
t = tf.constant("café")
t
```

Out[22]:
```
<tf.Tensor: id=38, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

In [23]:
```
tf.strings.length(t)
```

Out[23]:
```
<tf.Tensor: id=40, shape=(), dtype=int32, numpy=5>
```

In [24]:
```
tf.strings.length(t, unit="UTF8_CHAR")
```

Out[24]:
```
<tf.Tensor: id=42, shape=(), dtype=int32, numpy=4>
```

In [25]:
```
tf.strings.unicode_decode(t, "UTF8")
```

Out[25]:
```
<tf.Tensor: id=47, shape=(4,), dtype=int32, numpy=array([ 99,  97, 102, 233],
dtype=int32)>
```

String arrays

In [26]:
```
t = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
```

In [27]:
```
tf.strings.length(t, unit="UTF8_CHAR")
```

Out[27]:
```
<tf.Tensor: id=50, shape=(4,), dtype=int32, numpy=array([4, 6, 5, 2], dtype=in
t32)>
```

In [28]:
```
r = tf.strings.unicode_decode(t, "UTF8")
r
```

Out[28]:
```
tf.RaggedTensor(values=tf.Tensor(
[   67    97   102   233    67   111   102   102   101   101    99    97
   102   102   232 21654 21857], shape=(17,), dtype=int32), row_splits=tf.Tens
or([ 0  4 10 15 17], shape=(5,), dtype=int64))
```

Ragged tensors

In [29]:
```
r = tf.ragged.constant([[11, 12], [21, 22, 23], [], [41]])
r
```

Out[29]:
```
tf.RaggedTensor(values=tf.Tensor([11 12 21 22 23 41], shape=(6,), dtype=int3
2), row_splits=tf.Tensor([0 2 5 5 6], shape=(5,), dtype=int64))
```

In [30]:
```
print(r)
```

```
<tf.RaggedTensor [[11, 12], [21, 22, 23], [], [41]]>
```

In [31]:
```
print(r[1])
```

```
tf.Tensor([21 22 23], shape=(3,), dtype=int32)
```

In [32]:
```
print(r[1:2])
```

```
<tf.RaggedTensor [[21, 22, 23]]>
```

In [33]:
```
r2 = tf.ragged.constant([[51, 52], [], [71]])
print(tf.concat([r, r2], axis=0))
```

```
<tf.RaggedTensor [[11, 12], [21, 22, 23], [], [41], [51, 52], [], [71]]>
```

In [34]:
```
r3 = tf.ragged.constant([[13, 14, 15], [24], [], [42, 43]])
print(tf.concat([r, r3], axis=1))
```

```
<tf.RaggedTensor [[11, 12, 13, 14, 15], [21, 22, 23, 24], [], [41, 42, 43]]>
```

In [35]:
```
r.to_tensor()
```

Out[35]:
```
<tf.Tensor: id=281, shape=(4, 3), dtype=int32, numpy=
array([[11, 12,  0],
       [21, 22, 23],
       [ 0,  0,  0],
       [41,  0,  0]], dtype=int32)>
```

Sparse tensors

In [36]:
```
s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                    values=[1., 2., 3.],
                    dense_shape=[3, 4])
print(s)
```

```
SparseTensor(indices=tf.Tensor(
[[0 1]
 [1 0]
 [2 3]], shape=(3, 2), dtype=int64), values=tf.Tensor([1. 2. 3.], shape=(3,),
dtype=float32), dense_shape=tf.Tensor([3 4], shape=(2,), dtype=int64))
```

In [37]:
```
tf.sparse.to_dense(s)
```

Out[37]:
```
<tf.Tensor: id=290, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

In [38]:
```
s2 = s * 2.0
```

In [39]:
```
try:
    s3 = s + 1.
except TypeError as ex:
    print(ex)
```

```
unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

In [40]:
```
s4 = tf.constant([[10., 20.], [30., 40.], [50., 60.], [70., 80.]])
tf.sparse.sparse_dense_matmul(s, s4)
```

Out[40]:
```
<tf.Tensor: id=295, shape=(3, 2), dtype=float32, numpy=
array([[ 30.,  40.],
       [ 20.,  40.],
       [210., 240.]], dtype=float32)>
```

In [41]:
```
s5 = tf.SparseTensor(indices=[[0, 2], [0, 1]],
                     values=[1., 2.],
                     dense_shape=[3, 4])
print(s5)
```

```
SparseTensor(indices=tf.Tensor(
[[0 2]
 [0 1]], shape=(2, 2), dtype=int64), values=tf.Tensor([1. 2.], shape=(2,), dty
pe=float32), dense_shape=tf.Tensor([3 4], shape=(2,), dtype=int64))
```

In [42]:
```
try:
    tf.sparse.to_dense(s5)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
indices[1] = [0,1] is out of order [Op:SparseToDense]
```

In [43]:
```
s6 = tf.sparse.reorder(s5)
tf.sparse.to_dense(s6)
```

Out[43]:
```
<tf.Tensor: id=310, shape=(3, 4), dtype=float32, numpy=
array([[0., 2., 1., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]], dtype=float32)>
```

Variables

In [44]:
```
v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
v
```

Out[44]:
```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

In [45]:
```
v.value()
```

Out[45]:
```
<tf.Tensor: id=323, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

In [46]:
```
v.numpy()
```
Out[46]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```
In [47]:
```
v.assign(2 * v)
```
Out[47]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```
In [48]:
```
v[0, 1].assign(42)
```
Out[48]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```
In [49]:
```
v[1].assign([7., 8., 9.])
```
Out[49]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  6.],
       [ 7.,  8.,  9.]], dtype=float32)>
```
In [50]:
```
try:
    v[1] = [7., 8., 9.]
except TypeError as ex:
    print(ex)
```
```
'ResourceVariable' object does not support item assignment
```
In [51]:
```
sparse_delta = tf.IndexedSlices(values=[[1., 2., 3.], [4., 5., 6.]],
                                indices=[1, 0])
v.scatter_update(sparse_delta)
```
Out[51]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[4., 5., 6.],
       [1., 2., 3.]], dtype=float32)>
```
In [52]:
```
v.scatter_nd_update(indices=[[0, 0], [1, 2]],
                    updates=[100., 200.])
```
Out[52]:
```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[100.,   5.,   6.],
       [  1.,   2., 200.]], dtype=float32)>
```
Devices

In [53]:
```
with tf.device("/cpu:0"):
    t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
```
In [54]:
```
t.device
```
Out[54]:
```
'/job:localhost/replica:0/task:0/device:CPU:0'
```

In [55]:
```
if tf.test.is_gpu_available():
    with tf.device("/gpu:0"):
        t2 = tf.constant([[1., 2., 3.], [4., 5., 6.]])
    print(t2.device)
```

```
/job:localhost/replica:0/task:0/device:GPU:0
```

Exercise 1 – Custom loss function

Let's start by loading and preparing the California housing dataset. We first load it, then split it into a training set, a validation set and a test set, and finally we scale it:

In [56]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target.reshape(-1, 1), random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)
```

1.1)

Create an `my_mse()` function with two arguments: the true labels `y_true` and the model predictions `y_pred`. Make it return the mean squared error using TensorFlow operations. Note that you could write your own custom metrics in exactly the same way. **Tip**: recall that the MSE is the mean of the squares of prediction errors, which are the differences between the predictions and the labels, so you will need to use `tf.reduce_mean()` and `tf.square()`.

In [57]:
```
def my_mse(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

1.2)

Compile your model, passing it your custom loss function, then train it and evaluate it. **Tip**: don't forget to use the scaled sets.

In [58]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1),
])
```

In [59]:
```
model.compile(loss=my_mse, optimizer="sgd")
```

In [60]:
```
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
```

Out[60]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb8b049dac8>
```

In [61]:
```
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 27us/sample - loss: 0.4692
```

Out[61]:
```
0.46920335967411364
```

1.3)

Try building and compiling the model again, this time adding `"mse"` (or equivalently `"mean_squared_error"` or `keras.losses.mean_squared_error`) to the list of additional metrics, then train the model and make sure the `my_mse` is equal to the standard `mse`.

In [62]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1),
])
model.compile(loss=my_mse, optimizer="sgd", metrics=["mean_squared_error"])
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 39us/sample - loss: 0.4548 - mea
n_squared_error: 0.4548
```

Out[62]:
```
[0.4547801451165547, 0.45478037]
```

1.4)

If you want your code to be portable to other Python implementations of the Keras API, you should use the operations in `keras.backend` rather than TensorFlow operations directly. This package contains thin wrappers around the backend's operations (for example, `keras.backend.square()` simply calls `tf.square()`). Try reimplementing the `my_mse()` function this way and use it to train and evaluate your model again. **Tip**: people frequently define `K = keras.backend` to make their code more readable.

In [63]:
```
def my_portable_mse(y_true, y_pred):
    K = keras.backend
    return K.mean(K.square(y_pred - y_true))
```

In [64]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1),
])
model.compile(loss=my_portable_mse, optimizer="sgd", metrics=["mean_squared_er
ror"])
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 42us/sample - loss: 0.4683 - mea
n_squared_error: 0.4683
```

Out[64]:
```
[0.46828289226044056, 0.46828288]
```

Exercise 2 – Custom layer

2.1)

Some layers have no weights, such as `keras.layers.Flatten` or `keras.layers.ReLU`. If you want to create a custom layer without any weights, the simplest option is to create a `keras.layers.Lambda` layer and pass it the function to perform. For example, try creating a custom layer that applies the softplus function (log(exp(X) + 1), and try calling this layer like a regular function.

Tip: you can use `tf.math.softplus()` rather than computing the log and the exponential manually.

In [65]:
```
my_softplus = keras.layers.Lambda(lambda X: tf.nn.softplus(X))
```

In [66]:
```
my_softplus([-10., -5., 0., 5., 10.])
```

Out[66]:
```
<tf.Tensor: id=106990, shape=(5,), dtype=float32, numpy=
array([4.5417706e-05, 6.7153489e-03, 6.9314718e-01, 5.0067153e+00,
       1.0000046e+01], dtype=float32)>
```

2.2)

Create a regression model like in exercise 1, but add your softplus layer at the top (i.e., after the existing 1-unit dense layer). This can be useful to ensure that your model never predicts negative values.

In [67]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1),
    my_softplus
])
model.compile(loss=my_portable_mse, optimizer="sgd")
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 29us/sample - loss: 0.4599
```

Out[67]:
```
0.45991045208864434
```

2.3)

Alternatively, try using this softplus layer as the activation function of the output layer.

Notes:

- setting a layer's activation function is just a handy way of adding an extra weightless layer.
- Keras supports the softplus activation function out of the box:
  - set `activation="softplus"`
  - or set `activation=keras.activations.softplus`
  - or add a `keras.layers.Activation("softplus")` layer to your model.

In [68]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1, activation=my_softplus)
#   A few alternatives...
#   keras.layers.Dense(1, activation=tf.function(lambda X: my_softplus(X)))
#   keras.layers.Dense(1, activation="softplus")
#   keras.layers.Dense(1, activation=keras.activations.softplus)
#   keras.layers.Dense(1), keras.layers.Activation("softplus")
])

model.compile(loss=my_portable_mse, optimizer="sgd")
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 32us/sample - loss: 0.5035
```

Out[68]:
```
0.503474702206693
```

2.4)

Now let's create a custom layer with its own weights. Use the following template to create a `MyDense` layer that computes $\phi(\mathbf{X}\mathbf{W}) + \mathbf{b}$ ϕ(XW)+b, where ϕ ϕ is the (optional) activation function, $\mathbf{X}$ X is the input data, $\mathbf{W}$ W represents the kernel (i.e., connection weights), and $\mathbf{b}$ b represents the biases, then train and evaluate a model using this instead of a regular `Dense` layer.

Tips:

- The constructor `__init__()`:
  - It must have all your layer's hyperparameters as arguments, and save them to instance variables. You will need the number of `units` and the optional `activation` function. To support all kinds of activation functions (strings or functions), simply create a `keras.layers.Activation` passing it the `activation` argument.
  - The `**kwargs` argument must be passed to the base class's constructor (`super().__init__()`) so your class can support the `input_shape` argument, and more.
- The `build()` method:
  - The `build()` method will be called automatically by Keras when it knows the shape of the inputs. Note that the argument should really be called `batch_input_shape` since it includes the batch size.
  - You must call `self.add_weight()` for each weight you want to create, specifying its `name`, `shape` (which often depends on the `input_shape`), how to initialize it, and whether or not it is `trainable`. You need two weights: the `kernel` (connection weights) and the `biases`. The kernel must be initialized randomly. The biases are usually initialized with zeros. **Note**: you can find many initializers in `keras.initializers`.
  - Do not forget to call `super().build()`, so Keras knows that the model has been built.
  - Note: you could create the weights in the constructor, but it is preferable to create them in the `build()` method, because users of your class may not always know the `input_shape` when creating the model. The first time the model is used on some actual data, the `build()` method will automatically be called with the actual `input_shape`.
- The `call()` method:
  - This is where to code your layer's actual computations. As before, you can use TensorFlow operations directly, or use `keras.backend` operations if you want the layer to be portable to other Keras implementations.
- The `compute_output_shape()` method:
  - You do not need to implement this method when using tf.keras, as the `Layer` class provides a good implementation.
  - However, if want to port your code to another Keras implementation (such as keras-team), and if the output shape is different from the input shape, then you need to implement this method. Note that the input shape is actually the batch input shape, and the ouptut shape must be the batch output shape.

In [69]:
```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        self.units = units
        self.activation = keras.layers.Activation(activation)
        super(MyDense, self).__init__(**kwargs)

    def build(self, input_shape):
        self.kernel = self.add_weight(name='kernel',
                                      shape=(input_shape[1], self.units),
                                      initializer='uniform',
                                      trainable=True)
        self.biases = self.add_weight(name='bias',
                                      shape=(self.units,),
                                      initializer='zeros',
                                      trainable=True)
        super(MyDense, self).build(input_shape)

    @tf.function   # required, see https://github.com/tensorflow/tensorflow/is
sues/25096
    def call(self, X):
        return self.activation(X @ self.kernel + self.biases)
```

In [70]:
```
model = keras.models.Sequential([
    MyDense(30, activation="relu", input_shape=X_train.shape[1:]),
    MyDense(1)
])
```

In [71]:
```
model.compile(loss="mse", optimizer="sgd")
model.fit(X_train_scaled, y_train, epochs=10,
          validation_data=(X_valid_scaled, y_valid), verbose=0)
model.evaluate(X_test_scaled, y_test)
```

```
5160/5160==============================] - 0s 45us/sample - loss: 0.5592
```

Out[71]:
```
0.5591562871785127
```

Exercise 3 – TensorFlow Functions

3.1)

Examine and run the following code examples.

In [72]:
```
def scaled_elu(z, scale=1.0, alpha=1.0):
    is_positive = tf.greater_equal(z, 0.0)
    return scale * tf.where(is_positive, z, alpha * tf.nn.elu(z))
```

In [73]:
```
scaled_elu(tf.constant(-3.))
```

Out[73]:
```
<tf.Tensor: id=193629, shape=(), dtype=float32, numpy=-0.95021296>
```

In [74]:
```
scaled_elu(tf.constant([-3., 2.5]))
```

Out[74]:
```
<tf.Tensor: id=193639, shape=(2,), dtype=float32, numpy=array([-0.95021296,
2.5       ], dtype=float32)>
```

In [75]:
```
scaled_elu_tf = tf.function(scaled_elu)
scaled_elu_tf
```

Out[75]:
```
<tensorflow.python.eager.def_function.Function at 0x7f998c34deb8>
```

In [76]:
```
scaled_elu_tf(tf.constant(-3.))
```

Out[76]:
```
<tf.Tensor: id=193655, shape=(), dtype=float32, numpy=-0.95021296>
```

In [77]:
```
scaled_elu_tf(tf.constant([-3., 2.5]))
```

Out[77]:
```
<tf.Tensor: id=193670, shape=(2,), dtype=float32, numpy=array([-0.95021296,
2.5       ], dtype=float32)>
```

In [78]:
```
scaled_elu_tf.python_function is scaled_elu
```

Out[78]:
```
True
```

In [79]:
```
%timeit scaled_elu(tf.random.normal((1000, 1000)))
```

```
3.84 ms ± 48.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [80]:
```
%timeit scaled_elu_tf(tf.random.normal((1000, 1000)))
```

```
3.61 ms ± 17.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [81]:
```python
def display_tf_code(func):
    from IPython.display import display, Markdown
    code = tf.autograph.to_code(func)
    display(Markdown('```python\n{}\n```'.format(code)))
```

In [82]:
```python
display_tf_code(scaled_elu)
```

```python
from __future__ import print_function

def tf__scaled_elu(z, scale=None, alpha=None):
  try:
    with ag__.function_scope('scaled_elu'):
      is_positive = tf.greater_equal(z, 0.0)
      return scale * tf.where(is_positive, z, alpha * tf.nn.elu(z))
  except:
    ag__.rewrite_graph_construction_error(ag_source_map__)



tf__scaled_elu.autograph_info__ = {}
```

In [83]:
```python
var = tf.Variable(0)

@tf.function
def add_21():
    return var.assign_add(21)

@tf.function
def times_2():
    return var.assign(var * 2)
```

In [84]:
```python
add_21()
times_2()
```

Out[84]:
```
<tf.Tensor: id=210741, shape=(), dtype=int32, numpy=42>
```

In [85]:
```python
def times_4(x):
    return 4. * x

@tf.function
def times_4_plus_22(x):
    return times_4(x) + 22.
```

In [86]:
```python
times_4_plus_22(tf.constant(5.))
```

Out[86]:
```
<tf.Tensor: id=210753, shape=(), dtype=float32, numpy=42.0>
```

Compute 1 + 1/2 + 1/4 + ...: the order of execution of the operations with side-effects (e.g., `assign()`) is preserved (in TF 1.x, `tf.control_dependencies()` was needed in such cases):

In [87]:
```
total = tf.Variable(0.)
increment = tf.Variable(1.)

@tf.function
def converge_to_2(n_iterations):
    for i in tf.range(n_iterations):
        total.assign_add(increment)
        increment.assign(increment / 2.0)
    return total

converge_to_2(20)
```

Out[87]:
```
<tf.Tensor: id=210839, shape=(), dtype=float32, numpy=1.9999981>
```

3.2)

Write a function that computes the sum of squares from 1 to n, where n is an argument. Convert it to a graph function by using `tf.function` as a decorator. Display the code generated by autograph using the `display_tf_code()` function. Use `%timeit` to see how must faster the TensorFlow `Function` is compared to the Python function.

In [88]:
```
@tf.function
def sum_squares(n):
    s = tf.constant(0)
    for i in range(1, n + 1):
        s = s + i ** 2
    return s
```

In [89]:
```
sum_squares(tf.constant(5))
```

Out[89]:
```
<tf.Tensor: id=210910, shape=(), dtype=int32, numpy=55>
```

In [90]:
```
display_tf_code(sum_squares.python_function)
```

```
from __future__ import print_function
import tensorflow as tf

@tf.function
def tf__sum_squares(n):
  try:
    with ag__.function_scope('sum_squares'):
      s = tf.constant(0)

      def extra_test(s_1):
        with ag__.function_scope('extra_test'):
          return True

      def loop_body(loop_vars, s_1):
        with ag__.function_scope('loop_body'):
          i = loop_vars
          s_1 = s_1 + i ** 2
          return s_1,
      s = ag__.for_stmt(ag__.range_(1, n + 1), extra_test, loop_body, (s,))
      return s
  except:
    ag__.rewrite_graph_construction_error(ag_source_map__)



tf__sum_squares.autograph_info__ = {}
```

In [91]:
```
%timeit sum_squares(10000)
```

```
The slowest run took 5.53 times longer than the fastest. This could mean that
an intermediate result is being cached.
162 µs ± 127 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [92]:
```
%timeit sum_squares.python_function(10000)
```

```
233 ms ± 2.27 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

3.3)

Examine and run the following code examples.

In [93]:
```
@tf.function
def square(x):
    print("Calling", x)  # part of the TF Function
    tf.get_logger().warning("Tracing")  # NOT part of the TF Function
    return tf.square(x)
```

In [94]:
```
for i in range(5):
    square(tf.constant(i))
```

```
WARNING: Logging before flag parsing goes to stderr.
W0217 13:53:49.594420 140301240059648 tmpau_d33og.py:19] Tracing
```

```
Calling 0
Calling 1
Calling 2
Calling 3
Calling 4
```

In [95]:
```
for i in range(5):
    square(tf.constant(i, dtype=tf.float32))
```

```
W0217 13:53:49.704779 140301240059648 tmpe58jrf0m.py:19] Tracing
```

```
Calling 0.0
Calling 1.0
Calling 2.0
Calling 3.0
Calling 4.0
```

In [96]:
```
for i in range(5):
    square(tf.constant([i, i], dtype=tf.float32))
```

```
W0217 13:53:49.819299 140301240059648 tmpqy2y7jqv.py:19] Tracing
```

```
Calling [0. 0.]
Calling [1. 1.]
Calling [2. 2.]
Calling [3. 3.]
Calling [4. 4.]
```

In [97]:
```python
# WARNING: when passing non-tensor values, a trace happens for any new value!
# This is to allow optimization in case this value determines e.g., number of
 layers.
for i in range(5):
    square(i)
```

```
W0217 13:53:50.180286 140301240059648 tmporyao7o2.py:19] Tracing
```

```
Calling 0
```

```
W0217 13:53:50.294979 140301240059648 tmp_iljwmn3.py:19] Tracing
```

```
Calling 1
```

```
W0217 13:53:50.403625 140301240059648 tmp624uh8u8.py:19] Tracing
```

```
Calling 2
```

```
W0217 13:53:50.512877 140301240059648 tmpr4ilir43.py:19] Tracing
```

```
Calling 3
```

```
W0217 13:53:50.621813 140301240059648 tmp0s_tha_k.py:19] Tracing
```

```
Calling 4
```

3.4)

When you give Keras a custom loss function, it actually creates a graph function based on it, and then uses that graph function during training. The same is true of custom metric functions, and the `call()` method of custom layers and models. Create a `my_mse()` function, like you did earlier, but add an instruction to log a message inside it (do *not* use `print()` !), and verify that the message is only logged once when you compile and train the model. Optionally, you can also find out when Keras converts custom metrics, layers and models.

In [98]:
```python
# Custom loss function
def my_mse(y_true, y_pred):
    tf.get_logger().warning("Tracing loss my_mse()")
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

In [99]:
```python
# Custom metric function
def my_mae(y_true, y_pred):
    tf.get_logger().warning("Tracing metric my_mae()")
    return tf.reduce_mean(tf.abs(y_pred - y_true))
```

In [100]:
```python
# Custom layer
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        self.units = units
        self.activation = keras.layers.Activation(activation)
        super(MyDense, self).__init__(**kwargs)

    def build(self, input_shape):
        self.kernel = self.add_weight(name='kernel',
                                      shape=(input_shape[1], self.units),
                                      initializer='uniform',
                                      trainable=True)
        self.biases = self.add_weight(name='bias',
                                      shape=(self.units,),
                                      initializer='zeros',
                                      trainable=True)
        super(MyDense, self).build(input_shape)

    def call(self, X):
        tf.get_logger().warning("Tracing MyDense.call()")
        return self.activation(X @ self.kernel + self.biases)
```

In [101]:
```python
# Custom model
class MyModel(keras.models.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.hidden1 = MyDense(30, activation="relu")
        self.hidden2 = MyDense(30, activation="relu")
        self.output_ = MyDense(1)

    def call(self, input):
        tf.get_logger().warning("Tracing MyModel.call()")
        hidden1 = self.hidden1(input)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input, hidden2])
        output = self.output_(concat)
        return output

model = MyModel()
```

In [102]:
```python
model.compile(loss=my_mse, optimizer="sgd", metrics=[my_mae])
```

In [103]:
```
model.fit(X_train_scaled, y_train, epochs=2,
          validation_data=(X_valid_scaled, y_valid))
model.evaluate(X_test_scaled, y_test)
```

```
W0217 13:54:30.681071 140301240059648 training.py:2703] Tracing MyModel.call()
W0217 13:54:30.682790 140301240059648 deprecation.py:506] From /opt/anaconda3/
lib/python3.6/site-packages/tensorflow/python/keras/initializers.py:111: calli
ng RandomUniform.__init__ (from tensorflow.python.ops.init_ops) with dtype is
deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
constructor
W0217 13:54:30.689498 140301240059648 base_layer.py:558] Tracing MyDense.call
()
W0217 13:54:30.700104 140301240059648 base_layer.py:558] Tracing MyDense.call
()
W0217 13:54:30.712319 140301240059648 base_layer.py:558] Tracing MyDense.call
()
W0217 13:54:30.723696 140301240059648 training_utils.py:644] Tracing loss my_m
se()
W0217 13:54:30.733945 140301240059648 metrics.py:551] Tracing metric my_mae()
W0217 13:54:30.745486 140301240059648 training_utils.py:644] Tracing metric my
_mae()
```

```
Train on 11610 samples, validate on 3870 samples
Epoch 1/2
11610/11610==============================] - 1s 86us/sample - loss: 3.3026 - m
y_mae: 1.4837 - val_loss: 1.8555 - val_my_mae: 0.9966
Epoch 2/2
11610/11610==============================] - 1s 69us/sample - loss: 1.2436 - m
y_mae: 0.7962 - val_loss: 1.3280 - val_my_mae: 0.6464
5160/5160==============================] - 0s 36us/sample - loss: 0.8685 - my_
mae: 0.6459
```

Out[103]:
```
[0.8685485477595366, 0.6459101]
```

Notice that each custom function is traced just once, except for the metric function. That's a bit odd.

3.5)

Examine the following function, and try to call it with various argument types and shapes. Notice that only tensors of type `int32` and one dimension (of any size) are accepted now that we have specified the `input_signature`.

In [104]:
```
@tf.function(input_signature=[tf.TensorSpec([None], tf.int32, name="x")])
def cube(z):
    return tf.pow(z, 3)
```

In [105]:
```
cube(tf.constant([1, 2, 3]))
```

Out[105]:
```
<tf.Tensor: id=400017, shape=(3,), dtype=int32, numpy=array([ 1,  8, 27], dtyp
e=int32)>
```

In [106]:
```
cube(tf.constant([1, 2, 3, 4, 5]))
```

Out[106]:
```
<tf.Tensor: id=400020, shape=(5,), dtype=int32, numpy=array([  1,   8,  27,  6
4, 125], dtype=int32)>
```

In [107]:
```
try:
    cube([1, 2, 3])
except ValueError as ex:
    print(ex)
```

```
Structure of Python function inputs does not match input_signature.
```

In [108]:
```
try:
    cube(tf.constant([1., 2., 3]))
except ValueError as ex:
    print(ex)
```

```
Python inputs incompatible with input_signature: inputs ((<tf.Tensor: id=40002
2, shape=(3,), dtype=float32, numpy=array([1., 2., 3.], dtype=float32)>,)), in
put_signature ((TensorSpec(shape=(None,), dtype=tf.int32, name='x'),))
```

In [109]:
```
try:
    cube(tf.constant([[1, 2], [3, 4]]))
except ValueError as ex:
    print(ex)
```

```
Python inputs incompatible with input_signature: inputs ((<tf.Tensor: id=40002
4, shape=(2, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4]], dtype=int32)>,)), input_signature ((TensorSpec(shape=(None,),
dtype=tf.int32, name='x'),))
```

Exercise 4 – Function Graphs

4.1)

Examine and run the following code examples.

In [110]:
```
@tf.function(input_signature=[tf.TensorSpec([None], tf.int32, name="x")])
def cube(z):
    return tf.pow(z, 3)
```

In [111]:
```
cube_func_int32 = cube.get_concrete_function(tf.TensorSpec([None], tf.int32))
cube_func_int32
```

Out[111]:
```
<tensorflow.python.eager.function.ConcreteFunction at 0x7f939d660978>
```

In [112]:
```
cube_func_int32 is cube.get_concrete_function(tf.TensorSpec([5], tf.int32))
```

Out[112]:
```
True
```

In [113]:
```
cube_func_int32 is cube.get_concrete_function(tf.constant([1, 2, 3]))
```

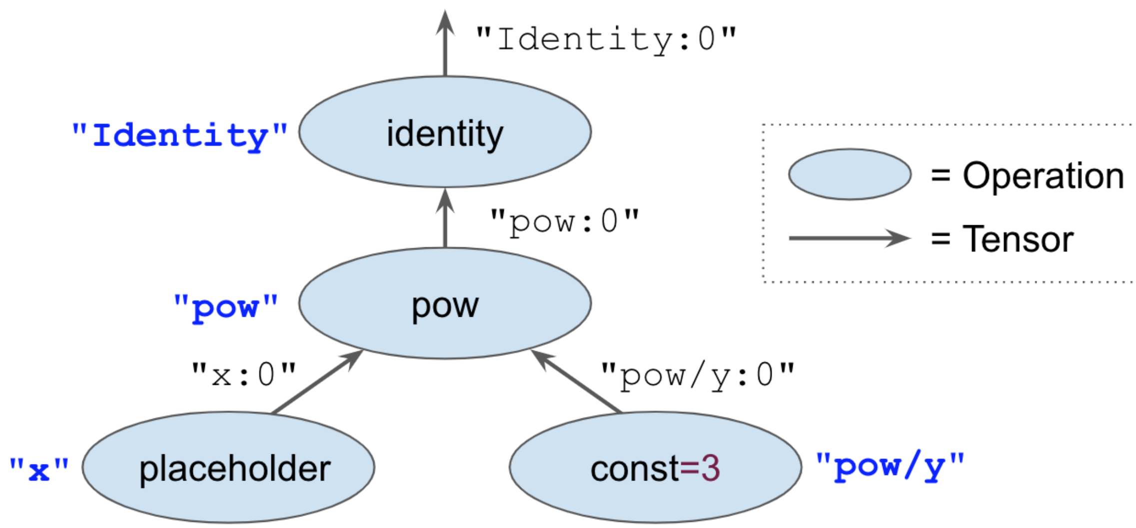
Out[113]:
```
True
```

In [114]:
```
cube_func_int32.graph
```

Out[114]:
```
<tensorflow.python.framework.func_graph.FuncGraph at 0x7f939d660748>
```

4.2)

The function's graph is represented on the following diagram. Call the graph's `get_operations()` method to get the list of operations. Each operation has an `inputs` attribute that returns an iterator over its input tensors (these are symbolic: contrary to tensors we have used up to now, they have no value). It also has an `outputs` attribute that returns the list of output tensors. Each tensor has an `op` attribute that returns the operation it comes from. Try navigating through the graph using these methods and attributes.



In [115]:
```
cube_func_int32.graph.get_operations()
```

Out[115]:
```
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'Pow/y' type=Const>,
 <tf.Operation 'Pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In [116]:
```
pow_op = cube_func_int32.graph.get_operations()[2]
pow_op
```

Out[116]:
```
<tf.Operation 'Pow' type=Pow>
```

In [117]:
```
pow_in = list(pow_op.inputs)
pow_in
```

Out[117]:
```
[<tf.Tensor 'x:0' shape=(None,) dtype=int32>,
 <tf.Tensor 'Pow/y:0' shape=() dtype=int32>]
```

In [118]:
```
pow_out = list(pow_op.outputs)
pow_out
```

Out[118]:
```
[<tf.Tensor 'Pow:0' shape=(None,) dtype=int32>]
```

In [119]:
```
pow_in = list(pow_op.inputs)
pow_in
```

Out[119]:
```
[<tf.Tensor 'x:0' shape=(None,) dtype=int32>,
 <tf.Tensor 'Pow/y:0' shape=() dtype=int32>]
```

In [120]:
```
pow_in[0].op
```
Out[120]:
```
<tf.Operation 'x' type=Placeholder>
```

4.3)

Each operation has a default name, such as `"pow"` (you can override it by setting the `name` attribute when you call the operation). In case of a name conflict, TensorFlow adds an underscore and anindex to make the name unique (e.g. `"pow 1"`). Moreover, each tensor has the same name as the operation that outputs it, followed by a colon `:` and the tensor's `index` (e.g., `"pow:0"`). Most operations have a single output tensor, so most tensors have a name that ends with `:0`. Try using `get_operation_by_name()` and `get_tensor_by_name()` to access any op and tensor you wish.

In [121]:
```
cube_func_int32.graph.get_operation_by_name("x")
```
Out[121]:
```
<tf.Operation 'x' type=Placeholder>
```

In [122]:
```
cube_func_int32.graph.get_tensor_by_name("x:0")
```
Out[122]:
```
<tf.Tensor 'x:0' shape=(None,) dtype=int32>
```

4.4)

Call the graph's `as_graph_def()` method and print the output. This is a protobuf representation of the computation graph: it is what makes TensorFlow models so portable.

In [123]:
```
cube_func_int32.graph.as_graph_def()
```

Out[123]:


```
node {
  name: "x"
  op: "Placeholder"
  attr {
    key: "_user_specified_name"
    value {
      s: "x"
    }
  }
  attr {
    key: "dtype"
    value {
      type: DT_INT32
    }
  }
  attr {
    key: "shape"
    value {
      shape {
        dim {
          size: -1
        }
      }
    }
  }
}
node {
  name: "Pow/y"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_INT32
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_INT32
        tensor_shape {
        }
        int_val: 3
      }
    }
  }
}
node {
  name: "Pow"
  op: "Pow"
  input: "x"
  input: "Pow/y"
  attr {
    key: "T"
    value {
      type: DT_INT32
    }
  }
}
node {
  name: "Identity"
```

```
    op: "Identity"
    input: "Pow"
    attr {
      key: "T"
      value {
        type: DT_INT32
      }
    }
  }
  versions {
    producer: 27
  }
```

4.5)

Get the concrete function's `function_def`, and look at its `signature`. This shows the names and types of the nodes in the graph that correspond to the function's inputs and outputs. This will come in handy when you deploy models to TensorFlow Serving or Google Cloud ML Engine.

In [124]:
```python
cube_func_int32.function_def.signature
```

Out[124]:
```
name: "__inference_cube_400032"
input_arg {
  name: "x"
  type: DT_INT32
}
output_arg {
  name: "identity"
  type: DT_INT32
}
```

Exercise 5 – Autodiff

5.1)

Examine and run the following code examples.

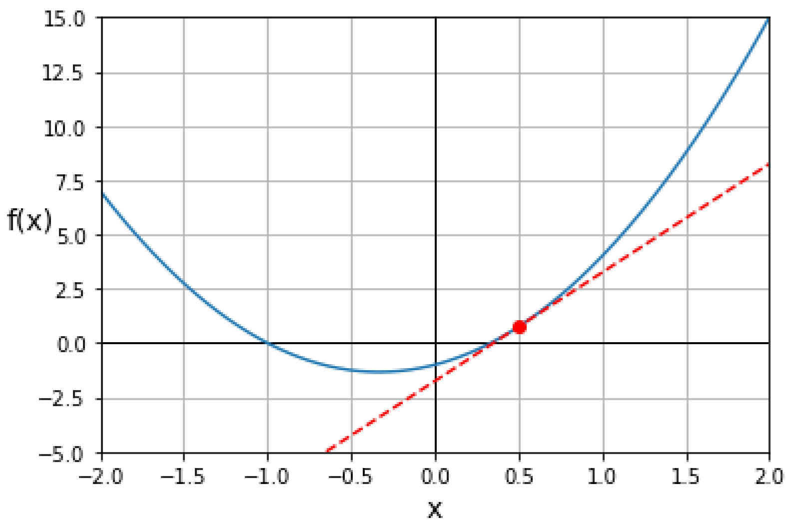
In [125]:
```python
def f(x):
    return 3. * x ** 2 + 2. * x - 1.
```

In [126]:
```python
def approximate_derivative(f, x, eps=1e-3):
    return (f(x + eps) - f(x - eps)) / (2. * eps)
```

In [127]:
```python
approximate_derivative(f, 1.0) # true derivative = 8
```

Out[127]:
```
7.999999999999119
```

In [128]:
```python
xs = np.linspace(-2, 2, 200)
fs = f(xs)
x0 = 0.5
df_x0 = approximate_derivative(f, x0)
tangent_x0 = df_x0 * (xs - x0) + f(x0)
plt.plot([-2, 2], [0, 0], "k-", linewidth=1)
plt.plot([0, 0], [-5, 15], "k-", linewidth=1)
plt.plot(xs, fs)
plt.plot(xs, tangent_x0, "r--")
plt.plot(x0, f(x0), "ro")
plt.grid(True)
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14, rotation=0)
plt.axis([-2, 2, -5, 15])
plt.show()
```



In [129]:
```python
def g(x1, x2):
    return (x1 + 5) * (x2 ** 2)
```

In [130]:
```python
def approximate_gradient(f, x1, x2, eps=1e-3):
    df_x1 = approximate_derivative(lambda x: f(x, x2), x1, eps)
    df_x2 = approximate_derivative(lambda x: f(x1, x), x2, eps)
    return df_x1, df_x2
```

In [131]:
```python
approximate_gradient(g, 2.0, 3.0) # true gradient = (9, 42)
```

Out[131]:
```
(8.999999999993236, 41.999999999994486)
```

In [132]:
```python
x1 = tf.Variable(2.0)
x2 = tf.Variable(3.0)
with tf.GradientTape() as tape:
    z = g(x1, x2)
grads = tape.gradient(z, [x1, x2])
grads
```

Out[132]:
```
[<tf.Tensor: id=400058, shape=(), dtype=float32, numpy=9.0>,
 <tf.Tensor: id=400070, shape=(), dtype=float32, numpy=42.0>]
```

In [133]:
```
x1 = tf.Variable(2.0)
x2 = tf.Variable(3.0)
with tf.GradientTape() as tape:
    z = g(x1, x2)

dz_x1 = tape.gradient(z, x1)
try:
    dz_x2 = tape.gradient(z, x2)
except RuntimeError as ex:
    print(ex)
```

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

In [134]:
```
x1 = tf.Variable(2.0)
x2 = tf.Variable(3.0)
with tf.GradientTape(persistent=True) as tape:
    z = g(x1, x2)

dz_x1 = tape.gradient(z, x1)
dz_x2 = tape.gradient(z, x2)
del tape
dz_x1, dz_x2
```

Out[134]:
```
(<tf.Tensor: id=400156, shape=(), dtype=float32, numpy=9.0>,
 <tf.Tensor: id=400195, shape=(), dtype=float32, numpy=42.0>)
```

In [135]:
```
x1 = tf.constant(2.0) # <= not Variable
x2 = tf.constant(3.0) # <= not Variable
with tf.GradientTape() as tape:
    z = g(x1, x2)

grads = tape.gradient(z, [x1, x2])
grads
```

Out[135]:
```
[None, None]
```

In [136]:
```
x1 = tf.constant(2.0)
x2 = tf.constant(3.0)
with tf.GradientTape() as tape:
    tape.watch(x1)
    tape.watch(x2)
    z = g(x1, x2)

grads = tape.gradient(z, [x1, x2])
grads
```

Out[136]:
```
[<tf.Tensor: id=400226, shape=(), dtype=float32, numpy=9.0>,
 <tf.Tensor: id=400238, shape=(), dtype=float32, numpy=42.0>]
```

In [137]:
```
x = tf.Variable(5.0)
with tf.GradientTape() as tape:
    z1 = 3 * x
    z2 = x ** 2
tape.gradient([z1, z2], x) # dz1_x + dz2_x = 3 + 2x = 3 + 2*5 = 13
```

Out[137]:
```
<tf.Tensor: id=400295, shape=(), dtype=float32, numpy=13.0>
```

In [138]:
```python
x1 = tf.Variable(2.0)
x2 = tf.Variable(3.0)
with tf.GradientTape(persistent=True) as hessian_tape:
    with tf.GradientTape() as jacobian_tape:
        z = g(x1, x2)
    jacobians = jacobian_tape.gradient(z, [x1, x2])
hessians = [hessian_tape.gradient(jacobian, [x1, x2])
            for jacobian in jacobians]
del hessian_tape
hessians
```

Out[138]:
```
[[None, <tf.Tensor: id=400358, shape=(), dtype=float32, numpy=6.0>],
 [<tf.Tensor: id=400412, shape=(), dtype=float32, numpy=6.0>,
  <tf.Tensor: id=400395, shape=(), dtype=float32, numpy=14.0>]]
```

5.2)

Implement Gradient Descent manually to find the value of `x` that minimizes the following function `f(x)`.

Tips:

- Define a variable `x` and initialize it to 0.
- Define the `learning_rate` (e.g., 0.1).
- Write a loop that will repeatedly (1) compute the gradient of `f` (actually a derivative in this case) at the current value of `x`, and (2) tweak `x` slightly in the opposite direction (by subtracting `learning_rate * df_dx`). You can use `x.assign_sub(...)` for this.
- Using calculus, we can find that the algorithm should converge to $x = -\frac{1}{3}$ x=−13. Indeed, the derivative of $f(x) = 3x^2 + 2x - 1$ f(x)=3x2+2x−1 is $f'(x) = 6x + 2$ f′(x)=6x+2, so the minimum is reached when $f'(x) = 0$ f′(x)=0 (slope is 0), so $6x + 2 = 0$ 6x+2=0, which leads to $x = -\frac{1}{3}$ x=−13.

In [139]:
```python
def f(x):
    return 3. * x ** 2 + 2. * x - 1.
```

In [140]:
```python
learning_rate = 0.1
x = tf.Variable(0.0)

for iteration in range(100):
    with tf.GradientTape() as tape:
        z = f(x)
    dz_dx = tape.gradient(z, x)
    x.assign_sub(learning_rate * dz_dx)
x
```

Out[140]:
```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.3333333>
```

5.3)

Now use an `SGD` optimizer instead of manually tweaking `x`.

Tips:

- You first need to create an `SGD` optimizer, optionally specifying the learning rate (e.g., `lr=0.1`).
- Next replace the manual tweaking of `x` in your previous code to use `optimizer.apply_gradients()` instead. You need to pass it a list of gradient/variable pairs (just one pair in this example).

In [141]:
```
x = tf.Variable(0.0)
optimizer = keras.optimizers.SGD(lr=0.1)

for iteration in range(100):
    with tf.GradientTape() as tape:
        z = f(x)
    dz_dx = tape.gradient(z, x)
    optimizer.apply_gradients([(dz_dx, x)])
x
```

Out[141]:
```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.3333333>
```

5.4)

Create a `Sequential` model for the California housing problem (no need to compile it), and train it using your own training loop, instead of using `fit()`. Evaluate your model on the validation set at the end of each epoch, and display the result.

Tips:

- You can use the following `random_batch()` function to get a new batch of training data at each iteration (the Data API would be much preferable, as we will see in the next notebook).
- You can use the model like a function to make predictions: `y_pred = model(X_batch)`
- You can use `keras.losses.mean_squared_error()` to compute the loss. Note that it returns one loss per instance, so you need to use `tf.reduce_mean()` to get the mean loss.
- You can use `model.trainable_variables` to get the full list of trainable variables in your model.
- You can use `zip(gradients, variables)` to create a list containing all the gradient/variable pairs.

In [142]:
```
def random_batch(X, y, batch_size = 32):
    idx = np.random.randint(0, len(X), size=batch_size)
    return X[idx], y[idx]
```

In [143]:
```
epochs = 10
batch_size = 32
steps_per_epoch = len(X_train) // batch_size
optimizer = keras.optimizers.SGD()
loss_fn = keras.losses.mean_squared_error

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])

for epoch in range(epochs):
    for step in range(steps_per_epoch):
        X_batch, y_batch = random_batch(X_train_scaled, y_train, batch_size)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
        grads = tape.gradient(loss, model.variables)
        grads_and_vars = zip(grads, model.variables)
        optimizer.apply_gradients(grads_and_vars)
    y_pred = model(X_valid_scaled)
    valid_loss = tf.reduce_mean(loss_fn(y_valid, y_pred))
    print("Epoch", epoch, "valid mse:", valid_loss.numpy())
```

```
Epoch 0 valid mse: 10.514079
Epoch 1 valid mse: 7.5957823
Epoch 2 valid mse: 0.8575303
Epoch 3 valid mse: 0.70870006
Epoch 4 valid mse: 0.61762285
Epoch 5 valid mse: 0.5635681
Epoch 6 valid mse: 0.55062544
Epoch 7 valid mse: 0.5125756
Epoch 8 valid mse: 0.50464547
Epoch 9 valid mse: 0.5018674
```

5.5)

Examine and run the following code examples, then update your training loop to display the training loss at each iteration.

Tips:

- You can use a `keras.metrics.MeanSquaredError` instance to efficiently track the running mean squared error at each iteration.
- Make sure you reset the metric's states at the start of each epoch.
- You can use `print("\r", mse, end="")` to display the MSE on the same line at each iteration.

In [144]:
```
metric = keras.metrics.MeanSquaredError()
metric([5.], [2.])  # error = (2 - 5)**2 = 9
metric([0.], [1.])  # error = (1 - 0)**2 = 1
metric.result()     # mean error = (9 + 1) / 2 = 5
```

Out[144]:
```
<tf.Tensor: id=605063, shape=(), dtype=float32, numpy=5.0>
```

In [145]:
```
metric.reset_states()
metric.result()
```

Out[145]:
```
<tf.Tensor: id=605069, shape=(), dtype=float32, numpy=0.0>
```

In [146]:
```
metric([1.], [3.])  # error = (3 - 1)**2 = 4
metric.result()     # mean error = 4 / 1 = 4
```

Out[146]:
```
<tf.Tensor: id=605086, shape=(), dtype=float32, numpy=4.0>
```

In [147]:
```
epochs = 10
batch_size = 32
steps_per_epoch = len(X_train) // batch_size
optimizer = keras.optimizers.SGD()
loss_fn = keras.losses.mean_squared_error
metric = keras.metrics.MeanSquaredError()  # ADDED

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])

for epoch in range(epochs):
    metric.reset_states()  # ADDED
    for step in range(steps_per_epoch):
        X_batch, y_batch = random_batch(X_train_scaled, y_train, batch_size)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            metric(y_batch, y_pred)  # ADDED
        grads = tape.gradient(loss, model.trainable_variables)
        grads_and_vars = zip(grads, model.trainable_variables)
        optimizer.apply_gradients(grads_and_vars)
        print("\rEpoch", epoch, " train mse:", metric.result().numpy(), end=""
)  # ADDED
    y_pred = model(X_valid_scaled)
    valid_loss = tf.reduce_mean(loss_fn(y_valid, y_pred))
    print("\tvalid mse:", valid_loss.numpy())
```

```
Epoch 0  train mse: 1.6207398   valid mse: 17.008974
Epoch 1  train mse: 0.7614396   valid mse: 9.84763
Epoch 2  train mse: 0.6617865   valid mse: 4.783753
Epoch 3  train mse: 0.6473666   valid mse: 0.61845475
Epoch 4  train mse: 0.56790835  valid mse: 0.74327624
Epoch 5  train mse: 0.56685776  valid mse: 0.54980206
Epoch 6  train mse: 0.53035283  valid mse: 0.60690194
Epoch 7  train mse: 0.52254856  valid mse: 0.50044566
Epoch 8  train mse: 0.50599176  valid mse: 0.5285307
Epoch 9  train mse: 0.497541    valid mse: 0.5259951
```

Conclusion

Great! You now know how to use TensorFlow's low-level API to write custom loss functions, layers, and models. You also learned how to optimize your functions by converting them to graphs: this allows TensorFlow to run operations in parallel and to perform various optimizations. Next, you learned how TensorFlow Functions and graphs are structured, and how to navigate through them. Finally, you learned how to use autodiff and write your own custom training loops.