Основы Tensorflow

TensorFlow — открытый фреймворк (библиотека) для построения и тренировки нейронных сетей для решения задач машинного обучения.

TensorFlow был изначально разработан компанией Google для внутренних целей, но потом переведен в свободный доступ.

Основным языком для работы с TensorFlow является **Python**, компания Google также поддерживает версии TensorFlow для **C++** и **Java**. Развиваются реализации **TensorFlow.js** для языка **JavaScript** и **Swift for Tensorflow** для языка **Swift**. Также развивается проект **TensorFlow Lite**, представляющий собой набор инструментов, обеспечивающих машинное обучение на мобильных устройствах (для языков Java, Swift, Objective-C, C++ и Python).

In [1]:
```
import tensorflow as tf
import numpy as np
tf.__version__
```

Out[1]:
```
'2.8.0'
```

Главным в TensorFlow является возможность т.н. **дифференцируемого программирования** (differentiable programming).

Дифференцируемое программирование позволяет автоматически вычислять производные функций в программе на языке программирования высокого уровня. Это позволяет оптимизировать параметры программы при помощи градиентных методов (градиентного спуска). Дифференцируемое программирование активно применяется при обучении нейронных сетей, в вероятностном программировании и байесовском выводе.

Большинство фреймворков дифференцируемого программирования работают путем построения графа, содержащего поток управления и структуры данных. Наиболее известные реализации обычно делят на две группы:

- Подходы на основе статических (скомпилированных) графов (TensorFlow версии 1, Theano, MXNet), как правило, обеспечивают хорошую оптимизацию компилятора и более легкое масштабирование до больших систем, но их статический характер ограничивает интерактивность и типы программ, которые можно легко создать (например, программы, включающие циклы или рекурсию), а также усложняют пользователям задачу разработки и отладки программы.
- Подходы на основе динамических графов (TensorFlow версии 2, PyTorch, AutoGrad) упрощают разработку и анализ программ, однако приводят к накладным расходам интерпретатора, снижению масштабируемости и снижению эффективности оптимизации компилятора.

Отличительными особенностями TensorFlow являются:

- Эффективное выполнение низкоуровневых тензорных операций на CPU, GPU или TPU.
- Вычисление градиента произвольных дифференцируемых выражений.
- Масштабирование вычислений на множество устройств, таких как кластеры графических процессоров.
- Экспорт программ («графов») во внешние среды выполнения (серверы, браузеры, мобильные и встроенные устройства).

Тензоры

Тензоры в TensorFlow представляют собой многомерные массивы элементов одного типа и весьма похожи на массивы в NumPy. Все тензоры являются неизменяемыми (immutable) объектами как числа и строки в Python – нельзя изменить содержимое тензора, но можно создать новый тензор.

Тензоры имеют свойства, аналогичные свойствам массивов NumPy:

- `dtype` : тип элементов тензора.
- `ndim` : ранг или количество измерений (осей) тензора. Скаляр имеет ранг 0, вектор имеет ранг 1, матрица имеет ранг 2.
- `shape` : форма тензора (длина (количество элементов) для каждого из измерений тензора).
- `size` : общее количество элементов в тензоре, произведение чисел в `shape` .

Создание тензоров

Рассмотрим примеры тензоров. Начнем со **скалярного тензора** или тензора **ранга 0**. Скалярный тензор содержит единственное значение, у него нет измерений с индексами.

In [2]:
```
rank_0_tensor = tf.constant(2022)
print(rank_0_tensor)
rank_0_tensor.ndim
```

```
tf.Tensor(2022, shape=(), dtype=int32)Metal device set to: AMD Radeon Pro 5500M


2022-05-06 10:37:34.451531: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-05-06 10:37:34.452068: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.
2022-05-06 10:37:34.452294: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
```

Out[2]:
```
0
```

По умолчанию тип тензора – целое число `int32` .

Вектор или тензор **ранга 1** напоминает список значений. Вектор имеет одну ось (измерение):

In [3]:
```
rank_1_tensor = tf.constant([1., 4., 9., 16., 25.])
print(rank_1_tensor)
rank_1_tensor.ndim
```

```
tf.Tensor([ 1.  4.  9. 16. 25.], shape=(5,), dtype=float32)
```

Out[3]:
```
1
```

Тип тензора `float32` определился по значениям в списке.

Матрица или тензор **ранга 2** имеет две оси (измерения):

In [4]:
```
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
print(rank_2_tensor)
rank_2_tensor.ndim
```

```
tf.Tensor(
[[1. 2.]
 [3. 4.]
 [5. 6.]], shape=(3, 2), dtype=float16)
```

Out[4]:
```
2
```

Здесь тип тензора `float16` явно указан при создании тензора.

Тензоры могут иметь более чем два измерения. Например, тензор с тремя измерениями (куб):

In [5]:
```
rank_3_tensor = tf.constant([
  [[1, 2, 3],
   [4, 5, 6]],
  [[7, 8, 9],
   [10, 11, 12]],
  [[13, 14, 15],
   [16, 17, 18]],
  [[19, 20, 21],
   [22, 23, 24]]
])

print(rank_3_tensor)
rank_3_tensor.ndim
```

```
tf.Tensor(
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]

 [[13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]]], shape=(4, 2, 3), dtype=int32)
```

Out[5]:
```
3
```

Тензор можно преобразовать в массив NumPy либо с помощью функции `np.array()`, либо с помощью метода `numpy()`:

In [6]:
```
np.array(rank_2_tensor)
```

Out[6]:
```
array([[1., 2.],
       [3., 4.],
       [5., 6.]], dtype=float16)
```

In [7]:
```
rank_2_tensor.numpy()
```

Out[7]:
```
array([[1., 2.],
       [3., 4.],
       [5., 6.]], dtype=float16)
```

Тензоры, как правило, содержат числа с плавающей точкой или целые числа, но могут иметь элементы других типов, в том числе:

- комплексные числа
- строки

Базовый класс `tf.Tensor` требует, чтобы тензоры были «прямоугольными», то есть вдоль каждой оси все элементы имели одинаковый размер. Однако существуют специализированные типы тензоров, которые могут обрабатывать данные различных форм, например, рваные тензоры (RaggedTensor) и разреженные тензоры (SparseTensor).

Тензоры могут создаваться различными функциями, аналогичными функциям в NumPy, например:

- `tf.zeros()` – тензор из нулей
- `tf.zeros_like()` – тензор из нулей заданной формы (shape)
- `tf.ones()` – тензор из единиц
- `tf.ones_like()` – тензор из единиц заданной формы (shape)
- `tf.eye()` – тензор для единичной матрицы
- `tf.fill()` – тензор заданной формы (shape) с элементами, равными заданному скалярному значению

Операции над тензорами

Над тензорами можно выполнять базовые математические операции, включая сложение, поэлементное умножение и матричное умножение. Для этого можно использовать функции Tensorflow:

In [8]:
```
a = tf.constant(np.arange(9).reshape(3,3))
b = tf.ones((3,3),dtype=tf.int64)

print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int64)

tf.Tensor(
[[0 1 2]
 [3 4 5]
 [6 7 8]], shape=(3, 3), dtype=int64)

tf.Tensor(
[[ 3  3  3]
 [12 12 12]
 [21 21 21]], shape=(3, 3), dtype=int64)
```

Tensorflow также выполняет перегрузку многих математических и логических операторов, например:

```
z = -x  # z = tf.negative(x)
z = x + y  # z = tf.add(x, y)
z = x - y  # z = tf.subtract(x, y)
z = x * y  # z = tf.mul(x, y)
z = x / y  # z = tf.div(x, y)
z = x // y  # z = tf.floordiv(x, y)
z = x % y  # z = tf.mod(x, y)
z = x ** y  # z = tf.pow(x, y)
z = x @ y  # z = tf.matmul(x, y)
z = x > y  # z = tf.greater(x, y)
z = x >= y  # z = tf.greater_equal(x, y)
z = x < y  # z = tf.less(x, y)
z = x <= y  # z = tf.less_equal(x, y)
z = abs(x)  # z = tf.abs(x)
z = x & y  # z = tf.logical_and(x, y)
z = x | y  # z = tf.logical_or(x, y)
z = x ^ y  # z = tf.logical_xor(x, y)
z = ~x  # z = tf.logical_not(x)
```

Поэтому указанные выше тензоры можно также получить следующим образом:

In [9]:
```
print(a + b, "\n") # поэлементное сложение
print(a * b, "\n") # поэлементное умножение
print(a @ b, "\n") # матричное умножение
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int64)

tf.Tensor(
[[0 1 2]
 [3 4 5]
 [6 7 8]], shape=(3, 3), dtype=int64)

tf.Tensor(
[[ 3  3  3]
 [12 12 12]
 [21 21 21]], shape=(3, 3), dtype=int64)
```

Тензоры можно использовать во всех операциях Tensorflow:

In [10]:
```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

print(tf.reduce_max(c))  # максимальное значение

print(tf.argmax(c))      # индекс максимального значения

print(tf.nn.softmax(c))  # вычислить функцию softmax
```

```
tf.Tensor(10.0, shape=(), dtype=float32)
tf.Tensor([1 0], shape=(2,), dtype=int64)
tf.Tensor(
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339462e-04]], shape=(2, 2), dtype=float32)
```

Индексирование в Tensorflow

TensorFlow следует стандартным правилам индексации Python и основным правилам индексации NumPy:

- индексы начинаются с 0
- отрицательные индексы отсчитываются в обратном порядке с конца
- двоеточия `:` используются для срезов: `start:stop:step`

In [11]:
```
rank_1_tensor = tf.constant(range(10))
print(rank_1_tensor.numpy())
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Индексирование тензора при помощи конкретного значения индекса не сохраняет измерение (ось) тензора:

In [12]:
```
print("Первый элемент:", rank_1_tensor[0].numpy())
print("Второй элемент:", rank_1_tensor[1].numpy())
print("Последний эл-т:", rank_1_tensor[-1].numpy())
```

```
Первый элемент: 0
Второй элемент: 1
Последний эл-т: 9
```

Индексирование с использованием срезов сохраняет измерение (ось):

In [13]:
```
print("Все элементы:", rank_1_tensor[:].numpy())
print("До 4-го элемента:", rank_1_tensor[:4].numpy())
print("От 4-го элемента до конца:", rank_1_tensor[4:].numpy())
print("От 2-го до 7 элемента:", rank_1_tensor[2:7].numpy())
print("Через один элемент:", rank_1_tensor[::2].numpy())
print("В обратном порядке:", rank_1_tensor[::-1].numpy())
```

```
Все элементы: [0 1 2 3 4 5 6 7 8 9]
До 4-го элемента: [0 1 2 3]
От 4-го элемента до конца: [4 5 6 7 8 9]
От 2-го до 7 элемента: [2 3 4 5 6]
Через один элемент: [0 2 4 6 8]
В обратном порядке: [9 8 7 6 5 4 3 2 1 0]
```

Тензоры более высокого ранга индексируются использованием нескольких индексов. Те же самые правила, что и в случае одного измерения, применяются к каждому измерению (оси) независимо.

In [14]:
```
print(rank_2_tensor.numpy())
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

При использовании целого числа для каждого индекса получаем скаляр:

In [15]:
```
print(rank_2_tensor[1, 1].numpy())
```

```
4.0
```

Можно использовать любую комбинацию целых индексов и срезов:

In [16]:
```
print("Вторая строка:", rank_2_tensor[1, :].numpy())
print("Второй столбец:", rank_2_tensor[:, 1].numpy())
print("Последняя строка:", rank_2_tensor[-1, :].numpy())
print("Первый элемент последнего столбца:", rank_2_tensor[0, -1].nu
print("Пропустить первую строку:\n", rank_2_tensor[1:, :].numpy())
```

```
Вторая строка: [3. 4.]
Второй столбец: [2. 4. 6.]
Последняя строка: [5. 6.]
Первый элемент последнего столбца: 2.0
Пропустить первую строку:
 [[3. 4.]
 [5. 6.]]
```

Изменение формы тензоров

Иногда изменение формы тензора является необходимым при проведении вычислений.

In [17]:
```
x = tf.constant([[1], [2], [3]])
print(x.shape)
```

```
(3, 1)
```

Вообще говоря, форма тензора является объектом класса `TensorShape`, но этот объект допускает индексирование и может быть преобразован в список:

In [18]:
```
x.shape[0],x.shape[1]
```

Out[18]:
```
(3, 1)
```

In [19]:
```
x.shape.as_list()
```

Out[19]:
```
[3, 1]
```

Можно преобразовать тензор к другой форме. Функция `tf.reshape()` выполняется быстро и эффективно, поскольку данные тензора не нужно дублировать:

In [20]:
```
x_reshaped = tf.reshape(x, [1, 3])
x_reshaped.shape
```

Out[20]:
```
TensorShape([1, 3])
```

In [21]:
```
print(x)
print(x_reshaped)
```

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

При изменении формы тензора данные сохраняются в памяти, и создается новый тензор с новой формой, указывающий на те же данные. TensorFlow использует упорядочение памяти по строкам в стиле языка `C`, где увеличение самого правого индекса соответствует одному шагу в памяти.

In [22]:
```
print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]

 [[13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]]], shape=(4, 2, 3), dtype=int32)
```

Если сгладить тензор (указав форму `[-1]`), то можно увидеть, в каком порядке он расположен в памяти.

In [23]:
```
print(tf.reshape(rank_3_tensor, [-1]))
```

```
tf.Tensor([ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 1
9 20 21 22 23 24], shape=(24,), dtype=int32)
```

Обычно `tf.reshape()` используется для объединения или разделения соседних осей (или добавления или удаления измерений (осей) длиной 1). Для тензора 4x2x3 возможно изменение формы до (4x2)x3 или 4x(2x3), поскольку при этом срезы не смешиваются:

In [24]:
```
print(tf.reshape(rank_3_tensor, [8, 3]), "\n")
print(tf.reshape(rank_3_tensor, [4, -1]), "\n")
print(tf.reshape(rank_3_tensor, [-1, 4]), "\n")
```

```
tf.Tensor(
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]
 [19 20 21]
 [22 23 24]], shape=(8, 3), dtype=int32)

tf.Tensor(
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]], shape=(4, 6), dtype=int32)

tf.Tensor(
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]], shape=(6, 4), dtype=int32)
```

Изменение формы тензора возможно для любой формы с тем же общим количеством элементов, но оно не принесет ничего полезного, если не соблюдается порядок осей. Перестановка измерений (осей) при помощи `tf.reshape()` не работает – для этого нужно использовать `tf.transpose()`.

In [25]:
```
print(tf.reshape(rank_3_tensor, [4, 3, 2]))
```

```
tf.Tensor(
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]

 [[13 14]
  [15 16]
  [17 18]]

 [[19 20]
  [21 22]
  [23 24]]], shape=(4, 3, 2), dtype=int32)
```

In [26]:
```
print(tf.reshape(rank_3_tensor, [6, 4]))
```

```
tf.Tensor(
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]], shape=(6, 4), dtype=int32)
```

In [27]:
```
try:
  tf.reshape(rank_3_tensor, [7, -1])
except Exception as e:
  print(f"{type(e).__name__}: {e}")
```

```
InvalidArgumentError: Input to reshape is a tensor with 24 values,
but the requested shape requires a multiple of 7 [Op:Reshape]
```

Переменные Tensorflow

Переменная TensorFlow представляет собой объект для хранения переменных данных, которые изменяют свое значение и оптимизируются в ходе обучения нейронной сети.

Переменные в TensorFlow создаются и отслеживаются с помощью класса `tf.Variable` . Экземпляры класса `tf.Variable` представляют собой тензоры, значения которых можно изменять, применив к ним функции. Модули более высокого уровня, такие как `tf.keras` , используют объекты `tf.Variable` для хранения параметров модели нейронной сети.

Создание переменных

Чтобы создать переменную TensorFlow, необходимо указать ее начальное значение, причем объект `tf.Variable` будет иметь тот же тип `dtype` , что и значение инициализации.

In [28]:
```
my_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
my_variable = tf.Variable(my_tensor)
my_variable
```

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

Переменные, как и тензоры, могут иметь различные типы (включая логический и комплексный типы):

In [29]:
```
bool_variable = tf.Variable([False, False, False, True])
bool_variable
```

Out[29]:
```
<tf.Variable 'Variable:0' shape=(4,) dtype=bool, numpy=array([Fals
e, False, False,  True])>
```

Переменная TensorFlow используется как тензор и, по сути, является структурой данных для хранения объекта `tf.Tensor` . У переменных, как и у тензоров, есть тип (`dtype`) и форма (`shape`), и их можно экспортировать в NumPy.

In [30]:
```
print("Форма:", my_variable.shape)
print("Тип:  ", my_variable.dtype)
print("NumPy:", my_variable.numpy())
```

```
Форма: (2, 2)
Тип:   <dtype: 'float32'>
NumPy: [[1. 2.]
 [3. 4.]]
```

Большинство тензорных операций применимо к переменным TensorFlow:

In [31]:
```
print("Переменная:", my_variable)
print("\nПреобразована в тензор:", tf.convert_to_tensor(my_variable
print("\nИндекс максимального значения:", tf.argmax(my_variable))
```

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

Преобразована в тензор: tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)

Индекс максимального значения: tf.Tensor([1 1], shape=(2,), dtype=
int64)
```

Переменные не могут изменить форму, точнее при изменении формы возвращается новый объект (тензор):

In [32]:
```
print("\nИзменение формы: ", tf.reshape(my_variable, [1,4]))
```

```
Изменение формы:  tf.Tensor([[1. 2. 3. 4.]], shape=(1, 4), dtype=f
loat32)
```

Как отмечалось выше, переменные хранят тензоры. Можно переназначить тензор, связанный с переменной, используя метод `assign()` . Вызов метода `assign()` обычно не создает новый тензор, вместо этого повторно используется память существующего тензора.

In [33]:
```
a_variable = tf.Variable([2.0, 3.0])
a_variable.assign([1, 2])
a_variable
```

Out[33]:
```
<tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([1
., 2.], dtype=float32)>
```

Обратите внимание, что сохранился тип переменной `float32` . Форма переменной не может быть изменена:

In [34]:
```
try:
  a_variable.assign([1.0, 2.0, 3.0])
except Exception as e:
  print(f"{type(e).__name__}: {e}")
```

```
ValueError: Cannot assign value to variable ' Variable:0': Shape m
ismatch.The variable shape (2,), and the assigned value shape (3,)
are incompatible.
```

Создание новых переменных из существующих создает копии тензоров, т.е. две переменные не будут использовать одни и те же ячейки памяти.

In [35]:
```
a_var = tf.Variable([2.0, 3.0])
b_var = tf.Variable(a_var)
a_var.assign([5, 6])

print('a =', a_var.numpy())
print('b =', b_var.numpy())
```

```
a = [5. 6.]
b = [2. 3.]
```

Также могут использоваться другие версии метода `assign` , например:

In [36]:
```
print(a_var.assign_add([2,3]).numpy())  # добавление
print(a_var.assign_sub([7,9]).numpy())  # вычитание
```

```
[7. 9.]
[0. 0.]
```

Имена переменных

Объект класса `tf.Variable` имеет тот же жизненный цикл, что и другие объекты Python. Когда отсутствуют ссылки на переменную, она автоматически освобождается.

Переменные TensorFlow могут иметь имена (`name`), что помогает в отслеживании и отладке переменных. Двум переменным может быть дано одно и то же имя.

In [37]:
```
a_var = tf.Variable(my_tensor, name="Test")
b_var = tf.Variable(my_tensor + 1, name="Test")
print('\na =', a_var)
print('\nb =', b_var)
```

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

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

К указанному в конструкторе имени переменной добавляется текст `":0"` , что связано с тем, что конструктор класса вернул одну переменную. Когда функция TensorFlow будет возвращать несколько переменных, к имени будет добавляться текст `":0"` , `":1"` и т.д. Хотя две созданные переменные имеют одно и то же имя, они не рассматриваются как равные:

In [38]:
```
print(a_var == b_var)
```

```
tf.Tensor(
[[False False]
 [False False]], shape=(2, 2), dtype=bool)
```

По умолчанию переменные автоматически получают уникальные имена, поэтому не нужно назначать их самостоятельно, если имена переменных не будут использоваться в программе.

Вычисление градиентов

Обучение нейронных сетей обычно основывается на градиентных методах. Чтобы вычислить градиент, TensorFlow запоминает, какие операции и в каком порядке происходят во время прямого прохода по нейронной сети. Затем, TensorFlow проходит узлы нейронной сети в обратном порядке для вычисления градиента.

Для запоминания операций в TensorFlow используется **лента** – класс `tf.GradientTape` . В качестве простого примера рассмотрим вычисление градиента (производной) функции $y = x^2$ в точке $x = 5$:

In [39]:
```
x = tf.Variable(5.0)

with tf.GradientTape() as tape:
  y = x**2
```

После того, как операции записаны, можно использовать метод `GradientTape.gradient(target, sources)` для вычисления градиента некоторой функции (цели) `target` относительно источников (переменных) `sources`. В нашем случае производная функции $y(x)$ по x в точке $x = 5$ равна $\frac{dy}{dx} = 2x = 10$:

In [40]:
```
dydx = tape.gradient(y, x)
dydx.numpy()
```

Out[40]:
```
9.999999
```

В приведенном выше примере использовались скалярные величины, но класс `tf.GradientTape` так же легко работает с любым тензором:

In [41]:
```
Variable(tf.random.normal((3, 2)), name='w')        # случайная матри
Variable(tf.zeros(2, dtype=tf.float32), name='b')  # вектор смещений
., 2., 3.]]                                         # вектор значений

.GradientTape(persistent=True) as tape:
 @ w + b                                            # y – вектор из 2
= tf.reduce_mean(y**2)                              # результат средн
```

Чтобы получить градиент `loss` по обеим переменным TensorFlow (w и b), нужно передать их в качестве источников методу для вычисления градиента. Лента (`tape`) обладает гибкостью в отношении формата источников и будет принимать любую вложенную комбинацию списков или словарей и возвращать градиент, структурированный таким же образом. Например, можно использовать список:

In [42]:
```
dldw, dldb = tape.gradient(loss, [w, b])
print('\ndl/dw =', dldw)
print('\ndl/db =', dldb)
```

```
dl/dw = tf.Tensor(
[[-0.55490613 -1.7482864 ]
 [-1.1098123  -3.4965727 ]
 [-1.6647184  -5.244859  ]], shape=(3, 2), dtype=float32)

dl/db = tf.Tensor([-0.55490613 -1.7482864 ], shape=(2,), dtype=flo
at32)
```

Можно расчитать градиент по словарю переменных:

In [43]:
```
my_vars = {'w': w, 'b': b}

grad = tape.gradient(loss, my_vars)
print('w ->', grad['w'])
print('b ->', grad['b'])
```

```
w -> tf.Tensor(
[[-0.55490613 -1.7482864 ]
 [-1.1098123  -3.4965727 ]
 [-1.6647184  -5.244859  ]], shape=(3, 2), dtype=float32)
b -> tf.Tensor([-0.55490613 -1.7482864 ], shape=(2,), dtype=float3
2)
```

По умолчанию ресурсы, удерживаемые лентой (`GradientTape`), освобождаются, как только вызывается метод `GradientTape.gradient()` . Чтобы вычислить несколько градиентов по одной ленте, можно создать постоянную ленту градиента (`persistent=True`). Это позволяет многократно вызывать метод градиента, а ресурсы высвобождаются, когда объект ленты удаляется сборщиком мусора.

In [44]:
```
del tape
```

Управление лентой

Поведение ленты по умолчанию — это записывать все операции после использования обучаемой переменной класса `tf.Variable` . Причинами этого являются:

- Лента должна знать, какие операции записывать при прямом проходе, чтобы вычислять градиенты при обратном проходе.
- Лента содержит ссылки на промежуточные результаты, поэтому вам не требуется записывать ненужные операции.
- Наиболее распространенный вариант использования ленты включает вычисление градиента потерь по отношению ко всем обучаемым переменным модели.

Например, в примерах ниже градиент не может быть рассчитан, потому что объект класса `tf.Tensor` по умолчанию не отслеживается лентой, а переменная `tf.Variable` может быть не обучаемой:

In [45]:
```
# обучаемая переменная
x0 = tf.Variable(1.0, name='x0')
# не обучаемая переменная
x1 = tf.Variable(1.0, name='x1', trainable=False)
# не переменная, а тензор, т.к. переменная + значение возвращает те
x2 = tf.Variable(1.0, name='x2') + 1.0
# не переменная, а тензор
x3 = tf.constant(3.0, name='x3')

with tf.GradientTape() as tape:
  y = (x0**2) + (x1**2) + (x2**2) + (x3**2)

grad = tape.gradient(y, [x0, x1, x2, x3])

for g in grad:
  print(g)
```

```
tf.Tensor(2.0, shape=(), dtype=float32)
None
None
None
```

Можно составить список из переменных, отслеживаемых лентой, используя метод `GradientTape.watched_variables` :

In [46]:
```
[var.name for var in tape.watched_variables()]
```

Out[46]:
```
['x0:0']
```

Лента предоставляет пользователю средства контроля над тем, что отслеживается лентой, а что нет. Например, чтобы записать градиенты относительно объекта класса `tf.Tensor` , нужно вызвать метод `GradientTape.watch` :

In [47]:
```
x = tf.constant(2.0)              # x - тензор
with tf.GradientTape() as tape:
  tape.watch(x)
  y = x**3

dydx = tape.gradient(y, x)        # dy/dx = 3x^2
print('dy/dx =', dydx.numpy())
```

```
dy/dx = 12.0
```

И наоборот, чтобы отключить поведение по умолчанию с отслеживанием всех объектов класса `tf.Variables` , можно установить флажок `watch_accessed_variables=False` при создании ленты градиента. В расчете ниже используется две переменные, но отслеживается градиент только для одной из переменных:

In [48]:
```
x0 = tf.Variable(0.0)
x1 = tf.Variable(1.0)

with tf.GradientTape(watch_accessed_variables=False) as tape:
  tape.watch(x1)
  y0 = tf.math.sin(x0)
  y1 = tf.nn.relu(x1)
  y = y0 + y1
  ys = tf.reduce_sum(y)
```

Поскольку метод `GradientTape.watch` не вызывался для переменной `x0`, по отношению к ней градиент не вычисляется:

In [49]:
```
grad = tape.gradient(ys, {'x0': x0, 'x1': x1})

print('dy/dx0:', grad['x0'])
print('dy/dx1:', grad['x1'].numpy())
```

```
dy/dx0: None
dy/dx1: 1.0
```

Ленты `GradientTapes` можно вкладывать друг в друга для вычисления производных более высокого порядка. Например:

In [50]:
```
x = tf.constant(3.0)

with tf.GradientTape() as gt1:
  gt1.watch(x)
  with tf.GradientTape() as gt2:
    gt2.watch(x)
    y = x * x
  dydx = gt2.gradient(y, x)     # dy/dx = 2 * x
d2ydx2 = gt1.gradient(dydx, x)  # d2y/dx2 = 2

print('dy/dx =', dydx)
print('d2y/dx2 =', d2ydx2)
```

```
dy/dx = tf.Tensor(6.0, shape=(), dtype=float32)
d2y/dx2 = tf.Tensor(2.0, shape=(), dtype=float32)
```

Использование промежуточных результатов

Можно также использовать градиенты относительно промежуточных значений, вычисленных внутри контекста ленты `tf.GradientTape`:

In [51]:
```
x = tf.constant(2.0)

with tf.GradientTape() as tape:
  tape.watch(x)
  y = x * x  # y = x^2
  z = y * y  # z = y^2 = x^4

print(tape.gradient(z, y).numpy())  # y = x^2 = 4, dz/dy = 2 y = 8
```

```
8.0
```

По умолчанию ресурсы ленты освобождаются, как только вызывается метод `GradientTape.gradient`. Чтобы вычислить несколько градиентов, можно создавать ленту градиента с флажком `persistent=True`. Это позволяет многократно вызывать метод градиента, поскольку ресурсы высвобождаются при сборке мусора для объекта ленты. Например:

In [52]:
```
x = tf.constant([1, 3.0])

with tf.GradientTape(persistent=True) as tape:
  tape.watch(x)
  y = x * x
  z = y * y

print(tape.gradient(z, x).numpy())  # 4. и 108. (4x^3 при x=1 и x=3
print(tape.gradient(y, x).numpy())  # 2. и 6.   (2x при x=1 и x=3)
```

```
[  4. 108.]
[2. 6.]
```

Для сборки мусора выполним команду `del`:

In [53]:
```
del tape
```

Градиенты для векторных величин

Градиент в TensorFlow - это принципиально операция над скалярной величиной:

In [54]:
```
x = tf.Variable(2.0)

with tf.GradientTape(persistent=True) as tape:
  y1 = x**2
  y2 = 1 / x

print('dy1/dx =', tape.gradient(y1, x).numpy())
print('dy2/dx =', tape.gradient(y2, x).numpy())
```

```
dy1/dx = 4.0
dy2/dx = -0.25
```

Градиент от нескольких функций вычисляется как градиент суммы функций или эквивалентно как сумма градиентов каждой функции:

In [55]:
```
x = tf.Variable(2.0)

with tf.GradientTape() as tape:
  y1 = x**2
  y2 = 1 / x

print(tape.gradient({'y1': y1, 'y2': y2}, x).numpy())
```

```
3.75
```

Точно так же, если функция является векторной, то вычисляется градиент суммы компонентов вектора:

In [56]:
```
x = tf.Variable(2.)

with tf.GradientTape() as tape:
  y = x * [3., 4.]
  print('y =', y)

print(tape.gradient(y, x).numpy())
```

```
y = tf.Tensor([6. 8.], shape=(2,), dtype=float32)
7.0
```

Ситуации, когда градиент не может быть вычислен

Возможно возникновение ряда ситуаций, когда градиент не может быть вычислен.

Поток управления

Лента градиента записывает операции по мере их выполнения в соответствии с потоком управления Python (операторами `if`, `while` и т.п.). В примере в каждой ветви `if` используется другая переменная, но для расчета градиента записывается только совершенная операция с переменной:

In [57]:
```
x = tf.constant(1.0)

v1 = tf.Variable(2.0)
v2 = tf.Variable(2.0)

with tf.GradientTape(persistent=True) as tape:
  tape.watch(x)
  if x > 0.0:
    result = v1
  else:
    result = v2**2

dv1, dv2 = tape.gradient(result, [v1, v2])

print('1 ->', dv1)
print('2 ->', dv2)
```

```
1 -> tf.Tensor(1.0, shape=(), dtype=float32)
2 -> None
```

Операторы управления сами по себе не дифференцируемы и не видимы для оптимизаторов на основе градиента. В приведенном выше примере в зависимости от значения `x` на ленту записывается либо `result = v0`, либо `result = v1**2`. Градиент по `x` всегда равен `None`:

In [58]:
```
dx = tape.gradient(result, x)

print(dx)
```

```
None
```

Отсутствие зависимости от переменной

Когда результат вычислений не связан с переменной, градиент будет равен `None`:

In [59]:
```
x = tf.Variable(1.)
y = tf.Variable(2.)

with tf.GradientTape() as tape:
  z = y * y

print(tape.gradient(z, x))
```

```
None
```

Тензор вместо переменной

По умолчанию лента градиента отслеживает операции с переменными, но не тензорами. Поэтому градиент не вычисляется, если по ошибке переменная заменена на тензор:

In [60]:
```
x = tf.Variable(2.0)

for epoch in range(2):
  with tf.GradientTape() as tape:
    y = x+1

  print(epoch, '->', type(x).__name__, ":", tape.gradient(y, x))
  x = x + 1   # правильно `x.assign_add(1)`
```

```
0 -> ResourceVariable : tf.Tensor(1.0, shape=(), dtype=float32)
1 -> EagerTensor : None
```

Вычисления за пределами TensorFlow

Лента не может записать вычисления, если они производятся вне TensorFlow, например, в NumPy:

In [61]:
```
x = tf.Variable([[1.0, 2.0],
                 [3.0, 4.0]], dtype=tf.float32)

with tf.GradientTape() as tape:
  x2 = x**2

  y = np.mean(x2, axis=0) # вычисления в NumPy

  y = tf.reduce_mean(y, axis=0) # массив NumPy конвертируется в тен

print(tape.gradient(y, x))
```

```
None
```

Градиент по целому числу или строке

Целые числа и строки не являются дифференцируемыми в TensorFlow:

In [62]:
```
x = tf.constant(10)

with tf.GradientTape() as gt:
  gt.watch(x)
  y = x * x

print(gt.gradient(y, x))
```

```
WARNING:tensorflow:The dtype of the watched tensor must be floatin
g (e.g. tf.float32), got tf.int32
WARNING:tensorflow:The dtype of the target tensor must be floating
(e.g. tf.float32) when calling GradientTape.gradient, got tf.int32
WARNING:tensorflow:The dtype of the source tensor must be floating
(e.g. tf.float32) when calling GradientTape.gradient, got tf.int32
None
```

Модуль Keras

Keras представляет собой высокоуровневый программный интерфейс (API) к фреймворку TensorFlow 2 для решения задач машинного обучения с акцентом на глубокое обучение. Keras предоставляет необходимые абстракции и строительные блоки для разработки и внедрения решений с применением машинного обучения.

Keras позволяет инженерам и исследователям в полной мере воспользоваться масштабируемостью и кроссплатформенными возможностями TensorFlow 2: можно запускать Keras на TPU или на больших кластерах графических процессоров, а также экспортировать модели Keras для запуска в браузере или на мобильном устройстве.

Если TensorFlow — это инфраструктура для дифференцируемого программирования, работающая с тензорами, переменными и градиентами, то Keras — это пользовательский интерфейс для глубокого обучения, работающий со слоями (layers), моделями (models), оптимизаторами (optimizers), функциями потерь (loss functions), метриками (metrics) и многим другим.

Слой (класс `Layer`) является фундаментальной абстракцией в Keras. Слой инкапсулирует состояние (веса и смещения) и некоторые вычисления, производимые в слое.

Модель (класс `Model`) группирует слои в объект с функциями обучения и вывода (вычисления выходных данных по входным данным модели).

Простейшим примером модели является последовательная модель (класс `Sequential`), группирующая слои в линейный стек.

Для более сложных архитектур нейронных сетей можно использовать функциональный API Keras, который позволяет строить произвольные графы слоев, или создать модель полностью с нуля при помощи производных классов класса `Model`.

Последовательная модель (Sequential)

Последовательная модель подходит для простого стека слоев, где каждый слой имеет ровно один входной тензор и один выходной тензор.

Приведем схематичный пример последовательной модели с тремя слоями и вызовем ее для тестовых входных данных:

In [63]:
```
model = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(2, activation="relu", name="layer1"),
        tf.keras.layers.Dense(3, activation="relu", name="layer2"),
        tf.keras.layers.Dense(4, name="layer3"),
    ]
)

x = tf.ones((5, 3)) # тестовые входные данные из единиц размерами 5
y = model(x)
y
```

Out[63]:
```
<tf.Tensor: shape=(5, 4), dtype=float32, numpy=
array([[ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ]],
      dtype=float32)>
```

Здесь класс `Dense` представляет собой реализацию регулярного плотного слоя нейронной сети с операцией `output = activation(dot(input, kernel) + bias)`, где `activation` - поэлементная функция активации, `kernel` - это матрица весов, созданная слоем, `bias` - созданный вектор смещения слоя.

Созданная в результате модель эквивалентна вызову:

In [64]:
```
# создаем 3 слоя
layer1 = tf.keras.layers.Dense(2, activation="relu", name="layer1")
layer2 = tf.keras.layers.Dense(3, activation="relu", name="layer2")
layer3 = tf.keras.layers.Dense(4, name="layer3")

# вызываем слои для входных данных
x = tf.ones((5, 3))
y = layer3(layer2(layer1(x)))
y
```

Out[64]:
```
<tf.Tensor: shape=(5, 4), dtype=float32, numpy=
array([[ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ],
       [ 0.23215866, -2.0438924 ,  1.9299359 , -1.0872209 ]],
      dtype=float32)>
```

Последовательная модель не подходит для случаев, когда:

- модель имеет несколько входов или несколько выходов
- какой-либо из слоев модели имеет несколько входов или несколько выходов
- требуется общий доступ к слою
- требуется нелинейная топология

Создание последовательной модели

Можно создать последовательную модель, передав список слоев конструктору класса `Sequential`:

In [65]:
```
model = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(2, activation="relu"),
        tf.keras.layers.Dense(3, activation="relu"),
        tf.keras.layers.Dense(4),
    ]
)
```

Слои модели доступны через атрибут `layers`:

In [66]:
```
model.layers
```

Out[66]:
```
[<keras.layers.core.dense.Dense at 0x158d284c0>,
 <keras.layers.core.dense.Dense at 0x158d3f4f0>,
 <keras.layers.core.dense.Dense at 0x158d3f880>]
```

Также можно создать последовательную модель постепенно (по шагам) с помощью метода `add()`:

In [67]:
```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(2, activation="relu"))
model.add(tf.keras.layers.Dense(3, activation="relu"))
model.add(tf.keras.layers.Dense(4))
```

Существует также метод `pop()` для удаления последнего слоя

In [68]:
```
model.pop()
print(len(model.layers))
```

```
2
```

Модели и слоям можно давать осмысленные имена при помощи аргумента `name`:

In [69]:
```
model = tf.keras.Sequential(name="my_model")
model.add(tf.keras.layers.Dense(2, activation="relu", name="layer1"
model.add(tf.keras.layers.Dense(3, activation="relu", name="layer2"
model.add(tf.keras.layers.Dense(4, name="layer3"))
```

Задание формы входных данных

Как правило, слои в Keras должны знать форму своих входных данных, чтобы иметь возможность создать свои параметры (веса). Когда слой создается, изначально он не имеет весов:

In [70]:
```
layer = tf.keras.layers.Dense(3)
layer.weights
```

Out[70]:
```
[]
```

Веса создаются при первом обращении к слою с входными данными, поскольку форма весов зависит от формы входных данных:

In [71]:
```
x = tf.ones((1, 4))
y = layer(x)
layer.weights # 3 нейрона, входные данные формы (4,)
```

Out[71]:
```
[<tf.Variable 'dense_6/kernel:0' shape=(4, 3) dtype=float32, numpy
=
 array([[-0.78281665, -0.04741734,  0.76290786],
        [-0.1265679 , -0.234272  ,  0.31430423],
        [-0.7213854 , -0.44200754, -0.8501039 ],
        [-0.8821614 ,  0.4276712 , -0.4549957 ]], dtype=float32)>,
 <tf.Variable 'dense_6/bias:0' shape=(3,) dtype=float32, numpy=arr
ay([0., 0., 0.], dtype=float32)>]
```

Это же относится и к последовательной модели. Когда создается последовательная модель без указания формы входных данных, модель еще не построена - у нее нет весов. Веса создаются, когда модель впервые видит входные данные:

In [72]:
```
model = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(2, activation="relu"),
        tf.keras.layers.Dense(3, activation="relu"),
        tf.keras.layers.Dense(4),
    ]
)

x = tf.ones((1, 4))
y = model(x)
model.weights
```

Out[72]:
```
[<tf.Variable 'dense_7/kernel:0' shape=(4, 2) dtype=float32, numpy
=
 array([[ 0.24774432,  0.73120403],
        [-0.6802559 ,  0.08526611],
        [-0.7091105 , -0.3321867 ],
        [ 0.1482575 , -0.6318102 ]], dtype=float32)>,
 <tf.Variable 'dense_7/bias:0' shape=(2,) dtype=float32, numpy=arr
ay([0., 0.], dtype=float32)>,
 <tf.Variable 'dense_8/kernel:0' shape=(2, 3) dtype=float32, numpy
=
 array([[-0.8020159,  0.7669101,  0.0343653],
        [ 1.0051837,  1.0598016,  0.8299874]], dtype=float32)>,
 <tf.Variable 'dense_8/bias:0' shape=(3,) dtype=float32, numpy=arr
ay([0., 0., 0.], dtype=float32)>,
 <tf.Variable 'dense_9/kernel:0' shape=(3, 4) dtype=float32, numpy
=
 array([[ 0.89354765,  0.7139703 ,  0.7641945 , -0.30922115],
        [ 0.08048248, -0.6754105 ,  0.5227494 , -0.8283121 ],
        [-0.7660275 , -0.84779024, -0.28379387,  0.88698816]],
       dtype=float32)>,
 <tf.Variable 'dense_9/bias:0' shape=(4,) dtype=float32, numpy=arr
ay([0., 0., 0., 0.], dtype=float32)>]
```

После того, как модель построена, можно вызвать метод `summary()`, чтобы отобразить ее содержимое:

In [73]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_7 (Dense)             (1, 2)                    10

 dense_8 (Dense)             (1, 3)                    9

 dense_9 (Dense)             (1, 4)                    16

=================================================================
Total params: 35
Trainable params: 35
Non-trainable params: 0
_________________________________________________________________
```

Иначе, после создания модели при помощи конструктора можно вызвать метод `build()`, указав в качестве аргумента форму входных данных:

In [74]:
```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(2, activation="relu"))
model.add(tf.keras.layers.Dense(3, activation="softmax"))
model.build(input_shape=(None,4))
model.weights
```

Out[74]:
```
[<tf.Variable 'dense_10/kernel:0' shape=(4, 2) dtype=float32, numpy=
 array([[ 0.24774432,  0.73120403],
        [-0.6802559 ,  0.08526611],
        [-0.7091105 , -0.3321867 ],
        [ 0.1482575 , -0.6318102 ]], dtype=float32)>,
 <tf.Variable 'dense_10/bias:0' shape=(2,) dtype=float32, numpy=array([0., 0.], dtype=float32)>,
 <tf.Variable 'dense_11/kernel:0' shape=(2, 3) dtype=float32, numpy=
 array([[-0.8020159,  0.7669101,  0.0343653],
        [ 1.0051837,  1.0598016,  0.8299874]], dtype=float32)>,
 <tf.Variable 'dense_11/bias:0' shape=(3,) dtype=float32, numpy=array([0., 0., 0.], dtype=float32)>]
```

Иногда при построении последовательной модели бывает полезно иметь возможность отображать сводку модели на данный момент, включая текущую выходную форму. В этом случае можно передать модели входной объект (класса `Input`), чтобы она знала свою форму входных данных с самого начала:

In [75]:
```
model = tf.keras.Sequential()
model.add(tf.keras.Input(shape=(4,)))
model.add(tf.keras.layers.Dense(2, activation="relu"))

model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_12 (Dense)            (None, 2)                 10

=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________
```

Обратите внимание, что объект `Input` не является слоем и не отображается как слой:

In [76]:
```
model.layers
```

Out[76]:
```
[<keras.layers.core.dense.Dense at 0x158d55af0>]
```

Простая альтернатива состоит в том, чтобы просто передать аргумент `input_shape` первому слою нейронной сети:

In [77]:
```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(10, activation='tanh', input_shape=
model.add(tf.keras.layers.Dense(1))

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_13 (Dense)            (None, 10)                20

 dense_14 (Dense)            (None, 1)                 11

=================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
_________________________________________________________________
```

Модели, построенные с заранее определенной формой входных данных, всегда имеют веса в слоях (даже до того, как модели будут показаны какие-либо входные данные) и всегда имеют определенную форму выходных данных.

Рекомендуется заранее указывать форму входных данных последовательной модели, если она известна.

Использование последовательной модели

После того, как последовательная модель построена, процесс ее обучения должен быть настроен с помощью метода `compile()`, например:

In [78]:
```
model.compile(loss='mse',
              optimizer='sgd',
              metrics=['mae'])
```

Здесь указано, что в качестве функции потерь используется среднеквадратичная ошибка (MSE), в качестве оптимизатора – стохастический градиентный спуск (SGD), а в качестве метрики – показатель средней абсолютной ошибки (MAE).

При необходимости можно дополнительно настроить процесс обучения, указывая параметры оптимизатора:

In [79]:
```
model.compile(loss=tf.keras.losses.mean_squared_error,
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=
    metrics=[tf.keras.metrics.MeanAbsoluteError()])
```

Чтобы обучить последовательную модель на имеющихся входных данных, можно использовать метод `fit()`:

In [80]:
```
x_train = 20 * np.random.random(200) - 10
y_train = np.sin(x_train)

model.fit(x_train, y_train, epochs=500, batch_size=32, verbose=0)
```

```
2022-05-06 10:37:37.377394: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:113] Plugin optimizer for devic
e_type GPU is enabled.
```

Out[80]:
```
<keras.callbacks.History at 0x158d47790>
```

Чтобы оценить ошибку (потери) на тестовой выборке и качество модели по заданной метрике, можно использовать метод `evaluate()`:

In [81]:
```
x_test = np.linspace(-10,10,1001)
y_test = np.sin(x_test)

loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
loss_and_metrics
```

```
8/8 [==============================] - 0s 6ms/step - loss: 0.1035
- mean_absolute_error: 0.2346

2022-05-06 10:37:58.377070: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:113] Plugin optimizer for devic
e_type GPU is enabled.
```

Out[81]:
```
[0.10346746444702148, 0.23459206521511078]
```

Далее можно прогнозировать значения для новых данных, используя метод `predict()`:

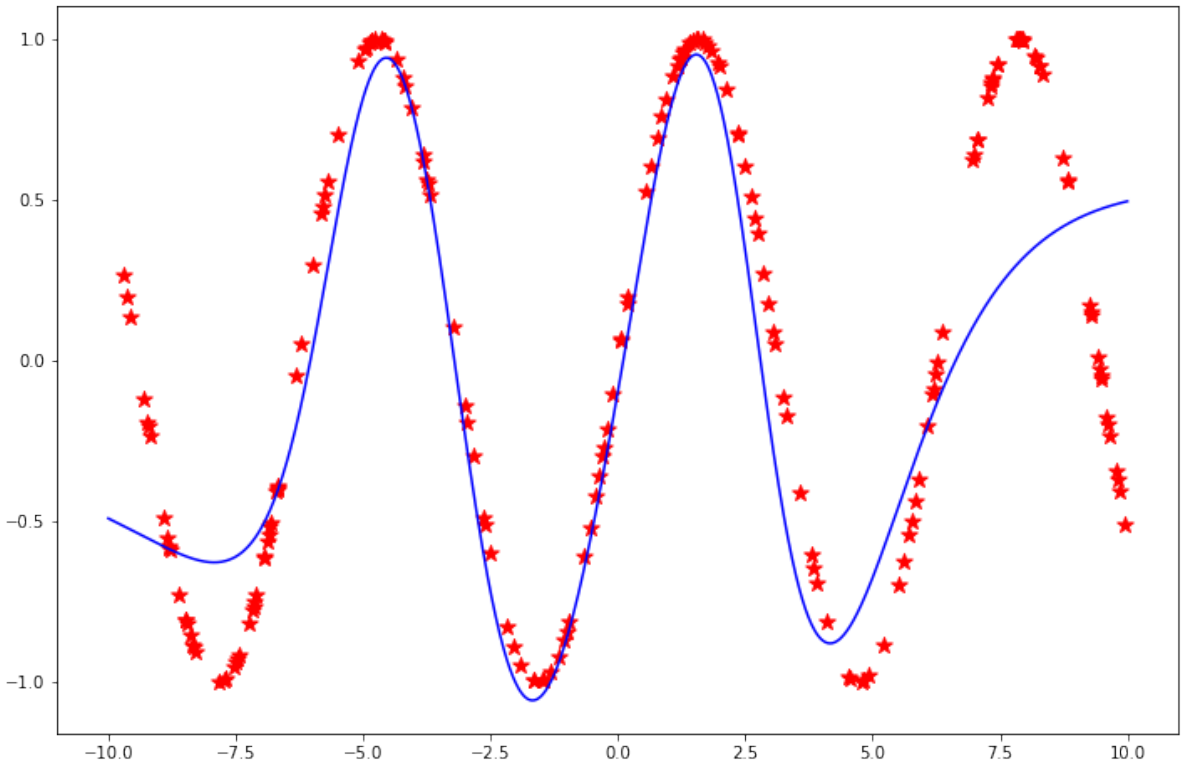
In [82]:
```
y_predict = model.predict(x_test, batch_size=128)
```

```
2022-05-06 10:37:58.566142: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:113] Plugin optimizer for devic
e_type GPU is enabled.
```

Визуализируем результаты прогноза и обучающие данные:

In [83]:
```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12,8)

plt.plot(x_test, y_predict, c='b')
plt.scatter(x_train, y_train, s=100, c='r', marker='*');
```



Keras также можно использовать для разработки новых процедур обучения нейронной сети или создания сложных архитектур нейронных сетей. Например, низкоуровневый цикл обучения нейронной сети может выглядеть так (для одной эпохи обучения):

In [ ]:

```
# подготовить оптимизатор
optimizer = tf.keras.optimizers.Adam()
# подготовить функцию потерь
loss_fn = tf.keras.losses.mean_squared_error

# проходим по пакетам (батчам) датасета
for inputs, targets in dataset:
    # создаем ленту градиента
    with tf.GradientTape() as tape:
        # прямой проход
        predictions = model(inputs)
        # вычисляем функцию потерь для пакета данных
        loss_value = loss_fn(targets, predictions)

    # вычисляем градиенты функции потерь по весам модели
    gradients = tape.gradient(loss_value, model.trainable_weights)
    # обновляем веса модели
    optimizer.apply_gradients(zip(gradients, model.trainable_weight
```

В качестве модели в приведенном выше цикле обучения можно использовать:

- последовательную модель
- модель на основе функционального API, позволяющего строить произвольные графы типовых слоев Keras
- модель с произвольными слоями, разработанными при помощи производных классов