

TensorFlow 2

1. Tensor

Tensor是各个深度学习框架中的核心数据结构，它用来表示高维数组。

1.1 张量创建

张量是一种Tensorflow特有的数据容器。创建张量和在Python中创建 list , tuple 等本质是一致的。

- `def constant(value, dtype=None, shape=None, name="Const")`
 - `value` : 张量的内容，可以是Python中的类型，也可以是Numpy中的类型
 - `dtype` : 数据类型，可以从 `tf.dtypes` 中寻找需要的数据类型
 - `name` : 张量的名称，实际是一个操作的名称

示例：

```
1 import numpy as np
2 import tensorflow as tf
3
4 # 传入一个普通的数值
5 x = tf.constant(1.2)
6 tf.print(x)
7
8 # 传入一个Python list
9 x = tf.constant([2, 3.2, 4])
10 tf.print(x)
11
12 # 传入一个Python tuple
13 x = tf.constant((1, 2, 3))
14 tf.print(x)
15
16 # 传入一个Numpy的ndarray数组
17 x = tf.constant(np.array([1, 2, 3]))
18 tf.print(x)
```

这里需要注意的是：

1. `dtype` 指定后，如果类型不匹配的话是会跑出异常的，例如以下代码：

```
1 x = tf.constant([1, 2, 3.2], dtype=tf.int32) # 因为3.2是浮点型，但是dtype是整型不能进行强制转换
```

2. 不能够在定义Tensor后，是不能够获取一些属性的，比如 `name` , `op` 等属性的，因为此时使用的Eager Execution模式。

```
1 x_name = x.name
2
3 AttributeError: Tensor.name is undefined when eager execution is enabled.
```

在计算图模式：

```

1 @tf.function
2 def f():
3     x = tf.constant([1, 2, 3])
4     print(x.name)
5 f()

```

输出:

```

1 Const:0

```

■ 创建特殊的张量

1. `def ones(shape, dtype=dtypes.float32, name=None)`
2. `def zeros(shape, dtype=dtypes.float32, name=None)`
3. `def fill(dims, value, name=None)`
4. `def eye(num_rows, num_columns=None, batch_shape=None, dtype=dtypes.float32, name=None)`
5. `def linspace_nd(start, stop, num, name=None, axis=0)`
6. `def range(start, limit=None, delta=1, dtype=None, name="range")`

例如:

```

1 import tensorflow as tf
2
3 x = tf.ones(shape=[2, 3])
4 print(x)
5
6 x = tf.zeros(shape=[3, 2])
7 print(x)
8
9 x = tf.fill([2, 2], 3.3)
10 print(x)
11
12 x = tf.eye(2)
13 print(x)
14
15 x = tf.linspace(0, 100, 5)
16 print(x)
17
18 x = tf.range(0, 100, 2)
19 print(x)

```

输出:

```

1 tf.Tensor(
2 [[1. 1. 1.]
3  [1. 1. 1.]], shape=(2, 3), dtype=float32)
4 tf.Tensor(
5 [[0. 0.]
6  [0. 0.]
7  [0. 0.]], shape=(3, 2), dtype=float32)
8 tf.Tensor(
9 [[3.3 3.3]
10  [3.3 3.3]], shape=(2, 2), dtype=float32)

```

```

11 tf.Tensor(
12  [[1.  0.]
13   [0.  1.]], shape=(2, 2), dtype=float32)
14 tf.Tensor([ 0.  25.  50.  75. 100.], shape=(5,), dtype=float64)
15 tf.Tensor(
16  [ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
17   48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94
18   96 98], shape=(50,), dtype=int32)

```

- 复制张量 `xxx_like`

- 随机张量，在 `tf.random` 模块下

1. `def random_normal(shape, mean=0.0, stddev=1.0 dtype=dtypes.float32, seed=None, name=None)`

该函数可以生成一个服从正态随机分布的张量，概率密度函数为

$$f(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2} (X-\mu)^T \Sigma^{-1} (X-\mu)} \quad (1)$$

```

1 import tensorflow as tf
2
3 x = tf.random.normal([5, 5])
4 print(x)

```

输出：

```

1 tf.Tensor(
2  [[ 1.0284482 -0.06882296  0.57281095 -0.52652025  1.002363 ]
3   [ 1.5339344 -0.4890544 -1.8685299  1.0711784  2.169645 ]
4   [-0.10586395 -0.7577186 -1.0247724  0.21309386 -0.6320886 ]
5   [-1.4260643 -0.6839064  0.8648016 -0.18941845  0.14517523]
6   [-0.60574925 -0.72606707 -1.0409023  0.38468215  2.2940114 ]], shape=(5, 5), dtype=float32)

```

2. `def truncated_normal(shape, mean=0.0, stddev=1.0 dtype=dtypes.float32, seed=None, name=None)`：截断的正态分布
3. `def random_uniform(shape, mean=0.0, stddev=1.0 dtype=dtypes.float32, seed=None, name=None)`：均匀分布
4. `def random_poisson(lam, shape, dtype=dtypes.float32, seed=None, name=None)`
5.

在创建张量的过程中通常需要指定一个名字，也就是 `name` 参数，这本质上和Python或Numpy中创建一个数据容器是有区别的，实际上我们创建的是一个操作（或者叫做算子op），而我们在Eager Execution的模式下可以立即执行得到结果的，具体地可以查看 `tf.Tensor` 源代码：

```

1 class Tensor:
2     """A `tf.Tensor` represents a multidimensional array of elements.
3
4     All elements are of a single known data type.
5
6     When writing a TensorFlow program, the main object that is
7     manipulated and passed around is the `tf.Tensor`.
8
9     A `tf.Tensor` has the following properties:

```

```

10
11     * a single data type (float32, int32, or string, for example)
12     * a shape
13
14     TensorFlow supports eager execution and graph execution. In eager
15     execution, operations are evaluated immediately. In graph
16     execution, a computational graph is constructed for later
17     evaluation.
18
19     TensorFlow defaults to eager execution. In the example below, the
20     matrix multiplication results are calculated immediately.
21     """
22     def __init__(self, op, value_index, dtype):
23         ...

```

由于计算图的存在，实际上我们在定义张量的时候是不能得到这个张量的值的，因为定义知识一个图纸的设计，而得到张量的值需要运行整个计算图才能获取，比如在Tensorflow 1.x版本中，计算图的定义和运行时分开的，这当然非常不利于调试。在Tensorflow 2中，有了Eager Execution模式，所以能够时刻地得到定义的张量的值，这也大大降低了性能。

说起计算图，在这里进行抛砖引玉，我们虽然利用Python的思维逻辑去创建了一个张量，看似没有任何问题，实际上是做了一个运算，意义是大不相同的，以下两行代码：

```

1 x = [1, 2, 3]
2 x = tf.constant([1, 2, 3])

```

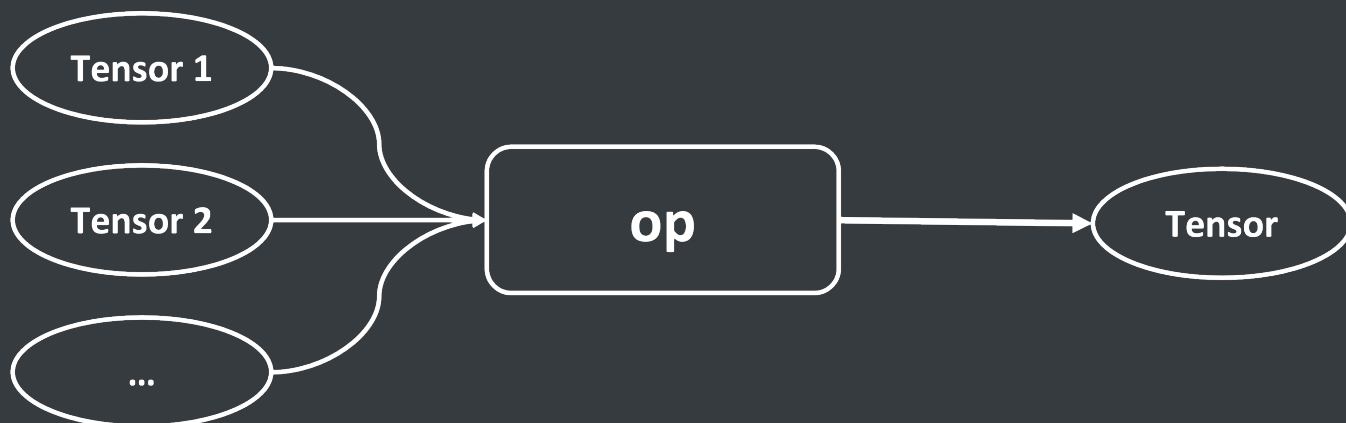
要说清楚的是，通常使用的 `tf.xxx` 都是一种操作（算子，Operation），包括 `constant`，`range`，`ones` 等。计算图是一个图（Graph）结构，它有两个概念：

- 节点（Node）：节点就是操作
- 边（Edge）：边可以看作是数据（当然不仅仅是数据，还有控制流）

计算图（Computational Graph）又是一种特殊的数据流图（Data Flow Graph），而这个数据就是张量（Tensor），所以这个框架的名字叫做TensorFlow。作为一个有向图，节点自然有入边和出边了，可以看作是函数的输入和输出，而多个节点也构成了一个新的图，就是子图，一个子图也有输入边和输出边，慢慢就汇聚成了一个大的计算图。

1.2 张量的操作

张量的操作就是计算图中的节点。这里主要总结一些著名的，常用的一些操作。



张量的操作非常多，弃用有一类就是element-wise操作，就是对张量中的每个元素进行操作，有一元操作，二元操作和多元操作，可以用如下公式描述

$$O = f(T_1, T_2, \dots) \quad (2)$$

此时，操作的结果通常与输入的形状（shape）是相同的，看如下例子：

```
1 import tensorflow as tf
2
3 x = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
4 y = tf.constant([[10, 20], [30, 40]], dtype=tf.float32)
5 # Python运算符
6 z = -x # 注意：没有 y = +x
7 print(z)
8 z = x + y # 除了+, 还有 - * / // % += -= *= /= < > <= >= != ==
9 print(z)
10 x = tf.constant([[1, 2], [3, 4]], dtype=tf.int32)
11 y = tf.constant([[10, 20], [30, 40]], dtype=tf.int32)
12 z = x & y # & | ~ ^ 没有移位运算符
13 print(z)
```

以上的一些操作和普通Python操作或Numpy的操作方式类似，个别是需要注意的，但是总的来说没有什么特殊的。

除了运算符，还有很多运算函数，比如Python内置的 `abs`，`pow` 等，也是和普通的Python函数类似，这些函数作用在张量的每个元素上。

除了built-in的函数，还有很多数学操作，三角函数等，都在 `tf.math` 包下：

- `tf.math.abs`
- `tf.math.accumulate_n`：累加多个张量

```
1 import tensorflow as tf
2
3 x = tf.constant([[ -2, 1], [2.3, -4]], dtype=tf.float32)
4 y = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
5 z = tf.constant([[10, 20], [30, 40]], dtype=tf.float32)
6
7 res = tf.math.accumulate_n([x, y, z])
8 print(res)
```

- 三角函数：`sin`，`cos`，`tan`，`asin`，`acos`，`atan`，`asinh`，`acosh`，.....
- `tf.math.angle`：计算复数的俯角 θ

$$a + bi \Rightarrow \theta = \text{atan2}(b, a) \quad (3)$$

其中`atan2(·)`为

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & , x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & , y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & , y < 0, x < 0 \\ +\frac{\pi}{2} & , y > 0, x = 0 \\ -\frac{\pi}{2} & , y < 0, x = 0 \\ \text{undefined} & , x = 0, y = 0 \end{cases} \quad (4)$$

- `tf.math.bessel_i0e`：贝塞尔值

$$i0e(x) = e^{-|x|} * i0(x) \quad (5)$$

其中 $i0(\cdot)$ 为

$$I_0(x) = \sum_{k=0}^{\infty} \frac{(x^2/4)^k}{(k!)^2} = J_0(ix) \quad (6)$$

参考：[SciPy Wiki](#)

- `tf.math.betainc`：计算Beta积分

$$I_x(a, b) = \frac{\int_0^x t^{a-1} (1-t)^{b-1} dt}{\int_0^{\infty} t^{a-1} (1-t)^{b-1} dt} \quad (7)$$

- `tf.math.bincount`：统计每个整型数值的个数

```
1 import tensorflow as tf
2
3 x = tf.constant([2, 3, 5, 3, 5, 2, 3, 3, 4], dtype=tf.int32)
4 y = tf.math.bincount(x) # ==> 0:0 1:0 2:2 3:4 4:1 5:2
```

- `tf.math.ceil`，`tf.math.floor`：取整函数
- `tf.math.conj`：共轭操作
- `tf.math.count_nonzero`：非0元素的个数

```
1 import tensorflow as tf
2
3 x = tf.constant([2, 3, 0, 0, 5, 0, 0, 0, 4], dtype=tf.int32)
4 y = tf.math.count_nonzero(x) # 4
```

- `tf.math.cumprod`：元素连乘

$$\operatorname{cumprod}(x_1, x_2, x_3, \dots, x_n) = \begin{bmatrix} x_1 & x_1 x_2 & x_1 x_2 x_3 & \cdots & \prod_{k=1}^i x_k & \cdots & \prod_{k=1}^n x_k \end{bmatrix} \quad (8)$$

```
1 import tensorflow as tf
2 x = tf.constant([[1, 2, 3], [4, 5, 6]], dtype=tf.float32)
3 y = tf.math.cumprod(x, axis=0)
4 print(y)
5 y = tf.math.cumprod(x, axis=1)
6 print(y)
```

输出:

```
1 tf.Tensor(  
2 [[ 1.  2.  3.]  
3  [ 4. 10. 18.]], shape=(2, 3), dtype=float32)  
4 tf.Tensor(  
5 [[ 1.  2.  6.]  
6  [ 4. 20. 120.]], shape=(2, 3), dtype=float32)
```

这里的轴 (axis) 的概念之后总结。

- `tf.math.cumsum` : 元素累加

$$\text{cumsum}(x_1, x_2, \dots, x_n) = \begin{bmatrix} x_1 & x_1 + x_2 & \dots & \sum_{k=1}^i x_k & \dots & \sum_{k=1}^n x_k \end{bmatrix} \quad (9)$$

- `tf.math.cumulative_logsumexp`

$$\text{cumulative_logsumexp}(x_1, x_2, \dots, x_n) = \begin{bmatrix} \log e^{x_1} & \log(e^{x_1} + e^{x_2}) & \dots & \log \sum_{k=1}^i e^{x_k} & \dots & \log \sum_{k=1}^n e^{x_k} \end{bmatrix} \quad (10)$$

- `tf.math.digamma` : Psi函数

$$\psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)} \sim \ln x - \frac{1}{2x} \quad (11)$$

- `tf.math.erf` :

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (12)$$

- `tf.math.erfc` :

$$\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \quad (13)$$

- `tf.math.expm` :

$$\text{expm}(x) = e^x - 1 \quad (14)$$

- `tf.math.igammac` 和 `tf.math.igamma` : Lower Gamma和Upper Gamma

$$P(a, x) = \frac{\Gamma(a, x)}{\Gamma(a)} = 1 - Q(a, x) \quad (15)$$

- `tf.math.is_non_decreasing` : 是否非递减序列

$$\text{is_non_decreasing}(x_1, x_2, \dots, x_n) = \mathbb{I}(x_i < x_j), \forall i, j, i \leq j \quad (16)$$

```
1 import tensorflow as tf  
2  
3 x = tf.constant([2, 3, 3, 4, 5, 7])  
4 y = tf.math.is_non_decreasing(x) # ==> True  
5 x = tf.constant([2, 3, 3, 4, 2, 7])  
6 y = tf.math.is_non_decreasing(x) # ==> False
```

- `tf.math.is_strictly_increasing`

- `tf.math.lbeta` : $\ln(|Beta(x)|)$

- `tf.math.lgamma` : $\ln(|Gamma(x)|)$

- `tf.math.log` : $\log(x)$
- `tf.math.log1p` : $\log(1+x)$
-

以上就是数学 `tf.math` 中的一些函数，但是这里还有一些函数故意跳过的，需要在后面再单独详细讲解和总结，以上有很多数学中的函数，实际上使用到的非常少。

张量中又个非常重要的操作，那就是两个张量相乘。矩阵乘法，假设有矩阵 $\mathbf{A} \in \mathbb{R}^{m \times p}$ 和矩阵 $\mathbf{B} \in \mathbb{R}^{s \times n}$ ，乘法规则如下：

$$[c_{ij}] = \sum_{k=1}^p a_{ik} b_{kj} \quad (17)$$

Tensorflow采用 `@` 运算符对矩阵乘法进行实现，自然也可以通过调用API进行使用

```
1 import tensorflow as tf
2
3 A = tf.constant([
4     [3, 2, 1],
5     [1, 4, 6]
6 ])
7
8 B = tf.constant([
9     [10, 32],
10    [14, 11],
11    [13, 8]
12 ])
13
14 print(A @ B)
15 print(tf.matmul(A, B))
```

输出：

```
1 tf.Tensor(
2 [[ 71 126]
3  [144 124]], shape=(2, 2), dtype=int32)
4 tf.Tensor(
5 [[ 71 126]
6  [144 124]], shape=(2, 2), dtype=int32)
```

在张量的运算中为了减少类型转换带来的开销，通常需要操作的多个张量之间的类型要保持一致，类型属性相关操作在 `tf.dtypes` 包下，例如：


```

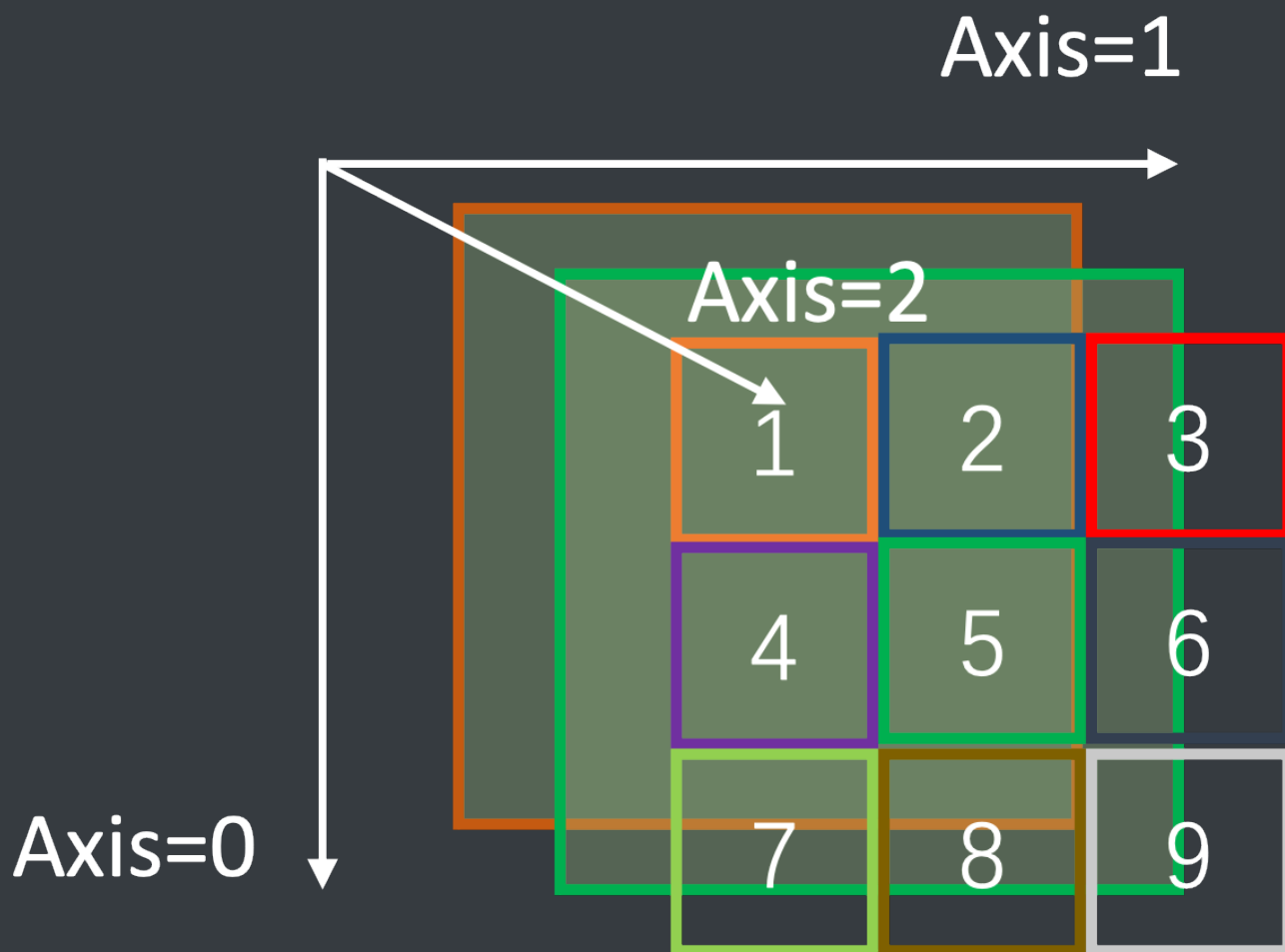
1 import tensorflow as tf
2
3 x = tf.constant([1, 2, 3], dtype=tf.int32)
4 y = tf.constant([1, 2, 3], dtype=tf.float32)
5
6 # Error
7 # print(x + y)
8
9 y = tf.cast(y, tf.int32)
10 print(x + y)

```

1.3 Tensor形状

Tensor中有一个非常关键的属性，就是**形状（shape）**，在内存中，所有的N维数组（张量）都是一个一维数组，所以，形状是Tensor的一种**视图（view）**。

首先关注下张量中轴的概念：



```

1 import tensorflow as tf
2
3 x = tf.constant(1.2)
4 tf.print('shape:', x.shape, 'rank:', tf.rank(x), 'size:', tf.size(x))
5 x = tf.constant([1, 2, 3, 4, 5])
6 tf.print('shape:', x.shape, 'rank:', tf.rank(x), 'size:', tf.size(x))
7 x = tf.constant([[1, 2], [3, 4]])
8 tf.print('shape:', x.shape, 'rank:', tf.rank(x), 'size:', tf.size(x))
9

```

输出：

```

1 shape: TensorShape([]) rank: 0 size: 1
2 shape: TensorShape([5]) rank: 1 size: 5
3 shape: TensorShape([2, 2]) rank: 2 size: 4

```

- shape：返回一个 TensorShape 类型的张量
- rank：是维度，数值是0维
- size：张量中的元素的数量

有很多操作会影响形状

- tf.reshape：

```

1 import tensorflow as tf
2
3 x = tf.range(16)
4 print(x.shape) # (16, )
5 x = tf.reshape(x, (4, 4))
6 print(x.shape) # (4, 4)
7 x = tf.reshape(x, (2, 8))
8 print(x.shape) # (2, 8)
9 x = tf.reshape(x, (2, 2, 4))
10 print(x.shape) # (2, 2, 4)

```

- 维度扩张

```

1 import tensorflow as tf
2
3 x = tf.reshape(tf.range(16), (4, 4))
4 print(x.shape) # ==> (4, 4)
5 y = tf.expand_dims(x, 0)
6 print(y.shape) # ==> (1, 4, 4)
7 y = tf.expand_dims(x, 1)
8 print(y.shape) # ==> (4, 1, 4)

```

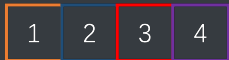

- 维度压榨

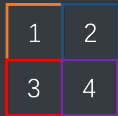
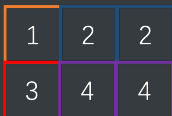
```

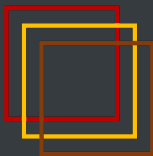
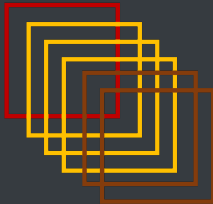
1 import tensorflow as tf
2
3 x = tf.reshape(tf.range(16), (4, 4, 1, 1))
4 print(x.shape) # ==> (4, 4, 1, 1)
5 y = tf.squeeze(x)
6 print(y.shape)

```

- `tf.repeat` : 重复张量沿着某个轴 (axis)

`tf.repeat(`  `, [1, 2, 2, 3])` \Rightarrow 

`tf.repeat(`  `, [1, 2], axis=1)` \Rightarrow 

`tf.repeat(`  `, [1, 3, 2], axis=1)` \Rightarrow 

```

1 import tensorflow as tf
2
3 x = tf.reshape(tf.range(16), (4, 4))
4 print(x)
5 y = tf.repeat(x, repeats=[2, 2, 1, 3], axis=0)
6 print(y)

```

输出: 【解释】 `axis=0` 从上往下开始复制, 2次、2次、1次、3次, 所以此时 `y.shape == (2+2+1+3, x.shape[1])`, 即 `y.shape=(8, 4)`

```

1 tf.Tensor(
2 [[ 0  1  2  3]
3  [ 4  5  6  7]
4  [ 8  9 10 11]
5  [12 13 14 15]], shape=(4, 4), dtype=int32)
6 tf.Tensor(
7 [[ 0  1  2  3]
8  [ 0  1  2  3]
9  [ 4  5  6  7]
10 [ 4  5  6  7]
11 [ 8  9 10 11]
12 [12 13 14 15]
13 [12 13 14 15]
14 [12 13 14 15]], shape=(8, 4), dtype=int32)

```

- 张量连接 `tf.concat`

Concat (

1	2	3	4
---	---	---	---

 ,

1	2	3	4	5
---	---	---	---	---

) \Rightarrow

1	2	3	4	1	2	3	4	5
---	---	---	---	---	---	---	---	---

Concat (

1	2
3	4

 ,

1	2
3	4
5	6

 , axis=0) \Rightarrow

1	2
3	4
1	2
3	4
5	6

```
1 import tensorflow as tf
2 x = tf.random.normal([2, 2])
3 y = tf.random.normal([4, 2])
4
5 z = tf.concat([x, y], axis=0)
6 print(z.shape) # ==> (6, 2) 沿着第一个轴（从上而下）进行拼接
```

【注意】 tf.concat 是不改变张量的 rank

■ 张量堆叠 tf.stack

```
1 import tensorflow as tf
2 x = tf.random.normal([3, 3])
3 y = tf.random.normal([3, 3])
4
5 z = tf.stack([x, y], axis=0)
6 print(z.shape) # ==> (2, 3, 3)
7
8 z = tf.stack([x, y], axis=1)
9 print(z.shape) # ==> (3, 2, 3)
```

■ tf.tile

```
1 import tensorflow as tf
2
3 x = tf.constant([[1, 2, 3], [4, 5, 6]])
4
5 y = tf.tile(x, tf.constant([2, 3]))
6
7 print(y)
```

输出:

```
1 tf.Tensor(
2 [[1 2 3 1 2 3 1 2 3]
3  [4 5 6 4 5 6 4 5 6]
4  [1 2 3 1 2 3 1 2 3]
5  [4 5 6 4 5 6 4 5 6]], shape=(4, 9), dtype=int32)
```

■ tf.pad

1.4 Tensor索引

TensorFlow中的Tensor索引和Numpy索引几乎一致。

举例：

```
1  import tensorflow as tf
2
3  x = tf.range(30)
4  x = tf.reshape(x, (5, 6))
5  print(x)
6
7  # 取某个值
8  print(x[0][0])
9  print(x[0, 0])
10
11 # 取某一行
12 print(x[0])
13 print(x[0, :])
14
15 # 取某一系列
16 print(x[:, 0])
17
18 # 取连续的几行（列同理）
19 print(x[2:4, :]) # 2, 3两行
20 print(x[1:, :]) # 取1, 2, 3, 4四行
21 print(x[:3, :]) # 取0, 1, 2三行
22
23 # 取不连续的行（列同理）
24 print(x[1::2, :]) # 取1, 3两行
25 print(x[::-1, :]) # 倒过来取行 4, 3, 2, 1, 0
26 print(x[4:1:-1, :]) # 取4, 3, 2三行
27
28 # 取子矩阵
29 print(x[1:3, 2:4]) # 取 1, 2两行, 2, 3两列, 一个2x2的子矩阵
```

输出：

```
1  tf.Tensor(
2  [[ 0  1  2  3  4  5]
3   [ 6  7  8  9 10 11]
4   [12 13 14 15 16 17]
5   [18 19 20 21 22 23]
6   [24 25 26 27 28 29]], shape=(5, 6), dtype=int32)
7  tf.Tensor(0, shape=(), dtype=int32)
8  tf.Tensor(0, shape=(), dtype=int32)
9  tf.Tensor([0 1 2 3 4 5], shape=(6,), dtype=int32)
10 tf.Tensor([0 1 2 3 4 5], shape=(6,), dtype=int32)
11 tf.Tensor([ 0  6 12 18 24], shape=(5,), dtype=int32)
12 tf.Tensor(
13 [[12 13 14 15 16 17]
```



```

7  tf.Tensor(
8  [[ 6  7  8  9 10 11]
9   [18 19 20 21 22 23]
10  [24 25 26 27 28 29]], shape=(3, 6), dtype=int32)
11  tf.Tensor(
12  [[ 0  5]
13   [ 6 11]
14   [12 17]
15   [18 23]
16   [24 29]], shape=(5, 2), dtype=int32)

```

▪ tf.gather_nd

```

1  import tensorflow as tf
2
3  x = tf.range(30)
4  x = tf.reshape(x, (5, 6))
5  print(x)
6
7
8  y = tf.gather_nd(x, indices=[[0, 1], [2, 3]])
9  print(y)

```

输出:

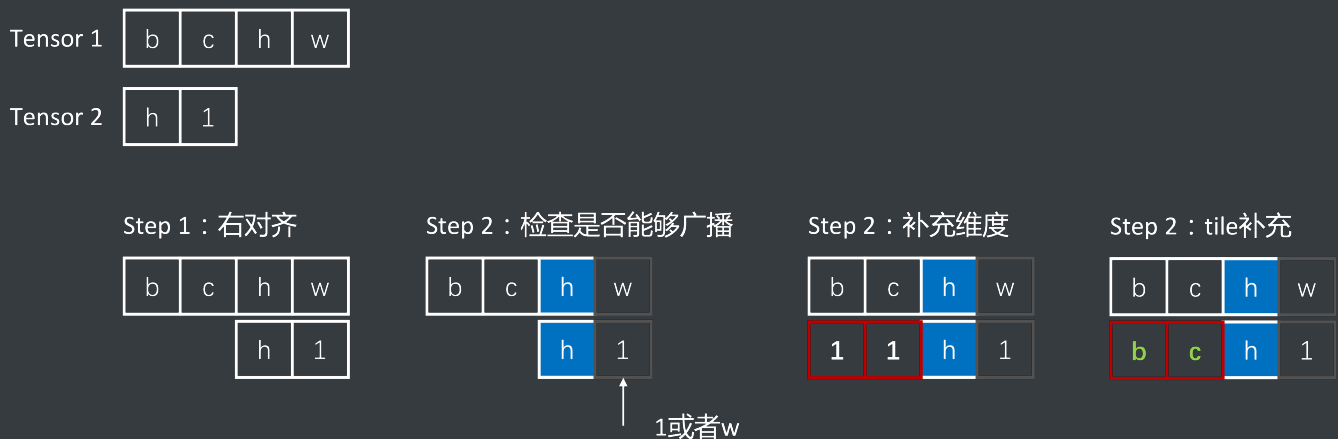
```

1  tf.Tensor(
2  [[ 0  1  2  3  4  5]
3   [ 6  7  8  9 10 11]
4   [12 13 14 15 16 17]
5   [18 19 20 21 22 23]
6   [24 25 26 27 28 29]], shape=(5, 6), dtype=int32)
7  tf.Tensor([ 1 15], shape=(2,), dtype=int32)

```

1.5 Tensor广播机制

当两个**维度不同**的张量进行运算时候，会发生广播（自动扩展），其原理非常简单，如下图所示：



```

1  import tensorflow as tf
2

```

```

3 x = tf.random.normal([10, 16, 64, 64])
4
5 y = 1.0 # shape=()
6
7 print((x + y).shape)
8
9 y = tf.random.normal([64, 1])
10 print((x + y).shape)
11
12 y = tf.random.normal([64, 64])
13 print((x + y).shape)
14
15 # tensorflow.python.framework.errors_impl.InvalidArgumentError: Incompatible shapes: [10,16,64,64] vs.
    [64,2] [Op:AddV2]
16 # y = tf.random.normal([64, 2]) # 报错
17 # print((x + y).shape)

```

1.6 不规则张量和稀疏张量

不规则张量在Tensorflow的ragged包中进行了支持，这种数据结构是Tensorflow独有的，当然这样的数据结构自然不会支持矩阵等操作，点操作（Elementwise-Op）都是支持的。

Ragged Tensor可以很好的在文本数据表达上使用，我们取10个句子，和图像不同，每个句子可能长度是不同的，虽然可以通过pad操作进行填充对齐，但是LSTM这种RNN的循环神经网络那些0是不优雅的，或者说某些情况下是错误的。

不规则张量使用场景毕竟还是不多的，这里不进行过分介绍和分析，下面简单的例子：

```

1 import tensorflow as tf
2
3 x = tf.ragged.constant([[1, 3, 2], [3, 1], [4, 6, 4, 4]])
4 print(x)
5
6 print(x + 10)

```

输出：

```

1 <tf.RaggedTensor [[1, 3, 2], [3, 1], [4, 6, 4, 4]]>
2 <tf.RaggedTensor [[11, 13, 12], [13, 11], [14, 16, 14, 14]]>

```

稀疏张量（Sparse Tensor）是一种特殊的张量，其中在该张量中有很多0，这个“很多”比较难定义，如下的例子：

```

1 from tensorflow import SparseTensor
2
3 x = SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4])
4 print(x)

```

输出：

```

1 SparseTensor(indices=tf.Tensor(
2   [[0 0]
3    [1 2]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2], shape=(2,), dtype=int32),
4   dense_shape=tf.Tensor([3 4], shape=(2,), dtype=int64))

```

1.7 Tensor的实现细节

在Tensorflow源码中，Tensor 是一个类，其成员变量有两个：

- shape：该张量的形状
- buffer：张量的实际数据存储

```

1 class Tensor {
2     // ...
3 private:
4     TensorShape shape_;
5     TensorBuffer* buf_;
6 };

```

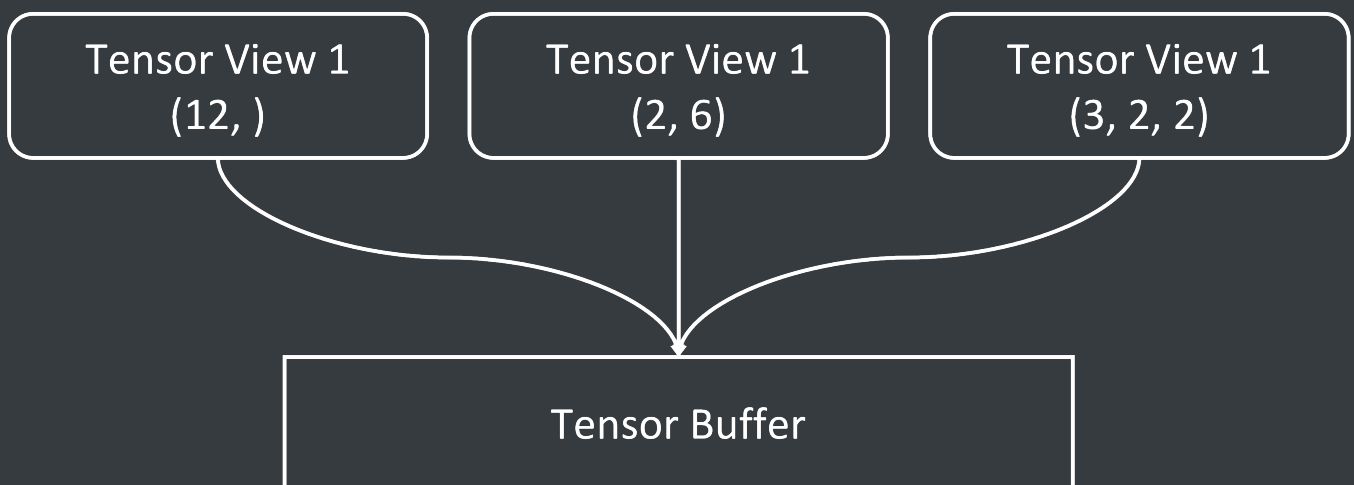
在源代码中，tensor.h 和 tensor.cc 不像是人写的，应该是Tensorflow官方通过 tensor.proto 的ProtBuff生成的，当然不是全部生成的。以上两个类型 TensorShape 和 TensorBuffer 不必了解其详细内容，我们可以当作以下结构就好：

```

1 class Tensor {
2     // ...
3 private:
4     std::vector<size_t> shape_;
5     void * buffer;
6 };

```

简单说就是一个数组表示形状，一个数据指针真正的取存取数据。由于只有一个数据指针，那么如果对于一个矩阵来说，我们需要取得第 i 行和第 j 列的话，需要映射成为数据指针的偏移量，然而当取更多的数据的时候，需要不止一个偏移量了，此时诞生了一个新的概念——视图（View），这也是非常普遍的一种设计思路。



TensorBuffer 是 RefCounted 的子类，其成员为：

```
1 class TensorBuffer : public core::RefCounted {
2 private:
3     void* const data_;
4 };
```

此外，真正的 TensorShape 也不仅仅是保存了张量的形状，我们发现张量少了一个非常重要的东西——**数据类型 (DataType)**，数据类型施救保存在 TensorShape 中的。TensorShape->TensorShapeBase->TensorShapeRep：

```
1 union {
2     uint8 buf[16];
3     // Force data to be aligned enough for a pointer.
4     Rep64* unused_aligner; # 为了对齐的，啥也没用
5 } u_;
6 int64_t num_elements_;
```

这里有16个字节。

最后，还有一个绕不开角色，就是这个Tensor是TF实现的吗？显然不是，它是由著名的第三方库 Eigen3 实现的，而TF中的 Tensorflow只是一种高级封装。

详细地实现过程请参见TF C++源代码。

2 数据集

数据是神经网络的生产资料，算力是神经网络的生产力，**数据集 (Dataset)** 是神经网络中的非常重要，也极大地影响了神经网络最后表现的性能。在机器学习领域，有一个著名的定律：

GIGO: Garbage in Garbage out (Wiki)

意思是当你的数据是脏的，那么是不可能得到好的模型的。

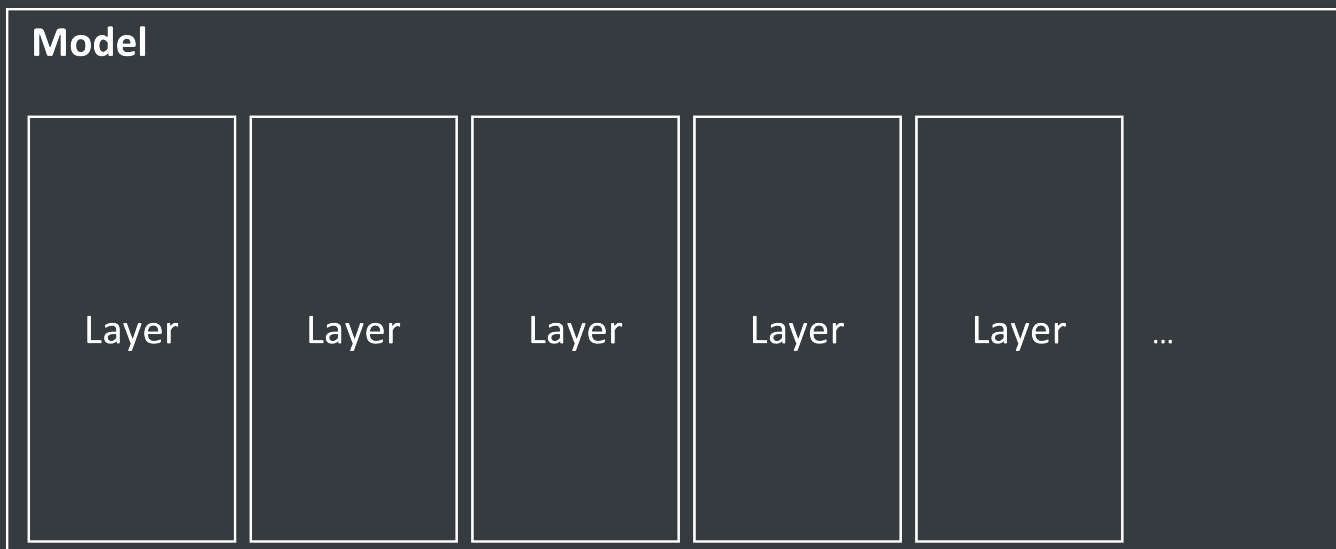
TF 2对数据集的构造和TF 1属于继承发展关系，但趋势是向着高级API进行发展。TF 2中与数据集相关的API在 `tf.data.xxx` 下，此外，TF官方还提供了另一种数据集hub，叫做 TensorFlow Datasets，可以通过此API框架对著名的数据集进行获取和处理。

Tip: TF datasets的数据下载之类的是在国内极其不友好，所以一般有些真正的项目中不使用这个框架，一般被用来测试，入门等。

TF datasets数据集列表：https://www.tensorflow.org/datasets/catalog/overview?hl=zh-cn#all_datasets

3 深度神经网络

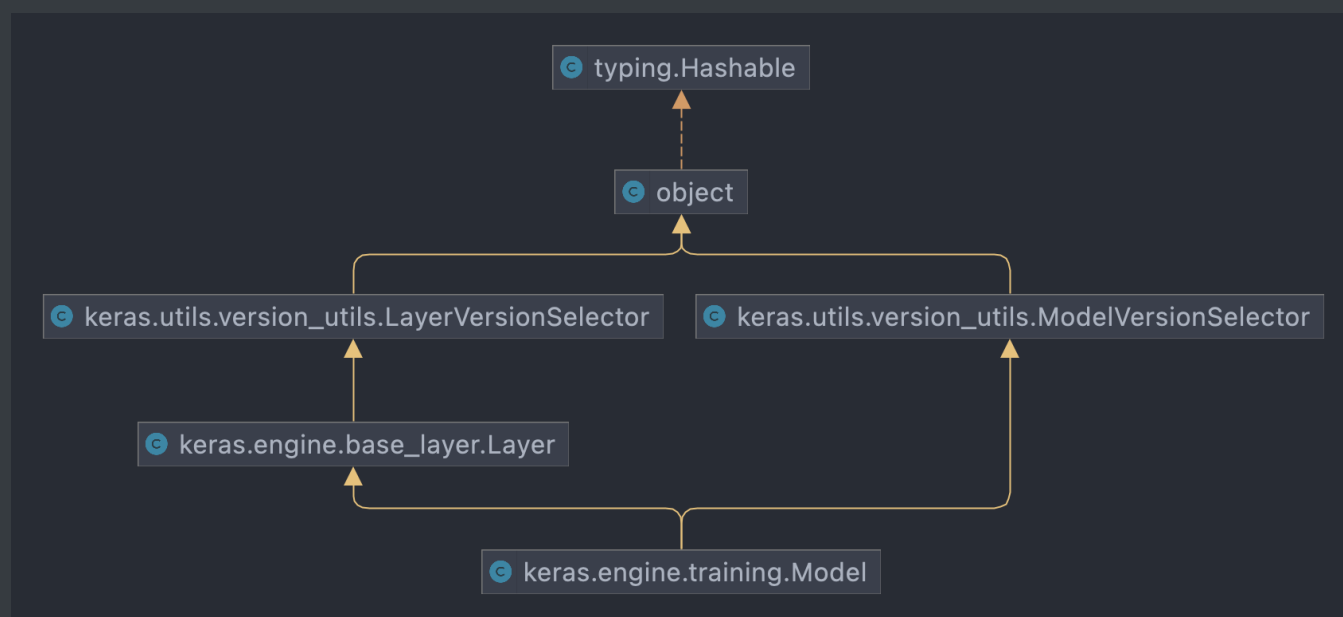
深度神经网络是机器学习的一种算法，它用于非结构化数据的预测、识别等非常有效。



神经网络就是我们经常所说的**模型（Model）**，在Tensorflow 2中，一个神经网络通过 **Model** 来抽象的。

在介绍神经网络API之前，要说明TF 2的神经网络API使用的是Keras，而在Tensorflow 2.x早期版本中，通常使用的是 `tf.keras.xxx`，实际上这些API和Keras的API `keras.xxx` 是相同的，自然，Keras早就被Google收购了。在Tensorflow 2.9中，可以直接使用 `keras.xxx` 构建。

3.1 Model 类



Model 的使用方式有两种：

- 通过继承的方式定义模型

```
1 import keras
2 import tensorflow as tf
3 from keras.layers import Dense
4
5
6 class Net(keras.Model):
7     def __init__(self):
8         super(Net, self).__init__()
```

```

9         self.hidden_layer = Dense(256, activation='relu')
10        self.output_layer = Dense(10, activation='softmax')
11
12        def call(self, inputs, training=None, mask=None):
13            return self.output_layer(self.hidden_layer(inputs))
14
15    model = Net()
16
17    x = tf.random.normal([1, 784])
18    y = model(x)
19    print(y.shape)

```

其中 Dense 表示全连接层。

- 通过函数式API创建一个 Model(inputs, outputs, name) , 需要提供输入层和输出层即可。

```

1  import tensorflow as tf
2  import keras
3  from keras.layers import Dense
4
5  input_layer = keras.Input(shape=[784,])
6  hidden = Dense(256, activation='relu')(input_layer)
7  output = Dense(10, activation='softmax')(hidden)
8
9  model = keras.Model(inputs=[input_layer, ], outputs=[output, ], name='test_model')
10 model.summary() # Print model structure
11
12 x = tf.random.normal([1, 784])
13 y = model(x)
14 print(y.shape)

```

输出:

```

1  Model: "test_model"
2  -----
3  Layer (type)                Output Shape          Param #
4  -----
5  input_1 (InputLayer)        [(None, 784)]         0
6
7  dense (Dense)                (None, 256)           200960
8
9  dense_1 (Dense)              (None, 10)            2570
10
11  -----
12  Total params: 203,530
13  Trainable params: 203,530
14  Non-trainable params: 0
15  -----
16  (1, 10)

```

以上是两种最简单的两种使用方式，接下来详细地对 Model 类进行剖析。

第一：创建模型需要哪些参数：

```

1  def __init__(self, *args, **kwargs):
2      # ...
3      # Filter the kwargs for multiple inheritance.
4      supported_kwargs = ['inputs', 'outputs', 'name', 'trainable', 'skip_init']
5      model_kwargs = {k: kwargs[k] for k in kwargs if k in supported_kwargs}
6      other_kwargs = {k: kwargs[k] for k in kwargs if k not in supported_kwargs}
7      inject_functional_model_class(self.__class__)
8      functional.Functional.__init__(self, *args, **model_kwargs)
9      # ...
10

```

首先，创建 Model 可以传入各种各样的参数，然后在 `__init__` 方法中会将这些参数进行分类为 `model_kwargs` 和 `other_kwargs`。`model_args` 支持

- `inputs`：模型的输入。当模型是多输入的时候，可以使用 list 进行传参
- `outputs`：模型的输出。当模型是多输出的时候，可以使用 list 进行传参
- `name`：模型的名称
- `trainable`：是否是可训练的，比如 Dropout 在训练和测试过程是不同状态的
- `skip_init`：官方的解释如下：

This is used by the `Model` class, since we have same logic to swap the class in the `__new__` method, which will lead to `__init__` get invoked twice. Use the `skip_init` to skip one of the invocation of `__init__` to avoid any side-effects.

核心意思是 `__init__` 和 `__new__` 是相同逻辑的，防止执行两次。

了解初始化参数之后，`Model` 的属性有：

- `Model` 本身的一些属性，就是在定义，创建模型的时候的属性，包括输入、输出信息等
- `Model` 训练的一些属性，比如优化器、损失函数、训练策略等

