TensorFlow 2.0 简明使用手册

作者：徐亦捷

前言

发现龙龙老师的《TensorFlow2.0 深度学习书》是一本很适合入门TensorFlow的书，我也因此书受益匪浅。然而，我更希望有一本在想不起来的时候可以随时翻阅的手册。同时，我有一定的ML基础，希望从此书中学习更多。因此，我将该书分为基础常用命令与各种网络几个部分，每个命令均有示例，便于更好理解。

目录

- 举例
- `tf.fill(shape, value)`
  - 举例
- `tf.random.normal(shape, mean, stddev)`
  - 举例
- `tf.random.uniform(shape, minval, maxval, dtype)`
  - 举例
- `tf.range(start, limit, delta)`
  - 举例
- `tf.reshape(x,list)`
  - 举例
- `x.ndim`
  - 举例
- `tf.expand_dims(x,axis)`
  - 举例
- `tf.squeeze(x, axis)`
  - 举例
- `tf.transpose(x, perm=list)`
  - 举例
- `tf.tile(x, multiples)`
  - 举例
- `tf.broadcast_to(x, new_shape)`
  - 举例
- `tf.concat(tensor, axis)`
  - 举例
- `tf.stack(tensor, axis)`
- `tf.gatger_nd(x, list)`
  - 举例
- `tf.boolean_mask(x, mask, axis)`
  - 举例
- `tf.where(cond, a, b)`
  - 举例
  - 注意
- `tf.scatter_nd(indices, updates, shape)`
  - 举例
  - 注意
- tf.meshgrid(x, y)`
  - 举例
- `@`
  - 举例
  - 注意
- `tf.one_hot(x, depth)`
  - 举例

数学运算

- 加减乘除
  - 举例
- 乘方
  - 举例
  - 注意
- 指数、对数
  - 举例
  - 注意

基础常用命令

`tf.constant()`

创建**常**张量，可以是数字、数组、字符串或布尔类型。

举例

```
a = tf.constant(1)
b = tf.constant([4,32,32,3])
c = tf.constant([[1,2,3],[4,5,6],[7,8,9]])
d = tf.constant('Hello World')
e = tf.constant(True)
print(a)
print(b)
print(c)
print(d)
print(e)
```

结果

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor([ 4 32 32  3], shape=(4,), dtype=int32)
tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
tf.Tensor(b'Hello World', shape=(), dtype=string)
tf.Tensor(True, shape=(), dtype=bool)
```

注意

不能和普通`python`变量混用。

`tf.is_tensor(x)`

判断是否为张量，结果`True`或`False`。

举例

```
a = tf.constant(1)
b = tf.constant([1,2,3])
c = tf.Variable(1)
d = tf.constant(True)
e = 1
print(tf.is_tensor(a))
print(tf.is_tensor(b))
print(tf.is_tensor(c))
print(tf.is_tensor(d))
print(tf.is_tensor(e))
```

结果

```
True
True
True
True
False
```

`x.numpy()`

将`x`转换为`numpy`的`ndarray`。

举例

```
a = tf.constant([1,2,3])
b = a.numpy()
print(a)
print(b)
```

结果

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

tf.strings

`tf.strings.lower()` 将字符全部转换为小写

`tf.strings.join()` 拼接字符串

`tf.strings.length()` 获取字符串长度

`tf.strings.split()`切分字符串

举例

```
a = tf.constant('Hello World')
b = tf.constant('I Love You')
print(tf.strings.lower(a))
print(tf.strings.length(a))
print(tf.strings.split(a,'hi'))
```

结果

```
tf.Tensor(b'hello world', shape=(), dtype=string)
tf.Tensor(11, shape=(), dtype=int32)
tf.Tensor([b'Hello World'], shape=(1,), dtype=string)
```

dtype=tf.intXX

将数值型张量保存为不同的精度。

支持 `tf.int16, tf.int32, tf.int64, tf.float16, tf.float32, tf.float64`。其中，`tf.float64` 即为 `tf.double`。

举例

```
print(tf.constant(123456789, dtype=tf.int16))
print(tf.constant(123456789, dtype=tf.int32))
```

结果

```
tf.Tensor(-13035, shape=(), dtype=int16)
tf.Tensor(123456789, shape=(), dtype=int32)
```

注意

使用恰当的精度有助于正确保存张量且节省内存空间。

x.dtype

返回张量`x`的保存精度。

举例

```
a = tf.constant(123456789, dtype=tf.int16)
b = tf.constant(123456789, dtype=tf.int32)
print(a.dtype)
print(b.dtype)
```

结果

```
<dtype: 'int16'>
<dtype: 'int32'>
```

注意

`x.dtype`后没有`()`。

`tf.cast(x, dtype)`

转换张量的精度。

举例

```
import numpy as np
a = tf.constant(np.pi, dtype=tf.float16)
print(tf.cast(a, tf.double))
a = tf.constant(123456789, dtype=tf.int32)
print(tf.cast(a, tf.int16))
a = tf.constant([True, False])
print(tf.cast(a, tf.int32))
a = tf.constant([-1, 0, 1, 2])
print(tf.cast(a, tf.bool))
```

结果

```
tf.Tensor(3.140625, shape=(), dtype=float64)
tf.Tensor(-13035, shape=(), dtype=int16)
tf.Tensor([1 0], shape=(2,), dtype=int32)
tf.Tensor([ True False  True  True], shape=(4,), dtype=bool)
```

注意

当高精度->低精度时，有可能数据溢出。

`tf.Variable()`

用于记录梯度信息，但消耗大量内存，须正确使用。

举例

```
a = tf.Variable([1,2,3,4])
print(a)
print(a.name)
print(a.trainable)
```

结果

```
<tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4])>
Variable:0
True
```

注意

我们一般不需要关注`name`属性，它是`TensorFlow`内部维护的。

`trainable`表示当前张量是否需要被优化，可以设置为`False`。

`tf.convert_to_tensor()`

将`Numpy Array`和`Python List`转换为张量。

举例

```
import numpy as np
a = [1,2,3]
b = np.array([1,2,3,4])
print(a)
print(b)
print(tf.convert_to_tensor(a))
print(tf.convert_to_tensor(b))
```

结果

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

注意

其可被`tf.constant()`取代。

`tf.zeros()`

创建全0张量。

举例

```
a = tf.zeros([1,2])
print(a)
```

结果

```
tf.Tensor([[0. 0.]], shape=(1, 2), dtype=float32)
```

tf.zeros_like()

创建与某个张量（或`Python List`、`Numpy Array`）shape相同的全0张量

举例

```
import numpy as np
a = tf.constant([1,2,3])
b = [1,2,3]
c = np.array([1,2,3])
a0 = tf.zeros_like(a)
b0 = tf.zeros_like(b)
c0 = tf.zeros_like(c)
print(a0)
print(b0)
print(c0)
```

结果

```
tf.Tensor([0 0 0], shape=(3,), dtype=int32)
tf.Tensor([0 0 0], shape=(3,), dtype=int32)
tf.Tensor([0 0 0], shape=(3,), dtype=int32)
```

tf.ones()

创建全1张量。

举例

```
a = tf.ones([1,2])
print(a)
```

结果

```
tf.Tensor([[1. 1.]], shape=(1, 2), dtype=float32)
```

tf.ones_like()

举例

```
import numpy as np
a = tf.constant([1,2,3])
b = [1,2,3]
c = np.array([1,2,3])
a0 = tf.zeros_like(a)
b0 = tf.zeros_like(b)
c0 = tf.zeros_like(c)
print(a0)
print(b0)
print(c0)
```

结果

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

`tf.fill(shape, value)`

以创建全为自定义数值 value 的张量。

举例

```
a = tf.fill([2,2], 100)
print(a)
```

结果

```
tf.Tensor(
[[100 100]
 [100 100]], shape=(2, 2), dtype=int32)
```

`tf.random.normal(shape, mean, stddev)`

创建形状为 shape，均值为 mean，标准差为 stddev 的正态分布$\mathcal{N}(mean, stddev^2)$。

默认地，`mean=0, stddev=1`。

举例

```
a = tf.random.normal([2,2], mean=1,stddev=2)
print(a)
```

结果

```
tf.Tensor(
[[-1.6261725  0.4671601]
 [ 4.5686626  7.1735497]], shape=(2, 2), dtype=float32)
```

`tf.random.uniform(shape, minval, maxval, dtype)`

创建采样自$[\text{minival}, \text{maxval})$区间的均匀分布的张量。

默认地，`minval=0, maxval=None, dtype=tf.float32`。

举例

```
a = tf.random.uniform([2,2], minval=0, maxval=10, dtype=tf.float64)
b = tf.random.uniform([2,2],maxval=100,dtype=tf.int32) #整形
print(a)
print(b)
```

结果

```
tf.Tensor(
[[5.32424271 8.28679821]
 [7.00816235 5.9412553 ]], shape=(2, 2), dtype=float64)
tf.Tensor(
[[74 23]
 [71  9]], shape=(2, 2), dtype=int32)
```

`tf.range(start, limit, delta)`

创建$[\text{start}, \text{limit})$，步长为delta的序列

举例

```
a = tf.range(1, 10, range=2)
print(a)
```

`tf.reshape(x,list)`

将张量`x`转换为shape为list的张量

举例

```
x = tf.range(96)
x = tf.reshape(x, [2,4,4,3])
print(x)
```

结果

```
tf.Tensor(
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]
   [ 9 10 11]]
```

```
 [[12 13 14]
  [15 16 17]
  [18 19 20]
  [21 22 23]]

 [[24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]

 [[36 37 38]
  [39 40 41]
  [42 43 44]
  [45 46 47]]]


[[[48 49 50]
  [51 52 53]
  [54 55 56]
  [57 58 59]]

 [[60 61 62]
  [63 64 65]
  [66 67 68]
  [69 70 71]]

 [[72 73 74]
  [75 76 77]
  [78 79 80]
  [81 82 83]]

 [[84 85 86]
  [87 88 89]
  [90 91 92]
  [93 94 95]]]], shape=(2, 4, 4, 3), dtype=int32)
```

`x.ndim`

返回张量x的维度值。

举例

```
x = tf.range(96)
x = tf.reshape(x, [2,4,4,3])
print(x.ndim)
```

结果

```
4
```

tf.expand_dims(x,axis)

对于变量x，当$\text{axis}>0$，在指定的 axis 轴前可以插入一个新的维度；当$\text{axis} < 0$，在指定的axis轴后插入一个新的维度；当$\text{axis} = 0$,即在原来第一个axis轴的地方新建一个维度。

举例

```
x = tf.random.uniform([28,28],maxval=10,dtype=tf.int32)
x = tf.expand_dims(x,axis=2)
print(x.shape)
x = tf.expand_dims(x, axis=-1)
print(x.shape)
x = tf.expand_dims(x, axis=0)
print(x.shape)
```

结果

```
(28, 28, 1)
(28, 28, 1, 1)
(1, 28, 28, 1, 1)
```

tf.squeeze(x, axis)

删除某个axis轴的长度为1的维度。

举例

```
x = tf.random.uniform([28,28,1],maxval=10,dtype=tf.int32)
x = tf.squeeze(x, axis=2)
print(x.shape)
```

结果

```
(28, 28)
```

如果不指定维度参数 axis，即 `tf.squeeze(x)`，那么他会默认删除所有长度为 1 的维度。

tf.transpose(x, perm=list)

将张量x的维度顺序转换为list所规定的顺序。

举例

```
x = tf.random.normal([2,32,32,3])
x = tf.transpose(x,perm=[0,3,1,2])
print(x.shape)
```

返回

```
(2, 3, 32, 32)
```

tf.tile(x, multiples)

使张量x在指定multiplies复制。

举例

```
a = tf.random.normal([4,3,3,2])
b = tf.tile(a, multiples=[2,3,3,1])
print(b.shape)
```

结果

```
(8, 9, 9, 2)
```

tf.broadcast_to(x, new_shape)

`tf.tile`消耗大量内存。一般地，我们可以使用广播 (Broadcasting)机制，操作与`tf.tile`类似。

举例

```
a = tf.constant([1,2,3])
a = tf.broadcast_to(a, [2,3])
print(a)
```

结果

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

tf.concat(tensor, axis)

将两个张量合并，不改变维度数。

举例

```
a = tf.random.normal([2,3,4])
b = tf.random.normal([3,3,4])
c = tf.concat([a,b], axis=0)
print(c.shape)
```

结果

```
(5, 3, 4)
```

tf.stack(tensor, axis)

直接在现有维度上面合并数据，改变维度数。

tf.gatger_nd(x, list)

对张量x按list同时采样多个点。

举例

```
a = tf.random.normal([2,2])
b = tf.gather_nd(a, [[0,0],[1,1]])
print(a)
print(b)
```

结果

```
tf.Tensor(
[[-0.21231703  0.3082871 ]
 [-0.20058835 -0.53994805]], shape=(2, 2), dtype=float32)
tf.Tensor([-0.21231703 -0.53994805], shape=(2,), dtype=float32)
```

tf.boolean_mask(x, mask, axis)

对张量x按掩码mask（在axis轴）进行采样（不常用）。

举例

```
a = tf.random.normal([2,2])
b = tf.boolean_mask(a, mask=[True, False], axis=0)
c = tf.boolean_mask(a, mask=[[True,False],[False, True]])
print(a)
print(b)
print(c)
```

结果

```
tf.Tensor(
[[-0.42816168  1.5068071 ]
 [-0.58368284  0.13115934]], shape=(2, 2), dtype=float32)
tf.Tensor([[-0.42816168  1.5068071 ]], shape=(1, 2), dtype=float32)
tf.Tensor([-0.42816168  0.13115934], shape=(2,), dtype=float32)
```

tf.where(cond, a, b)

根据cond的真假返回a或b中相应的值，即

$$\text{返回值} = \begin{cases} a_i, \text{cond}_i = \text{True} \\ b_i, \text{cond}_i = \text{False} \end{cases}$$

举例

```
a = tf.ones([2,2])
b = tf.zeros([2,2])
cond = tf.constant([[True, True],[False, False]])
c = tf.where(cond, a, b)
d = tf.where(a>0)
print(c)
print(d)
```

结果

```
tf.Tensor(
[[1. 1.]
 [0. 0.]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[0 0]
 [0 1]
 [1 0]
 [1 1]], shape=(4, 2), dtype=int64)
```

注意

若没有a和b参数，则返回条件为真的相应值的索引。

`tf.scatter_nd(indices, updates, shape)`

将updates按indices的索引顺序更新到shape大小的全0张量中。

举例

```
indices = tf.constant([[4],[3],[1],[7]])
updates = tf.constant([4.4, 3.3, 1.1, 7.7])
a = tf.scatter_nd(indices, updates, [8])
print(a)
```

结果

```
tf.Tensor([0.  1.1 0.  3.3 4.4 0.  0.  7.7], shape=(8,), dtype=float32)
```

注意

用于将卷积矩阵转换为稀疏矩阵时尤其有用。

tf.meshgrid(x, y)`

生成二维网格点坐标，时间复杂度比两个for嵌套要低。

举例

```
x = tf.linspace(-8.,8,100)
y = tf.linspace(-8.,8,100)
x,y = tf.meshgrid(x,y)
print(x.shape, y.shape)
```

结果

```
(100, 100) (100, 100)
```

@

用于矩阵相乘，可以是`Numpy Ndarray`、`Numpy Matrix`或`tf.constant`。

举例

```
a = np.array([[1,2],[3,4]])
b = np.array([[2,2],[3,4]])
c = np.matrix([[1,2],[3,4]])
d = np.matrix([[2,2],[3,4]])
e = tf.constant([[1,2],[3,4]])
f = tf.constant([[2,2],[3,4]])
g = a@b
h = c@d
i = e@f
print(g)
print(h)
print(i)
```

结果

```
[[ 8 10]
 [18 22]]
[[ 8 10]
 [18 22]]
tf.Tensor(
[[ 8 10]
 [18 22]], shape=(2, 2), dtype=int32)
```

注意

`@`支持Broadcast机制。

tf.one_hot(x, depth)

将张量或矩阵转变为`one-hot`编码。

举例

```
a = tf.constant([1,2,3,4,5])
b = tf.one_hot(a, depth=10)
print(b)
```

结果

```
tf.Tensor(
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]], shape=(5, 10), dtype=float32)
```

数学运算

加减乘除

加：`tf.add()`

减：`tf.substract()`

乘：`tf.multiply()`

除：`tf.divide()`

举例

```
a = tf.constant([2,4,6])
b = tf.constant([1,2,3])
plus = tf.add(a, b)
minus = tf.subtract(a, b)
times = tf.multiply(a, b)
divide = tf.divide(a, b)
print(plus)
print(minus)
print(times)
print(divide)
```

结果

```
tf.Tensor([3 6 9], shape=(3,), dtype=int32)
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([ 2  8 18], shape=(3,), dtype=int32)
tf.Tensor([2. 2. 2.], shape=(3,), dtype=float64)
```

乘方

使用`tf.pow(x,a)`或`x**a`，返回x^a。

举例

```
a = tf.constant([1,2,3])
b = tf.pow(a, 3)
c = a**3
print(b)
print(c)
```

返回

```
tf.Tensor([ 1  8 27], shape=(3,), dtype=int32)
tf.Tensor([ 1  8 27], shape=(3,), dtype=int32)
```

注意

对于常见的平方和平方根运算，可以使用`tf.square(x)`和`tf.sqrt(x)`实现。

指数、对数

使用`tf.pow(a, x)`或`a**x`，返回a^x。

举例

```
a = tf.constant([1,2,3])
b = tf.pow(3, a)
c = 3**a
print(b)
print(c)
```

返回

```
tf.Tensor([ 3  9 27], shape=(3,), dtype=int32)
tf.Tensor([ 3  9 27], shape=(3,), dtype=int32)
```

注意

对于e^x，可以使用`tf.exp(x)`。

对于$\ln x$，可以使用`tf.math.log(x)`。

对于其他底数的对数，可以使用换底公式。

索引与切片

索引

以访问第1个子张量的第1个子张量为例，可以使用`a[0][0]`或`a[0,0]`。

切片

通过`start:end:step`切片可以方便地提取$[\text{start}, \text{right})$，步长为step的数据。

有时，我们可以有选择地省略。

全部省略：`::`，表示最开始到最末尾，步长为1。`::`可以简化为`:`。

step可以为负数，即从后往前进行读取。

我们有以下常用格式。

切片方式	意义
`start:end:step`	$[\text{start}, \text{right})$，步长为step
`start:end`	$[\text{start}, \text{right})$，步长为1
`start:`	从 start 开始读取完后续所有元素，步长为 1
`start::step`	从 start 开始读取完后续所有元素，步长为 step
`:end:step`	$[0, \text{right})$，步长为step
`:end`	$[0, \text{right})$，步长为1
`::step`	以step为步长采样元素
`::`	读取所有元素
`:`	读取所有元素

当维度数量较大时，为了防止出现太多冒号，我们可以继续进行省略，用`...`表示期间的所有维度。

我们有以下常用格式。

切片方式	意义
`a,...b`	a 维度对齐到最左边，b 维度对齐到最右边，读取中间的全部维度
`a,...`	a 维度对齐到最左边，读取a 维度后的全部维度，a 维度按 a 方式 读取。这种情况等同于 a 索引/切片方式
`..., b`	b 维度对齐到最右边，读取b 之前的全部维度，b 维度按 b 方式 读取
`...`	读取张量所有数据

加载数据集

前期准备

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets
```

MNIST

```
(x, y), (x_test, y_test) = datasets.mnistload_data()
```

`tf.data.Dataset.from_tensor_slices((x,y))`

将数据转换成Dataset对象。

举例

```
(x, y), (x_test, y_test) = datasets.mnistload_data()
train_db = tf.data.Dataset.from_tensor_slices((x, y))
```

`x.shuffle(buffer_size)`

将Dataset x按buffer_size打散。一般，buffer_size一般是一个较大的整数，如10000。

举例

```
train_db = train_db.shuffle(10000)
```

`x.batch(batch_size)`

将Dataset x转换为多个batch_size的批训练数据。

举例

```
train_db = train_db.batch(128)
```

`x.map(func)`

将Dataset x按func函数进行预处理。

以MNIST数据集为例。

举例

```
def preprocess(x, y): # 自定义的预处理函数
    x = tf.cast(x, dtype=tf.float32) / 255. # 标准化到 0~1
    x = tf.reshape(x, [-1, 28*28])      # 打平
    y = tf.cast(y, dtype=tf.int32)    # 转成整形张量
    y = tf.one_hot(y, depth=10)    # one-hot 编码
    return x,y
train_db = train_db.map(preprocess)
```

`x.repeat(times)`

让Dataset x循环迭代20个epoch，等同于

```
for epoch in range(20): # 训练 Epoch 数
    for step, (x,y) in enumerate(train_db): # 迭代 Step 数
        # training...
```

举例

```
train_db = train_db(20)
```

常用神经网络运算命令

`tf.matmul(xi, wi)`

用于批量矩阵相乘，常用于计算向量乘积$\mathbf{x} \cdot \mathbf{w}$。

举例

```
o1 = tf.matmul(x, w1) + b_1
```

`tf.keras.layers.Dense(units, activation)`

构造一个输出节点数为units，激活函数为activation的全连接层。

举例

```
from tensorflow.keras import layers
fc = layers.Dense(512, activation=tf.nn.relu)
h1 = fc(x)
```

结果

```
<tf.Tensor: id=72, shape=(4, 512), dtype=float32, numpy=
array([[0.63339347, 0.21663809, 0.        , ..., 1.7361937 ,
0.39962345,
        2.4346168 ],…
```

注意

这个全连接层包括权值矩阵$\mathbf{w}$与偏置矩阵$\mathbf{b}$。

`layer.kernel`

返回全连接层layer的权值矩阵$\mathbf{w}$，layer需要为Dense类。

举例

```
layer.kernel
```

结果

```
<tf.Variable 'dense_1/kernel:0' shape=(784, 512) dtype=float32, numpy=
array([[-0.04067389,  0.05240148,  0.03931375, ..., -0.01595572,
  -0.01075954, -0.06222073],
```

`layer.bias`

返回全连接层layer的偏置向量$\mathbf{b}$，layer需要为Dense类。

举例

```
layer.bias
```

返回

```
<tf.Variable 'dense_1/bias:0' shape=(512,) dtype=float32, numpy=
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
```

`layer.trainable_variables`

返回全连接层layer的所有待优化的参数张量列表，layer需要为Dense类。

举例

```
layer.trainable_variables
```

返回

```
[<tf.Variable 'dense_1/kernel:0' shape=(784, 512) dtype=float32,…,
 <tf.Variable 'dense_1/bias:0' shape=(512,) dtype=float32, numpy=…]
```

`layer.non_trainable_variables`

网络层除了保存了待优化张量 trainable_variables，还有部分层包含了不参与梯度优化的张量，如BatchNormalization层。`layer.non_trainable_variables`返回所有不需要优化的参数列表，layer需要为Dense类。

举例

```
layer.non_trainable_variables
```

`layer.variables`

返回全连接层layer的所有参数列表，layer需要为Dense类。

举例

```
layer.variables
```

注意

对于全连接层，内部张量都参与梯度优化，故`layer.variables`的返回结果与`layer.trainable_variables`一致。

`tf.keras.layers.Sequential([Denses])`

通过 Sequential 容器将各个全连接层封装成一个网络大类对象，Denses为各全连接层。

举例

```
model = layers.Sequential([
    layers.Dense(256, activation=tf.nn.relu) , # 创建隐藏层 1
    layers.Dense(128, activation=tf.nn.relu) , # 创建隐藏层 2
    layers.Dense(64, activation=tf.nn.relu) , # 创建隐藏层 3
    layers.Dense(10, activation=None) , # 创建输出层
])
out = model(x)
```

常用激活函数

Sigmoid

$$f(x)=\frac{1}{1+e^{-x}}$$

Sigmoid函数历史悠久，拥有以下特点：

1. 可以将x压缩到$(0,1)$之间。
2. 连续可导。
3. 在x过大或过小时容易梯度弥散。
4. $f'(x)=f(x)\cdot[1-f(x)]$

使用方法

```
tf.nn.sigmoid(x)
```

ReLU

$$f(x)=\max\{0,x\}$$

ReLU (Rectified Linear Unit)，修正线性单元对小于 0 的值全部抑制为 0；对于正数则直接输出，这种单边抑制特性来源于生物学。

$$f'(x)=\begin{cases}1,x\geq 0\\0,x<0\end{cases}$$

使用方法

```
tf.nn.relu(x)
layers.ReLU()
```

LeakyReLU

$$f(x)=\begin{cases}x, x\geq 0\\ px, x<0\end{cases}$$

p为某个较小的正数。

ReLU函数在$x<0$时恒等于0,亦会梯度弥散。为了弥补这个问题，我们提出LeakyReLU。

$f'(x)=\begin{cases}1, x\geq 0\\ p, x<0\end{cases}$

使用方法

```
tf.nn.leaky_relu(x, alpha)
layers.LeakyReLU(alpha)
```

$\tanh$

$$f(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}=2\cdot\text{sigmoid}(2x)-1$$

其可将x压缩到$[-1,1]$。

$f'(x)=1-\tanh^2(x)$

使用方法

```
tf.nn.tanh()
```

常用损失函数

均方误差 (MSE)

$$f(x)=\frac{1}{n}\sum_{i=1}^{n}(y_i-o_i)^2$$

一般来说，$f(x)\geq 0$。当$f(x)=0$， 神经网络达到最优解。

$\frac{\partial f(x)}{\partial o_i}=(o_i-y_i)$

使用方法

```
loss = tf.keras.losses.MSE(y_onehot, o) # 每个样本的MSE
loss = tf.reduce_mean(loss) # batch的均方差
# 层方法
criteon = tf.keras.losses.MeanSquaredError()
loss = criteon(y_oneot, o)
```

平均绝对误差 (MAE)

$$f(x) = \frac{1}{n}\sum_{i=1}^{n}|y_i - o_i|$$

MAE, Mean Absolute Error.

使用方法

```
loss = tf.keras.losses.MAE(y, out) # 每个样本的MAE
mae_loss = tf.reduce_mean(loss) # batch的平均绝对误差
```

交叉熵

$$H(p,q) = -\sum_{i=0} p(i)\log_2 q(i)$$

在计算交叉熵损失函数时，一般将 Softmax 函数与交叉熵函数统一实现，此时

$$p_i = \frac{e^{z_i}}{\sum\limits_{k=1}^{K} e^{z_k}}$$

且

$$\frac{\partial p_i}{\partial z_j} = \begin{cases} p_i(1-p_j), i = j \\ -p_i \cdot p_j, i \neq j \end{cases}$$

$$\frac{\partial H(p,q)}{\partial z_i} = p_i - y_i$$

使用方法

```
tf.keras.losses.categorical_crossentropy
# 交叉熵函数 + Softmax
tf.keras.losses.categorical_crossentropy(y_true, y_pred,
from_logits=True)
```

其他损失函数

```
tf.keras.losses.mean_squared_error
tf.keras.losses.mean_absolute_error
tf.keras.losses.mean_absolute_percentage_error
tf.keras.losses.mean_squared_logarithmic_error
tf.keras.losses.squared_hinge tf.keras.losses.hinge
```

```
tf.keras.losses.categorical_crossentropy
tf.keras.losses.binary_crossentropy
tf.keras.losses.kullback_leibler_divergence
tf.keras.losses.poisson
tf.keras.losses.cosine_similarity
tf.keras.losses.logcosh
tf.keras.losses.categorical_hinge
# Softmax
tf.nn.softmax(x)
layers.Softmax(axis)
```

网络训练样例

```
Class Network(keras.Model):
    def __init__(self):
        super(Network, self).__init__() # 父类继承
        self.fc = layer.Dense(units, activation) # 构建几个网络层
    def call(self, inputs, training=None, mask=None):
        # 前向传播
        x = self.fc(inputs)
        x = self.fc(x)
        return x
model = Network() # 创建网络实例
model.build(input_shape=(batch_num, dimension))
model.summary() # 打印网络信息
optimizer = tf.keras.optimizers.RMSprop(0.001) # 创建优化器
# 网络训练
for epoch in range(epoch_num): # 遍历 epoch_num 个数据集
    for step, (x,y) in enumerate(train_db): # 遍历以此数据集
        # 梯度记录
        with tf.GradientTape() as tape:
            out = model(x)
            loss = tf.reduce_mean(tf.keras.losses.MSE(y, out))
            if step % 10 == 0:
                print(epoch, step, float(loss))
            # 计算梯度
            grads = tape.gradients(loss, model.trainable_variables)
            # 反向传播
            optimizer.apply_gradient(zip(grads,
model.trainable_variables))
```

训练可视化

TensorFlow提供了Web端监控网络训练进度的专门可视化工具TensorBoard，其可以利用将监控数据写入到文件系统，并利用Web后端监控对应的文件目录，从而远程查看网络监控数据。

模型端

首先创建监控数据的Summary，在需要的时候写入数据

```
import tensorflow as tf
# 创建监控类，监控数据写入log_dir目录
summary_writer = tf.summary.create_file_writer(log_dir)
with summary_writer.as_default():
    # 当前时间戳 step 上的数据为 loss，写入到 ID 位 train-loss 对象中
    tf.summary.scaler('train-loss',float(loss), step=step)
    # 测试成功率
    tf.summary.scaler('test-acc',float(total_correct/total),
step=step)
    # 可视化测试用图片，最多可视化9张
    tf.summary.image('val-onebyone-images:', val_images,
max_outputs=9, step=step)
```

注意

TensorBoard 通过字符串 ID 来区分不同类别的监控数据，因此对于误差数据，我们将它命名为”train-loss”，其他类的数据不可写入此对象，防止数据污染。

浏览器端

运行

```
tensorboard --logdir path
```

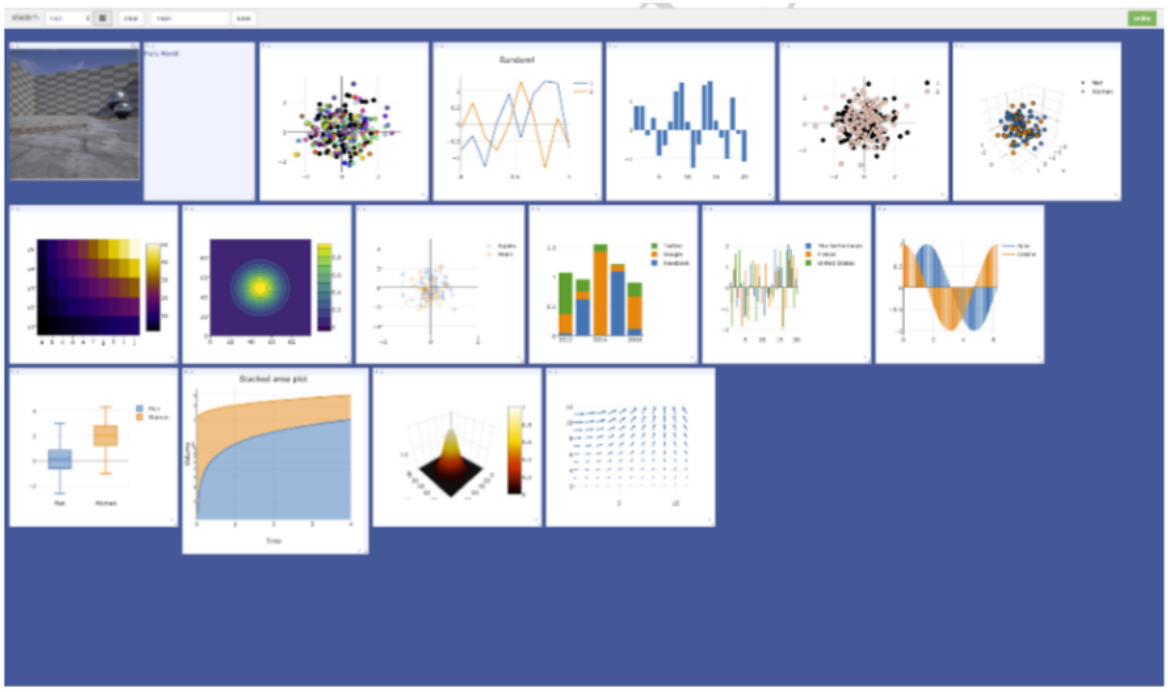
指定Web后端监控的文件目录 path。

除了监控标量数据和图片数据外，TensorBoard还支持通过`tf.summary.histogram`查看张量的数据直方图，以及通过`tf.suummary.text`打印文本信息。

```
with summary_writer.as_default():
    # 当前时间戳 step 上的数据为 loss，写入到 ID 位 train-loss 对象中
    tf.summary.scaler('train-loss', float(loss), step=step)
    # 可视化真实标签的直方图分布
    tf.summary.histogram('y-hist', y, step=step)
    # 查看文本信息
    tf.summary.text('loss-text'. str(float(loss)))
```

后记

除了 TensorBoard 可以监控 TensorFlow 的模型数据外，Facebook 开发的 Visdom 工具也可以可视化数据，而且可视化方式更丰富，实时性更高，使用起来更方便。



Visdom

然而，Visdom 只支持 Pytorch 的张量数据，对于 TensorFlow 的张量类型需要转换成 Numpy 数据。

过拟合与欠拟合

对于欠拟合，我们一般采用增加神经网络层数的方法增强其表达能力。对于过拟合，我们需要采用多种方法解决。

验证集与超参数

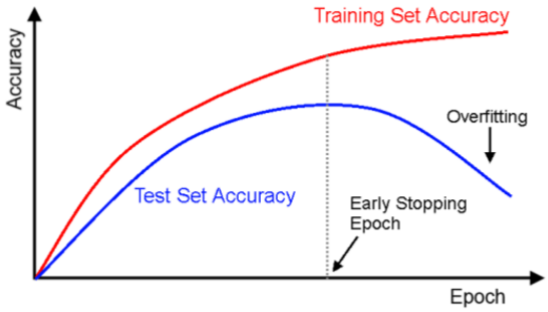
由于测试集的性能不能作为模型训练的反馈，而我们需要在模型训练时能够挑选出较合适的模型超参数，判断模型是否过拟合等，因此需要将训练集再次划分为训练集与验证集。验证集负责选择模型的超参数，功能包括：

1. 根据验证集表现调整学习率，权值衰减洗漱，训练次数等。
2. 根据验证集的性能表现重新调整网络拓扑结构。
3. 根据验证集的性能表现判断是否欠拟合/过拟合。

部分开发人员会错误地使用测试集来挑选最好的模型，然后将其作为模型泛化性能(甚至部分论文也会出现这种做法)汇报，此时的测试集其实是验证集的功能，因此汇报的“泛化性能”本质上是验证集上的性能，而不是真正的泛化性能。为了防止出现这种作弊行为，可以选择生成多个测试集，这样即使开发人员使用了其中一个测试集来挑选模型，我们还可以使用其他测试集来评价模型，这也是 Kaggle 比赛常用的做法。

提前停止

一般来说，我们在数次 step 或 epoch 后使用验证集（一般是几个 epoch 之后）。有时，准确率会先增后减。这时我们就要选择合适的 epoch，到达之后自动停止训练。



示意图

对于分类问题，我们可以记录模型的验证准确率，并监控验证准确率的变化，当发现验证准确率连续 P 个 Epoch 没有下降时，可以预测已经达到了最适合的 epoch 附近，从而提前终止训练，下面是该算法的伪代码：

```
随机初始化参数θ
repeat
    for step = 1,...,N do
        随机采样训练 batch {(x, y)}
        θ ← θ − η∇ℒ(f(x),y)
    end
    数个 epoch 后验证一次
    if 验证性能连续数次不提升 do
        提取停止训练
    end
    until 训练回合数 epoch 达到要求
测试性能
输出 网络参数，测试性能
```

正则化

以多项式函数模型为例：

$$y = \beta_0 + \beta_1 x^1 + \beta_2 x^2 + \cdots + \beta_n x^n + \epsilon$$

以上模型的容量可以通过n简单衡量。在训练中，如果$\beta_{k+1}, \beta_k + \cdots, \beta_n = 0$，则网络的实际容量退化到$k$次多项式的函数容量。因此，通过限制网络参数的稀疏性，可以约束网络的实际容量。

一般，我们通过在损失函数上添加额外的参数稀疏性惩罚项实现。之前，我们需要优化

$$\min \mathcal{L}(f_\theta(x), y), \quad (x, y) \in D^{\text{train}}$$

对模型参数添加额外约束，我们优化

$$\min \mathcal{L}(f_\theta(x), y) + \lambda * \Omega(\theta), \quad (x, y) \in D^{\text{train}}$$

$\Omega(\theta)$表示对网络参数θ的稀疏性约束函数。一般，参数θ的稀疏性约束通过约束参数θ的L范数实现。

新的优化目标除了要最小化原来的损失函数$\mathcal{L}(x,y)$之外，还需要约束网络参数的稀疏性，优化算法会在降低$\mathcal{L}(x,y)$的同时，尽可能地迫使网络参数θ_i变得稀疏，他们之间的权重关系通过超参数λ来平衡，较大的λ意味着网络的稀疏性更重要；较小的λ则意味着网络的训练误差更重要。通过选择合适的λ超参数可以获得较好的训练性能，同时保证网络的稀疏性，从而获得不错的泛化能力。

常用的正则化方式有$L_0, L_1 . L_2$正则化。

L0正则化

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_0$$

其中 L0 范数$\|\theta_i\|_0$定义为θ_i中非零元素的个数。通过约束$\sum_{\theta_i} \|\theta_i\|_0$的大小可以迫使网络中的连接权值大部分为 0，从而降低网络的实际参数量和网络容量。但是由于 L0 范数$\|\theta_i\|_0$并不可导，不能利用梯度下降算法进行优化，在神经网络中使用的并不多。

L1正则化

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_1$$

其中 L1 范数$\|\theta_i\|_1$定义为θ_i中元素的绝对值之和。L1 正则化也叫 Lasso Regularization，它是连续可导的，在神经网络中使用广泛。

实现

```
w = tf.random.normal([4,3])
loss_reg = tf.reduce_sum(tf.math.abs(w))
```

L2正则化

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_2$$

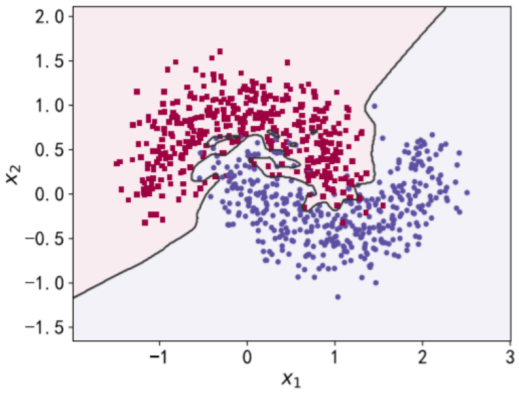
其中 L2 范数$\|\theta_i\|_2$定义为张量θ_i中所有元素的平方和。L2 正则化也叫 Ridge Regularization，它和 L1 正则化一样，也是连续可导的，在神经网络中使用广泛。

实现

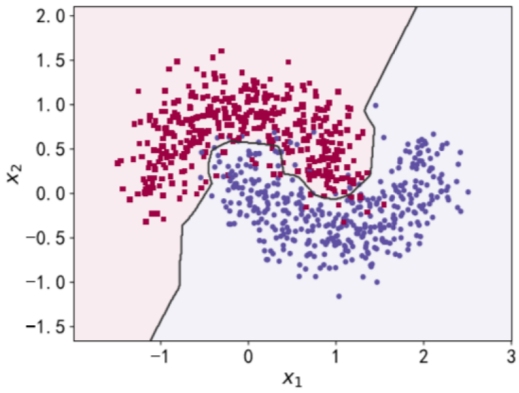
```
loss_reg = tf.reduce_sum(tf.square(w))
```

正则化效果

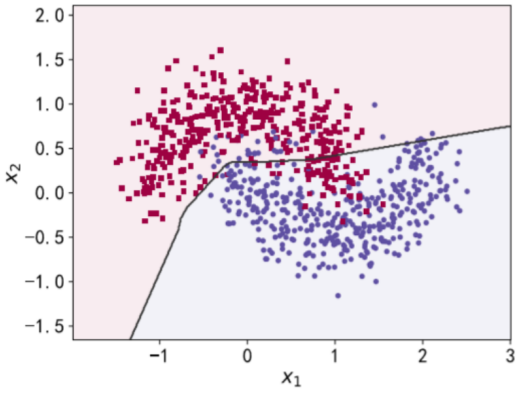
以`sklearn.datasets.make_moon`为例，在损失函数上添加 L2 正则化项，并通过改变不同的正则化超参λ来获得不同程度的正则化效果。如下图，随着正则化系数λ的增加，网络对参数稀疏性的惩罚变大，从而迫使优化算法搜索而网络容量更小的模型。在$\lambda = 0.00001$时，正则化的作用比较微弱，网络出现了过拟合现象；但是 $\lambda = 0.1$时，网络已经能够优化到合适的容量，并没有出现明显过拟合或者欠拟合现象。



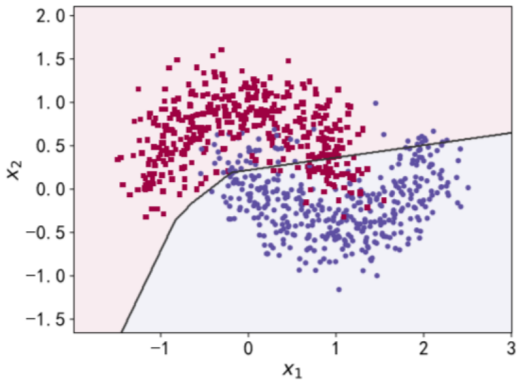
正则化系数：0.00001



正则化系数：0.001



正则化系数：0.1



正则化系数：0.13

不同 λ 下网络性能对比

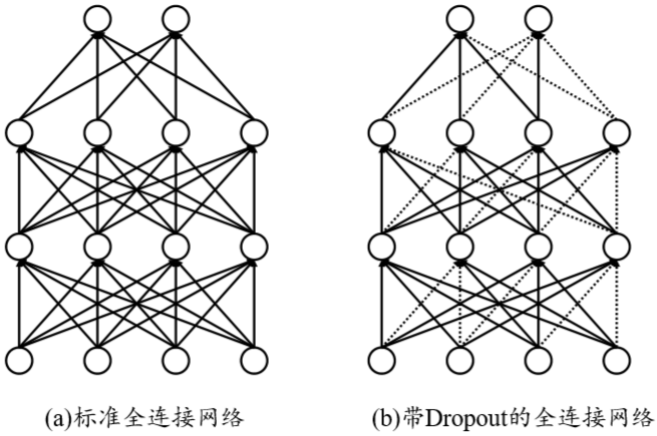
实际训练时，一般先尝试较小的正则化系数λ，观测网络是否出现过拟合现象。然后尝试逐渐增大λ参数来增加网络参数稀疏性，提高泛化能力。但是，过大的λ参数有可能导致网络不收敛，需要根据实际任务调节。

实际上，正则化让网络权值的平均值趋于0。以上图情况为例，我们做出以下表格：

正则化系数λ	W 最小值	W 最大值	W 平均值
0.00001	-1.6088	1.1599	0.0026
0.001	-0.1393	0.3168	0.0003
0.1	-0.0969	0.0832	0
0.13	-0.1104	0.0785	0

Dropout

Dropout 通过随机断开神经网络的连接，减少每次训练时实际参与计算的模型的参数量；但是在测试时， Dropout 会恢复所有的连接，保证模型测试时获得最好的性能。



Dropout 示意图

每条连接是否断开符合某种预设的概率分布，如断开概率为p的伯努利分布。

随着 Dropout 层的增加，网络模型训练时的实际容量减少，泛化能力变强。

实现

rate 参数为断开的概率值p。

```
x = tf.nn.dropout(x, rate=p)
# 层实现
model.add(layers.Dropout(rate = 0.5))
```

数据增强

我们都知道增加数据集大小是解决过拟合的有效方法。然而，收集样本数据和标注代价昂贵。取而代之，我们通过数据增强技术增加训练的样本数量，在一定程度上增强性能。数据增强(Data Augmentation)是指在维持样本标签不变的条件下，根据先验知识改变样本的特征，使得新产生的样本也符合或者近似符合数据的真实分布。

以人脸照片为例，我们知道旋转、缩放、平移、裁剪、改变视角、遮挡某些局部区域不会改变图片的类别标签，因此我们有多种数据增强技术。

此外，我们可以通过生成模型在原有数据上学习到数据的分布，从而生成新的样本，这种方式在一定程度上可以提升网络性能。条件生成网络(Conditional GAN, CGAN)可以生成带标签的样本数据。

CGAN 生成的手写数字图片

实现

```
def preprocess(x,y):
    # x: 图片路径 y: 图片的数字编码
    x = tf.io.read_file(x)
    x = tf.image.decode_jpeg(x, channels=3) # RGBA
    # 图片缩放到244x244， 大小根据网络设定调整
    x = tf.image.resize(x, [244, 244])
    # 旋转k个90度
    x = tf.image.rot90(x, k)
    # 随机水平翻转
    x = tf.image.random_flip_left_right(x)
    # 随机竖直反转
    x = tf.image.random_flip_up_down(x)
    # 裁剪
    # 先缩放到较小的尺寸
    x = tf.image.resize(x, [224, 224])
    # 随机剪裁到合适尺寸
    x = tf.image.random_crop(x, [224, 224, 3])
```

其他方式

除了上述介绍的典型图片数据增强方式以外，可以根据先验知识，在不改变图片标签信息的条件下，任意变换图片数据，获得新的图片。例如，在原图上叠加高斯噪声、改变图片的观察视角或是随即遮挡部分区域获得新图片。

CNN

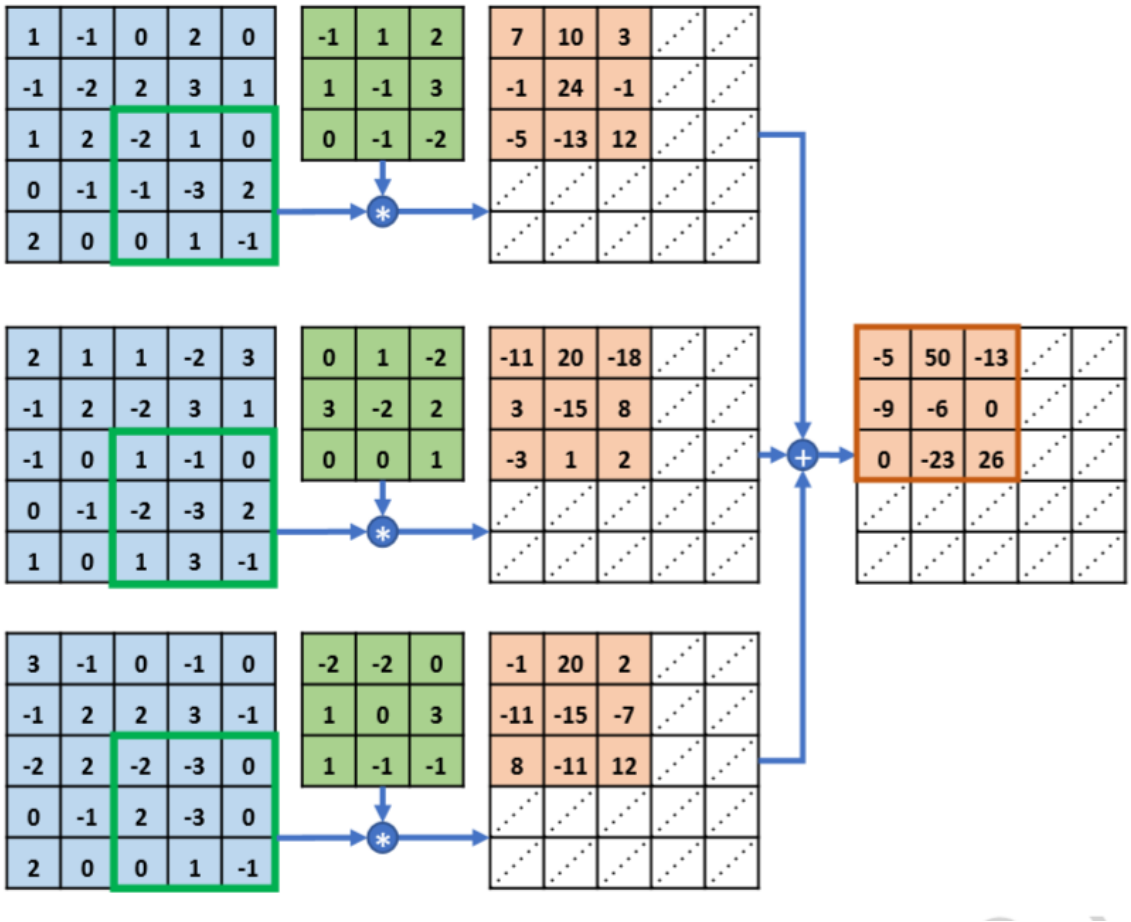
Convolutional Neural Network

概念梳理

单卷积核，多通道输入

最后将各通道结果相加。这样，最后只能得到一个输出矩阵。

举例



多卷积核，多通道输入

最后的结果为n（n为通道数）个单卷积核，多通道的输出矩阵的拼接（`tf.stack`）输出。

常见的卷积核

原图效果

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

锐化效果

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

模糊效果

$$\begin{pmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{pmatrix}$$

边缘提取效果

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

输出矩阵

设卷积核的数量为c_{out}，卷积核大小为k，步长为s，填充数为p（上下填充p_{h}相同，左右填充p_{w}相同），步长为s；输入X的高为h，宽为w，则输出矩阵的尺寸$[b, h', w', c_{\text{out}}]$有：

$$h' = \left[\frac{h + 2 \cdot p_{\text{h}} - k}{s}\right] + 1$$

$$w' = \left[\frac{w + 2 \cdot p_{\text{w}} - k}{s}\right] + 1$$

填充（Padding）

在输出矩阵四周填充0。一般地，我们使得达到与输入矩阵shape相同。

代码实现

普通方法

```
out = tf.nn.conv2d(x, w, strides=1, padding=[[0,0],[0,0],[0,0],[0,0]])
```

strides为步长。

其中，padding的设置格式为

```
padding = [[0,0],[上， 下],[左,右],[0,0]]
```

特别地，通过设置参数 `strides=1, padding='SAME'` 可以直接得到输入、输出同大小的卷积层，其中 padding 的具体数量由 TensorFlow 自动计算并完成填充操作。当$s > 1$时，设置 `padding='SAME'` 将使得输出高、宽将变为原来的$\frac{1}{s}$。

下面给出一个例子

举例

```
x = tf.random.normal([2,5,5,3])
w = tf.random.normal([3,3,3,4])
out = tf.nn.conv2d(x, w, strides=1, padding='SAME')
print(out.shape)
```

结果

```
(2, 5, 5, 4)
```

卷积层类

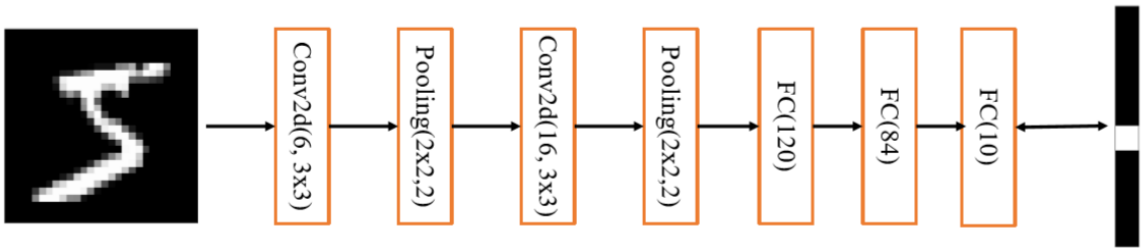
举例

```
layer = layers.conv2D(filters, kernel_size=size, stides,
padding='SAME')
out = layer(x)
```

filters为卷积核数量；kernel_size为卷积核大小：若size为整数则为size · size的卷积核，若想长宽不等，则需输入$(k_{\mathrm{h}}, k_{\mathrm{w}})$，第一，第二个分别为卷积核的长，宽；strides同理：第一，第二个分别为竖直，水平步长。

LeNet-5实战

输入X为28×28的矩阵，结果图如下：



实现

```
% tensorflow_version 2.x
import tensorflow as tf
from tensorflow.keras import Sequential, losses, optimizers, layers
(x,y),(x_test,y_test) = tf.keras.datasets.mnist.load_data()
x = tf.cast(x, dtype=tf.float32)
x_test = tf.cast(x_test, dtype=tf.float32)
db_train = tf.data.Dataset.from_tensor_slices((x, y))
db_train = db_train.shuffle(1000).batch(128)
db_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
db_test = db_test.batch(128)
network = Sequential([
    layers.Conv2D(6, kernel_size=3, strides=1), # 卷积层，6个3x3卷积核
    layers.MaxPooling2D(pool_size=2, strides=2), # 最大池化层，高宽减半
    layers.ReLU(), # 激活函数
    layers.Conv2D(16, kernel_size=3, strides=1), # 卷积层，16个3x3卷积
核
    layers.MaxPooling2D(pool_size=2, strides=2), # 最大池化层，高宽减半
    layers.ReLU(), # 激活函数
    layers.Flatten(), #打平
    layers.Dense(120, activation='relu'), # 全连接层，120个神经元
    layers.Dense(84, activation='relu'), # 全连接层，84个神经元
    layers.Dense(10) # 全连接层，10个神经元
])
network.build(input_shape=(4,28,28,1))
network.summary()
criteon = losses.CategoricalCrossentropy(from_logits=True) # SoftMax +
交叉熵
correct, total = 0, 0
optimizer = optimizers.Adam(0.01)

```

```
#学习
for epoch in range(2):
    for step, (x, y) in enumerate(db_train):
        # 构建梯度记录环境
        with tf.GradientTape() as tape:
            x = tf.expand_dims(x, axis=3)
            out = network(x)
            y_onehot = tf.one_hot(y,depth=10)
            loss = criteon(y_onehot, out)
        grads = tape.gradient(loss, network.trainable_variables)
        optimizer.apply_gradients(zip(grads,
network.trainable_variables))
        if step % 100 == 0:
            print('step',step,'loss',float(loss))
# 测试
for x_test,y_test in db_test:
    out = network(x)
    pred = tf.argmax(out, axis=-1)
    y = tf.cast(y, tf.int64)
    correct += float(tf.reduce_sum(tf.cast(tf.equal(pred,
y),tf.float32)))
    total += x.shape[0]
print('test accuracy:', (correct/total))
```

表示学习

我们通过将每层的特征图利用反卷积网络映射回图片，以观察学到的特征分布。我们发现，结果总是从边，角，色彩等底层图像到纹理等中层特征到具体物体的部分特征。

训练好的卷积神经网络往往能够学习到较好的特征，这种特征的提取方法一般是通用的。比如在猫、狗任务上学习到头、脚、身躯等特征的表示，在其他动物上也能够一定程度上使用。基于这种思想，可以将在任务 A 上训练好的深层神经网络的前面数个特征提取层迁移到任务 B 上，只需要训练任务 B 的分类逻辑(表现为网络的最末数层)，即可取得非常好的效果，这种方式是迁移学习的一种，从神经网络角度也称为网络微调(Fine-tuning)。

梯度传播

$$\frac{\partial L}{\partial w_{ij}} = \sum_{i,j} \frac{\partial L}{\partial o_i j} \cdot \frac{\partial o_i j}{\partial w_i j}$$

池化层

目的：减少矩阵大小，以此减少参数。

最大池化层

$$x' = \max\{X\}$$

平均池化层

$$x' = \text{avg}\{X\}$$

BatchNorm层

以Sigmoid为例，当x极小和极大时，容易造成梯度弥散。我们将输出矩阵的数据标准化到$[0,1]$之间，以此优化性能。

令所有数据的均值为μ，方差为σ_r^2。

$$\hat{x} = \frac{x - \mu_r}{\sqrt{\sigma_r^2 + \epsilon}}$$

ϵ是一个较小的数字，如$1e-8$，以防止除以0产生错误。

在训练每个Batch时，其均值与方差与整个数据集的相应不大。因此我们分别计算每个Batch的均值，方差：

$$\mu_{\mathrm{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\sigma_{\mathrm{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathrm{B}})^2$$

这两个值与整体样本的相应值相近，因此我们最终得到：

$$\hat{x}_{\mathrm{train}} = \frac{x_{\mathrm{train}} - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \epsilon}}$$

在测试阶段，我们先根据每个Batch的均值与方差得到所有训练数据的均值与方差 （每个Batch的数据近似服从同一种分布）：

$$\mu_r = \frac{1}{n}\sum_{i=1}^{n} \mu_{\mathrm{B}_i}$$
$$\sigma_{\mathrm{r}}^2 \approx \sum_{i=1}^{n} \sigma_{\mathrm{B_i}}^2$$

于是

$$\hat{x}_{\mathrm{test}} = \frac{x_{\mathrm{test}} - \mu_r}{\sqrt{\sigma_{\mathrm{r}}^2 + \epsilon}}$$

上述的标准化运算并没有引入额外的待优化变量，均值与方差不需要参与梯度更新。更进一步地，我们引入 "scale and shift"技巧，将$\hat{x}$再次映射变换：

$$\tilde{x} = \hat{x} * \gamma + \beta$$

其中γ参数实现对标准化后的$\hat{x}$再次进行缩放，β参数实现对标准化的$\hat{x}$进行平移。不同的 是，γ, β参数均由反向传播算法自动优化，实现网络层“按需”缩放平移数据的分布的目的。

前向传播

将BN层的输入记为x，输出为$\tilde{x}$。

训练阶段

计算当前Batch的$\mu_{\mathrm{B}}, \sigma_{\mathrm{B}}^2$，根据

$$\tilde{x}_{\text{train}} = \frac{x_{\text{train}} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} * \gamma + \beta$$

计算BN层的输出$\tilde{x}$

同时迭代更新全局训练数据的均值与方差，momentum为需要设置的一个超参数。

$$\mu_r \leftarrow \text{momentum}^* \mu_r + (1 - \text{momentum}) * \mu_B$$
$$\sigma_r^2 \leftarrow \text{momentum} * \sigma_r^2 + (1 - \text{momentum}) * \sigma_B^2$$

当monentum=0, μ_{r}和σ_{r}^2直接被设置为最新一个 Batch 的μ_{b}和σ_{B}^2。当monentum=1，整个数据集的均值与方差保持不变。在TensorFlow中，momentum默认识别为0.99。

测试阶段

BN层根据

$$\tilde{x}_{test} = \frac{x_{test} - \mu_r}{\sqrt{\sigma_r^2 + \epsilon}} * \gamma + \beta$$

计算输出$\tilde{x}_{test}$。$\mu_r, \sigma_r^2, \gamma, \beta$为迭代更新的结果，在测试阶段不会更新参数。

反向更新

根据损失$\mathcal{L}$求解梯度$\frac{\partial \mathcal{L}}{\partial \gamma}, \frac{\partial \mathcal{L}}{\partial \beta}$，并优化相关参数。

对于多通道的输入X，$\mu_{\text{B}}, \sigma_{\text{B}}^2$是每个通道上所有其他维度的均值和方差。

举例

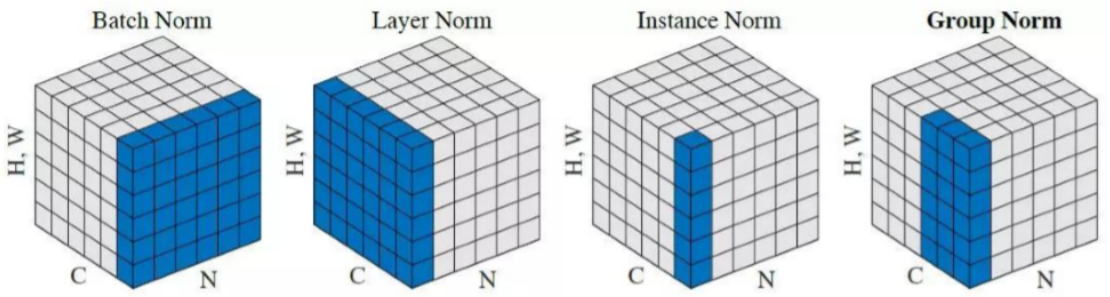
```
x = tf.random.normal([100,32,32,3])
x = tf.reshape(x,[-1,3])
print(x.shape)
ub = tf.reduce_mean(x, axis=0)
print(ub.shape)
```

输出

```
(102400, 3)
(3,)
```

除了在c轴上面统计数据$\mu_{\text{B}}, \sigma_{\text{B}}^2$的方式，我们也很容易将其推广至其他维度计算均值的方式，比如：

1. Layer Norm：统计每个样本的所有特征的均值和方差。
2. Instance Norm：统计每个样本的每个通道上特征的均值和方差 。
3. Group Norm：将c通道分成若干组，统计每个样本的通道组内的特征均值和方差 。



示意图

实现

```
layer = layers.BatchNormalization(training=True) # 训练
layer = layers.BatchNormalization(training=False) # 测试
# 训练阶段
with tf.GradientTape() as tape:
    x = tf.expand_dims(x, axis=3)
    out = network(x, training=True)
# 测试阶段
for x,y in db_test:
    x = tf.expand_dims(x, axis=3)
    out = network(x, training=False)
```

经典卷积网络

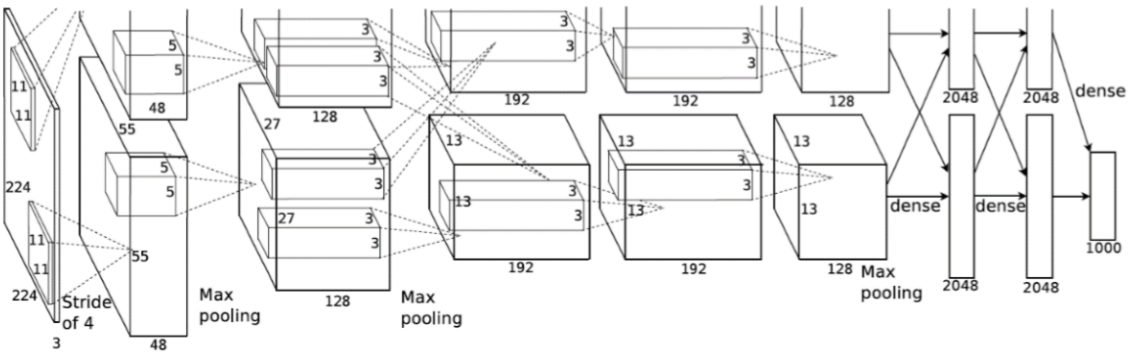
AlexNet

一种2012年提出的8层深度神经网络模型。它接收输入为 224x224 大小的彩色图片数据，经过五个 卷积层和三个全连接层后得到样本属于 1000 个类别的概率分布。

由于性能限制，作者将卷积层、前 2 个全连接层等拆开在两块显卡上面分别训练，最后一层合并到一 张显卡上面，进行反向传播更新。

为了降低特征图的维度，AlexNet 在第 1、2、5 个卷积层后添加了 Max Pooling 层，网络的参数量达到了 60M 个。AlexNet 的创新之处在于：

1. 层数达到了较深的 8 层。
2. 采用了 ReLU 激活函数，过去的神经网络大多采用 Sigmoid 激活，计算相对复杂，容易出现梯度弥散现象 。
3. 引入 Dropout，Dropout 提高了模型的泛化能力，防止过拟合。



示意图

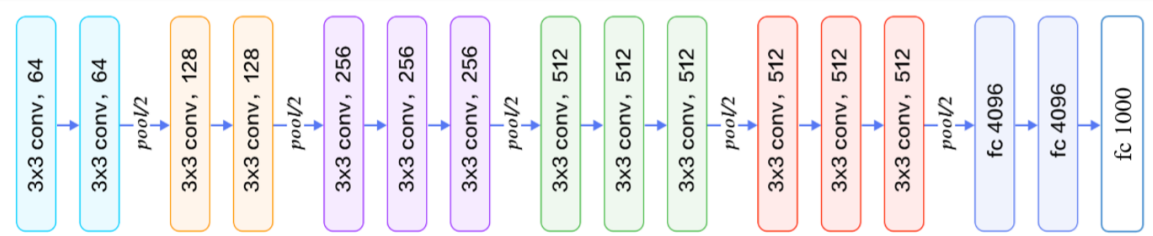
VGG

分为VGG11，VGG13，VGG16，VGG19等一系列网络芈姓，将网络深度最高提升到19层。

其亦接受224x224大小的彩色图片数据，经过两个Conv-Conv-Pooling单元和3个Conv-Conv-Conv-Pooling单元的堆叠，最后通过3层全裂阶层输出1000类别的概率分布。

其创新之处在于：

1. 层数提升至19层
2. 全部采用更小的3x3卷积核，而不是AlexNet中的7x7卷积核，参数更少，算力要求小。
3. 采用更小的2x2池化层与步长s=2，而AlexNet采用3x3的池化层。

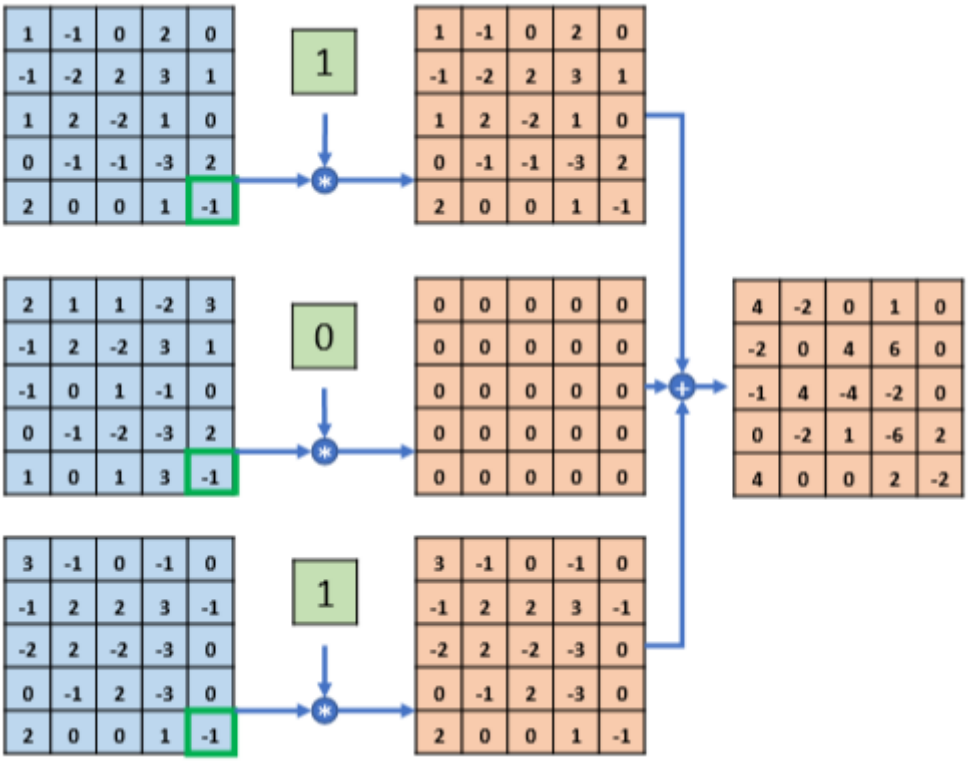


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

示意图

GoogLeNet

以性能更佳为目的，我们寻找到最小的1x1卷积核，最终将所有中间矩阵相加。这种卷积核不改变特征图的宽高，只对通道数c进行变换。

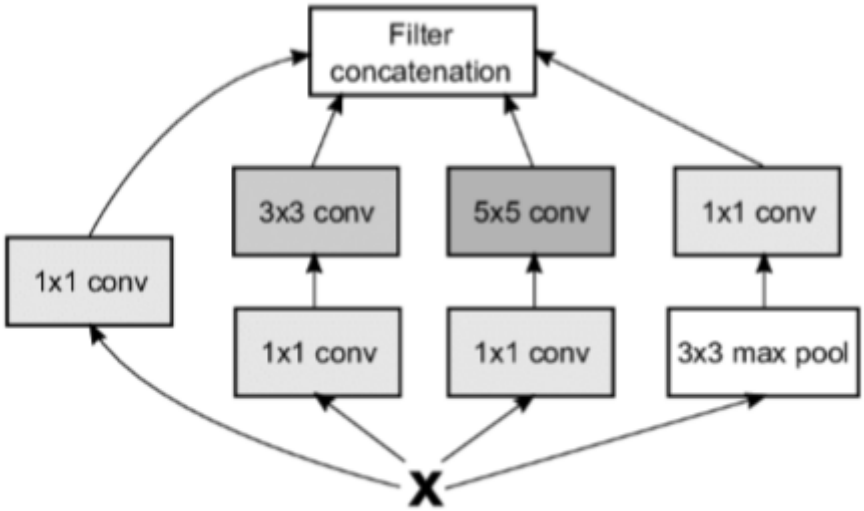


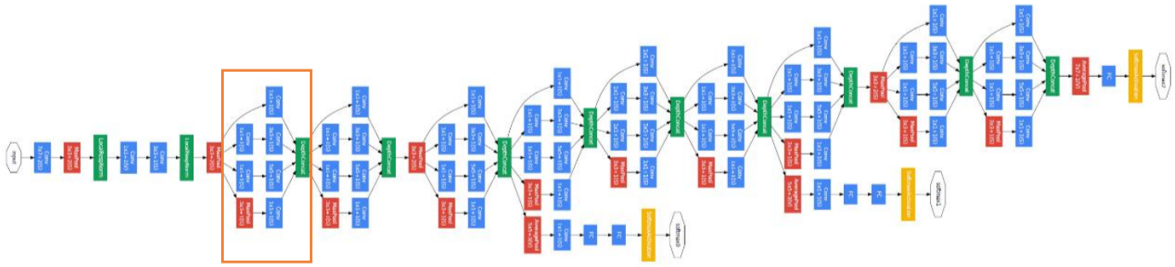
示意图

GoogLeNet的网络层达到22层，参数量却只有AlexNet的$\frac{1}{12}$，性能也更佳。

GoogLeNet采用模块化设计，大量堆叠Inception模块。输入为X，通过4个子网络得到4个网络输出，在通道轴频频姐，形成输出。4个子网络是：

1. 1x1卷积层
2. 1x1卷积层，再通过一个3x3卷积层
3. 1x1卷积层，再通过一个5x5卷积层
4. 3x3最大池化层，再通过一个1x1卷积层



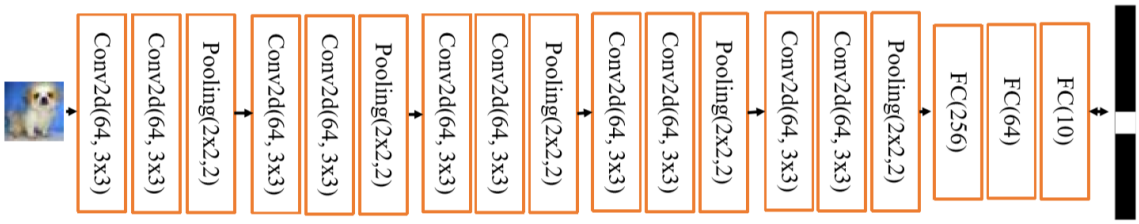


示意图

CIFAR10与VGG13实战

CIFAR10包括10大类彩色图片，每个种类6000张32x32图片，5000张训练，1000张测试集。

使用VGG13时，我们将网络输入调整为32x32，3个全连接层的维度调整为[256,64,10]，以此10分类。



修改过的网络示意图

```
import tensorflow as tf
from tensorflow.keras import datasets, Sequential, layers, losses, optimizers
def preprocess(x, y):
    x = tf.cast(x, dtype=tf.float32)
    y = tf.cast(y, dtype=tf.int32)
    return x, y
# 下载数据集
(x,y), (x_test, y_test) = datasets.cifar100.load_data()
# 维度 [b,1]->[b]
y = tf.squeeze(y, axis=1)
y_test = tf.squeeze(y_test, axis=1)
db_train = tf.data.Dataset.from_tensor_slices((x,y))
db_train = db_train.shuffle(1000).map(preprocess).batch(64)
db_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
db_test = db_test.shuffle(1000).map(preprocess).batch(64)
conv_net = Sequential([
    # Conv 1
    layers.Conv2D(64, kernel_size=[3,3], padding='SAME', activation=tf.nn.relu),
    layers.Conv2D(64, kernel_size=[3,3], padding='SAME', activation=tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2], strides=2, padding='SAME'),
    # Conv 2
    layers.Conv2D(128, kernel_size=[3,3], padding='SAME', activation=tf.nn.relu),
    layers.Conv2D(128, kernel_size=[3,3], padding='SAME', activation=tf.nn.relu),
```

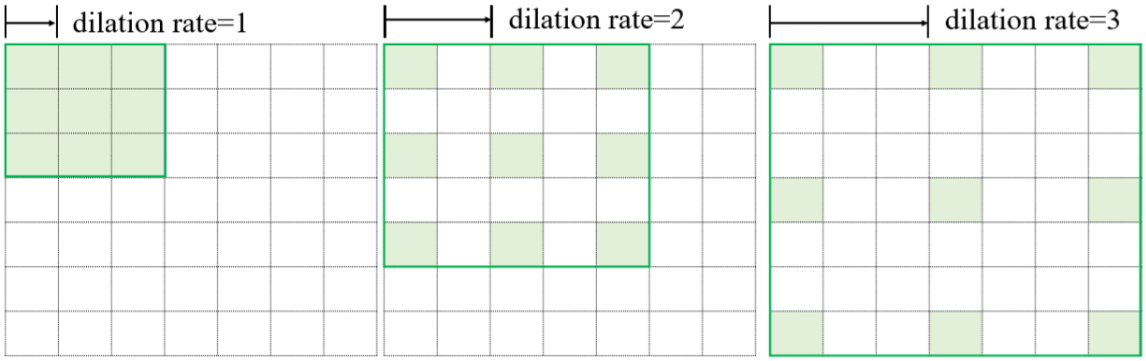
```
    layers.MaxPool2D(pool_size=[2,2], strides=2, padding='SAME'),
    # Conv 3
    layers.Conv2D(256, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.Conv2D(256, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2], strides=2, padding='SAME'),
    # Conv 4
    layers.Conv2D(512, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.Conv2D(512, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2], strides=2, padding='SAME'),
    # Conv 5
    layers.Conv2D(512, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.Conv2D(512, kernel_size=[3,3], padding='SAME',
activation=tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2], strides=2, padding='SAME'),
])
fc_net = Sequential([
    layers.Dense(256, activation=tf.nn.relu),
    layers.Dense(128, activation=tf.nn.relu),
    layers.Dense(10, activation=None),
])
conv_net.build(input_shape=[None,32,32,3])
fc_net.build(input_shape=[None,512])
conv_net.summary()
fc_net.summary()
variables = conv_net.trainable_variables + fc_net.trainable_variables
#学习
optimizer = optimizers.Adam(0.01)
criteon = losses.CategoricalCrossentropy(from_logits=True)
for epoch in range(3):
    for step, (x, y) in enumerate(db_train):
        # 构建梯度记录环境
        with tf.GradientTape() as tape:
            # [b,32,32,3]
            out = conv_net(x)
            # flatten ==> [b,512]
            out = tf.reshape(out, [-1, 512])
            # [b,512] --> [b,100]
            out = fc_net(out)
            # [b] --> [b,100]
            y_onehot = tf.one_hot(y,depth=10)
            # compute loss
            loss = criteon(y_onehot, out)
            loss = tf.reduce_min(loss)
        grads = tape.gradient(loss, variables)
        optimizer.apply_gradients(zip(grads, variables))
        if step % 1000 == 0:
```

```
            print('epoch: %d; step: %d; loss: %f'%((epoch+1), step, float(loss)))
# 测试
correct, total = 0, 0
for x,y in db_test:
    # x = tf.expand_dims(x, axis=3)
    out = conv_net(x)
    out = tf.reshape(out, [-1, 512])
    out = fc_net(out)
    pred = tf.argmax(out, axis=-1)
    y = tf.cast(y, tf.int64)
    correct += float(tf.reduce_sum(tf.cast(tf.equal(pred, y),tf.int32)))
    total += x.shape[0]
print('test accuracy: %4f'%(correct/total))
```

卷积层变种

空洞卷积

空洞卷积 (Dilated/ Atrous Convolution) 在普通卷积层的感受野上增加一个 dialation rate 参数，控制感受野区域的步长。尽管 dilation rate 的增大会使得感受野区 域增大，但是实际参与运算的点数仍然保持不变。



示意图

空洞卷积在不增加网络参数的条件下，提供了更大的感受野窗口。但是在使用空洞卷积设置网络模型时，需要精心设计 dilation rate 参数来避免出现网格效应，同时较大的 dilation rate 参数并不利于小物体的检测、语义分割等任务。

实现

```
import tensorflow as tf
from tensorflow.keras  import layers
x = tf.random.normal([1,7,7,1])
layer = layers.Conv2D(1, kernel_size=3, strides=1, dilation_rate = 2)
out = layer(x)
print(out.shape)
```

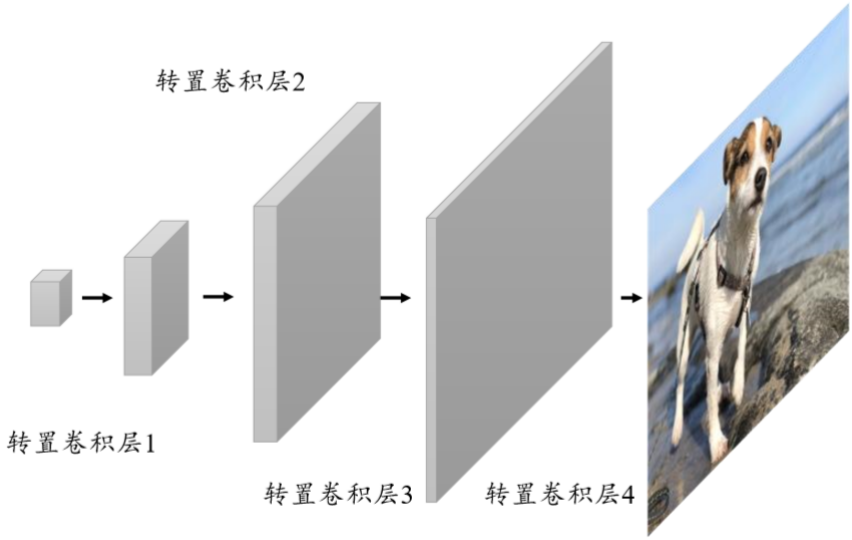
结果

```
(1, 3, 3, 1)
```

转置卷积

通过在输入之间填充大量的padding来实现输出高宽大于输入高宽，从而实现向上采样的目的。

为了简化讨论，我们此处只讨论输入$h=w$，即卷积核高宽相等的情况。



示意图

o+2p-k 为 s 倍数

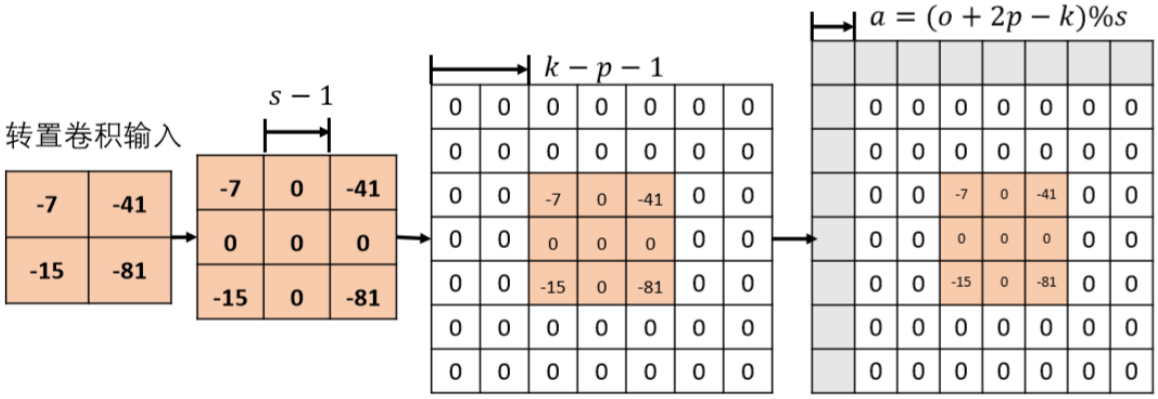
以2x2单通道特征图，3x3卷积核，步长s=2，填充p=0为例。

首先，在数据点之间均匀插入$s-1$个空白数据点，将2x2矩阵填充为3x3矩阵。然后，在3x3矩阵周围填充$k-p-1$（在这道题$=3-0-1=2$）行列，将3x3矩阵转换为7x7矩阵。

我们在这个7x7的矩阵上进行3x3卷积核，步长s'=1，填充p=0的普通卷积运算，输出长宽为：

$$o=\left[\frac{i+2*p-k}{s'}\right]+1=\left[\frac{7+2*0-3}{1}\right]+1=5$$

因此，我们得到5x5的输出。



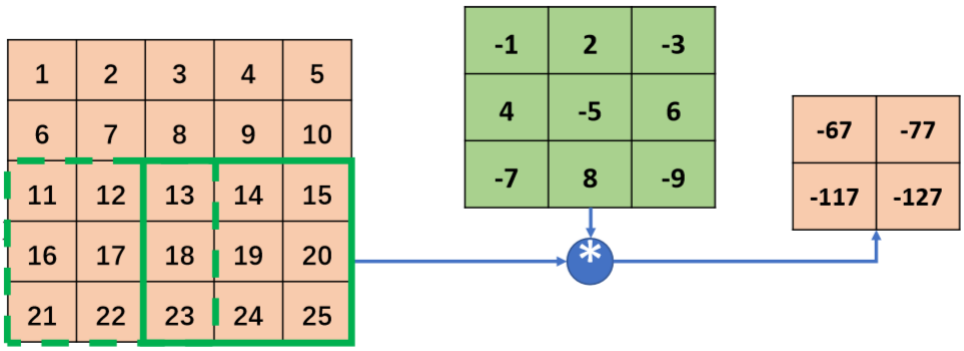
示意图

当$o+2p-k$为s倍数时，

$$o=(i-1)*s+k-2*p$$

转置卷积并不是普通卷积的逆过程，但是二者之间有一定的联系，同时转置卷积也是基于普通卷积实现的。在相同的设定下，输入经过普通卷积运算后得到o = Conv(x)，我们将o送入转置卷积运算后，得到 x' = ConvTranspose(o)，其中 x ≠ x'，但是 x 与 x' 形状相同。

我们可以用输入为 5x5，步长$s = 2$，填充$p = 0$，3x3 的普通卷积运算进行验证演示：



示意图

实现

```
import tensorflow as tf
x = tf.range(25)+1
x = tf.reshape(x, [1,5,5,1])
x = tf.cast(x, tf.float32)
w = tf.constant([[-1.,2.,-3.],[4.,-5.,6.],[-7.,8.,-9.]])
w = tf.expand_dims(w, axis=2)
w = tf.expand_dims(w, axis=3)
out = tf.nn.conv2d(x, w, strides=2, padding='VALID')
print(out.shape)
```

结果

```
(1, 2, 2, 1)
```

现在我们将普通卷积的输出作为转置卷积的输入，验证转置卷积的输出是否为 5x5。

```
import numpy as np
xx = tf.nn.conv2d_transpose(out, w, strides=2, padding='VALID',
output_shape=[1,5,5,1])
print(xx.shape)
print(np.any(x == xx))
```

结果

```
(1, 5, 5, 1)
False
```

可以看到，转置卷积能够恢复出同大小的普通卷积的输入，但转置卷积的输出并不等同于普通卷积的输入。

o+2p-k 不为 s 倍数

由卷积运算的输出o满足

$$o = \left[\frac{i + 2 * p - k}{s}\right] + 1$$

当$s > 1$，$\left[\frac{i + 2 * p - k}{s}\right]$向下取整会让多种不同的输入尺寸 i 对应到相同的输出尺寸o上。

我们以6x6的输入矩阵，3x3的卷积核，步长为1举例

```
x = tf.random.normal([1,6,6,1])
w = tf.constant([[-1.,2.,-3.],[4.,-5.,6.],[-7.,8.,-9.]])
w = w.reshape(w, [3,3,1,1])
out = tf.nn.conv2d(x, w, strides=2, padding='VALID')
print(out.shape)
```

结果

```
(1, 2, 2, 1)
```

与以上输入为 5x5，步长$s = 2$，填充$p = 0$，3x3 的普通卷积运算相比，其输出了相同尺寸的矩阵。从转置卷积的角度来说，输入尺寸 i 经过转置卷积运算后，可能获得不同的输出 o 大小。因此通过我们通过填充行列实现不同大小的输出 o，从而恢复普通卷积不同大小的输入的情况。我们有

$$a = \left[\frac{o + 2p - k}{s}\right]$$

此时转置卷积的输出变为：

$$o = (i - 1) * s + k - 2 * p + a$$

在 TensorFlow 中，不需要手动指定a参数，只需要指定输出尺寸即可，TensorFlow 会自动推导需要填充的行列数a，前提是输出尺寸合法

```
xx = tf.nn.conv2d_transpose(out, w, strides=2, padding='VALID',
output_shape=[1,6,6,1])
print(xx.shape)
```

结果

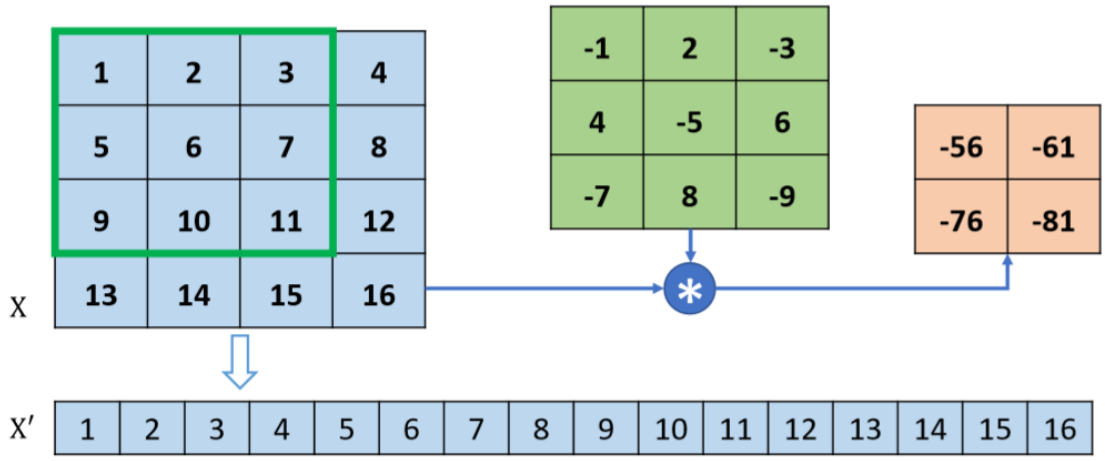
```
(1, 6, 6, 1)
```

矩阵角度

转置卷积的转置是指卷积核矩阵 W 产生的稀疏矩阵 W'在计算过程中需要先转置 W'^T，再进行矩阵相乘运算，而普通卷积并没有转置 W'的步骤。这也是它被称为转置卷积的名字由来。

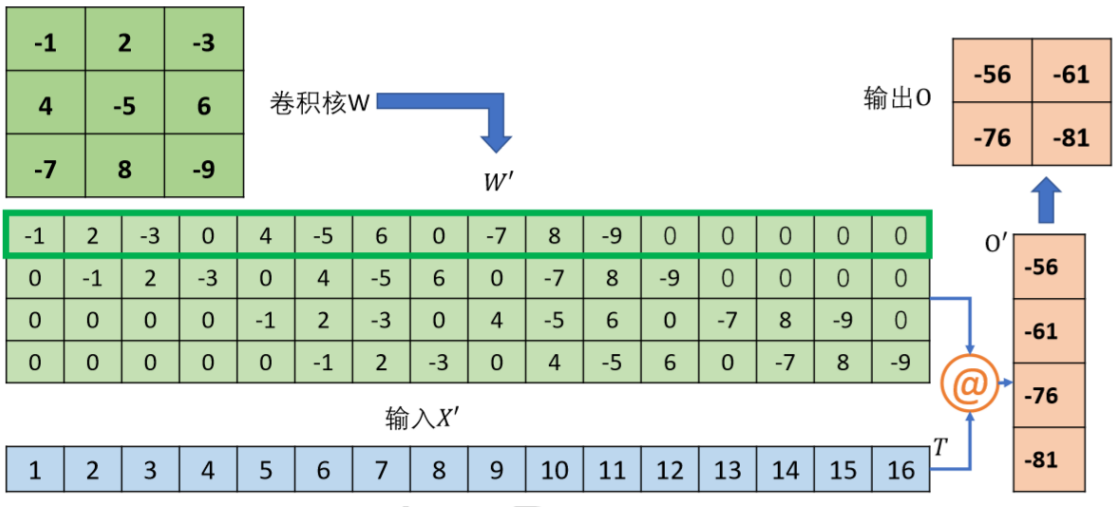
为了加速运算，我们有时将卷积核W根据strides重排成稀疏矩阵W'，再计算 W'@X' 一次完成运算。

以 4 行 4 列的输入 X，高宽为 3x3，步长为 1，无 padding 的卷积核 W 为例，首先将 X 打平成一维形式X'：



示意图

然后我们从卷积核 W 转换成稀疏矩阵 W'：原理是卷积核和输入矩阵不重叠的部分为0，以达成并行计算。



生成稀疏矩阵 W′

此时通过

$$O' = W' @ X'$$

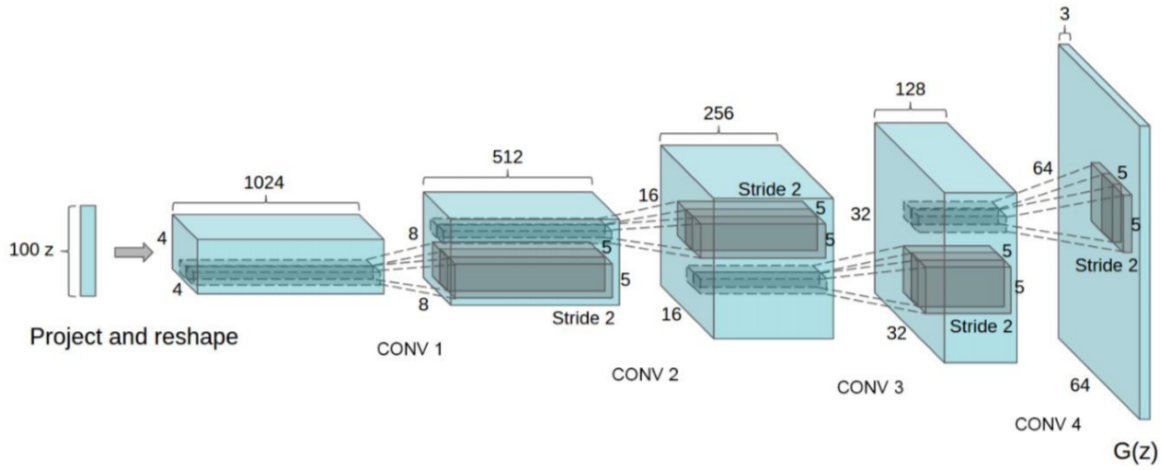
即可得到输出矩阵O。

同时，通过以下操作可以生成与X同形状大小的张量X'：

$$X' = W'^{T} @ O'$$

然后通过reshape操作可以得到与原来X尺寸一致，内容不同的矩阵。

由于转置卷积在矩阵运算时，需要将W'转置后才能与转置卷积的输入O 矩阵相乘，故称为转置卷积。 转置卷积具有"放大特征图"的功能，在生成对抗网络、语义分割等中得到了广泛应用，如 DCGAN 中的生成器通过堆叠转置卷积层实现逐层 "放大"特征图，最后获得十分逼真的生成图片。



DCGAN示意图

实现

```
x = tf.range(16) + 1
x = tf.reshape(x, [1,4,4,1])
x = tf.cast(x, tf.float32)
w = tf.constant([[-1.,2.,-3.],[4.,-5.,6.],[-7.,8.,-9.]])
w = tf.reshape(w, [3,3,1,1])
out = tf.nn.conv2d(x, w, strides=1, padding='VALID')
xx = tf.nn.conv2d_transpose(out, w, strides=1, padding='VALID',
output_shape=[1,4,4,1])
xx = tf.reshape(xx, [4,4])
print(out)
print(xx)
```

结果

```
tf.Tensor(
[[[[-56.]
   [-61.]]

  [[-76.]
   [-81.]]]], shape=(1, 2, 2, 1), dtype=float32)
tf.Tensor(
[[  56.  -51.   46.  183.]
 [-148.  -35.   35. -123.]
 [  88.   35.  -35.   63.]
 [ 532.  -41.   36.  729.]], shape=(4, 4), dtype=float32)
```

可见，out 与 xx 形状相同而值不同。

在使用 `tf.nn.conv2d_transpose` 进行转置卷积运算时，需要额外手动设置输出的高宽。`tf.nn.conv2d_transpose` 并不支持自定义 padding 设置，只能设置为 VALID 或者 SAME。

当 `padding='VALID'` 时，输出大小为

$$o = (i-1) * s + k$$

当`padding='SAME'`时，输出大小为

$$o = (i-1) * s + 1$$

层实现

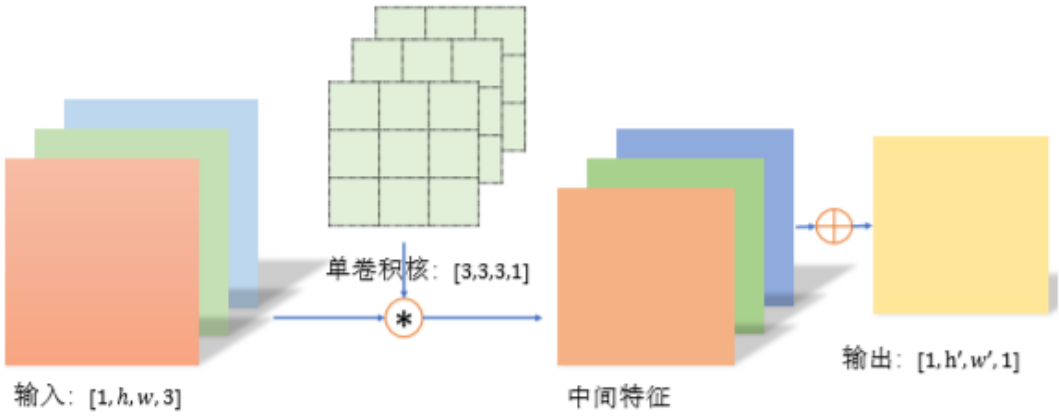
```
layer1 = layers.Conv2DTranspose(1, kernel_size=3, strides=1,
padding='VALID')
layer2 = layers.Conv2DTranspose(1, kernel_size=3, strides=1,
padding='SAME')
xx2 = layer1(out)
xx3 = layer2(out)
print(xx2)
print(xx3)
```

结果

```
Tensor("conv2d_transpose_3/BiasAdd:0", shape=(1, 4, 4, 1),
dtype=float32)
Tensor("conv2d_transpose_4/BiasAdd:0", shape=(1, 2, 2, 1),
dtype=float32)
```

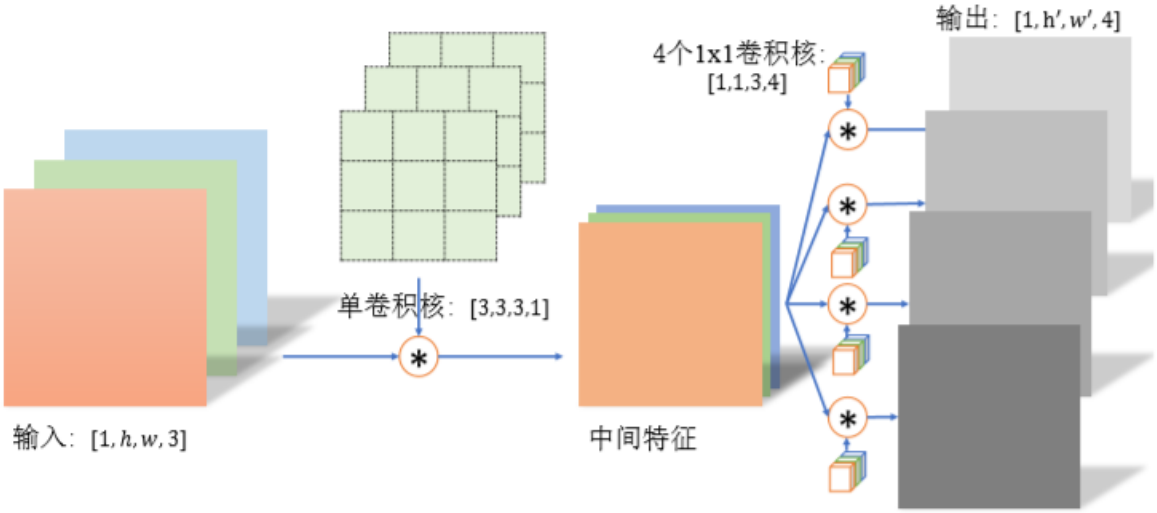
分离卷积

以深度可分离卷积 (Depth-wise Separable Convolution) 为例。普通卷积在对多通道输入进行运算时，卷积核的每个通道与输入的每个通道分别进行卷积运算，得到多通道的特征图，再对应元素相加产生单个卷积核的最终输出：



普通卷积

分离卷积则不同，卷积核的每个通道与输入的每个通道进行卷积运算，得到多个通道的中间特征。接着，我们在多通道的中间特征张量进行多个1x1卷积核的普通卷积运算，得到多个高宽不变的输出，在通道轴上拼接，形成最终的输出。如图所示：



分离卷积

采用分离卷积的输入和输出的参数量大约是普通卷积的$\frac{1}{3}$。分离卷积在 Xception 和 MobileNets 等对计算代价敏感的领域中得到了大量应用。

深度残差网络

当网络层数越深时，我们虽然更有可能获得更好的泛化能力，但整个网络会更加难以训练。这是因为梯度弥散的现象。

我们发现，浅层神经网络不容易出现梯度弥散现象，我们便因此给深度神经网络添加一种回退到浅层神经网络的机制，以提高模型性能。通过在输入和输出之间添加一条直接连接的 Skip Connection 可以让神经网络具有回退的能力。以VGG为例，若VGG13出现梯度弥散，而VGG10没有，我们就在VGG13的最后两个卷积层添加 Skip Connection。这样，网络模型可以自动选择是否经过这两个卷积层，还是直接跳过而选择 Skip Connection，或是结合这两个卷积层和 Skip Connection 的输出。

添加了 Skip Connection 的VGG13

ResNet 极好地增强了层数较大的神经网络的性能，甚至成功训练出1202层的极深层神经网络。ResNet 在 ILSVRC 2015 挑战赛 ImageNet 数据集上的分类、检测等任务上面均获得了最好性能。

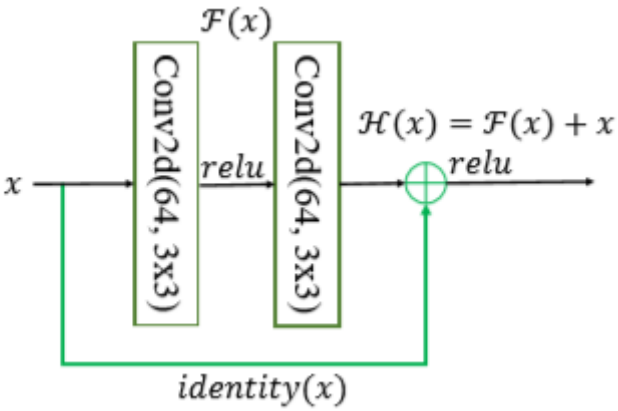
原理

ResNet 通过在卷积层的输入和输出之间添加 Skip Connection 实现层数回退机制。如下图 所示，输入x通过两个卷积层，得到特征变换后的输出F(x)，与输入x进行对应元素的相加运算，得到最终输出

$$H(x) = x + F(x)$$

H(x) 被称为残差模块（Residual Block, ResBlock）。由于被 Skip Connection 包围的卷积神经网 络需要学习映射$F(x) = H(x) - x$，故称为残差网络。

为了能够满足输入x与卷积层的输出 F(x) 能够相加运算，需要输入 x 的 shape 与 F(x) 的 shape 完全一致。当出现 shape 不一致时，一般通过在 Skip Connection 上添加额外的卷积运算环节将输入 x 变换到与 F(x) 相同的 shape，如下图中 identity(x) 函数所示，其中 identity(x) 以 1x1 的卷积运算居多，主要用于调整输入的通道数。



残差模块

ResBlock 实现

深度残差网络模型并没有增加新的网络类型，只是通过在输入和输出之间添加一条 Skip Connection，因此并没有针对 ResNet 的底层实现。在 TensorFlow 中通过调用普通卷积层即可实现残差模块。

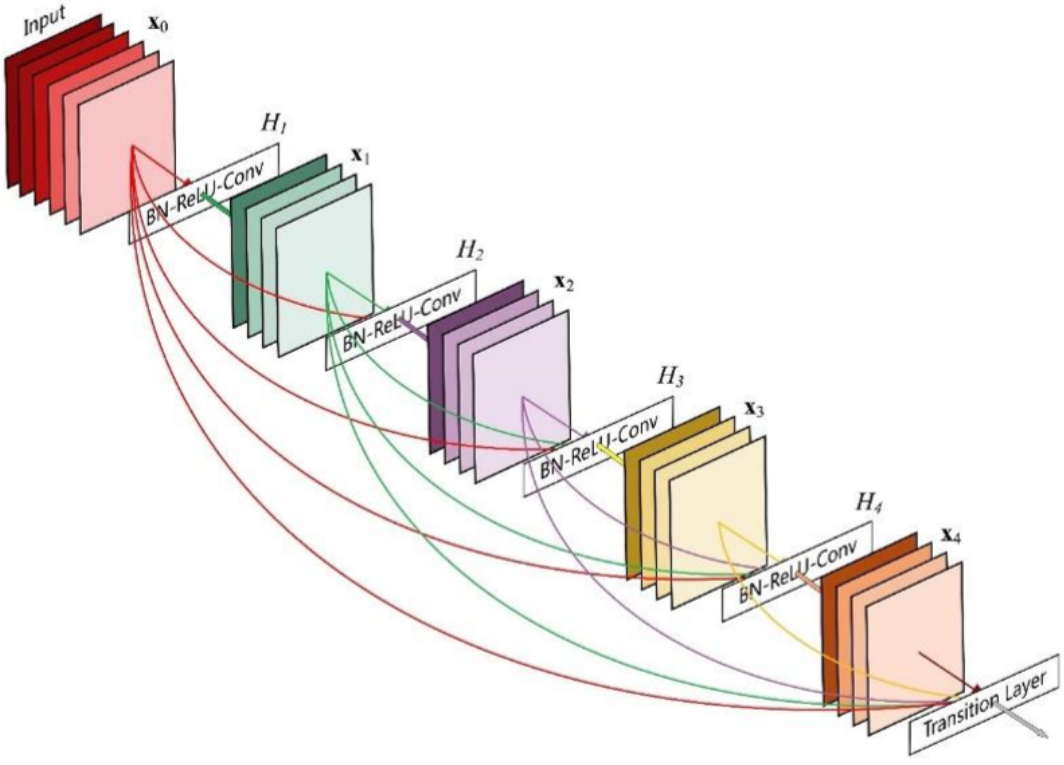
```
import tensorflow as tf
from tensorflow.keras import layers, Sequential
class BasicBlock(layers.Layer):
    # 残差模块类
    def __init__(self, filter_num, stride=1):
        super(BasicBlock, self).__init__()
        # 卷积层1
        self.conv1 = layers.Conv2D(filter_num, (3,3), strides=stride,
padding='SAME')
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')
        # 卷积层2
        self.conv2 = layers.Conv2D(filter_num, (3,3), strides=1,
padding='SAME')
        self.bn2 = layers.BatchNormalization()
        if stride != 1: # identity层
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num,
(1,1),strides=stride))
        else:
            self.downsample = lambda x:x
    def call(self, inputs, training=None):
        # 前向传播
        out = self.conv1(inputs)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
```

```
        # 通过 identity()  转换
        identity = self.downsample(inputs)
        # f(x) + x
        output = layers.add([out, identity])
        # 通过激活函数
        output = tf.nn.relu(output)
        return output
```

DenseNet

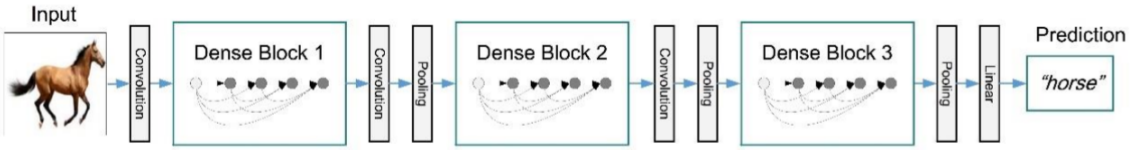
DenseNet 将前面所有层的特征图信息通过 Skip Connection 与当前层输出进行聚合，与 ResNet 的对应位置相加不同，DenseNet 采用在通道轴 c 维度进行拼接操作，聚合特征信息。

如下图所示，输入x_0通过H1卷积层得到输出x_1，x_1与x_0在通道轴上进行拼接，得到聚合后的特征张量，送入H2卷积层，得到输出x_2，同样的方法，x_2与前面所有层的特征信息: x_1与x_0进行聚合，再送入下一层。如此循环，直至最后一层的输出x_4和前面所有层的特征信息：$\{x_i\}_{i=0,1,2,3}$进行聚合得到模块的最终输出。这样一种基于 Skip Connection 稠密连接的模块叫做 Dense Block。



Dense Block

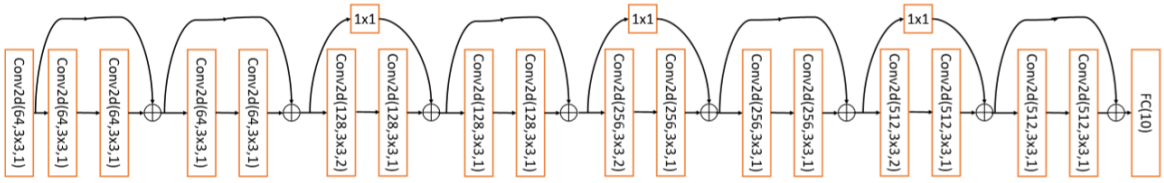
DenseNet 通过堆叠多个 Dense Block 构成复杂的深层神经网络，如图所示：



一个典型的 DenseNet 结构

CIFAR10和 ResNet18 实战

标准的 ResNet18 接受输入为 224x224 大小的图片数据，我们将 ResNet18 进行适量修整，使得它输入大小为 32x32，输出维度为 10。调整后的 ResNet18 网络结构如图所示：



调整后的 ResNet 结构

```
import tensorflow as tf
from tensorflow.keras import layers, Sequential, Model, datasets, optimizers
class BasicBlock(layers.Layer):
    # 残差模块类
    def __init__(self, filter_num, stride=1):
        super(BasicBlock, self).__init__()
        # 卷积层1
        self.conv1 = layers.Conv2D(filter_num, (3,3), strides=stride, padding='SAME')
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')
        # 卷积层2
        self.conv2 = layers.Conv2D(filter_num, (3,3), strides=1, padding='SAME')
        self.bn2 = layers.BatchNormalization()
        if stride != 1: # identity层
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num, (1,1),strides=stride))
        else:
            self.downsample = lambda x:x
    def call(self, inputs, training=None):
        # 前向传播
        out = self.conv1(inputs)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        # 通过 identity()  转换
        identity = self.downsample(inputs)
        # f(x) + x
        output = layers.add([out, identity])
        # 通过激活函数
        output = tf.nn.relu(output)
        return output
```

在设计深度卷积神经网络时，一般按照特征图高宽 h/w 逐渐减少，通道数 c 逐渐增大的经验法则。可以通过堆叠通道数逐渐增大的 Res Block 来实现高层特征的提取，通过 build_resblock 可以一次完成多个残差模块的新建。

```
def build_resblock(self, filter_num, blocks, stride=1):
    res_block = Sequential()
```

```
    # 只有第一个 BasicBlock 的步长可能不为 1，实现下采样
    res_blocks.add(BasicBlock(filter_num, stride))
    for _ in range(1, blocks): # 其他BasicBlock步长都为1
        res_blocks.add(BasicBlock(filter_num, stride=1))
        return res_blocks
class ResNet(keras.Model):
    # 通用的 ResNet 实现类
    def __init__(self, layer_dims, num_classes=10):  # [2, 2, 2, 2]
        super(ResNet, self).__init__()
        # 根网络，预处理
        self.stem = Sequential([layers.Conv2D(64, (3, 3), strides=(1, 1)),
                                layers.BatchNormalization(),
                                layers.Activation('relu'),
                                layers.MaxPool2D(pool_size=(2, 2), strides=(1, 1), padding='same')
                                ])
        # 堆叠 4 个 Block，每个 block 包含了多个 BasicBlock,设置步长不一样
        self.layer1 = self.build_resblock(64,  layer_dims[0])
        self.layer2 = self.build_resblock(128, layer_dims[1], stride=2)
        self.layer3 = self.build_resblock(256, layer_dims[2], stride=2)
        self.layer4 = self.build_resblock(512, layer_dims[3], stride=2)
        # 通过 Pooling 层将高宽降低为 1x1
        self.avgpool = layers.GlobalAveragePooling2D()
        # 最后连接一个全连接层分类
        self.fc = layers.Dense(num_classes)

    def call(self, inputs, training=None):
        # 通过根网络
        x = self.stem(inputs)
        # 一次通过 4 个模块
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        # 通过池化层
        x = self.avgpool(x)
        # 通过全连接层
        x = self.fc(x)
        return x
```

通过调整每个 Res Block 的堆叠数量和通道数可以产生不同的 ResNet，如通过 64-64-128-128-256-256-512-512 通道数配置，共 8 个 Res Block，可得到 ResNet18 的网络模型。每个 ResBlock 包含了 2 个主要的卷积层，因此主要卷积层数量是 8*2=16，加上网络最前面的普通卷积层和全连接层，共 18 层。

```
def resnet18():
    # 通过调整模块内部 BasicBlock 的数量和配置实现不同的 ResNet
    return ResNet([2, 2, 2, 2])

def resnet34():
     # 通过调整模块内部 BasicBlock 的数量和配置实现不同的 ResNet
    return ResNet([3, 4, 6, 3])

(x, y), (x_test, y_test) = datasets.cifar10.load_data()  # 加载数据集
y = tf.squeeze(y, axis=1)  # 删除不必要的维度
y_test = tf.squeeze(y_test, axis=1)  # 删除不必要的维度
train_db = tf.data.Dataset.from_tensor_slices((x, y))  # 构建训练集 #
随机打散，预处理，批量化
train_db = train_db.shuffle(1000).map(preprocess).batch(512)
test_db = tf.data.Dataset.from_tensor_slices(
    (x_test, y_test))  # 构建测试集 # 随机打散，预处理，批量化
test_db = test_db.map(preprocess).batch(512)
optimizer = optimizers.adam(0.01)


def preprocess(x, y):
    # 将数据映射到-1~1     x = 2*tf.cast(x, dtype=tf.float32) / 255. -
1
    y = tf.cast(y, dtype=tf.int32)  # 类型转换
    return x, y


# 网络训练
for epoch in range(50):  # 训练 epoch
    for step, (x, y) in enumerate(train_db):
        with tf.GradientTape() as tape:
            # [b, 32, 32, 3] => [b, 10],前向传播
            logits = model(x)
            # [b] => [b, 10],one-hot 编码
            y_onehot = tf.one_hot(y, depth=10)
            # 计算交叉熵
            loss = tf.losses.categorical_crossentropy(
                y_onehot, logits, from_logits=True)
            loss = tf.reduce_mean(loss)
            # 计算梯度信息
            grads = tape.gradient(loss, model.trainable_variables)
            # 更新网络参数
            optimizer.apply_gradients(zip(grads,
model.trainable_variables))
```

ResNet18 的网络参数量共 11M 个，经过 50 个 epoch 后，网络的准确率达到了 79.3%。我们这里的实战代码比较精简，在精挑超参数、数据增强等手段加持下，准确率可以达到更高。