# Tensorflow一些常用基本概念与函数（1）

### 1、tensorflow的基本运作

为了快速的熟悉TensorFlow编程，下面从一段简单的代码开始：

import tensorflow tf

#定义‘符号’变量，也称为占位符

a = tf.placeholder("float")

b = tf.placeholder("float")

y = tf.mul(a, b) #构造一个op节点

sess = tf.Session()#建立会话

#运行会话，输入数据，并计算节点，同时打印结果

print sess.run(y, feed\_dict={a: , b: })

# 任务完成, 关闭会话.

sess.close()

其中tf.mul(a, b)函数便是tf的一个基本的算数运算，接下来介绍跟多的相关函数。

### 2、tf函数

TensorFlow 将图形定义转换成分布式执行的操作, 以充分利用可用的计算资源(如 CPU 或 GPU。一般你不需要显式指定使用 CPU 还是 GPU, TensorFlow 能自动检测。如果检测到 GPU, TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作.

并行计算能让代价大的算法计算加速执行，TensorFlow也在实现上对复杂操作进行了有效的改进。大部分核相关的操作都是设备相关的实现，比如GPU。下面是一些重要的操作/核：

| **操作组** | **操作** |
| --- | --- |
| Maths | Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal |
| Array | Concat, Slice, Split, Constant, Rank, Shape, Shuffle |
| Matrix | MatMul, MatrixInverse, MatrixDeterminant |
| Neuronal Network | SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool |
| Checkpointing | Save, Restore |
| Queues and syncronizations | Enqueue, Dequeue, MutexAcquire, MutexRelease |
| Flow control | Merge, Switch, Enter, Leave, NextIteration |

#### TensorFlow的算术操作如下：

| **操作** | **描述** |
| --- | --- |
| tf.add(x, y, name=None) | 求和 |
| tf.sub(x, y, name=None) | 减法 |
| tf.mul(x, y, name=None) | 乘法 |
| tf.div(x, y, name=None) | 除法 |
| tf.mod(x, y, name=None) | 取模 |
| tf.abs(x, name=None) | 求绝对值 |
| tf.neg(x, name=None) | 取负 (y = -x). |
| tf.sign(x, name=None) | 返回符号 y = sign(x) = -1 if x < 0; 0 if x == 0; 1 if x > 0. |
| tf.inv(x, name=None) | 取反 |
| tf.square(x, name=None) | 计算平方 (y = x \* x = x^2). |
| tf.round(x, name=None) | 舍入最接近的整数 # ‘a’ is [0.9, 2.5, 2.3, -4.4] tf.round(a) ==> [ 1.0, 3.0, 2.0, -4.0 ] |
| tf.sqrt(x, name=None) | 开根号 (y = \sqrt{x} = x^{1/2}). |
| tf.pow(x, y, name=None) | 幂次方  # tensor ‘x’ is [[2, 2], [3, 3]] # tensor ‘y’ is [[8, 16], [2, 3]] tf.pow(x, y) ==> [[256, 65536], [9, 27]] |
| tf.exp(x, name=None) | 计算e的次方 |
| tf.log(x, name=None) | 计算log，一个输入计算e的ln，两输入以第二输入为底 |
| tf.maximum(x, y, name=None) | 返回最大值 (x > y ? x : y) |
| tf.minimum(x, y, name=None) | 返回最小值 (x < y ? x : y) |
| tf.cos(x, name=None) | 三角函数cosine |
| tf.sin(x, name=None) | 三角函数sine |
| tf.tan(x, name=None) | 三角函数tan |
| tf.atan(x, name=None) | 三角函数ctan |

#### 张量操作Tensor Transformations

* 数据类型转换Casting

| **操作** | **描述** |
| --- | --- |
| tf.string\_to\_number (string\_tensor, out\_type=None, name=None) | 字符串转为数字 |
| tf.to\_double(x, name=’ToDouble’) | 转为64位浮点类型–float64 |
| tf.to\_float(x, name=’ToFloat’) | 转为32位浮点类型–float32 |
| tf.to\_int32(x, name=’ToInt32’) | 转为32位整型–int32 |
| tf.to\_int64(x, name=’ToInt64’) | 转为64位整型–int64 |
| tf.cast(x, dtype, name=None) | 将x或者x.values转换为dtype # tensor a is [1.8, 2.2], dtype=tf.float tf.cast(a, tf.int32) ==> [1, 2] # dtype=tf.int32 |

* 形状操作Shapes and Shaping

| **操作** | **描述** |
| --- | --- |
| tf.shape(input, name=None) | 返回数据的shape # ‘t’ is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] shape(t) ==> [2, 2, 3] |
| tf.size(input, name=None) | 返回数据的元素数量 # ‘t’ is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]] size(t) ==> 12 |
| tf.rank(input, name=None) | 返回tensor的rank 注意：此rank不同于矩阵的rank， tensor的rank表示一个tensor需要的索引数目来唯一表示任何一个元素 也就是通常所说的 “order”, “degree”或”ndims” #’t’ is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] # shape of tensor ‘t’ is [2, 2, 3] rank(t) ==> 3 |
| tf.reshape(tensor, shape, name=None) | 改变tensor的形状 # tensor ‘t’ is [1, 2, 3, 4, 5, 6, 7, 8, 9] # tensor ‘t’ has shape [9] reshape(t, [3, 3]) ==>  [[1, 2, 3], [4, 5, 6], [7, 8, 9]] #如果shape有元素[-1],表示在该维度打平至一维 # -1 将自动推导得为 9: reshape(t, [2, -1]) ==>  [[1, 1, 1, 2, 2, 2, 3, 3, 3], [4, 4, 4, 5, 5, 5, 6, 6, 6]] |
| tf.expand\_dims(input, dim, name=None) | 插入维度1进入一个tensor中 #该操作要求-1-input.dims() # ‘t’ is a tensor of shape [2] shape(expand\_dims(t, 0)) ==> [1, 2] shape(expand\_dims(t, 1)) ==> [2, 1] shape(expand\_dims(t, -1)) ==> [2, 1] <= dim <= input.dims() |

* 切片与合并（Slicing and Joining）

| **操作** | **描述** |
| --- | --- |
| tf.slice(input\_, begin, size, name=None) | 对tensor进行切片操作 其中size[i] = input.dim\_size(i) - begin[i] 该操作要求 0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n] #’input’ is  #[[[1, 1, 1], [2, 2, 2]],[[3, 3, 3], [4, 4, 4]],[[5, 5, 5], [6, 6, 6]]] tf.slice(input, [1, 0, 0], [1, 1, 3]) ==> [[[3, 3, 3]]] tf.slice(input, [1, 0, 0], [1, 2, 3]) ==>  [[[3, 3, 3], [4, 4, 4]]] tf.slice(input, [1, 0, 0], [2, 1, 3]) ==>  [[[3, 3, 3]], [[5, 5, 5]]] |
| tf.split(split\_dim, num\_split, value, name=’split’) | 沿着某一维度将tensor分离为num\_split tensors # ‘value’ is a tensor with shape [5, 30] # Split ‘value’ into 3 tensors along dimension 1 split0, split1, split2 = tf.split(1, 3, value) tf.shape(split0) ==> [5, 10] |
| tf.concat(concat\_dim, values, name=’concat’) | 沿着某一维度连结tensor t1 = [[1, 2, 3], [4, 5, 6]] t2 = [[7, 8, 9], [10, 11, 12]] tf.concat(0, [t1, t2]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]] tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]] 如果想沿着tensor一新轴连结打包,那么可以： tf.concat(axis, [tf.expand\_dims(t, axis) for t in tensors]) 等同于tf.pack(tensors, axis=axis) |
| tf.pack(values, axis=0, name=’pack’) | 将一系列rank-R的tensor打包为一个rank-(R+1)的tensor # ‘x’ is [1, 4], ‘y’ is [2, 5], ‘z’ is [3, 6] pack([x, y, z]) => [[1, 4], [2, 5], [3, 6]]  # 沿着第一维pack pack([x, y, z], axis=1) => [[1, 2, 3], [4, 5, 6]] 等价于tf.pack([x, y, z]) = np.asarray([x, y, z]) |
| tf.reverse(tensor, dims, name=None) | 沿着某维度进行序列反转 其中dim为列表，元素为bool型，size等于rank(tensor) # tensor ‘t’ is  [[[[ 0, 1, 2, 3], #[ 4, 5, 6, 7],  #[ 8, 9, 10, 11]], #[[12, 13, 14, 15], #[16, 17, 18, 19], #[20, 21, 22, 23]]]] # tensor ‘t’ shape is [1, 2, 3, 4] # ‘dims’ is [False, False, False, True] reverse(t, dims) ==> [[[[ 3, 2, 1, 0], [ 7, 6, 5, 4], [ 11, 10, 9, 8]], [[15, 14, 13, 12], [19, 18, 17, 16], [23, 22, 21, 20]]]] |
| tf.transpose(a, perm=None, name=’transpose’) | 调换tensor的维度顺序 按照列表perm的维度排列调换tensor顺序， 如为定义，则perm为(n-1…0) # ‘x’ is [[1 2 3],[4 5 6]] tf.transpose(x) ==> [[1 4], [2 5],[3 6]] # Equivalently tf.transpose(x, perm=[1, 0]) ==> [[1 4],[2 5], [3 6]] |
| tf.gather(params, indices, validate\_indices=None, name=None) | 合并索引indices所指示params中的切片 tf.gather |
| tf.one\_hot (indices, depth, on\_value=None, off\_value=None,  axis=None, dtype=None, name=None) | indices = [0, 2, -1, 1] depth = 3 on\_value = 5.0  off\_value = 0.0  axis = -1  #Then output is [4 x 3]:  output =  [5.0 0.0 0.0] // one\_hot(0)  [0.0 0.0 5.0] // one\_hot(2)  [0.0 0.0 0.0] // one\_hot(-1)  [0.0 5.0 0.0] // one\_hot(1) |

#### 矩阵相关运算

| **操作** | **描述** |
| --- | --- |
| tf.diag(diagonal, name=None) | 返回一个给定对角值的对角tensor # ‘diagonal’ is [1, 2, 3, 4] tf.diag(diagonal) ==>  [[1, 0, 0, 0] [0, 2, 0, 0] [0, 0, 3, 0] [0, 0, 0, 4]] |
| tf.diag\_part(input, name=None) | 功能与上面相反 |
| tf.trace(x, name=None) | 求一个2维tensor足迹，即对角值diagonal之和 |
| tf.transpose(a, perm=None, name=’transpose’) | 调换tensor的维度顺序 按照列表perm的维度排列调换tensor顺序， 如为定义，则perm为(n-1…0) # ‘x’ is [[1 2 3],[4 5 6]] tf.transpose(x) ==> [[1 4], [2 5],[3 6]] # Equivalently tf.transpose(x, perm=[1, 0]) ==> [[1 4],[2 5], [3 6]] |
| tf.matmul(a, b, transpose\_a=False,  transpose\_b=False, a\_is\_sparse=False,  b\_is\_sparse=False, name=None) | 矩阵相乘 |
| tf.matrix\_determinant(input, name=None) | 返回方阵的行列式 |
| tf.matrix\_inverse(input, adjoint=None, name=None) | 求方阵的逆矩阵，adjoint为True时，计算输入共轭矩阵的逆矩阵 |
| tf.cholesky(input, name=None) | 对输入方阵cholesky分解， 即把一个对称正定的矩阵表示成一个下三角矩阵L和其转置的乘积的分解A=LL^T |
| tf.matrix\_solve(matrix, rhs, adjoint=None, name=None) | 求解tf.matrix\_solve(matrix, rhs, adjoint=None, name=None) matrix为方阵shape为[M,M],rhs的shape为[M,K]，output为[M,K] |

| **操作** | **描述** |
| --- | --- |
| tf.complex(real, imag, name=None) | 将两实数转换为复数形式 # tensor ‘real’ is [2.25, 3.25] # tensor imag is [4.75, 5.75] tf.complex(real, imag) ==> [[2.25 + 4.75j], [3.25 + 5.75j]] |
| tf.complex\_abs(x, name=None) | 计算复数的绝对值，即长度。 # tensor ‘x’ is [[-2.25 + 4.75j], [-3.25 + 5.75j]] tf.complex\_abs(x) ==> [5.25594902, 6.60492229] |
| tf.conj(input, name=None) | 计算共轭复数 |
| tf.imag(input, name=None) tf.real(input, name=None) | 提取复数的虚部和实部 |
| tf.fft(input, name=None) | 计算一维的离散傅里叶变换，输入数据类型为complex64 |

#### 归约计算(Reduction)

| **操作** | **描述** |
| --- | --- |
| tf.reduce\_sum(input\_tensor, reduction\_indices=None,  keep\_dims=False, name=None) | 计算输入tensor元素的和，或者安照reduction\_indices指定的轴进行求和 # ‘x’ is [[1, 1, 1] # [1, 1, 1]] tf.reduce\_sum(x) ==> 6 tf.reduce\_sum(x, 0) ==> [2, 2, 2] tf.reduce\_sum(x, 1) ==> [3, 3] tf.reduce\_sum(x, 1, keep\_dims=True) ==> [[3], [3]] tf.reduce\_sum(x, [0, 1]) ==> 6 |
| tf.reduce\_prod(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 计算输入tensor元素的乘积，或者安照reduction\_indices指定的轴进行求乘积 |
| tf.reduce\_min(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 求tensor中最小值 |
| tf.reduce\_max(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 求tensor中最大值 |
| tf.reduce\_mean(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 求tensor中平均值 |
| tf.reduce\_all(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 对tensor中各个元素求逻辑’与’ # ‘x’ is  # [[True, True] # [False, False]] tf.reduce\_all(x) ==> False tf.reduce\_all(x, 0) ==> [False, False] tf.reduce\_all(x, 1) ==> [True, False] |
| tf.reduce\_any(input\_tensor,  reduction\_indices=None,  keep\_dims=False, name=None) | 对tensor中各个元素求逻辑’或’ |
| tf.accumulate\_n(inputs, shape=None,  tensor\_dtype=None, name=None) | 计算一系列tensor的和 # tensor ‘a’ is [[1, 2], [3, 4]] # tensor b is [[5, 0], [0, 6]] tf.accumulate\_n([a, b, a]) ==> [[7, 4], [6, 14]] |
| tf.cumsum(x, axis=0, exclusive=False,  reverse=False, name=None) | 求累积和 tf.cumsum([a, b, c]) ==> [a, a + b, a + b + c] tf.cumsum([a, b, c], exclusive=True) ==> [0, a, a + b] tf.cumsum([a, b, c], reverse=True) ==> [a + b + c, b + c, c] tf.cumsum([a, b, c], exclusive=True, reverse=True) ==> [b + c, c, 0] |

#### 分割(Segmentation)

| **操作** | **描述** |
| --- | --- |
| tf.segment\_sum(data, segment\_ids, name=None) | 根据segment\_ids的分段计算各个片段的和 其中segment\_ids为一个size与data第一维相同的tensor 其中id为int型数据，最大id不大于size c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]]) tf.segment\_sum(c, tf.constant([0, 0, 1])) ==>[[0 0 0 0]  [5 6 7 8]] 上面例子分为[0,1]两id,对相同id的data相应数据进行求和, 并放入结果的相应id中， 且segment\_ids只升不降 |
| tf.segment\_prod(data, segment\_ids, name=None) | 根据segment\_ids的分段计算各个片段的积 |
| tf.segment\_min(data, segment\_ids, name=None) | 根据segment\_ids的分段计算各个片段的最小值 |
| tf.segment\_max(data, segment\_ids, name=None) | 根据segment\_ids的分段计算各个片段的最大值 |
| tf.segment\_mean(data, segment\_ids, name=None) | 根据segment\_ids的分段计算各个片段的平均值 |
| tf.unsorted\_segment\_sum(data, segment\_ids, num\_segments, name=None) | 与tf.segment\_sum函数类似， 不同在于segment\_ids中id顺序可以是无序的 |
| tf.sparse\_segment\_sum(data, indices,  segment\_ids, name=None) | 输入进行稀疏分割求和 c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]]) # Select two rows, one segment. tf.sparse\_segment\_sum(c, tf.constant([0, 1]), tf.constant([0, 0]))  ==> [[0 0 0 0]] 对原data的indices为[0,1]位置的进行分割， 并按照segment\_ids的分组进行求和 |

#### 序列比较与索引提取(Sequence Comparison and Indexing)

| **操作** | **描述** |
| --- | --- |
| tf.argmin(input, dimension, name=None) | 返回input最小值的索引index |
| tf.argmax(input, dimension, name=None) | 返回input最大值的索引index |
| tf.listdiff(x, y, name=None) | 返回x，y中不同值的索引 |
| tf.where(input, name=None) | 返回bool型tensor中为True的位置 # ‘input’ tensor is  #[[True, False] #[True, False]] # ‘input’ 有两个’True’,那么输出两个坐标值. # ‘input’的rank为2, 所以每个坐标为具有两个维度. where(input) ==> [[0, 0], [1, 0]] |
| tf.unique(x, name=None) | 返回一个元组tuple(y,idx)，y为x的列表的唯一化数据列表， idx为x数据对应y元素的index # tensor ‘x’ is [1, 1, 2, 4, 4, 4, 7, 8, 8] y, idx = unique(x) y ==> [1, 2, 4, 7, 8] idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4] |
| tf.invert\_permutation(x, name=None) | 置换x数据与索引的关系 # tensor x is [3, 4, 0, 2, 1] invert\_permutation(x) ==> [2, 4, 3, 0, 1] |

#### 神经网络(Neural Network)

* 激活函数（Activation Functions）

| **操作** | **描述** |
| --- | --- |
| tf.nn.relu(features, name=None) | 整流函数：max(features, 0) |
| tf.nn.relu6(features, name=None) | 以6为阈值的整流函数：min(max(features, 0), 6) |
| tf.nn.elu(features, name=None) | elu函数，exp(features) - 1 if < 0,否则features [Exponential Linear Units (ELUs)](http://arxiv.org/abs/1511.07289) |
| tf.nn.softplus(features, name=None) | 计算softplus：log(exp(features) + 1) |
| tf.nn.dropout(x, keep\_prob,  noise\_shape=None, seed=None, name=None) | 计算dropout，keep\_prob为keep概率 noise\_shape为噪声的shape |
| tf.nn.bias\_add(value, bias, data\_format=None, name=None) | 对value加一偏置量 此函数为tf.add的特殊情况，bias仅为一维， 函数通过广播机制进行与value求和, 数据格式可以与value不同，返回为与value相同格式 |
| tf.sigmoid(x, name=None) | y = 1 / (1 + exp(-x)) |
| tf.tanh(x, name=None) | 双曲线切线激活函数 |

* 卷积函数（Convolution）

| **操作** | **描述** |
| --- | --- |
| tf.nn.conv2d(input, filter, strides, padding,  use\_cudnn\_on\_gpu=None, data\_format=None, name=None) | 在给定的4D input与 filter下计算2D卷积 输入shape为 [batch, height, width, in\_channels] |
| tf.nn.conv3d(input, filter, strides, padding, name=None) | 在给定的5D input与 filter下计算3D卷积 输入shape为[batch, in\_depth, in\_height, in\_width, in\_channels] |

* 池化函数（Pooling）

| **操作** | **描述** |
| --- | --- |
| tf.nn.avg\_pool(value, ksize, strides, padding,  data\_format=’NHWC’, name=None) | 平均方式池化 |
| tf.nn.max\_pool(value, ksize, strides, padding,  data\_format=’NHWC’, name=None) | 最大值方法池化 |
| tf.nn.max\_pool\_with\_argmax(input, ksize, strides, padding, Targmax=None, name=None) | 返回一个二维元组(output,argmax),最大值pooling，返回最大值及其相应的索引 |
| tf.nn.avg\_pool3d(input, ksize, strides,  padding, name=None) | 3D平均值pooling |
| tf.nn.max\_pool3d(input, ksize, strides,  padding, name=None) | 3D最大值pooling |

* 数据标准化（Normalization）

| **操作** | **描述** |
| --- | --- |
| tf.nn.l2\_normalize(x, dim, epsilon=1e-12, name=None) | 对维度dim进行L2范式标准化 output = x / sqrt(max(sum(x\*\*2), epsilon)) |
| tf.nn.sufficient\_statistics(x, axes, shift=None,  keep\_dims=False, name=None) | 计算与均值和方差有关的完全统计量 返回4维元组,\*元素个数，\*元素总和，\*元素的平方和，\*shift结果 [参见算法介绍](https://www.google.com/url?q=https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Computing_shifted_data&usg=AFQjCNG5RoY7Xvpv4xg-Wy-UJvAPh2zDQw) |
| tf.nn.normalize\_moments(counts, mean\_ss, variance\_ss, shift, name=None) | 基于完全统计量计算均值和方差 |
| tf.nn.moments(x, axes, shift=None,  name=None, keep\_dims=False) | 直接计算均值与方差 |

* 损失函数（Losses）

| **操作** | **描述** |
| --- | --- |
| tf.nn.l2\_loss(t, name=None) | output = sum(t \*\* 2) / 2 |

* 分类函数（Classification）

| **操作** | **描述** |
| --- | --- |
| tf.nn.sigmoid\_cross\_entropy\_with\_logits (logits, targets, name=None)\* | 计算输入logits, targets的交叉熵 |
| tf.nn.softmax(logits, name=None) | 计算softmax softmax[i, j] = exp(logits[i, j]) / sum\_j(exp(logits[i, j])) |
| tf.nn.log\_softmax(logits, name=None) | logsoftmax[i, j] = logits[i, j] - log(sum(exp(logits[i]))) |
| tf.nn.softmax\_cross\_entropy\_with\_logits (logits, labels, name=None) | 计算logits和labels的softmax交叉熵 logits, labels必须为相同的shape与数据类型 |
| tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits (logits, labels, name=None) | 计算logits和labels的softmax交叉熵 |
| tf.nn.weighted\_cross\_entropy\_with\_logits (logits, targets, pos\_weight, name=None) | 与sigmoid\_cross\_entropy\_with\_logits()相似， 但给正向样本损失加了权重pos\_weight |

* 符号嵌入（Embeddings）

| **操作** | **描述** |
| --- | --- |
| tf.nn.embedding\_lookup (params, ids, partition\_strategy=’mod’,  name=None, validate\_indices=True) | 根据索引ids查询embedding列表params中的tensor值 如果len(params) > 1，id将会安照partition\_strategy策略进行分割 1、如果partition\_strategy为”mod”， id所分配到的位置为p = id % len(params) 比如有13个ids，分为5个位置，那么分配方案为： [[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8], [4, 9]] 2、如果partition\_strategy为”div”,那么分配方案为： [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10], [11, 12]] |
| tf.nn.embedding\_lookup\_sparse(params, sp\_ids, sp\_weights, partition\_strategy=’mod’,  name=None, combiner=’mean’) | 对给定的ids和权重查询embedding 1、sp\_ids为一个N x M的稀疏tensor， N为batch大小，M为任意，数据类型int64 2、sp\_weights的shape与sp\_ids的稀疏tensor权重， 浮点类型，若为None，则权重为全’1’ |

* 循环神经网络（Recurrent Neural Networks）

| **操作** | **描述** |
| --- | --- |
| tf.nn.rnn(cell, inputs, initial\_state=None, dtype=None,  sequence\_length=None, scope=None) | 基于RNNCell类的实例cell建立循环神经网络 |
| tf.nn.dynamic\_rnn(cell, inputs, sequence\_length=None,  initial\_state=None, dtype=None, parallel\_iterations=None,  swap\_memory=False, time\_major=False, scope=None) | 基于RNNCell类的实例cell建立动态循环神经网络 与一般rnn不同的是，该函数会根据输入动态展开 返回(outputs,state) |
| tf.nn.state\_saving\_rnn(cell, inputs, state\_saver, state\_name,  sequence\_length=None, scope=None) | 可储存调试状态的RNN网络 |
| tf.nn.bidirectional\_rnn(cell\_fw, cell\_bw, inputs,  initial\_state\_fw=None, initial\_state\_bw=None, dtype=None, sequence\_length=None, scope=None) | 双向RNN, 返回一个3元组tuple (outputs, output\_state\_fw, output\_state\_bw) |

*—***tf.nn.rnn简要介绍***—   
cell: 一个RNNCell实例   
inputs: 一个shape为[batch\_size, input\_size]的tensor   
initial\_state: 为RNN的state设定初值，可选   
sequence\_length：制定输入的每一个序列的长度，size为[batch\_size],值范围为[0, T)的int型数据   
其中T为输入数据序列的长度   
@   
@针对输入batch中序列长度不同，所设置的动态计算机制   
@对于在时间t，和batch的b行，有   
(output, state)(b, t) = ? (zeros(cell.output\_size), states(b, sequence\_length(b) - 1)) : cell(input(b, t), state(b, t - 1))*

* 求值网络（Evaluation）

| **操作** | **描述** |
| --- | --- |
| tf.nn.top\_k(input, k=1, sorted=True, name=None) | 返回前k大的值及其对应的索引 |
| tf.nn.in\_top\_k(predictions, targets, k, name=None) | 返回判断是否targets索引的predictions相应的值 是否在在predictions前k个位置中， 返回数据类型为bool类型，len与predictions同 |

* [监督候选采样网络（Candidate Sampling）](https://www.tensorflow.org/versions/r0.10/extras/candidate_sampling.pdf)

对于有巨大量的多分类与多标签模型，如果使用全连接softmax将会占用大量的时间与空间资源，所以采用候选采样方法仅使用一小部分类别与标签作为监督以加速训练。

| **操作** | **描述** |
| --- | --- |
| **Sampled Loss Functions** |  |
| tf.nn.nce\_loss(weights, biases, inputs, labels, num\_sampled, num\_classes, num\_true=1, sampled\_values=None, remove\_accidental\_hits=False, partition\_strategy=’mod’, name=’nce\_loss’) | 返回noise-contrastive的训练损失结果 |
| tf.nn.sampled\_softmax\_loss(weights, biases, inputs, labels,  num\_sampled, num\_classes, num\_true=1, sampled\_values=None, remove\_accidental\_hits=True, partition\_strategy=’mod’,  name=’sampled\_softmax\_loss’) | 返回sampled softmax的训练损失 [参考- Jean et al., 2014第3部分](http://arxiv.org/pdf/1412.2007.pdf) |
| **Candidate Samplers** |  |
| tf.nn.uniform\_candidate\_sampler(true\_classes, num\_true,  num\_sampled, unique, range\_max, seed=None, name=None) | 通过均匀分布的采样集合 返回三元tuple 1、sampled\_candidates 候选集合。 2、期望的true\_classes个数，为浮点值 3、期望的sampled\_candidates个数，为浮点值 |
| tf.nn.log\_uniform\_candidate\_sampler(true\_classes, num\_true, num\_sampled, unique, range\_max, seed=None, name=None) | 通过log均匀分布的采样集合，返回三元tuple |
| tf.nn.learned\_unigram\_candidate\_sampler (true\_classes, num\_true, num\_sampled, unique,  range\_max, seed=None, name=None) | 根据在训练过程中学习到的分布状况进行采样 返回三元tuple |
| tf.nn.fixed\_unigram\_candidate\_sampler(true\_classes, num\_true, num\_sampled, unique, range\_max, vocab\_file=”,  distortion=1.0, num\_reserved\_ids=0, num\_shards=1, shard=0, unigrams=(), seed=None, name=None) | 基于所提供的基本分布进行采样 |

# 保存与恢复变量

| **操作** | **描述** |
| --- | --- |
| 类tf.train.Saver(Saving and Restoring Variables) |  |
| tf.train.Saver.\_\_init\_\_(var\_list=None, reshape=False,  sharded=False, max\_to\_keep=5,  keep\_checkpoint\_every\_n\_hours=10000.0,  name=None, restore\_sequentially=False, saver\_def=None, builder=None) | 创建一个存储器Saver var\_list定义需要存储和恢复的变量 |
| tf.train.Saver.save(sess, save\_path, global\_step=None,  latest\_filename=None, meta\_graph\_suffix=’meta’, write\_meta\_graph=True) | 保存变量 |
| tf.train.Saver.restore(sess, save\_path) | 恢复变量 |
| tf.train.Saver.last\_checkpoints | 列出最近未删除的checkpoint 文件名 |
| tf.train.Saver.set\_last\_checkpoints(last\_checkpoints) | 设置checkpoint文件名列表 |
| tf.train.Saver.set\_last\_checkpoints\_with\_time(last\_checkpoints\_with\_time) | 设置checkpoint文件名列表和时间戳 |

**相关链接：**

[1] 安装Tensorflow（[Linux](http://lib.csdn.net/base/linux) ubuntu）<http://blog.csdn.net/lenbow/article/details/51203526>   
[2] ubuntu下CUDA编译的GCC降级安装<http://blog.csdn.net/lenbow/article/details/51596706>   
[3] ubuntu手动安装最新Nvidia显卡驱动<http://blog.csdn.net/lenbow/article/details/51683783>   
[4] Tensorflow的CUDA升级，以及相关配置<http://blog.csdn.net/lenbow/article/details/52118116>   
[5] 基于gensim的Doc2Vec简析<http://blog.csdn.net/lenbow/article/details/52120230>   
[6] TensorFlow的分布式学习框架简介<http://blog.csdn.net/lenbow/article/details/52130565>

Measure

Measure

免费注册印象笔记帐户以保存文章，以后随时在手机、平板或电脑上阅读。

[创建帐户](chrome-extension://iooicodkiihhpojmmeghjclgihfjdjhj/front/in_isolation/reformat.html)

# Tensorflow一些常用基本概念与函数（2）

版权声明：本文为博主原创文章，未经博主允许不得转载。

摘要：本文主要对tf的一些常用概念与方法进行描述。为‘Tensorflow一些常用基本概念与函数’系列之二。

## 1、tensorflow的基本运作

为了快速的熟悉TensorFlow编程，下面从一段简单的代码开始：

import tensorflow tf

#定义‘符号’变量，也称为占位符

a = tf.placeholder("float")

b = tf.placeholder("float")

y = tf.mul(a, b) #构造一个op节点

sess = tf.Session()#建立会话

#运行会话，输入数据，并计算节点，同时打印结果

print sess.run(y, feed\_dict={a: , b: })

# 任务完成, 关闭会话.

sess.close()

其中tf.mul(a, b)函数便是tf的一个基本的算数运算，接下来介绍跟多的相关函数。

## 2、tf函数

TensorFlow 将图形定义转换成分布式执行的操作, 以充分利用可用的计算资源(如 CPU 或 GPU。一般你不需要显式指定使用 CPU 还是 GPU, TensorFlow 能自动检测。如果检测到 GPU, TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作.

并行计算能让代价大的算法计算加速执行，TensorFlow也在实现上对复杂操作进行了有效的改进。大部分核相关的操作都是设备相关的实现，比如GPU。本文主要涉及的相关概念或操作有以下内容：

| **操作组** | **操作** |
| --- | --- |
| Building Graphs | Core graph data structures，Tensor types，Utility functions |
| Inputs and Readers | Placeholders，Readers，Converting，Queues，Input pipeline |

### ****2.1 建立图(Building Graphs)****

本节主要介绍建立tensorflow图的相关类或函数

#### \* ****核心图的数据结构（Core graph data structures）****

**tf.Graph**

| **操作** | **描述** |
| --- | --- |
| class tf.Graph | tensorflow中的计算以图数据流的方式表示 一个图包含一系列表示计算单元的操作对象 以及在图中流动的数据单元以tensor对象表现 |
| tf.Graph.\_\_init\_\_() | 建立一个空图 |
| tf.Graph.as\_default() | 一个将某图设置为默认图，并返回一个上下文管理器 如果不显式添加一个默认图，系统会自动设置一个全局的默认图。 所设置的默认图，在模块范围内所定义的节点都将默认加入默认图中 |
| tf.Graph.as\_graph\_def (from\_version=None, add\_shapes=False) | 返回一个图的序列化的GraphDef表示 序列化的[GraphDef](https://github.com/tensorflow/tensorflow/blob/r0.10/tensorflow/core/framework/graph.proto)可以导入至另一个图中(使用 import\_graph\_def()) 或者使用C++ Session API |
| tf.Graph.finalize() | 完成图的构建，即将其设置为只读模式 |
| tf.Graph.finalized | 返回True，如果图被完成 |
| tf.Graph.control\_dependencies(control\_inputs) | 定义一个控制依赖，并返回一个上下文管理器 with g.control\_dependencies([a, b, c]): # `d` 和 `e` 将在 `a`, `b`, 和`c`执行完之后运行. d = … e = … |
| tf.Graph.device(device\_name\_or\_function) | 定义运行图所使用的设备，并返回一个上下文管理器 with g.device('/gpu:0'): ... with g.device('/cpu:0'): ... |
| tf.Graph.name\_scope(name) | 为节点创建层次化的名称，并返回一个上下文管理器 |
| tf.Graph.add\_to\_collection(name, value) | 将value以name的名称存储在收集器(collection)中 |
| tf.Graph.get\_collection(name, scope=None) | 根据name返回一个收集器中所收集的值的列表 |
| tf.Graph.as\_graph\_element (obj, allow\_tensor=True, allow\_operation=True) | 返回一个图中与obj相关联的对象，为一个操作节点或者tensor数据 |
| tf.Graph.get\_operation\_by\_name(name) | 根据名称返回操作节点 |
| tf.Graph.get\_tensor\_by\_name(name) | 根据名称返回tensor数据 |
| tf.Graph.get\_operations() | 返回图中的操作节点列表 |
| tf.Graph.gradient\_override\_map(op\_type\_map) | 用于覆盖梯度函数的上下文管理器 |

#class tf.Graph

#tensorflow运行时需要设置默认的图

g = tf.Graph()

g.as\_default():

# Define operations and tensors in `g`.

c = tf.constant()

assert c.graph g

##也可以使用tf.get\_default\_graph()获得默认图，也可在基础上加入节点或子图

c = tf.constant()

assert c.graph tf.get\_default\_graph()

#tf.Graph.as\_default

#以下两段代码功能相同

#1、使用Graph.as\_default():

g = tf.Graph()

g.as\_default():

c = tf.constant()

assert c.graph g

#2、构造和设置为默认

tf.Graph().as\_default() g:

c = tf.constant()

assert c.graph g

#tf.Graph.control\_dependencies(control\_inputs)

# 错误代码

my\_func(pred, tensor):

t = tf.matmul(tensor, tensor)

tf.control\_dependencies([pred]):

# 乘法操作(op)没有创建在该上下文，所以没有被加入依赖控制

return t

# 正确代码

my\_func(pred, tensor):

tf.control\_dependencies([pred]):

# 乘法操作(op)创建在该上下文，所以被加入依赖控制中

#执行完pred之后再执行matmul

return tf.matmul(tensor, tensor)

# tf.Graph.name\_scope(name)

# 一个图中包含有一个名称范围的堆栈，在使用name\_scope(...)之后，将压(push)新名称进栈中，

#并在下文中使用该名称

tf.Graph().as\_default() g:

c = tf.constant(, name=)

assert c.op.name ==

c\_1 = tf.constant(, name=)

assert c\_1.op.name == "c\_1"

# Creates a scope called "nested"

g.name\_scope("nested") scope:

nested\_c = tf.constant(, name=)

assert nested\_c.op.name == "nested/c"

# Creates a nested scope called "inner".

g.name\_scope("inner"):

nested\_inner\_c = tf.constant(, name=)

assert nested\_inner\_c.op.name == "nested/inner/c"

# Create a nested scope called "inner\_1".

g.name\_scope("inner"):

nested\_inner\_1\_c = tf.constant(, name=)

assert nested\_inner\_1\_c.op.name == "nested/inner\_1/c"

# Treats `scope` as an absolute name scope, and

# switches to the "nested/" scope.

g.name\_scope(scope):

nested\_d = tf.constant(, name=)

assert nested\_d.op.name == "nested/d"

g.name\_scope():

e = tf.constant(, name=)

assert e.op.name ==

**tf.Operation**

| **操作** | **描述** |
| --- | --- |
| class tf.Operation | 代表图中的一个节点，用于计算tensors数据 该类型将由python节点构造器产生(比如tf.matmul()) 或者Graph.create\_op() 例如c = tf.matmul(a, b)创建一个Operation类 为类型为”MatMul”,输入为’a’,’b’，输出为’c’的操作类 |
| tf.Operation.name | 操作节点(op)的名称 |
| tf.Operation.type | 操作节点(op)的类型，比如”MatMul” |
| tf.Operation.inputs tf.Operation.outputs | 操作节点的输入与输出 |
| tf.Operation.control\_inputs | 操作节点的依赖 |
| tf.Operation.run(feed\_dict=None, session=None) | 在会话(Session)中运行该操作 |
| tf.Operation.get\_attr(name) | 获取op的属性值 |

**tf.Tensor**

| **操作** | **描述** |
| --- | --- |
| class tf.Tensor | 表示一个由操作节点op产生的值， TensorFlow程序使用tensor数据结构来代表所有的数据,  计算图中, 操作间传递的数据都是 tensor，一个tensor是一个符号handle, 里面并没有表示实际数据，而相当于数据流的载体 |
| tf.Tensor.dtype | tensor中数据类型 |
| tf.Tensor.name | 该tensor名称 |
| tf.Tensor.value\_index | 该tensor输出外op的index |
| tf.Tensor.graph | 该tensor所处在的图 |
| tf.Tensor.op | 产生该tensor的op |
| tf.Tensor.consumers() | 返回使用该tensor的op列表 |
| tf.Tensor.eval(feed\_dict=None, session=None) | 在会话中求tensor的值 需要使用with sess.as\_default()或者eval(session=sess) |
| tf.Tensor.get\_shape() | 返回用于表示tensor的shape的类TensorShape |
| tf.Tensor.set\_shape(shape) | 更新tensor的shape |
| tf.Tensor.device | 设置计算该tensor的设备 |

#tf.Tensor.get\_shape()

c = tf.constant([[, , ], [, , ]])

print(c.get\_shape())

==> TensorShape([Dimension(), Dimension()])

#现在有个用于图像处理的tensor->image

print(image.get\_shape())

==> TensorShape([Dimension(), Dimension(), Dimension()])

# 假如我们知道数据集中图像尺寸为28 x 28，那么可以设置

image.set\_shape([, , ])

print(image.get\_shape())

==> TensorShape([Dimension(), Dimension(), Dimension()])

#### \* ****tensor类型(Tensor types)****

**tf.DType**

| **操作** | **描述** |
| --- | --- |
| class tf.DType | 数据类型主要包含 tf.float16，tf.float16,tf.float32,tf.float64, tf.bfloat16,tf.complex64,tf.complex128, tf.int8,tf.uint8,tf.uint16,tf.int16,tf.int32, tf.int64,tf.bool,tf.string |
| tf.DType.is\_compatible\_with(other) | 判断other的数据类型是否将转变为该DType |
| tf.DType.name | 数据类型名称 |
| tf.DType.base\_dtype | 返回该DType的基础DType，而非参考的数据类型(non-reference) |
| tf.DType.as\_ref | 返回一个基于DType的参考数据类型 |
| tf.DType.is\_floating | 判断是否为浮点类型 |
| tf.DType.is\_complex | 判断是否为复数 |
| tf.DType.is\_integer | 判断是否为整数 |
| tf.DType.is\_unsigned | 判断是否为无符号型数据 |
| tf.DType.as\_numpy\_dtype | 返回一个基于DType的numpy.dtype类型 |
| tf.DType.max tf.DType.min | 返回这种数据类型能表示的最大值及其最小值 |
| tf.as\_dtype(type\_value) | 返回由type\_value转变得的相应tf数据类型 |

#### \* ****通用函数（Utility functions）****

| **操作** | **描述** |
| --- | --- |
| tf.device(device\_name\_or\_function) | 基于默认的图，其功能便为Graph.device() |
| tf.container(container\_name) | 基于默认的图，其功能便为Graph.container() |
| tf.name\_scope(name) | 基于默认的图，其功能便为 Graph.name\_scope() |
| tf.control\_dependencies(control\_inputs) | 基于默认的图，其功能便为Graph.control\_dependencies() |
| **tf.convert\_to\_tensor (value, dtype=None, name=None, as\_ref=False)** | 将value转变为tensor数据类型 |
| tf.get\_default\_graph() | 返回返回当前线程的默认图 |
| tf.reset\_default\_graph() | 清除默认图的堆栈，并设置全局图为默认图 |
| tf.import\_graph\_def(graph\_def, input\_map=None, return\_elements=None, name=None, op\_dict=None, producer\_op\_list=None) | 将graph\_def的图导入到python中 |

#### \* ****图收集（Graph collections）****

| **操作** | **描述** |
| --- | --- |
| tf.add\_to\_collection(name, value) | 基于默认的图，其功能便为Graph.add\_to\_collection() |
| tf.get\_collection(key, scope=None) | 基于默认的图，其功能便为Graph.get\_collection() |

#### \* ****定义新操作节点（Defining new operations）****

**tf.RegisterGradient**

| **操作** | **描述** |
| --- | --- |
| class tf.RegisterGradient | 返回一个用于寄存op类型的梯度函数的装饰器 |
| tf.NoGradient(op\_type) | 设置操作节点类型op\_type的节点没有指定的梯度 |
| class tf.RegisterShape | 返回一个用于寄存op类型的shape函数的装饰器 |
| class tf.TensorShape | 表示tensor的shape |
| tf.TensorShape.merge\_with(other) | 与other合并shape信息，返回一个TensorShape类 |
| tf.TensorShape.concatenate(other) | 与other的维度相连结 |
| tf.TensorShape.ndims | 返回tensor的rank |
| tf.TensorShape.dims | 返回tensor的维度 |
| tf.TensorShape.as\_list() | 以list的形式返回tensor的shape |
| tf.TensorShape.is\_compatible\_with(other) | 判断shape是否为兼容 TensorShape(None)与其他任何shape值兼容 |
| class tf.Dimension |  |
| tf.Dimension.is\_compatible\_with(other) | 判断dims是否为兼容 |
| tf.Dimension.merge\_with(other) | 与other合并dims信息 |
| **tf.op\_scope(values, name, default\_name=None)** | 在python定义op时，返回一个上下文管理器 |

#tf.RegisterGradient

#该装饰器只使用于定义一个新的op类型时候，如果一个op有m个输入，n个输出。那么该梯度函数应该设置原始的

#操作类型，以及n个Tensor对象（表示每一个op输出的梯度），以及m个对象(表示每一个op输入的偏梯度)

#以操作节点类型为'Sub'为例，两输入为x,y。为一个输出x-y

@tf.RegisterGradient("Sub")

\_sub\_grad(unused\_op, grad):

return grad, tf.neg(grad)

#tf.op\_scope

#定义一个名称为my\_op的python操作节点op

my\_op(a, b, c, name=None):

tf.op\_scope([a, b, c], name, "MyOp") scope:

a = tf.convert\_to\_tensor(a, name=)

b = tf.convert\_to\_tensor(b, name=)

c = tf.convert\_to\_tensor(c, name=)

# Define some computation that uses `a`, `b`, and `c`.

return foo\_op(..., name=scope)

### 2.2 ****输入和读取器(Inputs and Readers)****

本节主要介绍tensorflow中数据的读入相关类或函数

#### \* ****占位符（Placeholders）****

tf提供一种占位符操作，在执行时需要为其提供数据data。

| **操作** | **描述** |
| --- | --- |
| tf.placeholder(dtype, shape=None, name=None) | 为一个tensor插入一个占位符 eg:x = tf.placeholder(tf.float32, shape=(1024, 1024)) |
| tf.placeholder\_with\_default(input, shape, name=None) | 当输出没有fed时，input通过一个占位符op |
| tf.sparse\_placeholder(dtype, shape=None, name=None) | 为一个稀疏tensor插入一个占位符 |

#### \* ****读取器（Readers）****

tf提供一系列读取各种数据格式的类。对于多文件输入，可以使用函数[tf.train.string\_input\_producer](https://www.tensorflow.org/versions/r0.10/api_docs/python/io_ops.html#string_input_producer)，该函数将创建一个保持文件的FIFO队列，以供reader使用。或者如果输入的这些文件名有相雷同的字符串，也可以使用函数[tf.train.match\_filenames\_once](https://www.tensorflow.org/versions/r0.10/api_docs/python/io_ops.html#match_filenames_once)。

| **操作** | **描述** |
| --- | --- |
| class tf.ReaderBase | 不同的读取器类型的基本类 |
| tf.ReaderBase.read(queue, name=None) | 返回下一个记录对(key, value),queue为tf文件队列FIFOQueue |
| tf.ReaderBase.read\_up\_to(queue, num\_records, name=None) | 返回reader产生的num\_records对(key, value) |
| tf.ReaderBase.reader\_ref | 返回应用在该reader上的Op |
| tf.ReaderBase.reset(name=None) | 恢复reader为初始状态 |
| tf.ReaderBase.restore\_state(state, name=None) | 恢复reader为之前的保存状态state |
| tf.ReaderBase.serialize\_state(name=None) | 返回一个reader解码后产生的字符串tansor |
| class tf.TextLineReader |  |
| tf.TextLineReader.num\_records\_produced(name=None) | 返回reader已经产生的记录(records )数目 |
| tf.TextLineReader.num\_work\_units\_completed(name=None) | 返回该reader已经完成的处理的work数目 |
| tf.TextLineReader.read(queue, name=None) | 返回reader所产生的下一个记录对 (key, value)，该reader可以限定新产生输出的行数 |
| tf.TextLineReader.reader\_ref | 返回应用在该reader上的Op |
| tf.TextLineReader.reset(name=None) | 恢复reader为初始状态 |
| tf.TextLineReader.restore\_state(state, name=None) | 恢复reader为之前的保存状态state |
| tf.TextLineReader.serialize\_state(name=None) | 返回一个reader解码后产生的字符串tansor |
| class tf.WholeFileReader | 一个阅读器，读取整个文件，返回文件名称key,以及文件中所有的内容value,该类的方法同上，不赘述 |
| class tf.IdentityReader | 一个reader，以key和value的形式，输出一个work队列。该类其他方法基本同上 |
| class tf.TFRecordReader | 读取TFRecord格式文件的reader。该类其他方法基本同上 |
| class tf.FixedLengthRecordReader | 输出 |

#### \* ****数据转换（Converting）****

tf提供一系列方法将各种格式数据转换为tensor表示。

| **操作** | **描述** |
| --- | --- |
| tf.decode\_csv(records, record\_defaults,  field\_delim=None, name=None) | 将csv转换为tensor，与tf.TextLineReader搭配使用 |
| tf.decode\_raw(bytes, out\_type,  little\_endian=None, name=None) | 将bytes转换为一个数字向量表示，bytes为一个字符串类型的tensor 与函数 tf.FixedLengthRecordReader搭配使用，详见[tf的CIFAR-10例子](https://www.tensorflow.org/code/tensorflow/models/image/cifar10/cifar10_input.py) |

选取与要输入的文件格式相匹配的reader，并将文件队列提供给reader的读方法( read method)。读方法将返回文件唯一标识的key，以及一个记录(record)（有助于对出现一些另类的records时debug），以及一个标量的字符串值。再使用一个（或多个）解码器(decoder) 或转换操作(conversion ops)将字符串转换为tensor类型。

#读取文件队列，使用reader中read的方法，返回key与value

filename\_queue = tf.train.string\_input\_producer(["file0.csv", "file1.csv"])

reader = tf.TextLineReader()

key, value = reader.read(filename\_queue)

record\_defaults = [[], [], [], [], []]

col1, col2, col3, col4, col5 = tf.decode\_csv(

value, record\_defaults=record\_defaults)

features = tf.pack([col1, col2, col3, col4])

tf.Session() sess:

# Start populating the filename queue.

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(coord=coord)

i range():

# Retrieve a single instance:

example, label = sess.run([features, col5])

coord.request\_stop()

coord.join(threads)

#### \* ****Example protocol buffer****

提供了一些[Example protocol buffers](https://www.tensorflow.org/code/tensorflow/core/example/example.proto)，tf所[推荐的用于训练样本的数据格式](https://www.tensorflow.org/versions/r0.10/how_tos/reading_data/index.html#standard-tensorflow-format)，它们包含特征信息，[详情可见](https://www.tensorflow.org/code/tensorflow/core/example/feature.proto)。   
这是一种与前述将手上现有的各种数据类型转换为支持的格式的方法，这种方法更容易将网络结构与数据集融合或匹配。这种tensorflow所推荐的数据格式是一个包含tf.train.Example protocol buffers (包含特征[Features](https://www.tensorflow.org/code/tensorflow/core/example/feature.proto)域)的TFRecords文件。   
1、获取这种格式的文件方式为，首先将一般的数据格式填入Example protocol buffer中，再将 protocol buffer序列化为一个字符串，然后使用tf.python\_io.TFRecordWriter类的相关方法将字符串写入一个TFRecords文件中，[参见MNIST例子](https://www.tensorflow.org/code/tensorflow/examples/how_tos/reading_data/convert_to_records.py)，将MNIST 数据转换为该类型数据。   
2、读取TFRecords格式文件的方法为，使用tf.TFRecordReader读取器和tf.parse\_single\_example解码器。parse\_single\_example操作将 example protocol buffers解码为tensor形式。[参见MNIST例子](https://www.tensorflow.org/code/tensorflow/examples/how_tos/reading_data/fully_connected_reader.py)

| **操作** | **描述** |
| --- | --- |
| class tf.VarLenFeature | 解析变长的输入特征feature相关配置 |
| class tf.FixedLenFeature | 解析定长的输入特征feature相关配置 |
| class tf.FixedLenSequenceFeature | 序列项目中的稠密(dense )输入特征的相关配置 |
| **tf.parse\_example(serialized, features,  name=None, example\_names=None)** | 将一组Example protos解析为tensor的字典形式 解析serialized所给予的序列化的一些Example protos 返回一个由特征keys映射Tensor和SparseTensor值的字典 |
| **tf.parse\_single\_example(serialized, features,  name=None, example\_names=None)** | 解析一个单独的Example proto，与tf.parse\_example方法雷同 |
| tf.decode\_json\_example(json\_examples, name=None) | 将JSON编码的样本记录转换为二进制protocol buffer字符串 |

#tf.parse\_example的使用举例

#输入序列化数据如下：

serialized = [

features

{ feature { key: value { float\_list { value: [, ] } } } },

features

{ feature []},

features

{ feature { key: value { float\_list { value: [] } } }

]

#那么输出为一个字典(dict),如下：

{: SparseTensor(indices=[[, ], [, ], [, ]],

values=[, , ],

shape=(, )) }

#########

#再来看一个例子，给定两个序列化的原始输入样本：

[

features {

feature { key: value { bytes\_list { value: [ "knit", "big" ] } } }

feature { key: "gps" value { float\_list { value: [] } } }

},

features {

feature { key: value { bytes\_list { value: [ "emmy" ] } } }

feature { key: "dank" value { int64\_list { value: [ ] } } }

feature { key: "gps" value { } }

}

]

#相关参数如下：

example\_names: ["input0", "input1"],

features: {

: VarLenFeature(tf.string),

"dank": VarLenFeature(tf.int64),

"gps": VarLenFeature(tf.float32),

}

#那么有如下输出：

{

: SparseTensor(

indices=[[, ], [, ], [, ]],

values=["knit", "big", "emmy"]

shape=[, ]),

"dank": SparseTensor(

indices=[[, ]],

values=[],

shape=[, ]),

"gps": SparseTensor(

indices=[],

values=[],

shape=[, ]),

}

#########

#对于两个样本的输出稠密结果情况

[

features {

feature { key: "age" value { int64\_list { value: [ ] } } }

feature { key: "gender" value { bytes\_list { value: [ ] } } }

},

features {

feature { key: "age" value { int64\_list { value: [] } } }

feature { key: "gender" value { bytes\_list { value: [ ] } } }

}

]

#我们可以使用以下参数

example\_names: ["input0", "input1"],

features: {

"age": FixedLenFeature([], dtype=tf.int64, default\_value=-),

"gender": FixedLenFeature([], dtype=tf.string),

}

#期望的结果如下

{

"age": [[], [-]],

"gender": [[], []],

}

##Example protocol buffer相关使用的例子

#将mnist的数据转换为TFRecords文件格式

import os

import tensorflow tf

tensorflow.contrib.learn.python.learn.datasets import mnist

SOURCE\_URL = 'http://yann.lecun.com/exdb/mnist/'

TRAIN\_IMAGES = 'train-images-idx3-ubyte.gz' # MNIST filenames

TRAIN\_LABELS = 'train-labels-idx1-ubyte.gz'

TEST\_IMAGES = 't10k-images-idx3-ubyte.gz'

TEST\_LABELS = 't10k-labels-idx1-ubyte.gz'

tf.app.flags.DEFINE\_string('directory', '/tmp/data',

'Directory to download data files and write the '

'converted result')

tf.app.flags.DEFINE\_integer('validation\_size', ,

'Number of examples to separate from the training '

'data for the validation set.')

FLAGS = tf.app.flags.FLAGS

\_int64\_feature(value):

return tf.train.Feature(int64\_list=tf.train.Int64List(value=[value]))

\_bytes\_feature(value):

return tf.train.Feature(bytes\_list=tf.train.BytesList(value=[value]))

convert\_to(data\_set, name):

images = data\_set.images

labels = data\_set.labels

num\_examples = data\_set.num\_examples

images.shape[] != num\_examples:

raise ValueError('Images size %d does not match label size %d.' %

(images.shape[], num\_examples))

rows = images.shape[]

cols = images.shape[]

depth = images.shape[]

filename = os.path.join(FLAGS.directory, name + '.tfrecords')

print('Writing', filename)

writer = tf.python\_io.TFRecordWriter(filename)

index range(num\_examples):

image\_raw = images[index].tostring()

example = tf.train.Example(features=tf.train.Features(feature={

'height': \_int64\_feature(rows),

'width': \_int64\_feature(cols),

'depth': \_int64\_feature(depth),

'label': \_int64\_feature(int(labels[index])),

'image\_raw': \_bytes\_feature(image\_raw)}))

writer.write(example.SerializeToString())

writer.close()

(argv):

# Get the data.

data\_sets = mnist.read\_data\_sets(FLAGS.directory,

dtype=tf.uint8,

reshape=False)

# Convert to Examples and write the result to TFRecords.

convert\_to(data\_sets.train, 'train')

convert\_to(data\_sets.validation, 'validation')

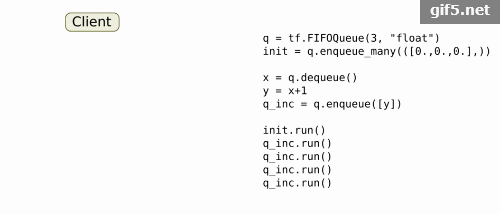
convert\_to(data\_sets.test, 'test')

\_\_name\_\_ == '\_\_main\_\_':

tf.app.run()

#### \* ****队列(Queues)****

tensorflow提供了几个队列应用，用来将tf计算图与tensors的阶段流水组织到一起。队列是使用tensorflow计算的一个强大的机制，正如其他Tensorflow的元素一样，一个队列也是tf图中的一个节点(node),它是一个有状态的node，就像一个变量：其他节点可以改变其内容。   
我们来看一个简单的例子，如下gif图，我们将创建一个先入先出队列(FIFOQueue)并且将值全设为0，然后我们构建一个图以获取队列出来的元素，对该元素加1操作，并将结果再放入队列末尾。渐渐地，队列中的数字便增加。



| **操作** | **描述** |
| --- | --- |
| class tf.QueueBase | 基本的队列应用类.队列(queue)是一种数据结构， 该结构通过多个步骤存储tensors, 并且对tensors进行入列(enqueue)与出列(dequeue)操作 |
| tf.QueueBase.enqueue(vals, name=None) | 将一个元素编入该队列中。如果在执行该操作时队列已满， 那么将会阻塞直到元素编入队列之中 |
| tf.QueueBase.enqueue\_many(vals, name=None) | 将零个或多个元素编入该队列中 |
| tf.QueueBase.dequeue(name=None) | 将元素从队列中移出。如果在执行该操作时队列已空， 那么将会阻塞直到元素出列，返回出列的tensors的tuple |
| tf.QueueBase.dequeue\_many(n, name=None) | 将一个或多个元素从队列中移出 |
| tf.QueueBase.size(name=None) | 计算队列中的元素个数 |
| tf.QueueBase.close (cancel\_pending\_enqueues=False, name=None) | 关闭该队列 |
| f.QueueBase.dequeue\_up\_to(n, name=None) | 从该队列中移出n个元素并将之连接 |
| tf.QueueBase.dtypes | 列出组成元素的数据类型 |
| tf.QueueBase.from\_list(index, queues) | 根据queues[index]的参考队列创建一个队列 |
| tf.QueueBase.name | 返回最队列下面元素的名称 |
| tf.QueueBase.names | 返回队列每一个组成部分的名称 |
| **class tf.FIFOQueue** | 在出列时依照先入先出顺序，其他方法与tf.QueueBase雷同 |
| class tf.PaddingFIFOQueue | 一个FIFOQueue ，同时根据padding支持batching变长的tensor |
| **class tf.RandomShuffleQueue** | 该队列将随机元素出列，其他方法与tf.QueueBase雷同 |

#### \* ****文件系统的处理(Dealing with the filesystem)****

| **操作** | **描述** |
| --- | --- |
| tf.matching\_files(pattern, name=None) | 返回与pattern匹配模式的文件名称 |
| **tf.read\_file(filename, name=None)** | 读取并输出输入文件的整个内容 |

#### \* ****输入管道(Input pipeline)****

用于设置输入预取数的管道TF函数，函数 “producer”添加一个队列至图中，同时一个相应用于运行队列中子图(subgraph)的QueueRunner

| **操作** | **描述** |
| --- | --- |
| tf.train.match\_filenames\_once(pattern, name=None) | 保存与pattern的文件列表 |
| tf.train.limit\_epochs(tensor, num\_epochs=None, name=None) | 返回一个num\_epochs次数，然后报告OutOfRange错误 |
| tf.train.input\_producer(input\_tensor, element\_shape=None,  num\_epochs=None, shuffle=True, seed=None, capacity=32,  shared\_name=None, summary\_name=None, name=None) | 为一个输入管道输出input\_tensor中的多行至一个队列中 |
| tf.train.range\_input\_producer(limit, num\_epochs=None,  shuffle=True, seed=None, capacity=32,  shared\_name=None, name=None) | 产生一个从1至limit-1的整数至队列中 |
| tf.train.slice\_input\_producer(tensor\_list, num\_epochs=None,  shuffle=True, seed=None, capacity=32,  shared\_name=None, name=None) | 对tensor\_list中的每一个tensor切片 |
| **tf.train.string\_input\_producer(string\_tensor, num\_epochs=None, shuffle=True, seed=None, capacity=32,  shared\_name=None, name=None)** | 为一个输入管道输出一组字符串(比如文件名)至队列中 |

#### \* ****在输入管道末端批量打包(Batching at the end of an input pipeline)****

该相关函数增添一个队列至图中以将数据一样本打包为batch。它们也会添加 一个QueueRunner，以便执行的已经被填满队列的子图

| **操作** | **描述** |
| --- | --- |
| tf.train.batch(tensors, batch\_size, num\_threads=1, capacity=32, enqueue\_many=False, shapes=None,  dynamic\_pad=False, allow\_smaller\_final\_batch=False,  shared\_name=None, name=None) | 在输入的tensors中创建一些tensor数据格式的batch， 若输入为shape[\*, x, y, z]，那么输出则为[batch\_size, x, y, z] 返回一个列表或者一个具有与输入tensors相同类型tensors的字典 |
| tf.train.batch\_join(tensors\_list, batch\_size,  capacity=32, enqueue\_many=False, shapes=None,  dynamic\_pad=False, allow\_smaller\_final\_batch=False,  shared\_name=None, name=None) | 将一个tensors的列表添加至一个队列中以创建样本的batches len(tensors\_list)个线程将启动， 线程i将tensors\_list[i]的tensors入列 tensors\_list[i1][j]与tensors\_list[i2][j]有相同的类型和shape |
| **tf.train.shuffle\_batch(tensors, batch\_size, capacity,  min\_after\_dequeue, num\_threads=1, seed=None,  enqueue\_many=False, shapes=None,  allow\_smaller\_final\_batch=False, shared\_name=None, name=None)** | 使用随机乱序的方法创建batches tensors:用于入列的一个list或者dict capacity:一个整数，表示队列中元素最大数目 |
| tf.train.shuffle\_batch\_join(tensors\_list, batch\_size,  capacity, min\_after\_dequeue, seed=None,  enqueue\_many=False, shapes=None,  allow\_smaller\_final\_batch=False,  shared\_name=None, name=None) | 随机乱序的tensors创建batches， 其中tensors\_list参数为tensors元组或tensors字典的列表 len(tensors\_list)个线程将启动， 线程i将tensors\_list[i]的tensors入列 tensors\_list[i1][j]与tensors\_list[i2][j]有相同的类型和shape |

# 一个简单例子，使用tf.train.shuffle\_batch创建一个具有32张图像和32个标签的batches.

image\_batch, label\_batch = tf.train.shuffle\_batch(

[single\_image, single\_label],

batch\_size=,

num\_threads=,

capacity=50000,

min\_after\_dequeue=10000)

#Batching函数相关例子，以函数tf.train.shuffle\_batch为例

#为training, evaluation等操作将样本batching，以下代码使用随机顺序打包样本

read\_my\_file\_format(filename\_queue):

reader = tf.SomeReader()

key, record\_string = reader.read(filename\_queue)

example, label = tf.some\_decoder(record\_string)

processed\_example = some\_processing(example)

return processed\_example, label

input\_pipeline(filenames, batch\_size, num\_epochs=None):

filename\_queue = tf.train.string\_input\_producer(

filenames, num\_epochs=num\_epochs, shuffle=)

example, label = read\_my\_file\_format(filename\_queue)

# min\_after\_dequeue defines how big a buffer we will randomly sample

# from -- bigger means better shuffling but slower start up and more

# memory used.

# capacity must be larger than min\_after\_dequeue and the amount larger

# determines the maximum we will prefetch. Recommendation:

# min\_after\_dequeue + (num\_threads + a small safety margin) \* batch\_size

min\_after\_dequeue = 10000

capacity = min\_after\_dequeue + \* batch\_size

example\_batch, label\_batch = tf.train.shuffle\_batch(

[example, label], batch\_size=batch\_size, capacity=capacity,

min\_after\_dequeue=min\_after\_dequeue)

return example\_batch, label\_batch

#如果需要跟多的并行或文件之间的样本乱序操作，可以使用函数tf.train.shuffle\_batch\_join多实例化reader

read\_my\_file\_format(filename\_queue):

# 与上例子相同

input\_pipeline(filenames, batch\_size, read\_threads, num\_epochs=None):

filename\_queue = tf.train.string\_input\_producer(

filenames, num\_epochs=num\_epochs, shuffle=)

example\_list = [read\_my\_file\_format(filename\_queue)

\_ range(read\_threads)]

min\_after\_dequeue = 10000

capacity = min\_after\_dequeue + \* batch\_size

example\_batch, label\_batch = tf.train.shuffle\_batch\_join(

example\_list, batch\_size=batch\_size, capacity=capacity,

min\_after\_dequeue=min\_after\_dequeue)

return example\_batch, label\_batch

**相关链接：**

[1] 安装Tensorflow（[Linux](http://lib.csdn.net/base/linux) ubuntu）<http://blog.csdn.net/lenbow/article/details/51203526>   
[2] ubuntu下CUDA编译的GCC降级安装<http://blog.csdn.net/lenbow/article/details/51596706>   
[3] ubuntu手动安装最新Nvidia显卡驱动<http://blog.csdn.net/lenbow/article/details/51683783>   
[4] Tensorflow的CUDA升级，以及相关配置<http://blog.csdn.net/lenbow/article/details/52118116>   
[5] 基于gensim的Doc2Vec简析<http://blog.csdn.net/lenbow/article/details/52120230>   
[6] TensorFlow的分布式学习框架简介<http://blog.csdn.net/lenbow/article/details/52130565>   
[7] Tensorflow一些常用基本概念与函数（1）<http://blog.csdn.net/lenbow/article/details/52152766>

Measure

Measure

免费注册印象笔记帐户以保存文章，以后随时在手机、平板或电脑上阅读。

[创建帐户](chrome-extension://iooicodkiihhpojmmeghjclgihfjdjhj/front/in_isolation/reformat.html)

# Tensorflow一些常用基本概念与函数（3）

2016-08-15 17:29 4073人阅读 [评论](http://blog.csdn.net/lenbow/article/details/52213105#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/lenbow/article/details/52213105#report)

版权声明：本文为博主原创文章，未经博主允许不得转载。

摘要：本系列主要对tf的一些常用概念与方法进行描述。本文主要针对tensorflow的数据IO、图的运行等相关函数进行讲解。为‘Tensorflow一些常用基本概念与函数’系列之三。

本文所讲的内容主要为以下相关函数：

| **操作组** | **操作** |
| --- | --- |
| Data IO (Python functions) | TFRecordWrite，rtf\_record\_iterator |
| Running Graphs | Session management，Error classes |

## 2、tf函数

### 2.1 数据IO {Data IO (Python functions)}

一个TFRecords 文件为一个字符串序列。这种格式并非随机获取，它比较适合大规模的数据流，而不太适合需要快速分区或其他非序列获取方式。

#### 数据IO {Data IO (Python functions)}

| **操作** | **描述** |
| --- | --- |
| class tf.python\_io.TFRecordWriter | 一个用于将记录(records)写入TFRecords文件的类 |
| tf.python\_io.TFRecordWriter.\_\_init\_\_(path, options=None) | 打开文件路径，并创建一个TFRecordWriter以供写入 |
| tf.python\_io.TFRecordWriter.write(record) | 将一个字符串records写入文件中 |
| tf.python\_io.TFRecordWriter.close() | 关闭文件 |
| tf.python\_io.tf\_record\_iterator(path, options=None) | 从TFRecords文件中读取记录的迭代器 |

### 2.2 运行图(Running Graphs)

#### ****会话管理 (Session management)****

| **操作** | **描述** |
| --- | --- |
| class tf.Session | 运行TF操作的类, 一个Session对象将操作节点op封装在一定的环境内运行， 同时tensor对象将被计算求值 |
| tf.Session.\_\_init\_\_(target=”, graph=None, config=None) | 创建一个新的会话 |
| tf.Session.run(fetches, feed\_dict=None,  options=None, run\_metadata=None) | 运行fetches中的操作节点并求其值 |
| tf.Session.close() | 关闭会话 |
| tf.Session.graph | 返回加载值该会话的图(graph) |
| **tf.Session.as\_default()** | 设置该对象为默认会话，并返回一个上下文管理器 |
| tf.Session.reset(target, containers=None, config=None) | 重设target的资源容器，并关闭所有连接的会话 在0.10版本该功能仅应用在分布会话中 target:为执行引擎所连接的目标，其包含有资源容器， 该资源容器分布在同一个集群的所有works上 |
| class tf.InteractiveSession | 使用在交互式上下文环境的tf会话，比如shell，ipython |
| tf.InteractiveSession.close() | 关闭一个InteractiveSession |
| tf.get\_default\_session() | 返回当前线程的默认会话 |

**tf.Session**

#一个简单的tf.Session例子

# 建立一个graph.

a = tf.constant()

b = tf.constant()

c = a \* b

# 将graph载入到一个会话session中

sess = tf.Session()

# 计算tensor `c`.

print(sess.run(c))

#一个会话可能会占用一些资源，比如变量、队列和读取器(reader)。释放这些不再使用的资源非常重要。

#使用close()方法关闭会话，或者使用上下文管理器，释放资源。

# 使用`close()`方法.

sess = tf.Session()

sess.run(...)

sess.close()

# 使用上下文管理器

tf.Session() sess:

sess.run(...)

tf.Session()的变量设置， ConfigProto protocol buffer为会话提供了不同的配置选项。比如，创建一个会话，对设备布局使用软约束条件，以及对分布

# Launch the graph in a session that allows soft device placement and

# logs the placement decisions.

sess = tf.Session(config=tf.ConfigProto(allow\_soft\_placement=,

log\_device\_placement=))

**tf.Session.run**

a = tf.constant([, ])

b = tf.constant([, ])

# 'fetches' 可以为单个数

v = session.run(a)

# v is the numpy array [10, 20]

# 'fetches' 可以为一个list.

v = session.run([a, b])

# v a Python list with 2 numpy arrays: the numpy array [10, 20] and the

# 1-D array [1.0, 2.0]

# 'fetches' 可以是 lists, tuples, namedtuple, dicts中的任意:

MyData = collections.namedtuple('MyData', [, ])

v = session.run({: MyData(a, b), : [b, a]})

# v 为一个dict，并有

# v['k1'] is a MyData namedtuple with 'a' the numpy array [10, 20] and

# 'b' the numpy array [1.0, 2.0]

# v['k2'] is a list with the numpy array [1.0, 2.0] and the numpy array

# [10, 20].

**tf.Session.as\_default()**   
使用关键字with指定会话， 可以在会话中执行[Operation.run()](http://blog.csdn.net/lenbow/article/details/52181159)或[Tensor.eval()](http://blog.csdn.net/lenbow/article/details/52181159)，以得到运行的tensor结果

c = tf.constant(..)

sess = tf.Session()

sess.as\_default():

assert tf.get\_default\_session() sess

print(c.eval())

使用函数tf.get\_default\_session()来得到当前默认的会话   
需要注意的是，退出该as\_default上下文管理器时，并没有关闭该会话(session )，必须明确的关闭会话

c = tf.constant(...)

sess = tf.Session()

sess.as\_default():

print(c.eval())

sess.as\_default():

print(c.eval())

#关闭会话

sess.close()

#使用 with tf.Session()方式可以创建并自动关闭会话

**tf.InteractiveSession**

sess = tf.InteractiveSession()

a = tf.constant()

b = tf.constant()

c = a \* b

# 我们直接使用'c.eval()' 而没有通过'sess'

print(c.eval())

sess.close()

以上的例子，在非交互会话的版本中为，

a = tf.constant()

b = tf.constant()

c = a \* b

tf.Session():

# We can also use 'c.eval()' here.

print(c.eval())

#### ****错误类 (Error classes)****

| **操作** | **描述** |
| --- | --- |
| class tf.OpError | 一个基本的错误类型，在当TF执行失败时候报错 |
| tf.OpError.op | 返回执行失败的操作节点， 有的操作如Send或Recv可能不会返回，那就要用用到node\_def方法 |
| **tf.OpError.node\_def** | 以NodeDef proto形式表示失败的op |
| tf.OpError.error\_code | 描述该错误的整数错误代码 |
| tf.OpError.message | 返回错误信息 |
| class tf.errors.CancelledError | 当操作或者阶段呗取消时候报错 |
| class tf.errors.UnknownError | 未知错误类型 |
| class tf.errors.InvalidArgumentError | 在接收到非法参数时候报错 |
| class tf.errors.NotFoundError | 当发现不存在所请求的一个实体时候，比如文件或目录 |
| class tf.errors.AlreadyExistsError | 当创建的实体已经存在的时候报错 |
| class tf.errors.PermissionDeniedError | 没有执行权限做某操作的时候报错 |
| class tf.errors.ResourceExhaustedError | 资源耗尽时报错 |
| class tf.errors.FailedPreconditionError | 系统没有条件执行某个行为时候报错 |
| class tf.errors.AbortedError | 操作中止时报错，常常发生在并发情形 |
| class tf.errors.OutOfRangeError | 超出范围报错 |
| class tf.errors.UnimplementedError | 某个操作没有执行时报错 |
| class tf.errors.InternalError | 当系统经历了一个内部错误时报出 |
| class tf.errors.DataLossError | 当出现不可恢复的错误 例如在运行 [tf.WholeFileReader.read()](http://blog.csdn.net/lenbow/article/details/52181159)读取整个文件的同时文件被删减 |
| tf.errors.XXXXX.\_\_init\_\_(node\_def, op, message) | 使用该形式方法创建以上各种错误类 |

相关链接：

[1] 安装Tensorflow（[Linux](http://lib.csdn.net/base/linux) ubuntu）<http://blog.csdn.net/lenbow/article/details/51203526>   
[2] ubuntu下CUDA编译的GCC降级安装<http://blog.csdn.net/lenbow/article/details/51596706>   
[3] ubuntu手动安装最新Nvidia显卡驱动<http://blog.csdn.net/lenbow/article/details/51683783>   
[4] Tensorflow的CUDA升级，以及相关配置<http://blog.csdn.net/lenbow/article/details/52118116>   
[5] 基于gensim的Doc2Vec简析<http://blog.csdn.net/lenbow/article/details/52120230>   
[6] TensorFlow的分布式学习框架简介<http://blog.csdn.net/lenbow/article/details/52130565>   
[7] Tensorflow一些常用基本概念与函数（1）<http://blog.csdn.net/lenbow/article/details/52152766>   
[8] Tensorflow一些常用基本概念与函数（2）<http://blog.csdn.net/lenbow/article/details/52181159>

Measure

Measure

免费注册印象笔记帐户以保存文章，以后随时在手机、平板或电脑上阅读。

[创建帐户](chrome-extension://iooicodkiihhpojmmeghjclgihfjdjhj/front/in_isolation/reformat.html)

# Tensorflow一些常用基本概念与函数（4）

版权声明：本文为博主原创文章，未经博主允许不得转载。

摘要：本系列主要对tf的一些常用概念与方法进行描述。本文主要针对tensorflow的模型训练Training与[测试](http://lib.csdn.net/base/softwaretest)Testing等相关函数进行讲解。为‘Tensorflow一些常用基本概念与函数’系列之四。

本文所讲的内容主要为以下列表中相关函数。函数training()通过梯度下降法为最小化损失函数增加了相关的优化操作，在训练过程中，先实例化一个优化函数，比如 tf.train.GradientDescentOptimizer，并基于一定的学习率进行梯度优化训练：

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

然后，可以设置 一个用于记录全局训练步骤的单值。以及使用minimize()操作，该操作不仅可以优化更新训练的模型参数，也可以为全局步骤(global step)计数。与其他tensorflow操作类似，这些训练操作都需要在[tf.session](http://blog.csdn.net/lenbow/article/details/52181159)会话中进行

global\_step = tf.Variable(0, name='global\_step', trainable=False)

train\_op = optimizer.minimize(loss, global\_step=global\_step)

| **操作组** | **操作** |
| --- | --- |
| Training | Optimizers，Gradient Computation，Gradient Clipping，Distributed execution |
| Testing | Unit tests，Utilities，Gradient checking |

## 2、Tensorflow函数

### 2.1 训练 (Training)

一个TFRecords 文件为一个字符串序列。这种格式并非随机获取，它比较适合大规模的数据流，而不太适合需要快速分区或其他非序列获取方式。

#### █ ****优化 (Optimizers)****

tf中各种优化类提供了为损失函数计算梯度的方法，其中包含比较经典的优化[算法](http://lib.csdn.net/base/datastructure)，比如GradientDescent 和Adagrad。

▶▶**class tf.train.Optimizer**

| **操作** | **描述** |
| --- | --- |
| class tf.train.Optimizer | 基本的优化类，该类不常常被直接调用，而较多使用其子类， 比如GradientDescentOptimizer, AdagradOptimizer 或者MomentumOptimizer |
| tf.train.Optimizer.\_\_init\_\_(use\_locking, name) | 创建一个新的优化器， 该优化器必须被其子类(subclasses)的构造函数调用 |
| **tf.train.Optimizer.minimize(loss, global\_step=None,  var\_list=None, gate\_gradients=1,  aggregation\_method=None, colocate\_gradients\_with\_ops=False,  name=None, grad\_loss=None)** | 添加操作节点，用于最小化loss，并更新var\_list 该函数是简单的合并了compute\_gradients()与apply\_gradients()函数 返回为一个优化更新后的var\_list，如果global\_step非None，该操作还会为global\_step做自增操作 |
| tf.train.Optimizer.compute\_gradients(loss,var\_list=None, gate\_gradients=1, aggregation\_method=None,  colocate\_gradients\_with\_ops=False, grad\_loss=None) | 对var\_list中的变量计算loss的梯度 该函数为函数minimize()的第一部分，返回一个以元组(gradient, variable)组成的列表 |
| tf.train.Optimizer.apply\_gradients(grads\_and\_vars, global\_step=None, name=None) | 将计算出的梯度应用到变量上，是函数minimize()的第二部分，返回一个应用指定的梯度的操作Operation，对global\_step做自增操作 |
| tf.train.Optimizer.get\_name() | 获取名称 |

▷ class tf.train.Optimizer   
用法

# Create an optimizer with the desired parameters.

opt = GradientDescentOptimizer(learning\_rate=)

# Add Ops to the graph to minimize a cost by updating a list of variables.

# "cost" is a Tensor, and the list of variables contains tf.Variable objects.

opt\_op = opt.minimize(cost, var\_list=<list of variables>)

# Execute opt\_op to do one step of training:

opt\_op.run()

▶▶**在使用它们之前处理梯度**   
使用minimize()操作，该操作不仅可以计算出梯度，而且还可以将梯度作用在变量上。如果想在使用它们之前处理梯度，可以按照以下三步骤使用optimizer ：

1、使用函数compute\_gradients()计算梯度

2、按照自己的愿望处理梯度

3、使用函数apply\_gradients()应用处理过后的梯度

# 创建一个optimizer.

opt = GradientDescentOptimizer(learning\_rate=)

# 计算<list of variables>相关的梯度

grads\_and\_vars = opt.compute\_gradients(loss, <list of variables>)

# grads\_and\_vars为tuples (gradient, variable)组成的列表。

#对梯度进行想要的处理，比如cap处理

capped\_grads\_and\_vars = [(MyCapper(gv[]), gv[]) gv grads\_and\_vars]

# 令optimizer运用capped的梯度(gradients)

opt.apply\_gradients(capped\_grads\_and\_vars)

▶▶**选通梯度(Gating Gradients)**   
函数minimize() 与compute\_gradients()都含有一个参数gate\_gradient，用于控制在应用这些梯度时并行化的程度。

其值可以取：GATE\_NONE, GATE\_OP 或 GATE\_GRAPH   
GATE\_NONE : 并行地计算和应用梯度。提供最大化的并行执行，但是会导致有的数据结果没有再现性。比如两个matmul操作的梯度依赖输入值，使用GATE\_NONE可能会出现有一个梯度在其他梯度之前便应用到某个输入中，导致出现不可再现的(non-reproducible)结果   
GATE\_OP: 对于每个操作Op，确保每一个梯度在使用之前都已经计算完成。这种做法防止了那些具有多个输入，并且梯度计算依赖输入情形中，多输入Ops之间的竞争情况出现。   
GATE\_GRAPH: 确保所有的变量对应的所有梯度在他们任何一个被使用前计算完成。该方式具有最低级别的并行化程度，但是对于想要在应用它们任何一个之前处理完所有的梯度计算时很有帮助的。

#### █ ****Slots****

一些optimizer的之类，比如 MomentumOptimizer 和 AdagradOptimizer 分配和管理着额外的用于训练的变量。这些变量称之为’Slots’，Slots有相应的名称，可以向optimizer访问的slots名称。有助于在log debug一个训练算法以及报告slots状态

| **操作** | **描述** |
| --- | --- |
| tf.train.Optimizer.get\_slot\_names() | 返回一个由Optimizer所创建的slots的名称列表 |
| tf.train.Optimizer.get\_slot(var, name) | 返回一个name所对应的slot，name是由Optimizer为var所创建 var为用于传入 minimize() 或 apply\_gradients()的变量 |
| **class tf.train.GradientDescentOptimizer** | 使用梯度下降算法的Optimizer |
| tf.train.GradientDescentOptimizer.\_\_init\_\_(learning\_rate, use\_locking=False, name=’GradientDescent’) | 构建一个新的梯度下降优化器(Optimizer) |
| **class tf.train.AdadeltaOptimizer** | 使用[Adadelta算法](http://arxiv.org/abs/1212.5701)的Optimizer |
| tf.train.AdadeltaOptimizer.\_\_init\_\_(learning\_rate=0.001,  rho=0.95, epsilon=1e-08,  use\_locking=False, name=’Adadelta’) | 创建Adadelta优化器 |
| **class tf.train.AdagradOptimizer** | 使用[Adagrad算法](http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf)的Optimizer |
| tf.train.AdagradOptimizer.\_\_init\_\_(learning\_rate,  initial\_accumulator\_value=0.1,  use\_locking=False, name=’Adagrad’) | 创建Adagrad优化器 |
| class tf.train.MomentumOptimizer | 使用[Momentum](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)算法的Optimizer |
| tf.train.MomentumOptimizer.\_\_init\_\_(learning\_rate,  momentum, use\_locking=False,  name=’Momentum’, use\_nesterov=False) | 创建momentum优化器 momentum：动量，一个tensor或者浮点值 |
| **class tf.train.AdamOptimizer** | 使用[Adam 算法](http://arxiv.org/abs/1412.6980)的Optimizer |
| tf.train.AdamOptimizer.\_\_init\_\_(learning\_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08, use\_locking=False, name=’Adam’) | 创建Adam优化器 |
| class tf.train.FtrlOptimizer | 使用[FTRL 算法](https://www.eecs.tufts.edu/~dsculley/papers/ad-click-prediction.pdf)的Optimizer |
| tf.train.FtrlOptimizer.\_\_init\_\_(learning\_rate,  learning\_rate\_power=-0.5,  initial\_accumulator\_value=0.1,  l1\_regularization\_strength=0.0,  l2\_regularization\_strength=0.0, use\_locking=False, name=’Ftrl’) | 创建FTRL算法优化器 |
| class tf.train.RMSPropOptimizer | 使用[RMSProp算法](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)的Optimizer |
| tf.train.RMSPropOptimizer.\_\_init\_\_(learning\_rate,  decay=0.9, momentum=0.0, epsilon=1e-10,  use\_locking=False, name=’RMSProp’) | 创建RMSProp算法优化器 |

▷ [*tf.train.AdamOptimizer*](http://arxiv.org/pdf/1412.6980.pdf)   
Adam 的基本运行方式，首先初始化：

m\_0 <- 0 (Initialize initial 1st moment vector)

v\_0 <- 0 (Initialize initial 2nd moment vector)

t <- 0 (Initialize timestep)

在[论文](http://arxiv.org/pdf/1412.6980.pdf)中的 section2 的末尾所描述了更新规则，该规则使用梯度g来更新变量：

t <- t + 1

lr\_t <- learning\_rate \* sqrt(1 - beta2^t) / (1 - beta1^t)

m\_t <- beta1 \* m\_{t-1} + (1 - beta1) \* g

v\_t <- beta2 \* v\_{t-1} + (1 - beta2) \* g \* g

variable <- variable - lr\_t \* m\_t / (sqrt(v\_t) + epsilon)

其中epsilon 的默认值1e-8可能对于大多数情况都不是一个合适的值。例如，当在ImageNet上训练一个 Inception network时比较好的选择为1.0或者0.1。   
需要注意的是，在稠密数据中即便g为0时， m\_t, v\_t 以及variable都将会更新。而在稀疏数据中，m\_t, v\_t 以及variable不被更新且值为零。

#### █ ****梯度计算与截断(Gradient Computation and Clipping)****

TensorFlow 提供了计算给定tf计算图的求导函数，并在图的基础上增加节点。优化器(optimizer )类可以自动的计算网络图的导数，但是优化器中的创建器(creators )或者专业的人员可以通过本节所述的函数调用更底层的方法。

| **操作** | **描述** |
| --- | --- |
| tf.gradients(ys, xs, grad\_ys=None, name=’gradients’,  colocate\_gradients\_with\_ops=False, gate\_gradients=False,  aggregation\_method=None) | 构建一个符号函数，计算ys关于xs中x的偏导的和， 返回xs中每个x对应的sum(dy/dx) |
| tf.stop\_gradient(input, name=None) | 停止计算梯度， 在EM算法、Boltzmann机等可能会使用到 |
| tf.clip\_by\_value(t, clip\_value\_min, clip\_value\_max, name=None) | 基于定义的min与max对tesor数据进行截断操作， 目的是为了应对梯度爆发或者梯度消失的情况 |
| tf.clip\_by\_norm(t, clip\_norm, axes=None, name=None) | 使用L2范式标准化tensor最大值为clip\_norm 返回 t \* clip\_norm / l2norm(t) |
| tf.clip\_by\_average\_norm(t, clip\_norm, name=None) | 使用平均L2范式规范tensor数据t， 并以clip\_norm为最大值 返回 t \* clip\_norm / l2norm\_avg(t) |
| tf.clip\_by\_global\_norm(t\_list,  clip\_norm, use\_norm=None, name=None) | 返回t\_list[i] \* clip\_norm / max(global\_norm, clip\_norm) 其中global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list])) |
| tf.global\_norm(t\_list, name=None) | 返回global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list])) |

#### █ ****退化学习率(Decaying the learning rate)****

| **操作** | **描述** |
| --- | --- |
| tf.train.exponential\_decay(learning\_rate, global\_step,  decay\_steps, decay\_rate, staircase=False, name=None) | 对学习率进行指数衰退 |

▷ tf.train.exponential\_decay

#该函数返回以下结果

decayed\_learning\_rate = learning\_rate \*

decay\_rate ^ (global\_step / decay\_steps)

##例： 以0.96为基数，每100000 步进行一次学习率的衰退

global\_step = tf.Variable(, trainable=False)

starter\_learning\_rate =

learning\_rate = tf.train.exponential\_decay(starter\_learning\_rate, global\_step,

100000, , staircase=)

# Passing global\_step to minimize() will increment it at each step.

learning\_step = (

tf.train.GradientDescentOptimizer(learning\_rate)

.minimize(...my loss..., global\_step=global\_step)

)

#### █ ****移动平均(Moving Averages)****

一些训练优化算法，比如GradientDescent 和Momentum 在优化过程中便可以使用到移动平均方法。使用移动平均常常可以较明显地改善结果。

| **操作** | **描述** |
| --- | --- |
| class tf.train.ExponentialMovingAverage | 将指数衰退加入到移动平均中 |
| tf.train.ExponentialMovingAverage.apply(var\_list=None) | 对var\_list变量保持移动平均 |
| tf.train.ExponentialMovingAverage.average\_name(var) | 返回var均值的变量名称 |
| tf.train.ExponentialMovingAverage.average(var) | 返回var均值变量 |
| tf.train.ExponentialMovingAverage.variables\_to\_restore(moving\_avg\_variables=None) | 返回用于保存的变量名称的映射 |

▷ tf.train.ExponentialMovingAverage

# Example usage when creating a training model:

# Create variables.

var0 = tf.Variable(...)

var1 = tf.Variable(...)

# ... use the variables to build a training model...

...

# Create an op that applies the optimizer. This is what we usually

# would use as a training op.

opt\_op = opt.minimize(my\_loss, [var0, var1])

# Create an ExponentialMovingAverage object

ema = tf.train.ExponentialMovingAverage(decay=0.9999)

# Create the shadow variables, and add ops to maintain moving averages

# of var0 and var1.

maintain\_averages\_op = ema.apply([var0, var1])

# Create an op that will update the moving averages after each training

# step. This is what we will use in place of the usual training op.

tf.control\_dependencies([opt\_op]):

training\_op = tf.group(maintain\_averages\_op)

...train the model by running training\_op...

#Example of restoring the shadow variable values:

# Create a Saver that loads variables from their saved shadow values.

shadow\_var0\_name = ema.average\_name(var0)

shadow\_var1\_name = ema.average\_name(var1)

saver = tf.train.Saver({shadow\_var0\_name: var0, shadow\_var1\_name: var1})

saver.restore(...checkpoint filename...)

# var0 and var1 now hold the moving average values

▷ tf.train.ExponentialMovingAverage.variables\_to\_restore

variables\_to\_restore = ema.variables\_to\_restore()

saver = tf.train.Saver(variables\_to\_restore)

#### █ ****协调器和队列运行器(Coordinator and QueueRunner)****

查看[queue](http://blog.csdn.net/lenbow/article/details/52181159)中，queue相关的内容，了解tensorflow中队列的运行方式。

| **操作** | **描述** |
| --- | --- |
| class tf.train.Coordinator | 线程的协调器 |
| tf.train.Coordinator.clear\_stop() | 清除停止标记 |
| tf.train.Coordinator.join(threads=None, stop\_grace\_period\_secs=120) | 等待线程终止 threads:一个threading.Threads的列表，启动的线程，将额外加入到registered的线程中 |
| tf.train.Coordinator.register\_thread(thread) | Register一个用于join的线程 |
| tf.train.Coordinator.request\_stop(ex=None) | 请求线程结束 |
| tf.train.Coordinator.should\_stop() | 检查是否被请求停止 |
| tf.train.Coordinator.stop\_on\_exception() | 上下文管理器，当一个例外出现时请求停止 |
| tf.train.Coordinator.wait\_for\_stop(timeout=None) | 等待Coordinator提示停止进程 |
| **class tf.train.QueueRunner** | 持有一个队列的入列操作列表，用于线程中运行 queue:一个队列 enqueue\_ops: 用于线程中运行的入列操作列表 |
| tf.train.QueueRunner.create\_threads(sess,  coord=None, daemon=False, start=False) | 创建运行入列操作的线程，返回一个线程列表 |
| tf.train.QueueRunner.from\_proto(queue\_runner\_def) | 返回由queue\_runner\_def创建的QueueRunner对象 |
| tf.train.add\_queue\_runner(qr, collection=’queue\_runners’) | 增加一个QueueRunner到graph的收集器(collection )中 |
| tf.train.start\_queue\_runners(sess=None, coord=None, daemon=True, start=True, collection=’queue\_runners’) | 启动所有graph收集到的队列运行器(queue runners) |

▷ class tf.train.Coordinator

#Coordinator的使用，用于多线程的协调

:

...

coord = Coordinator()

# Start a number of threads, passing the coordinator to each of them.

...start thread ..(coord, ...)

...start thread N...(coord, ...)

# Wait for all the threads to terminate, give them 10s grace period

coord.join(threads, stop\_grace\_period\_secs=)

except RuntimeException:

...one of the threads took more than s to stop after request\_stop()

...was called.

except Exception:

...exception that was passed to coord.request\_stop()

▷ tf.train.Coordinator.stop\_on\_exception()

coord.stop\_on\_exception():

# Any exception raised in the body of the with

# clause is reported to the coordinator before terminating

# the execution of the body.

...body...

:

...body...

exception Exception ex:

coord.request\_stop(ex)

#### █ ****布执行(Distributed execution)****

可以阅读[TensorFlow的分布式学习框架简介](http://blog.csdn.net/lenbow/article/details/52130565)查看更多tensorflow分布式细节。

| **操作** | **描述** |
| --- | --- |
| **class tf.train.Server** | 一个进程内的tensorflow服务，用于分布式训练 |
| tf.train.Server.**init**(server\_or\_cluster\_def,  job\_name=None, task\_index=None, protocol=None, config=None, start=True) | 创建一个新的服务，其中job\_name, task\_index,  和protocol为可选参数， 优先级高于server\_or\_cluster\_def中相关信息 server\_or\_cluster\_def : 为一个tf.train.ServerDef  或 tf.train.ClusterDef 协议(protocol)的buffer， 或者一个tf.train.ClusterSpec对象 |
| **tf.train.Server.create\_local\_server(config=None, start=True)** | 创建一个新的运行在本地主机的单进程集群 |
| **tf.train.Server.target** | 返回tf.Session所连接的目标服务器 |
| tf.train.Server.server\_def | 返回该服务的tf.train.ServerDef |
| **tf.train.Server.start()** | 开启服务 |
| tf.train.Server.join() | 阻塞直到服务已经关闭 |

| **#** |
| --- |
| class tf.train.Supervisor | 一个训练辅助器，用于checkpoints模型以及计算的summaries。该监视器只是一个小的外壳(wrapper),用于Coordinator, a Saver, 和a SessionManager周围 |
| **tf.train.Supervisor.\_\_init\_\_(graph=None, ready\_op=0, is\_chief=True, init\_op=0, init\_feed\_dict=None, local\_init\_op=0, logdir=None,  summary\_op=0, saver=0, global\_step=0,  save\_summaries\_secs=120, save\_model\_secs=600,  recovery\_wait\_secs=30, stop\_grace\_secs=120, checkpoint\_basename=’model.ckpt’, session\_manager=None, summary\_writer=0, init\_fn=None)** | 创建一个监视器Supervisor |
| tf.train.Supervisor.managed\_session(master=”, config=None, start\_standard\_services=True, close\_summary\_writer=True) | 返回一个管路session的上下文管理器 |
| tf.train.Supervisor.prepare\_or\_wait\_for\_session(master=”, config=None, wait\_for\_checkpoint=False, max\_wait\_secs=7200, start\_standard\_services=True) | 确保model已经准备好 |
| **tf.train.Supervisor.start\_standard\_services(sess)** | 为sess启动一个标准的服务 |
| tf.train.Supervisor.start\_queue\_runners(sess, queue\_runners=None) | 为QueueRunners启动一个线程，queue\_runners为一个QueueRunners列表 |
| tf.train.Supervisor.summary\_computed(sess, summary, global\_step=None) | 指示计算的summary |
| tf.train.Supervisor.stop(threads=None, close\_summary\_writer=True) | 停止服务以及协调器(coordinator),并没有关闭session |
| tf.train.Supervisor.request\_stop(ex=None) | 参考Coordinator.request\_stop() |
| tf.train.Supervisor.should\_stop() | 参考Coordinator.should\_stop() |
| tf.train.Supervisor.stop\_on\_exception() | 参考 Coordinator.stop\_on\_exception() |
| tf.train.Supervisor.Loop(timer\_interval\_secs, target, args=None, kwargs=None) | 开启一个循环器线程用于调用一个函数 每经过timer\_interval\_secs秒执行，target(\*args, \*\*kwargs) |
| tf.train.Supervisor.coord | 返回监督器(Supervisor)使用的协调器(Coordinator ) |

| **#** |
| --- |
| class tf.train.SessionManager | 训练的辅助器，用于从checkpoint恢复数据以及创建一个session |
| tf.train.SessionManager.\_\_init\_\_(local\_init\_op=None, ready\_op=None, graph=None, recovery\_wait\_secs=30) | 创建一个SessionManager |
| tf.train.SessionManager.prepare\_session(master, init\_op=None, saver=None, checkpoint\_dir=None, wait\_for\_checkpoint=False, max\_wait\_secs=7200, config=None, init\_feed\_dict=None, init\_fn=None) | 创建一个session，并确保model可以被使用 |
| tf.train.SessionManager.recover\_session(master, saver=None, checkpoint\_dir=None, wait\_for\_checkpoint=False, max\_wait\_secs=7200, config=None) | 创建一个session，如果可以的话，使用恢复方法创建 |
| tf.train.SessionManager.wait\_for\_session(master, config=None, max\_wait\_secs=inf) | 创建一个session，并等待model准备完成 |

| **#** |
| --- |
| **class tf.train.ClusterSpec** | 将一个集群表示为一系列“tasks”，并整合至“jobs”中 |
| tf.train.ClusterSpec.as\_cluster\_def() | 返回该cluster中一个tf.train.ClusterDef协议的buffer |
| tf.train.ClusterSpec.as\_dict() | 返回一个字典，由job名称对应于网络地址 |
| tf.train.ClusterSpec.job\_tasks(job\_name) | 返回一个给定的job对应的task列表 |
| tf.train.ClusterSpec.jobs | 返回该cluster的job名称列表 |
| **tf.train.replica\_device\_setter(ps\_tasks=0, ps\_device=’/job:ps’, worker\_device=’/job:worker’, merge\_devices=True, cluster=None, ps\_ops=None)** | 返回一个设备函数(device function)，以在建立一个副本graph的时候使用，设备函数(device function)用在with tf.device(device\_function)中 |

▷ tf.train.Server

server = tf.train.Server(...)

tf.Session(server.target):

▷ tf.train.Supervisor   
相关参数：   
ready\_op : 一维 字符串 tensor。该tensor是用过监视器在prepare\_or\_wait\_for\_session()计算，检查model是否准备好可以使用。如果准备好，将返回一个空阵列，如果为None，该model没有被检查。   
is\_chief : 如果为True，创建一个主监视器用于负责初始化与模型的恢复，若为False，则依赖主监视器。   
init\_op : 一个操作，用于模型不能恢复时的初始化操作。默认初始化所有操作   
local\_init\_op : 可被所有监视器运行的初始化操作。   
logdir : 设置log目录   
summary\_op : 一个操作(Operation )，返回Summary 和事件logs，需要设置 logdir   
saver : 一个Saver对象   
save\_summaries\_secs : 保存summaries的间隔秒数   
save\_model\_secs : 保存model的间隔秒数   
checkpoint\_basename : checkpoint保存的基本名称

* 使用在单进程中

tf.Graph().as\_default():

...add operations to the graph...

# Create a Supervisor that will checkpoint the model in '/tmp/mydir'.

sv = Supervisor(logdir='/tmp/mydir')

# Get a TensorFlow session managed by the supervisor.

sv.managed\_session(FLAGS.master) sess:

# Use the session to train the graph.

while sv.should\_stop():

sess.run(<my\_train\_op>)

# 在上下文管理器with sv.managed\_session()内，所有在graph的变量都被初始化。

# 或者说，一些服务器checkpoint相应模型并增加summaries至事件log中。

# 如果有例外发生，should\_stop()将返回True

* 使用在多副本运行情况中   
  要使用副本训练已经部署在集群上的相同程序，必须指定其中一个task为主要，该task处理 initialization, checkpoints, summaries, 和recovery相关事物。其他task依赖该task。

# Choose a task as the chief. This could be based on server\_def.task\_index,

# or job\_def.name, or job\_def.tasks. It's entirely up to the end user.

# But there can be only one \*chief\*.

is\_chief = (server\_def.task\_index == )

server = tf.train.Server(server\_def)

tf.Graph().as\_default():

...add operations to the graph...

# Create a Supervisor that uses log directory on a shared file system.

# Indicate if you are the 'chief'

sv = Supervisor(logdir='/shared\_directory/...', is\_chief=is\_chief)

# Get a Session in a TensorFlow server on the cluster.

sv.managed\_session(server.target) sess:

# Use the session to train the graph.

while sv.should\_stop():

sess.run(<my\_train\_op>)

如果有task崩溃或重启，managed\_session() 将检查是否Model被初始化。如果已经初始化，它只需要创建一个session并将其返回至正在训练的正常代码中。如果model需要被初始化，主task将对它进行重新初始化，而其他task将等待模型初始化完成。   
注意：该程序方法一样适用于单进程的work，该单进程标注自己为主要的便行

▷ supervisor中master的字符串形式   
无论运行在本机或者集群上，都可以使用以下值设定master flag：

* 定义为 ” ，要求一个进程内且没有使用RPC的session
* 定义为 ‘local’，要求一个使用基于RPC的主服务接口(“Master interface” )的session来运行tensorflow程序。更多细节可以查看 tf.train.Server.create\_local\_server()相关内容。
* 定义为 ‘grpc://hostname:port’，要求一个指定的RPC接口的session，同时运行内部进程的master接入远程的tensorflow workers。可用server.target返回该形式

▷ supervisor高级用法

* 启动额外的服务   
  managed\_session()启动了 Checkpoint 和Summary服务。如果需要运行更多的服务，可以在managed\_session()控制的模块中启动他们。

#例如： 开启一个线程用于打印loss. 设置每60秒该线程运行一次，我们使用sv.loop()

...

sv = Supervisor(logdir='/tmp/mydir')

sv.managed\_session(FLAGS.master) sess:

sv.loop(, print\_loss, (sess))

while sv.should\_stop():

sess.run(my\_train\_op)

* 启动更少的的服务   
  managed\_session() 启动了 “summary” 和 “checkpoint” 线程，这些线程通过构建器或者监督器默认自动创建了summary\_op 和saver操作。如果想运行自己的 summary 和checkpointing方法，关闭这些服务，通过传递None值给summary\_op 和saver参数。

在chief中每个step，创建summaries

# Create a Supervisor with no automatic summaries.

sv = Supervisor(logdir='/tmp/mydir', is\_chief=is\_chief, summary\_op=)

# As summary\_op was None, managed\_session() does not start the

# summary thread.

sv.managed\_session(FLAGS.master) sess:

step xrange(1000000):

sv.should\_stop():

break

is\_chief step % == :

# Create the summary every 100 chief steps.

sv.summary\_computed(sess, sess.run(my\_summary\_op))

:

# Train normally

sess.run(my\_train\_op)

▷ tf.train.Supervisor.managed\_session

train:

sv = tf.train.Supervisor(...)

sv.managed\_session(<master>) sess:

step xrange(..):

sv.should\_stop():

break

sess.run(<my training op>)

...do other things needed at each training step...

▷ tf.train.SessionManager

tf.Graph().as\_default():

...add operations to the graph...

# Create a SessionManager that will checkpoint the model in '/tmp/mydir'.

sm = SessionManager()

sess = sm.prepare\_session(master, init\_op, saver, checkpoint\_dir)

# Use the session to train the graph.

while :

sess.run(<my\_train\_op>)

#其中prepare\_session()初始化和恢复一个模型参数。

#另一个进程将等待model准备完成，代码如下

tf.Graph().as\_default():

...add operations to the graph...

# Create a SessionManager that will wait for the model to become ready.

sm = SessionManager()

sess = sm.wait\_for\_session(master)

# Use the session to train the graph.

while :

sess.run(<my\_train\_op>)

#wait\_for\_session()等待一个model被其他进程初始化

▷ tf.train.ClusterSpec   
一个tf.train.ClusterSpec表示一系列的进程，这些进程都参与分布式tensorflow的计算。每一个 tf.train.Server都在一个独有的集群中构建。   
创建一个具有两个jobs及其5个tasks的集群们需要定义从job名称列表到网络地址列表之间的映射。

cluster = tf.train.ClusterSpec({"worker": ["worker0.example.com:2222",

"worker1.example.com:2222",

"worker2.example.com:2222"],

: ["ps0.example.com:2222",

"ps1.example.com:2222"]})

▷ tf.train.replica\_device\_setter

# To build a cluster with two ps jobs on hosts ps0 and ps1, and 3 worker

# jobs on hosts worker0, worker1 and worker2.

cluster\_spec = {

: ["ps0:2222", "ps1:2222"],

"worker": ["worker0:2222", "worker1:2222", "worker2:2222"]}

tf.device(tf.replica\_device\_setter(cluster=cluster\_spec)):

# Build your graph

v1 = tf.Variable(...) # assigned to /job:ps/task:0

v2 = tf.Variable(...) # assigned to /job:ps/task:1

v3 = tf.Variable(...) # assigned to /job:ps/task:0

# Run compute

#### █ ****汇总操作(Summary Operations)****

我们可以在一个session中获取summary操作的输出，并将其传输到SummaryWriter以添加至一个事件记录文件中。

| **操作** | **描述** |
| --- | --- |
| tf.scalar\_summary(tags, values, collections=None, name=None) | 输出一个标量值的summary协议buffer tag的shape需要与values的相同，用来做summaries的tags，为字符串 |
| tf.image\_summary(tag, tensor, max\_images=3, collections=None, name=None) | 输出一个图像tensor的summary协议buffer |
| tf.audio\_summary(tag, tensor, sample\_rate, max\_outputs=3, collections=None, name=None) | 输出一个音频tensor的summary协议buffer |
| tf.histogram\_summary(tag, values, collections=None, name=None) | 输出一个直方图的summary协议buffer |
| tf.nn.zero\_fraction(value, name=None) | 返回0在value中的小数比例 |
| tf.merge\_summary(inputs, collections=None, name=None) | 合并summary |
| tf.merge\_all\_summaries(key=’summaries’) | 合并在默认graph中手机的summaries |

▶▶**将记录汇总写入文件中(Adding Summaries to Event Files)**

| **操作** | **描述** |
| --- | --- |
| class tf.train.SummaryWriter | 将summary协议buffer写入事件文件中 |
| tf.train.SummaryWriter.\_\_init\_\_(logdir, graph=None, max\_queue=10, flush\_secs=120, graph\_def=None) | 创建一个SummaryWriter实例以及新建一个事件文件 |
| tf.train.SummaryWriter.add\_summary(summary, global\_step=None) | 将一个summary添加到事件文件中 |
| tf.train.SummaryWriter.add\_session\_log(session\_log, global\_step=None) | 添加SessionLog到一个事件文件中 |
| tf.train.SummaryWriter.add\_event(event) | 添加一个事件到事件文件中 |
| tf.train.SummaryWriter.add\_graph(graph, global\_step=None, graph\_def=None) | 添加一个Graph到时间文件中 |
| tf.train.SummaryWriter.add\_run\_metadata(run\_metadata, tag, global\_step=None) | 为一个单一的session.run()调用添加一个元数据信息 |
| tf.train.SummaryWriter.flush() | 刷新时间文件到硬盘中 |
| tf.train.SummaryWriter.close() | 将事件问价写入硬盘中并关闭该文件 |
| tf.train.summary\_iterator(path) | 一个用于从时间文件中读取时间协议buffer的迭代器 |

▷ tf.train.SummaryWriter   
创建一个SummaryWriter 和事件文件。如果我们传递一个Graph进入该构建器中，它将被添加到事件文件当中，这一点与使用add\_graph()具有相同功能。   
TensorBoard 将从事件文件中提取该graph，并将其显示。所以我们能直观地看到我们建立的graph。我们通常从我们启动的session中传递graph：

...create a graph...

# Launch the graph in a session.

sess = tf.Session()

# Create a summary writer, add the 'graph' to the event file.

writer = tf.train.SummaryWriter(<some-directory>, sess.graph)

▷ tf.train.summary\_iterator

#打印时间文件中的内容

e tf.train.summary\_iterator(path to events file):

print(e)

#打印指定的summary值

# This example supposes that the events file contains summaries with a

# summary value tag 'loss'. These could have been added by calling

# `add\_summary()`, passing the output of a scalar summary op created with

# with: `tf.scalar\_summary(['loss'], loss\_tensor)`.

e tf.train.summary\_iterator(path to events file):

v e.summary.value:

v.tag == 'loss':

print(v.simple\_value)

#### █ ****训练的通用函数及其他(Training utilities)****

| **操作** | **描述** |
| --- | --- |
| tf.train.global\_step(sess, global\_step\_tensor) | 一个用于获取全局step的小辅助器 |
| tf.train.write\_graph(graph\_def, logdir, name, as\_text=True) | 将一个graph proto写入一个文件中 |
| class tf.train.LooperThread | 可重复地执行代码的线程 |
| tf.train.LooperThread.**init**(coord, timer\_interval\_secs, target=None, args=None, kwargs=None) | 创建一个LooperThread |
| tf.train.LooperThread.is\_alive() | 返回是否该线程是活跃的 |
| tf.train.LooperThread.join(timeout=None) | 等待线程结束 |
| tf.train.LooperThread.loop(coord, timer\_interval\_secs, target, args=None, kwargs=None) | 启动一个LooperThread，用于周期地调用某个函数 调用函数target(args) |
| tf.py\_func(func, inp, Tout, stateful=True, name=None) | 将python函数包装成tf中操作节点 |

▷ tf.train.global\_step

# Creates a variable to hold the global\_step.

global\_step\_tensor = tf.Variable(, trainable=False, name='global\_step')

# Creates a session.

sess = tf.Session()

# Initializes the variable.

sess.run(global\_step\_tensor.initializer)

print('global\_step: %s' % tf.train.global\_step(sess, global\_step\_tensor))

global\_step:

▷ tf.train.write\_graph

v = tf.Variable(, name='my\_variable')

sess = tf.Session()

tf.train.write\_graph(sess.graph\_def, '/tmp/my-model', 'train.pbtxt')

▷ tf.py\_func

#tf.py\_func(func, inp, Tout, stateful=True, name=None)

#func：为一个python函数

#inp：为输入函数的参数，Tensor列表

#Tout： 指定func返回的输出的数据类型，是一个列表

my\_func:

# x will be a numpy array with the contents of the placeholder below

return np.sinh(x)

inp = tf.placeholder(tf.float32, [...])

y = py\_func(my\_func, [inp], [tf.float32])

### 2.2 测试 (Testing)

TensorFlow 提供了一个方便的继承unittest.TestCase类的方法，该类增加有关TensorFlow 测试的方法。如下例子：

import tensorflow tf

class SquareTest(tf.test.TestCase):

testSquare(self):

self.test\_session():

x = tf.square([, ])

self.assertAllEqual(x.eval(), [, ])

\_\_name\_\_ == '\_\_main\_\_':

tf.test.main()

#### █ ****共用(Utilities)****

| **操作** | **描述** |
| --- | --- |
| tf.test.main() | 运行所有的单元测试 |
| tf.test.assert\_equal\_graph\_def(actual, expected) | 断言 两个GraphDefs 是否几乎一样 |
| tf.test.get\_temp\_dir() | 返回测试期间使用的临时目录 |
| tf.test.is\_built\_with\_cuda() | 返回是否Tensorflow支持CUDA(GPU)的build |

#### █ ****梯度检查(Gradient checking)****

可对比compute\_gradient 和compute\_gradient\_error函数的用法

| **操作** | **描述** |
| --- | --- |
| tf.test.compute\_gradient(x, x\_shape, y, y\_shape, x\_init\_value=None, delta=0.001, init\_targets=None) | 计算并返回理论的和数值的Jacobian矩阵 |
| tf.test.compute\_gradient\_error(x, x\_shape, y, y\_shape, x\_init\_value=None, delta=0.001, init\_targets=None) | 计算梯度的error。在计算所得的与数值估计的Jacobian中 为dy/dx计算最大的error |

**相关链接：**

[1] 安装Tensorflow（[Linux](http://lib.csdn.net/base/linux) ubuntu）<http://blog.csdn.net/lenbow/article/details/51203526>   
[2] ubuntu下CUDA编译的GCC降级安装<http://blog.csdn.net/lenbow/article/details/51596706>   
[3] ubuntu手动安装最新Nvidia显卡驱动<http://blog.csdn.net/lenbow/article/details/51683783>   
[4] Tensorflow的CUDA升级，以及相关配置<http://blog.csdn.net/lenbow/article/details/52118116>   
[5] 基于gensim的Doc2Vec简析<http://blog.csdn.net/lenbow/article/details/52120230>   
[6] TensorFlow的分布式学习框架简介<http://blog.csdn.net/lenbow/article/details/52130565>   
[7] Tensorflow一些常用基本概念与函数（1）<http://blog.csdn.net/lenbow/article/details/52152766>   
[8] Tensorflow一些常用基本概念与函数（2）<http://blog.csdn.net/lenbow/article/details/52181159>   
[9] Tensorflow一些常用基本概念与函数（3）<http://blog.csdn.net/lenbow/article/details/52213105>

Measure

Measure

免费注册印象笔记帐户以保存文章，以后随时在手机、平板或电脑上阅读。

[创建帐户](chrome-extension://iooicodkiihhpojmmeghjclgihfjdjhj/front/in_isolation/reformat.html)