# Tenserflow

## class Estimator

Estimator 类，用来训练和验证 TensorFlow 模型。   
Estimator 对象包含了一个模型 model\_fn，这个模型给定输入和参数，会返回训练、验证或者预测等所需要的操作节点。   
所有的输出（检查点、事件文件等）会写入到 model\_dir，或者其子文件夹中。如果 model\_dir 为空，则默认为临时目录。   
config 参数为 tf.estimator.RunConfig 对象，包含了执行环境的信息。如果没有传递 config，则它会被 Estimator 实例化，使用的是默认配置。   
params 包含了超参数。Estimator 只传递超参数，不会检查超参数，因此 params 的结构完全取决于开发者。   
Estimator 的所有方法都不能被子类覆盖（它的构造方法强制决定的）。子类应该使用 model\_fn来配置母类，或者增添方法来实现特殊的功能。   
Estimator 不支持 Eager Execution（eager execution能够使用Python 的debug工具、数据结构与控制流。并且无需使用placeholder、session，计算结果能够立即得出）。

类内方法

1、\_\_init\_\_(self, model\_fn, model\_dir=None, config=None, params=None, warm\_start\_from=None)

构造一个 Estimator 的实例.。   
参数：

model\_fn: 模型函数。函数的格式如下：   
参数：   
1、features: 这是 input\_fn 返回的第一项（input\_fn 是 train, evaluate 和 predict 的参数）。类型应该是单一的 Tensor 或者 dict。   
2、labels: 这是 input\_fn 返回的第二项。类型应该是单一的 Tensor 或者 dict。如果 mode 为 ModeKeys.PREDICT，则会默认为 labels=None。如果 model\_fn 不接受 mode，model\_fn 应该仍然可以处理 labels=None。   
3、mode: 可选。指定是训练、验证还是测试。参见 ModeKeys。   
4、params: 可选，超参数的 dict。 可以从超参数调整中配置 Estimators。   
5、config: 可选，配置。如果没有传则为默认值。可以根据 num\_ps\_replicas 或 model\_dir 等配置更新 model\_fn。   
返回：   
EstimatorSpec

model\_dir: 保存模型参数、图等的地址，也可以用来将路径中的检查点加载至 estimator 中来继续训练之前保存的模型。如果是 PathLike， 那么路径就固定为它了。如果是 None，那么 config 中的 model\_dir 会被使用（如果设置了的话），如果两个都设置了，那么必须相同；如果两个都是 None，则会使用临时目录。

config: 配置类。

params: 超参数的dict，会被传递到 model\_fn。keys 是参数的名称，values 是基本 python 类型。

warm\_start\_from: 可选，字符串，检查点的文件路径，用来指示从哪里开始热启动。或者是 tf.estimator.WarmStartSettings 类来全部配置热启动。如果是字符串路径，则所有的变量都是热启动，并且需要 Tensor 和词汇的名字都没有变。

异常：

RuntimeError： 开启了 eager execution

ValueError：model\_fn 的参数与 params 不匹配

ValueError：这个函数被 Estimator 的子类所覆盖

2、train(self, input\_fn, hooks=None, steps=None, max\_steps=None, saving\_listeners=None)

根据所给数据 input\_fn， 对模型进行训练。   
参数：

input\_fn：一个函数，提供由小 batches 组成的数据， 供训练使用。必须返回以下之一：   
1、一个 'tf.data.Dataset'对象：Dataset的输出必须是一个元组 (features, labels)，元组要求如下。   
2、一个元组 (features, labels)：features 是一个 Tensor 或者一个字典（特征名为 Tensor），labels是一个 Tensor 或者一个字典（特征名为 Tensor）。features 和 labels 都被 model\_fn 所使用，应该符合 model\_fn 输入的要求。

hooks：SessionRunHook 子类实例的列表。用于在训练循环内部执行。

steps：模型训练的步数。如果是 None， 则一直训练，直到input\_fn 抛出了超过界限的异常。steps 是递进式进行的。如果执行了两次训练（steps=10），则总共训练了 20 次。如果中途抛出了越界异常，则训练在 20 次之前就会停止。如果你不想递进式进行，请换为设置 max\_steps。如果设置了 steps，则 max\_steps 必须是 None。

max\_steps：模型训练的最大步数。如果为 None，则一直训练，直到input\_fn 抛出了超过界限的异常。如果设置了 max\_steps， 则 steps 必须是 None。如果中途抛出了越界异常，则训练在 max\_steps 次之前就会停止。执行两次 train(steps=100) 意味着 200 次训练；但是，执行两次 train(max\_steps=100) 意味着第二次执行不会进行任何训练，因为第一次执行已经做完了所有的 100 次。

saving\_listeners：CheckpointSaverListener 对象的列表。用于在保存检查点之前或之后立即执行的回调函数。

返回：   
self：为了链接下去。   
异常：   
ValueError：steps 和 max\_steps 都不是 None   
ValueError：steps 或 max\_steps <= 0

3、evaluate(self, input\_fn, steps=None, hooks=None, checkpoint\_path=None, name=None)

根据所给数据 input\_fn， 对模型进行验证。   
对于每一步，执行 input\_fn（返回数据的一个 batch）。   
一直进行验证，直到：

steps 个 batches 进行完毕，或者

input\_fn 抛出了越界异常（OutOfRangeError 或 StopIteration）

参数：

input\_fn：一个函数，构造了验证所需的输入数据，必须返回以下之一：   
1、一个 'tf.data.Dataset'对象：Dataset的输出必须是一个元组 (features, labels)，元组要求如下。   
2、一个元组 (features, labels)：features 是一个 Tensor 或者一个字典（特征名为 Tensor），labels是一个 Tensor 或者一个字典（特征名为 Tensor）。features 和 labels 都被 model\_fn 所使用，应该符合 model\_fn 输入的要求。

steps：模型验证的步数。如果是 None， 则一直验证，直到input\_fn 抛出了超过界限的异常。

hooks：SessionRunHook 子类实例的列表。用于在验证内部执行。

checkpoint\_path： 用于验证的检查点路径。如果是 None， 则使用 model\_dir 中最新的检查点。

name：验证的名字。使用者可以针对不同的数据集运行多个验证操作，比如训练集 vs 测试集。不同验证的结果被保存在不同的文件夹中，且分别出现在 tensorboard 中。

返回：   
返回一个字典，包 标、global\_step（包含验证进行的全局步数）   
异常：   
ValueError：如果 step 小于等于0   
ValueError：如果 model\_dir 指定的模型没有被训练，或者指定的 checkpoint\_path 为空。

4、predict(self, input\_fn, predict\_keys=None, hooks=None, checkpoint\_path=None, yield\_single\_examples=True)

对给出的特征进行预测   
参数：

input\_fn：一个函数，构造特征。预测一直进行下去，直到 input\_fn 抛出了越界异常（OutOfRangeError 或 StopIteration）。函数必须返回以下之一：   
1、一个 'tf.data.Dataset'对象：Dataset的输出必须是一个元组 (features, labels)，元组要求如下。   
2、features：一个 Tensor 或者一个字典（特征名为 Tensor）。features 被 model\_fn 所使用，应该符合 model\_fn 输入的要求。   
3、一个元组，其中第一项为 features。

predict\_keys：字符串列表，要预测的键值。当 EstimatorSpec.predictions 是一个 dict 时使用。如果使用了 predict\_keys， 那么剩下的预测值会从字典中过滤掉。如果是 None，则返回全部。

hooks：SessionRunHook 子类实例的列表。用于在预测内部回调。

checkpoint\_path： 用于预测的检查点路径。如果是 None， 则使用 model\_dir 中最新的检查点。

yield\_single\_examples：If False, yield the whole batch as returned by the model\_fn instead of decomposing the batch into individual elements. This is useful if model\_fn returns some tensors whose first dimension is not equal to the batch size.

返回：   
predictions tensors 的值   
异常：   
ValueError：model\_dir 中找不到训练好的模型。   
ValueError：预测值的 batch 长度不同，且 yield\_single\_examples 为 True。   
ValueError：predict\_keys 和 predictions 之间有冲突。例如，predict\_keys 不是 None，但是 EstimatorSpec.predictions 不是一个 dict。

## tf.cast

cast(x, dtype, name=None)

将x的数据格式转化成dtype.例如，原来x的数据格式是bool，

那么将其转化成float以后，就能够将其转化成0和1的序列。反之也可以

a = tf.Variable([1,0,0,1,1])

b = tf.cast(a,dtype=tf.bool)

sess = tf.Session()

sess.run(tf.initialize\_all\_variables())

print(sess.run(b))

#[ True False False True True]

## tf.argmax

tf.argmax(vector, 1)：返回的是vector中的最大值的索引号，如果vector是一个向量，那就返回一个值，如果是一个矩阵，那就返回一个向量，这个向量的每一个维度都是相对应矩阵行的最大值元素的索引号。

import tensorflow as tf

import numpy as np

A = [[1,3,4,5,6]]

B = [[1,3,4], [2,4,1]]

with tf.Session() as sess:

print(sess.run(tf.argmax(A, 1)))

print(sess.run(tf.argmax(B, 1)))

输出：

[4]

[2 1]

## tf.layers.dropout

dropout：一种防止神经网络过拟合的手段。

随机的拿掉网络中的部分神经元，从而减小对W权重的依赖，以达到减小过拟合的效果。

注意：dropout只能用在训练中，测试的时候不能dropout，要用完整的网络测试哦。

## tf.contrib.layers.flatten

tf.contrib.layers.flatten(P)这个函数就是把P保留第一个维度，把第一个维度包含的每一子张量展开成一个行向量，返回张量是一个二维的， shape=（batch\_size，….）,一般用于卷积神经网络全链接层前的预处理

## Estimator、EstimatorSpec、DataSet 和Experiment

这几个对象分别封装了：

Estimator：封装网络模型和参数

EstimatorSpec：具体的模型

DataSet：封装训练数据、评估数据、自动迭代器

Experiment：封装了一个实验，参数包括Estimator估算器，输入数据等。

关于这几个模型的一个介绍的博客：https://zhuanlan.zhihu.com/p/29073452

github上一个代码例程：https://gist.github.com/peterroelants/9956ec93a07ca4e9ba5bc415b014bcca?spm=5176.100239.blogcont194302.52.iPqDIt

——————————————————————————————————————————

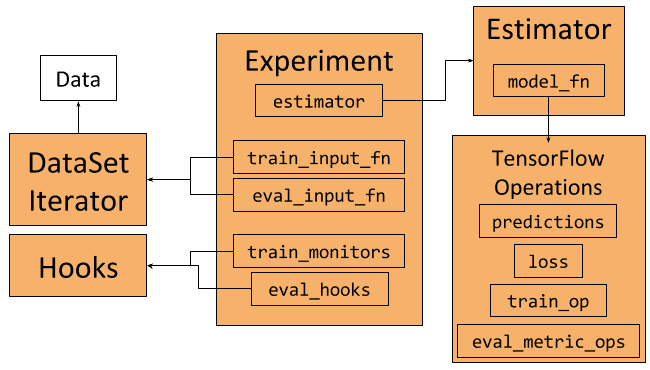
关于这几个对象的关系：

Experiment：

（1）——>Estimator——>EstimatorSpec

（2）——>DataSet

在这篇文章中，我们将看到一个使用了这些最新的高级构件的例子，包括[Estimator](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator?spm=5176.100239.blogcont194302.16.iPqDIt" \t "_blank)（估算器）、[Experiment](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/learn/Experiment?spm=5176.100239.blogcont194302.17.iPqDIt" \t "_blank)（实验）和[Dataset](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/data/Dataset?spm=5176.100239.blogcont194302.18.iPqDIt" \t "_blank)（数据集）。值得注意的是，你可以独立地使用Experiment和Dataset。我在这里假设你已经了解TensorFlow的基础知识；如果没有的话，那么TensorFlow官网上提供的[教程](http://link.zhihu.com/?target=https://www.tensorflow.org/get_started/?spm=5176.100239.blogcont194302.19.iPqDIt" \t "_blank)值得学习。

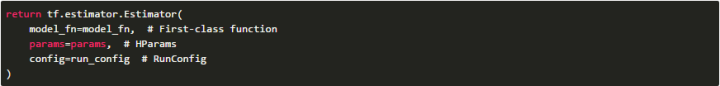


Experiment、Estimator和DataSet框架以及它们之间的交互。

我们在本文中将使用[MNIST](http://link.zhihu.com/?target=https://en.wikipedia.org/wiki/MNIST_database?spm=5176.100239.blogcont194302.21.iPqDIt" \t "_blank)作为数据集。这是一个使用起来很简单的数据集，可以从TensorFlow官网获取到。你可以在这个[gist](http://link.zhihu.com/?target=https://gist.github.com/peterroelants/9956ec93a07ca4e9ba5bc415b014bcca?spm=5176.100239.blogcont194302.22.iPqDIt" \t "_blank)中找到完整的代码示例。使用这些框架的其中一个好处是，我们不需要直接处理[图](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/Graph?spm=5176.100239.blogcont194302.23.iPqDIt" \t "_blank)和[会话](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/Session?spm=5176.100239.blogcont194302.24.iPqDIt" \t "_blank)。

Estimator（估算器）类

[Estimator](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator?spm=5176.100239.blogcont194302.25.iPqDIt" \t "_blank)类代表了一个模型，以及如何对这个模型进行训练和评估。我们可以像下面这段代码创建一个Estimator：



要创建Estimator，需要传入一个模型函数、一组参数和一些配置。

传入的\*\*参数\*\*应该是模型超参数的一个集合。这可以是一个dictionary，但是我们将在这个例子中把它表示成一个HParams对象，就像namedtuple一样。

传入的\*\*配置\*\*用于指定如何运行训练和评估，以及在哪里存储结果。这个配置是一个RunConfig对象，该对象会把模型运行环境相关的信息告诉Estimator。

模型函数是一个Python函数，它根据给定的输入构建模型。

模型函数

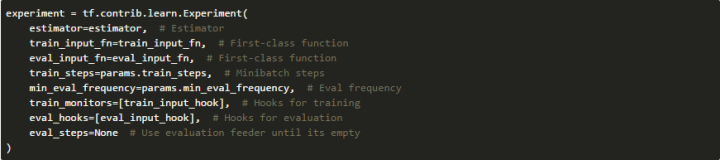
模型函数是一个Python函数，并作为[一级函数](http://link.zhihu.com/?target=https://en.wikipedia.org/wiki/First-class_function?spm=5176.100239.blogcont194302.26.iPqDIt" \t "_blank)传递给Estimator。稍后我们会看到，TensorFlow在其他地方也使用了一级函数。将模型表示为一个函数的好处是可以通过实例化函数来多次创建模型。模型可以在训练过程中用不同的输入重新创建，例如，在训练过程中运行验证测试。

模型函数把\*\*输入特征\*\*作为参数，将相应的\*\*标签\*\*作为张量。它也能以某种方式来告知用户模型是在训练、评估或是在执行推理。模型函数的最后一个参数是\*\*超参数\*\*集合，它们与传递给Estimator的超参数集合相同。模型函数返回一个\*\*EstimatorSpec\*\*对象，该对象定义了一个完整的模型。

EstimatorSpec对象用于对操作进行预测、损失、训练和评估，因此，它定义了一个用于训练、评估和推理的完整的模型图。由于EstimatorSpec只可用于常规的TensorFlow操作，因此，我们可以使用像[TF-Slim](http://link.zhihu.com/?target=https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim?spm=5176.100239.blogcont194302.27.iPqDIt" \t "_blank)这样的框架来定义模型。

Experiment（实验）类

[Experiment](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/learn/Experiment?spm=5176.100239.blogcont194302.28.iPqDIt" \t "_blank)类定义了如何训练模型，它与Estimator完美地集成在一起。我们可以像如下代码创建一个Experiment对象：



以下几种情况会把Experiment对象作为输入：

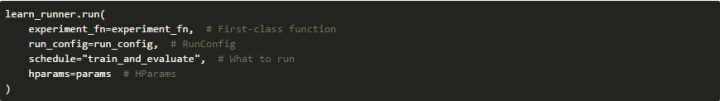
一个\*\*estimator\*\*（例如我们上面定义的）。

作为一级函数\*\*训练和评估数据\*\*。这里使用了与前面提到的模型函数相同的概念。如果需要的话，通过传入函数而不是操作，可以重新创建输入图。稍后我们还会谈到这个。

[训练和评估hook（钩子）](http://link.zhihu.com/?target=https://www.tensorflow.org/api_guides/python/train?spm=5176.100239.blogcont194302.29.iPqDIt%23Training_Hooks" \t "_blank)。钩子可用于保存或监视特定的内容，或者在图或会话中设置某些操作。例如，我们将其传入到操作中，帮助初始化数据加载器。

描述需要训练多久以及何时评估的各种参数。

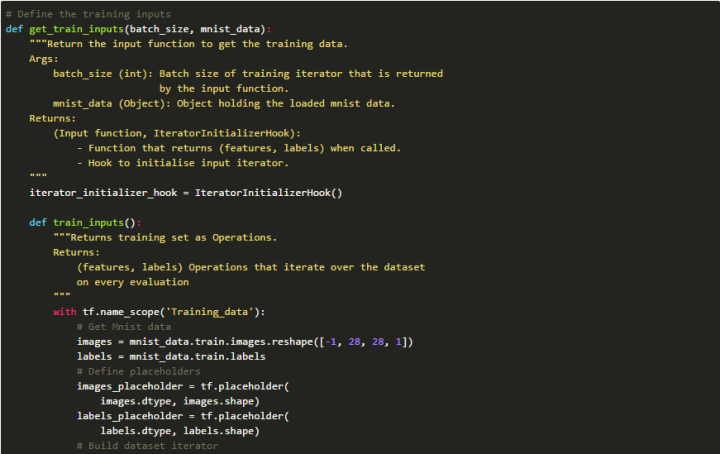
一旦定义了experiment，我们就可以像下面这段代码那样使用[learn\_runner.run](http://link.zhihu.com/?target=http://tf.contrib.learn.learn_runner.run/?spm=5176.100239.blogcont194302.30.iPqDIt" \t "_blank)来运行它训练和评估模型：

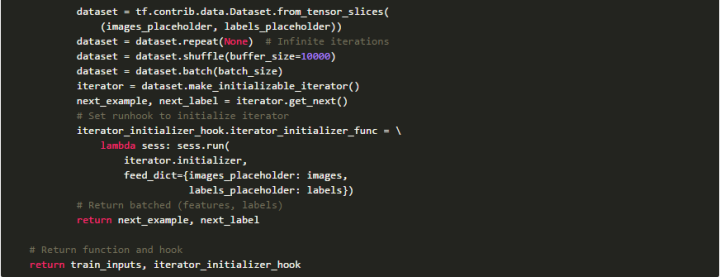


与模型函数和数据函数一样，learn\_runner将一个创建experiment的函数作为参数传入。

Dataset（数据集）类

我们将使用[Dataset](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/data/Dataset?spm=5176.100239.blogcont194302.31.iPqDIt" \t "_blank)类和相应的[Iterator](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/data/Iterator?spm=5176.100239.blogcont194302.32.iPqDIt" \t "_blank)来表示数据的训练和评估，以及创建在训练过程中迭代数据的数据馈送器。 在本示例中，我们将使用在Tensorflow中可用的[MNIST](http://link.zhihu.com/?target=https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/tutorials/mnist?spm=5176.100239.blogcont194302.33.iPqDIt" \t "_blank)数据，并为其构建一个Dataset包装。例如，我们将把训练输入数据表示为：

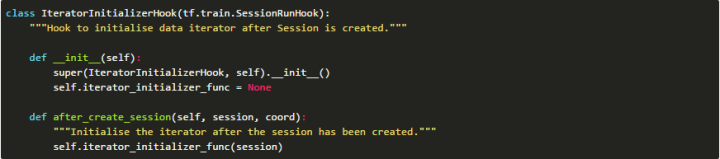




调用这个get\_train\_inputs将返回一个一级函数，用于在TensorFlow图中创建数据加载操作，以及返回一个用于初始化迭代器的Hook。

本示例中使用的MNIST数据最初是一个[Numpy](http://link.zhihu.com/?target=https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html?spm=5176.100239.blogcont194302.34.iPqDIt" \t "_blank)数组。我们创建了一个[占位符](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/placeholder?spm=5176.100239.blogcont194302.35.iPqDIt" \t "_blank)张量来获取数据；使用占位符的目的是为了避免数据的复制。接下来，我们在from\_tensor\_slices的帮助下创建一个切片数据集。我们要确保该数据集可以运行无限次数，并且数据被重新洗牌并放入指定大小的批次中。

要迭代数据，就需要从数据集中创建一个迭代器。由于我们正在使用占位符，因此需要使用NumPy数据在相关会话中[对占位符进行初始化](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/contrib/data/Dataset?spm=5176.100239.blogcont194302.36.iPqDIt%23make_initializable_iterator" \t "_blank)。可以通过创建一个可初始化的迭代器来实现这个。在创建图的时候，将创建一个自定义的IteratorInitializerHook对象来初始化迭代器：



IteratorInitializerHook继承自[SessionRunHook](http://link.zhihu.com/?target=https://www.tensorflow.org/api_docs/python/tf/train/SessionRunHook?spm=5176.100239.blogcont194302.37.iPqDIt" \t "_blank)。这个钩子将在相关会话创建后立即调用after\_create\_session，并使用正确的数据初始化占位符。这个钩子由我们的get\_train\_inputs函数返回，并在创建时传递给Experiment对象。

train\_inputs函数返回的数据加载操作是TensorFlow的操作，该操作每次评估时都会返回一个新的批处理。

运行代码

现在，我们已经定义了所有内容，可以使用下面这个命令运行代码了：

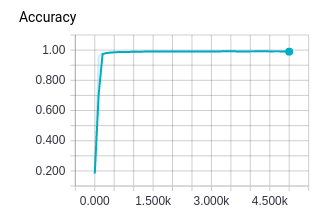
IMG_263

如果不传入参数，它将使用文件开头的默认标志来确定数据和模型保存的位置。

在训练过程中，在终端上会输出这段时间内的全局步骤、损失和准确性等信息。除此之外，Experiment和Estimator框架将记录TensorBoard可视化的某些统计信息。如果我们运行这个命令：

IMG_264

在训练过程中，在终端上会输出这段时间内的全局步骤、损失和准确性等信息。除此之外，Experiment和Estimator框架将记录TensorBoard可视化的某些统计信息。如果我们运行这个命令：



TensorBoard可视化中的评估准确度

## 坑

### 版本坑

问题一：TypeError: Expected int32, got list containing Tensors of type ‘\_Message’ instead.

tensorflow 函数tf.cocat([fw,bw],2)出错:

Expected int32, got list containing Tensors of type ‘\_Message’ inst   
查看原因是11版本的函数形式为:tf.concat(2,[fw,bw]),即应把串联的维度与串联值位置调换即可.

问题二：Input ‘split\_dim’ of ‘Split’ Op has type float32 that does not match expected type of int32

This is because in Tensorflow versions < 0.12.0 the split function takes the arguments as:

x = tf.split(0, n\_steps, x) # tf.split(axis, num\_or\_size\_splits, value)   
The tutorial you are working from was written for versions > 0.12.0, which has been changed to be consistent with Numpy’s split syntax:

x = tf.split(x, n\_steps, 0) # tf.split(value, num\_or\_size\_splits, axis)

问题三：TypeError: concat() got an unexpected keyword argument ‘axis’

tf.concat(concat\_dim=axis, values=inputs, name=name)   
修改为： tf.concat(inputs,1,name=name)

问题四：ValueError: ‘size’ must be a 1-D Tensor of 2 elements

img = tf.image.resize\_images(img, new\_shape[0], new\_shape[1])   
改为   
img = tf.image.resize\_images(img, new\_shape)

问题五： ‘module’ object has no attribute ‘pack’

因为TF后面的版本修改了这个函数的名称，把 tf.pack 改为 tf.stack。

问题六：The value of a feed cannot be a tf.Tensor object. Acceptable feed values include Python scalars, strings, lists, or numpy ndarrays

数据集是feed输入的，feed的数据格式是有要求的   
解决：img,label = sess.run[img,label],用返回值

问题七：AttributeError: ‘module’ object has no attribute ‘per\_image\_whitening’

For anyone else who has this problem, per\_image\_whitening was replaced by per\_image\_standardization in v0.12.

问题八：AttributeError: ‘module’ object has no attribute ‘image\_summary’

tf.image\_summary should be renamed to tf.summary.image;

问题九：AttributeError: ‘module’ object has no attribute ‘mul’

tf.mul(a,b) 这里的矩阵a和矩阵b的shape必须相等 tf.mul()是矩阵的element-wise相乘（即Hadamard乘积）   
tf.matmul(a,b) 这里的矩阵a和矩阵b的shape应是a的行数对等与b的列数，tf.matmul()是矩阵的一般相乘。   
解决：[tf.mul，tf.sub ] 和 [tf.neg] 不再使用，改为 [tf.multiply]，[tf.subtract] 和 [tf.negative]。

问题十：AttributeError: ‘module’ object has no attribute ‘scalar\_summary’

修改为:tf.summary.scalar(‘batch\_loss’, loss)原因:新版本做了调整 …

## 读数据

filename\_queue = tf.train.string\_input\_producer(["records/train.tfrecords"])

reader = tf.TFRecordReader()  
\_, serialized = reader.read(filename\_queue)

features = tf.parse\_single\_example(serialized,  
 features={  
 "context": tf.FixedLenFeature([], dtype=tf.string),  
 "tag": tf.FixedLenFeature([], dtype=tf.string),  
 "sequence\_length": tf.FixedLenFeature([], dtype=tf.int64)  
 })  
sequence\_length = tf.cast(features['sequence\_length'], tf.int32)  
context = tf.decode\_raw(features['context'], tf.int64)  
tag = tf.decode\_raw(features['tag'], tf.int64)

//batch化

sequence\_lengths, (contexts, tags) = training.bucket\_by\_sequence\_length(input\_length=sequence\_length,  
 tensors=[context, tag],  
 batch\_size=batch\_size,  
 bucket\_boundaries=BUCKET\_BOUNDARIES,  
 dynamic\_pad=True,  
 capacity=capacity)

## 读数据

<http://blog.csdn.net/lujiandong1/article/details/53376802>

Tensorflow数据读取有三种方式：

Preloaded data: 预加载数据

Feeding: Python产生数据，再把数据喂给后端。

Reading from file: 从文件中直接读取

这三种有读取方式有什么区别呢？ 我们首先要知道TensorFlow(TF)是怎么样工作的。

TF的核心是用C++写的，这样的好处是运行快，缺点是调用不灵活。而Python恰好相反，所以结合两种语言的优势。涉及计算的核心算子和运行框架是用C++写的，并提供API给Python。Python调用这些API，设计训练模型(Graph)，再将设计好的Graph给后端去执行。简而言之，Python的角色是Design，C++是Run。

一、预加载数据：

[python] [view plain](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "view plain) [copy](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "copy) [print](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "print)[?](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "?)

import tensorflow as tf

# 设计Graph

x1 = tf.constant([2, 3, 4])

x2 = tf.constant([4, 0, 1])

y = tf.add(x1, x2)

# 打开一个session --> 计算y

with tf.Session() as sess:

    print sess.run(y)

import tensorflow as tf

# 设计Graph

x1 = tf.constant([2, 3, 4])

x2 = tf.constant([4, 0, 1])

y = tf.add(x1, x2)

# 打开一个session --> 计算y

with tf.Session() as sess:

print sess.run(y)

二、python产生数据，再将数据喂给后端

[python] [view plain](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "view plain) [copy](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "copy) [print](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "print)[?](http://blog.csdn.net/lujiandong1/article/details/53376802" \o "?)

import tensorflow as tf

# 设计Graph

x1 = tf.placeholder(tf.int16)

x2 = tf.placeholder(tf.int16)

y = tf.add(x1, x2)

# 用Python产生数据

li1 = [2, 3, 4]

li2 = [4, 0, 1]

# 打开一个session --> 喂数据 --> 计算y

with tf.Session() as sess:

    print sess.run(y, feed\_dict={x1: li1, x2: li2})

import tensorflow as tf

# 设计Graph

x1 = tf.placeholder(tf.int16)

x2 = tf.placeholder(tf.int16)

y = tf.add(x1, x2)

# 用Python产生数据

li1 = [2, 3, 4]

li2 = [4, 0, 1]

# 打开一个session --> 喂数据 --> 计算y

with tf.Session() as sess:

print sess.run(y, feed\_dict={x1: li1, x2: li2})

说明：在这里x1, x2只是占位符，没有具体的值，那么运行的时候去哪取值呢？这时候就要用到sess.run()中的feed\_dict参数，将Python产生的数据喂给后端，并计算y。  
这两种方案的缺点：

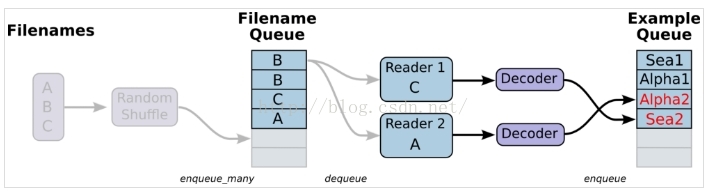
1、预加载：将数据直接内嵌到Graph中，再把Graph传入Session中运行。当数据量比较大时，Graph的传输会遇到效率问题。

2、用占位符替代数据，待运行的时候填充数据。

前两种方法很方便，但是遇到大型数据的时候就会很吃力，即使是Feeding，中间环节的增加也是不小的开销，比如数据类型转换等等。最优的方案就是在Graph定义好文件读取的方法，让TF自己去从文件中读取数据，并解码成可使用的样本集。

三、从文件中读取，简单来说就是将数据读取模块的图搭好

## TFrecorder

除了直接读取数据文件，比如csv和bin文件，tensorflow还可以建立一种自有格式的数据文件，称之为tfrecorder，这种文件储存类似于字典，调用方便，可以直接包含标签集。

TFRecords其实是一种二进制文件，虽然它不如其他格式好理解，但是它能更好的利用内存，更方便复制和移动，并且不需要单独的标签文件（等会儿就知道为什么了）… …总而言之，这样的文件格式好处多多，所以让我们用起来吧。

TFRecords文件包含了tf.train.Example 协议内存块(protocol buffer)(协议内存块包含了字段 Features)。我们可以写一段代码获取你的数据， 将数据填入到Example协议内存块(protocol buffer)，将协议内存块序列化为一个字符串， 并且通过tf.python\_io.TFRecordWriter 写入到TFRecords文件。

从TFRecords文件中读取数据， 可以使用tf.TFRecordReader的tf.parse\_single\_example解析器。这个操作可以将Example协议内存块(protocol buffer)解析为张量。

第一，tensorflow里的graph能够记住状态（state），这使得TFRecordReader能够记住tfrecord的位置，并且始终能返回下一个。而这就要求我们在使用之前，必须初始化整个graph，这里我们使用了函数tf.initialize\_all\_variables()来进行初始化。

第二，tensorflow中的队列和普通的队列差不多，不过它里面的operation和tensor都是符号型的（symbolic），在调用sess.run()时才执行。

第三， TFRecordReader会一直弹出队列中文件的名字，直到队列为空。

总结：

生成tfrecord文件

定义record reader解析tfrecord文件

构造一个批生成器（batcher）

构建其他的操作

初始化所有的操作

启动QueueRunner

解析tfrecoreder文件的解析器是 parse\_single\_example，阅读器是tf.TFRecordReader

注意，这里的多线程队列读取的运行机制是，管道启动，sess每run一次img节点就会执行一次操作，因为fetch到了数据，所以队列就弹出。

## 预处理

首先看一下这里要用到的数据集是小莎士比亚文集。很简单就是一个txt文本文件，我们的预处理工作主要是获得该数据集中所有出现的字符，存到vocab里面，并按照其出现次数多少构建索引列表，最后将我们的数据转化为int型索引。此外也包括譬如划分batch等功能。其实要实现的功能很简单，如果要我自己写的话应该是写很多个循环把想要的数据一次次遍历出来即可，但是这里作者使用collections.Counter函数以及dict、map、zip几个功能函数很简单的实现了，这也是我们需要学习的地方。接下来看一下代码：

class TextLoader():

def \_\_init\_\_(self, data\_dir, batch\_size, seq\_length, encoding='utf-8'):

self.data\_dir = data\_dir

self.batch\_size = batch\_size

self.seq\_length = seq\_length

self.encoding = encoding

#第一次运行程序时只有input.txt一个文件，剩下两个文件是运行之后产生的

input\_file = os.path.join(data\_dir, "input.txt")

vocab\_file = os.path.join(data\_dir, "vocab.pkl")

tensor\_file = os.path.join(data\_dir, "data.npy")

#如果是第一次执行则调用preprocess函数，否则调用load\_preprocessed函数。

if not (os.path.exists(vocab\_file) and os.path.exists(tensor\_file)):

print("reading text file")

self.preprocess(input\_file, vocab\_file, tensor\_file)

else:

print("loading preprocessed files")

self.load\_preprocessed(vocab\_file, tensor\_file)

self.create\_batches()

self.reset\_batch\_pointer()

def preprocess(self, input\_file, vocab\_file, tensor\_file):

with codecs.open(input\_file, "r", encoding=self.encoding) as f:

data = f.read()

#使用Counter函数对输入数据进行统计。counter保存data中每个字符出现的次数

counter = collections.Counter(data)

#对counter进行排序，出现次数最多的排在前面

count\_pairs = sorted(counter.items(), key=lambda x: -x[1])

#将data中出现的所有字符保存，这里有65个，所以voacb\_size=65

self.chars, \_ = zip(\*count\_pairs)

self.vocab\_size = len(self.chars)

#按照字符出现次数多少顺序将chars保存，vocab中存储的是char和顺序，这样方便将data转化为索引

self.vocab = dict(zip(self.chars, range(len(self.chars))))

with open(vocab\_file, 'wb') as f:

#保存chars

cPickle.dump(self.chars, f)

#将data中每个字符转化为索引下标。

self.tensor = np.array(list(map(self.vocab.get, data)))

np.save(tensor\_file, self.tensor)

def load\_preprocessed(self, vocab\_file, tensor\_file):

#如果是第二次运行，则可以直接读取之前保存的chars和tensor

with open(vocab\_file, 'rb') as f:

self.chars = cPickle.load(f)

self.vocab\_size = len(self.chars)

self.vocab = dict(zip(self.chars, range(len(self.chars))))

self.tensor = np.load(tensor\_file)

self.num\_batches = int(self.tensor.size / (self.batch\_size \*

self.seq\_length))

def create\_batches(self):

#首先将数据按batch\_size切割，然后每个batch\_size在按照seq\_length进行切割

self.num\_batches = int(self.tensor.size / (self.batch\_size \*

self.seq\_length))

if self.num\_batches == 0:

assert False, "Not enough data. Make seq\_length and batch\_size small."

self.tensor = self.tensor[:self.num\_batches \* self.batch\_size \* self.seq\_length]

xdata = self.tensor

#构造target，这里使用上一个词预测下一个词，所以直接将x向后一个字符即可

ydata = np.copy(self.tensor)

ydata[:-1] = xdata[1:]

ydata[-1] = xdata[0]

#将数据进行切分，这里我们假设数据总长度为10000，batch\_size为100， seq\_length为10.

# 所以num\_batches=10，所以，xdata在reshape之后变成[100, 100],然后在第二个维度上切成10份，

# 所以最终得到[100, 10, 10]的数据

self.x\_batches = np.split(xdata.reshape(self.batch\_size, -1),

self.num\_batches, 1)

self.y\_batches = np.split(ydata.reshape(self.batch\_size, -1),

self.num\_batches, 1)

def next\_batch(self):

x, y = self.x\_batches[self.pointer], self.y\_batches[self.pointer]

self.pointer += 1

return x, y

def reset\_batch\_pointer(self):

self.pointer = 0

## 模型构建

这一部分我们将使用tf构建RNN模型，看代码之前先来关注tf中几个比较重要的函数和其参数：   
1，tf.contrib.rnn.BasicRNNCell/GRUCell/BasicLSTMCell/NASCell   
这里以BasicLSTMCell为例，

\_\_init\_\_(

num\_units,

forget\_bias=1.0,

input\_size=None,

state\_is\_tuple=True,

activation=tf.tanh,

reuse=None

)

num\_units：cell中的神经元个数，注意再很多教程中将RNNcell表示为一个小圆圈，但其实其中有很多个神经元。   
forget\_bias：忘记门的偏置大小   
state\_is\_tuple：若为真则接受和返回的状态保存在长度为2的tuple中。c\_state和m\_state。   
reuse：该单元的参数是否重复使用。如果不是真，而且不是第一次使用，则会报错。

zero\_state(

batch\_size,

dtype

)

将初始状态设为0，需要传入的参数是batch\_size。并且根据cell的state\_size返回不同结果。如果state\_size为正数，则返回[batch\_size x state\_size]的全零初始状态；如果state\_size为列表，则返回一个列表[batch\_size x s] for each s in state\_size.

2，tf.contrib.rnn.DropoutWrapper函数   
为我们上面选择的RNNCell添加dropout属性，抑制过拟合。

\_\_init\_\_(

cell,

input\_keep\_prob=1.0,

output\_keep\_prob=1.0,

state\_keep\_prob=1.0,

variational\_recurrent=False,

input\_size=None,

dtype=None,

seed=None

)

从构造函数可以看出，对每个cell而言，我们都有input、output、state三个层面的dropout。   
cell: an RNNCell, 可以使我们上面选择的某一种RNNCell.   
input\_keep\_prob: 0-1，输入的dropout几率   
output\_keep\_prob: 0-1，输出的dropout几率   
state\_keep\_prob: 0-1，state的dropout几率，在output的基础上进行dropout   
variational\_recurrent: 若为真，则说明所有时间步上应用相同的dropout，并且需要设置input\_size参数。

3，tf.contrib.rnn.MultiRNNCell函数

\_\_init\_\_(

cells,

state\_is\_tuple=True

)

1. cells: cell list，包含n层rnn的cell.

2. state\_is\_tuple: 若为真则接受和返回的状态是n（层数）元tuple.

zero\_state(

batch\_size,

dtype

)

4，tf.contrib.legacy\_seq2seq.rnn\_decoder函数

rnn\_decoder(

decoder\_inputs,

initial\_state,

cell,

loop\_function=None,

scope=None

)

该函数实现了一个简单的多层rnn模型。上面的MultiRNNCell函数构造了一个时间步的多层rnn，本函数则实现将其循环num\_steps个时间步。

decoder\_inputs：输入列表，是一个长度为num\_steps的列表，每个元素是[batch\_size, input\_size]的2-D维的tensor

initial\_state：初始化状态，2-D的tensor，shape为 [batch\_size x cell.state\_size].

cell：RNNCell

loop\_function：如果不为空，则将该函数应用于第i个输出以得到第i+1个输入，此时decoder\_inputs变量除了第一个元素之外其他元素会被忽略。其形式定义为：loop(prev, i)=next。prev是[batch\_size x output\_size]，i是表明第i步，next是[batch\_size x input\_size]。

这里我们可以看一下该函数的源代码加深理解：

with variable\_scope.variable\_scope(scope or "rnn\_decoder"):

state = initial\_state

outputs = []

prev = None

#遍历n个时间步

for i, inp in enumerate(decoder\_inputs):

#下面这两个if语句只有在第2个时间步之后才会被执行

if loop\_function is not None and prev is not None:

with variable\_scope.variable\_scope("loop\_function", reuse=True):

inp = loop\_function(prev, i)

if i > 0:

variable\_scope.get\_variable\_scope().reuse\_variables()

#重点是循环执行这个

output, state = cell(inp, state)

outputs.append(output)

if loop\_function is not None:

prev = output

return outputs, state

看了上面几个重要函数的介绍，那么对下面代码就不难理解了。在tf中，构造RNN模型的一般思路就是RNNcell–>dropout–>MultiRNNCell–>重复time\_steps步（这里使用legacy\_seq2seq.rnn\_decoder函数实现，我们也可以自己定义自己的模型）。接下来我们看一下model.py文件中关于模型构建部分的代码。

class Model():

def \_\_init\_\_(self, args, training=True):

self.args = args

if not training:

args.batch\_size = 1

args.seq\_length = 1

#几种可选的rnn cell

if args.model == 'rnn':

cell\_fn = rnn.BasicRNNCell

elif args.model == 'gru':

cell\_fn = rnn.GRUCell

elif args.model == 'lstm':

cell\_fn = rnn.BasicLSTMCell

elif args.model == 'nas':

cell\_fn = rnn.NASCell

else:

raise Exception("model type not supported: {}".format(args.model))

cells = []

#因为是多层RNN，所以在recoll时我们要输入的是一个多层的cell，

# 根据是否处于训练过程和需要dropout添加dropout层

for \_ in range(args.num\_layers):

cell = cell\_fn(args.rnn\_size)

if training and (args.output\_keep\_prob < 1.0 or args.input\_keep\_prob < 1.0):

cell = rnn.DropoutWrapper(cell,

input\_keep\_prob=args.input\_keep\_prob,

output\_keep\_prob=args.output\_keep\_prob)

cells.append(cell)

#MultiRNNCell接受我们之前定义的多层RNNcell列表。

# state\_is\_tuple默认为True，表示输入和输出都用tuple存储，将来会丢弃False的选项。

self.cell = cell = rnn.MultiRNNCell(cells, state\_is\_tuple=True)

self.input\_data = tf.placeholder(

tf.int32, [args.batch\_size, args.seq\_length])

self.targets = tf.placeholder(

tf.int32, [args.batch\_size, args.seq\_length])

#定义初始化状态，可以直接调用cell.zero\_state函数，参数为batch\_size

self.initial\_state = cell.zero\_state(args.batch\_size, tf.float32)

with tf.variable\_scope('rnnlm'):

softmax\_w = tf.get\_variable("softmax\_w",

[args.rnn\_size, args.vocab\_size])

softmax\_b = tf.get\_variable("softmax\_b", [args.vocab\_size])

#将输入索引转化为索引

embedding = tf.get\_variable("embedding", [args.vocab\_size, args.rnn\_size])

inputs = tf.nn.embedding\_lookup(embedding, self.input\_data)

# dropout beta testing: double check which one should affect next line

if training and args.output\_keep\_prob:

inputs = tf.nn.dropout(inputs, args.output\_keep\_prob)

#将输入切分

inputs = tf.split(inputs, args.seq\_length, 1)

inputs = [tf.squeeze(input\_, [1]) for input\_ in inputs]

def loop(prev, \_):

prev = tf.matmul(prev, softmax\_w) + softmax\_b

prev\_symbol = tf.stop\_gradient(tf.argmax(prev, 1))

return tf.nn.embedding\_lookup(embedding, prev\_symbol)

#直接调用rnn\_decoder函数构建RNN模型

outputs, last\_state = legacy\_seq2seq.rnn\_decoder(inputs, self.initial\_state, cell, loop\_function=loop if not training else None, scope='rnnlm')

output = tf.reshape(tf.concat(outputs, 1), [-1, args.rnn\_size])

#下面就是loss和梯度计算，优化器定义部分

self.logits = tf.matmul(output, softmax\_w) + softmax\_b

self.probs = tf.nn.softmax(self.logits)

loss = legacy\_seq2seq.sequence\_loss\_by\_example(

[self.logits],

[tf.reshape(self.targets, [-1])],

[tf.ones([args.batch\_size \* args.seq\_length])])

self.cost = tf.reduce\_sum(loss) / args.batch\_size / args.seq\_length

with tf.name\_scope('cost'):

self.cost = tf.reduce\_sum(loss) / args.batch\_size / args.seq\_length

self.final\_state = last\_state

self.lr = tf.Variable(0.0, trainable=False)

tvars = tf.trainable\_variables()

#RNN中常用的梯度截断，防止出现梯度过大难以求导的现象

grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(self.cost, tvars),

args.grad\_clip)

with tf.name\_scope('optimizer'):

optimizer = tf.train.AdamOptimizer(self.lr)

self.train\_op = optimizer.apply\_gradients(zip(grads, tvars))

# instrument tensorboard

tf.summary.histogram('logits', self.logits)

tf.summary.histogram('loss', loss)

tf.summary.scalar('train\_loss', self.cost)

## 模型训练

模型定义完成之后，我们要进行的就是读入数据，构建模型并开始训练等一系列操作。这里并没有设么新鲜的代码，我们直接看程序就可以了：

def train(args):

#读入数据

data\_loader = TextLoader(args.data\_dir, args.batch\_size, args.seq\_length)

args.vocab\_size = data\_loader.vocab\_size

# check compatibility if training is continued from previously saved model

if args.init\_from is not None:

# 继续从之前的模型接着训练（可以先不看）

assert os.path.isdir(args.init\_from)," %s must be a a path" % args.init\_from

assert os.path.isfile(os.path.join(args.init\_from,"config.pkl")),"config.pkl file does not exist in path %s"%args.init\_from

assert os.path.isfile(os.path.join(args.init\_from,"chars\_vocab.pkl")),"chars\_vocab.pkl.pkl file does not exist in path %s" % args.init\_from

ckpt = tf.train.get\_checkpoint\_state(args.init\_from)

assert ckpt, "No checkpoint found"

assert ckpt.model\_checkpoint\_path, "No model path found in checkpoint"

# open old config and check if models are compatible

with open(os.path.join(args.init\_from, 'config.pkl'), 'rb') as f:

saved\_model\_args = cPickle.load(f)

need\_be\_same = ["model", "rnn\_size", "num\_layers", "seq\_length"]

for checkme in need\_be\_same:

assert vars(saved\_model\_args)[checkme]==vars(args)[checkme],"Command line argument and saved model disagree on '%s' "%checkme

# open saved vocab/dict and check if vocabs/dicts are compatible

with open(os.path.join(args.init\_from, 'chars\_vocab.pkl'), 'rb') as f:

saved\_chars, saved\_vocab = cPickle.load(f)

assert saved\_chars==data\_loader.chars, "Data and loaded model disagree on character set!"

assert saved\_vocab==data\_loader.vocab, "Data and loaded model disagree on dictionary mappings!"

if not os.path.isdir(args.save\_dir):

os.makedirs(args.save\_dir)

with open(os.path.join(args.save\_dir, 'config.pkl'), 'wb') as f:

cPickle.dump(args, f)

with open(os.path.join(args.save\_dir, 'chars\_vocab.pkl'), 'wb') as f:

cPickle.dump((data\_loader.chars, data\_loader.vocab), f)

#构建模型

model = Model(args)

with tf.Session() as sess:

# 写入Summary

summaries = tf.summary.merge\_all()

writer = tf.summary.FileWriter(

os.path.join(args.log\_dir, time.strftime("%Y-%m-%d-%H-%M-%S")))

writer.add\_graph(sess.graph)

#参数初始化

sess.run(tf.global\_variables\_initializer())

saver = tf.train.Saver(tf.global\_variables())

# restore model

if args.init\_from is not None:

saver.restore(sess, ckpt.model\_checkpoint\_path)

#开始循环送入数据并训练

for e in range(args.num\_epochs):

sess.run(tf.assign(model.lr,

args.learning\_rate \* (args.decay\_rate \*\* e)))

data\_loader.reset\_batch\_pointer()

state = sess.run(model.initial\_state)

for b in range(data\_loader.num\_batches):

start = time.time()

x, y = data\_loader.next\_batch()

feed = {model.input\_data: x, model.targets: y}

for i, (c, h) in enumerate(model.initial\_state):

feed[c] = state[i].c

feed[h] = state[i].h

train\_loss, state, \_ = sess.run([model.cost, model.final\_state, model.train\_op], feed)

# instrument for tensorboard

summ, train\_loss, state, \_ = sess.run([summaries, model.cost, model.final\_state, model.train\_op], feed)

writer.add\_summary(summ, e \* data\_loader.num\_batches + b)

end = time.time()

print("{}/{} (epoch {}), train\_loss = {:.3f}, time/batch = {:.3f}"

.format(e \* data\_loader.num\_batches + b,

args.num\_epochs \* data\_loader.num\_batches,

e, train\_loss, end - start))

if (e \* data\_loader.num\_batches + b) % args.save\_every == 0\

or (e == args.num\_epochs-1 and

b == data\_loader.num\_batches-1):

# save for the last result

checkpoint\_path = os.path.join(args.save\_dir, 'model.ckpt')

saver.save(sess, checkpoint\_path,

global\_step=e \* data\_loader.num\_batches + b)

print("model saved to {}".format(checkpoint\_path))

## tf.Variable

.import tensorflow as tf;

.import numpy as np;

.import matplotlib.pyplot as plt;

.

.a1 = tf.Variable(tf.random\_normal(shape=[2,3], mean=0, stddev=1), name='a1')

.a2 = tf.Variable(tf.constant(1), name='a2')

.a3 = tf.Variable(tf.ones(shape=[2,3]), name='a3')

.

.with tf.Session() as sess:

.    sess.run(tf.initialize\_all\_variables())

.    print sess.run(a1)

.    print sess.run(a2)

.    print sess.run(a3)

输出：

[[ 0.76599932  0.99722123 -0.89361787]

 [ 0.19991693 -0.16539733  2.16605783]]

1

[[ 1.  1.  1.]

 [ 1.  1.  1.]]

## 安装

两个小的tips

版本：anaconda 4.2 tensorflow 0.12.1

windows下安装tensorflow，可以依照我的博客：[http://blog.csdn.net/xxzhangx/article/details/54379255](http://blog.csdn.net/xxzhangx/article/details/54379255" \t "http://blog.csdn.net/xxzhangx/article/details/_blank)

pip install --ignore-installed --upgrade tensorflow-gpu

### 报错处理

错误：

[CPU instructions not compiled with TensorFlow](https://stackoverflow.com/questions/42463594/cpu-instructions-not-compiled-with-tensorflow)

处理：

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL']='2'

import tensorflow as tf

this should silence the warnings. 'TF\_CPP\_MIN\_LOG\_LEVEL' represents the Tensorflow environment variable responsible for logging. Also if you are on Ubuntu you may use this code below：

export TF\_CPP\_MIN\_LOG\_LEVEL=2

## 函数表

### 小函数



tf.equal(A, B)

是对比这两个矩阵或者向量的相等的元素，如果是相等的那就返回True，反正返回False，返回的值的矩阵维度和A是一样的

tf.group() 与 tf.tuple()

如果我们有很多 tensor 或 op想要一起run，这时这两个函数就是一个很好的帮手了。

w = tf.Variable(1)

mul = tf.multiply(w, 2)

add = tf.add(w, 2)

group = tf.group(mul, add)

tuple = tf.tuple([mul, add])# sess.run(group)和sess.run(tuple)都会求Tensor(add)#Tensor(mul)的值。区别是，tf.group()返回的是`op`#tf.tuple()返回的是list of tensor。#这样就会导致，sess.run(tuple)的时候，会返回 Tensor(mul),Tensor(add)的值.#而 sess.run(group)不会

tensorflow ConfigProto

tf.ConfigProto一般用在创建session的时候。用来对session进行参数配置

with tf.Session(config = tf.ConfigProto(...),...)

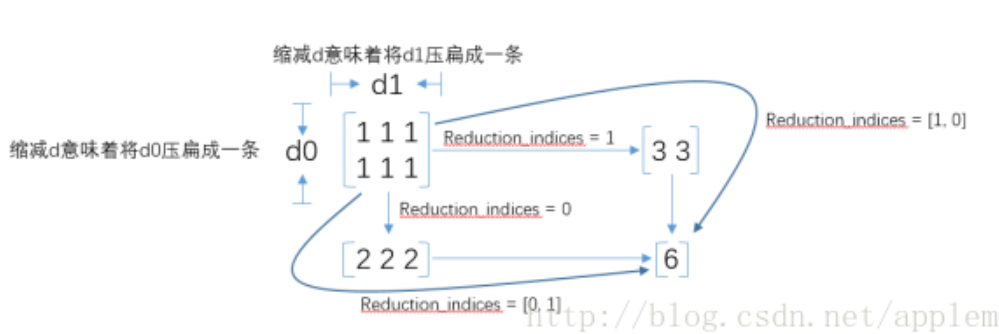
#tf.ConfigProto()的参数

log\_device\_placement=True : 是否打印设备分配日志

allow\_soft\_placement=True ： 如果你指定的设备不存在，允许TF自动分配设备

tf.ConfigProto(log\_device\_placement=True,allow\_soft\_placement=True)

reduce\_sum



tf.constant\_initializer

tf.truncated\_normal\_initializer

tf.nn.xw\_plus\_b((x, weights) + biases.)

 相当于matmul(x, weights) + biases.

tf.cast此函数是类型转换函数

decode\_raw

从二进制文件中读取固定长度纪录， 可以使用[tf.FixedLengthRecordReader](http://wiki.jikexueyuan.com/project/tensorflow-zh/how_tos/tensorflow-zh/SOURCE/api_docs/python/io_ops.html)的[tf.decode\_raw](http://wiki.jikexueyuan.com/project/tensorflow-zh/how_tos/tensorflow-zh/SOURCE/api_docs/python/io_ops.html)操作。decode\_raw操作可以讲一个字符串转换为一个uint8的张量。

for \_ in range(5) 循环5次

tf.assign(state, new\_value)

这个函数的功能主要是把A的值变为new\_number

y = max f(t) 代表：y 是f(t)函式所有的值中最大的output。

y = arg max f(t) 代表：y 是f(t)函式中，會產生最大output的那個參數t。

数值乘法mul

a = tf.placeholder(tf.float32)

b = tf.placeholder(tf.float32)y = tf.mul(a, b)

sess = tf.Session()

print (sess.run(y, feed\_dict={a: 3, b: 3}))

数值减法sub

数值除法div

数值取模mod

数值绝对值abs

数值非负值neg例如:a=-3, neg (a) = 3

数值符号函数sign例如: a=-3,sign(a) = -1

数值平方square

数值最近的整数round

数值平方根sqrt

数值幂次pow例如: a=2,b=3,pow(a,b) = 8

数值最近的整数exp

数值取对数log

数值取最大值maximum

数值最小值minimum

数值余弦函数cos

数值正弦函数sin

tf.matrix\_inverse  方阵的逆矩阵

tf.matrix\_determinant  方阵的行列式

tf.diag  给定对角线上的值，返回对角tensor

矩阵乘法

A \* B 表示按元素计算

tf.mul(A,B)  表示按元素计算

tf.matmul(A,B) 表示矩阵乘法

tf.rank(a) 返回tensor的rank

#’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

某一维求和

Tensor：tf.reduce\_sum(b,reduction\_indices=1)

numpy：np.sum(b,axis=1)

合并、连接数组

Tensor

tf.concat(0,[a,b])第一个参数表述位数

若a （1，128，128，3）  b( 1，128，128，3）

tf.concat(0,[a,b])  ( 2，128，128，3）

获取整行整列数据

Tensor

temp = tf.constant(0,shape=[5,5])

temp1 = temp[0,:] 获取某行

temp2 = temp[:,1] 获取某列

temp[1,1]  获取某个元素

temp[1:3,1:3]  获取某个范围的行列元素

tf. get\_variable()

因为如果使用Variable 的话每次都会新建变量，但是大多数时候我们是希望一些变量重用的，所以就用到了get\_variable()。它会去搜索变量名，然后没有就新建，有就直接用。

### 随机化函数

tf.random\_normal(shape,mean=0.0,stddev=1.0,dtype=tf.float32,seed=None,name=None)

返回一个tensor其中的元素的值服从正态分布。

tf.truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

返回一个tensor其中的元素服从截断正态分布。这是一个截断的产生正太分布的函数，就是说产生正太分布的值如果与均值的差值大于两倍的标准差，那就重新生成。

tf.random\_uniform(shape,minval=0,maxval=None,dtype=tf.float32,seed=None,name=None)

返回一个形状为shape的tensor，其中的元素服从minval和maxval之间的均匀分布。

tf.random\_shuffle(value,seed=None,name=None)

对value（是一个tensor）的第一维进行随机化。

       [[1,2],            [[2,3],

        [2,3],     ==>  [1,2],

        [3,4]]             [3,4]]

tf.set\_random\_seed(seed)

设置产生随机数的种子。

### 常量值函数

tf.zeros(shape, dtype=tf.float32, name=None)

tf.zeros\_like(tensor, dtype=None, name=None)

tf.ones(shape, dtype=tf.float32, name=None)

tf.ones\_like(tensor, dtype=None, name=None)

tf.fill(dims, value, name=None)

tf.constant(value, dtype=None, shape=None, name='Const')

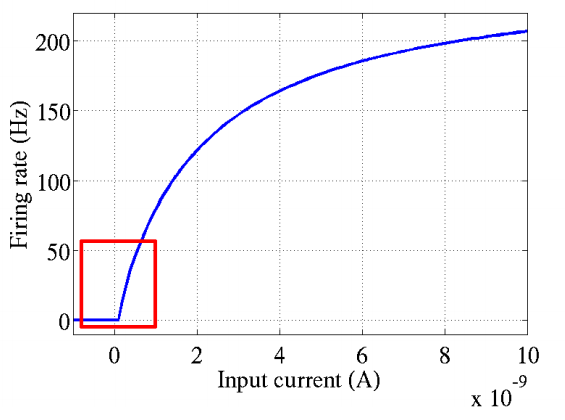
### 激活函数

连续的平滑非线性函数（sigmoid，tanh和softplus），连续但不平滑的非线性函数（relu，relu6和relu\_x）和随机正则化函数（dropout）

所有的激活函数都是单独应用在每个元素上面的，并且输出张量的维度和输入张量的维度一样。

近似生物神经激活函数：Softplus&ReLu

2001年，神经科学家Dayan、Abott从生物学角度，模拟出了脑神经元接受信号更精确的激活模型，该模型如左图所示：



这个模型对比Sigmoid系主要变化有三点：①单侧抑制 ②相对宽阔的兴奋边界 ③稀疏激活性（重点，可以看到红框里前端状态完全没有激活）

同年，Charles Dugas等人在做正数回归预测[论文](http://95.173.210.46/url?q=http://papers.nips.cc/paper/1920-incorporating-second-order-functional-knowledge-for-better-option-pricing.pdf&sa=U&ei=4SM6VefqBIbaPL-HgPAN&ved=0CAUQFjAA&usg=AFQjCNHY_71xurXxTe9sgH1iHqjZHwg2rA" \t "https://www.cnblogs.com/neopenx/p/_blank)中偶然使用了Softplus函数，Softplus函数是Logistic-Sigmoid函数原函数。

 Softplus(x)=log(1+e x ) Softplus(x)=log(1+ex)

按照论文的说法，一开始想要使用一个指数函数（天然正数）作为激活函数来回归，但是到后期梯度实在太大，难以训练，于是加了一个log来减缓上升趋势。

加了1是为了保证非负性。同年，Charles Dugas等人在NIPS会议[论文](http://papers.nips.cc/paper/2062-estimating-car-insurance-premia-a-case-study-in-high-dimensional-data-inference" \t "https://www.cnblogs.com/neopenx/p/_blank)中又调侃了一句，Softplus可以看作是强制非负校正函数max(0,x) max(0,x) 平滑版本。

偶然的是，同是2001年，ML领域的Softplus/Rectifier激活函数与神经科学领域的提出脑神经元激活频率函数有神似的地方，这促成了新的激活函数的研究。

校正激活函数max(0,x)

#### 怎么选择激活函数

我觉得这种问题不可能有定论的吧，只能说是个人建议。

如果你使用 ReLU，那么一定要小心设置 learning rate，而且要注意不要让你的网络出现很多 “dead” 神经元，如果这个问题不好解决，那么可以试试 Leaky ReLU、PReLU 或者 Maxout.

友情提醒：最好不要用 sigmoid，你可以试试 tanh，不过可以预期它的效果会比不上 ReLU 和 Maxout.

还有，通常来说，很少会把各种激活函数串起来在一个网络中使用的。

#### Vanishing Gradient Problem

更倾向于使用线性神经激活函数的另外一个原因是，减轻梯度法训练深度网络时的Vanishing Gradient Problem。

看过BP推导的人都知道，误差从输出层反向传播算梯度时，在各层都要乘当前层的输入神经元值，激活函数的一阶导数。

即Grad=Error⋅Sigmoid ′ (x)⋅x Grad=Error⋅Sigmoid′(x)⋅x 。使用双端饱和(即值域被限制)Sigmoid系函数会有两个问题：

①Sigmoid'(x)∈(0,1)  导数缩放

②x∈(0,1)或x∈(-1,1)  饱和值缩放

这样，经过每一层时，Error都是成倍的衰减，一旦进行递推式的多层的反向传播，梯度就会不停的衰减，消失，使得网络学习变慢。

而校正激活函数的梯度是1，且只有一端饱和，梯度很好的在反向传播中流动，训练速度得到了很大的提高。

Softplus函数则稍微慢点，Softplus'(x)=Sigmoid(x)∈(0,1) ，但是也是单端饱和，因而速度仍然会比Sigmoid系函数快。

#### relu

线性整流函数（Rectified Linear Unit, ReLU）,又称修正线性单元, 是一种[人工神经网络](https://baike.baidu.com/item/%E4%BA%BA%E5%B7%A5%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank)中常用的激活函数（activation function），通常指代以[斜坡函数](https://baike.baidu.com/item/%E6%96%9C%E5%9D%A1%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank)及其变种为代表的非线性函数。

比较常用的线性整流函数有[斜坡函数](https://baike.baidu.com/item/%E6%96%9C%E5%9D%A1%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank) f(x) = max(0, x)，以及带泄露整流函数 (Leaky ReLU)，其中为x神经元(Neuron)的输入。线性整流被认为有一定的生物学原理 [1]  ，并且由于在实践中通常有着比其他常用激活函数（譬如[逻辑函数](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank)）更好的效果，而被如今的[深度神经网络](https://baike.baidu.com/item/%E6%B7%B1%E5%BA%A6%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank)广泛使用于诸如图像识别等[计算机视觉](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E8%A7%86%E8%A7%89" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank) [1]  人工智能领域。

即max(features, 0)。

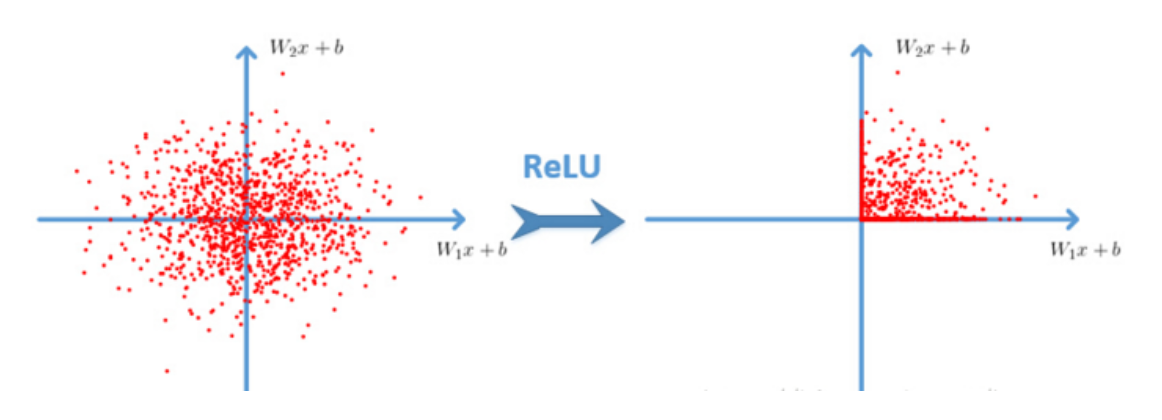
a = tf.constant([-1.0, 2.0])

with tf.Session() as sess:

b = tf.nn.relu(a)

print sess.run(b)

输出[ 0. 2.]



相比于 sigmoid/tanh，ReLU 只需要一个阈值就可以得到激活值，而不用去算一大堆复杂的运算。

ReLU 的缺点： 当然 ReLU 也有缺点，就是训练的时候很”脆弱”，很容易就”die”了. 什么意思呢？

举个例子：一个非常大的梯度流过一个 ReLU 神经元，更新过参数之后，这个神经元再也不会对任何数据有激活现象了。

如果这个情况发生了，那么这个神经元的梯度就永远都会是0.

实际操作中，如果你的learning rate 很大，那么很有可能你网络中的40%的神经元都”dead”了。   
当然，如果你设置了一个合适的较小的learning rate，这个问题发生的情况其实也不会太频繁。

优势：

更加有效率的[梯度下降](https://baike.baidu.com/item/%E6%A2%AF%E5%BA%A6%E4%B8%8B%E9%99%8D" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%95%B4%E6%B5%81%E5%87%BD%E6%95%B0/_blank)以及反向传播：避免了梯度爆炸和梯度消失问题。

简化计算过程：没有了其他复杂激活函数中诸如指数函数的影响；同时活跃度的分散性使得神经网络整体计算成本下降。

#### relu6

即min(max(features, 0), 6)

a = tf.constant([-1.0, 12.0])

with tf.Session() as sess:

b = tf.nn.relu6(a)

print sess.run(b)

输出[ 0. 6.]

#### Leaky-ReLU

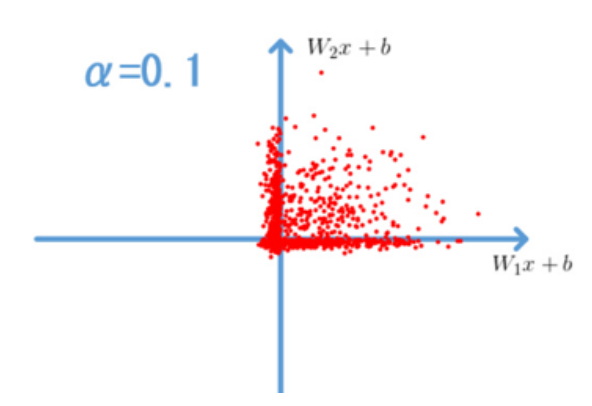
Leaky ReLUs： 就是用来解决这个 “dying ReLU” 的问题的。与 ReLU 不同的是：

f(x)=αx，(x<0)

f(x)=x，(x>=0)

这里的 α 是一个很小的常数。这样，即修正了数据分布，又保留了一些负轴的值，使得负轴信息不会全部丢失。

关于Leaky ReLU 的效果，众说纷纭，没有清晰的定论。有些人做了实验发现 Leaky ReLU 表现的很好；有些实验则证明并不是这样。



#### softplus

即log( exp( features ) + 1)

a = tf.constant([-1.0, 12.0])

with tf.Session() as sess:

b = tf.nn.softplus(a)

print sess.run(b)

输出[ 0.31326166 12.00000572]

#### dropout

tf.nn.dropout(x, keep\_prob, noise\_shape = None, seed = None, name = None)

一个神经元将以概率keep\_prob决定是否放电，如果不放电，那么该神经元的输出将是0，如果该神经元放电，那么该神经元的输出值将被放大到原来的1/keep\_prob倍。这里的放大操作是为了保持神经元输出总个数不变。比如，神经元的值为[1, 2]，keep\_prob的值是0.5，并且是第一个神经元是放电的，第二个神经元不放电，那么神经元输出的结果是[2, 0]，也就是相当于，第一个神经元被当做了1/keep\_prob个输出，即2个。这样保证了总和2个神经元保持不变。

默认情况下，每个神经元是否放电是相互独立的。但是，如果noise\_shape被修改了，那么他对于变量x就是一个广播形式，而且当且仅当 noise\_shape[i] == shape(x)[i] ，x中的元素是相互独立的。比如，如果 shape(x) = [k, l, m, n], noise\_shape = [k, 1, 1, n] ，那么每个批和通道都是相互独立的，但是每行和每列的数据都是关联的，即要不都为0，要不都还是原来的值。

a = tf.constant([[-1.0, 2.0, 3.0, 4.0]])

with tf.Session() as sess:

b = tf.nn.dropout(a, 0.5, noise\_shape = [1,4])

print sess.run(b)

b = tf.nn.dropout(a, 0.5, noise\_shape = [1,1])

print sess.run(b)

输出：

[[-2. 0. 0. 8.]]

[[-0. 0. 0. 0.]]

或：

[[-2. 4. 0. 8.]]

[[-0. 0. 0. 0.]]

或：

[[-0. 4. 0. 0.]]

[[-2. 4. 6. 8.]]

#### bias\_add

tf.nn.bias\_add(value, bias, name = None)

将偏差项 bias 加到 value 上面

a = tf.constant([[1.0, 2.0],[1.0, 2.0],[1.0, 2.0]])  
b = tf.constant([2.0,1.0])  
c = tf.constant([1.0])  
sess = tf.Session()  
print(sess.run(tf.nn.bias\_add(a, b)))  
# 因为 a 最后一维的维度是 2 ，但是 c 的维度是 1，所以以下语句将发生错误  
# print(sess.run(tf.nn.bias\_add(a, c)))  
# 但是 tf.add() 可以正确运行  
print(sess.run(tf.add(a, c)))

[[ 3. 3.]

[ 3. 3.]

[ 3. 3.]]

[[ 2. 3.]

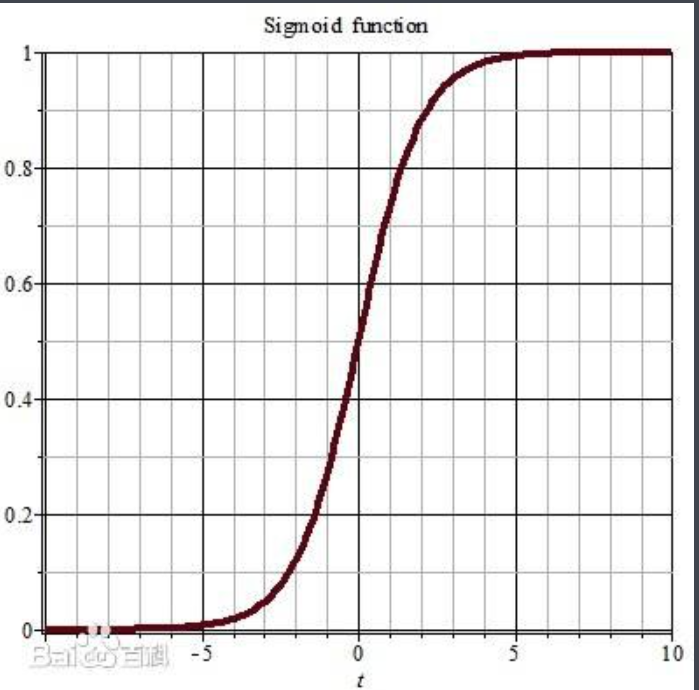
[ 2. 3.]

[ 2. 3.]]

#### sigmoid

tf.sigmoid(x, name = None)

解释：这个函数的作用是计算 x 的 sigmoid 函数。具体计算公式为 y = 1 / (1 + exp(-x))。



使用例子：

a = tf.constant([[1.0, 2.0], [1.0, 2.0], [1.0, 2.0]])

sess = tf.Session()

print sess.run(tf.sigmoid(a))

输出

[[ 0.7310586 0.88079703]

[ 0.7310586 0.88079703]

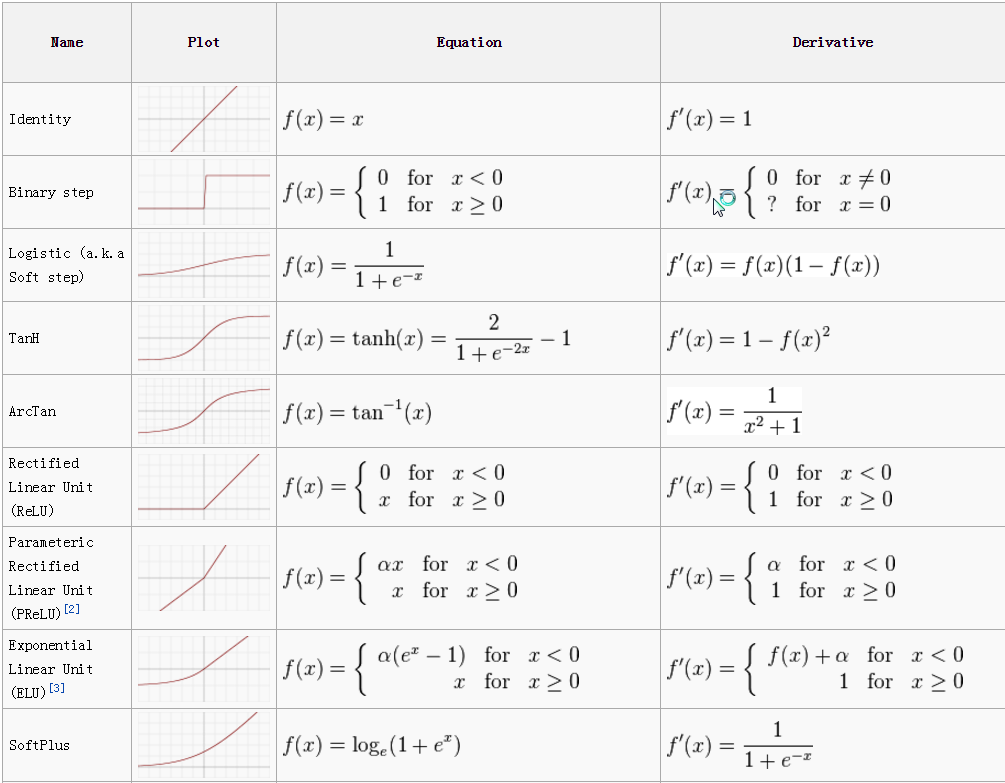
[ 0.7310586 0.88079703]]

#### tanh

tf.tanh(x, name = None)

解释：这个函数的作用是计算 x 的 tanh 函数。具体计算公式为 ( exp(x) - exp(-x) ) / ( exp(x) + exp(-x) )。

#### 其他



### nn

#### embedding\_lookup

import tensorflow as tf;

import numpy as np;

c = np.random.random([10,1])

b = tf.nn.embedding\_lookup(c, [1, 3])

with tf.Session() as sess:

    sess.run(tf.initialize\_all\_variables())

    print sess.run(b)

print c

输出：

[[ 0.77505197]  
 [ 0.20635818]]  
[[ 0.23976515]  
 [ 0.77505197]  
 [ 0.08798201]  
 [ 0.20635818]  
 [ 0.37183035]  
 [ 0.24753178]  
 [ 0.17718483]  
 [ 0.38533808]  
 [ 0.93345168]  
 [ 0.02634772]]

### 启动图

（1）tf.Session()

         构造阶段完成后, 才能启动图. 启动图的第一步是创建一个 Session 对象, 如果无任何创建参数, 会话构造器将启动默认图.

（2）tf.InteractivesSession()

         为了便于使用诸如 IPython之类的 Python 交互环境, 可以使用InteractiveSession 代替 Session 类, 使用 Tensor.eval()和 Operation.run()方法代替Session.run(). 这样可以避免使用一个变量来持有会话。

（3）tf.train.Supervisor().managed\_session()   
         与上面两种启动图相比较来说，Supervisor() 帮助我们处理一些事情：

         (a) 自动去 checkpoint 加载数据或者初始化数据

       （b) 自动有一个 Saver ，可以用来保存 checkpoint

               eg: sv.saver.save(sess, save\_path)

          (c) 有一个 summary\_computed 用来保存 Summary

         因此我们可以省略了以下内容：

          （a）手动初始化或者从 checkpoint  中加载数据

          （b）不需要创建 Saver 类， 使用 sv 内部的就可以

          （c）不需要创建 Summary\_Writer()

managed\_session在异步模式就是参数初始化完成之后，大家就可以开始干活了。   
prepare\_or\_wait\_for\_session在同步模式，不但参数初始化完成，还得主节点也准备好了，其他节点才开始干活。

例子：

import tensorflow as tf

matrix1 = tf.constant([[3., 3.]])

matrix2 = tf.constant([[2.], [2.]])

preduct = tf.matmul(matrix1, matrix2)

sv = tf.train.Supervisor(logdir=None, init\_op=tf.global\_variables\_initializer())

with sv.managed\_session() as sess:

print sess.run(preduct)

## sess.run() 和 Tensor.eval()

sess.run()在同一步获取多个tensor中的值，使用Tensor.eval()时只能在同一步当中获取一个tensor值，并且每次使用 eval 和 run时，都会执行整个计算图。

### contrib.layers函数

（1）tf.nn ：提供神经网络相关操作的支持，包括卷积操作（conv）、池化操作（pooling）、归一化、loss、分类操作、embedding、RNN、Evaluation。

（2）tf.layers：主要提供的高层的神经网络，主要和卷积相关的，个人感觉是对tf.nn的进一步封装，tf.nn会更底层一些。

（3）tf.contrib：tf.contrib.layers提供够将计算图中的  网络层、正则化、摘要操作、是构建计算图的高级操作，但是tf.contrib包含不稳定和实验代码，有可能以后API会改变。

tf.contrib.layers.optimizers模块：tf.contrib.layers.optimizers包括的优化器有Adagrad，SGD，Momentum等。它们用来解决数值分析的优化问题，比如，优化参数空间寻找最优模型；  
tf.contrib.layers.regularizers模块：tf.contrib.layers.regularizers包括的正则化有L1规则化和L2规则化。规则化经常被用来抑制模型训练时特征数过大导致的过拟合（overfitting）问题；有时也作为Lasso回归和Ridge回归的构建模块；  
tf.contrib.layers.initializers模块：tf.contrib.layers.initializers一般用来做模型初始化。包括深度学习在内的许多算法都要求计算梯度来优化模型。随机初始化模型参数有助于在参数空间中找到最优参数解。TensorFlow提供的有Xavier初始化器，用来在所有层中保持梯度大体相同；  
tf.contrib.layers.feature\_column模块：tf.contrib.layers.feature\_column提供函数（比如，bucketing/binning，crossing/compostion，和embedding）来转换连续特征和离散特征；  
tf.contrib.layers.embedding模块：tf.contrib.layers.embedding转化高维分类特征成低维、密集实数值向量。

contrib.crf.crf\_decode

crf\_decode(potentials,transition\_params,sequence\_length)

在tensorflow内解码

参数:

potentials: 一个形状为[batch\_size, max\_seq\_len, num\_tags] 的tensor,

transition\_params: 一个形状为[num\_tags, num\_tags] 的转移矩阵

sequence\_length: 一个形状为[batch\_size] 的 ,表示batch中每个序列的长度

返回:

decode\_tags:一个形状为[batch\_size, max\_seq\_len] 的tensor,类型是tf.int32.表示最好的序列标记.

best\_score: 有个形状为[batch\_size] 的tensor, 包含每个序列解码标签的分数.

### nn.moments和nn.batch\_normalization

mean, variance = tf.nn.moments(x, axes, name=None, keep\_dims=False)

计算统计矩，mean 是一阶矩即均值，variance 则是二阶中心矩即方差，axes=[0]表示按列计算；

tf.nn.batch\_normalization(x, mean, variance, offset, scale, variance\_epsilon, name=None)

相当于进行如下计算

W = (W - mean) / tf.sqrt(var + 0.001)

W = W \* scale + shift

tf.nn.batch\_norm\_with\_global\_normalization(x, mean, variance, beta, gamma, variance\_epsilon, scale\_after\_normalization, name=None)；

tf.nn.moments 计算返回的 mean 和 variance 作为 tf.nn.batch\_normalization 参数调用；

例子：

计算每个列的均值及方差。

W = tf.constant([[-2.,12.,6.],[3.,2.,8.]], )

mean,var = tf.nn.moments(W, axes = [0])

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

resultMean = sess.run(mean)

print(resultMean)

resultVar = sess.run(var)

print(resultVar)

[ 0.5 7. 7. ]

[ 6.25 25. 1. ]

一维一维往下走计算高纬度矩阵

如我们需计算的 tensor 的 shape 为一个四元组 [batch\_size, height, width, kernels]，一个示例程序如下：

import tensorflow as tf

shape = [128, 32, 32, 64]a = tf.Variable(tf.random\_normal(shape)) # a：activations

axis = list(range(len(shape)-1)) # len(x.get\_shape())

a\_mean, a\_var = tf.nn.moments(a, axis)

这里我们仅给出 a\_mean, a\_var 的维度信息，

sess = tf.Session()

sess.run(tf.global\_variables\_initalizer())

sess.run(a\_mean).shape # (64, )

sess.run(a\_var).shape # (64, ) ⇒ 也即是以 kernels 为单位，batch 中的全部样本的均值与方

### where

a = tf.constant([False,False,True,False,True],dtype=tf.bool)

b = tf.where(tf.equal(a,True))

sess = tf.Session()

print(sess.run(b))

print结果   
[[2]   
[4]]

tf.where(input, name=None)一种用法，在实际应用中发现了另外一种使用方法tf.where(input, a,b)，其中a，b均为尺寸一致的tensor，作用是将a中对应input中true的位置的元素值不变，其余元素进行替换，替换成b中对应位置的元素值，下面使用代码来说明：

import tensorflow as tf

import numpy as np

sess=tf.Session()

a=np.array([[1,0,0],[0,1,1]])

a1=np.array([[3,2,3],[4,5,6]])

print(sess.run(tf.equal(a,1)))



print(sess.run(tf.where(tf.equal(a,1),a1,1-a1)))



### cond

format：tf.cond(pred, fn1, fn2, name=None)

z = tf.multiply(a, b)

result = tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square(y))

f.cond()类似于c语言中的if...else...，用来控制数据流向，但是仅仅类似而已，其中差别还是挺大的.

上面例子执行这样的操作，如果x<y则result这个操作是tf.add(x,z),反之则是tf.square(y)。

因为z在cond函数中的至少一个分支被用到，所以z = tf.multiply(a, b)  总是被无条件执行.

### linspace和range

tf.linspace(start,stop,num,name=None)

     返回一个tensor，该tensor中的数值在start到stop区间之间取等差数列（包含start和stop），如果num>1则差值为(stop-start)/(num-1)，以保证最后一个元素的值为stop。

     其中，start和stop必须为tf.float32或tf.float64。num的类型为int。

     tf.linspace(10.0, 12.0, 3, name="linspace") => [ 10.0 11.0 12.0]

tf.range(start,limit=None,delta=1,name='range')

     返回一个tensor等差数列，该tensor中的数值在start到limit之间，不包括limit，delta是等差数列的差值。

     start，limit和delta都是int32类型。

     # 'start' is 3

     # 'limit' is 18

     # 'delta' is 3

     tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]

     # 'limit' is 5 start is 0

     tf.range(start, limit) ==> [0, 1, 2, 3, 4]

### split

沿着某一维度将tensor分离为num\_split tensors

tf.split(split\_dim, num\_split, value, name=’split’)

|  |
| --- |
| # ‘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] |

### slice

对tensor进行切片操作

tf.slice(input\_, begin, size, name=None)

#’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],

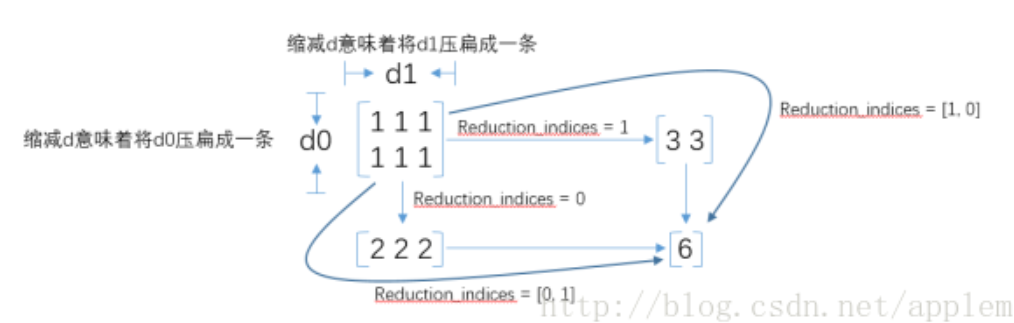
[4, 4, 4]]]

tf.slice(input, [1, 0, 0], [2, 1, 3]) ==>

[[[3, 3, 3]],

[[5, 5, 5]]]

### reduce\_sum



### tile

import tensorflow as tf

temp = tf.tile([1,2,3],[2])

temp2 = tf.tile([[1,2],[3,4],[5,6]],[2,3])

with tf.Session() as sess:

print(sess.run(temp))

print(sess.run(temp2))

[1 2 3 1 2 3]

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

import tensorflow as tf

temp = tf.tile([[1,2,3],[1,2,3]],[1,1])

temp2 = tf.tile([[1,2,3],[1,2,3]],[2,1])

temp3 = tf.tile([[1,2,3],[1,2,3]],[2,2])

with tf.Session() as sess:

print(sess.run(temp))

print(sess.run(temp2))

print(sess.run(temp3))

[[1 2 3]   
[1 2 3]]

[[1 2 3]   
[1 2 3]   
[1 2 3]   
[1 2 3]]

[[1 2 3 1 2 3]   
[1 2 3 1 2 3]   
[1 2 3 1 2 3]   
[1 2 3 1 2 3]]

### reshape

tf.reshape(tensor, shape, name=None)   
函数的作用是将tensor变换为参数shape的形式。   
其中shape为一个列表形式，特殊的一点是列表中可以存在-1。-1代表的含义是不用我们自己指定这一维的大小，函数会自动计算，但列表中只能存在一个-1。（当然如果存在多个-1，就是一个存在多解的方程了）

好了我想说的重点还有一个就是根据shape如何变换矩阵。其实简单的想就是，

reshape（t, shape） => reshape(t, [-1]) => reshape(t, shape)

首先将矩阵t变为一维矩阵，然后再对矩阵的形式更改就可以了。

### pack

打包

tf.pack(values, axis=0, name=’pack’)

|  |
| --- |
| # ‘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]) |

### np.dot

numpy的dot

两个array的相乘\*指的是对应元素的相乘；两个array的dot表示矩阵的相乘。

### multiply和matmul

tf.multiply是点乘，即Returns x \* y element-wise.

tf.matmul是矩阵乘法，即Multiplies matrix a by matrix b, producing a \* b.

### reverse

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]]]] |

### transpose

tf.transpose(a, perm=None, name=’transpose’)

调换tensor的维度顺序

|  |
| --- |
| 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]] |

### convert\_to\_tensor

m1 = [[1.0, 2.0],

[3.0, 4.0]]

m2 = np.array([[1.0, 2.0],

[3.0, 4.0]], dtype=np.float32)

m3 = tf.constant([[1.0, 2.0],

[3.0, 4.0]])

t1 = tf.convert\_to\_tensor(m1, dtype=tf.float32)

t2 = tf.convert\_to\_tensor(m2, dtype=tf.float32)

t3 = tf.convert\_to\_tensor(m3, dtype=tf.float32)

第一个变量m1是一个列表，第二个变量m2是NumPy类库中的一个ndarray，最后一个变量m3是TensorFlow的Tensor对象。TensorFlow中所有运算符（如neg）都设计为对张量对象进行操作。tf.convert\_to\_tensor（...）这个方法，我们可以用在任何地方，以确保我们处理张量而不是其他类型。TensorFlow类库中的大多数方法都已经调用了它。使用tf.convert\_to\_tensor（...）并不是必须的，在这里使用它，是因为它有助于我们理解隐式类型的跨类库处理。

### [tf.train.batch和tf.train.shuffle\_batch的用法](http://blog.csdn.net/wuguangbin1230/article/details/72810706)

tf.train.batch([example, label], batch\_size=batch\_size, capacity=capacity)：[example, label]表示样本和样本标签，这个可以是一个样本和一个样本标签，batch\_size是返回的一个batch样本集的样本个数。capacity是队列中的容量。这主要是按顺序组合成一个batch

tf.train.shuffle\_batch([example, label], batch\_size=batch\_size, capacity=capacity, min\_after\_dequeue)。这里面的参数和上面的一样的意思。不一样的是这个参数min\_after\_dequeue，一定要保证这参数大于capacity参数的值，否则会出错。这个代表队列中的元素大于它的时候就输出乱的顺序的batch。也就是说这个函数的输出结果是一个乱序的样本排列的batch，不是按照顺序排列的。

上面的函数返回值都是一个batch的样本和样本标签，只是一个是按照顺序，另外一个是随机的

## yield

我们先抛开 generator，以一个常见的编程题目来展示 yield 的概念。

如何生成斐波那契數列

斐波那契（Fibonacci）數列是一个非常简单的递归数列，除第一个和第二个数外，任意一个数都可由前两个数相加得到。用计算机程序输出斐波那契數列的前 N 个数是一个非常简单的问题，许多初学者都可以轻易写出如下函数：

清单 1. 简单输出斐波那契數列前 N 个数

def fab(max):

n, a, b = 0, 0, 1

while n < max:

print b

a, b = b, a + b

n = n + 1

执行 fab(5)，我们可以得到如下输出：

>>> fab(5)

1

1

2

3

5

结果没有问题，但有经验的开发者会指出，直接在 fab 函数中用 print 打印数字会导致该函数可复用性较差，因为 fab 函数返回 None，其他函数无法获得该函数生成的数列。

要提高 fab 函数的可复用性，最好不要直接打印出数列，而是返回一个 List。以下是 fab 函数改写后的第二个版本：

清单 2. 输出斐波那契數列前 N 个数第二版

def fab(max):

n, a, b = 0, 0, 1

L = []

while n < max:

L.append(b)

a, b = b, a + b

n = n + 1

return L

可以使用如下方式打印出 fab 函数返回的 List：

>>> for n in fab(5):

... print n

...

1

1

2

3

5

改写后的 fab 函数通过返回 List 能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List 来保存中间结果，而是通过 iterable 对象来迭代。例如，在 Python2.x 中，代码：

清单 3. 通过 iterable 对象来迭代

for i in range(1000): pass

会导致生成一个 1000 个元素的 List，而代码：

for i in xrange(1000): pass

则不会生成一个 1000 个元素的 List，而是在每次迭代中返回下一个数值，内存空间占用很小。因为 xrange 不返回 List，而是返回一个 iterable 对象。

利用 iterable 我们可以把 fab 函数改写为一个支持 iterable 的 class，以下是第三个版本的 Fab：

清单 4. 第三个版本

class Fab(object):

def \_\_init\_\_(self, max):

self.max = max

self.n, self.a, self.b = 0, 0, 1

def \_\_iter\_\_(self):

return self

def next(self):

if self.n < self.max:

r = self.b

self.a, self.b = self.b, self.a + self.b

self.n = self.n + 1

return r

raise StopIteration()

Fab 类通过 next() 不断返回数列的下一个数，内存占用始终为常数：

>>> for n in Fab(5):

... print n

...

1

1

2

3

5

然而，使用 class 改写的这个版本，代码远远没有第一版的 fab 函数来得简洁。如果我们想要保持第一版 fab 函数的简洁性，同时又要获得 iterable 的效果，yield 就派上用场了：

清单 5. 使用 yield 的第四版

def fab(max):

n, a, b = 0, 0, 1

while n < max:

yield b

# print b

a, b = b, a + b

n = n + 1

第四个版本的 fab 和第一版相比，仅仅把 print b 改为了 yield b，就在保持简洁性的同时获得了 iterable 的效果。

调用第四版的 fab 和第二版的 fab 完全一致：

>>> for n in fab(5):

... print n

...

1

1

2

3

5

简单地讲，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

也可以手动调用 fab(5) 的 next() 方法（因为 fab(5) 是一个 generator 对象，该对象具有 next() 方法），这样我们就可以更清楚地看到 fab 的执行流程：

清单 6. 执行流程

>>> f = fab(5)

>>> f.next()

1

>>> f.next()

1

>>> f.next()

2

>>> f.next()

3

>>> f.next()

5

>>> f.next()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

当函数执行结束时，generator 自动抛出 StopIteration 异常，表示迭代完成。在 for 循环里，无需处理 StopIteration 异常，循环会正常结束。

我们可以得出以下结论：

一个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

如何判断一个函数是否是一个特殊的 generator 函数？可以利用 isgeneratorfunction 判断：

清单 7. 使用 isgeneratorfunction 判断

>>> from inspect import isgeneratorfunction

>>> isgeneratorfunction(fab)

True

要注意区分 fab 和 fab(5)，fab 是一个 generator function，而 fab(5) 是调用 fab 返回的一个 generator，好比类的定义和类的实例的区别：

清单 8. 类的定义和类的实例

>>> import types

>>> isinstance(fab, types.GeneratorType)

False

>>> isinstance(fab(5), types.GeneratorType)

True

fab 是无法迭代的，而 fab(5) 是可迭代的：

>>> from collections import Iterable

>>> isinstance(fab, Iterable)

False

>>> isinstance(fab(5), Iterable)

True

每次调用 fab 函数都会生成一个新的 generator 实例，各实例互不影响：

>>> f1 = fab(3)

>>> f2 = fab(5)

>>> print 'f1:', f1.next()

f1: 1

>>> print 'f2:', f2.next()

f2: 1

>>> print 'f1:', f1.next()

f1: 1

>>> print 'f2:', f2.next()

f2: 1

>>> print 'f1:', f1.next()

f1: 2

>>> print 'f2:', f2.next()

f2: 2

>>> print 'f2:', f2.next()

f2: 3

>>> print 'f2:', f2.next()

f2: 5

另一个例子

另一个 yield 的例子来源于文件读取。如果直接对文件对象调用 read() 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。通过 yield，我们不再需要编写读文件的迭代类，就可以轻松实现文件读取：

清单 9. 另一个 yield 的例子

def read\_file(fpath):

BLOCK\_SIZE = 1024

with open(fpath, 'rb') as f:

while True:

block = f.read(BLOCK\_SIZE)

if block:

yield block

else:

return

以上仅仅简单介绍了 yield 的基本概念和用法，yield 在 Python 3 中还有更强大的用法，我们会在后续文章中讨论。

### layers.dense

import tensorflow as tf

batch\_size = 5

ones = tf.ones([batch\_size,6,8,20])

logits = tf.layers.dense(ones,10)

print(logits.get\_shape())

输出：

(5, 6, 8, 10)

定义：

dense(

    inputs,

    units,

    activation=None,

    use\_bias=True,

    kernel\_initializer=None,

    bias\_initializer=tf.zeros\_initializer(),

    kernel\_regularizer=None,

    bias\_regularizer=None,

    activity\_regularizer=None,

    trainable=True,

    name=None,

    reuse=None

)  
inputs: 输入数据，2维tensor.  
units: 该层的神经单元结点数。  
activation: 激活函数.  
use\_bias: Boolean型，是否使用偏置项.  
kernel\_initializer: 卷积核的初始化器.  
bias\_initializer: 偏置项的初始化器，默认初始化为0.  
kernel\_regularizer: 卷积核化的正则化，可选.  
bias\_regularizer: 偏置项的正则化，可选.  
activity\_regularizer: 输出的正则化函数.  
trainable: Boolean型，表明该层的参数是否参与训练。如果为真则变量加入到图集合中 GraphKeys.TRAINABLE\_VARIABLES (see tf.Variable).  
name: 层的名字.  
reuse: Boolean型, 是否重复使用参数.  
全连接层执行操作 outputs = activation(inputs.kernel + bias)  
  
如果执行结果不想进行激活操作，则设置activation=None。  
  
例：

#全连接层

dense1 = tf.layers.dense(inputs=pool3, units=1024, activation=tf.nn.relu)

dense2= tf.layers.dense(inputs=dense1, units=512, activation=tf.nn.relu)

logits= tf.layers.dense(inputs=dense2, units=10, activation=None)

也可以对全连接层的参数进行正则化约束：

dense1 = tf.layers.dense(inputs=pool3, units=1024, activation=tf.nn.relu，kernel\_regularizer=tf.contrib.layers.l2\_regularizer(0.003))

### TFRecords

TFRecords文件包含了tf.train.Example 协议内存块(protocol buffer)(协议内存块包含了字段 Features)。我们可以写一段代码获取你的数据， 将数据填入到Example协议内存块(protocol buffer)，将协议内存块序列化为一个字符串， 并且通过tf.python\_io.TFRecordWriter 写入到TFRecords文件。

从TFRecords文件中读取数据， 可以使用tf.TFRecordReader的tf.parse\_single\_example解析器。这个操作可以将Example协议内存块(protocol buffer)解析为张量。

from PIL import Image

writer = tf.python\_io.TFRecordWriter("train.tfrecords")for index, name in enumerate(classes):

class\_path = cwd + name + "/"

for img\_name in os.listdir(class\_path):

img\_path = class\_path + img\_name

img = Image.open(img\_path)

img = img.resize((224, 224))

img\_raw = img.tobytes() #将图片转化为原生bytes

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

"label": tf.train.Feature(int64\_list=tf.train.Int64List(value=[index])),

'img\_raw': tf.train.Feature(bytes\_list=tf.train.BytesList(value=[img\_raw]))

}))

writer.write(example.SerializeToString()) #序列化为字符串

writer.close()

关于Example Feature的相关定义和详细内容，我推荐去官网查看相关API。

基本的，一个Example中包含Features，Features里包含Feature（这里没s）的字典。最后，Feature里包含有一个 FloatList， 或者ByteList，或者Int64List

就这样，我们把相关的信息都存到了一个文件中，所以前面才说不用单独的label文件。而且读取也很方便。

for serialized\_example in tf.python\_io.tf\_record\_iterator("train.tfrecords"):

example = tf.train.Example()

example.ParseFromString(serialized\_example)

image = example.features.feature['image'].bytes\_list.value

label = example.features.feature['label'].int64\_list.value

# 可以做一些预处理之类的

print image, label

**使用队列读取**

img, label = read\_and\_decode("train.tfrecords")

#使用shuffle\_batch可以随机打乱输入

img\_batch, label\_batch = tf.train.shuffle\_batch([img,label], batch\_size=30,capacity=2000, min\_after\_dequeue=1000)

init = tf.initialize\_all\_variables()

with tf.Session() as sess:

sess.run(init)

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

for i in range(3):

val, l= sess.run([img\_batch, label\_batch])

#我们也可以根据需要对val， l进行处理

#l = to\_categorical(l, 12)

print(val.shape, l)

### reduce\_mean和reduce\_max

求最大值tf.reduce\_max(input\_tensor, reduction\_indices=None, keep\_dims=False, name=None)

求平均值tf.reduce\_mean(input\_tensor, reduction\_indices=None, keep\_dims=False, name=None)

参数1--input\_tensor:待求值的tensor。

参数2--reduction\_indices:在哪一维上求解。

参数（3）（4）可忽略

举例说明：

# 'x' is [[1., 2.]# [3., 4.]]

x是一个2维数组，分别调用reduce\_\*函数如下：

首先求平均值：

tf.reduce\_mean(x) ==> 2.5 #如果不指定第二个参数，那么就在所有的元素中取平均值

tf.reduce\_mean(x, 0) ==> [2., 3.] #指定第二个参数为0，则第一维的元素取平均值，即每一列求平均值

tf.reduce\_mean(x, 1) ==> [1.5, 3.5] #

### sequence\_loss\_by\_example

sequence\_loss\_by\_example(logits, targets, weights）

这个函数用于计算所有examples（假设一句话有n个单词，一个单词及单词所对应的label就是一个example,所有example就是一句话中所有单词）的加权交叉熵损失，logits参数是一个2D Tensor构成的列表对象，每一个2D Tensor的尺寸为[batch\_size x num\_decoder\_symbols]，函数的返回值是一个1D float类型的Tensor，尺寸为batch\_size，其中的每一个元素代表当前输入序列example的交叉熵。另外，还有一个与之类似的函数sequence\_loss，它对sequence\_loss\_by\_example函数返回的结果进行了一个tf.reduce\_sum运算，因此返回的是一个标称型float Tensor。   
进一步理解：   
logits 的shape = [batch\_size\*numsteps, vocab\_size], vocab\_size是（分类）类别的个数   
targets 的shape = [batch\_size\*num\_steps]   
sequence\_loss\_by\_example的做法是，针对logits中的每一个num\_step,即[batch\_size, vocab\_size], 对所有vocab\_size个预测结果，得出预测值最大的那个类别，与target中的值相比较计算Loss值

### [legacy\_seq2seq](https://lan2720.github.io/2017/03/10/tensorflow%E7%9A%84legacy-seq2seq/" \t "http://blog.csdn.net/u012871493/article/details/_blank)

tensorflow要重新给出一套seq2seq的接口，把之前的seq2seq搬到了legacy\_seq2seq下，今天读的就是来自[这里](https://github.com/tensorflow/tensorflow/blob/r1.0/tensorflow/contrib/legacy_seq2seq/python/ops/seq2seq.py" \t "http://blog.csdn.net/u012871493/article/details/_blank)的代码。目前很多代码还是使用了老的seq2seq接口，因此仍有熟悉的必要。

#### \_extract\_argmax\_and\_embed

|  |  |
| --- | --- |
| 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 | def \_extract\_argmax\_and\_embed(embedding,  output\_projection=None,  update\_embedding=True):  """Get a loop\_function that extracts the previous symbol and embeds it.  Args:  embedding: embedding tensor for symbols.  output\_projection: None or a pair (W, B). If provided, each fed previous  output will first be multiplied by W and added B.  update\_embedding: Boolean; if False, the gradients will not propagate  through the embeddings.  Returns:  A loop function.  """  def loop\_function(prev, \_):  if output\_projection is not None:  prev = nn\_ops.xw\_plus\_b(prev, output\_projection[0], output\_projection[1])  prev\_symbol = math\_ops.argmax(prev, 1)  # Note that gradients will not propagate through the second parameter of  # embedding\_lookup.  emb\_prev = embedding\_ops.embedding\_lookup(embedding, prev\_symbol)  if not update\_embedding:  emb\_prev = array\_ops.stop\_gradient(emb\_prev)  return emb\_prev  return loop\_function |

#### rnn\_decoder

|  |  |
| --- | --- |
| 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 | def rnn\_decoder(decoder\_inputs,  initial\_state,  cell,  loop\_function=None,  scope=None):  """RNN decoder for the sequence-to-sequence model.  Args:  decoder\_inputs: A list of 2D Tensors [batch\_size x input\_size].  initial\_state: 2D Tensor with shape [batch\_size x cell.state\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function and size.  loop\_function: If not None, this function will be applied to the i-th output  in order to generate the i+1-st input, and decoder\_inputs will be ignored,  except for the first element ("GO" symbol). This can be used for decoding,  but also for training to emulate http://arxiv.org/abs/1506.03099.  Signature -- loop\_function(prev, i) = next  \* prev is a 2D Tensor of shape [batch\_size x output\_size],  \* i is an integer, the step number (when advanced control is needed),  \* next is a 2D Tensor of shape [batch\_size x input\_size].  scope: VariableScope for the created subgraph; defaults to "rnn\_decoder".  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors with  shape [batch\_size x output\_size] containing generated outputs.  state: The state of each cell at the final time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  (Note that in some cases, like basic RNN cell or GRU cell, outputs and  states can be the same. They are different for LSTM cells though.)  """  with variable\_scope.variable\_scope(scope or "rnn\_decoder"):  state = initial\_state  outputs = []  prev = None  for i, inp in enumerate(decoder\_inputs):  if loop\_function is not None and prev is not None:  with variable\_scope.variable\_scope("loop\_function", reuse=True):  inp = loop\_function(prev, i)  if i > 0:  variable\_scope.get\_variable\_scope().reuse\_variables()  output, state = cell(inp, state)  outputs.append(output)  if loop\_function is not None:  prev = output  return outputs, state |

decoder\_inputs：是a list，其中的每一个元素表示的是t\_i时刻的输入，每一时刻的输入又会有batch\_size个，每一个输入（通差是表示一个word或token）又是input\_size维度的。  
loop\_function: 如果loop\_function有设置的话，decoder input中第一个”GO”会输入，但之后时刻的input就会被忽略，取代的是input\_ti+1 = loop\_function(output\_ti)  
这里定义的loop\_function，有2个参数，（prev,i），输出为next

输出：  
outputs：既然是每一时刻的input都会对应得到一个output，自然outputs的shape和decoder\_inputs是一样，是a list，每个元素的shape=[batch\_size, input\_size]（但是这里为了区别，认为是output\_size）  
state：最后一个时刻t的cell state，shape=[batch\_size, cell.state\_size]

#### basic\_rnn\_seq2seq

|  |  |
| --- | --- |
| 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 | def basic\_rnn\_seq2seq(encoder\_inputs,  decoder\_inputs,  cell,  dtype=dtypes.float32,  scope=None):  """Basic RNN sequence-to-sequence model.  This model first runs an RNN to encode encoder\_inputs into a state vector,  then runs decoder, initialized with the last encoder state, on decoder\_inputs.  Encoder and decoder use the same RNN cell type, but don't share parameters.  Args:  encoder\_inputs: A list of 2D Tensors [batch\_size x input\_size].  decoder\_inputs: A list of 2D Tensors [batch\_size x input\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function and size.  dtype: The dtype of the initial state of the RNN cell (default: tf.float32).  scope: VariableScope for the created subgraph; default: "basic\_rnn\_seq2seq".  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors with  shape [batch\_size x output\_size] containing the generated outputs.  state: The state of each decoder cell in the final time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  """  with variable\_scope.variable\_scope(scope or "basic\_rnn\_seq2seq"):  enc\_cell = copy.deepcopy(cell)  \_, enc\_state = core\_rnn.static\_rnn(enc\_cell, encoder\_inputs, dtype=dtype)  return rnn\_decoder(decoder\_inputs, enc\_state, cell) |

encoder\_inputs：a list，每个元素是时刻t的输入，每一时刻又存在batch\_size个输入（word or token），并且每个token用input\_size来表示（embedding）。因此，是a list of [batch\_size, input\_size]  
decoder\_inputs：同上，但是这两个list的长度可能不同，前者根据encoder\_max\_length指定，decoder根据decoder\_max\_length指定。  
输出：  
outputs：shape和decoder\_inputs相同，差别在于这里用output\_size和input\_size区别【why  
state：还是最后一个时刻的cell state，[batch\_size, cell.state\_size]

注意到这里用到深拷贝:

深拷贝是在另一块地址中创建一个新的变量或容器，同时容器内的元素的地址也是新开辟的，仅仅是值相同而已，是完全的副本。也就是说（ 新瓶装新酒 ）。

encode阶段使用的是core\_rnn.static\_rnn()不知道这个函数和别的rnn有什么不同？

decode阶段，很基本，直接使用了上面提到的rnn\_decoder来生成最后的outputs和state，返回。

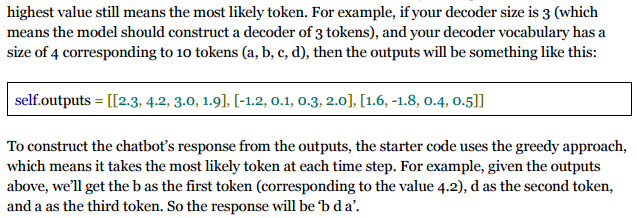
static\_rnn

代码[在这](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/rnn/python/ops/core_rnn.py" \l "L41" \t "http://blog.csdn.net/u012871493/article/details/_blank)，比较繁琐，就不详细解读了。

#### embedding\_rnn\_decoder

|  |  |
| --- | --- |
| 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 | def embedding\_rnn\_decoder(decoder\_inputs,  initial\_state,  cell,  num\_symbols,  embedding\_size,  output\_projection=None,  feed\_previous=False,  update\_embedding\_for\_previous=True,  scope=None):  """RNN decoder with embedding and a pure-decoding option.  Args:  decoder\_inputs: A list of 1D batch-sized int32 Tensors (decoder inputs).  initial\_state: 2D Tensor [batch\_size x cell.state\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function.  num\_symbols: Integer, how many symbols come into the embedding.  embedding\_size: Integer, the length of the embedding vector for each symbol.  output\_projection: None or a pair (W, B) of output projection weights and  biases; W has shape [output\_size x num\_symbols] and B has  shape [num\_symbols]; if provided and feed\_previous=True, each fed  previous output will first be multiplied by W and added B.  feed\_previous: Boolean; if True, only the first of decoder\_inputs will be  used (the "GO" symbol), and all other decoder inputs will be generated by:  next = embedding\_lookup(embedding, argmax(previous\_output)),  In effect, this implements a greedy decoder. It can also be used  during training to emulate http://arxiv.org/abs/1506.03099.  If False, decoder\_inputs are used as given (the standard decoder case).  update\_embedding\_for\_previous: Boolean; if False and feed\_previous=True,  only the embedding for the first symbol of decoder\_inputs (the "GO"  symbol) will be updated by back propagation. Embeddings for the symbols  generated from the decoder itself remain unchanged. This parameter has  no effect if feed\_previous=False.  scope: VariableScope for the created subgraph; defaults to  "embedding\_rnn\_decoder".  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors. The  output is of shape [batch\_size x cell.output\_size] when  output\_projection is not None (and represents the dense representation  of predicted tokens). It is of shape [batch\_size x num\_decoder\_symbols]  when output\_projection is None.  state: The state of each decoder cell in each time-step. This is a list  with length len(decoder\_inputs) -- one item for each time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  Raises:  ValueError: When output\_projection has the wrong shape.  """  with variable\_scope.variable\_scope(scope or "embedding\_rnn\_decoder") as scope:  if output\_projection is not None:  dtype = scope.dtype  proj\_weights = ops.convert\_to\_tensor(output\_projection[0], dtype=dtype)  proj\_weights.get\_shape().assert\_is\_compatible\_with([None, num\_symbols])  proj\_biases = ops.convert\_to\_tensor(output\_projection[1], dtype=dtype)  proj\_biases.get\_shape().assert\_is\_compatible\_with([num\_symbols])  embedding = variable\_scope.get\_variable("embedding",  [num\_symbols, embedding\_size])  loop\_function = \_extract\_argmax\_and\_embed(  embedding, output\_projection,  update\_embedding\_for\_previous) if feed\_previous else None  emb\_inp = (embedding\_ops.embedding\_lookup(embedding, i)  for i in decoder\_inputs)  return rnn\_decoder(  emb\_inp, initial\_state, cell, loop\_function=loop\_function) |

刚才讲了一个basic的decoder叫rnn\_decoder：rnn\_decoder(decoder\_inputs,initial\_state,cell,loop\_function=None,scope=None)，现在来一个稍微高级一点的。  
对比一下发现这个decoder没有loop\_function，多出来了num\_symbols，embedding\_size，output\_projection=None，feed\_previous=False，update\_embedding\_for\_previous=True。这些都是什么呢？

参数：  
decoder\_inputs：既然这个标榜了embedding，那么input肯定和rnn\_decoder有些不同。这里input变为1维，[batch\_size, ]也就是说，输入不需要自己做embedding了，直接输入tokens在vocab中对应的idx（即ids）即可，内部会自动帮我们进行id到embedding的转化。  
num\_symbols：就是vocab\_size  
embedding\_size：每个token需要embedding成的维数，比如100  
output\_projection：(W, b)就是将输出做一个映射。为什么要映射，因为此时input相当于a list of [batch\_size, 1]，内部帮我们做一个embedding，得到embedded\_input=[batch\_size, embedding\_size ]，经过cell之后，得到[batch\_size, output\_size]（这个过程就是之前的rnn\_decoder做的事情）。这样之后，如果我们设置了feed\_previous=True，也就是需要将前一时刻的output作为下一时刻的input，那么前一时刻的output中要从vocab\_size中选出一个分数最高的token来，即argmax(previous\_output)。过程如下图描述的那样：  
[](http://ww1.sinaimg.cn/large/901f9a6fly1fdggohp275j20hp062ta6" \o "" \t "http://blog.csdn.net/u012871493/article/details/_blank)  
但是，现在的output维度是output\_size，并不能知道每个vocab的得分情况。因此要从output\_size映射到vocab\_size（这里的num\_symbols）。  
我们知道，x(某一时刻的output)的shape=[batch\_size, output\_size]，映射的公式是xw+b，那么w的shape=[output\_zize, num\_symbols]

update\_embedding\_for\_previous：如果前一时刻的output不作为当前的input的话(feed\_previous=False)，这个参数没影响（）；否则，该参数默认是True，但如果设置成false，则表示不对前一个embedding进行更新，那么bp的时候只会更新”GO”的embedding，其他token（decoder生成的）embedding不变。

输出：  
outputs：如果output\_projection=None的话，也就是不进行映射(直接输出的是num\_symbols的个数)，那么a list of [batch\_size, num\_symbols]；如果不为None，说明outputs要进行映射，则outputs是a list of [batch\_size, num\_symbols]  
state同上

#### embedding\_rnn\_seq2seq

|  |  |
| --- | --- |
| 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 | def embedding\_rnn\_seq2seq(encoder\_inputs,  decoder\_inputs,  cell,  num\_encoder\_symbols,  num\_decoder\_symbols,  embedding\_size,  output\_projection=None,  feed\_previous=False,  dtype=None,  scope=None):  """Embedding RNN sequence-to-sequence model.  This model first embeds encoder\_inputs by a newly created embedding (of shape  [num\_encoder\_symbols x input\_size]). Then it runs an RNN to encode  embedded encoder\_inputs into a state vector. Next, it embeds decoder\_inputs  by another newly created embedding (of shape [num\_decoder\_symbols x  input\_size]). Then it runs RNN decoder, initialized with the last  encoder state, on embedded decoder\_inputs.  Args:  encoder\_inputs: A list of 1D int32 Tensors of shape [batch\_size].  decoder\_inputs: A list of 1D int32 Tensors of shape [batch\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function and size.  num\_encoder\_symbols: Integer; number of symbols on the encoder side.  num\_decoder\_symbols: Integer; number of symbols on the decoder side.  embedding\_size: Integer, the length of the embedding vector for each symbol.  output\_projection: None or a pair (W, B) of output projection weights and  biases; W has shape [output\_size x num\_decoder\_symbols] and B has  shape [num\_decoder\_symbols]; if provided and feed\_previous=True, each  fed previous output will first be multiplied by W and added B.  feed\_previous: Boolean or scalar Boolean Tensor; if True, only the first  of decoder\_inputs will be used (the "GO" symbol), and all other decoder  inputs will be taken from previous outputs (as in embedding\_rnn\_decoder).  If False, decoder\_inputs are used as given (the standard decoder case).  dtype: The dtype of the initial state for both the encoder and encoder  rnn cells (default: tf.float32).  scope: VariableScope for the created subgraph; defaults to  "embedding\_rnn\_seq2seq"  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors. The  output is of shape [batch\_size x cell.output\_size] when  output\_projection is not None (and represents the dense representation  of predicted tokens). It is of shape [batch\_size x num\_decoder\_symbols]  when output\_projection is None.  state: The state of each decoder cell in each time-step. This is a list  with length len(decoder\_inputs) -- one item for each time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  """ |

既然有了embedding\_rnn\_decoder，那么对应的就有embedding\_rnn\_seq2seq。之前讲过basic\_rnn\_seq2seq(encoder\_inputs, decoder\_inputs, cell, dtype=dtypes.float32, scope=None)  
inputs：还是像之前说的，既然embedding是内部帮我们完成，则inputs shape= a list of [batch\_size]，每个位置都只是一个token id。内部使用一个embedding wrapper，做lookup，生成a list of [batch\_size, embedding\_size]  
对比之下，多了几个参数：  
num\_encoder\_symbols：通俗的说其实就是encoder端的vocab\_size。enc和dec两端词汇量不同主要在于不同语言的translate task中，如果单纯是中文到中文的生成，不存在两端词汇量的不同。  
num\_decoder\_symbols：同上  
embedding\_size：每个vocab需要用多少维的vector表示  
output\_projection=None：  
feed\_previous=False：如果feed\_previous只是简单的一个True or False，则直接返回embedding\_rnn\_decoder的结果。重点是feed\_previous还能传入一个boolean tensor（暂时无此需求）

#### attention\_decoder

|  |  |
| --- | --- |
| 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 | def attention\_decoder(decoder\_inputs,  initial\_state,  attention\_states,  cell,  output\_size=None,  num\_heads=1,  loop\_function=None,  dtype=None,  scope=None,  initial\_state\_attention=False):  """RNN decoder with attention for the sequence-to-sequence model.  In this context "attention" means that, during decoding, the RNN can look up  information in the additional tensor attention\_states, and it does this by  focusing on a few entries from the tensor. This model has proven to yield  especially good results in a number of sequence-to-sequence tasks. This  implementation is based on http://arxiv.org/abs/1412.7449 (see below for  details). It is recommended for complex sequence-to-sequence tasks.  Args:  decoder\_inputs: A list of 2D Tensors [batch\_size x input\_size].  initial\_state: 2D Tensor [batch\_size x cell.state\_size].  attention\_states: 3D Tensor [batch\_size x attn\_length x attn\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function and size.  output\_size: Size of the output vectors; if None, we use cell.output\_size.  num\_heads: Number of attention heads that read from attention\_states.  loop\_function: If not None, this function will be applied to i-th output  in order to generate i+1-th input, and decoder\_inputs will be ignored,  except for the first element ("GO" symbol). This can be used for decoding,  but also for training to emulate http://arxiv.org/abs/1506.03099.  Signature -- loop\_function(prev, i) = next  \* prev is a 2D Tensor of shape [batch\_size x output\_size],  \* i is an integer, the step number (when advanced control is needed),  \* next is a 2D Tensor of shape [batch\_size x input\_size].  dtype: The dtype to use for the RNN initial state (default: tf.float32).  scope: VariableScope for the created subgraph; default: "attention\_decoder".  initial\_state\_attention: If False (default), initial attentions are zero.  If True, initialize the attentions from the initial state and attention  states -- useful when we wish to resume decoding from a previously  stored decoder state and attention states.  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors of  shape [batch\_size x output\_size]. These represent the generated outputs.  Output i is computed from input i (which is either the i-th element  of decoder\_inputs or loop\_function(output {i-1}, i)) as follows.  First, we run the cell on a combination of the input and previous  attention masks:  cell\_output, new\_state = cell(linear(input, prev\_attn), prev\_state).  Then, we calculate new attention masks:  new\_attn = softmax(V^T \* tanh(W \* attention\_states + U \* new\_state))  and then we calculate the output:  output = linear(cell\_output, new\_attn).  state: The state of each decoder cell the final time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  Raises:  ValueError: when num\_heads is not positive, there are no inputs, shapes  of attention\_states are not set, or input size cannot be inferred  from the input.  """ |

刚才讲完了embedding\_rnn\_decoder，则再来看看attention\_decoder。  
和基本的rnn\_decoder相比（rnn\_decoder(decoder\_inputs, initial\_state, cell, loop\_function=None, scope=None)）  
多了几个参数：  
attention\_states：attention\_states作为addition info出现，  
output\_size=None：如果是None的话默认为cell.output\_size  
num\_heads=1 :应该pay attention的点的个数，比如要focus到attention\_states的几个点，默认为只关注1个点  
initial\_state\_attention=False：如果是True的话，attention由state和attention\_states进行初始化，如果False，则attention初始化为0

#### embedding\_attention\_decoder

|  |  |
| --- | --- |
| 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 | def embedding\_attention\_decoder(decoder\_inputs,  initial\_state,  attention\_states,  cell,  num\_symbols,  embedding\_size,  num\_heads=1,  output\_size=None,  output\_projection=None,  feed\_previous=False,  update\_embedding\_for\_previous=True,  dtype=None,  scope=None,  initial\_state\_attention=False):  """RNN decoder with embedding and attention and a pure-decoding option.  Args:  decoder\_inputs: A list of 1D batch-sized int32 Tensors (decoder inputs).  initial\_state: 2D Tensor [batch\_size x cell.state\_size].  attention\_states: 3D Tensor [batch\_size x attn\_length x attn\_size].  cell: core\_rnn\_cell.RNNCell defining the cell function.  num\_symbols: Integer, how many symbols come into the embedding.  embedding\_size: Integer, the length of the embedding vector for each symbol.  num\_heads: Number of attention heads that read from attention\_states.  output\_size: Size of the output vectors; if None, use output\_size.  output\_projection: None or a pair (W, B) of output projection weights and  biases; W has shape [output\_size x num\_symbols] and B has shape  [num\_symbols]; if provided and feed\_previous=True, each fed previous  output will first be multiplied by W and added B.  feed\_previous: Boolean; if True, only the first of decoder\_inputs will be  used (the "GO" symbol), and all other decoder inputs will be generated by:  next = embedding\_lookup(embedding, argmax(previous\_output)),  In effect, this implements a greedy decoder. It can also be used  during training to emulate http://arxiv.org/abs/1506.03099.  If False, decoder\_inputs are used as given (the standard decoder case).  update\_embedding\_for\_previous: Boolean; if False and feed\_previous=True,  only the embedding for the first symbol of decoder\_inputs (the "GO"  symbol) will be updated by back propagation. Embeddings for the symbols  generated from the decoder itself remain unchanged. This parameter has  no effect if feed\_previous=False.  dtype: The dtype to use for the RNN initial states (default: tf.float32).  scope: VariableScope for the created subgraph; defaults to  "embedding\_attention\_decoder".  initial\_state\_attention: If False (default), initial attentions are zero.  If True, initialize the attentions from the initial state and attention  states -- useful when we wish to resume decoding from a previously  stored decoder state and attention states.  Returns:  A tuple of the form (outputs, state), where:  outputs: A list of the same length as decoder\_inputs of 2D Tensors with  shape [batch\_size x output\_size] containing the generated outputs.  state: The state of each decoder cell at the final time-step.  It is a 2D Tensor of shape [batch\_size x cell.state\_size].  Raises:  ValueError: When output\_projection has the wrong shape.  """ |

其实是前面讲的embedding\_decoder和attention\_decoder的结合版。

#### embedding\_attention\_seq2seq

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def embedding\_attention\_seq2seq(encoder\_inputs,  decoder\_inputs,  cell,  num\_encoder\_symbols,  num\_decoder\_symbols,  embedding\_size,  num\_heads=1,  output\_projection=None,  feed\_previous=False,  dtype=None,  scope=None,  initial\_state\_attention=False) |

与embedding\_attention\_decoder相对应的seq2seq模型

#### sequence\_loss\_by\_example

|  |  |
| --- | --- |
| 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 | def sequence\_loss\_by\_example(logits,  targets,  weights,  average\_across\_timesteps=True,  softmax\_loss\_function=None,  name=None):  """Weighted cross-entropy loss for a sequence of logits (per example).  Args:  logits: List of 2D Tensors of shape [batch\_size x num\_decoder\_symbols].  targets: List of 1D batch-sized int32 Tensors of the same length as logits.  weights: List of 1D batch-sized float-Tensors of the same length as logits.  average\_across\_timesteps: If set, divide the returned cost by the total  label weight.  softmax\_loss\_function: Function (labels-batch, inputs-batch) -> loss-batch  to be used instead of the standard softmax (the default if this is None).  name: Optional name for this operation, default: "sequence\_loss\_by\_example".  Returns:  1D batch-sized float Tensor: The log-perplexity for each sequence.  Raises:  ValueError: If len(logits) is different from len(targets) or len(weights).  """  if len(targets) != len(logits) or len(weights) != len(logits):  raise ValueError("Lengths of logits, weights, and targets must be the same "  "%d, %d, %d." % (len(logits), len(weights), len(targets)))  with ops.name\_scope(name, "sequence\_loss\_by\_example",  logits + targets + weights):  log\_perp\_list = []  for logit, target, weight in zip(logits, targets, weights):  if softmax\_loss\_function is None:  # TODO(irving,ebrevdo): This reshape is needed because  # sequence\_loss\_by\_example is called with scalars sometimes, which  # violates our general scalar strictness policy.  target = array\_ops.reshape(target, [-1])  crossent = nn\_ops.sparse\_softmax\_cross\_entropy\_with\_logits(  labels=target, logits=logit)  else:  crossent = softmax\_loss\_function(target, logit)  log\_perp\_list.append(crossent \* weight)  log\_perps = math\_ops.add\_n(log\_perp\_list)  if average\_across\_timesteps:  total\_size = math\_ops.add\_n(weights)  total\_size += 1e-12 # Just to avoid division by 0 for all-0 weights.  log\_perps /= total\_size  return log\_perps |

返回值：  
1D batch-sized float Tensor：为每一个序列（一个batch中有batch\_size个sequence）计算其log perplexity，也是名称中by\_example的含义

输入：  
(注意：一个batch上的所有数据都被pad成相同长度？因此它们的time\_length是一样的？)  
logits：a list依次存储一系列时刻上的输出，每一时刻的输出都是batch\_size为单位的，其中的每一个输入对应的输出是整个vocab上的得分，因此是num\_decoder\_symbols。因此，logits应该是a list of [batch\_size, num\_decoder\_symbols]  
targets：a list表示依次的所有时刻的target，每一时刻又有batch\_size个输入，因此对应batch\_size个target，因此shape=a list of [batch\_size, ]  
weights：每个example，在每一时刻都有对自身当前token的权重。因此shape=a list of [batch\_size,]  
疑问：weights是做什么用的？为什么要对每个token设置权重？

解读代码：  
首先会生成一个crossent，shape=[batch\_size, ]，再和weights相乘，还是得到[batch\_size, ]，表示每个example在当前时刻t位置的得分(batch\_size个)，append到log\_perp\_list中（最终shape是a list of [batch\_size, ]）  
所有的time length循环完毕之后，累加这些time length，得到一个shape=[batch\_size,]的变量，叫做log\_perps。

#### sequence\_loss

|  |  |
| --- | --- |
| 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 | def sequence\_loss(logits,  targets,  weights,  average\_across\_timesteps=True,  average\_across\_batch=True,  softmax\_loss\_function=None,  name=None):  """Weighted cross-entropy loss for a sequence of logits, batch-collapsed.  Args:  logits: List of 2D Tensors of shape [batch\_size x num\_decoder\_symbols].  targets: List of 1D batch-sized int32 Tensors of the same length as logits.  weights: List of 1D batch-sized float-Tensors of the same length as logits.  average\_across\_timesteps: If set, divide the returned cost by the total  label weight.  average\_across\_batch: If set, divide the returned cost by the batch size.  softmax\_loss\_function: Function (labels-batch, inputs-batch) -> loss-batch  to be used instead of the standard softmax (the default if this is None).  name: Optional name for this operation, defaults to "sequence\_loss".  Returns:  A scalar float Tensor: The average log-perplexity per symbol (weighted).  Raises:  ValueError: If len(logits) is different from len(targets) or len(weights).  """  with ops.name\_scope(name, "sequence\_loss", logits + targets + weights):  cost = math\_ops.reduce\_sum(  sequence\_loss\_by\_example(  logits,  targets,  weights,  average\_across\_timesteps=average\_across\_timesteps,  softmax\_loss\_function=softmax\_loss\_function))  if average\_across\_batch:  batch\_size = array\_ops.shape(targets[0])[0]  return cost / math\_ops.cast(batch\_size, cost.dtype)  else:  return cost |

其实主体还是上面讲的sequence\_loss\_by\_example，只不过对上面的[batch\_size,]的结果进行sum，如果默认average\_across\_batch的话，就sum/batch\_size，平均每一个sequence的log perplexity；要是设置了不平均，则返回的是整个batch上的sum of log perplexity

#### model\_with\_buckets

|  |  |
| --- | --- |
| 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 | def model\_with\_buckets(encoder\_inputs,  decoder\_inputs,  targets,  weights,  buckets,  seq2seq,  softmax\_loss\_function=None,  per\_example\_loss=False,  name=None):  """Create a sequence-to-sequence model with support for bucketing.  The seq2seq argument is a function that defines a sequence-to-sequence model,  e.g., seq2seq = lambda x, y: basic\_rnn\_seq2seq(  x, y, core\_rnn\_cell.GRUCell(24))  Args:  encoder\_inputs: A list of Tensors to feed the encoder; first seq2seq input.  decoder\_inputs: A list of Tensors to feed the decoder; second seq2seq input.  targets: A list of 1D batch-sized int32 Tensors (desired output sequence).  weights: List of 1D batch-sized float-Tensors to weight the targets.  buckets: A list of pairs of (input size, output size) for each bucket.  seq2seq: A sequence-to-sequence model function; it takes 2 input that  agree with encoder\_inputs and decoder\_inputs, and returns a pair  consisting of outputs and states (as, e.g., basic\_rnn\_seq2seq).  softmax\_loss\_function: Function (labels-batch, inputs-batch) -> loss-batch  to be used instead of the standard softmax (the default if this is None).  per\_example\_loss: Boolean. If set, the returned loss will be a batch-sized  tensor of losses for each sequence in the batch. If unset, it will be  a scalar with the averaged loss from all examples.  name: Optional name for this operation, defaults to "model\_with\_buckets".  Returns:  A tuple of the form (outputs, losses), where:  outputs: The outputs for each bucket. Its j'th element consists of a list  of 2D Tensors. The shape of output tensors can be either  [batch\_size x output\_size] or [batch\_size x num\_decoder\_symbols]  depending on the seq2seq model used.  losses: List of scalar Tensors, representing losses for each bucket, or,  if per\_example\_loss is set, a list of 1D batch-sized float Tensors.  Raises:  ValueError: If length of encoder\_inputs, targets, or weights is smaller  than the largest (last) bucket.  """ |

参数：  
encoder\_inputs：一开始我有个疑问，这里的inputs是ids的形式还是传入input\_size的形式，仔细想想实际是这样的。这个inputs具体的shape形式要根据后面seq2seq定义的那个函数决定，一般就只传入两个参数x, y分别对应encoder\_inputs和decoder\_inputs（另外特定seq2seq需要的参数需要在自定义的这个seq2seq函数内部传入）。这个时候，如果我们使用的是embedding\_seq2seq，那么实际的inputs就应该是ids的样子；否则，就是input\_size的样子。  
targets：a list因为每一时刻都会有target，并且每一时刻输入的是batch\_size个，因此每一时刻的target是[batch\_size,]的形式，最终导致targets是a list of [batch\_size, ]  
buckets：a list of (input\_size, output\_size)  
per\_example\_loss：默认是False，表示losses是[batch\_size, ]。比如刚才讲到的sequence\_loss\_by\_example的结果是[batch\_size,]，再者sequence\_loss的结果是一个scalar。

实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for j, bucket in enumerate(buckets):  with variable\_scope.variable\_scope(  variable\_scope.get\_variable\_scope(), reuse=True if j > 0 else None):  bucket\_outputs, \_ = seq2seq(encoder\_inputs[:bucket[0]],  decoder\_inputs[:bucket[1]])  outputs.append(bucket\_outputs) |

根据实现可以看到，比如设置了3个buckets=[(2, 4), (5, 7), (8, 10)]，第1个bucket是(2,4)，那么先截取encoder\_inputs中每个（batch\_size个）sequences的前2个tokens，和同理截取decoder\_inputs中前4个tokens（encoder\_inputs的第一维度就是time）。  
然后把截取部分进行seq2seq，得到输出是a list of [batch\_size, output\_size]（这个list的长度为4，output是按decoder的长度算），然后将这个输出加入到outputs中。  
最终得到的outputs就是一个bucket\_size长度（这里为3）的列表，列表中每个元素是长度不等的list（之所以长度不等是因为每个bucket所定义的max\_decoder\_length不等，依次增大）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | if per\_example\_loss:  losses.append(  sequence\_loss\_by\_example(  outputs[-1],  targets[:bucket[1]],  weights[:bucket[1]],  softmax\_loss\_function=softmax\_loss\_function))  else:  losses.append(  sequence\_loss(  outputs[-1],  targets[:bucket[1]],  weights[:bucket[1]],  softmax\_loss\_function=softmax\_loss\_function)) |

计算完当前bucket的outputs后，就应该计算当前bucket的loss。由于当前bucket的output刚刚append，因此outputs[-1]就是当前bucket的output。又因为我们截取了decoder\_inputs，因此targets和weights都要截取成相同的长度。这样的话就得到当前bucket的loss，append到losses中。

因此，最后的outputs和losses，我们只要索引bucket的idx，就可以得到该bucket上的output和loss

### tf.trainable\_variables & tf.all\_variables

tf.trainable\_variables返回的是需要训练的变量列表

tf.all\_variables返回的是所有变量的列表

### tf.clip\_by\_global\_norm

Gradient Clipping的引入是为了处理gradient explosion或者gradients vanishing的问题。当在一次迭代中权重的更新过于迅猛的话，很容易导致loss divergence。Gradient Clipping的直观作用就是让权重的更新限制在一个合适的范围。

具体的细节是   
１．在solver中先设置一个clip\_gradient   
２．在前向传播与反向传播之后，我们会得到每个权重的梯度diff，这时不像通常那样直接使用这些梯度进行权重更新，而是先求所有权重梯度的平方和sumsq\_diff，如果sumsq\_diff > clip\_gradient，则求缩放因子scale\_factor = clip\_gradient / sumsq\_diff。这个scale\_factor在(0,1)之间。如果权重梯度的平方和sumsq\_diff越大，那缩放因子将越小。   
３．最后将所有的权重梯度乘以这个缩放因子，这时得到的梯度才是最后的梯度信息。

这样就保证了在一次迭代更新中，所有权重的梯度的平方和在一个设定范围以内，这个范围就是clip\_gradient.

tf.clip\_by\_global\_norm

tf.clip\_by\_global\_norm(t\_list, clip\_norm, use\_norm=None, name=None)

通过权重梯度的总和的比率来截取多个张量的值。   
t\_list 是梯度张量， clip\_norm 是截取的比率, 这个函数返回截取过的梯度张量和一个所有张量的全局范数。

t\_list[i] 的更新公式如下:

t\_list[i] \* clip\_norm / max(global\_norm, clip\_norm)

其中global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list]))   
global\_norm 是所有梯度的平方和，如果 clip\_norm > global\_norm ，就不进行截取。   
但是这个函数的速度比clip\_by\_norm() 要慢，因为在截取之前所有的参数都要准备好。

## Tip

虽然placeholder的shape参数是可选的，但有了它，TensorFlow能够自动捕捉因数据维度不一致导致的错误。

组装图阶段，Graph类不是线程安全的，添加operations最好在单线程内完成

## tf.clip\_by\_value

tf.clip\_by\_value(A, min, max)：输入一个张量A，把A中的每一个元素的值都压缩在min和max之间。小于min的让它等于min，大于max的元素的值等于max。

例如：

[python] view plain copy

import tensorflow as tf;

import numpy as np;

A = np.array([[1,1,2,4], [3,4,8,5]])

with tf.Session() as sess:

print sess.run(tf.clip\_by\_value(A, 2, 5))

## concat(values, axis, name=”concat”)

参数：

values：需要链接的矩阵的集合，通常可以是一个list。

axis：需要进行链接的维度，若矩阵是n维的，则axis的取值为0~n-1。

name：名称，是一个可选参数。

import tensorflow as tf

# Tensorflow交互式会话

tf.InteractiveSession()

# 定义两个矩阵，大小为2x3x4

a = tf.Variable(tf.truncated\_normal(shape=[2,3,4], dtype=tf.float32))

b = tf.Variable(tf.truncated\_normal(shape=[2,3,4], dtype=tf.float32))

# 按照维度0链接

c1 = tf.concat([a, b], axis=0)

# 按照维度1链接

c2 = tf.concat([a, b], axis=1)

# 按照维度2链接

c3 = tf.concat([a, b], axis=2)

# 初始化变量

tf.global\_variables\_initializer().run()

# 输出

print("01")

print(c1)

print(c1.eval())

print("02")

print(c2)

print(c2.eval())

print("03")

print(c3)

print(c3.eval())

程序运行结果如下：

01

Tensor("concat:0", shape=(4, 3, 4), dtype=float32)

[[[-0.08826777 1.92810595 -0.79408133 -0.34322619]

[-1.71443737 0.70375884 -0.78194672 -0.41254947]

[ 0.89348751 -0.08941202 0.70108914 0.64701825]]

[[ 1.50688016 0.45680258 -1.08100998 0.24127837]

[ 0.58221173 -1.41846514 -1.63450527 -0.41922286]

[ 0.48436531 -1.20013559 0.95647675 -0.03131635]]

[[-0.03254275 -1.8339541 -0.81978613 -1.25303519]

[-1.55067682 -0.37825376 -0.63578284 -0.83120823]

[ 0.09672505 -0.43550658 -0.31754431 -0.37109831]]

[[ 1.59722102 -0.32856748 -1.33017409 1.43195128]

[-0.58259052 -1.60538054 0.07504115 0.8916716 ]

[-1.23682356 -0.24931362 1.19812703 -0.81907171]]]

02

Tensor("concat\_1:0", shape=(2, 6, 4), dtype=float32)

[[[-0.08826777 1.92810595 -0.79408133 -0.34322619]

[-1.71443737 0.70375884 -0.78194672 -0.41254947]

[ 0.89348751 -0.08941202 0.70108914 0.64701825]

[-0.03254275 -1.8339541 -0.81978613 -1.25303519]

[-1.55067682 -0.37825376 -0.63578284 -0.83120823]

[ 0.09672505 -0.43550658 -0.31754431 -0.37109831]]

[[ 1.50688016 0.45680258 -1.08100998 0.24127837]

[ 0.58221173 -1.41846514 -1.63450527 -0.41922286]

[ 0.48436531 -1.20013559 0.95647675 -0.03131635]

[ 1.59722102 -0.32856748 -1.33017409 1.43195128]

[-0.58259052 -1.60538054 0.07504115 0.8916716 ]

[-1.23682356 -0.24931362 1.19812703 -0.81907171]]]

03

Tensor("concat\_2:0", shape=(2, 3, 8), dtype=float32)

[[[-0.08826777 1.92810595 -0.79408133 -0.34322619 -0.03254275 -1.8339541

-0.81978613 -1.25303519]

[-1.71443737 0.70375884 -0.78194672 -0.41254947 -1.55067682 -0.37825376

-0.63578284 -0.83120823]

[ 0.89348751 -0.08941202 0.70108914 0.64701825 0.09672505 -0.43550658

-0.31754431 -0.37109831]]

[[ 1.50688016 0.45680258 -1.08100998 0.24127837 1.59722102 -0.32856748

-1.33017409 1.43195128]

[ 0.58221173 -1.41846514 -1.63450527 -0.41922286 -0.58259052 -1.60538054

0.07504115 0.8916716 ]

[ 0.48436531 -1.20013559 0.95647675 -0.03131635 -1.23682356 -0.24931362

1.19812703 -0.81907171]]]

/×××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××/

这里要注意的是：如果是两个向量，它们是无法调用

tf.concat(1, [t1, t2])

来连接的，因为它们对应的shape只有一个维度，当然不能在第二维上连了，虽然实际中两个向量可以在行上连，但是放在程序里是会报错的

如果要连，必须要调用tf.expand\_dims来扩维：

t1=tf.constant([1,2,3])

t2=tf.constant([4,5,6])

#concated = tf.concat(1, [t1,t2])这样会报错

t1=tf.expand\_dims(tf.constant([1,2,3]),1)

t2=tf.expand\_dims(tf.constant([4,5,6]),1)

concated = tf.concat(1, [t1,t2])#

## class tf.train.AdadeltaOptimizer

实现了 Adadelta算法的优化器，可以算是下面的Adagrad算法改进版本

构造函数：

tf.train.AdadeltaOptimizer.init(learning\_rate=0.001, rho=0.95, epsilon=1e-08, use\_locking=False, name=’Adadelta’)

作用：构造一个使用Adadelta算法的优化器

参数：

learning\_rate: tensor或者浮点数，学习率

rho: tensor或者浮点数. The decay rate.

epsilon: A Tensor or a floating point value. A constant epsilon used to better conditioning the grad update.

use\_locking: If True use locks for update operations.

name: 【可选】这个操作的名字，默认是”Adadelta”

## tf.truncated\_normal与tf.random\_normal

tf.truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

1

从截断的正态分布中输出随机值。

生成的值服从具有指定平均值和标准偏差的正态分布，如果生成的值大于平均值2个标准偏差的值则丢弃重新选择。

在正态分布的曲线中，横轴区间（μ-σ，μ+σ）内的面积为68.268949%。

横轴区间（μ-2σ，μ+2σ）内的面积为95.449974%。

横轴区间（μ-3σ，μ+3σ）内的面积为99.730020%。

X落在（μ-3σ，μ+3σ）以外的概率小于千分之三，在实际问题中常认为相应的事件是不会发生的，基本上可以把区间（μ-3σ，μ+3σ）看作是随机变量X实际可能的取值区间，这称之为正态分布的“3σ”原则。

在tf.truncated\_normal中如果x的取值在区间（μ-2σ，μ+2σ）之外则重新进行选择。这样保证了生成的值都在均值附近。

参数:

shape: 一维的张量，也是输出的张量。

mean: 正态分布的均值。

stddev: 正态分布的标准差。

dtype: 输出的类型。

seed: 一个整数，当设置之后，每次生成的随机数都一样。

name: 操作的名字。

tf.random\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

从正态分布中输出随机值。

参数:

shape: 一维的张量，也是输出的张量。

mean: 正态分布的均值。

stddev: 正态分布的标准差。

dtype: 输出的类型。

seed: 一个整数，当设置之后，每次生成的随机数都一样。

name: 操作的名字。

代码

a = tf.Variable(tf.random\_normal([2,2],seed=1))

b = tf.Variable(tf.truncated\_normal([2,2],seed=2))

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

sess.run(init)

print(sess.run(a))

print(sess.run(b))

输出：

[[-0.81131822 1.48459876]

[ 0.06532937 -2.44270396]]

[[-0.85811085 -0.19662298]

[ 0.13895047 -1.22127688]]

指定seed之后，a的值不变，b的值也不变。

## tf.slice

函数原型：slice(input\_, begin, size, name=None)

参数：

input：待切片的矩阵tensor。

begin：起始位置，表示从哪一个数据开始进行切片。这个起始位置从0开始。若input是一个n维的矩阵，则begin是一个长度为n的tensor。

size：切片的大小（尺寸），表示则起始位置开始获取每一维上的若干数据。是一个长度与begin相同的tensor。若size中第n个的数据为-1，则表示在该维度上，从起始位置开始的所有数据均被返回。

name：该操作的名称，是一个可选参数，默认为None。

对于一个n维的矩阵，需满足如下关系：

0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n]

import tensorflow as tf

# Tensorflow交互式会话

tf.InteractiveSession()

# 定义5x5大小的一个矩阵变量

a = tf.Variable(tf.truncated\_normal(shape=[5, 5], dtype=tf.float32))

# 进行切片操作，起始位置为[1,1]（从0开始），大小[2,2]

b = tf.slice(a, [1, 1], [2, 2])

# 同上

c = tf.Variable(tf.truncated\_normal(shape=[2, 6, 5], dtype=tf.float32))

d = tf.slice(c, [0, 2, 3], [2, 3, 1])

# 全局变量初始化

tf.global\_variables\_initializer().run()

# 输出

print("Example 01")

print("the original matrix:\n", a.eval())

print("after being sliced:\n", b.eval())

print("Example 02")

print("the original matrix:\n", c.eval())

print("after being sliced:\n", d.eval())

程序运行结果如下：（结果或有不同）

Example 01

the original matrix:

[[ 1.37798977 0.27846026 0.07193759 0.44368556 0.65868556]

[-0.57639289 -0.64335102 -0.62483543 0.38987917 0.29301718]

[ 0.18187736 0.11397317 1.85999572 -0.26037475 0.98114467]

[ 0.69557261 0.01183218 -0.27376401 -1.15162456 1.11336803]

[-0.66582751 -0.04991583 -1.58189285 0.98189503 -1.11317801]]

after being sliced:

[[-0.64335102 -0.62483543]

[ 0.11397317 1.85999572]]

Example 02

the original matrix:

[[[-0.44467756 -1.05340731 -0.32313645 -0.69316941 0.04659459]

[ 0.01275753 -0.11907347 1.70015264 0.60470396 -0.23756829]

[ 0.07424127 1.01376414 -1.15661514 -0.46597373 -1.82189155]

[-0.66635352 -0.34318891 0.49555108 0.13062055 -0.67137426]

[ 0.04240284 0.55397838 -0.09988129 -0.93551743 0.6810317 ]

[ 1.06745911 0.49900523 1.0482769 0.39871195 1.23199737]]

[[ 1.22305858 -0.839634 0.63722724 -1.39846325 -0.04114933]

[-1.11448932 0.20783874 0.39737079 1.13769484 -0.09408376]

[-0.66636425 0.37878662 -0.32013494 -0.26526076 1.53422773]

[-0.55344075 0.23021726 0.10251451 0.08433547 1.19850338]

[ 1.73070538 -0.50309545 -0.52816319 -0.41802529 -1.52679396]

[-1.60076332 0.88759929 0.01327948 -0.7242741 -0.70737672]]]

after being sliced:

[[[-0.46597373]

[ 0.13062055]

[-0.93551743]]

[[-0.26526076]

[ 0.08433547]

[-0.41802529]]]

## 全连接

dense层定义在 tensorflow/python/layers/core.py.

3、tf.layers.dense

复制代码

dense(

inputs,

units,

activation=None,

use\_bias=True,

kernel\_initializer=None,

bias\_initializer=tf.zeros\_initializer(),

kernel\_regularizer=None,

bias\_regularizer=None,

activity\_regularizer=None,

trainable=True,

name=None,

reuse=None

)

复制代码

inputs: 输入数据，2维tensor.

units: 该层的神经单元结点数。

activation: 激活函数.

use\_bias: Boolean型，是否使用偏置项.

kernel\_initializer: 卷积核的初始化器.

bias\_initializer: 偏置项的初始化器，默认初始化为0.

kernel\_regularizer: 卷积核化的正则化，可选.

bias\_regularizer: 偏置项的正则化，可选.

activity\_regularizer: 输出的正则化函数.

trainable: Boolean型，表明该层的参数是否参与训练。如果为真则变量加入到图集合中 GraphKeys.TRAINABLE\_VARIABLES (see tf.Variable).

name: 层的名字.

reuse: Boolean型, 是否重复使用参数.

全连接层执行操作 outputs = activation(inputs.kernel + bias)

如果执行结果不想进行激活操作，则设置activation=None。

例：

#全连接层

dense1 = tf.layers.dense(inputs=pool3, units=1024, activation=tf.nn.relu)

dense2= tf.layers.dense(inputs=dense1, units=512, activation=tf.nn.relu)

logits= tf.layers.dense(inputs=dense2, units=10, activation=None)

也可以对全连接层的参数进行正则化约束：

dense1 = tf.layers.dense(inputs=pool3, units=1024, activation=tf.nn.relu，kernel\_regularizer=tf.contrib.layers.l2\_regularizer(0.003))

## 池化层

定义在 tensorflow/python/layers/pooling.py.

有最大值池化和均值池化。

1、tf.layers.max\_pooling2d

复制代码

max\_pooling2d(

inputs,

pool\_size,

strides,

padding='valid',

data\_format='channels\_last',

name=None

)

复制代码

inputs: 进行池化的数据。

pool\_size: 池化的核大小(pool\_height, pool\_width)，如[3，3]. 如果长宽相等，也可以直接设置为一个数，如pool\_size=3.

strides: 池化的滑动步长。可以设置为[1,1]这样的两个整数. 也可以直接设置为一个数，如strides=2

padding: 边缘填充，'same' 和'valid‘选其一。默认为valid

data\_format: 输入数据格式，默认为channels\_last ，即 (batch, height, width, channels),也可以设置为channels\_first 对应 (batch, channels, height, width).

name: 层的名字。

例：

pool1=tf.layers.max\_pooling2d(inputs=x, pool\_size=[2, 2], strides=2)

一般是放在卷积层之后，如：

复制代码

conv=tf.layers.conv2d(

inputs=x,

filters=32,

kernel\_size=[5, 5],

padding="same",

activation=tf.nn.relu)

pool=tf.layers.max\_pooling2d(inputs=conv, pool\_size=[2, 2], strides=2)

复制代码

2.tf.layers.average\_pooling2d

复制代码

average\_pooling2d(

inputs,

pool\_size,

strides,

padding='valid',

data\_format='channels\_last',

name=None

)

复制代码

参数和前面的最大值池化一样。

## tensorflow中的逆卷积操作

在tensorflow中逆卷积

outputs = nn.conv2d\_transpose(

inputs,

self.kernel,

output\_shape\_tensor,

strides,

padding=self.padding.upper(),

data\_format=utils.convert\_data\_format(self.data\_format, ndim=4))

这里的output\_shape\_tensor, width 和height的计算方法如下，可以根据需要的输出，设计filter\_size 和padding.

输出tensor的大小计算

def deconv\_output\_length(input\_length, filter\_size, padding, stride):

"""Determines output length of a transposed convolution given input length.

Arguments:

input\_length: integer.

filter\_size: integer.

padding: one of "same", "valid", "full".

stride: integer.

Returns:

The output length (integer).

"""

if input\_length is None:

return None

input\_length \*= stride

if padding == 'valid':

input\_length += max(filter\_size - stride, 0)

elif padding == 'full':

input\_length -= (stride + filter\_size - 2)

return input\_length

在slim.conv2d\_transpose中 不需要指定output\_shape, 我们可以根据需求设定kernel的大小

conv2d\_transpose(

inputs,

filters,

kernel\_size,

strides=(1, 1),

padding='valid',

data\_format='channels\_last',

activation=None,

use\_bias=True,

kernel\_initializer=None,

bias\_initializer=tf.zeros\_initializer(),

kernel\_regularizer=None,

bias\_regularizer=None,

activity\_regularizer=None,

kernel\_constraint=None,

bias\_constraint=None,

trainable=True,

name=None,

reuse=None

)

## tf.nn.softmax\_cross\_entropy\_with\_logits(logits, labels, name=None)

除去name参数用以指定该操作的name，与方法有关的一共两个参数：

第一个参数logits：就是神经网络最后一层的输出，如果有batch的话，它的大小就是[batchsize，num\_classes]，单样本的话，大小就是num\_classes

第二个参数labels：实际的标签，大小同上

具体的执行流程大概分为两步：

第一步是先对网络最后一层的输出做一个softmax，这一步通常是求取输出属于某一类的概率，对于单样本而言，输出就是一个num\_classes大小的向量（[Y1，Y2,Y3...]其中Y1，Y2，Y3...分别代表了是属于该类的概率）

softmax的公式是：

至于为什么是用的这个公式？这里不介绍了，涉及到比较多的理论证明

第二步是softmax的输出向量[Y1，Y2,Y3...]和样本的实际标签做一个交叉熵，公式如下：

其中指代实际的标签中第i个的值（用mnist数据举例，如果是3，那么标签是[0，0，0，1，0，0，0，0，0，0]，除了第4个值为1，其他全为0）

就是softmax的输出向量[Y1，Y2,Y3...]中，第i个元素的值

显而易见，预测越准确，结果的值越小（别忘了前面还有负号），最后求一个平均，得到我们想要的loss

注意！！！这个函数的返回值并不是一个数，而是一个向量，如果要求交叉熵，我们要再做一步tf.reduce\_sum操作,就是对向量里面所有元素求和，最后才得到，如果求loss，则要做一步tf.reduce\_mean操作，对向量求均值！

## tf.one\_hot()

功能：dense to one hot

import tensorflow as tf

indices = [[3], [5], [0], [7]]

indices = tf.concat(0, indices)

indices = tf.reshape(indice, (4, 1))

a = tf.one\_hot(indices, depth=10, on\_value=None, off\_value=None, axis=None, dtype=None, name=None)

print ("a is : ")

print a

b = tf.reshape(a, (4, 10))

print ("a is : ")

print b

'''

a is :

Tensor("one\_hot:0", shape=(4, 1, 10), dtype=float32)

a is :

Tensor("Reshape\_1:0", shape=(4, 10), dtype=float32)

'''

Args:

indices: A Tensor of indices.

depth: A scalar defining the depth of the one hot dimension.

on\_value: A scalar defining the value to fill in output when indices[j] = i. (default: 1)

off\_value: A scalar defining the value to fill in output when indices[j] != i. (default: 0)

axis: The axis to fill (default: -1, a new inner-most axis).

dtype: The data type of the output tensor.

Returns:

output: The one-hot tensor.

## GradientDescentOptimizer

这个类是实现梯度下降算法的优化器。(结合理论可以看到，这个构造函数需要的一个学习率就行了)

\_\_init\_\_(learning\_rate, use\_locking=False,name=’GradientDescent’)

作用：创建一个梯度下降优化器对象

参数：

learning\_rate: A Tensor or a floating point value. 要使用的学习率

use\_locking: 要是True的话，就对于更新操作（update operations.）使用锁

name: 名字，可选，默认是”GradientDescent”.

compute\_gradients(loss,var\_list=None,gate\_gradients=GATE\_OP,aggregation\_method=None,colocate\_gradients\_with\_ops=False,grad\_loss=None)

作用：对于在变量列表（var\_list）中的变量计算对于损失函数的梯度,这个函数返回一个（梯度，变量）对的列表，其中梯度就是相对应变量的梯度了。这是minimize()函数的第一个部分，

参数：

loss: 待减小的值

var\_list: 默认是在GraphKey.TRAINABLE\_VARIABLES.

gate\_gradients: How to gate the computation of gradients. Can be GATE\_NONE, GATE\_OP, or GATE\_GRAPH.

aggregation\_method: Specifies the method used to combine gradient terms. Valid values are defined in the class AggregationMethod.

colocate\_gradients\_with\_ops: If True, try colocating gradients with the corresponding op.

grad\_loss: Optional. A Tensor holding the gradient computed for loss.

apply\_gradients(grads\_and\_vars,global\_step=None,name=None)

作用：把梯度“应用”（Apply）到变量上面去。其实就是按照梯度下降的方式加到上面去。这是minimize（）函数的第二个步骤。 返回一个应用的操作。

参数:

grads\_and\_vars: compute\_gradients()函数返回的(gradient, variable)对的列表

global\_step: Optional Variable to increment by one after the variables have been updated.

name: 可选，名字

minimize(loss,global\_step=None,var\_list=None,gate\_gradients=GATE\_OP,aggregation\_method=None,colocate\_gradients\_with\_ops=False,name=None,grad\_loss=None)

作用：非常常用的一个函数

通过更新var\_list来减小loss，这个函数就是前面compute\_gradients() 和apply\_gradients().的结合

## concat

在哪维度连接，哪个维度就变多

老版本的维度在前，新版本在后

是连接两个矩阵的操作

tf.concat(concat\_dim, values, name='concat')

除去name参数用以指定该操作的name，与方法有关的一共两个参数：

第一个参数concat\_dim：必须是一个数，表明在哪一维上连接

第二个参数values：就是两个或者一组待连接的tensor了

     如果concat\_dim是0，那么在某一个shape的第一个维度上连，对应到实际，就是叠放到列上

[python] [view plain](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "view plain) [copy](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "copy) [print](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "print)[?](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "?)

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]]

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]]

             如果concat\_dim是1，那么在某一个shape的第二个维度上连

[python] [view plain](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "view plain) [copy](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "copy) [print](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "print)[?](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "?)

t1 = [[1, 2, 3], [4, 5, 6]]

t2 = [[7, 8, 9], [10, 11, 12]]

tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12

t1 = [[1, 2, 3], [4, 5, 6]]

t2 = [[7, 8, 9], [10, 11, 12]]

tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12

             如果有更高维，最后连接的依然是指定那个维：

             values[i].shape = [D0, D1, ... Dconcat\_dim(i), ...Dn]连接后就是：[D0, D1, ... Rconcat\_dim, ...Dn]

[python] [view plain](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "view plain) [copy](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "copy) [print](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "print)[?](http://blog.csdn.net/mao_xiao_feng/article/details/53366163" \o "?)

# tensor t3 with shape [2, 3]

# tensor t4 with shape [2, 3]

tf.shape(tf.concat(0, [t3, t4])) ==> [4, 3]

tf.shape(tf.concat(1, [t3, t4])) ==> [2, 6]

# tensor t3 with shape [2, 3]

# tensor t4 with shape [2, 3]

tf.shape(tf.concat(0, [t3, t4])) ==> [4, 3]

tf.shape(tf.concat(1, [t3, t4])) ==> [2, 6]

## Optimizer

Optimizers 类

Optimizer

The Optimizer base class provides methods to compute gradients for a loss and apply gradients to variables. A collection of subclasses implement classic optimization algorithms such as GradientDescent and Adagrad.

Optimizers这个基类提供函数用来计算梯度，并将梯度状态更新。该基类的一系类子类实现了一些经典的算法，比如说GradientDescent和Adagrad

class tf.train.Optimizer

1

Base class for optimizers.

This class defines the API to add Ops to train a model. You never use this class directly, but instead instantiate one of its subclasses such as GradientDescentOptimizer, AdagradOptimizer, or MomentumOptimizer.

用于优化的基类定义了一系列API，通过这些API可以在训练模型时添加一些操作。例如：

Usage

# Create an optimizer with the desired parameters.

opt = GradientDescentOptimizer(learning\_rate=0.1)

# 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 variables.Variable

# objects.

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

Execute opt\_op to do one step of training:

opt\_op.run()

Processing gradients before applying them.

Calling minimize() takes care of both computing the gradients and applying them to the variables. If you want to process the gradients before applying them you can instead use the optimizer in three steps：

Compute the gradients with compute\_gradients().

Process the gradients as you wish.

Apply the processed gradients with apply\_gradients().

可以通过minimize()函数来同时计算梯度并更新该梯度所对应的参数状态（其计算的就是对应的参数的梯度），若想先计算梯度，然后再将梯度对应参数状态更新，可以通过以下几步来使用：

\* 利用 compute\_gradients() 函数先计算梯度

\* 按照自己的需求来处理梯度

\* 调用apply\_gradients() 函数来更新该梯度所对应的参数的状态。

\* 这样有一个好处是，梯度可以按照自己的需求来改动后再使用。

例如：

# Create an optimizer.

opt = GradientDescentOptimizer(learning\_rate=0.1)

# Compute the gradients for a list of variables.

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

# grads\_and\_vars is a list of tuples (gradient, variable). Do whatever you

# need to the 'gradient' part, for example cap them, etc.

capped\_grads\_and\_vars = [(MyCapper(gv[0]), gv[1])) for gv in grads\_and\_vars]

# Ask the optimizer to apply the capped gradients.

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

基类函数：

tf.train.Optimizer.\_\_init\_\_(use\_locking, name)

Create a new Optimizer.

This must be called by the constructors of subclasses.

必须是子类的构造器来调用

Args:

use\_locking: Bool. If True apply use locks to prevent concurrent updates to variables.

name: A non-empty string. The name to use for accumulators created for the optimizer.

Raises:

ValueError: if name is malformed.

tf.train.Optimizer.minimize(loss, global\_step=None, var\_list=None, gate\_gradients=1, name=None)

Add operations to minimize ‘loss’ by updating ‘var\_list’.

This method simply combines calls compute\_gradients() and apply\_gradients(). If you want to process the gradient before applying them call compute\_gradients() and apply\_gradients() explicitly instead of using this function.

该函数是compute\_gradients() and apply\_gradients() 的结合 ，如果你想在得到梯度后，先修改梯度再更新参数状态，可以调用compute\_gradients()得到梯度，修改后再调用apply\_gradients(), 而minimize()是直接得到梯度不做修改就更新参数状态。

Args:

loss: A Tensor containing the value to minimize.

global\_step: Optional Variable to increment by one after the variables have been updated.

var\_list: Optional list of variables.Variable to update to minimize ‘loss’. Defaults to the list of variables collected in the graph under the key GraphKeys.TRAINABLE\_VARIABLES.

gate\_gradients: How to gate the computation of gradients. Can be GATE\_NONE, GATE\_OP, or GATE\_GRAPH.

name: Optional name for the returned operation.

Returns:

An Operation that updates the variables in ‘var\_list’. If ‘global\_step’ was not None, that operation also increments global\_step.\*\*

Raises:

ValueError: if some of the variables are not variables.Variable objects.

tf.train.Optimizer.compute\_gradients(loss, var\_list=None, gate\_gradients=1)

Compute gradients of “loss” for the variables in “var\_list”.

This is the first part of minimize(). It returns a list of (gradient, variable) pairs where “gradient” is the gradient for “variable”. Note that “gradient” can be a Tensor, a IndexedSlices, or None if there is no gradient for the given variable.

minimize()的第一步，返回一个元素为(梯度， 参数)对的列表[ (), () ],其中每个tuple对中梯度是该参数的梯度值。

Args:

loss: A Tensor containing the value to minimize.

var\_list: Optional list of variables.Variable to update to minimize “loss”. Defaults to the list of variables collected in the graph under the key GraphKey.TRAINABLE\_VARIABLES. 参数变量列表（也就是要求的参数权重W列表）。

gate\_gradients: How to gate the computation of gradients. Can be GATE\_NONE, GATE\_OP, or GATE\_GRAPH.

Returns:

A list of (gradient, variable) pairs.

Raises:

TypeError: If var\_list contains anything else than variables.Variable.

ValueError: If some arguments are invalid.

`tf.train.Optimizer.apply\_gradients(grads\_and\_vars, global\_step=None, name=None)

Apply gradients to variables.

This is the second part of minimize(). It returns an Operation that applies gradients.

minimize() 的第二部分

Args:

grads\_and\_vars: List of (gradient, variable) pairs as returned by compute\_gradients().

上一步得到的[(gradient, variable)]

global\_step: Optional Variable to increment by one after the variables have been updated.

？？？？？？

name: Optional name for the returned operation. Default to the name passed to the Optimizer constructor.

Returns:

An Operation that applies the specified gradients. If ‘global\_step’ was not None, that operation also increments global\_step。

返回的是一个操作。

Raises:

TypeError: if grads\_and\_vars is malformed.

Gating Gradients

Both minimize() and compute\_gradients() accept a gate\_gradient argument that controls the degree of parallelism during the application of the gradients.

The possible values are: GATE\_NONE, GATE\_OP, and GATE\_GRAPH.

GATE\_NONE: Compute and apply gradients in parallel. This provides the maximum parallelism in execution, at the cost of some non-reproducibility in the results. For example the two gradients of MatMul depend on the input values: With GATE\_NONE one of the gradients could be applied to one of the inputs before the other gradient is computed resulting in non-reproducible results.

并行计算并应用梯度，该参数在计算一些非重复性损失的时候提供最大的并行度。比如矩阵相乘的两个梯度依赖于输入的矩阵值，但是在GATE\_NONE 下，其中的一个梯度可以作用于输入，在另一个不重复的梯度计算出来前

GATE\_OP: For each Op, make sure all gradients are computed before they are used. This prevents race conditions for Ops that generate gradients for multiple inputs where the gradients depend on the inputs.

对于每个梯度应用操作，确保所有的梯度都先计算出来了。对于梯度依赖于多输入的情况，这种方式能有效地防止梯度计算时的资源恶性竞争

GATE\_GRAPH: Make sure all gradients for all variables are computed before any one of them is used. This provides the least parallelism but can be useful if you want to process all gradients before applying any of them.

确保在更新参数状态前，所有梯度都被计算出来了，这种方式提供了一定的并行度,。

Slots

Some optimizer subclasses, such as MomentumOptimizer and AdagradOptimizer allocate and manage additional variables associated with the variables to train. These are called Slots. Slots have names and you can ask the optimizer for the names of the slots that it uses. Once you have a slot name you can ask the optimizer for the variable it created to hold the slot value.

一些优化的子类，如MomentumOptimizer和AdagradOptimizer在参数训练时会分配和管理一些额外的参数。这些额外的参数都与训练的参数有关，是所谓的slots。可以调用优化器来获得slots的的名称。一旦你有一个slot名字，可以调用优化类创建的额外参数来保存该slot的值。

This can be useful if you want to log debug a training algorithm, report stats about the slots, etc.

该方式在训练调试算法或者获取slot的状态时非常有效

tf.train.Optimizer.get\_slot\_names()

Return a list of the names of slots created by the Optimizer.

See get\_slot().

Returns:

A list of strings.

tf.train.Optimizer.get\_slot(var, name)

Return a slot named “name” created for “var” by the Optimizer.

Some Optimizer subclasses use additional variables. For example Momentum and Adagrad use variables to accumulate updates. This method gives access to these Variables if for some reason you need them.

有些优化子类需要使用参数来累计更新，这样的话用slot就可以获取某些参数的状态

Use get\_slot\_names() to get the list of slot names created by the Optimizer.

Args:

var: A variable passed to minimize() or apply\_gradients().

name: A string.

Returns:

The Variable for the slot if it was created, None otherwise.

子类

class tf.train.GradientDescentOptimizer

tf.train.GradientDescentOptimizer.\_\_init\_\_(learning\_rate, use\_locking=False, name='GradientDescent')

class tf.train.AdagradOptimizer

Optimizer that implements the Adagrad algorithm.

tf.train.AdagradOptimizer.\_\_init\_\_(learning\_rate, initial\_accumulator\_value=0.1, use\_locking=False, name='Adagrad')

class tf.train.MomentumOptimizer

Optimizer that implements the Momentum algorithm.

tf.train.MomentumOptimizer.\_\_init\_\_(learning\_rate, momentum, use\_locking=False, name='Momentum')

class tf.train.AdamOptimizer

Optimizer that implements the Adam algorithm.

tf.train.AdamOptimizer.\_\_init\_\_(learning\_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08, use\_locking=False, name='Adam')

Construct a new Adam optimizer.

class tf.train.FtrlOptimizer

Optimizer that implements the FTRL algorithm.

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')

class tf.train.RMSPropOptimizer

Optimizer that implements the RMSProp algorithm.

tf.train.RMSPropOptimizer.\_\_init\_\_(learning\_rate, decay, momentum=0.0, epsilon=1e-10, use\_locking=False, name='RMSProp')

Gradient Computation

TensorFlow provides functions to compute the derivatives for a given TensorFlow computation graph, adding operations to the graph. The optimizer classes automatically compute derivatives on your graph, but creators of new Optimizers or expert users can call the lower-level functions below.

对于有其他需求，可以用更底层的方法来操作梯度

tf.gradients(ys, xs, grad\_ys=None, name='gradients', colocate\_gradients\_with\_ops=False, gate\_gradients=False, aggregation\_method=None)

class tf.AggregationMethod

tf.stop\_gradient(input, name=None)

Gradient Clipping

TensorFlow provides several operations that you can use to add clipping functions to your graph. You can use these functions to perform general data clipping, but they’re particularly useful for handling exploding or vanishing gradients.

下述方法时合用于处理梯度爆炸和梯度消散

tf.clip\_by\_value(t, clip\_value\_min, clip\_value\_max, name=None)

tf.clip\_by\_norm(t, clip\_norm, name=None)

tf.clip\_by\_average\_norm(t, clip\_norm, name=None)

tf.clip\_by\_global\_norm(t\_list, clip\_norm, use\_norm=None, name=None)

tf.global\_norm(t\_list, name=None)

Decaying the learning rate 学习速率衰减应用

tf.train.exponential\_decay(learning\_rate, global\_step, decay\_steps, decay\_rate, staircase=False, name=None)

Applies exponential decay to the learning rate.适用于学习率的指数衰减

Moving Averages动态均值

有些训练算法，如Gradient Descent和 Momentum在优化时，若保持的变量的动态平均会很有用。使用动态平均的评价往往能显著改善效果。

tf.train.ExponentialMovingAverage

tf.train.ExponentialMovingAverage.\_\_init\_\_(decay, num\_updates=None, name='ExponentialMovingAverage')

Creates a new ExponentialMovingAverage object.

tf.train.ExponentialMovingAverage.apply(var\_list=None)

\*tf.train.ExponentialMovingAverage.average(var)

## 两种方法都可以获得变量的shape

tf.shape(x)

其中x可以是tensor，也可不是tensor，返回是一个tensor.

shape=tf.placeholder(tf.float32, shape=[None, 227,227,3] )

1

我们经常会这样来feed数据,如果在运行的时候想知道None到底是多少,这时候,只能通过tf.shape(x)[0]这种方式来获得.

tensor.get\_shape()

只有tensor有这个方法， 返回是一个tuple

## tf.clip\_by\_value

tf.clip\_by\_value(A, min, max)：输入一个张量A，把A中的每一个元素的值都压缩在min和max之间。小于min的让它等于min，大于max的元素的值等于max。

例如：

[python] view plain copy

import tensorflow as tf;

import numpy as np;

A = np.array([[1,1,2,4], [3,4,8,5]])

with tf.Session() as sess:

print sess.run(tf.clip\_by\_value(A, 2, 5))

输出：

[[2 2 2 4]

[3 4 5 5]]

## Graph

如果我们创建了一个，并想用它取代上面的默认图，把它指定为一个新的默认图，至少是临时换一下，可以调用该Graph实例的as\_default()方法，并得到一个Python中的上下文管理器（context manager），来管理临时默认图的生命周期，即with ...下的代码区域。

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

# 此时定义的operation和tensor都自动添加到图g上

c = tf.constant(30.0)

小提示：组装图阶段，Graph类不是线程安全的，添加operations最好在单线程内完成

## session

Tensorflow基于一个高效的C++模块进行运算。与这个模块的连接叫做session。一般而言，使用TensorFlow程序的流程是先创建一个图，然后在session中加载它。

这里，使用更加方便的InteractiveSession类。通过它，你可以更加灵活地构建你的代码。它能让你在运行图的时候，插入一些构建计算图的操作。这能给使用交互式文本shell如iPython带来便利。如果你没有使用InteractiveSession的话，你需要在开始session和加载图之前，构建整个计算图。

import tensorflow as tf

sess = tf.InteractiveSession()

# 启动图后, 变量必须先经过`初始化` (init) op 初始化,# 首先必须增加一个`初始化` op 到图中.

init\_op = tf.initialize\_all\_variables()

当Session加载Graph的时候，Graph里面的计算节点都不会被触发执行。当运行sess.run(output)的时候，会沿着指定的Tensor output来进图路径往回触发相对应的节点进行计算（图中红色线表示的那部分）。当我们需要output的值时，触发Operation tf.add(add1, mul1)被执行，而该节点则需要Tensor add1和Tensor mul1的值，则往回触发Operation tf.add(a, b)和Operation tf.mul(b, c)。以此类推。

所以在计算Graph时，并不一定是Graph中的所有节点都被计算了，而是指定的计算节点或者该节点的输出结果被需要时。

# 任务完成后就需要关闭

sess.close()

当然，在我们写代码的时候，有时候会忘记写sess.close().这里我们可以使用系统的带的with来实现session的自动关闭。

with tf.Session() as sess:

result = sess.run([product])

print result

一个Session可能会拥有一些资源，例如Variable或者Queue。当我们不再需要该session的时候，需要将这些资源进行释放。有两种方式，

调用session.close()方法；

使用with tf.Session()创建上下文（Context）来执行，当上下文退出时自动释放。

import tensorflow as tf

# Build a graph.

a = tf.constant([1.0, 2.0])

b = tf.constant([3.0, 4.0])

c = a \* b

# Launch the graph in a session.

sess = tf.Session()

# Evaluate the tensor 'c'.

print sess.run(c)

sess.close()

# result: [3., 8.]

可以改变为：

import tensorflow as tf

# Build a graph.

a = tf.constant([1.0, 2.0])

b = tf.constant([3.0, 4.0])

c = a \* b

with tf.Session() as sess:

print sess.run(c)

1. 在with语句中定义的Session，在该上下文中便成为默认session；上面的例子可以修改成：

import tensorflow as tf

# Build a graph.

a = tf.constant([1.0, 2.0])

b = tf.constant([3.0, 4.0])

c = a \* b

with tf.Session():

print c.eval()

2. 在with语句中调用Session.as\_default()方法。 上面的例子可以修改成：

import tensorflow as tf

# Build a graph.

a = tf.constant([1.0, 2.0])

b = tf.constant([3.0, 4.0])

c = a \* b

sess = tf.Session()

with sess.as\_default():

print c.eval()

sess.close()

## Operation

Operation实例就是数据流图中的节点，负责tensors的计算，即输入是若干Tensor实例，输出也是若干Tensor实例。

Operation实例与实例的type之间的区别：

这里的Operation实例，也就是operation或op，与我们想的加法、减法等操作在概念上有略微差异，后者侧重于对方法的描述，前者则参与到图中，作为一个节点，叫”operator”更合适

该实例的type，也就是op type，才是指像加法、减法这样的操作方法，如“MatMul”表示矩阵乘这个操作方法

每个Operation实例的名字在图中都是唯一的，因为对应一个特定节点，但是相同的操作方法op type在图中可以有多个。它们在protocol buffers分别被定义为NodeDef和OpDef

创建一个Operation实例有两种方法：

第一种：调用一个op构造函数，如c=tf.matmul(a,b)，则创建一个表示矩阵乘操作的op节点，其中a, b, c都为tensor，a和b作输入，c作输出

第二种：调用方法Graph.create\_op()

启动一个session后，执行Operation实例也有两种方法：

第一种：把该op传入session的方法run()

第二种：直接调用op.run()，这实际上是tf.get\_default\_session().run(op)的简写

执行Operation或者求值Tensor有两种方式：

调用Session.run()方法： 该方法的定义如下所示，参数fetches便是一个或者多个Operation或者Tensor。

tf.Session.run(fetches, feed\_dict=None)

调用Operation.run()或则Tensor.eval()方法： 这两个方法都接收参数session，用于指定在哪个session中计算。但该参数是可选的，默认为None，此时表示在进程默认session中计算。

## tf.reduce\_sum

reduce\_sum应该理解为压缩求和，用于降维

# 'x' is [[1, 1, 1]

# [1, 1, 1]]

#求和

tf.reduce\_sum(x) ==> 6

#按列求和

tf.reduce\_sum(x, 0) ==> [2, 2, 2]

#按行求和

tf.reduce\_sum(x, 1) ==> [3, 3]

#按照行的维度求和

tf.reduce\_sum(x, 1, keep\_dims=True) ==> [[3], [3]]

#行列求和

tf.reduce\_sum(x, [0, 1]) ==> 6

## Tensor

Tensor对象是符号，不是具体值

一个Tensor包含一个静态类型Rank和一个shape.

启动session后，想得到tensor的具体数值，需要调用Session.run()或t.eval()来计算。t.eval()实际上是tf.get\_default\_session().run(t)。

每次使用 eval 和 run时，都会执行整个计算图，为了获取计算的结果，将它分配给[tf.Variable](https://www.tensorflow.org/how_tos/variables/" \t "http://blog.csdn.net/zcf1784266476/article/details/_blank)，然后获取。

这其中最主要的区别就在于你可以使用sess.run()在同一步获取多个tensor中的值.使用Tensor.eval()时只能在同一步当中获取一个tensor值

t = tf.constant(42.0)

sess = tf.Session()with sess.as\_default(): # or `with sess:` to close on exit

assert sess is tf.get\_default\_session()

assert t.eval() == sess.run(t)

这其中最主要的区别就在于你可以使用sess.run()在同一步获取多个tensor中的值，   
例如：

t = tf.constant(42.0)

u = tf.constant(37.0)

tu = tf.mul(t, u)

ut = tf.mul(u, t)

with sess.as\_default():

tu.eval() # runs one step

ut.eval() # runs one step

sess.run([tu, ut]) # evaluates both tensors in a single step

1.variable变量，一般是可以被更更新或更改的数值，即在流图运行过程中可以被不断动态调整的值。我们训练一个模型的时候，会用到Tensorflow中的变量(Variables)，我们需要它来保持和更新参数值，和张量一样，变量也保存在内存缓冲区当中。

我们要预先对变量初始化，Tensorflow的变量必须先初始化然后才有值！而常值张量是不需要的，变量可以先设置好初始化方式，但是真正初始化是要sess.run(tf.global\_variables\_initializer())

才真的初始化。

2.constant  常量张量

3.placeholder：占位符 动态改变值  feeddict

**Tensorflow 和numpy区别**

相同点：都提供n位数组

不同点：numpy支持ndarray，而Tensorflow里有tensor；numpy不提供创建张量函数和求导，也不提供GPU支持。

显示

Tensor

需要加eval函数

ta = tf.zeros((2,2))

print(ta)

Tensor("zeros\_1:0", shape=(2, 2), dtype=float32)

print(ta.eval())

numpy

a = np.zeros((2,2))

print(a)

## 准备训练数据

一般TensorFlow应用代码包含Graph的定义和Session的运行，代码量不大可以封装到一个文件中，如[cancer\_classifier.py](https://github.com/tobegit3hub/deep_recommend_system/blob/master/cancer_classifier.py)文件。训练前需要准备样本数据和测试数据，一般数据文件是空格或者逗号分隔的CSV文件，但TensorFlow建议使用二进制的，这样可以支持QueuRunner和Coordinator进行多线程数据读取，并且可以通过batch size和epoch参数来控制训练时单次batch的大小和对样本文件迭代训练多少轮。如果直接读取CSV文件，需要在代码中记录下一次读取数据的指针，而且在样本无法全部加载到内存时使用非常不便。

在[data](https://github.com/tobegit3hub/deep_recommend_system/tree/master/data)目录，项目已经提供了CSV与TFRecords格式转换工具[convert\_cancer\_to\_tfrecords](https://github.com/tobegit3hub/deep_recommend_system/blob/master/data/convert_cancer_to_tfrecords.py).py，参考这个脚本你就可以parse任意格式的CSV文件，转成TensorFlow支持的TFRecords格式。无论是大数据还是小数据，通过简单的脚本工具就可以直接对接TensorFlow，项目中还提供[print\_cancer\_tfrecords.py](https://github.com/tobegit3hub/deep_recommend_system/blob/master/data/print_cancer_tfrecords.py" \o "print_cancer_tfrecords.py)脚本来调用API直接读取TFRecords文件的内容。

关于Tensorflow读取数据，官网给出了三种方法：

供给数据(Feeding)： 在TensorFlow程序运行的每一步， 让Python代码来供给数据。

从文件读取数据： 在TensorFlow图的起始， 让一个输入管线从文件中读取数据。

预加载数据： 在TensorFlow图中定义常量或变量来保存所有数据(仅适用于数据量比较小的情况)。

对于数据量较小而言，可能一般选择直接将数据加载进内存，然后再分batch输入网络进行训练（tip:使用这种方法时，结合yield 使用更为简洁，大家自己尝试一下吧，我就不赘述了）。但是，如果数据量较大，这样的方法就不适用了，因为太耗内存，所以这时最好使用tensorflow提供的队列queue，也就是第二种方法 从文件读取数据。对于一些特定的读取，比如csv文件格式，官网有相关的描述，在这儿我介绍一种比较通用，高效的读取方法（官网介绍的少），即使用tensorflow内定标准格式——TFRecords

我们使用tf.train.Example来定义我们要填入的数据格式，然后使用tf.python\_io.TFRecordWriter来写入。

## 权重初始化

在创建模型之前，先来创建权重和偏置。一般来说，初始化时应加入轻微噪声，来打破对称性，防止零梯度的问题。因为我们用的是ReLU，所以用稍大于0的值来初始化偏置能够避免节点输出恒为0的问题（dead neurons）。为了不在建立模型的时候反复做初始化操作，我们定义两个函数用于初始化。

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1) #截尾正太分布，stddev代表标准差

return tf.Variable(initial)

def bias\_variable(shape):

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

## ConfigProto

tf.ConfigProto一般用在创建session的时候。用来对session进行参数配置

with tf.Session(config = tf.ConfigProto(...),...)

#tf.ConfigProto()的参数

log\_device\_placement=True : 是否打印设备分配日志

allow\_soft\_placement=True ： 如果你指定的设备不存在，允许TF自动分配设备

tf.ConfigProto(log\_device\_placement=True,allow\_soft\_placement=True)

控制GPU资源使用率

#allow growth

config = tf.ConfigProto()

config.gpu\_options.allow\_growth = True

session = tf.Session(config=config, ...)

# 使用allow\_growth option，刚一开始分配少量的GPU容量，然后按需慢慢的增加，由于不会释放

#内存，所以会导致碎片

# per\_process\_gpu\_memory\_fraction

gpu\_options=tf.GPUOptions(per\_process\_gpu\_memory\_fraction=0.7)

config=tf.ConfigProto(gpu\_options=gpu\_options)

session = tf.Session(config=config, ...)

#设置每个GPU应该拿出多少容量给进程使用，0.4代表 40%

控制使用哪块GPU

~/ CUDA\_VISIBLE\_DEVICES=0 python your.py#使用GPU0

~/ CUDA\_VISIBLE\_DEVICES=0,1 python your.py#使用GPU0,1

#注意单词不要打错

#或者在 程序开头

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0' #使用 GPU 0

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0,1' # 使用 GPU 0，1

## 条件随机场CRF

Ⅰ tf.contrib.crf.crf\_log\_likelihood

crf\_log\_likelihood(inputs,tag\_indices,sequence\_lengths,transition\_params=None)

在一个条件随机场里面计算标签序列的log-likelihood

参数:

inputs: 一个形状为[batch\_size, max\_seq\_len, num\_tags] 的tensor,一般使用BILSTM处理之后输出转换为他要求的形状作为CRF层的输入.

tag\_indices: 一个形状为[batch\_size, max\_seq\_len] 的矩阵,其实就是真实标签.

sequence\_lengths: 一个形状为 [batch\_size] 的向量,表示每个序列的长度.

transition\_params: 形状为[num\_tags, num\_tags] 的转移矩阵

返回:

log\_likelihood: 标量,log-likelihood

transition\_params: 形状为[num\_tags, num\_tags] 的转移矩阵

Ⅱ tf.contrib.crf.viterbi\_decode

viterbi\_decode(score,transition\_params)

通俗一点,作用就是返回最好的标签序列.这个函数只能够在测试时使用,在tensorflow外部解码

参数:

score: 一个形状为[seq\_len, num\_tags] matrix of unary potentials.

transition\_params: 形状为[num\_tags, num\_tags] 的转移矩阵

返回:

viterbi: 一个形状为[seq\_len] 显示了最高分的标签索引的列表.

viterbi\_score: A float containing the score for the Viterbi sequence.

Ⅲ.tf.contrib.crf.crf\_decode

crf\_decode(potentials,transition\_params,sequence\_length)

在tensorflow内解码

参数:

potentials: 一个形状为[batch\_size, max\_seq\_len, num\_tags] 的tensor,

transition\_params: 一个形状为[num\_tags, num\_tags] 的转移矩阵

sequence\_length: 一个形状为[batch\_size] 的 ,表示batch中每个序列的长度

返回:

decode\_tags:一个形状为[batch\_size, max\_seq\_len] 的tensor,类型是tf.int32.表示最好的序列标记.

best\_score: 有个形状为[batch\_size] 的tensor, 包含每个序列解码标签的分数.

a−−√b∫yx

## GPU设备

tf.ConfigProto()函数用在创建session的时候，用来对session进行参数配置：

config = tf.ConfigProto(allow\_soft\_placement=True, allow\_soft\_placement=True)

config.gpu\_options.per\_process\_gpu\_memory\_fraction = 0.4 #占用40%显存

sess = tf.Session(config=config)

1. 记录设备指派情况 : tf.ConfigProto(log\_device\_placement=True)

设置tf.ConfigProto()中参数log\_device\_placement = True ,可以获取到 operations 和 Tensor 被指派到哪个设备(几号CPU或几号GPU)上运行,会在终端打印出各项操作是在哪个设备上运行的。

2. 自动选择运行设备 ： tf.ConfigProto(allow\_soft\_placement=True)

在tf中，通过命令 "with tf.device('/cpu:0'):",允许手动设置操作运行的设备。如果手动设置的设备不存在或者不可用，就会导致tf程序等待或异常，为了防止这种情况，可以设置tf.ConfigProto()中参数allow\_soft\_placement=True，允许tf自动选择一个存在并且可用的设备来运行操作。

3. 限制GPU资源使用：

为了加快运行效率，TensorFlow在初始化时会尝试分配所有可用的GPU显存资源给自己，这在多人使用的服务器上工作就会导致GPU占用，别人无法使用GPU工作的情况。

tf提供了两种控制GPU资源使用的方法，一是让TensorFlow在运行过程中动态申请显存，需要多少就申请多少;第二种方式就是限制GPU的使用率。

一、动态申请显存

config = tf.ConfigProto()

config.gpu\_options.allow\_growth = True

session = tf.Session(config=config)

二、限制GPU使用率

config = tf.ConfigProto()

config.gpu\_options.per\_process\_gpu\_memory\_fraction = 0.4 #占用40%显存

session = tf.Session(config=config)

或者：

gpu\_options=tf.GPUOptions(per\_process\_gpu\_memory\_fraction=0.4)

config=tf.ConfigProto(gpu\_options=gpu\_options)

session = tf.Session(config=config)

设置使用哪块GPU

方法一、在python程序中设置：

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0' #使用 GPU 0

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0,1' # 使用 GPU 0，1

方法二、在执行python程序时候：

CUDA\_VISIBLE\_DEVICE=0,1 python yourcode.py

推荐使用更灵活一点的第二种方法。

## 接受命令行参数

有了TFRecords，我们就可以编写代码来训练神经网络模型了，但众所周知，深度学习有过多的Hyperparameter需要调优，我们就优化算法、模型层数和不同模型都需要不断调整，这时候使用命令行参数是非常方便的。

TensorFlow底层使用了[python-gflags](https://github.com/gflags/python-gflags)项目，然后封装成tf.app.flags接口，使用起来非常简单和直观，在实际项目中一般会提前定义命令行参数，尤其在后面将会提到的Cloud Machine Learning服务中，通过参数来简化Hyperparameter的调优。



## TensorBoard

TensorFlow还集成了一个功能强大的图形化工具，也即是TensorBoard，一般只需要在代码中加入我们关心的训练指标，TensorBoard就会自动根据这些参数绘图，通过可视化的方式来了解模型训练的情况。

tf.scalar\_summary(‘loss’, loss)  
tf.scalar\_summary(‘accuracy’, accuracy)  
tf.scalar\_summary(‘auc’, auc\_op)

## 条件随机场CRF

## sess.run() 中的feed\_dict

我们都知道feed\_dict的作用是给使用placeholder创建出来的tensor赋值。其实，他的作用更加广泛：feed 使用一个 值临时替换一个 op 的输出结果. 你可以提供 feed 数据作为 run() 调用的参数. feed 只在调用它的方法内有效, 方法结束, feed 就会消失.

import tensorflow as tf

y = tf.Variable(1)

b = tf.identity(y)with tf.Session() as sess:

tf.global\_variables\_initializer().run()

print(sess.run(b,feed\_dict={y:3})) #使用3 替换掉

#tf.Variable(1)的输出结果，所以打印出来3

#feed\_dict{y.name:3} 和上面写法等价

print(sess.run(b)) #由于feed只在调用他的方法范围内有效，所以这个打印的结果是 1

输出是3 1

## tf.slice()

tf.slice(input\_, begin, size, name=None)

先看例子

import tensorflow as tf

import numpy as np

sess = tf.InteractiveSession()

a = np.array([[1,2,3,4,5],[4,5,6,7,8],[9,10,11,12,13]])

tf.slice(a,[1,2],[-1,2]).eval()

#array([[ 6, 7],

# [11, 12]])

理解tf.slice()最好是从返回值上去理解，现在假设input的shape是[a1, a2, a3], begin的值是[b1, b2, b3],size的值是[s1, s2, s3],那么tf.slice()返回的值就是 input[b1:b1+s1, b2:b2+s2, b3:b3+s3]。

如果 si=−1si=−1 ，那么 返回值就是 input[b1:b1+s1,..., bi: ,...]

注意：input[1:2] 取不到input[2]

## tf.pad

对张量按照你的需求进行填充

定义在：tensorflow/python/ops/array\_ops.py

函数原型：

pad(

tensor,

paddings,

mode='CONSTANT',

name=None

)

参数说明：

tensor: A 'Tensor'. #tensor是要填充的张量;

paddings: A 'Tensor' of type 'int32'. #paddings也是一个张量，是需要扩张的维度，代表每一维填充多少行和列

(注意！！paddings的rank也就是秩一定要和要填充的tensor的秩一样才行)，类型是Int32;

mode: One of "CONSTANT", "REFLECT", or "SYMMETRIC" (case-insensitive) #mode可选三个值，分别是"CONSTANT","REFLECT"和"SYMMETRIC"，代表三种不同的填充方式

name: A name for the operation (optional). #pad操作的重命名，一般不需要管，默认是None

mode参数详细说明：

mode="CONSTANT" 是填充0

mode="REFLECT"是映射填充，上下（1维）填充顺序和paddings是相反的，左右（零维）顺序补齐

mode="SYMMETRIC"是对称填充，上下（1维）填充顺序是和paddings相同的，左右（零维）对称补齐

返回值说明：

A 'Tensor'. Has the same type as 'tensor'.

即返回值也是一个Tensor，类型和参数'tensor'是一样的。

举例说明：

本例使用的tensor都是rank=2的，注意paddings的rank也要等于2，否则会报错：

Example1：

t=[[2,3,4],[5,6,7]], paddings=[[1,1],[2,2]]， mode="CONSTANT"

那么sess.run(tf.pad(t,paddings,"CONSTANT"))的输出结果为：

array([[0, 0, 0, 0, 0, 0, 0],

[0, 0, 2, 3, 4, 0, 0],

[0, 0, 5, 6, 7, 0, 0],

[0, 0, 0, 0, 0, 0, 0]], dtype=int32)

mode="CONSTANT"这种填充模式是最简单的，一看就能明白

上，下，左，右分别填充了1行、1行以及2列、2列

刚好和paddings=[[1,1],[2,2]]相等，

然后全部用0填充

Example 2:

t=[[2,3,4],[5,6,7]]， paddings=[[1,2],[2,3]],mode="CONSTANT"

sess.run(tf.pad(t,paddings,"CONSTANT"))的输出结果为：

array([[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 2, 3, 4, 0, 0, 0],

[0, 0, 5, 6, 7, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)

同样是mode="CONSTANT"，这里把paddings=[[1,1],[2,2]]改成paddings=[[1,2],[2,3]]

其实道理还是一样的

上，下，左，右分别填充啦1行、2行以及2列、3列

刚好和paddings=[[1,2],[2,3]]相等

然后全部用0填充

Example 3:

t=[[2,3,4],[5,6,7]], paddings=[[1,1],[2,2]], mode='REFLECT'

sess.run(tf.pad(t,paddings,"REFLECT"))的输出结果为：

array([[7, 6, 5, 6, 7, 6, 5],

[4, 3, 2, 3, 4, 3, 2],

[7, 6, 5, 6, 7, 6, 5],

[4, 3, 2, 3, 4, 3, 2]], dtype=int32)

mode="REFLECT"这种填充模式相比mode="CONSTANT"会复杂一点

上，下，左，右分别填充了1行、1行以及2列、2列

刚好和paddings=[[1,1],[2,2]]相等

这点和mode="CONSTANT"一样，没有区别

区别在于填充的值不同：

上下左右的值进行了映射填充，

上下值填充的顺序和t是相反的，（我们先对同一列的值进行填充：例如看第三列，原来tensor的第二行第三列的值为2，第三行第三列的值为5，因此上下填充时，第一行第三列填充5，而第四行第三列就填充2了，即反过来了，大家这样理解就可以了）

左右值填充的顺序和t是也是相反的（也可以说是顺序补齐，分别从中间向左右两边扩张，是顺序的）

# 't' is [[2,3,4]].

# 'paddings' is [[2,0]] 左边填充2列，右边填充0列

pad(t, paddings, "REFLECT") ==>

[[4, 3, 2, 3, 4]]

可以看出是反向填充，

右边也一样

# 't' is [[2,3,4]].

# 'paddings' is [[0,2]] 左边填充0列，右边填充2列

pad(t, paddings, "REFLECT") ==>

[[2, 3, 4, 3, 2]]

Example 4:

t=[[2,3,4],[5,6,7]], paddings=[[1,1],[2,2]], mode='SYMMETRIC'

sess.run(tf.pad(t,paddings,"SYMMETRIC"))的输出结果为：

array([[3, 2, 2, 3, 4, 4, 3],

[3, 2, 2, 3, 4, 4, 3],

[6, 5, 5, 6, 7, 7, 6],

[6, 5, 5, 6, 7, 7, 6]], dtype=int32)

mode="SYMMETRIC"这种填充模式相比mode="REFLECT"差不多

区别在于：

上下左右的值进行了对称填充，

上下值是按照t相同顺序填充的（我们先对同一列的值进行填充：例如看第三列，原来tensor的第二行第三列的值为2，第三行第三列的值为5，因此上下填充时，第一行第三列填充2，而第四行第三列就填充5了，即和它最近的那个数值是一样的，大家这样理解就可以了）

左右值只是进行对称补齐（分别从中间向左右两边扩张，是关于中心对称的）

# 't' is [[2,3,4]].

# 'paddings' is [[2,0]] 左边填充2列，右边填充0列

pad(t, paddings, "REFLECT") ==>

[[3, 2, 2, 3, 4]]

可以看出是对称填充，

# 't' is [[2,3,4]].

# 'paddings' is [[0,2]] 左边填充0列，右边填充2列

pad(t, paddings, "REFLECT") ==>

[[2, 3, 4, 4, 3]]

## With

A session may own resources, such as [variables](https://www.tensorflow.org/versions/r0.11/api_docs/python/state_ops.html" \l "Variable" \t "http://blog.csdn.net/u011534057/article/details/_blank), [queues](https://www.tensorflow.org/versions/r0.11/api_docs/python/io_ops.html" \l "QueueBase" \t "http://blog.csdn.net/u011534057/article/details/_blank), and [readers](https://www.tensorflow.org/versions/r0.11/api_docs/python/io_ops.html" \l "ReaderBase" \t "http://blog.csdn.net/u011534057/article/details/_blank). It is important to release these resources when they are no longer required. To do this, either invoke the [close()](https://www.tensorflow.org/versions/r0.11/api_docs/python/client.html" \l "Session.close" \t "http://blog.csdn.net/u011534057/article/details/_blank) method on the session, or use the session as a context manager. The following two examples are equivalent:

# Using the `close()` method.

sess = tf.Session()

sess.run(...)

sess.close()

# Using the context manager.

with tf.Session() as sess:

sess.run(...)

## Feeds

Feeds机制使用一个Tensor值临时地替换op的输出结果，你需要在run方法中提供feed数据，这个feed数据只会在run方法运行时调用。

input1 = tf.placeholder(tf.float32)

input2 = tf.placeholder(tf.float32)

output = tf.mul(input1, input2)

with tf.Session() as sess:

print(sess.run([output], feed\_dict={input1:[7.], input2:[2.]}))

# output:# [array([ 14.], dtype=float32)]

## GPU

而且如果检测到GPU，会尽可能的使用GPU来实现对程序的计算。而当计算机上有多个GPU的时候，我们可以通过tf.device()来指定哪个GPU来执行。具体示例如下：

with tf.Session() as sess:with tf.device("/gpu:1"):

a = tf.constant([[3., 3.]])

b = tf.constant([[2.],[2.])

product = tf.matmul(a,b)

...

设备用字符串进行标识. 目前支持的设备包括:

"/cpu:0": 机器的 CPU.

"/gpu:0": 机器的第一个 GPU, 如果有的话.

"/gpu:1": 机器的第二个 GPU, 以此类推.

## conv2d和conv1d

卷积层：

　　自然图像有其固有特性，也就是说，图像的一部分的统计特性与其他部分是一样的。这也意味着我们在这一部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。

　　对于图像，当从一个大尺寸图像中随机选取一小块，比如说8x8作为样本，并且从这个小块样本中学习到了一些特征，这时我们可以把从这个8x8样本中学习到的特征作为探测器，应用到这个图像的任意地方中去。特别是，我们可以用从8x8样本中所学习到的特征跟原本的大尺寸图像作卷积，从而对这个大尺寸图像上的任一位置获得一个不同特征的激活值。

#### conv2d

tf.nn.conv2d(input, filter, strides, padding, use\_cudnn\_on\_gpu=None, name=None)

除去name参数用以指定该操作的name，与方法有关的一共五个参数：

第一个参数input：指需要做卷积的输入图像，它要求是一个Tensor，具有[batch, in\_height, in\_width, in\_channels]这样的shape，具体含义是[训练时一个batch的图片数量, 图片高度, 图片宽度, 图像通道数]，注意这是一个4维的Tensor，要求类型为float32和float64其中之一

第二个参数filter：相当于CNN中的卷积核，它要求是一个Tensor，具有[filter\_height, filter\_width, in\_channels, out\_channels]这样的shape，具体含义是[卷积核的高度，卷积核的宽度，图像通道数，卷积核个数]，要求类型与参数input相同，有一个地方需要注意，第三维in\_channels，就是参数input的第四维

第三个参数strides：卷积时在图像每一维的步长，这是一个一维的向量，长度4

第四个参数padding：string类型的量，只能是"SAME","VALID"其中之一，这个值决定了不同的卷积方式（后面会介绍）

第五个参数：use\_cudnn\_on\_gpu:bool类型，是否使用cudnn加速，默认为true

结果返回一个Tensor，这个输出，就是我们常说的feature map，shape仍然是[batch, height, width, channels]这种形式。

1.考虑一种最简单的情况，现在有一张3×3单通道的图像（对应的shape：[1，3，3，1]），用一个1×1的卷积核（对应的shape：[1，1，1，1]）去做卷积，最后会得到一张3×3的feature map

2.增加图片的通道数，使用一张3×3五通道的图像（对应的shape：[1，3，3，5]），用一个1×1的卷积核（对应的shape：[1，1，1，1]）去做卷积，仍然是一张3×3的feature map，这就相当于每一个像素点，卷积核都与该像素点的每一个通道做卷积。

3.把卷积核扩大，现在用3×3的卷积核做卷积，最后的输出是一个值，相当于情况2的feature map所有像素点的值求和

4.使用更大的图片将情况2的图片扩大到5×5，仍然是3×3的卷积核，令步长为1，输出3×3的feature map

5.上面我们一直令参数padding的值为‘VALID’，当其为‘SAME’时，表示卷积核可以停留在图像边缘，如下，输出5×5的feature map

6.如果卷积核有多个

input = tf.Variable(tf.random\_normal([1,5,5,5]))

filter = tf.Variable(tf.random\_normal([3,3,5,7]))

op = tf.nn.conv2d(input, filter, strides=[1, 1, 1, 1], padding='SAME')

此时输出7张5×5的feature map

7.步长不为1的情况，文档里说了对于图片，因为只有两维，通常strides取[1，stride，stride，1]

input = tf.Variable(tf.random\_normal([1,5,5,5]))

filter = tf.Variable(tf.random\_normal([3,3,5,7]))

op = tf.nn.conv2d(input, filter, strides=[1, 2, 2, 1], padding='SAME')

此时，输出7张3×3的feature map

8.如果batch值不为1，同时输入10张图

input = tf.Variable(tf.random\_normal([10,5,5,5]))

filter = tf.Variable(tf.random\_normal([3,3,5,7]))

op = tf.nn.conv2d(input, filter, strides=[1, 2, 2, 1], padding='SAME')

每张图，都有7张3×3的feature map，输出的shape就是[10，3，3，7]

#### conv1d

import tensorflow as tf  
import numpy as np  
  
#Input : X  
x\_image = tf.placeholder(tf.float32, shape=[5])  
x = tf.reshape(x\_image, [1, 5, 1])  
  
#Filter: W  
w\_cpu = np.array([0, -1, 1], dtype=np.float32)  
w = tf.Variable(w\_cpu)  
w = tf.reshape(w, [3, 1, 1])  
  
#stride & padding  
strides = 1  
padding = 'VALID'  
  
#convolution  
y = tf.nn.conv1d(x, w, strides, padding)  
  
x\_data = np.array([2,2,1,0,0],dtype=np.float32)  
  
with tf.Session() as sess:  
    init = tf.global\_variables\_initializer()  
    sess.run(init)  
  
    x = sess.run(x, feed\_dict={x\_image:x\_data})  
    w = sess.run(w, feed\_dict={x\_image:x\_data})  
    y = sess.run(y, feed\_dict={x\_image:x\_data})  
  
    print ("The shape of x:{}, and the x.reshape(5,5) is {}".format(x.shape,  x.reshape(5, 1)))  
    print ("The shape of w:{}, and the w.reshape(3,3) is {}".format(w.shape,  w.reshape(3, 1)))  
    print ("The shape of y:{}, and the y.reshape(3,3) is {}".format(y.shape,  y))

## layers.max\_pooling2d

池化层：

Pooling 的本质，其实是采样。Pooling 对于输入的 Feature Map，选择某种方式对其进行压缩。

在通过卷积获得了特征（features）之后，下一步我们希望利用这些特征去做分类。理论上讲，人们可以把所有解析出来的特征关联到一个分类器，例如softmax分类器，但计算量非常大。例如：对于一个96X96像素的图像，假设我们已经通过8X8个输入学习得到了400个特征。而每一个卷积都会得到一个(96 − 8 + 1) \* (96 − 8 + 1) = 7921的结果集，由于已经得到了400个特征，所以对于每个样例（example）结果集的大小就将达到892 \* 400 = 3,168,400 个特征。这样学习一个拥有超过3百万特征的输入的分类器是相当不明智的，并且极易出现过度拟合（over-fitting）.

　　所以就有了pooling这个方法，翻译作“池化”？感觉pooling这个英语单词还是挺形象的，翻译“作池”化就没那么形象了。其实也就是把特征图像区域的一部分求个均值或者最大值，用来代表这部分区域。如果是求均值就是mean pooling，求最大值就是max pooling。

max\_pooling2d(

inputs,

pool\_size,

strides,

padding='valid',

data\_format='channels\_last',

name=None

)

inputs: 进行池化的数据。

pool\_size: 池化的核大小(pool\_height, pool\_width)，如[3，3]. 如果长宽相等，也可以直接设置为一个数，如pool\_size=3.

strides: 池化的滑动步长。可以设置为[1,1]这样的两个整数. 也可以直接设置为一个数，如strides=2

padding: 边缘填充，'same' 和'valid‘选其一。默认为valid

data\_format: 输入数据格式，默认为channels\_last ，即 (batch, height, width, channels),也可以设置为channels\_first 对应 (batch, channels, height, width).

name: 层的名字。

## nn.max\_pool

tf.nn.max\_pool(value, ksize, strides, padding, name=None)

value：池化的输入，一般池化层接在卷积层的后面，所以输出通常为feature map。feature map依旧是[batch, in\_height, in\_width, in\_channels]这样的参数。

ksize：池化窗口的大小，参数为四维向量，通常取[1, height, width, 1]，因为我们不想在batch和channels上做池化，所以这两个维度设为了1。ps：估计面tf.nn.conv2d中stries的四个取值也有              相同的意思。

stries：步长，同样是一个四维向量。

padding：填充方式同样只有两种不重复了。

## variable\_scope

tf.dd(name\_or\_scope,default\_name=None,values=None,initializer=None,regularizer=None,caching\_device=None,partitioner=None,custom\_getter=None,reuse=None,dtype=None)

返回一个用于定义创建variable（层）的op的上下文管理器。

该上下文管理器验证（可选）值来自同一图形，确保图形是默认图形，并推送名称范围和variable范围。

而这就不得不谈到tf. get\_variable()了。因为如果使用Variable 的话每次都会新建变量，但是大多数时候我们是希望一些变量重用的，所以就用到了get\_variable()。它会去搜索变量名，然后没有就新建，有就直接用。  
既然用到变量名了，就涉及到了名字域的概念。通过不同的域来区别变量名，毕竟让我们给所有变量都直接取不同名字还是有点辛苦的。所以为什么会有scope 的概念。  
name\_scope 作用于操作，variable\_scope 可以通过设置reuse 标志以及初始化方式来影响域下的变量。  
当然对我们而言还有个更直观的感受就是：在tensorboard 里可视化的时候用名字域进行封装后会更清晰

## name\_scope

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

在 tf.name\_scope下时，tf.get\_variable()创建的变量名不受 name\_scope 的影响，而且在未指定共享变量时，如果重名会报错，tf.Variable()会自动检测有没有变量重名，如果有则会自行处理。

import tensorflow as tf

with tf.name\_scope('name\_scope\_x'):

var1 = tf.get\_variable(name='var1', shape=[1], dtype=tf.float32)

var3 = tf.Variable(name='var2', initial\_value=[2], dtype=tf.float32)

var4 = tf.Variable(name='var2', initial\_value=[2], dtype=tf.float32)

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

print(var1.name, sess.run(var1))

print(var3.name, sess.run(var3))

print(var4.name, sess.run(var4))

# 输出结果：

# var1:0 [-0.30036557] 可以看到前面不含有指定的'name\_scope\_x'

# name\_scope\_x/var2:0 [ 2.]

# name\_scope\_x/var2\_1:0 [ 2.] 可以看到变量名自行变成了'var2\_1'，避免了和'var2'冲突

如果使用tf.get\_variable()创建变量，且没有设置共享变量，重名时会报错

import tensorflow as tf

with tf.name\_scope('name\_scope\_1'):

var1 = tf.get\_variable(name='var1', shape=[1], dtype=tf.float32)

var2 = tf.get\_variable(name='var1', shape=[1], dtype=tf.float32)

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

print(var1.name, sess.run(var1))

print(var2.name, sess.run(var2))

# ValueError: Variable var1 already exists, disallowed. Did you mean

# to set reuse=True in VarScope? Originally defined at:

# var1 = tf.get\_variable(name='var1', shape=[1], dtype=tf.float32)

所以要共享变量，需要使用tf.variable\_scope()

import tensorflow as tf

with tf.variable\_scope('variable\_scope\_y') as scope:

var1 = tf.get\_variable(name='var1', shape=[1], dtype=tf.float32)

scope.reuse\_variables() # 设置共享变量

var1\_reuse = tf.get\_variable(name='var1')

var2 = tf.Variable(initial\_value=[2.], name='var2', dtype=tf.float32)

var2\_reuse = tf.Variable(initial\_value=[2.], name='var2', dtype=tf.float32)

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

print(var1.name, sess.run(var1))

print(var1\_reuse.name, sess.run(var1\_reuse))

print(var2.name, sess.run(var2))

print(var2\_reuse.name, sess.run(var2\_reuse))

# 输出结果：

# variable\_scope\_y/var1:0 [-1.59682846]

# variable\_scope\_y/var1:0 [-1.59682846] 可以看到变量var1\_reuse重复使用了var1

# variable\_scope\_y/var2:0 [ 2.]

# variable\_scope\_y/var2\_1:0 [ 2.]

也可以这样

with tf.variable\_scope('foo') as foo\_scope:

v = tf.get\_variable('v', [1])

with tf.variable\_scope('foo', reuse=True):

v1 = tf.get\_variable('v')

assert v1 == v

或者这样：

with tf.variable\_scope('foo') as foo\_scope:

v = tf.get\_variable('v', [1])

with tf.variable\_scope(foo\_scope, reuse=True):

v1 = tf.get\_variable('v')

assert v1 == v

## 训练和评估模型

用更加复杂的ADAM优化器来做梯度最速下降，在feed\_dict中加入额外的参数keep\_prob来控制dropout比例。然后每100次迭代输出一次日志。

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y\_conv))

train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

correct\_prediction = tf.equal(tf.argmax(y\_conv,1), tf.argmax(y\_,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

sess.run(tf.initialize\_all\_variables())

for i in range(20000):

batch = mnist.train.next\_batch(50)

if i%100 == 0:

train\_accuracy = accuracy.eval(feed\_dict={

x:batch[0], y\_: batch[1], keep\_prob: 1.0})

print "step %d, training accuracy %g"%(i, train\_accuracy)

train\_step.run(feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})

print "test accuracy %g"%accuracy.eval(feed\_dict={

x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.0})

## nn.softmax\_cross\_entropy\_with\_logits

tf.nn.softmax\_cross\_entropy\_with\_logits(\_sentinel=None, labels=None, logits=None, dim=-1, name=None)

注意：如果labels的每一行是one-hot表示，也就是只有一个地方为1，其他地方为0，可以使用tf.sparse\_softmax\_cross\_entropy\_with\_logits()

警告：

1. 这个操作的输入logits是未经缩放的，该操作内部会对logits使用softmax操作

2. 参数labels,logits必须有相同的形状 [batch\_size, num\_classes] 和相同的类型(float16, float32, float64)中的一种

参数：\_sentinel: 一般不使用

labels: labels的每一行labels[i]必须为一个概率分布

logits: 未缩放的对数概率

dims: 类的维度，默认-1，也就是最后一维

name: 该操作的名称

返回值：长度为batch\_size的一维Tensor

在计算loss的时候，最常见的一句话就是tf.nn.softmax\_cross\_entropy\_with\_logits，那么它到底是怎么做的呢？

首先明确一点，loss是代价值，也就是我们要最小化的值

tf.nn.softmax\_cross\_entropy\_with\_logits(logits, labels, name=None)

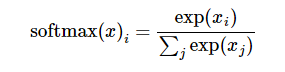
除去name参数用以指定该操作的name，与方法有关的一共两个参数：

第一个参数logits：就是神经网络最后一层的输出，如果有batch的话，它的大小就是[batchsize，num\_classes]，单样本的话，大小就是num\_classes

第二个参数labels：实际的标签，大小同上

具体的执行流程大概分为两步：

第一步是先对网络最后一层的输出做一个softmax，这一步通常是求取输出属于某一类的概率，对于单样本而言，输出就是一个num\_classes大小的向量（[Y1，Y2,Y3...]其中Y1，Y2，Y3...分别代表了是属于该类的概率）

softmax的公式是：

至于为什么是用的这个公式？这里不介绍了，涉及到比较多的理论证明

第二步是softmax的输出向量[Y1，Y2,Y3...]和样本的实际标签做一个交叉熵，公式如下：

IMG_257

其中IMG_258指代实际的标签中第i个的值（用mnist数据举例，如果是3，那么标签是[0，0，0，1，0，0，0，0，0，0]，除了第4个值为1，其他全为0）

IMG_259就是softmax的输出向量[Y1，Y2,Y3...]中，第i个元素的值

显而易见，预测IMG_260越准确，结果的值越小（别忘了前面还有负号），最后求一个平均，得到我们想要的loss

注意！！！这个函数的返回值并不是一个数，而是一个向量，如果要求交叉熵，我们要再做一步tf.reduce\_sum操作,就是对向量里面所有元素求和，最后才得到IMG_261，如果求loss，则要做一步tf.reduce\_mean操作，对向量求均值！

理论讲完了，上代码

import tensorflow as tf

#our NN's output

logits=tf.constant([[1.0,2.0,3.0],[1.0,2.0,3.0],[1.0,2.0,3.0]])

#step1:do softmax

y=tf.nn.softmax(logits)

#true label

y\_=tf.constant([[0.0,0.0,1.0],[0.0,0.0,1.0],[0.0,0.0,1.0]])

#step2:do cross\_entropy

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y))

#do cross\_entropy just one step

cross\_entropy2=tf.reduce\_sum(tf.nn.softmax\_cross\_entropy\_with\_logits(logits, y\_))#dont forget tf.reduce\_sum()!!

with tf.Session() as sess:

    softmax=sess.run(y)

    c\_e = sess.run(cross\_entropy)

    c\_e2 = sess.run(cross\_entropy2)

    print("step1:softmax result=")

    print(softmax)

    print("step2:cross\_entropy result=")

    print(c\_e)

    print("Function(softmax\_cross\_entropy\_with\_logits) result=")

    print(c\_e2)

import tensorflow as tf

#our NN's output

logits=tf.constant([[1.0,2.0,3.0],[1.0,2.0,3.0],[1.0,2.0,3.0]])

#step1:do softmax

y=tf.nn.softmax(logits)

#true label

y\_=tf.constant([[0.0,0.0,1.0],[0.0,0.0,1.0],[0.0,0.0,1.0]])

#step2:do cross\_entropy

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y))

#do cross\_entropy just one step

cross\_entropy2=tf.reduce\_sum(tf.nn.softmax\_cross\_entropy\_with\_logits(logits, y\_))#dont forget tf.reduce\_sum()!!

with tf.Session() as sess:

softmax=sess.run(y)

c\_e = sess.run(cross\_entropy)

c\_e2 = sess.run(cross\_entropy2)

print("step1:softmax result=")

print(softmax)

print("step2:cross\_entropy result=")

print(c\_e)

print("Function(softmax\_cross\_entropy\_with\_logits) result=")

print(c\_e2)

输出结果是：

[python] [view plain](http://blog.csdn.net/mao_xiao_feng/article/details/53382790" \o "view plain) [copy](http://blog.csdn.net/mao_xiao_feng/article/details/53382790" \o "copy) [print](http://blog.csdn.net/mao_xiao_feng/article/details/53382790" \o "print)[?](http://blog.csdn.net/mao_xiao_feng/article/details/53382790" \o "?)

step1:softmax result=

[[ 0.09003057  0.24472848  0.66524094]

 [ 0.09003057  0.24472848  0.66524094]

 [ 0.09003057  0.24472848  0.66524094]]

step2:cross\_entropy result=

1.22282

Function(softmax\_cross\_entropy\_with\_logits) result=

1.2228

step1:softmax result=

[[ 0.09003057 0.24472848 0.66524094]

[ 0.09003057 0.24472848 0.66524094]

[ 0.09003057 0.24472848 0.66524094]]

step2:cross\_entropy result=

1.22282

Function(softmax\_cross\_entropy\_with\_logits) result=

1.2228

最后大家可以试试e^1/(e^1+e^2+e^3)是不是0.09003057，发现确实一样！！这也证明了我们的输出是符合公式逻辑的

## Supervisor

你可以直接run一个training的op若干次来训练你的tensorflow模型，同时当你训练结束的时候你可以保存训练参数的checkpoint。对于那些数小时便可训练好的小模型来说这样干是很不错的。   
然而对于需要很多天训练的大模型来说，就需要更鲁棒的训练过程了。这要求训练时要:

干净的处理shutdown以及crash

在shutdown或者crash之后可以恢复

可以通过tensorboard来监控

为了在shutdown之后可以恢复训练，训练过程中必须规律性地run summary 的op，同时将返回的值加到事件文件（events file）中。 tensorboard监控事件文件并且显式图，来报告随着时间进行的训练过程。

tf.train.Supervisor 提供了一系列服务来帮助实现一个鲁棒的训练过程。

最简单的使用supervisor的方案就是去：

创建一个Supervisor对象，将要保存checkpoints以及summaries的目录路径传递给该对象。

利用tf.train.Supervisor.managed\_session向supervisor请求一个session。

利用该session来执行训练的op，在每一步都核查supervisor是否要求训练结束。

...create graph...

my\_train\_op = ...

sv = tf.train.Supervisor(logdir="/my/training/directory")

with sv.managed\_session() as sess:

for step in range(100000):

if sv.should\_stop():

break

sess.run(my\_train\_op)

启动服务   
在这个最简单的情景中，managed\_session()调用开启了一些列服务，这些服务在它们各自的线程中运作，同时利用该托管的session在你的图中run ops。

如果你的图中有一个name为global\_step的整型变量，服务会使用它的值来衡量执行的训练步数。

checkpointing 服务：在logdir里保存图中变量的一份拷贝。如果图中添加了global\_step变量，则checkpoint文件名使用该变量的值。默认每十分钟保存一次。

summary服务：run所有的summary ops，同时将它们的输出加到logdir中的事件文件中。默认为两分钟执行一次。

步数（step）计数：计算已经有多少步被执行了，从变量global\_step的变化中来得到。向事件文件中添加一个summary用来报告每秒global steps的数目。summary标签为”global\_step/sec”。同样野默认为两分钟执行一次。

Queue Runners,如果有图中添加有tf.train.QueueRunner，superivsor会在自己的线程中开始它们。

所有的时间间隔都可以在创建supervisor对象的时候改变。

## one-hot

考虑多类情况。非onehot，标签是类似0 1 2 3...n这样。  
而onehot标签则是顾名思义，一个长度为n的数组，只有一个元素是1.0，其他元素是0.0。  
例如在n为4的情况下，标签2对应的onehot标签就是 0.0 0.0 1.0 0.0  
使用onehot的直接原因是现在多分类cnn网络的输出通常是softmax层，而它的输出是一个概率分布，从而要求输入的标签也以概率分布的形式出现，进而算交叉熵之类。

## MultiRNNCell

tf.nn.rnn\_cell.MultiRNNCell([list RNNcell], state\_is\_tuple=True).这个函数里面主要这两个参数，第一个参数就是输入的RNN实例形成的列表，第二个参数就是让状态是一个元祖，官方推荐就是用True。

例如：

import tensorflow as tf;

import numpy as np;

X = tf.random\_normal(shape=[3,5,6], dtype=tf.float32)

X = tf.reshape(X, [-1, 5, 6])

cell = tf.nn.rnn\_cell.BasicLSTMCell(10)#也可以换成别的，比如GRUCell，BasicRNNCell等等

lstm\_multi = tf.nn.rnn\_cell.MultiRNNCell([cell]\*2,  state\_is\_tuple=True)

state = lstm\_multi.zero\_state(3, tf.float32)

output, state = tf.nn.dynamic\_rnn(lstm\_multi, X, initial\_state=state, time\_major=False)

with tf.Session() as sess:

    sess.run(tf.initialize\_all\_variables())

    print output.get\_shape()

print sess.run(state)

## BasicLSTMCell

tf.nn.rnn\_cell.BasicLSTMCell(n\_hidden, forget\_bias=1.0, state\_is\_tuple=True): n\_hidden表示神经元的个数，forget\_bias就是LSTM们的忘记系数，如果等于1，就是不会忘记任何信息。如果等于0，就都忘记。state\_is\_tuple默认就是True，官方建议用True，就是表示返回的状态用一个元祖表示。这个里.

## tensorflow 可视化

tensorflow的可视化是使用summary和tensorboard合作完成的.

基本用法

首先明确一点,summary也是op.

输出网络结构

with tf.Session() as sess:

writer = tf.summary.FileWriter(your\_dir, sess.graph)

命令行运行tensorboard --logdir your\_dir,然后浏览器输入127.0.1.1:6006注：tf1.1.0 版本的tensorboard端口换了(0.0.0.0:6006)   
这样你就可以在tensorboard中看到你的网络结构图了

## Coordinator & QueueRunner

Coordinator还比较好理解，可以理解为信号量之类的东西。QueueRunner比较难理解，通篇看介绍文档，都没有找到QueueRunner这个代码，后来终于发现一段文字：

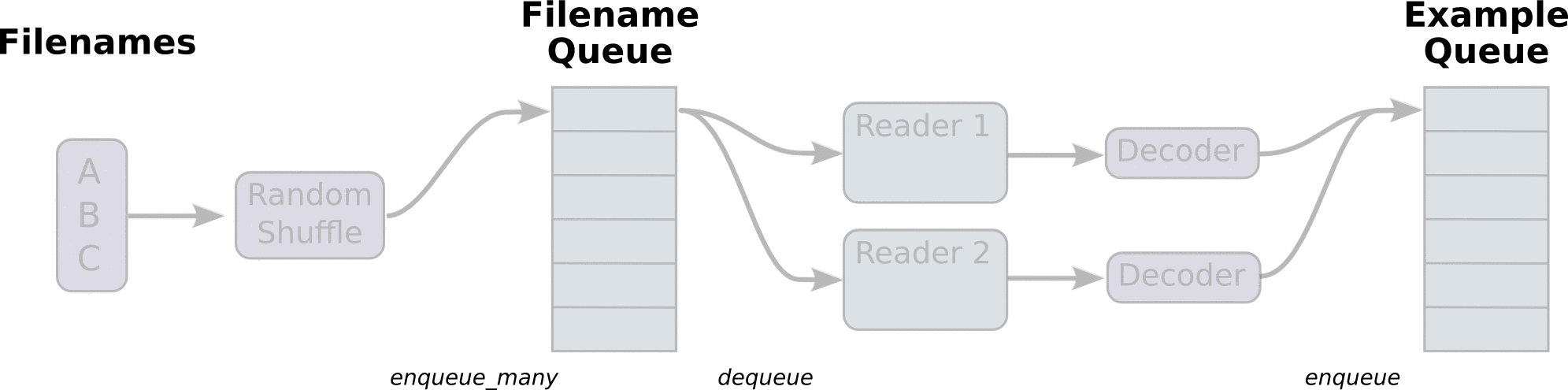
创建线程并使用QueueRunner对象来预取

简单来说：使用上面列出的许多tf.train函数添加[QueueRunner](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank)到你的数据流图中。在你运行任何训练步骤之前，需要调用[tf.train.start\_queue\_runners](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank)函数，否则数据流图将一直挂起。[tf.train.start\_queue\_runners](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank) 这个函数将会启动输入管道的线程，填充样本到队列中，以便出队操作可以从队列中拿到样本。

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

也就是说，QueueRunner是一个不存在于代码中的东西，而是后台运作的一个概念。由tf.train函数添加。

首先，我们先创建数据流图，这个数据流图由一些流水线的阶段组成，阶段间用队列连接在一起。第一阶段将生成文件名，我们读取这些文件名并且把他们排到文件名队列中。第二阶段从文件中读取数据（使用Reader），产生样本，而且把样本放在一个样本队列中。根据你的设置，实际上也可以拷贝第二阶段的样本，使得他们相互独立，这样就可以从多个文件中并行读取。在第二阶段的最后是一个排队操作，就是入队到队列中去，在下一阶段出队。因为我们是要开始运行这些入队操作的线程，所以我们的训练循环会使得样本队列中的样本不断地出队。



在tf.train中要创建这些队列和执行入队操作，就要添加[tf.train.QueueRunner](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank)到一个使用[tf.train.add\_queue\_runner](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank)函数的数据流图中。每个QueueRunner负责一个阶段，处理那些需要在线程中运行的入队操作的列表。一旦数据流图构造成功，[tf.train.start\_queue\_runners](http://wiki.jikexueyuan.com/project/api_docs/python/train.html" \t "https://blog.csdn.net/sunquan_ok/article/details/_blank)函数就会要求数据流图中每个QueueRunner去开始它的线程运行入队操作。

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

代码中根本没有QueueRunner的啊

## zero\_state

cell = tf.nn.rnn\_cell.BasicRNNCell(num\_units=128) # state\_size = 128

print(cell.state\_size) # 128

h0 = cell.zero\_state(32, np.float32)

# 32 是 batch\_size

 # 通过zero\_state得到一个全0的初始状态，形状为(batch\_size, state\_size)

## summary

tf.scalar tf.summary

经过摸索,简单使用summary,尤其注意下面代码加粗变量,加粗的是二进制变量,需要写入文件的;

以及运行的软件要有/tmp/XIAO文件夹的权限

import tensorflow as tf

sess = tf.Session()

with tf.name\_scope("XiaoGongWei"):

    a = tf.placeholder(dtype=tf.float32)

    b = tf.Variable([1.0],dtype=tf.float32)

    W = tf.Variable([1,2],dtype=tf.float32)

    addAB = W \* a + b

    tf.summary.scalar("wight\_max",tf.reduce\_mean(W))

    tf.summary.scalar("b\_value",tf.reduce\_mean(b))

    tf.summary.scalar("a\_value",tf.reduce\_mean(a))

    <strong>merged</strong> = tf.summary.merge\_all()

    train\_summary = tf.summary.FileWriter('/tmp/XIAO/',sess.graph)

    sess.run(tf.global\_variables\_initializer())

    for i in range(100):

        <strong>tfmensumm</strong>,myadd = sess.run([<strong>merged</strong>,addAB],feed\_dict={a:i})

        print(myadd)

        train\_summary.add\_summary(tfmensumm,i)

    train\_summary.close()

# /usr/bin/python3

import tensorflow as tf

sess = tf.Session()

with tf.name\_scope("XiaoGongWei"):

a = tf.placeholder(dtype=tf.float32)

b = tf.Variable([1.0],dtype=tf.float32)

W = tf.Variable([1,2],dtype=tf.float32)

addAB = W \* a + b

tf.summary.scalar("wight\_max",tf.reduce\_mean(W))

tf.summary.scalar("b\_value",tf.reduce\_mean(b))

tf.summary.scalar("a\_value",tf.reduce\_mean(a))

merged = tf.summary.merge\_all()

train\_summary = tf.summary.FileWriter('/tmp/XIAO/',sess.graph)

sess.run(tf.global\_variables\_initializer())

for i in range(100):

tfmensumm,myadd = sess.run([merged,addAB],feed\_dict={a:i})

print(myadd)

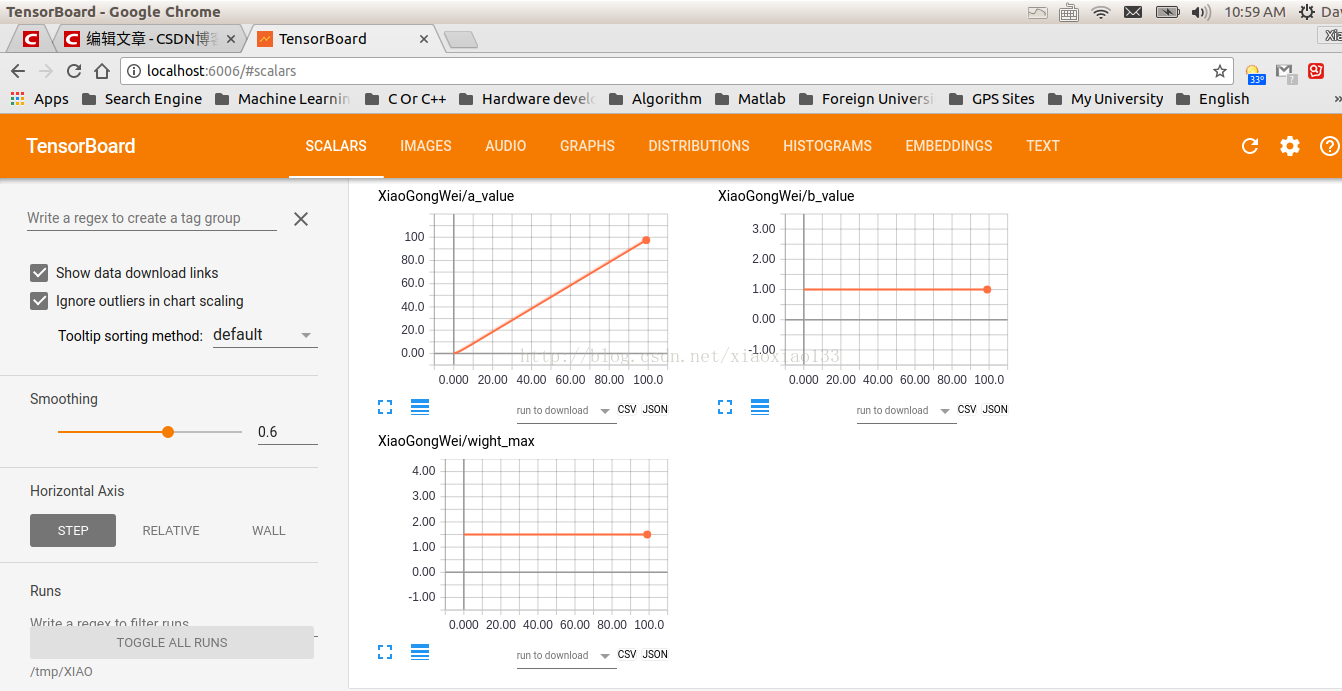
train\_summary.add\_summary(tfmensumm,i)

train\_summary.close()

运行代码1次后会在对应/tmp/XIAO出现一个文件

然后终端运行 tensorboard --logdir=/tmp/XIAO

打开浏览器输入:0.0.0.0:6006就会出现scalar保存结果



## tf.device

如果你不想使用系统来为 operation 指派设备, 而是手工指派设备, 你可以用 with tf.device 创建一个设备环境, 这个环境下的 operation 都统一运行在环境指定的设备上.

# 新建一个graph.

with tf.device('/cpu:0'):

a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')

b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')

c = tf.matmul(a, b)

# 新建session with log\_device\_placement并设置为True.

sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))

# 运行这个op.

print sess.run(c)

你会发现现在 a 和 b 操作都被指派给了 cpu:0.

Device mapping:

/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus

id: 0000:05:00.0

b: /job:localhost/replica:0/task:0/cpu:0

a: /job:localhost/replica:0/task:0/cpu:0

MatMul: /job:localhost/replica:0/task:0/gpu:0

[[ 22. 28.]

[ 49. 64.]]

## tf.split

tf.split(dimension, num\_split, input)：dimension的意思就是输入张量的哪一个维度，如果是0就表示对第0维度进行切割。num\_split就是切割的数量，如果是2就表示输入张量被切成2份，每一份是一个列表。

例如：

[python] [view plain](http://blog.csdn.net/uestc_c2_403/article/details/73350457" \o "view plain) [copy](http://blog.csdn.net/uestc_c2_403/article/details/73350457" \o "copy) [print](http://blog.csdn.net/uestc_c2_403/article/details/73350457" \o "print)[?](http://blog.csdn.net/uestc_c2_403/article/details/73350457" \o "?)

import tensorflow as tf;

import numpy as np;

A = [[1,2,3],[4,5,6]]

x = tf.split(1, 3, A)

with tf.Session() as sess:

    c = sess.run(x)

    for ele in c:

        print ele

## embedding\_lookup

input\_ids = tf.placeholder(dtype=tf.int32, shape=[None])

embedding = tf.Variable(np.identity(5, dtype=np.int32))

input\_embedding = tf.nn.embedding\_lookup(embedding, input\_ids)

sess = tf.InteractiveSession()

sess.run(tf.global\_variables\_initializer())

print(embedding.eval())

print(sess.run(input\_embedding, feed\_dict={input\_ids:[1, 2, 3, 0, 3, 2, 1]}))

代码中先使用palceholder定义了一个未知变量input\_ids用于存储索引，和一个已知变量embedding，是一个5\*5的对角矩阵。   
运行结果为：

embedding = [[1 0 0 0 0]

[0 1 0 0 0]

[0 0 1 0 0]

[0 0 0 1 0]

[0 0 0 0 1]]

input\_embedding = [[0 1 0 0 0]

[0 0 1 0 0]

[0 0 0 1 0]

[1 0 0 0 0]

[0 0 0 1 0]

[0 0 1 0 0]

[0 1 0 0 0]]

简单的讲就是根据input\_ids中的id，寻找embedding中的对应元素。比如，input\_ids=[1,3,5]，则找出embedding中下标为1,3,5的向量组成一个矩阵返回。

如果将input\_ids改写成下面的格式：

input\_embedding = tf.nn.embedding\_lookup(embedding, input\_ids)print(sess.run(input\_embedding, feed\_dict={input\_ids:[[1, 2], [2, 1], [3, 3]]}))

输出结果就会变成如下的格式：

[[[0 1 0 0 0]

[0 0 1 0 0]]

[[0 0 1 0 0]

[0 1 0 0 0]]

[[0 0 0 1 0]

[0 0 0 1 0]]]

对比上下两个结果不难发现，相当于在np.array中直接采用下标数组获取数据。需要注意的细节是返回的tensor的dtype和传入的被查询的tensor的dtype保持一致；和ids的dtype无关。

## np.identity() 和 np.eye()

两者在创建单位矩阵上，并无区别，两者的区别主要在接口上；

np.identity(n, dtype=None)：只能获取方阵，也即标准意义的单位阵；

np.eye(N, M=None, k=0, dtype=<type ‘float’>)；

N : int，Number of rows in the output.（行数，必选）

M : int, optional，Number of columns in the output. If None, defaults to N.（）

np.eye可以创建矩形矩阵，且k值可以调节，为1的对角线的位置偏离度，0居中，1向上偏离1，2偏离2，以此类推，-1向下偏离。值绝对值过大就偏离出去了，整个矩阵就全是0了。

## gradient

tensorflow中有一个计算梯度的函数tf.gradients(ys, xs)，要注意的是，xs中的x必须要与ys相关，不相关的话，会报错。   
代码中定义了两个变量w1， w2， 但res只与w1相关

#wrongimport tensorflow as tf

w1 = tf.Variable([[1,2]])

w2 = tf.Variable([[3,4]])

res = tf.matmul(w1, [[2],[1]])

grads = tf.gradients(res,[w1,w2])

with tf.Session() as sess:

tf.global\_variables\_initializer().run()

re = sess.run(grads)

print(re)

错误信息   
TypeError: Fetch argument None has invalid type

# rightimport tensorflow as tf

w1 = tf.Variable([[1,2]])

w2 = tf.Variable([[3,4]])

res = tf.matmul(w1, [[2],[1]])

grads = tf.gradients(res,[w1])

with tf.Session() as sess:

tf.global\_variables\_initializer().run()

re = sess.run(grads)

print(re)# [array([[2, 1]], dtype=int32)]

对于grad\_ys的测试：

import tensorflow as tf

w1 = tf.get\_variable('w1', shape=[3])

w2 = tf.get\_variable('w2', shape=[3])

w3 = tf.get\_variable('w3', shape=[3])

w4 = tf.get\_variable('w4', shape=[3])

z1 = w1 + w2+ w3

z2 = w3 + w4

grads = tf.gradients([z1, z2], [w1, w2, w3, w4], grad\_ys=[tf.convert\_to\_tensor([2.,2.,3.]),

tf.convert\_to\_tensor([3.,2.,4.])])

with tf.Session() as sess:

tf.global\_variables\_initializer().run()

print(sess.run(grads))

[array([ 2., 2., 3.],dtype=float32),

array([ 2., 2., 3.], dtype=float32),

array([ 5., 4., 7.], dtype=float32),

array([ 3., 2., 4.], dtype=float32)]

### tf.stop\_gradient

阻挡节点BP的梯度

import tensorflow as tf

w1 = tf.Variable(2.0)

w2 = tf.Variable(2.0)

a = tf.multiply(w1, 3.0)

a\_stoped = tf.stop\_gradient(a)

# b=w1\*3.0\*w2

b = tf.multiply(a\_stoped, w2)

gradients = tf.gradients(b, xs=[w1, w2])

print(gradients)#输出#[None, <tf.Tensor 'gradients/Mul\_1\_grad/Reshape\_1:0' shape=() dtype=float32>]

可见，一个节点被 stop之后，这个节点上的梯度，就无法再向前BP了。由于w1变量的梯度只能来自a节点，所以，计算梯度返回的是None。

a = tf.Variable(1.0)

b = tf.Variable(1.0)

c = tf.add(a, b)

c\_stoped = tf.stop\_gradient(c)

d = tf.add(a, b)

e = tf.add(c\_stoped, d)

gradients = tf.gradients(e, xs=[a, b])

with tf.Session() as sess:

tf.global\_variables\_initializer().run()

print(sess.run(gradients))#输出 [1.0, 1.0]

虽然 c节点被stop了，但是a，b还有从d传回的梯度，所以还是可以输出梯度值的。

import tensorflow as tf

w1 = tf.Variable(2.0)

w2 = tf.Variable(2.0)

a = tf.multiply(w1, 3.0)

a\_stoped = tf.stop\_gradient(a)

# b=w1\*3.0\*w2

b = tf.multiply(a\_stoped, w2)

opt = tf.train.GradientDescentOptimizer(0.1)

gradients = tf.gradients(b, xs=tf.trainable\_variables())

tf.summary.histogram(gradients[0].name, gradients[0])# 这里会报错，因为gradients[0]是None#其它地方都会运行正常，无论是梯度的计算还是变量的更新。总觉着tensorflow这么设计有点不好，#不如改成流过去的梯度为0

train\_op = opt.apply\_gradients(zip(gradients, tf.trainable\_variables()))

print(gradients)with tf.Session() as sess:

tf.global\_variables\_initializer().run()

print(sess.run(train\_op))

print(sess.run([w1, w2]))

### 高阶导数

tensorflow 求 高阶导数可以使用 tf.gradients 来实现

import tensorflow as tf

with tf.device('/cpu:0'):

a = tf.constant(1.)

b = tf.pow(a, 2)

grad = tf.gradients(ys=b, xs=a) # 一阶导

print(grad[0])

grad\_2 = tf.gradients(ys=grad[0], xs=a) # 二阶导

grad\_3 = tf.gradients(ys=grad\_2[0], xs=a) # 三阶导

print(grad\_3)

with tf.Session() as sess:

print(sess.run(grad\_3))

Note: 有些 op，tf 没有实现其高阶导的计算，例如 tf.add …, 如果计算了一个没有实现 高阶导的 op的高阶导， gradients 会返回 None。

## legacy\_seq2seq

入门教程

<http://blog.csdn.net/liuchonge/article/details/71424432>

源码解读

http://blog.csdn.net/u012871493/article/details/72350332

### rnn\_decoder

在计算loss的时候，最常见的一句话就是tf.nn.softmax\_cross\_entropy\_with\_logits，那么它到底是怎么做的呢？

首先明确一点，loss是代价值，也就是我们要最小化的值

## tf.nn.softmax\_cross\_entropy\_with\_logits(logits, labels, name=None)

除去name参数用以指定该操作的name，与方法有关的一共两个参数：

第一个参数logits：就是神经网络最后一层的输出，如果有batch的话，它的大小就是[batchsize，num\_classes]，单样本的话，大小就是num\_classes

第二个参数labels：实际的标签，大小同上

具体的执行流程大概分为两步：

第一步是先对网络最后一层的输出做一个softmax，这一步通常是求取输出属于某一类的概率，对于单样本而言，输出就是一个num\_classes大小的向量（[Y1，Y2,Y3...]其中Y1，Y2，Y3...分别代表了是属于该类的概率）

softmax的公式是：

至于为什么是用的这个公式？这里不介绍了，涉及到比较多的理论证明

第二步是softmax的输出向量[Y1，Y2,Y3...]和样本的实际标签做一个交叉熵，公式如下：

其中指代实际的标签中第i个的值（用mnist数据举例，如果是3，那么标签是[0，0，0，1，0，0，0，0，0，0]，除了第4个值为1，其他全为0）

就是softmax的输出向量[Y1，Y2,Y3...]中，第i个元素的值

显而易见，预测越准确，结果的值越小（别忘了前面还有负号），最后求一个平均，得到我们想要的loss

注意！！！这个函数的返回值并不是一个数，而是一个向量，如果要求交叉熵，我们要再做一步tf.reduce\_sum操作,就是对向量里面所有元素求和，最后才得到，如果求loss，则要做一步tf.reduce\_mean操作，对向量求均值！

理论讲完了，上代码

import tensorflow as tf

#our NN's output

logits=tf.constant([[1.0,2.0,3.0],[1.0,2.0,3.0],[1.0,2.0,3.0]])

#step1:do softmax

y=tf.nn.softmax(logits)

#true label

y\_=tf.constant([[0.0,0.0,1.0],[0.0,0.0,1.0],[0.0,0.0,1.0]])

#step2:do cross\_entropy

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y))

#do cross\_entropy just one step

cross\_entropy2=tf.reduce\_sum(tf.nn.softmax\_cross\_entropy\_with\_logits(logits, y\_))#dont forget tf.reduce\_sum()!!

with tf.Session() as sess:

softmax=sess.run(y)

c\_e = sess.run(cross\_entropy)

c\_e2 = sess.run(cross\_entropy2)

print("step1:softmax result=")

print(softmax)

print("step2:cross\_entropy result=")

print(c\_e)

print("Function(softmax\_cross\_entropy\_with\_logits) result=")

print(c\_e2)

输出结果是：

step1:softmax result=

[[ 0.09003057 0.24472848 0.66524094]

[ 0.09003057 0.24472848 0.66524094]

[ 0.09003057 0.24472848 0.66524094]]

step2:cross\_entropy result=

1.22282

Function(softmax\_cross\_entropy\_with\_logits) result=

1.2228

最后大家可以试试e^1/(e^1+e^2+e^3)是不是0.09003057，发现确实一样！！这也证明了我们的输出是符合公式逻辑的

## 交叉熵

交叉熵可在神经网络(机器学习)中作为损失函数，p表示真实标记的分布，q则为训练后的模型的预测标记分布，交叉熵损失函数可以衡量p与q的相似性。交叉熵作为损失函数还有一个好处是使用sigmoid函数在梯度下降时能避免均方误差损失函数学习速率降低的问题，因为学习速率可以被输出的误差所控制。tensorflow中自带的函数可以轻松的实现交叉熵的计算。

tf.nn.softmax\_cross\_entropy\_with\_logits(\_sentinel=None, labels=None, logits=None, dim=-1, name=None)

Computes softmax cross entropy between logits and labels.

注意：如果labels的每一行是one-hot表示，也就是只有一个地方为1，其他地方为0，可以使用tf.sparse\_softmax\_cross\_entropy\_with\_logits()

警告：

1. 这个操作的输入logits是未经缩放的，该操作内部会对logits使用softmax操作

2. 参数labels,logits必须有相同的形状 [batch\_size, num\_classes] 和相同的类型(float16, float32, float64)中的一种

参数：\_sentinel: 一般不使用

labels: labels的每一行labels[i]必须为一个概率分布

logits: 未缩放的对数概率

dims: 类的维度，默认-1，也就是最后一维

name: 该操作的名称

返回值：长度为batch\_size的一维Tensor

下面用个小例子来看看该函数的用法

import tensorflow as tf

labels = [[0.2,0.3,0.5],

[0.1,0.6,0.3]]

logits = [[2,0.5,1],

[0.1,1,3]]

logits\_scaled = tf.nn.softmax(logits)

result1 = tf.nn.softmax\_cross\_entropy\_with\_logits(labels=labels, logits=logits)

result2 = -tf.reduce\_sum(labels\*tf.log(logits\_scaled),1)

result3 = tf.nn.softmax\_cross\_entropy\_with\_logits(labels=labels, logits=logits\_scaled)

with tf.Session() as sess:

print sess.run(result1)

print sess.run(result2)

print sess.run(result3)

>>>[ 1.41436887 1.66425455]

>>>[ 1.41436887 1.66425455]

>>>[ 1.17185783 1.17571414]

上述例子中，labels的每一行是一个概率分布，而logits未经缩放(每行加起来不为1)，我们用定义法计算得到交叉熵result2,和套用tf.nn.softmax\_cross\_entropy\_with\_logits()得到相同的结果, 但是将缩放后的logits\_scaled输tf.nn.softmax\_cross\_entropy\_with\_logits(), 却得到错误的结果，所以一定要注意，这个操作的输入logits是未经缩放的

下面来看tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(\_sentinel=None, labels=None, logits=None, name=None)

这个函数与上一个函数十分类似，唯一的区别在于labels.

注意：对于此操作，给定标签的概率被认为是排他的。labels的每一行为真实类别的索引

警告：

1. 这个操作的输入logits同样是是未经缩放的，该操作内部会对logits使用softmax操作

2. 参数logits的形状 [batch\_size, num\_classes] 和labels的形状[batch\_size]

返回值：长度为batch\_size的一维Tensor, 和label的形状相同，和logits的类型相同

import tensorflow as tf

labels = [0,2]

logits = [[2,0.5,1],

[0.1,1,3]]

logits\_scaled = tf.nn.softmax(logits)

result1 = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=labels, logits=logits)

with tf.Session() as sess:

print sess.run(result1)

>>>[ 0.46436879 0.17425454]

## 如果梯度为None进行clip\_by\_value会报错

Replace

map(lambda gv: [tf.clip\_by\_value(gv[0], -10., 10.), gv[1]], grads\_vars\_G)

by

map(lambda gv: gv if gv[0] is None else [tf.clip\_by\_value(gv[0], -10., 10.), gv[1]], grads\_vars\_G)

The optimizer yields None when gradients are null instead of zeroes.

## pad\_sequences

填充序列

keras.preprocessing.sequence.pad\_sequences(sequences, maxlen=None, dtype='int32',

padding='pre', truncating='pre', value=0.)

将长为nb\_samples的序列（标量序列）转化为形如(nb\_samples,nb\_timesteps)2D numpy array。如果提供了参数maxlen，nb\_timesteps=maxlen，否则其值为最长序列的长度。其他短于该长度的序列都会在后部填充0以达到该长度。长于nb\_timesteps的序列将会被截断，以使其匹配目标长度。padding和截断发生的位置分别取决于padding和truncating.

参数

•sequences：浮点数或整数构成的两层嵌套列表

•maxlen：None或整数，为序列的最大长度。大于此长度的序列将被截短，小于此长度的序列将在后部填0.

•dtype：返回的numpy array的数据类型

•padding：‘pre’或‘post’，确定当需要补0时，在序列的起始还是结尾补

•truncating：‘pre’或‘post’，确定当需要截断序列时，从起始还是结尾截断

•value：浮点数，此值将在填充时代替默认的填充值0

返回值

返回形如(nb\_samples,nb\_timesteps)的2D张量

## Flags

import tensorflow as tf

flags = tf.flags #flags是一个文件：flags.py，用于处理命令行参数的解析工作

logging = tf.logging

#调用flags内部的DEFINE\_string函数来制定解析规则

flags.DEFINE\_string("para\_name\_1","default\_val", "description")

flags.DEFINE\_bool("para\_name\_2","default\_val", "description")

#FLAGS是一个对象，保存了解析后的命令行参数

FLAGS = flags.FLAGS

def main(\_):

FLAGS.para\_name #调用命令行输入的参数

if \_\_name\_\_ == "\_\_main\_\_": #使用这种方式保证了，如果此文件被其它文件import的时候，不会执行main中的代码

tf.app.run() #解析命令行参数，调用main函数 main(sys.argv)

调用方法：

~/ python script.py --para\_name\_1=name --para\_name\_2=name2

## tf.contrib.crf.crf\_log\_likelihood

crf\_log\_likelihood(inputs,tag\_indices,sequence\_lengths,transition\_params=None)

在一个条件随机场里面计算标签序列的log-likelihood

参数:

inputs: 一个形状为[batch\_size, max\_seq\_len, num\_tags] 的tensor,一般使用BILSTM处理之后输出转换为他要求的形状作为CRF层的输入.

tag\_indices: 一个形状为[batch\_size, max\_seq\_len] 的矩阵,其实就是真实标签.

sequence\_lengths: 一个形状为 [batch\_size] 的向量,表示每个序列的长度.

transition\_params: 形状为[num\_tags, num\_tags] 的转移矩阵

返回：

log\_likelihood: 标量,log-likelihood

transition\_params: 形状为[num\_tags, num\_tags] 的转移矩阵