

CIFAR-10 图像分类

一、基础python 语法

如果某个版本中出现了某个新的功能特性，而且这个特性和当前版本中使用的的不兼容，也就是它在该版本中不是语言标准，那么我如果想要使用的话就需要从future模块导入。

1.1 from future import print_function 用法

在开头加上from **future** import print_function这句之后，即使在python2.X，使用print就得像python3.X那样加括号使用。（python2.X中print不需要括号，而在python3.X中则需要。）

1.2 from future import division

在python2 中导入未来的支持的语言特征中division(精确除法), 但是python3中使用的是精确除法

导入python未来支持的语言特征division(精确除法)，当我们没有在程序中导入该特征时，"/"操作符执行的是截断除法(Truncating Division),当我们导入精确除法之后，"/"执行的是精确除法，如下所示：

```
>>> 3/4
0
>>> from __future__ import division
>>> 3/4
0.75
```

1.3 from future import absolute_import

python中的绝对导入与相对导入

相对导入：在不指明 package 名的情况下导入自己这个 package 的模块，比如一个 package 下有 a.py 和 b.py 两个文件，在 a.py 里 from . import b 即是相对导入 b.py

绝对导入：指明顶层 package 名。比如 import a，Python 会在 sys.path里寻找所有名为 a 的顶层模块。

1.4 import re

Python 的 re 模块（Regular Expression 正则表达式）提供各种正则表达式的匹配操作，在文本解析、复杂字符串分析和信息提取时是一个非常有用的工具，下面我主要总结了re的常用方法。

```
print re.doc
```

[re模块的用法](#)

1.5 re.sub(pattern, repl, string, count=0, flags=0)

- pattern：表示正则表达式中的模式字符串；
- repl：被替换的字符串（既可以是字符串，也可以是函数）；
- string：要被处理的，要被替换的字符串；
- count：匹配的次数, 默认是全部替换
- flags：具体用处不详

```
Python 3.5.6 (default, Mar 29 2019, 21:29:42)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> boy_name = 'Tom and jupy'
>>> re.sub('Tom|jupy', 'nacy', boy_name)
'nacy and nacy'
```

1.6 import tarfile

Read and write tar archive files

打包及重命名文件

```
>>> import tarfile
# 以w模式创建文件
>>> tar = tarfile.open('tar_file.tar', 'w')
# 添加一个文件, arcname可以重命名文件
>>> tar.add('/tmp/folder/file.txt', arcname='file.log')
# 添加一个目录
>>> tar.add('/tmp/folder/tmp')
# 关闭
>>> tar.close()
```

1.6 from six.moves import urllib

urllib.request.urlretrieve(url,[filepath,[recall_func,[data]]])

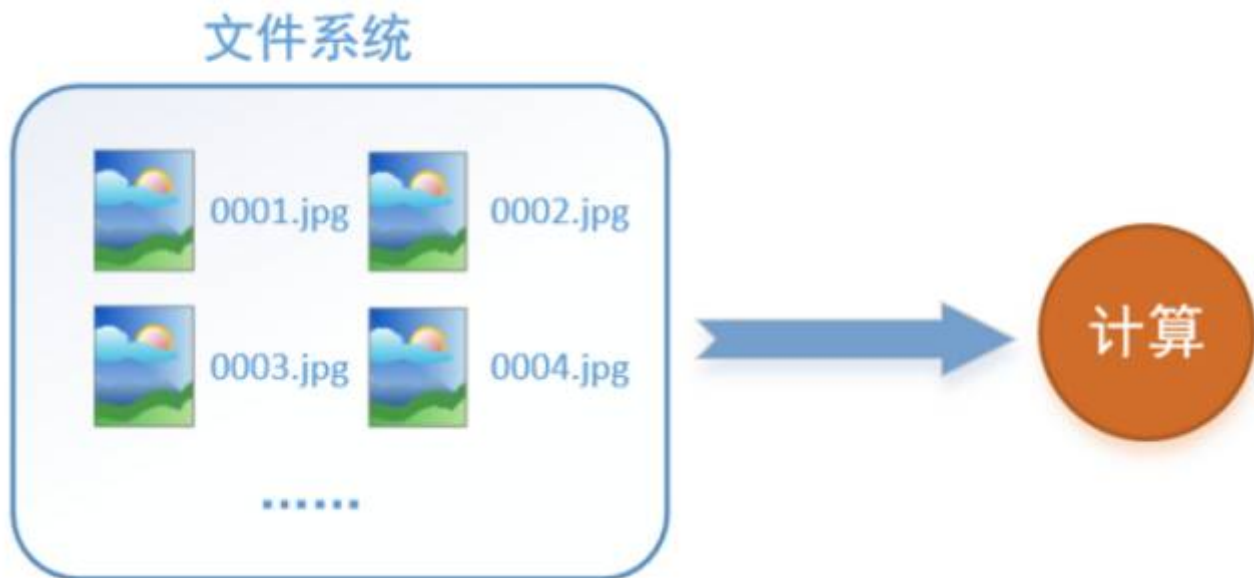
```
from six.moves import urllib
import os
import sys
import tarfile
```

```
FLAGS = tf.app.flags.FLAGS#提取系统参数作用的变量
tf.app.flags.DEFINE_string('dir', 'D:/download_html', 'directory of
html')#将下载目录保存到变量dir中,通过FLAGS.dir提取
directory = FLAGS.dir#从FLAGS中提取dir变量
```

```
url = 'http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz'
filename = url.split('/')[ -1]#-1表示分割后的最后一个元素
filepath = os.path.join(directory, filename)
if not os.path.exists(directory):
    os.makedirs(directory)
if not os.path.exists(filepath):
    def _recall_func(num, block_size, total_size):
        sys.stdout.write('\r>> downloading %s %.1f%%' %
(filename, float(num*block_size)/float(total_size)*100.0))
        sys.stdout.flush()
    urllib.request.urlretrieve(url, filepath, _recall_func)
    print()
    file_info = os.stat(filepath)
    print('Successfully download', filename, file_info.st_size, 'bytes')
```

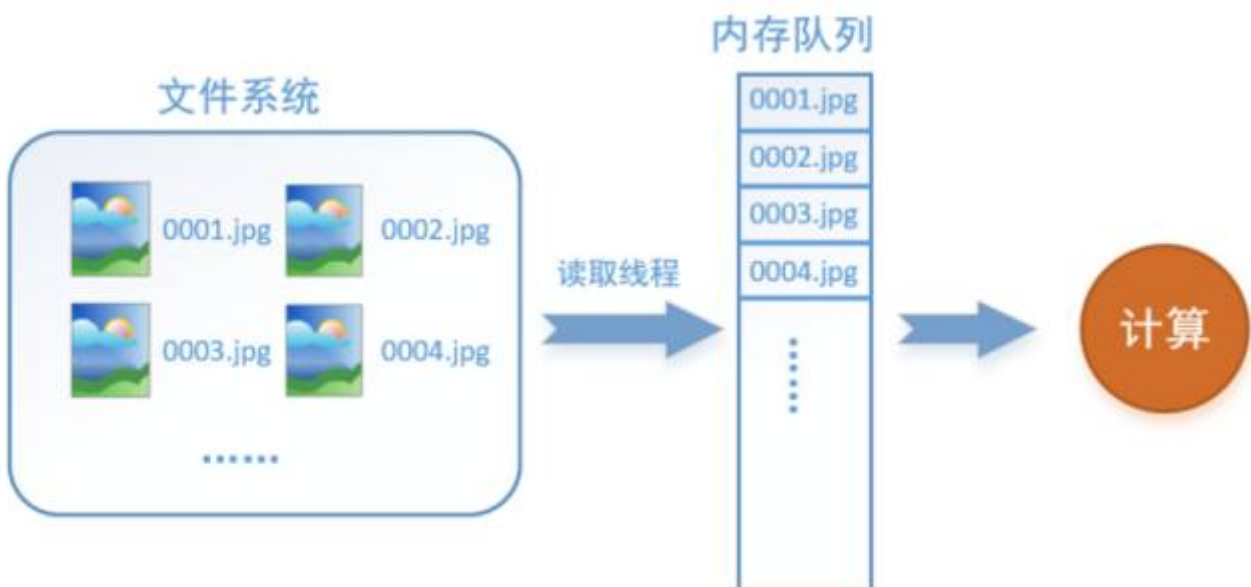
```
tar = tarfile.open(filepath, 'r:gz') #指定解压路径和解压方式为解压gzip  
tar.extractall(directory) #全部解压
```

二、TensorFlow读取机制图解

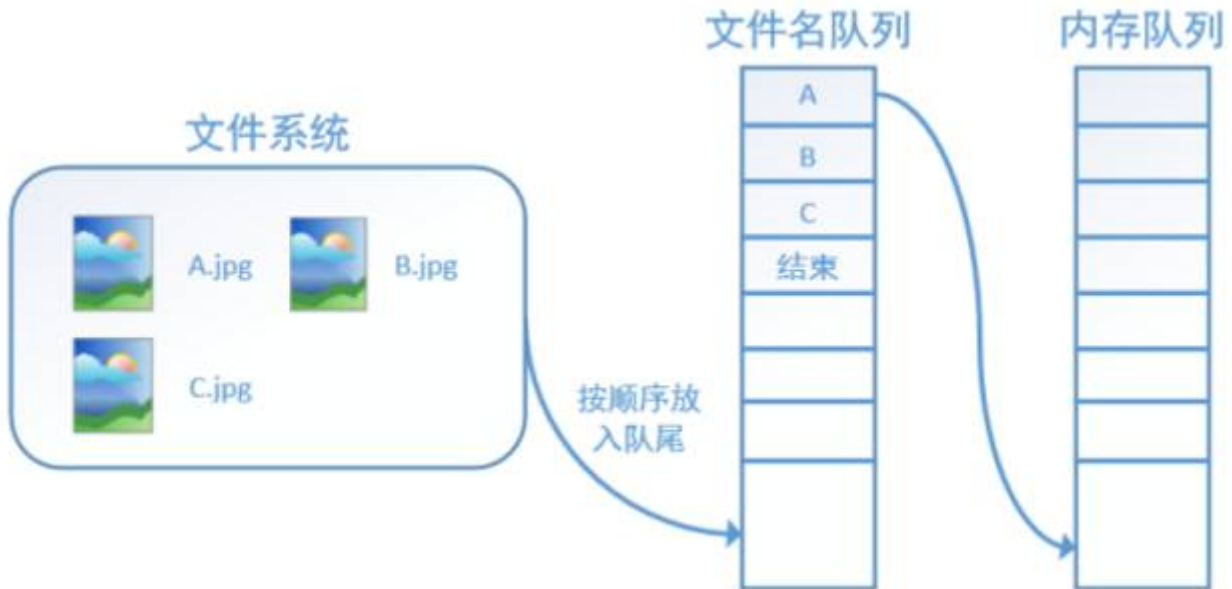


假设我们的硬盘中有一个图片数据集0001.jpg, 0002.jpg, 0003.jpg.....我们只需要把它们读取到内存中, 然后提供给GPU或是CPU进行计算就可以了。这听起来很容易, 但事实远没有那么简单。事实上, 我们必须要把数据先读入后才能进行计算, 假设读入用时0.1s, 计算用时0.9s, 那么就意味着每过1s, GPU都会有0.1s无事可做, 这就大大降低了运算的效率。

如何解决这个问题? 方法就是将读入数据和计算分别放在两个线程中, 将数据读入内存的一个队列, 如下图所示:



TensorFlow使用文件名队列+内存队列双队列的形式读入文件, 可以很好地管理epoch。下面我们用图片的形式来说明这个机制的运行方式。如下图, 还是以数据集A.jpg, B.jpg, C.jpg为例, 假定我们要跑一个epoch, 那么我们就在文件名队列中把A、B、C各放入一次, 并在之后标注队列结束。



对于文件名队列，我们使用`tf.train.string_input_producer`函数。这个函数需要传入一个文件名list，系统会自动将它转为一个文件名队列。

- `num_epochs`：它就是我们上文中提到的epoch数
- `shuffle`：是指在一个epoch内文件的顺序是否被打乱。

```
tf.train.string_input_producer(
    string_tensor,
    num_epochs=None,
    shuffle=True,
    seed=None,
    capacity=32,
    shared_name=None,
    name=None,
    cancel_op=None
)
```

2.1 tf.WholeFileReader()

读取队列目录中的所有文件，并把文件 **全部内容** 提取出key和value返回。

通常该函数一般用在一个文件就是一个图片的情况下。

读取的代码如下

```
# 输入文件列表
filename = ['A.jpg', 'B.jpg', 'C.jpg']
# 创建文件队列
filename_queue = tf.train.string_input_producer(filename, shuffle=False,
                                                num_epochs=5)

# 创建reader对象
reader = tf.WholeFileReader()
# 从队列中读取key和value值
key, value = reader.read(filename_queue)
# 初始化变量
tf.local_variables_initializer().run()
```

```
# 入栈线程启动
```

```
threads = tf.train.start_queue_runners(sess=sess)
```

如果要使用,请在队列(Queue)中的排列文件名.Read的输出将是一个文件名(key)和该文件的内容(value).

2.2 tf.FixedLengthRecordReader()

固定长度记录一个或多个二进制文件，送入队列中。

tf.FixedLengthRecordReader (record_bytes, header_bytes=None, footer_bytes=None, hop_bytes=None, name=None, encoding=None)

record_bytes: 整形数，输出的Record中每个文件的长度

```
label_bytes = 1 # 2 for CIFAR-100
result.height = 32
result.width = 32
result.depth = 3
image_bytes = result.height * result.width * result.depth
# Every record consists of a label followed by the image, with a
# fixed number of bytes for each.
record_bytes = label_bytes + image_bytes

# Read a record, getting filenames from the filename_queue. No
# header or footer in the CIFAR-10 format, so we leave header_bytes
# and footer_bytes at their default of 0.
reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
result.key, value = reader.read(filename_queue)
```

▼ 三、tensorflow 相关操作

3.1 tf.app.flags.DEFINE_xxx() 和 tf.app.flags.FLAGS

tf.app.flags.DEFINE_xxx()就是添加命令行的optional argument（可选参数），而tf.app.flags.FLAGS可以从对应的命令行参数取出参数。

3.1.1 新建test.py文件

在文件中输入如下代码，代码的功能是创建几个命令行参数，然后把命令行参数输出显示

```
# test.py
import tensorflow as tf

FLAGS=tf.app.flags.FLAGS

tf.app.flags.DEFINE_float(
    'flag_float', 0.01, 'input a float')# 定义命令行参数 --flag_float

tf.app.flags.DEFINE_integer(
    'flag_int', 400, 'input a int')
tf.app.flags.DEFINE_boolean(
    'flag_bool', True, 'input a bool')
```

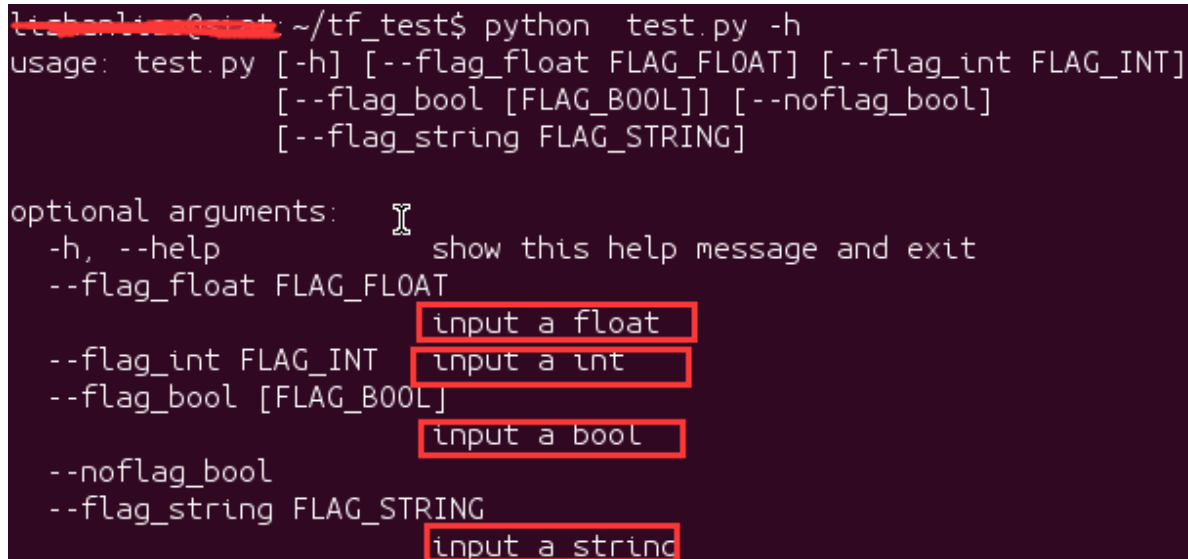
```
tf.app.flags.DEFINE_string(
    'flag_string', 'yes', 'input a string')

print(FLAGS.flag_float)
print(FLAGS.flag_int)
print(FLAGS.flag_bool)
print(FLAGS.flag_string)
```

3.1.2 在命令终端中查看效果图

在命令行中查看帮助信息，在命令行输入 `python test.py -h`

如下图所示：

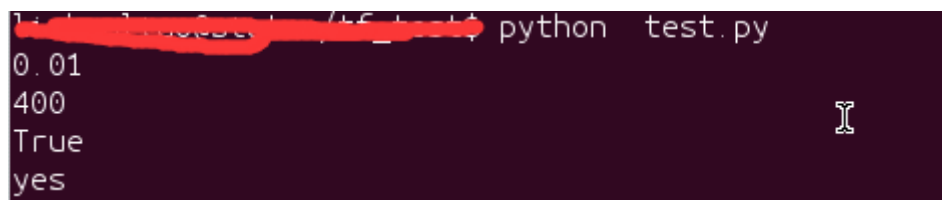


```
litahonli@cs101: ~/tf_test$ python test.py -h
usage: test.py [-h] [--flag_float FLAG_FLOAT] [--flag_int FLAG_INT]
               [--flag_bool [FLAG_BOOL]] [--noflag_bool]
               [--flag_string FLAG_STRING]

optional arguments:
  -h, --help            show this help message and exit
  --flag_float FLAG_FLOAT
                        input a float
  --flag_int FLAG_INT   input a int
  --flag_bool [FLAG_BOOL]
                        input a bool
  --noflag_bool
  --flag_string FLAG_STRING
                        input a string
```

注意红色框中的信息，这个就是我们用 `DEFINE_XXX` 添加命令行参数时的第三个参数

3.1.3 直接运行test.py



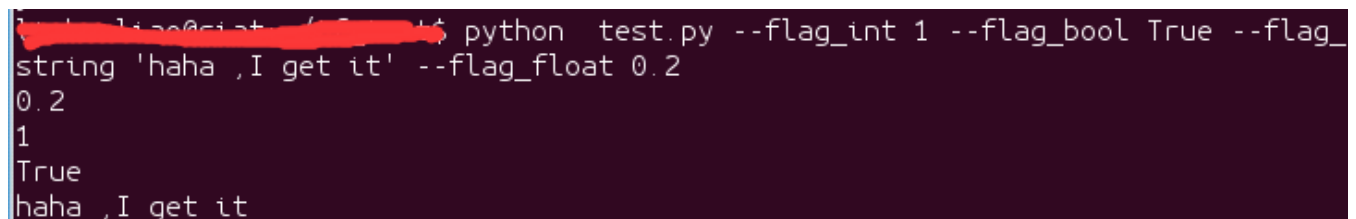
```
litahonli@cs101: ~/tf_test$ python test.py
0.01
400
True
yes
```

运行后的效果图如上图所示，由于没有输入参数所以直接使用默认的参数作为输入。

3.1.4 带命令行参数的运行test.py文件

```
python3 test.py --flag_int 1 --flag_bool True --flag_string 'I m get
something wrong'
```

执行后的效果如下图所示



```
litahonli@cs101: ~/tf_test$ python test.py --flag_int 1 --flag_bool True --flag_string 'haha ,I get it' --flag_float 0.2
0.2
1
True
haha ,I get it
```

3.2 tensorflow 可视化操作

tf.summary在tensorflow中的summary.py 文件中，里面包括了所有的tensorflow的可视化函数

3.2.1 tf.summary.histogram 绘制可视化的直方图

```
tf.summary.histogram(
    name,
    values,
    collections=None,
    family=None
)
```

- name: 生成的节点名称.作为TensorBoard中的一个系列名称.
- values: 一个实数张量.用于构建直方图的值.
- collections: 图形集合键的可选列表.添加新的summary操作到这些集合中.默认为GraphKeys.SUMMARIES.
- family: summary标签名称的前缀,用于在Tensorboard上显示的标签名称.(可选项)

```
import tensorflow as tf
k = tf.placeholder(tf.float32)
# 创建一个均值变化的正态分布 (由0到5左右)
mean_moving_normal = tf.random_normal(shape=[1000], mean=(5*k),
stddev=1)
# 将该分布记录到直方图汇总中
tf.summary.histogram("normal/moving_mean", mean_moving_normal)
sess = tf.Session()
writer = tf.summary.FileWriter("./temp/histogram_example")
summaries = tf.summary.merge_all()
# 设置一个循环并将摘要写入磁盘
N = 400
for step in range(N):
    k_val = step/float(N)
    summ = sess.run(summaries, feed_dict={k: k_val})
    writer.add_summary(summ, global_step=step)
```

终端中执行python3 test.py

然后执行 tensorboard --logdir histogram_example/

打开浏览器效果如下图所示:

 tensorboard 效果图

3.2.2 tf.summary.scalar

```
tf.summary.scalar(
    name,
    tensor,
    collections=None,
```

```
    family=None  
)
```

- name:生成节点的名字，也会作为TensorBoard中的系列的名字。
- tensor:包含一个值的实数Tensor。
- collection: 图的集合键值的可选列表。新的求和op被添加到这个集合中。缺省为 [GraphKeys.SUMMARIES]
- family:可选项；设置时用作求和标签名称的前缀，这影响着TensorBoard所显示的标签名。

```
import tensorflow as tf
```

```
a = tf.placeholder(tf.float32, shape=[])  
b = tf.constant(1, dtype=tf.int32)
```

```
tf.summary.scalar("a", a)  
tf.summary.scalar("b", b)
```

```
sess = tf.Session()
```

```
init_op = tf.global_variables_initializer()  
merged_summaries = tf.summary.merge_all()  
writer = tf.summary.FileWriter("train", sess.graph)
```

```
sess.run(init_op)
```

```
for step in range(6):  
    feed_dict = {a: step}  
    summary = sess.run(merged_summaries, feed_dict=feed_dict)  
    writer.add_summary(summary=summary, global_step=step)
```

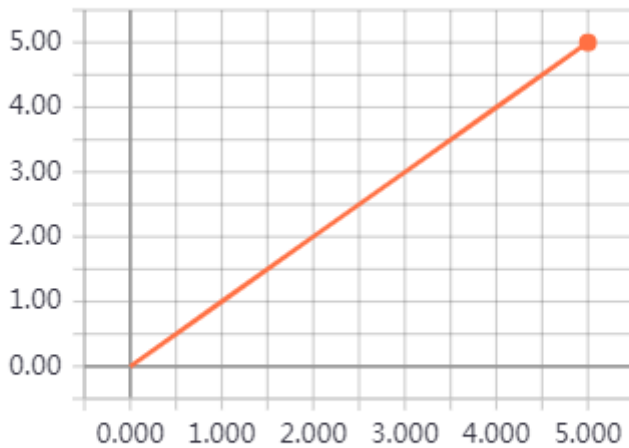
作者: z2539329562

来源: CSDN

原文: <https://blog.csdn.net/z2539329562/article/details/84201113>

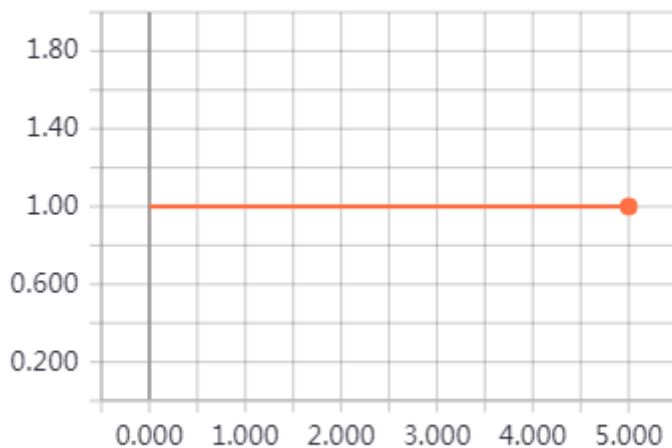
版权声明: 本文为博主原创文章, 转载请附上博文链接!

a



b

b


<https://blog.csdn.net/z2539329562>

3.2.3 tf.summary.merge_all()

分两种管理相关变量的方法，一种是自动管理模式一种是手动选择性的管理模式。

注意：自动管理模式下，导入已保存的模型继续训练时，会抛异常，该种场景下建议使用手动模式

3.2.3.1. 自动管理模式

```
summary_writer = tf.summary.FileWriter('E:/data/tensorflow-
master/1.Cnn_Captcha/result/', flush_secs=60)
summary_writer.add_graph(sess.graph)#添加graph图
tf.summary.scalar('loss', loss)
tf.summary.scalar('accuracy', accuracy)
sum_ops = tf.summary.merge_all()#自动管理
metall = sess.run(sum_ops, feed_dict={X: batch_x_test, Y: batch_y_test,
keep_prob: 1.})
summary_writer.add_summary(metall, global_step=step) # 写入文件
```

作者：数学改变世界

来源：CSDN

原文：https://blog.csdn.net/qq_34197612/article/details/79313430

版权声明：本文为博主原创文章，转载请附上博文链接！

merge_all 可以将所有summary全部保存到磁盘，以便tensorboard显示。

3.2.3.2. 手动管理模式

```
summary_writer = tf.summary.FileWriter('E:/data/tensorflow-
master/1.Cnn_Captcha/result/', flush_secs=60)
summary_writer.add_graph(sess.graph)#添加graph图
loss_scalar = tf.summary.scalar('loss', loss)
accuracy_scalar = tf.summary.scalar('accuracy', accuracy)
loss_metall, accuracy_metall, = sess.run([loss_scalar,
accuracy_scalar], feed_dict={X: batch_x_test, Y: batch_y_test,
keep_prob: 1.})
summary_writer.add_summary(loss_metall, global_step=step) # 写入文件
summary_writer.add_summary(accuracy_metall, global_step=step) # 写入文件
```

作者：数学改变世界

来源：CSDN

原文：https://blog.csdn.net/qq_34197612/article/details/79313430

版权声明：本文为博主原创文章，转载请附上博文链接！

3.2.3.3 [writer.add_summary\(summary\)](#)

```
writer = tf.summary.FileWriter('./graphs', sess.graph)
writer.add_summary(summary)
```

将summary 添加到输出管理文件 **graphs** 中

3.2.4 tf.summary.image

```
tf.summary.image(
    name,
    tensor,
    max_outputs=3,
    collections=None,
    family=None
)
```

带图像的输出生

- **tensor**: 输出的图像
- **max_outputs**: 页面容器指定能够展示的最大图片数量

```
import tensorflow as tf
tf.reset_default_graph() # To clear the defined variables and
operations of the previous cell

# create the variables
```

```

w_gs = tf.get_variable('W_Grayscale', shape=[30, 10],
initializer=tf.truncated_normal_initializer(mean=0, stddev=1))
w_c = tf.get_variable('W_Color', shape=[50, 30],
initializer=tf.truncated_normal_initializer(mean=0, stddev=1))

# __step 0:__ reshape it to 4D-tensors
w_gs_resaped = tf.reshape(w_gs, (3, 10, 10, 1))
w_c_resaped = tf.reshape(w_c, (5, 10, 10, 3))

# __step 1:__ create the summaries
gs_summary = tf.summary.image('Grayscale', w_gs_resaped)
c_summary = tf.summary.image('Color', w_c_resaped, max_outputs=5)

# __step 2:__ merge all summaries
merged = tf.summary.merge_all()

# create the op for initializing all variables
init = tf.global_variables_initializer()

# launch the graph in a session
with tf.Session() as sess:
    # __step 3:__ creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    # initialize all variables
    sess.run(init)
    # __step 4:__ evaluate the merged op to get the summaries
    summary = sess.run(merged)
    # __step 5:__ add summary to the writer (i.e. to the event file)
    to write on the disc
    writer.add_summary(summary)
    print('Done writing the summaries')

```

 输出图像

3.2.5 Tensorflow-tf.nn.zero_fraction()

```

tf.nn.zero_fraction(
    value,
    name=None
)

```

- 参数1: value: 数值型的Tensor, 对其计算0元素在所有元素中所占比例.
- 参数2: name: 非必填, 设置操作的名称.
- 返回: 输入Tensor的0元素的个数与输入Tensor的所有元素的个数的比值.

作用是将输入的Tensor中0元素在所有元素中所占的比例计算并返回

因为relu激活函数有时会大面积的将输入参数设为0, 所以此函数可以有效衡量relu激活函数的有效性。

示例代码如下:

```

import tensorflow as tf
sess = tf.Session()

```

```
v = [[0 0 1 2] [10 0 11 211]]
```

3.3 tensorflow 硬件使用的配置函数

3.3.1 tf.device()

使用 `tf.device()` 指定模型运行的具体设备，可以指定运行在GPU还是CUP上，以及哪块GPU上。

3.3.1.1 设置使用GPU

```
import tensorflow as tf

with tf.device('/gpu:1'):
    v1 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v1')
    v2 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v2')
    sumV12 = v1 + v2

    with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
        print sess.run(sumV12)
```

使用 `tf.device('/gpu:1')` 指定Session在第二块GPU上运行

`ConfigProto()` 中参数 `log_device_placement=True` 会打印出执行操作所用的设备，以上输出：

```
add: (Add): /job:localhost/replica:0/task:0/gpu:1
2018-03-29 02:18:39.750293: I tensorflow/core/common_runtime/simple_placer.cc:847] add: (Add)/job:localhost/replica:0/task:0/gpu:1
v2: (Const): /job:localhost/replica:0/task:0/gpu:1
2018-03-29 02:18:39.750336: I tensorflow/core/common_runtime/simple_placer.cc:847] v2: (Const)/job:localhost/replica:0/task:0/gpu:1
v1: (Const): /job:localhost/replica:0/task:0/gpu:1
2018-03-29 02:18:39.750360: I tensorflow/core/common_runtime/simple_placer.cc:847] v1: (Const)/job:localhost/replica:0/task:0/gpu:1
[ 2.  4.  6.]
```

如果安装的是GPU版本的tensorflow，机器上有支持的GPU，也正确安装了显卡驱动、CUDA和cuDNN，默认情况下，Session会在GPU上运行：

```
import tensorflow as tf

v1 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v1')
v2 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v2')
sumV12 = v1 + v2

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    print sess.run(sumV12)
```

作者：-牧野-

来源：CSDN

原文：<https://blog.csdn.net/dcrmg/article/details/79747882>

版权声明：本文为博主原创文章，转载请附上博文链接！

默认在GPU:0上执行：

```
add: (Add): /job:localhost/replica:0/task:0/gpu:0
2018-03-29 02:13:04.970634: I tensorflow/core/common_runtime/simple_placer.cc:847] add: (Add)/job:localhost/replica:0/task:0/gpu:0
v2: (Const): /job:localhost/replica:0/task:0/gpu:0
2018-03-29 02:13:04.970670: I tensorflow/core/common_runtime/simple_placer.cc:847] v2: (Const)/job:localhost/replica:0/task:0/gpu:0
v1: (Const): /job:localhost/replica:0/task:0/gpu:0
2018-03-29 02:13:04.970698: I tensorflow/core/common_runtime/simple_placer.cc:847] v1: (Const)/job:localhost/replica:0/task:0/gpu:0
[ 2.  4.  6.]
```

3.3.1.2 设置使用cpu

tensorflow中不同的GPU使用/gpu:0和/gpu:1区分，而CPU不区分设备号，统一使用 /cpu:0

```
import tensorflow as tf

with tf.device('/cpu:0'):
    v1 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v1')
    v2 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v2')
    sumV12 = v1 + v2

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    print sess.run(sumV12)
```

作者：-牧野-

来源：CSDN

原文：<https://blog.csdn.net/dcrmg/article/details/79747882>

版权声明：本文为博主原创文章，转载请附上博文链接！

```
add: (Add): /job:localhost/replica:0/task:0/cpu:0
2018-03-29 02:06:55.793991: I tensorflow/core/common_runtime/simple_placer.cc:847] add: (Add)/job:localhost/replica:0/task:0/cpu:0
v2: (Const): /job:localhost/replica:0/task:0/cpu:0
2018-03-29 02:06:55.794034: I tensorflow/core/common_runtime/simple_placer.cc:847] v2: (Const)/job:localhost/replica:0/task:0/cpu:0
v1: (Const): /job:localhost/replica:0/task:0/cpu:0
2018-03-29 02:06:55.794083: I tensorflow/core/common_runtime/simple_placer.cc:847] v1: (Const)/job:localhost/replica:0/task:0/cpu:0
[ 2.  4.  6.]
```

3.3.2 tf.ConfigProto()

tensorflow中使用tf.ConfigProto()配置Session运行参数&&GPU设备指定

tf.ConfigProto()函数用在创建session的时候，用来对session进行参数配置

3.3.2.1 tf.ConfigProto(log_device_placement=True) # 只拍设备使用情况

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。

3.3.2.2 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。

3.3.2.3 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。

动态申请显存

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.Session(config=config)
```

3.3.2.4 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)
```

3.4 tensorflow常见函数

3.4.1 tf.train.string_input_producer

将文件list转化为文件名队列，在最新版的tf中已经被弃用

生成一个文件队列，每次read执行，出栈一个文件，后面文件自动填上

- num_epochs：是队列训练的回合数
- shuffle：指在一个epoch内文件的顺序是否打乱

3.4.2 tf.train.start_queue_runners

入栈线程启动器

- 调用 tf.train.slice_input_producer，从本地文件里抽取tensor，准备放入Filename Queue（文件名队列）中；
- 调用 tf.train.batch，从文件名队列中提取tensor，使用单个或多个线程，准备放入文件队列；
- 调用 tf.train.Coordinator() 来创建一个线程协调器，用来管理之后在Session中启动的所有线程；
- 调用tf.train.start_queue_runners, 启动入队线程，由多个或单个线程，按照设定规则，把文件读入Filename Queue中。函数返回线程ID的列表，一般情况下，系统有多少个核，就会启动多少个入队线程（入队具体使用多少个线程在tf.train.batch中定义）；
- 文件从Filename Queue中读入内存队列的操作不用手动执行，由tf自动完成；
- 调用sess.run 来启动数据出列和执行计算；
- 使用 coord.should_stop()来查询是否应该终止所有线程，当文件队列（queue）中的所有文件都已经读取出列的时候，会抛出一个 OutofRangeError 的异常，这时候就应该停止Session中的所有线程了；
- 使用coord.request_stop()来发出终止所有线程的命令，使用coord.join(threads)把线程加入主线程，等待threads结束。

作者：-牧野- 来源：CSDN 原文：<https://blog.csdn.net/dcrmg/article/details/79780331> 版权声明：本文为博主原创文章，转载请附上博文链接！

3.4.3 tf.decode_raw

将原来编码为字符串类型的变量重新变回来，这个方法在数据集dataset中很常用，因为制作图片源数据一般写进tfrecord里用to_bytes的形式，也就是字符串。

这里将原始数据取出来 必须制定原始数据的格式，原始数据是什么格式这里解析必须是什么格式，要不然会出现形状的不对应问题！

作者: han963xiao 来源: CSDN 原文: <https://blog.csdn.net/u012571510/article/details/82112452> 版权声明: 本文为博主原创文章, 转载请附上博文链接!

3.4.4 tf.strided_slice(input_, begin, end)

提取张量的一部分

- 一个维度一个维度地看:begin 加 stride,直到二者的和大于等于end
- [begin,end),左闭右开
- 清楚各个维度指的是哪部分
- 返回的张量中,元素的个数:end与begin对应元素做差再相乘,结果取绝对值

提取队列中reader后的文件中的label参数

```
result.label = tf.cast(
    tf.strided_slice(record_bytes, [0], [label_bytes]), tf.int32)
```

3.3.5 tf.transpose(x, perm=[1, 0])

定义两行三列的矩阵

```
x = [[1,3,5],
     [2,4,6]]
```

对于二维数组, perm=[0,1],0代表二维数组的行, 1代表二维数组的列

执行演示:

```
tf.transpose(x, perm=[1, 0])
```

```
>>> x = [[1,2,3],[4,5,6]]
>>> tf.transpose(x, perm=[1, 0])
<tf.Tensor 'transpose_9:0' shape=(3, 2) dtype=int32>
>>> tf.transpose(x, perm=[0, 1])
<tf.Tensor 'transpose_10:0' shape=(2, 3) dtype=int32>
>>>
```

3.3.6 tf.variable_scope

神经网络中上下文的变量管理工具, 可以将神经网络中的参数, 添加到一个组中, 在tensorboard中便于精简的显示。

3.4 数据增强

3.4.1 tf.random_crop 随机裁剪

随机地将张量裁剪为给定的大小. tf.random_crop (value, size, seed=None, name=None)

将size座位裁剪尺度的标准, 将输入value按照size大小尺寸随机进行裁剪, 返回和value相同秩, 和size相同大小的张量。

3.4.2 tf.image.random_flip_left_right

tf.image.random_flip_left_right(image, seed=None)

将输入的图片随机翻转。

- image: 输入待翻转的图片
- seed: 一个Python整数,用于创建一个随机种子

3.4.3 tf.image.random_brightness

tf.image.random_brightness(image, max_delta, seed=None)

基于随机种子，随机调整图像的亮度

在 $[-\text{max_delta}, \text{max_delta}]$ 的范围随机调整图片的亮度。

3.4.4 tf.image.random_contrast

`tf.image.random_contrast(image, lower, upper, seed=None)`

基于随机种子，来调整图像的对比度

3.4.5 tf.image.per_image_standardization

图像标准化处理。函数返回处理以后的图像，大小与通道数目与原图像保持一致。

3.4.6 tf.nn.conv2d()

```
tf.nn.conv2d(
    input,
    filter=None, # 填充方式
    strides=None, # 窗口滑动的步长
    padding=None, # 填充
    use_cudnn_on_gpu=True,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None,
    filters=None
)
```

定义卷积网络层

- **input**: 输入的要卷积的图片，要求为一个张量，shape为 $[\text{batch}, \text{in_height}, \text{in_width}, \text{in_channel}]$ ，其中batch为图片的数量，in_height 为图片高度，in_width 为图片宽度，in_channel 为图片的通道数，灰度图该值为1，彩色图为3。（也可以用其它值，但是具体含义不是很理解）
- **filter**: 卷积核，要求也是一个张量，shape为 $[\text{filter_height}, \text{filter_width}, \text{in_channel}, \text{out_channels}]$ ，其中 filter_height 为卷积核高度，filter_width 为卷积核宽度，in_channel 是图像通道数，和 input 的 in_channel 要保持一致，out_channel 是卷积核数量。
- **strides**: 卷积时在图像每一维的步长，这是一个一维的向量， $[1, \text{strides}, \text{strides}, 1]$ ，第一位和最后一位固定必须是1
- **padding**: string类型，值为“SAME”和“VALID”，表示的是卷积的形式，是否考虑边界。“SAME”是考虑边界，不足的时候用0去填充周围，“VALID”则不考虑
- **use_cudnn_on_gpu**: bool类型，是否使用cudnn加速，默认为true

作者：左理想fisher 来源：CSDN 原文：<https://blog.csdn.net/zuolixiangfisher/article/details/80528989> 版权声明：本文为博主原创文章，转载请附上博文链接！

3.4.7 tf.nn.max_pool()

池化操作

- 第一个参数value: 需要池化的输入，一般池化层接在卷积层后面，所以输入通常是feature map，依然是 $[\text{batch}, \text{height}, \text{width}, \text{channels}]$ 这样的shape。
- 第二个参数ksize: 池化窗口的大小，取一个四维向量，一般是 $[1, \text{height}, \text{width}, 1]$ ，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。
- 第三个参数strides: 和卷积类似，窗口在每一个维度上滑动的步长，一般也是 $[1, \text{stride}, \text{stride}, 1]$ 。
- 第四个参数padding: 和卷积类似，可以取'VALID' 或者'SAME'。

3.4.7 tf.nn.bias_add()

```
tf.nn.bias_add(
    value,
    bias,
    data_format=None,
    name=None
)
```

将bias添加到value.

3.4.8 [tf.nn.local_response_normalization](#)

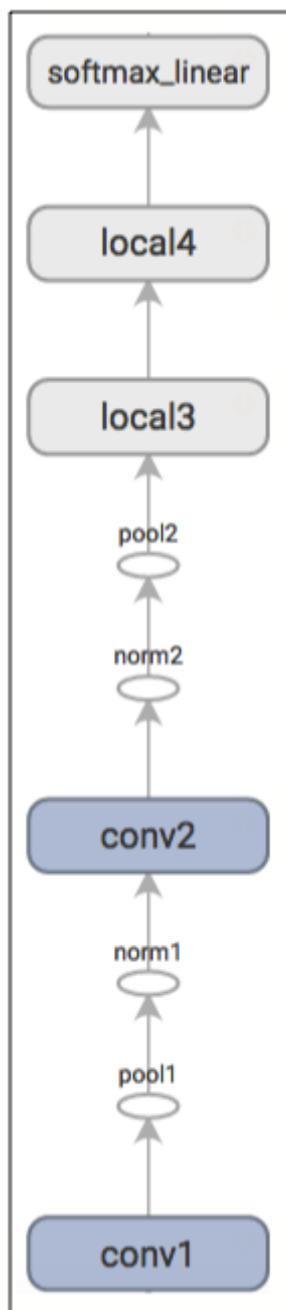
下的局部响应归一化函数(现在大多模型已经不用了)

局部响应归一化原理是仿造生物学上活跃的神经元对相邻神经元的抑制现象（侧抑制），然后根据论文有公式如下

$$b_{x,y}^i = \frac{a_{x,y}^i}{(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2)^\beta}$$

3.5 模型识别结构

2层卷积层，3层全连接层



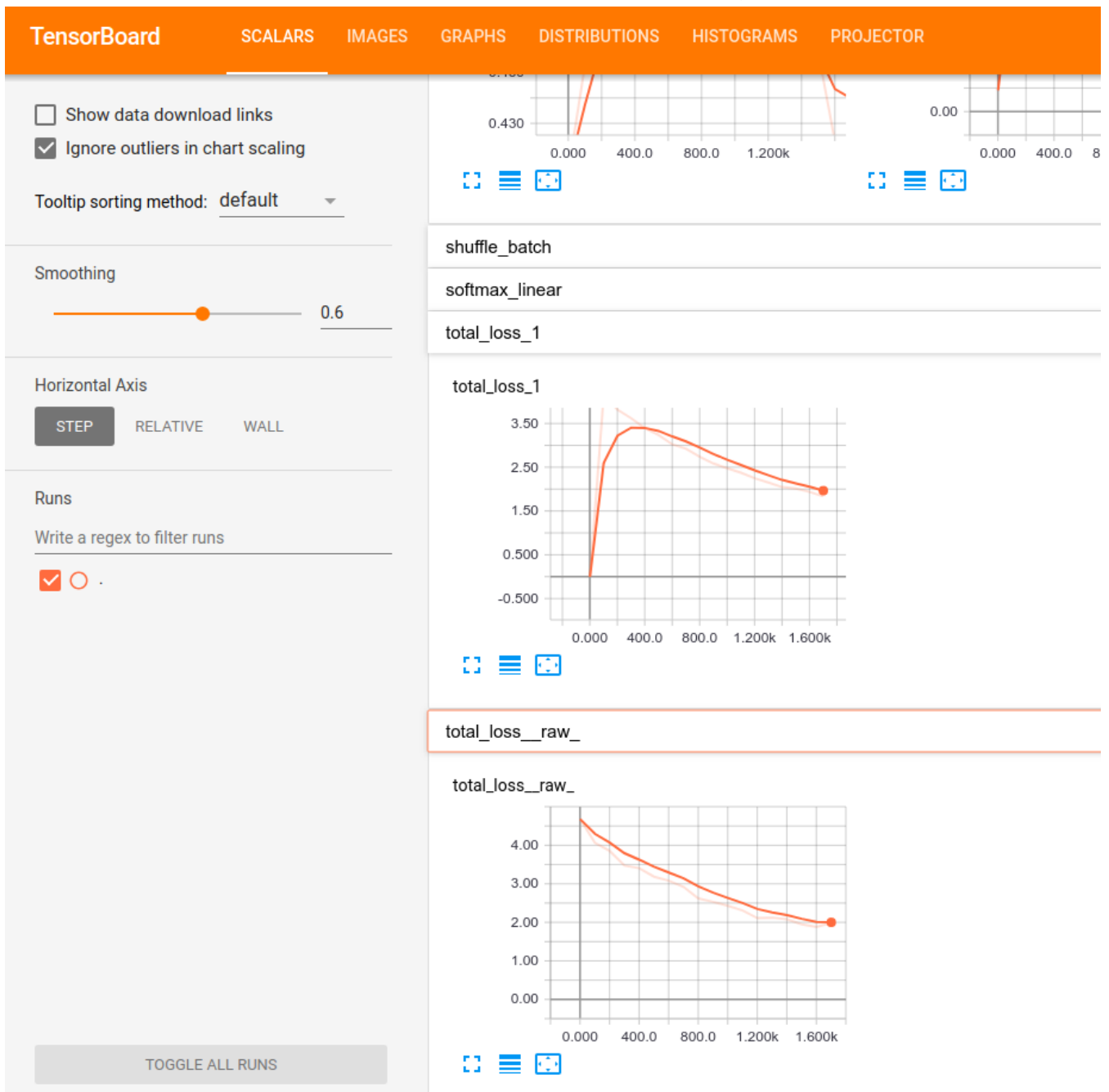
中间用到了数据增强

其他的相关算法

执行

```
python cifar10_train.py --train_dir cifar10_train/ --data_dir  
cifar10_data/
```

开始训练



训练loss值变化

```

2019-07-29 16:39:54.982557: step 1490, loss = 1.87 (222.0 examples/sec;
0.577 sec/batch)
2019-07-29 16:40:00.662475: step 1500, loss = 1.95 (225.4 examples/sec;
0.568 sec/batch)
2019-07-29 16:40:06.520239: step 1510, loss = 1.81 (218.5 examples/sec;
0.586 sec/batch)
2019-07-29 16:40:12.260342: step 1520, loss = 1.91 (223.0 examples/sec;
0.574 sec/batch)
2019-07-29 16:40:17.816579: step 1530, loss = 1.94 (230.4 examples/sec;
0.556 sec/batch)
2019-07-29 16:40:23.711302: step 1540, loss = 2.10 (217.1 examples/sec;
0.589 sec/batch)
2019-07-29 16:40:29.573939: step 1550, loss = 2.00 (218.3 examples/sec;
0.586 sec/batch)
2019-07-29 16:40:35.264799: step 1560, loss = 1.91 (224.9 examples/sec;

```

```

0.569 sec/batch)
2019-07-29 16:40:40.842676: step 1570, loss = 1.92 (229.5 examples/sec;
0.558 sec/batch)
2019-07-29 16:40:46.729267: step 1580, loss = 2.01 (217.4 examples/sec;
0.589 sec/batch)
2019-07-29 16:40:52.528980: step 1590, loss = 1.82 (220.7 examples/sec;
0.580 sec/batch)
2019-07-29 16:40:58.291411: step 1600, loss = 1.88 (222.1 examples/sec;
0.576 sec/batch)
2019-07-29 16:41:05.535584: step 1610, loss = 1.92 (176.7 examples/sec;
0.724 sec/batch)
2019-07-29 16:41:11.931400: step 1620, loss = 1.93 (200.1 examples/sec;
0.640 sec/batch)
2019-07-29 16:41:17.502566: step 1630, loss = 1.73 (229.8 examples/sec;
0.557 sec/batch)
2019-07-29 16:41:23.211044: step 1640, loss = 1.95 (224.2 examples/sec;
0.571 sec/batch)
2019-07-29 16:41:29.137543: step 1650, loss = 1.77 (216.0 examples/sec;
0.593 sec/batch)
2019-07-29 16:41:34.706557: step 1660, loss = 1.85 (229.8 examples/sec;
0.557 sec/batch)
2019-07-29 16:41:40.312529: step 1670, loss = 2.17 (228.3 examples/sec;
0.561 sec/batch)
2019-07-29 16:41:46.293831: step 1680, loss = 1.81 (214.0 examples/sec;
0.598 sec/batch)
2019-07-29 16:41:52.692196: step 1690, loss = 1.84 (200.1 examples/sec;
~ 0.6 sec/batch)

```

▼ 四、常用术语

深度学习中存在很多的常用术语，要知道别人在讲什么就得知道他们讲的术语都是什么意思。

4.1 Feature Map(特征映射)

在cnn的每个卷积层，数据都是以三维形式存在的。你可以把它看成许多个二维图片叠在一起（像豆腐皮竖直的贴成豆腐块一样），其中每一个称为一个feature map。

输入层：在输入层，如果是灰度图片，那就只有一个feature map；如果是彩色图片（RGB），一般就是3个feature map（红绿蓝）

[下图中三大部分依次是输入RGB图片,卷积核（也称过滤器），卷积结果（输出），*代表卷积操作，最左部分三片代表3个feature map;

如果是灰色图片（二维），则最左只有一片代表一个feature map，对应的卷积核（过滤器也是二维）]

 Feature map

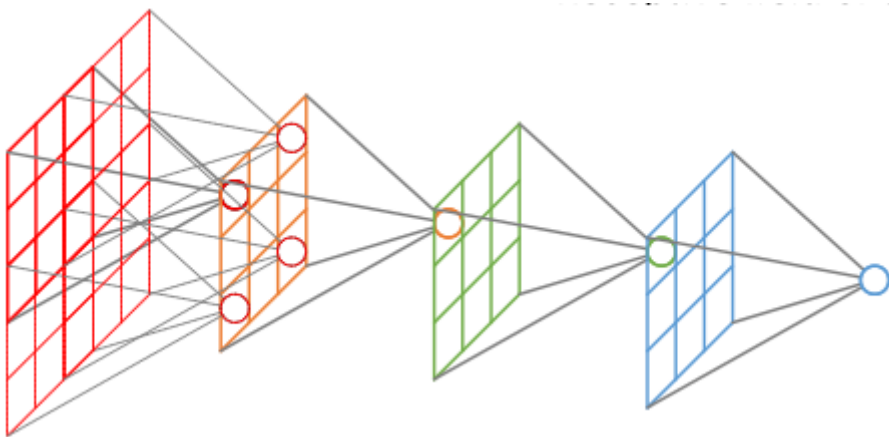
其它层：层与层之间会有若干个卷积核（kernel）（也称为过滤器），上一层每个feature map跟每个卷积核做卷积，都会产生下一层的一个feature map，有N个卷积核，下层就会产生N个feather map。

参考来源：<https://www.cnblogs.com/yh-blog/p/10052915.html>

4.2 卷积神经网络中感受野

4.2.1 感受野的定义

在卷积神经网络中，感受野（Receptive Field）的定义是卷积神经网络每一层输出的特征图（feature map）上的像素点在输入图片上映射的区域大小。再通俗点的解释是，特征图上的一个点对应输入图上的区域，如图1所示。



4.2.2 感受野大小的计算

- 最后一层（卷积层或池化层）输出特征图感受野的大小等于卷积核的大小。
- 第*i*层卷积层的感受野大小和第*i*层的卷积核大小和步长有关系，同时也与第（*i*+1）层感受野大小有关。
- 计算感受野的大小时忽略了图像边缘的影响，即不考虑padding的大小。

关于感受野大小的计算方式是采用从最后一层往下计算的方法，即先计算最深层在前一层上的感受野，然后逐层传递到第一层，使用的公式可以表示如下：

$$RF_i = (RF_{i+1} - 1) \times stride_i + Ksize_i$$

其中 RF_i 是第*i*层卷积的感受野。 RF_{i+1} 是第*i* + 1层上的感受野。stride 是卷积的步长， $Ksize$ 是本层卷积核的大小。

VGG16的感受野

我们从最后一层的池化层开始计算感受野：

Layer Type	Kernel Size	Stride
Input		
Conv1	3*3	1
Pool1	2*2	2
Conv2	3*3	1
Pool2	2*2	2
Conv3	3*3	1
Conv4	3*3	1
Pool3	2*2	2
Output	https://blog.csdn.net/program_developer	

pool3：RF=2（最后一层池化层输出特征图的感受野大小等于卷积核的大小）

conv4：RF=（2-1）*1+3=4。

conv3：RF=（4-1）*1+3=6。

pool2：RF=（6-1）*2+2=12。

conv2：RF=（12-1）*1+3=14。

pool1: RF= (14-1) *2+2=28。

conv1: RF= (28-1) *1+3=30。

因此, pool3输出的特征图在输入图片上的感受野为30*30。

作者: Microstrong0305

来源: CSDN

原文: https://blog.csdn.net/program_developer/article/details/80958716

版权声明: 本文为博主原创文章, 转载请附上博文链接!

4.3 CNN中的卷积核

4.3.1、什么是卷积:

图像中不同数据窗口的数据和卷积核(一个滤波矩阵)作内积的操作叫做卷积。其计算过程又称为滤波(filter), **本质是提取图像不同频段的特征。**

4.3.2、什么是卷积核:

也称为**滤波器filter**, 带着一组固定权重的神经元, 通常是nm二维的矩阵, **n和m也是神经元的感受野**。nm矩阵中存的是对感受野中数据处理的系数。一个卷积核的滤波可以用来提取特定的特征(例如可以提取物体轮廓、颜色深浅等)。

通过卷积层从原始数据中提取出新的特征的过程又成为feature map(特征映射)。filter_size是指filter的大小, 例如3*3; filter_num是指每种filter_size的filter个数, 通常是通道个数

4.3.3、什么是卷积层:

多个滤波器叠加便成了卷积层。


4.3.4、一个卷积层有多少个参数:

一个卷积核使用一套权值以便“扫视”数据每一处时以同样的方式抽取特征, 最终得到的是一种特征。在tensorflow定义conv2d时需要指定卷积核的尺寸, 本人现在的理解是一个卷积层的多个卷积核使用相同的mn, 只是权重不同。则一个卷积层的参数总共mn*filter_num个, 比全连接少了很多。

4.3.5、通道(channel) 怎么理解:

通道可以理解为视角、角度。例如同样是提取边界特征的卷积核, 可以按照R、G、B三种元素的角度提取边界, RGB在边界这个角度上有不同的表达; 再比如需要检查一个人的机器学习能力, 可以从特征工程、模型选择、参数调优等多个方面检测

4.3.6、卷积核的计算

 卷积核的计算过程

作者: cheney康 来源: CSDN 原文: <https://blog.csdn.net/cheneykl/article/details/79740810> 版权声明: 本文为博主原创文章, 转载请附上博文链接!

▼ 五、代码解读

CIFAR-10的模型结构由2层卷积神经网络和3层全链接组成

5.1 第一层卷积神经网络1

```
# conv1
with tf.variable_scope('conv1') as scope:
    # 定义卷积核
    kernel = _variable_with_weight_decay('weights', shape=[5, 5, 3, 64],
                                          stddev=1e-4, wd=0.0)

    conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
    biases = _variable_on_cpu('biases', [64],
                              tf.constant_initializer(0.0))
    bias = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(bias, name=scope.name)
    _activation_summary(conv1)

# pool1
pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2,
1],
                        padding='SAME', name='pool1')

# norm1
norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm1')
```

第一层卷积神经网络代码如下，

with tf.variable_scope('conv1') as scope:

变量创建层的对象类管理器（自我理解就是变量分组，在tensorboard中可以用一个框将分组内的变量隐藏起来，双击后又可以展示这些变量。）

kernel = _variable_with_weight_decay('weights', shape=[5, 5, 3, 64], stddev=1e-4, wd=0.0)

定义卷积核，大小为[5, 5, 3, 64]，并对卷积核进行初始化，初始化值为截断型的正太分布的初始化值。

_variable_with_weight_decay:

包含了L2正则化后的loss损失函数。tf.nn.l2_loss

conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')

strides: 卷积时在图像每一维的步长，这是一个一维的向量，[1, strides, strides, 1]，第一位和最后一位固定必须是1

而这里strides=[1, 1, 1, 1]表示的为滑动的步长为横向为1，纵向为1。

padding: string类型，值为“SAME”和“VALID”，表示的是卷积的形式，是否考虑边界。“SAME”是考虑边界，不足的时候用0去填充周围，“VALID”则不考虑

这里padding='SAME'表示图像边界处用全零填充。

这一步操作是将输入的图像样本数据，和卷积核进行卷积，得到不同的feature map,也可以理解为提取不同的图像映射的特征。

biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))

这个函数的内部代码如下

```

with tf.device('/cpu:0'):
    var = tf.get_variable(name, shape, initializer=initializer)
return var

```

由代码所示，表示对偏置biases的定义直接通过cpu来完成，并且初始化为0，大小为64个维度的大小。

```
bias = tf.nn.bias_add(conv, biases)
```

这一步是在卷积操作后将偏置添加上。

```
conv1 = tf.nn.relu(bias, name=scope.name)
```

激活函数，relu非线性函数（非饱和的激活函数），将卷积后的偏置求和结果进行非线性的转换。

```
_activation_summary(conv1)
```

这个函数就是将结果添加到tensorboard中的直方图表和散点图表中，方便可视化查看训练过程中的变化

```
tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
               padding='SAME', name='pool1')
```

- ksize=[1, 3, 3, 1],表示池化的窗口大小为3x3的窗。
- strides=[1, 2, 2, 1],表示滑动的步长为横向为2，纵向也为2大小。

填充方式也是全零填充

参考链接: <https://www.jianshu.com/p/80e46fa0040f>

```
norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm1')
```

这个函数是实现局部响应归一化函数的功能。

5.2 第二层卷积神经网络2

```

# conv2
with tf.variable_scope('conv2') as scope:
    kernel = _variable_with_weight_decay('weights', shape=[5, 5, 64,
64],
                                       stddev=1e-4, wd=0.0)
    conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = _variable_on_cpu('biases', [64],
tf.constant_initializer(0.1))
    bias = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(bias, name=scope.name)
    _activation_summary(conv2)

# norm2
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')

```



```
# pool2
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                        strides=[1, 2, 2, 1], padding='SAME',
                        name='pool2')
```

整个流程为：

- 1. 第一层的卷积计算结果作为输入和第二层的卷积核做运算
- 2. 将第二层的卷积核计算结果加上偏置
- 3. 将上一步的计算结果做非线性转换，通过非饱和的非线性函数relu
- 4. 将第三步的计算结果进行局部响应归一化操作
- 5. 将第四步的结果进行池化操作

5.3 第三层全连接1

```
# local3
with tf.variable_scope('local3') as scope:
    # Move everything into depth so we can perform a single matrix
    multiply.
    dim = 1
    for d in pool2.get_shape()[1:].as_list():
        dim *= d
    reshape = tf.reshape(pool2, [FLAGS.batch_size, dim])

    weights = _variable_with_weight_decay('weights', shape=[dim, 384],
                                           stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [384],
                              tf.constant_initializer(0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases,
                        name=scope.name)
    _activation_summary(local3)
```

第三层为全连接层，操作流程如下，首先将卷积层计算的feature map数据全部打平成一维的数据，然后进行全连接操作。

reshape = tf.reshape(pool2, [FLAGS.batch_size, dim])

数据维度变换函数，将卷积操作得到的pool2 转换成dim=1， FLAGS.batch_size为每回合输入的图片数量，进行维度转换。

值得注意的是，代码中的每次操作都是有用正则化将权值和偏置的误差加入到正则化处理中去。可以防止过拟合。

local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)

全连接操作，首先是线性计算部分，偏置和权值参与计算，然后经过激活函数relu实现非线性的转换。

5.4 第四层全连接2

```
# local4
with tf.variable_scope('local4') as scope:
    weights = _variable_with_weight_decay('weights', shape=[384, 192],
```

```

                                stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [192],
tf.constant_initializer(0.1))
    local4 = tf.nn.relu(tf.matmul(local3, weights) + biases,
name=scope.name)
    _activation_summary(local4)

```

5.5 第五层全连接3

```

# softmax, i.e. softmax(WX + b)
with tf.variable_scope('softmax_linear') as scope:
    weights = _variable_with_weight_decay('weights', [192, NUM_CLASSES],
                                stddev=1/192.0, wd=0.0)
    biases = _variable_on_cpu('biases', [NUM_CLASSES],
                                tf.constant_initializer(0.0))
    softmax_linear = tf.add(tf.matmul(local4, weights), biases,
name=scope.name)
    _activation_summary(softmax_linear)

```

注意的是，第五层全连接结果没有再使用激活函数relu，因为不需要非线性转换了，输出的结果就是10分类的可能结果概率。

▼ 5.6 模型的前向传播完整代码

```

def inference(images):
    """Build the CIFAR-10 model.

    Args:
        images: Images returned from distorted_inputs() or inputs().

    Returns:
        Logits.
    """
    # We instantiate all variables using tf.get_variable() instead of
    # tf.Variable() in order to share variables across multiple GPU training runs.
    # If we only ran this model on a single GPU, we could simplify this function
    # by replacing all instances of tf.get_variable() with tf.Variable().
    #
    # conv1
    with tf.variable_scope('conv1') as scope:
        kernel = _variable_with_weight_decay('weights', shape=[5, 5, 3, 64],
                                stddev=1e-4, wd=0.0)
        conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
        biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
        bias = tf.nn.bias_add(conv, biases)
        conv1 = tf.nn.relu(bias, name=scope.name)
        _activation_summary(conv1)

    # pool1
    pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                                padding='SAME', name='pool1')

    # norm1
    norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                                name='norm1')

    # conv2
    with tf.variable_scope('conv2') as scope:
        kernel = _variable_with_weight_decay('weights', shape=[5, 5, 64, 64],
                                stddev=1e-4, wd=0.0)
        conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
        biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.1))

```

```

bias = tf.nn.bias_add(conv, biases)
conv2 = tf.nn.relu(bias, name=scope.name)
_activation_summary(conv2)

# norm2
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')

# pool2
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                        strides=[1, 2, 2, 1], padding='SAME', name='pool2')

# local3
with tf.variable_scope('local3') as scope:
    # Move everything into depth so we can perform a single matrix multiply.
    dim = 1
    for d in pool2.get_shape()[1:].as_list():
        dim *= d
    reshape = tf.reshape(pool2, [FLAGS.batch_size, dim])

    weights = _variable_with_weight_decay('weights', shape=[dim, 384],
                                           stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [384], tf.constant_initializer(0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)
    _activation_summary(local3)

# local4
with tf.variable_scope('local4') as scope:
    weights = _variable_with_weight_decay('weights', shape=[384, 192],
                                           stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [192], tf.constant_initializer(0.1))
    local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name=scope.name)
    _activation_summary(local4)

# softmax, i.e. softmax(WX + b)
with tf.variable_scope('softmax_linear') as scope:
    weights = _variable_with_weight_decay('weights', [192, NUM_CLASSES],
                                           stddev=1/192.0, wd=0.0)
    biases = _variable_on_cpu('biases', [NUM_CLASSES],
                              tf.constant_initializer(0.0))
    softmax_linear = tf.add(tf.matmul(local4, weights), biases, name=scope.name)
    _activation_summary(softmax_linear)

return softmax_linear

```