1. **安装**

**1.1. 安装依赖项**

# Ubuntu/Linux 64-bit

$ sudo apt-get update

$ sudo apt-get install -y build-essential curl libcurl3-dev git libfreetype6-dev libpng12-dev libzmq3-dev pkg-config python-dev python-pip software-properties-common swig zip zlib1g-dev libcupti-dev

$ sudo pip install numpy wheel

# CentOS/Linux 64-bit

$ yum upgrade

$ yum install -y gcc gcc-c++ make openssl-devel kernel-devel kernel-doc kernel-headers curl git freetype-devel libpng-devel pkgconfig python-devel python-pip swig zip zlib-devel epel-release dkms

以下这步保证kernel-devel与内核版本保持一致

$ yum -y install "kernel-devel-uname-r == $(uname -r)"

**1.2. 安装Protocol Buffers (编译安装tensorflow/tensorflow serving时需要)**

# Ubuntu/Linux 64-bit

$ sudo apt-get install autoconf automake libtool curl make g++ unzip

# CentOS/Linux 64-bit

$ yum install autoconf automake libtool curl make gcc-c++ unzip

下载GitHub源码，编译安装

$ git clone https://github.com/google/protobuf.git

$ cd protobuf

$ ./autogen.sh

$ ./configure

$ make

$ make check

$ sudo make install

$ sudo ldconfig

**1.3. 安装Bazel (编译安装tensorflow/tensorflow serving时需要)**

需要版本: bazel 0.15.0

源安装 (ubuntu)

# Ubuntu/Linux 64-bit

$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel.list

$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -

$ sudo apt-get update && sudo apt-get install bazel

$ sudo apt-get upgrade bazel

# CentOS/Linux 64-bit

$ wget https://copr.fedorainfracloud.org/coprs/vbatts/bazel/repo/epel-7/vbatts-bazel-epel-7.repo

$ mv vbatts-bazel-epel-7.repo /etc/yum.repos.d/

$ yum install bazel

编译安装 (centos)

下载地址https://github.com/bazelbuild/bazel/releases

$ wget https://github.com/bazelbuild/bazel/releases/download/0.5.1/bazel-0.5.1-dist.zip

$ mkdir bazel-0.5.1

$ unzip bazel-0.5.1-dist.zip -d bazel-0.5.1/

$ cd bazel-0.5.1/

$ ./compile.sh

编译后将生成可执行文件/bazel-0.5.1/output/bazel

添加环境变量

$ sudo vim /etc/profile

添加如下内容

export BAZEL\_HOME=/home/fanyuguang/Project/bazel-0.5.1

export PATH=$PATH:$BAZEL\_HOME/output

执行 $ source /etc/profile 使环境变量立即生效

**1.4. 安装NVIDIA Driver (需要GPU支持)**

下载地址http://www.nvidia.com/Download/index.aspx

下载文件NVIDIA-Linux-x86\_64-390.42.run

禁用系统默认安装的Nouveau驱动

编辑/lib/modprobe.d/dist-blacklist.conf文件

$ sudo vim /lib/modprobe.d/dist-blacklist.conf

注释掉# blacklist nvidiafb，并添加blacklist nouveau和options nouveau modeset=0，如下

……

#blacklist nvidiafb

blacklist nouveau

options nouveau modeset=0

……

重建initramfs image

$ sudo mv /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r).img.bak

$ sudo dracut /boot/initramfs-$(uname -r).img $(uname -r)

重启系统

$ reboot

确认Nouveau驱动是否已经被禁用，若命令执行结果为空，说明已经被禁用

$ lsmod | grep nouveau

安装NVIDIA Driver

终端执行命令，从图形界面模式进入命令行模式

$ init 3

$ sudo ./NVIDIA-Linux-x86\_64-390.42.run

注意，安装过程中询问是否dkms支持时，选择yes，dkms可以帮维护内核外的驱动程序，在内核版本变动之后可以自动重新生成新的模块

从命令行模式切换回图形界面模式，并重启系统

$ init 5

$ reboot

**1.5. 安装NVIDIA CUDA Toolkit (需要GPU支持)**

# Ubuntu/Linux 64-bit

下载地址https://developer.nvidia.com/cuda-toolkit

下载文件cuda-repo-ubuntu1604-9-0-local\_9.0.176-1\_amd64.deb

$ sudo dpkg -i cuda-repo-ubuntu1604-9-0-local\_9.0.176-1\_amd64.deb

$ sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub

$ sudo apt-get update

$ sudo apt-get install cuda

# CentOS/Linux 64-bit

下载地址https://developer.nvidia.com/cuda-toolkit

下载文件cuda-repo-rhel7-8-0-local-8.0.44-1.x86\_64.rpm

$ sudo rpm -i cuda-repo-rhel7-9-0-local-9.0.176-1.x86\_64.rpm

$ sudo yum clean all

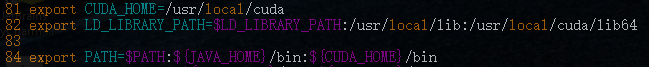
$ sudo yum install cuda

执行结束，CUDA将默认被安装到/usr/local/cuda目录

配置环境变量

$ sudo vim /etc/profile

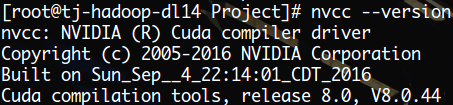
添加如下内容



使环境变量立即生效

$ source /etc/profile

执行 $ nvcc –version 查看CUDA版本



**1.6. 安装NVIDIA cuDNN (需要GPU支持)**

# CentOS/Linux 64-bit

下载地址https://developer.nvidia.com/cudnn

下载文件cudnn-9.0-linux-x64-v7.tgz

$ tar -xzvf cudnn-9.0-linux-x64-v7.tgz

$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include

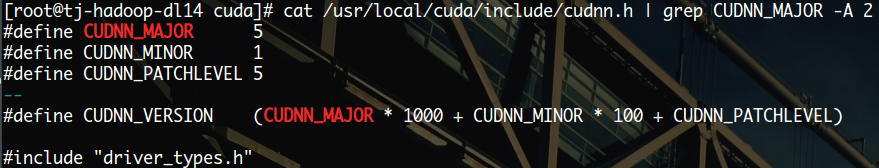
$ sudo cp cuda/lib64/libcudnn\* /usr/local/cuda/lib64

$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn\*

执行如下命令查看cuDNN版本

$ cat /usr/local/cuda/include/cudnn.h | grep CUDNN\_MAJOR -A 2

下图为显示，说明cnDNN版本为5.1.5



**1.7. PIP安装Tensorflow**

# Ubuntu/Linux 64-bit, CPU only, Python 2.7

$ sudo pip install tensorflow

# Ubuntu/Linux 64-bit, GPU enabled, Python 2.7

# Requires CUDA toolkit 8.0 and CuDNN v5.1.5

$ sudo pip install tensorflow-gpu

**1.8. 编译安装Tensorflow**

$ git clone https://github.com/tensorflow/tensorflow

$ cd tensorflow

$ ./configure

CPU编译

$ bazel build --config=opt //tensorflow/tools/pip\_package:build\_pip\_package

GPU编译

$ bazel build --config=opt --config=cuda //tensorflow/tools/pip\_package:build\_pip\_package

默认编译缓存文件会保存到/current\_user/.cache/\_bazel\_current\_user/目录下，若想更改目录设置output\_user\_root参数，如下

$ bazel --output\_user\_root=/home/fanyuguang/.cache/bazel/\_bazel\_fanyuguang build --config=opt --config=cuda //tensorflow/tools/pip\_package:build\_pip\_package

生成pip版Tensorflow安装文件.whl到/tmp/tensorflow\_pkg/目录下，并通过pip install安装

$ bazel-bin/tensorflow/tools/pip\_package/build\_pip\_package /tmp/tensorflow\_pkg

$ sudo pip install /tmp/tensorflow\_pkg/tensorflow-1.2.1-cp27-cp27mu-linux\_x86\_64.whl

**1.9. 若要Java中使用tensorflow，可Java Maven 引入依赖库**

<dependency>

<groupId>org.tensorflow</groupId>

<artifactId>tensorflow</artifactId>

<version>1.2.1</version>

</dependency>

**1.10. 安装gRPC (编译安装tensorflow serving时需要)**

$ sudo pip install grpcio

**1.11. PIP安装tensorflow serving**

**仅用于运行Python client，而不想通过Bazel编译安装**

$ sudo pip install tensorflow-serving-api

**1.12. apt-get安装tensorflow serving (仅Ubuntu)**

$ echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-serving-apt stable tensorflow-model-server tensorflow-model-server-universal" | sudo tee /etc/apt/sources.list.d/tensorflow-serving.list

$ curl https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-serving.release.pub.gpg | sudo apt-key add -

$ sudo apt-get update && sudo apt-get install tensorflow-model-server

更新

$ sudo apt-get upgrade tensorflow-model-server

**1.13. 编译安装tensorflow serving**

下载git代码，同时下载tensorflow, models两个git项目

$ git clone --recurse-submodules https://github.com/tensorflow/serving

进入serving/tensorflow目录，并配置tensorflow，配置方式同tensorflow安装

$ cd serving/tensorflow

$ ./configure

$ cd ..

CPU编译

$ bazel build -c opt tensorflow\_serving/...

GPU编译

$ bazel build -c opt --config=cuda //tensorflow\_serving/...

同样，默认编译缓存文件会保存到/current\_user/.cache/\_bazel\_current\_user/目录下，若想更改目录设置output\_user\_root参数

$ bazel --output\_user\_root=/home/fanyuguang/.cache/bazel/\_bazel\_fanyuguang build -c opt --config=cuda //tensorflow\_serving/...

通过如下命令启动Tensorflow Serving的gRPC Server服务，启动时需要传入参数，这里不做说明

$ bazel-bin/tensorflow\_serving/model\_servers/tensorflow\_model\_server

1. **导入包**

$ import tensorflow as tf

1. **简单框架及执行流程**

**3.1. 会话**

在TensorFlow中需先将各种运算操作（op）构造为图（graph），当使用会话（Session）运行（run）图或图中的某个op时，才会真正执行该op及该op依赖的op操作

图是由op堆叠组成的，用于描述计算过程，每个op的输入输出均为0个或多个tensorflow自有的tensor数据类型，图在Session中执行，Session对图中的每个op分发到cpu或gpu中执行，如果未指定，则tensorflow会自动检测gpu及cpu，并优先将计算分配到gpu上执行，如果机器上有超过一个可用gpu，除第一个外，其他gpu默认不参与计算，若想分配这些gpu，则需要显示指定

对于python接口tensor内部存储的数据是numpy的ndarray对象，在c++接口中，tensor是tensorflow::Tensor实例，目前python接口相较于c++接口提供了更多的辅助函数

下面是一个简单的用tensorflow求两个矩阵的乘积，第一个矩阵形状为(1, 2)，第二个矩阵形状为(2, 1)，将输出结果为[[5.]]

import tensorflow as tf

with tf.device('/cpu:0'):

matrix1 = tf.constant([[1., 2.]])

with tf.device('/gpu:0'):

matrix2 = tf.constant([[1.], [2.]])

with tf.device('/gpu:1')

product = tf.matmul(matrix1, matrix2)

with tf.Session() as sess:

result = sess.run(product)

print result

**3.2. 交互式会话**

使用tf.InteractiveSession()代替tf.Session()，使用Tensor.eval()可以方便的查看中间状态，其中Tensor指代计算图中的某个输出Tensor数据，Operation，使用交互式会话便于在IPython的交互式环境中使用

sess = tf.InteractiveSession()

matrix1 = tf.constant([[1., 2.]])

print matrix1.eval()

matrix2 = tf.constant([[1.], [2.]])

product = tf.matmul(matrix1, matrix2)

sess.run(product)

sess.close()

相关术语

learning\_rate学习率，一般为0.01

lr\_decay学习速率衰减

keep\_prob参数为0到1的小数，表示训练过程中以keep\_prob的比例保持某些神经网络单元，而其他网络单元暂时从网络中丢弃，用于防止过拟合

hidden\_size输入数据特征维数

batch\_size每批数据的规模

vocal\_size词典规模

1. **基本操作**

**4.1. 基础数据类型**

* tf.float16 : 16-bit half-precision floating-point.
* tf.float32 : 32-bit single-precision floating-point.
* tf.float64 : 64-bit double-precision floating-point.
* tf.bfloat16 : 16-bit truncated floating-point.
* tf.complex64 : 64-bit single-precision complex.
* tf.complex128 : 128-bit double-precision complex.
* tf.int8 : 8-bit signed integer.
* tf.uint8 : 8-bit unsigned integer.
* tf.uint16 : 16-bit unsigned integer.
* tf.int16 : 16-bit signed integer.
* tf.int32 : 32-bit signed integer.
* tf.int64 : 64-bit signed integer.
* tf.bool : Boolean.
* tf.string : String.
* tf.qint8 : Quantized 8-bit signed integer.
* tf.quint8 : Quantized 8-bit unsigned integer.
* tf.qint16 : Quantized 16-bit signed integer.
* tf.quint16 : Quantized 16-bit unsigned integer.
* tf.qint32 : Quantized 32-bit signed integer.
* tf.resource : Handle to a mutable resource.

**4.2. Constant常量**

* tf.constant(value, dtype=None, shape=None, name="Const", verify\_shape=False)

创建常量Tensor，输入可以为int, float 或python数组list，如果为list，并且value长度小于shape，则会以list最后元素填充shape

tensor = tf.constant([1, 2, 3, 4, 5, 6, 7]) => [1 2 3 4 5 6 7]

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

[1 2 3]]

tensor = tf.constant(-1.0, shape=[2, 3]) => [[-1. -1. -1.]

[-1. -1. -1.]]

tensor = tf.constant([1, 2, 3], shape=[3, 2]) ==> [[1, 2]

[3, 3]

[3, 3]]

value = tf.constant(1.0)

tensor = tf.constant([1, 2, 3, 4, 5, 6, 7], dtype=int32)

tensor = tf.constant(-1.0, shape=[2, 3])

**4.3. Tensor张量**

Tensor分为稀疏张量SparseTensor和密集张量Tensor

SparseTensor存储维数不相同的特征信息，如存储word特征时，对于长短不同的句子其特征信息的长度也不相同

Tensor存储维数相同的特征信息，如存储图片信息时，一般情况下，数据集内的图片会预处理成相同的宽高尺寸，图片存进Tensor将具有统一的维度信息

Tensor具有shape, indices, values三个属性

shape为一个列表，指明Tensor的维度，如[10]或[10, 10]分别表明Tensor是一个一维矩阵和二维矩阵

indices为一个列表，列表中元素指明Tensor中非0值的位置下标索引，如[[0], [1], [2], [3]]或[[0, 0], [0, 1], [0, 2], [1, 1], [1, 2]]

values为一个列表，列表中元素指明Tensor中非0值，该非0值下标索引与indices一一对应

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

生成尺寸为shape的张量，指定数据类型为dtype，并初始化全部元素为0

tf.zeros([3, 4], tf.int32) ==> [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

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

生成尺寸为shape的张量，指定数据类型为dtype，并初始化全部元素为1

tf.ones([2, 3], tf.int32) ==> [[1, 1, 1], [1, 1, 1]]

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

生成dtype，shape与传入tensor相同的张量，并初始化全部元素为0

# 'tensor' is [[1, 2, 3], [4, 5, 6]]

tf.zeros\_like(tensor) ==> [[0, 0, 0], [0, 0, 0]]

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

生成dtype，shape与传入tensor相同的张量，并初始化全部元素为1

# 'tensor' is [[1, 2, 3], [4, 5, 6]]

tf.ones\_like(tensor) ==> [[1, 1, 1], [1, 1, 1]]

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

生成shape为dims的tensor张量，并初始化全部元素为value

fill([2, 3], 9) ==> [[9, 9, 9]

[9, 9, 9]]

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

生成尺寸为shape的tensor张量，并用期望为mean，标准差为stddev的正态分布随机数进行填充，默认为标准正太分布

x = tf.random\_normal(shape=[1,5], mean=0.0, stddev=1.0)

==>[[-0.36128798 0.58550537 -0.88363433 -0.2677258 1.05080092]]

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

生成尺寸为shape的tensor张量，并用介于minval和maxval之间的值的均匀分布随机数进行填充

* tf.range(start, limit=None, delta=1, dtype=None, name="range")

生成一个等差列表tensor，其中值大于等于start，小于limit，等差为delta

tf.range(start=3, limit=18, delta=3) ==> [3, 6, 9, 12, 15]

tf.range(start=3, limit=1, delta=-0.5) ==> [3, 2.5, 2, 1.5]

tf.range(limit=5) ==> [0, 1, 2, 3, 4]

**4.4. Variable变量**

创建变量Tensor，并将张量作为初始值传入构造函数Variable()，张量可初始化为0或者随机值，创建时变量的shape移动与张量的shape保持一致

weights = tf.Variable(tf.random\_normal([784, 200], stddev=0.35), name="weights")

biases = tf.Variable(tf.zeros([200]), name="biases")

由一个现有变量初始化一个新变量，传入现有变量的initialized\_value()属性，其中0.2表示新变量的值为现有变量各值的2倍

weights2 = tf.Variable(weights.initialized\_value() \* 0.2, name=”weights2”)

执行如下命令初始化所有变量

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

with tf.Session() as sess:

sess.run(init\_op)

这里新版本为init\_op = tf.global\_variables\_initializer()

tf.get\_variable与tf.variable\_scope

变量保存在内存缓冲区当中

**4.5. 共享变量**

* tf.variable\_scope(name\_or\_scope, default\_name=None, values=None, initializer=None, regularizer=None, caching\_device=None, partitioner=None, custom\_getter=None, reuse=None, dtype=None)

为变量指定作用域空间，并以name\_or\_scope为该作用域空间命名，name\_or\_scope可以是作用域空间的名字，如’scope1’，也可以是作用域空间的对象，如scope1，当reuse参数默认为None，表示该作用域空间内的所有变量都将被创建，若reuse为True表示该作用域空间内的变量将复用已有的名为name\_or\_scope的作用域空间内的变量，并且为True时，作用域内不能创建新的变量，只能复用。initializer指定作用域空间内变量创建时的默认初始化器。

变量拥有name属性，表明该变量的作用域，为前缀加上变量名，如下面示例变量var1的name属性为scope1/scope1\_1/var1:0

也可在作用域空间内加入tf.get\_variable\_scope().reuse\_variables()，表明作用域空间内该语句后面的变量将复用已有的变量进行创建

当name\_or\_scope为名字时，若reuse=True，则要求变量的前缀与复用的变量前缀一致，如下例var3要与var1一致。为对象时，没有这个要求，并且作用域内的变量将按照复用的对象的作用域进行改写，如var4作用域由scope2/scope2\_1/var4:0改写为scope1/scope1\_1/var1:0

* tf.get\_variable(name, shape=None, dtype=None, initializer=None, regularizer=None, trainable=True, collections=None, caching\_device=None, partitioner=None, validate\_shape=True, custom\_getter=None)

通过name创建或者返回一个变量，name指定变量的名字

下面为示例，变量var2和变量var3的值将与var1相等，变量var4将重新创建

with tf.variable\_scope('scope1') as scope1:

with tf.variable\_scope('scope1\_1') as scope1\_1:

var1 = tf.get\_variable('var1', shape=[2, 3])

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

var2 = tf.get\_variable('var1', shape=[2, 3])

with tf.variable\_scope('scope1'):

with tf.variable\_scope('scope1\_1', reuse=True):

var3 = tf.get\_variable('var1', shape=[2, 3])

with tf.variable\_scope('scope2'):

with tf.variable\_scope('scope2\_1'):

with tf.variable\_scope(scope1\_1, reuse=True):

var4 = tf.get\_variable('var1', shape=[2, 3])

with tf.variable\_scope('scope2\_2'):

var5 = tf.get\_variable('var5', shape=[2, 3])

# var1.name is scope1/scope1\_1/var1:0

# var2.name is scope1/scope1\_1/var1:0

# var3.name is scope1/scope1\_1/var1:0

# var4.name is scope1/scope1\_1/var1:0

# var5.name is scope2/scope2\_2/var5:0

* tf.name\_scope(name, default\_name=None, values=None)

名称作用域，指定运算过程中算数运算操作（ops）的作用域

with tf.variable\_scope("foo"):

with tf.name\_scope("bar"):

v = tf.get\_variable("v", [1])

x = 1.0 + v

assert v.name == "foo/v:0"

assert x.op.name == "foo/bar/add"

**4.6. 数据填充**

* tf.placeholder(dtype, shape=None, name=None)

允许张量在运算时才填入数据，参数dtype指定填入的数据类型，参数shape指定填充张量的形状，如未指定则可填充任意形状的张量，参数name为该操作提供一个名字，函数将返回一个张量

input1 = tf.placeholder(dtype=tf.float32, shape=[2, 3])

input2 = tf.placeholder(dtype=tf.float32, shape=[3, 2])

output = tf.matmul(input1, input2)

with tf.Session() as sess:

feed\_dict = {input1:np.array([[1, 2, 3], [4, 5, 6]]), input2:np.array([[1, 2], [3, 4], [5, 6]])}

print sess.run(output, feed\_dict=feed\_dict)

**4.7. 算数运算**

* tf.abs(x, name=None)

求绝对值，输入可以是Tensor, SparseTensor, float32, float64, int32, int64，返回Tensor或SparseTensor

* tf.neg(x, name=None)

tensor中数值求反

# tensor 'x' is [[1 2 3]

# [4 5 6]]

tf.neg(x) ==> [[-1 -2 -3]

[-4 -5 -6]]

* tf.cast(x, dtype, name=None)

转换数据类型

# tensor a is [1.8, 2.2], dtype=tf.float

tf.cast(a, tf.int32) ==> [1, 2] # dtype=tf.int32

* tf.to\_float(x, name="ToFloat")

转x中每个元素为float32类型

* tf.round(x, name=None)

对x中每个元素取整，舍入最接近的整数

# 'a' is [0.9, 2.5, 2.3, 1.5, -4.5]

tf.round(a) ==> [ 1.0, 2.0, 2.0, 2.0, -4.0 ]

* tf.equal(x, y, name=None)

计算x, y中每个元素是否相等，结果为各元素判断结果的bool值，保证 x, y维度一致，返回bool类型的Tensor

* tf.add(x, y, name=None)

两个Tensor求和

* tf.sub(x, y, name=None)

两个Tensor求差，x - y

* tf.mul(x, y, name=None)

两个Tensor相乘，输入x, y必须数据类型一致

# tensor 'x' is [[1 2 3]

# [4 5 6]]

# tensor 'y' is [2]

tf.mul(x, y) ==> [[2 4 6]

[8 10 12]]

* tf.divide(x, y, name=None)

计算x/y，输入x，y均为tensor张量，应确保x, y的shape相同，或者

# tensor 'x' is [[1 2 3]

# [4 5 6]]

# tensor 'y1' is [1 2 4]

tf.divide(x, y1) ==> [[1. 1. 0.75]

[4. 2.5 1.5]]

# tensor 'y2' is [[1]

# [2]]

tf.divide(x, y2) ==> [[1. 2. 3.]

[2. 2.5 3.]]

# tensor 'y3' is [2]

tf.divide(x, y3) ==> [[0.5 1. 1.5]

[2. 2.5 3.]]

# tensor 'y4' is [[1 2 3]

# [4 5 6]]

tf.divide(x, y4) ==> [[1. 1. 1.]

[1. 1. 1.]]

* tf.square(x, name=None)

计算x中每个元素的平方

* tf.sqrt(x, name=None)

计算x中每个元素的开方

* tf.pow(x, y, name=None)

计算x的y次方，要确保x与y的shape一致

# tensor 'x' is [[2, 2], [3, 3]]

# tensor 'y' is [[8, 16], [2, 3]]

tf.pow(x, y) ==> [[256, 65536], [9, 27]]

**4.8. 张量运算**

* tf.matmul(a, b, transpose\_a=False, transpose\_b=False, adjoint\_a=False, adjoint\_b=False, a\_is\_sparse=False, b\_is\_sparse=False, name=None)

矩阵乘法

# 2-D tensor `a`

a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]) => [[1. 2. 3.]

[4. 5. 6.]]

# 2-D tensor `b`

b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2]) => [[7. 8.]

[9. 10.]

[11. 12.]]

c = tf.matmul(a, b) => [[58 64]

[139 154]]

* tf.one\_hot(indices, depth, on\_value=None, off\_value=None, axis=None, dtype=None, name=None)
* tf.reshape(tensor, shape, name=None)

转换tensor的shape为指定的shape，若shape=[-1]表示展开为一维的tensor，若shape=[\*, \*, -1, \*]，即-1出现在了中间，则会根据tensor的原尺寸计算出-1所在的维度的值，并代替-1完成下面运算

# tensor 't' is [[[1, 1], [2, 2]],

# [[3, 3], [4, 4]]]

# tensor 't' has shape [2, 2, 2]

reshape(t, [2, 4]) ==> [[1, 1, 2, 2],

[3, 3, 4, 4]]

# tensor 't' is [[[1, 1, 1],

# [2, 2, 2]],

# [[3, 3, 3],

# [4, 4, 4]],

# [[5, 5, 5],

# [6, 6, 6]]]

# tensor 't' has shape [3, 2, 3]

reshape(t, [2, 3]) ==>

reshape(t, [-1]) ==> [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6]

reshape(t, [2, -1]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],

[4, 4, 4, 5, 5, 5, 6, 6, 6]]

reshape(t, [-1, 9]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],

[4, 4, 4, 5, 5, 5, 6, 6, 6]]

reshape(t, [ 2, -1, 3]) ==> [[[1, 1, 1],

[2, 2, 2],

[3, 3, 3]],

[[4, 4, 4],

[5, 5, 5],

[6, 6, 6]]]

# tensor 't' is [7]

# shape `[]` reshapes to a scalar

reshape(t, []) ==> 7

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

将输入为value的tensor沿着第一维进行全部随机重排

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

* tf.convert\_to\_tensor(value, dtype=None, name=None, as\_ref=False, preferred\_dtype=None)

转换数据为Tensor类型，输入可以为Tensor类型，python数组list，numpy矩阵arrays

tf.convert\_to\_tensor(tf.constant([[1.0, 2.0], [3.0, 4.0]]), dtype=tf.float32)

tf.convert\_to\_tensor([[1.0, 2.0], [3.0, 4.0]], dtype=tf.float32)

tf.convert\_to\_tensor(np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32), dtype=tf.float32)

* tensor = tf.sparse\_to\_dense(sparse\_indices, output\_shape, sparse\_values, default\_value=0, validate\_indices=True, name=None)

words = tf.sparse\_to\_dense(sparse\_indices=words.indices[0:50], output\_shape=[50], sparse\_values=words.values[0:50], default\_value=0)

SparseTensor转Tensor, 其中output\_shape指定Tensor的维数，default\_value指定padding时填充值

* tf.add\_to\_collection(name, value)

将value保存为名字为name的共享集合中，供后面使用，collection中name键对应的值为一个list，value将加入到这个list中

* tf.get\_collection(name)

根据name键获取存储的值list

**4.9. 序列计算**

* tf.argmax(input, axis=None, name=None, dimension=None)

返回维度dimension上最大值的索引下标

# tensor is [[ 0.04261756 -0.34297419 -0.87816691 -0.15430689]

# [ 0.18663144 0.86972666 -0.06103253 0.38307118]

# [ 0.84588599 -0.45432305 -0.39736366 0.38526249]]

tf.argmax(input=a,dimension=0) ==> [2 1 1 2]

tf.argmax(input=a,dimension=1) ==> [0 1 0]

* tf.argmin(input, axis=None, name=None, dimension=None)

返回维度dimension上最小值的索引下标，同argmax

* tf.reduce\_sum(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

求和，输入x为Tensor，axis参数表示在哪些维度上进行求和，keep\_dims参数默认False则该维度求和后合并到上一维，若为True表示保持该维度，

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

[1, 1, 1]]

tf.reduce\_sum(x) ==> 6

tf.reduce\_sum(x, 0) ==> [2, 2, 2]

tf.reduce\_sum(x, 1) ==> [3, 3]

tf.reduce\_sum(x, 1, keep\_dims=True) ==> [[3], [3]]

tf.reduce\_sum(x, [0, 1]) ==> 6，第0维和第1维都进行求和，

* tf.reduce\_mean(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

求均值，类似tf.reduce\_sum

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

# [2., 2.]]

tf.reduce\_mean(x) ==> 1.5

tf.reduce\_mean(x, 0) ==> [1.5, 1.5]

tf.reduce\_mean(x, 1) ==> [1., 2.]

* tf.reduce\_prod(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

求输入input\_tensor中各元素的乘积，axis用于指定维度

# tensor 'x' is [[1 2 3]

# [4 5 6]]

tf.reduce\_prod(x) ==> 720

tf.reduce\_prod(x, 0) ==> [4 10 18]

tf.reduce\_prod(x, 0) ==> [6 120]

* tf.reduce\_min(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

求输入input\_tensor中最小值

* tf.reduce\_max(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

求输入input\_tensor中最大值

* tf.reduce\_all(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

对输入input\_tensor中各值求逻辑与

# 'x' is [[True, True]

# [False, False]]

tf.reduce\_all(x) ==> False

tf.reduce\_all(x, 0) ==> [False, False]

tf.reduce\_all(x, 1) ==> [True, False]

* tf.reduce\_any(input\_tensor, axis=None, keep\_dims=False, name=None, reduction\_indices=None)

对输入input\_tensor中各值求逻辑或

# 'x' is [[True, True]

# [False, False]]

tf.reduce\_any(x) ==> True

tf.reduce\_any(x, 0) ==> [True, True]

tf.reduce\_any(x, 1) ==> [True, False]

* tf.count\_nonzero(input\_tensor, axis=None, keep\_dims=False, dtype=dtypes.int64, name=None, reduction\_indices=None)

统计tensor中非0值的数量，axis用于指定维度

# 'x' is [[0, 1, 0]

# [1, 1, 0]]

tf.count\_nonzero(x) ==> 3

tf.count\_nonzero(x, 0) ==> [1, 2, 0]

tf.count\_nonzero(x, 1) ==> [1, 2]

tf.count\_nonzero(x, 1, keep\_dims=True) ==> [[1], [2]]

tf.rnal/local\_config\_cuda/cuda/include/host\_config.h:161:0,

from external/local\_config\_cuda/cuda/include/cuda\_runtime.h:76,

from <command-line>:0:

/usr/include/features.h:330:4: warning: #warning \_FORTIFY\_SOURCE requires compiling with optimization (-O) [-Wcpp]

# warning \_FORTIFY\_SOURCE requires compiling with optimization (-O)

^

In file included from external/local\_config\_cuda/cuda/include/host\_config.h:161:0,

from external/local\_config\_cuda/cuda/include/cuda\_runtime.h:76,

from <command-line>:0:

/usr/include/features.h:330:4: warning: #warning \_FORTIFY\_SOURCE requires compiling with optimization (-O) [-Wcpp]

# warning \_FORTIFY\_SOURCE requires compiling with optimization (-O)

^

nvcc warning : option '--relaxed-constexpr' has been deprecated and replaced by option '--expt-relaxed-constexpr'.

[5,703 / 7,555] Compiling external/org\_tensorflow/tensorflow/core/kernels/non\_max\_suppression\_op.cc

count\_nonzero(x, [0, 1]) ==> 3

* tf.nn.top\_k(input, k=1, sorted=True, name=None)

计算最大k个值的值和索引下标

1. **文件操作**

文件名filename

文件路径path

目录名dirname

查看文件或目录是否存在

$ tf.gfile.Exists(filename)

递归的删除某目录及其下面所有文件

$ tf.gfile.DeleteRecursively(dirname)

递归的创建目录

$ tf.gfile.MakeDirs(dirname)

1. **GPU使用**

tensorflow的调度策略是如果有多个gpu就调度到第一个gpu，如果没有gpu就调度cpu

对于普通运算gpu速度比cpu差很多倍

程序外部指定gpu，可以限制进程看到的gpu数

$ CUDA\_VISIBLE\_DEVICES='0,1' python train.py

$ CUDA\_VISIBLE\_DEVICES='0' nohup ./tensorflow\_model\_server --port=9000 --model\_name=inception --model\_base\_path=inception-export > tensorflow\_model\_server.txt 2>&1 &

通过tf.device灵活指定gpu

指定使用cpu0

with tf.device('/cpu:0'):

指定使用gpu0

with tf.device('/gpu:0'):

轮流使用gpu

for i in xrange(FLAGS.num\_gpus):

with tf.device('/gpu:%d' % i):

获取程序运行中设备指派情况，可将log\_device\_placement设置为True

$ sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))

若指派的cpu或gpu不存在系统会抛出InvalidArgumentError，可将allow\_soft\_placement设置为True，这样系统会自动分配一个存在并且支持的设备来运行

$ sess = tf.Session(config=tf.ConfigProto(allow\_soft\_placement=True, log\_device\_placement=True))

1. **准备数据**

将train及validation数据保存为TFRecords文件供训练时读取，其中feature, label均为int的数组，label编号为0, 1, 2...

tfrecords\_path = 'data/tfrecords/train.tfrecords'

writer = tf.python\_io.TFRecordWriter(tfrecords\_path)

for feature, label in datasets:

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

'feature': tf.train.Feature(int64\_list=tf.train.Int64List(value=feature)),

'label': tf.train.Feature(int64\_list=tf.train.Int64List(value=label)),

}))

writer.write(example.SerializeToString())

writer.close()

从TFRecords分批读取数据，num\_epochs 参数控制训练数据文件迭代的次数，

tfrecords\_path = 'data/tfrecords/train.tfrecords'

filename\_queue = tf.train.string\_input\_producer(tf.train.match\_filenames\_once(tfrecords\_path), num\_epochs=None)

reader = tf.TFRecordReader()

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

features = tf.parse\_single\_example(serialized\_example, features={

'feature': tf.VarLenFeature(tf.int64),

'label': tf.FixedLenFeature([], tf.int64),

})

feature = features['feature']

feature = tf.sparse\_to\_dense(sparse\_indices=feature.indices[0:50], output\_shape=[50],

sparse\_values=feature.values[0:50], default\_value=0)

label = features['label']

batch\_size = 1024

min\_after\_dequeue = 10000

capacity = min\_after\_dequeue + 3 \* batch\_size

num\_threads = 1

feature\_batch, label\_batch = tf.train.shuffle\_batch([feature, label], batch\_size=batch\_size, capacity=capacity, min\_after\_dequeue=min\_after\_dequeue, num\_threads=num\_threads)

feature\_batch, label\_batch即为训练数据集中的一部分数据，供传入model中使用

1. **CNN卷积神经网络**

**8.1. 卷积函数**

* tf.nn.conv2d(input, filter, strides, padding, use\_cudnn\_on\_gpu=None, data\_format=None, name=None)

在TensorFlow中，一幅图像可以用一个四维的Tensor来表示，这四个维度分别是[batch,heighht,width,channels]

input为输入图像的4维的tensor，其shape为[batch, in\_height, in\_width, in\_channels]，具体含义为[batch图片数量，图片高度，图片宽度，图像通道数]

filter指定滤波器tensor，为cnn中的卷积核，其shape为[filter\_height, filter\_width, in\_channels, out\_channels]，具体含义为[卷积核的高度，卷积核的宽度，图像通道数(与input的in\_channels一致)，卷积核个数]，filter一般为3\*3的大小[\*, 3, 3, \*]

strides指定卷积时卷积核在图像每一维的滑动步长，为一维的长度为4的list，4个维度对应输入input的[batch, in\_height, in\_width, in\_channels]，通常strides[0]=strides[3]=1,strides[1]=strides[2]=1

padding指定边界扩展，包括"SAME", "VALID"，"SAME"表示卷积前对图像进行填充，保证卷积后图像与卷积前的尺寸相同， "VALID"表示不对卷积前的图像进行填充，卷积后图像尺寸一般会变小

use\_cudnn\_on\_gpu指定是否使用cudnn加速，默认为true

data\_format指定输入输入的数据存储格式，可选"NHWC", "NCHW"，默认格式为"NHWC"，即[batch, in\_height, in\_width, in\_channels]

下例为输入10张5×5的图像，通道数为3,用3×3的7个卷积核进行卷积，卷积时，宽高两个维度的步长均为2，采用填充的padding方式，经过卷积将得到[10, 5, 5, 7]的输出

# input = tf.Variable(tf.random\_normal([10,5,5,3]))

# filter = tf.Variable(tf.random\_normal([3,3,3,7]))

# op = tf.nn.conv2d(input, filter, strides=[1, 2, 2, 1], padding='SAME')

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

对卷积加偏置，wx+b，conv2d的filter相当于w，bias\_add的bias相当于b，输入为value，bias的shape为[num\_filters\_out,]，num\_filters\_out与conv2d的输出即filter的卷积核个数out\_channels保持一致

**8.2. BatchNorm函数**

**8.3. Scale函数**

**8.4. 激活函数**

* tf.nn.relu(features, name=None)

relu激活函数，输入features

* tf.sigmoid(x, name=None)

计算x的sigmoid值，y = 1 / (1 + exp(-x))

* tf.tanh(x, name=None)

计算x的双曲正切

**8.5. 池化函数**

* tf.nn.avg\_pool(value, ksize, strides, padding, data\_format="NHWC", name=None)

平均池化，输入数据为value，ksize为pooling窗口在每一维上的尺寸，一般为[1, 2, 2, 1]，strides为pooling窗口在每一维上的滑动步长，一般为[1, 2, 2, 1]，padding指定边界扩展，包括"SAME", "VALID"，data\_format指定输入输入的数据存储格式

* tf.nn.max\_pool(value, ksize, strides, padding, data\_format="NHWC", name=None)

最大值池化，参数同avg\_pool

**8.6. Concat函数**

* tf.concat(concat\_dim, values, name="concat")

合并列表values中各Tensor的concat\_dim维

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

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

tf.concat(0, [t1, t2]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

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

# 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]

**8.7. dropout函数**

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

对数据进行dropout，丢弃一部分数据作为网络的输入，用于防止过拟合，输入为x，keep\_prob指定保持连接的比例，也是每个节点被保留的概率，

**8.8. 全连接函数**

**8.9. loss损失函数**

* tf.nn.softmax\_cross\_entropy\_with\_logits(logits, labels, dim=-1, name=None)

这个函数用来计算神经网络预测值与实际值之间的差异损失，logits和labels的维度一致，均为[batch\_size, num\_classes]

logits为网络最后一层的输入，会先进行Softmax计算，求取输出属于每个类别的概率，计算结果再与labels计算交叉熵，labels为训练数据输入时预先打的标签

返回结果为一个向量，要求交叉熵还需要进一步对向量所有元素求和tf.reduce\_sum()，如果求loss，需要对向量所有元素求均值tf.reduce\_mean()

* tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits, labels, name=None)

tf.nn.seq2seq.sequence\_loss\_by\_example(logits, targets, weights, average\_across\_timesteps=True, softmax\_loss\_function=None, name=None)

**8.10. softmax分类函数**

* tf.nn.softmax(logits, dim=-1, name=None)

用于网络最后一层后的分类层，进行分类

**8.11. 梯度计算函数**

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

对学习率进行指数衰退，decayed\_learning\_rate = learning\_rate \* decay\_rate ^ (global\_step / decay\_steps)

* tf.train.GradientDescentOptimizer(learning\_rate)

使用梯度下降算法优化

* tf.train.AdagradOptimizer(learning\_rate)

使用Adagrad算法优化

* tf.train.MomentumOptimizer(learning\_rate)

使用Momentum算法优化

* tf.train.AdamOptimizer(learning\_rate)

使用Adam算法优化

* tf.train.FtrlOptimizer(learning\_rate)

使用FTRL算法优化

* tf.train.RMSPropOptimizer(learning\_rate)

使用RMSProp算法优化

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

基于一定的学习率进行梯度优化训练，minimize优化更新训练的模型参数，并同时为global step计数

with tf.device('/cpu:0'):

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

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

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

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

* tf.trainable\_variables()

返回模型中所有可训练的参数

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

用来计算导数，参数ys和xs可以是一个tensor或tensor列表，函数返回一个长度为len(xs)的tensor列表，列表中每个tensor是ys中每个值对xs[i]求导之和。

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

对梯度进行修正，防止梯度爆炸问题，参数t\_list为待修正的张量，参数clip\_norm为修剪比例，函数返回修建后的张量list\_clipped和一个中间计算量global\_norm

应用tf.gradients和tf.clip\_by\_global\_norm进行梯度下降是比较底层的写法，optimizer.minimize内部用的正是optimizer.apply\_gradients这个方法

tvars = tf.trainable\_variables()

optimizer = tf.train.GradientDescentOptimizer(lr)

gradients = tf.gradients(loss, tvars)

也可同时对多个loss进行下降gradients = tf.gradients([loss1, loss2], tvars)

grads, \_ = tf.clip\_by\_global\_norm(gradients, max\_gradient\_norm)

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

1. **RNN循环神经网络**

**9.1. 词向量操作**

* tf.nn.embedding\_lookup(params, ids, partition\_strategy="mod", name=None, validate\_indices=True, max\_norm=None)

建立词向量矩阵，并为ids中每个词在矩阵中找到对应的向量表示，建立词向量矩阵用于刻画词与词之间的关系

构造一个尺寸为[vocab\_size, hidden\_size]的矩阵作为embeddings容器，表示有vocab\_size个向量，每个向量代表一个词，并且长度为hidden\_size，每个向量中分量的值在-1到1之间随机分布，之所以长度为hidden\_size是embedding结果将输入到rnn网络中，rnn单元的隐层个数为hidden\_size，为方便计算xw+b，故将长度设为这个值，embedding\_lookup将在矩阵中索引与inputs中每个词对应的词向量，词向量矩阵中的值将在训练过程中不断更新

with tf.device('/cpu:0'):

embedding = tf.get\_variable('embedding', [vocab\_size, hidden\_size], initializer=tf.random\_uniform\_initializer(), dtype=tf.float32)

inputs\_embedding = tf.nn.embedding\_lookup(embedding, inputs)

**9.2. 循环神经网络基本单元**

* tf.nn.rnn\_cell.BasicRNNCell(num\_units, input\_size=None, activation=tanh)

基本rnn单元

* tf.nn.rnn\_cell.BasicLSTMCell(num\_units, forget\_bias=1.0, input\_size=None, state\_is\_tuple=True, activation=tanh)

基本的长短期记忆网络，BasicLSTMCell为LSTMCell一个简单实现版，没有实现lstm的clipping，projection layer，peep-hole等

* tf.nn.rnn\_cell.LSTMCell(num\_units, input\_size=None, use\_peepholes=False, cell\_clip=None, initializer=None, num\_proj=None, proj\_clip=None, num\_unit\_shards=1, num\_proj\_shards=1, forget\_bias=1.0, state\_is\_tuple=True, activation=tanh)

长短期记忆网络

初始化lstm一个单元，参数num\_units为LSTM中行向量大小，即一个单元按时间序列展开为多少维，也是特征经过该单元后输出多少维，lstm中存在两个状态Ct和ht，若state\_is\_tuple参数为True这两个状态将分开记录，若state\_is\_tuple参数为False这两个状态将按列连接起来，False将马上被官方弃用，建议置为True

* tf.nn.rnn\_cell.MultiRNNCell(cells, state\_is\_tuple=True)
* tf.nn.rnn\_cell.GRUCell(num\_units, input\_size=None, activation=tanh)

门限循环网络

**9.3. rnn函数**

* tf.nn.rnn(cell, inputs, initial\_state=None, dtype=None, sequence\_length=None, scope=None)

rnn函数，参数cell为输入的rnn单元，如LSTMCell, MultiRNNCell等，参数inputs为输入数据，大小为num\_steps的list，list中每个元素为tensor，形状为(batch\_size, hidden\_size)，数据经过embedding后一般为(batch\_size, num\_steps, hidden\_size)的tensor，可通过inputs\_embedding = tf.unpack(inputs\_embedding, axis=1)转换为需要的形状，initial\_state指定cell的初始化状态，参数dtype指定初始化cell时的数据类型，如未提供initial\_state参数，则需指明dtype参数，参数sequence\_length指定batch中每个输入数据的长度，即num\_steps，sequence\_length的值为大小为batch\_size的list，每个元素为num\_steps，即sequence\_length=[num\_steps]\*batch\_size

* tf.nn.dynamic\_rnn(cell, inputs, sequence\_length=None, initial\_state=None, dtype=None, parallel\_iterations=None, swap\_memory=False, time\_major=False, scope=None)

dynamic\_rnn功能与rnn相同，但输入数据为一个tensor，当参数time\_major为False时，输入数据形状为(batch\_size, num\_steps, hidden\_size)，当参数time\_major为True时，输入数据形状为(num\_steps, batch\_size, hidden\_size)

函数返回(outputs, final\_state)的元组

* tf.nn.bidirectional\_rnn(cell\_fw, cell\_bw, inputs, initial\_state\_fw=None, initial\_state\_bw=None, dtype=None, sequence\_length=None, scope=None)

bidirectional\_rnn为双向rnn，参数cell\_fw与cell\_bw指定双向的两个rnn单元，参数inputs指定输入数据，同rnn，是长度为num\_steps的list，每个元素为一个tensor，形状为(batch\_size, hidden\_size)，参数initial\_state\_fw和initial\_state\_bw指定输入单元的初始化状态，dtype指定初始化器数据类型，参数sequence\_length同rnn指定batch中每个输入数据的长度

函数返回(outputs, forward\_final\_state, backward\_final\_state)的元组

* tf.nn.bidirectional\_dynamic\_rnn(cell\_fw, cell\_bw, inputs, sequence\_length=None, initial\_state\_fw=None, initial\_state\_bw=None, dtype=None, parallel\_iterations=None, swap\_memory=False, time\_major=False, scope=None)

bidirectional\_dynamic\_rnn功能与bidirectional\_rnn相同，但要求输入数据为一个tensor，当参数time\_major为False时，输入数据形状为(batch\_size, num\_steps, hidden\_size)，当参数time\_major为True时，输入数据形状为(num\_steps, batch\_size, hidden\_size)

函数返回(outputs, forward\_final\_state, backward\_final\_state)的元组

**9.4. dropout函数**

* tf.nn.rnn\_cell.DropoutWrapper(cell, input\_keep\_prob=1.0, output\_keep\_prob=1.0, seed=None)

对单元进行dropout，参数input\_keep\_prob控制数据输入该lstm\_cell单元时dropout掉一部分数据, output\_keep\_prob控制该lstm\_cell的输出作为下一层的输入时dropout掉一部分数据

**9.5. 训练参数操作**

1. **保存和加载模型参数**

* tf.train.Saver(var\_list=None, reshape=False, sharded=False, max\_to\_keep=5, keep\_checkpoint\_every\_n\_hours=10000.0, name=None, restore\_sequentially=False, saver\_def=None, builder=None, defer\_build=False, allow\_empty=False, write\_version=saver\_pb2.SaverDef.V2, pad\_step\_number=False)

创建Saver管理模型中所有变量

saver = tf.train.Saver()

Saver()中未传入任何参数表示saver处理graph中所有变量，若想保存及恢复模型的一部分变量，可以给Saver()传入python字典，字典中的键

保存模型参数，max\_to\_keep参数设置最多保存最近几个模型，若想保存全部模型设置max\_to\_keep=None，其中global\_step提供但前迭代次数，global\_step与save\_path（"/tmp/checkpoint\_dir/model.ckpt",）将会拼接在一起做为文件名进行模型参数保存

v1 = tf.Variable(..., name="v1")

v2 = tf.Variable(..., name="v2")

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

saver = tf.train.Saver()

with tf.Session() as sess:

sess.run(init\_op)

# Do some work with the model.

..

# Save the variables to disk.

save\_path = saver.save(sess, "/tmp/checkpoint\_dir/model.ckpt", global\_step=train\_step)

print "Model saved in file: ", save\_path

恢复变量，当从二进制文件中恢复变量时，不需要事先初始化tf.initialize\_all\_variables()

v1 = tf.Variable(..., name="v1")

v2 = tf.Variable(..., name="v2")

saver = tf.train.Saver()

with tf.Session() as sess:

# Restore variables from disk.

saver.restore(sess, "/tmp/checkpoint\_dir/model.ckpt")

print "Model restored."

# Do some work with the model

...

1. **记录并可视化训练状态**

在训练迭代过程中，将迭代的数据附加给tf.scalar\_summary，并分配一个标签名称，如下为记录损失率，准确率，学习率

$ tf.scalar\_summary('loss', loss)

$ tf.scalar\_summary('accuracy', accuracy)

$ tf.scalar\_summary('lr', lr)

汇总信息，并将结果写入到'/tmp/tensorboard\_logs'这个目录下

merged\_summary\_op = tf.merge\_all\_summaries()

summary\_writer = tf.train.SummaryWriter('/tmp/tensorboard\_logs', sess.graph)

total\_step = 0

while training:

total\_step += 1

session.run(training\_op)

if total\_step % 100 == 0:

summary\_str = session.run(merged\_summary\_op)

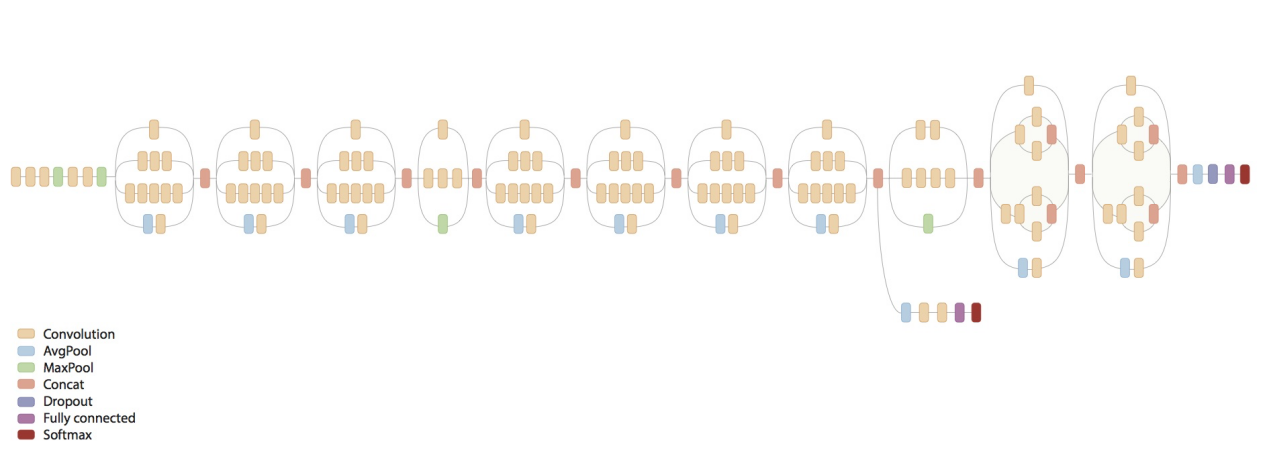
summary\_writer.add\_summary(summary\_str, total\_step)

启动TensorBoard查看训练过程中的数据变化，参数 logdir 指向 SummaryWriter 序列化数据的存储路径

$ tensorboard --logdir=/tmp/tensorboard\_logs

在浏览器中输入localhost:6006查看TensorBoard

1. **Inception详解**



网络模型

**12.1. slim/scopes.py**

若一个代码块内多次调用的某一函数，如ops.conv2d()，则将其声明在scopes.arg\_scope内，并设定ops.conv2d()的默认部分参数，在代码块内调用ops.conv2d()并传入参数时，参数列表即会补充arg\_scope参数列表内的参数，如果调用时传入参数与arg\_scope参数列表中存在重复，传入参数会覆盖arg\_scope参数。

下面为示例：

with scopes.arg\_scope(ops.conv2d, padding='SAME', stddev=0.01, weight\_decay=0.0005):

net = ops.conv2d(inputs, 64, [11, 11], 4, padding='VALID', scope='conv1')

net = ops.conv2d(net, 256, [5, 5], scope='conv2')

实际执行为：

ops.conv2d(inputs, 64, [11, 11], 4, padding='VALID', stddev=0.01, weight\_decay=0.0005, scope='conv1')

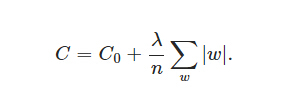
与

ops.conv2d(inputs, 256, [5, 5], padding='SAME', stddev=0.01, weight\_decay=0.0005, scope='conv2')

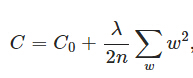
**12.2. slim/losses.py**

正则函数regularizer，有利于防止过拟合，提高泛化能力

l1\_regularizer，l1正则化，在原始的代价函数后面加上，所有权重w的绝对值的和乘以λ/n



l2\_regularizer，L2正则化就是在代价函数后面再加上一个正则化项：



C0代表原始的代价函数，后面那一项就是L2正则化项，它是这样来的：所有参数w的平方的和，除以训练集的样本大小n。λ就是正则项系数，权衡正则项与C0项的比重。另外还有一个系数1/2，1/2经常会看到，主要是为了后面求导的结果方便，后面那一项求导会产生一个2，与1/2相乘刚好凑整。

实际上tensorflow已有l2正则化函数tf.nn.l2\_loss

l1\_l2\_regularizer，l1l2正则函数

同时添加l1正则项和l2正则项

l1损失函数和l2损失函数同l1,l2正则化，并将损失值加入共享集合中

cross\_entropy\_loss，交叉熵函数，调用函数tf.nn.softmax\_cross\_entropy\_with\_logits(logits, labels, dim=-1, name=None)

**12.3. slim/variables.py**

定义了两个collections

MODEL\_VARIABLES保存所有变量

VARIABLES\_TO\_RESTORE保存fine-tuning时需要恢复的变量集合

**12.4. slim/ops.py**

定义了网络的卷积层conv2d，池化层max\_pool，avg\_pool，dropout，全连接层fc，归一化层batch\_norm，one\_hot\_encoding

max\_pool降维时取最大值，avg\_pool降维时取平均值

**12.5. slim/inception\_model.py**

inception\_v3函数定义整个网络结构

返回值logits和end\_points，logits为网络最后一层输出，end\_points为字典类型保存网络中间各层的结果，包括logits和最后softmax后的分类结果

concat层用于将几个并行的网络结果进行合并

**12.6. slim/slim.py**

引入slim文件夹下所有py文件

**12.7. inception\_model.py**

定义inference，loss，\_activation\_summaries三个函数

inference调用slim/inception\_model.py的inception\_v3函数，返回网络最后一层的logits及网络中一个辅助的auxiliary\_logits

**12.8. inception\_train.py**

用于网络的训练

**12.9. imagenet\_train.py**

调用inception\_train.py的train方法进行训练

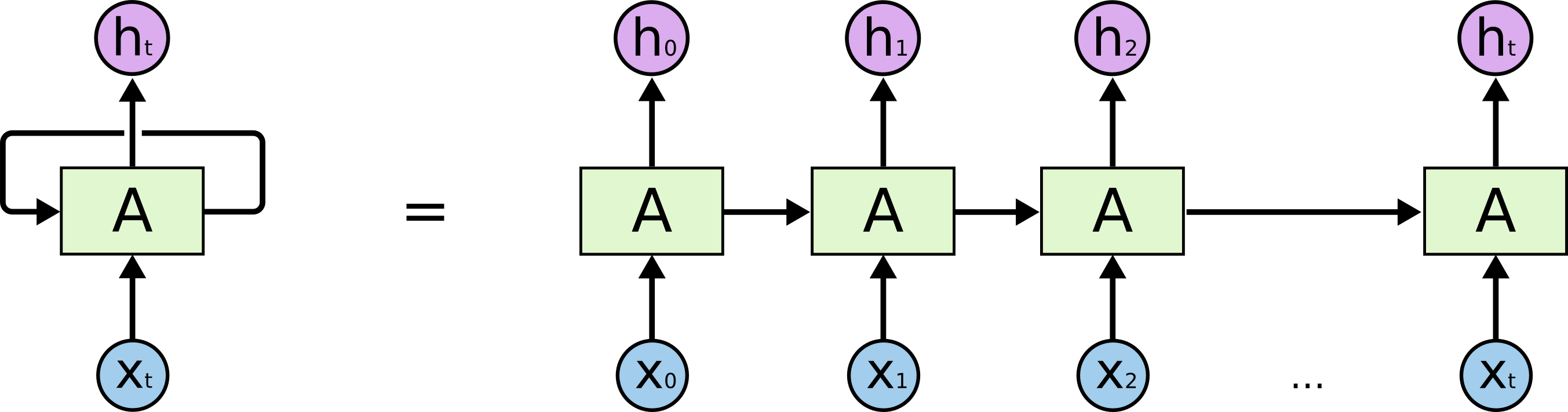
1. **LSTM详解**

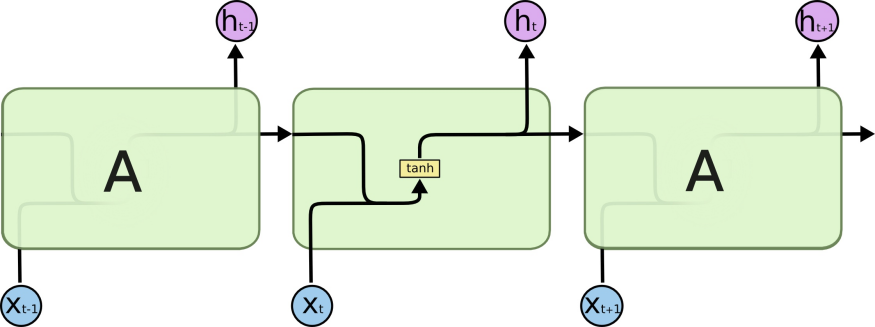
lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units=hidden\_size, forget\_bias=0.0, state\_is\_tuple=True)

lstm\_cell = tf.nn.rnn\_cell.LSTMCell(num\_units=hidden\_size, forget\_bias=0.0, state\_is\_tuple=True)

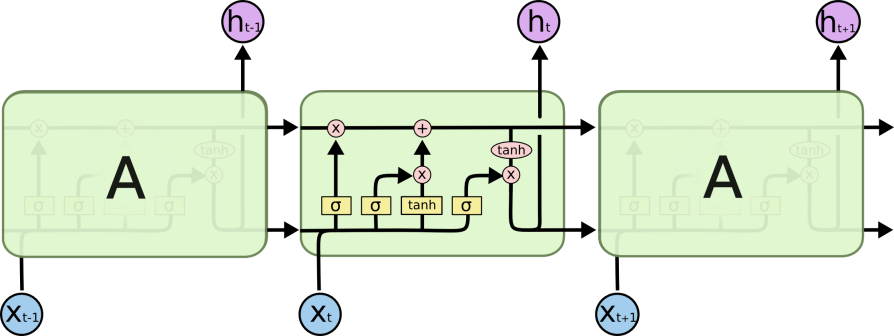
初始化lstm一个单元，参数num\_units为LSTM中行向量大小，即一个单元按时间序列展开为多少维，也是特征经过该单元后输出多少维，如下图的t，forget\_bias，lstm中存在两个状态Ct和ht，若state\_is\_tuple参数为True这两个状态将分开记录，若state\_is\_tuple参数为False这两个状态将按列连接起来，False将马上被官方弃用，建议置为True

BasicLSTMCell为LSTMCell一个简单实现版，没有实现lstm的clipping，projection layer，peep-hole等。





上面两幅图为rnn单元模型，输入为xt，输出为ht，tanh使得输出值在-1到1之间



上图为lstm单元模型，输入为xt，输出为ht，其中单元展开上面的连线为元胞状态(Cell State)，下面的连线为隐状态(Hidden State)，c和h记录训练过程中的中间状态，三个Sigmoid单元分别为遗忘门（Forget Gate），输入门（Input Gate），输出门（Output Gate），使得输出在0到1之间，用于控制信息，0表示完全丢弃该信息，1表示完全保留该信息。

lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(lstm\_cell, output\_keep\_prob=keep\_prob)

对单元进行dropout，参数input\_keep\_prob控制数据输入该lstm\_cell单元时dropout掉一部分数据, output\_keep\_prob控制该lstm\_cell的输出作为下一层的输入时dropout掉一部分数据

cell = tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell] \* num\_layers, state\_is\_tuple=True)

将lstm单元堆叠起来，第一个参数中的num\_layers表示堆叠几层

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

初始化网络状态为全零

with tf.device('/cpu:0'):

embedding = tf.get\_variable('embedding', [vocab\_size, hidden\_size], initializer=tf.random\_uniform\_initializer(), dtype=tf.float32)

inputs\_embedding = tf.nn.embedding\_lookup(embedding, inputs)

输入数据inputs为一个二维矩阵，维度为(batch\_size, num\_steps)，num\_steps表示语句序列的长度，也就是sequence\_length，每个元素对应词典vocabulary中的index，embedding对输入数据进行处理，将每个词表示为一个长度为hidden\_size的dense vector，建立一个[vocab\_size, hidden\_size]维的词表，该dense vector在一定程度上反映出word与word之间的关系，hidden\_size要与初始化LSTMCell时的num\_units保持一致，embedding后，inputs\_embedding的维度为(batch\_size, num\_steps, hidden\_size)

目前tensorflow中embedding操作只能在cpu中进行

if is\_training and keep\_prob < 1:

inputs\_embedding = tf.nn.dropout(inputs\_embedding, keep\_prob)

对输入数据进行dropout，防止过拟合，inputs\_embedding将作为MultiRNNCell的输入

outputs = []

state = initial\_state

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

for time\_step in range(num\_steps):

if time\_step > 0: tf.get\_variable\_scope().reuse\_variables()

# cell\_output shape: [batch\_size, hidden\_size]

(cell\_output, state) = cell(inputs\_embedding[:, time\_step, :], state)

outputs.append(cell\_output)

将数据输入cell进行训练，inputs\_embedding的维度为(batch\_size, num\_steps, hidden\_size)，共迭代num\_steps次，cell(inputs\_embedding[:, time\_step, :]表示从前向后每次输入语句中的一个单词，即每次输入数据的维度为(batch\_size, 1, hidden\_size)