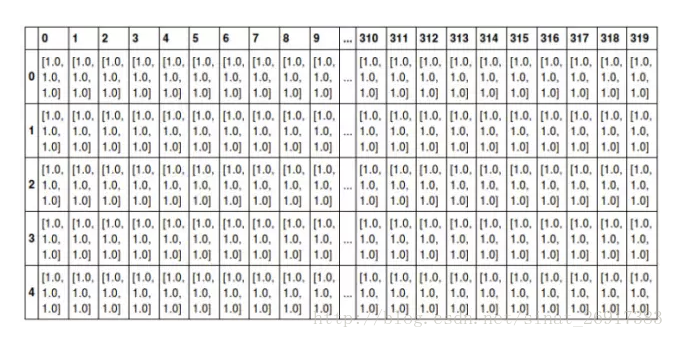
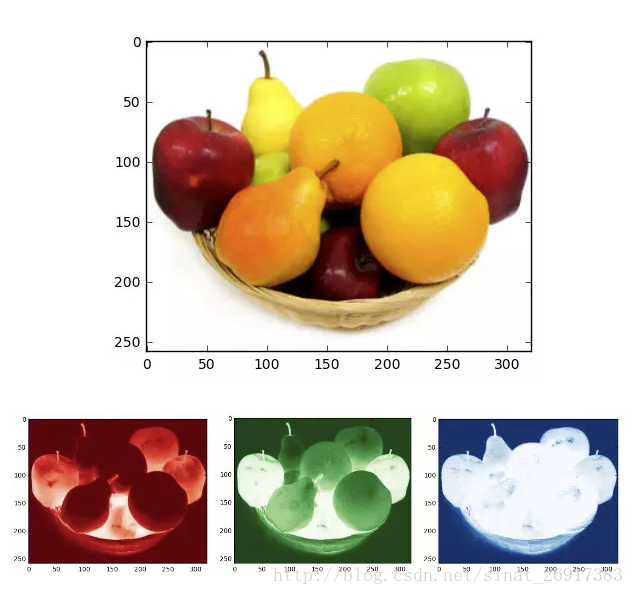
# tensorflow入门

## 1.深度学习框架的核心部件

### 1.1 张量

张量是所有深度学习框架中最核心的组件，因为后续的所有运算和优化算法都是基于张量进行的。几何代数中定义的张量是基于向量和矩阵的推广，通俗一点理解的话，我们可以将标量视为零阶张量，矢量视为一阶张量，那么矩阵就是二阶张量。

举例来说，我们可以将任意一张RGB彩色图片表示成一个三阶张量（三个维度分别是图片的高度、宽度和色彩数据）。如下图所示是一张普通的水果图片，按照RGB三原色表示，其可以拆分为三张红色、绿色和蓝色的灰度图片，如果将这种表示方法用张量的形式写出来，就是图中最下方的那张表格。



图中只显示了前5行、320列的数据，每个方格代表一个像素点，其中的数据[1.0, 1.0, 1.0]即为颜色。假设用[1.0, 0, 0]表示红色，[0, 1.0, 0]表示绿色，[0, 0, 1.0]表示蓝色，那么如图所示，前面5行的数据则全是白色。

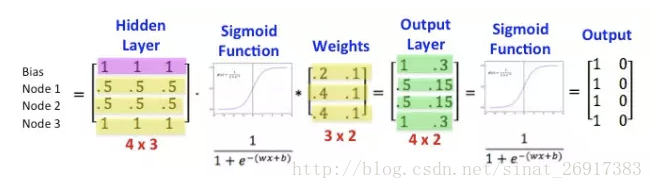
将这一定义进行扩展，我们也可以用四阶张量表示一个包含多张图片的数据集，其中的四个维度分别是：图片在数据集中的编号，图片高度、宽度，以及色彩数据。

**为什么需要使用张量来进行图片处理？？**

当数据处理完成后，我们还可以方便地将张量再转换回想要的格式。例如Python NumPy包中numpy.imread和numpy.imsave两个方法，分别用来将图片转换成张量对象（即代码中的Tensor对象），和将张量再转换成图片保存起来。

### 1.2 基于张量的操作

有了张量对象之后，下面一步就是一系列针对这一对象的数学运算和处理过程。



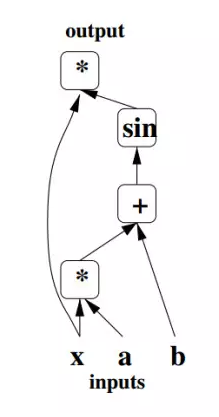
其实，所谓的“学习”就是不断纠正神经网络的实际输出结果和预期结果之间误差的过程。这里的一系列操作包含的范围很宽，可以是简单的矩阵乘法，也可以是卷积、池化和LSTM等稍复杂的运算。

### 1.3 计算图

有了张量和基于张量的各种操作之后，下一步就是将各种操作整合起来，输出我们需要的结果。

但不幸的是，随着操作种类和数量的增多，有可能引发各种意想不到的问题，包括多个操作之间应该并行还是顺次执行，如何协同各种不同的底层设备，以及如何避免各种类型的冗余操作等等。这些问题有可能拉低整个深度学习网络的运行效率或者引入不必要的Bug，而计算图正是为解决这一问题产生的。

据AI科技评论了解，计算图首次被引入人工智能领域是在2009年的论文《Learning Deep Architectures for AI》。当时的图片如下所示，作者用不同的占位符（\*，+，sin）构成操作结点，以字母x、a、b构成变量结点，再以有向线段将这些结点连接起来，组成一个表征运算逻辑关系的清晰明了的“图”型数据结构，这就是最初的计算图。



后来随着技术的不断演进，加上脚本语言和低级语言各自不同的特点（概括地说，脚本语言建模方便但执行缓慢，低级语言则正好相反），因此业界逐渐形成了这样的一种开发框架：前端用Python等脚本语言建模，后端用C++等低级语言执行（这里低级是就应用层而言），以此综合了两者的优点。可以看到，这种开发框架大大降低了传统框架做跨设备计算时的代码耦合度，也避免了每次后端变动都需要修改前端的维护开销。而这里，在前端和后端之间起到关键耦合作用的就是计算图。

需要注意的是，通常情况下开发者不会将用于中间表示得到的计算图直接用于模型构造，因为这样的计算图通常包含了大量的冗余求解目标，也没有提取共享变量，因而通常都会经过依赖性剪枝、符号融合、内存共享等方法对计算图进行优化。

目前，各个框架对于计算图的实现机制和侧重点各不相同。例如Theano和MXNet都是以隐式处理的方式在编译中由表达式向计算图过渡。而Caffe则比较直接，可以创建一个Graph对象，然后以类似Graph.Operator(xxx)的方式显示调用。

**因为计算图的引入，开发者得以从宏观上俯瞰整个神经网络的内部结构，就好像编译器可以从整个代码的角度决定如何分配寄存器那样，计算图也可以从宏观上决定代码运行时的GPU内存分配，以及分布式环境中不同底层设备间的相互协作方式。**

### 1.4 自动微分工具

计算图带来的另一个好处是让模型训练阶段的梯度计算变得模块化且更为便捷，也就是自动微分法。

将待处理数据转换为张量，针对张量施加各种需要的操作，通过自动微分对模型展开训练，然后得到输出结果开始测试。那么如何微分中提高效率呢？

**第一种方法：模拟传统的编译器**

每一种张量操作的实现代码都会预先加入C语言的转换部分，然后由编译器在编译阶段将这些由C语言实现的张量操作综合在一起。目前pyCUDA和Cython等编译器都已经实现了这一功能。

**第二种方法：利用脚本语言实现前端建模**

用低级语言如C++实现后端运行，这意味着高级语言和低级语言之间的交互都发生在框架内部，因此每次的后端变动都不需要修改前端，也不需要完整编译（只需要通过修改编译参数进行部分编译），因此整体速度也就更快。

**第三种方法：现成的扩展包**

例如最初用Fortran实现的BLAS（基础线性代数子程序），就是一个非常优秀的基本矩阵（张量）运算库，此外还有英特尔的MKL（Math Kernel Library）等，开发者可以根据个人喜好灵活选择。

一般的BLAS库只是针对普通的CPU场景进行了优化，但目前大部分的深度学习模型都已经开始采用并行GPU的运算模式，因此利用诸如NVIDIA推出的针对GPU优化的cuBLAS和cuDNN等更据针对性的库可能是更好的选择。

### 1.5 BLAS cuBLAS cuDNN等拓展包

现在，通过上述所有模块，我们已经可以搭建一个全功能的深度学习框架：将待处理数据转换为张量，针对张量施加各种需要的操作，通过自动微分对模型展开训练，然后得到输出结果开始测试。这时还缺什么呢？答案是**运算效率**。

由于此前的大部分实现都是基于高级语言的（如Java、Python、Lua等），而即使是执行最简单的操作，高级语言也会比低级语言消耗更多的CPU周期，更何况是结构复杂的深度神经网络，因此运算缓慢就成了高级语言的一个天然的缺陷。

目前针对这一问题有两种解决方案。

**第一种方法是模拟传统的编译器。**就好像传统编译器会把高级语言编译成特定平台的汇编语言实现高效运行一样，这种方法将高级语言转换为C语言，然后在C语言基础上编译、执行。为了实现这种转换，每一种张量操作的实现代码都会预先加入C语言的转换部分，然后由编译器在编译阶段将这些由C语言实现的张量操作综合在一起。目前pyCUDA和Cython等编译器都已经实现了这一功能。

**第二种方法就是前文提到的，利用脚本语言实现前端建模**，**用低级语言如C++实现后端运行**，这意味着高级语言和低级语言之间的交互都发生在框架内部，因此每次的后端变动都不需要修改前端，也不需要完整编译（只需要通过修改编译参数进行部分编译），因此整体速度也就更快。

除此之外，由于低级语言的最优化编程难度很高，而且大部分的基础操作其实也都有公开的最优解决方案，因此另一个显著的加速手段就是利用现成的扩展包。例如最初用Fortran实现的BLAS（基础线性代数子程序），就是一个非常优秀的基本矩阵（张量）运算库，此外还有英特尔的MKL（Math Kernel Library）等，开发者可以根据个人喜好灵活选择。

值得一提的是，一般的BLAS库只是针对普通的CPU场景进行了优化，但目前大部分的深度学习模型都已经开始采用并行GPU的运算模式，因此利用诸如NVIDIA推出的针对GPU优化的cuBLAS和cuDNN等更据针对性的库可能是更好的选择。

## 2. tensorflow入门教程

### 2.1 整体介绍

tensorFlow是一种将计算表示为图的编程系统。图中的节点称为ops(operation的简称)。一个ops使用0个或以上的Tensors，通过执行某些运算，产生0个或以上的Tensors。一个Tensor是一个多维数组。

tensorFlow中的图描述了计算过程，图通过Session的运行而执行计算。Session将图的节点们(即ops)放置到计算设备(如CPUs和GPUs)上，然后通过方法执行它们；这些方法执行完成后，将返回tensors。在Python中的tensor的形式是numpy ndarray对象，而在C/C++中则是tensorflow::Tensor。

tensorflow的一些特点：

1.使用图 **(**graph**)** 来表示计算任务**.**

2.在被称之为 会话 **(**Session**)** 的上下文 **(**context**)** 中执行图**.**

3.使用 tensor 表示数据**.**

4.通过 变量 **(**Variable**)** 维护状态**.**

5.使用 feed 和 fetch 可以为任意的操作**(**arbitrary operation**)**

### 2.2 计算图

tensorFlow 程序通常被组织成一个构建阶段和一个执行阶段. 在构建阶段, op 的执行步骤 被描述成一个图. 在执行阶段, 使用会话执行执行图中的 op。

其实这就是一个声明式编程结构，就好比炒菜，我们都是把主材和佐料就准备好，才添油烹制。tensorflow的计算方式也是如此，我们先在构建阶段将这个网络（如神经网络）构建出来，然后我们使用TensorFlow提供的Session方法开启一个运行（run()）将我们的网络放进去run一下，就可以得到我们想到的结果。这就是声明式的编程方式。

**构建图：**

import tensorflow as tf

# 创建一个常量 op, 产生一个1x2 矩阵这个op被称作是一个节点

# 加到默认图中

#

#构造器的返回值代表常量 op的返回值

a **=** tf**.**constant**([[**3.**,** 3.**]])**

b **=** tf**.**constant**([[**2.**],[**2.**])**

# 创建一个矩阵乘法操作， a, b作为输入

# 返回 product 代表乘法的结果

product **=** tf**.**matmul**(**a**,**b**)**

现在已经创建好了一个 两个矩阵相乘并返回product结果的图。也就是说主材和佐料已经准备好了。接下来就是到了生火炒菜的时间了。为了得到product，我们就必须在一个会话（Session）中启动这个一个图了。

**启动一个图：**

#接着上面的代码

sess **=** tf**.**Session**()**

# 调用 sess 的 'run()' 方法来执行矩阵乘法 op, 传入 'product' 作为该方法的参数.

# 上面提到, 'product' 代表了矩阵乘法 op 的输出, 传入它是向方法表明, 我们希望取回

# 矩阵乘法 op 的输出.

#

# 整个执行过程是自动化的, 会话负责传递 op 所需的全部输入. op 通常是并发执行的.

#

# 函数调用 'run(product)' 触发了图中三个 op (两个常量 op 和一个矩阵乘法 op) 的执行.

#

# 返回值 'result' 是一个 numpy `ndarray` 对象.

sess**.**run**(**product**)**

print result

# ==> [[12.]]

# 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

TensorFlow自从0.8版本开始支持分布式处理的机器学习，而且同时，TensorFlow会充分的利用计算机资源：cpu GPU 等。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**.**

### 2.3 交互式使用

文档中的 Python 示例使用一个会话 Session 来 启动图, 并调用 \*\*Session.run()\*\*方法执行操作.

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。

# 进入一个交互式的会话

import tensorflow as tf

sess **=** tf**.**InteractiveSession**()**

x **=** tf**.**Variable**([**1.0**,** 2.0**])**

a **=** tf**.**constant**([**3.0**,** 3.0**])**

# 使用初始化来初始化 Variable

x**.**initializer**.**run**()**

# 增加一个减法操作 sub op 从x 减去a,然后输出结果

sub **=** tf**.**sub**(**x**,**a**)**

print sub**.**eval**()**

# ==> [-2 -1]

### 2.4 打印

TensorFlow中的许多方法返回的都是一个Tensor对象。在Debug的过程中，我们发现只能看到Tensor对象的一些属性信息，无法查看Tensor具体的输出值；而对于Variable和Constant，我们很容易对其进行创建操作，但是如何得到它们的值呢？

假设ts是我们想要查看的对象(Variable / Constant / 0输入的Tensor)，运行

ts\_res **=** sess**.**run**(**ts**)**

print**(**ts\_res**)**

#其中，sess为之前创建或默认的session. 运行后将得到一个narray格式的ts\_res对象，通过print函数我们可以很方便的查看其中的内容。

但是，如果ts是一个有输入要求的Tensor，需要在查看其输出值前，填充(feed)输入数据。如下（假设ts只有一种输入）：

input **=** ×××××× # the input data need to feed

ts\_res **=** sess**.**run**(**ts**,** feed\_dict**=**input**)**

print**(**ts\_res**)**

### 2.5 fetch和feed

**fetch:**

为了取回操作的输出内容, 可以在使用 Session 对象的 run() 调用 执行图时, 传入一些 tensor, 这些 tensor 会帮助你取回结果. 在之前的例子里, 我们只取回了单个节点 state, 但是你也可以取回多个 tensor:

input1 **=** tf**.**constant**(**3.0**)**

input2 **=** tf**.**constant**(**2.0**)**

input3 **=** tf**.**constant**(**5.0**)**

intermed **=** tf**.**add**(**input2**,** input3**)**

mul **=** tf**.**mul**(**input1**,** intermed**)**

with tf**.**Session**():**

result **=** sess**.**run**([**mul**,** intermed**])**

print result

# 输出:

# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]

**feed：**

上述示例在计算图中引入了 tensor, 以常量或变量的形式存储. TensorFlow 还提供了 feed机制, 该机制 可以临时替代图中的任意操作中的 tensor 可以对图中任何操作提交补丁, 直接插入一个 tensor.

feed使用一个 tensor 值临时替换一个操作的输出结果. 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. 最常见的用例是将某些特殊的操作指定为 "feed" 操作, 标记的方法是使用 tf.placeholder() 为这些操作创建占位符.

input1 **=** tf**.**placeholder**(**tf**.**types**.**float32**)**

input2 **=** tf**.**placeholder**(**tf**.**types**.**float32**)**

output **=** tf**.**mul**(**input1**,** input2**)**

with tf**.**Session**()** as sess**:**

print sess**.**run**([**output**],** feed\_dict**={**input1**:[**7.**],** input2**:[**2.**]})**

# 输出:

# [array([ 14.], dtype=float32)]

如果没有正确提供 feed, placeholder() 操作将会产生错误.

### 2.6 Variable

tensorflow中很重要的一个部分就是Variable,它能构建一个变量，在计算图的运算过程中，其值会一直保存到程序运行结束，而一般的tensor张量在tensorflow运行过程中只是在计算图中流过，并不会保存下来，**因此varibale主要用来保存tensorflow构建的一些结构中的参数，这样，这些参数才不会随着运算的消失而消失，才能最终得到一个模型。比如神经网络中的权重和bias等，在训练过后，总是希望这些参数能够保存下来，而不是直接就消失了，所以这个时候要用到Variable。**注意，所有和varible有关的操作在计算的时候都要使用session会话来控制，包括计算，打印等等。

#### 2.6.1 创建

当你创建一个变量时，你传递一个tensor数据作为它的初始值给Variable()构造器。Tensorflow提供了一堆操作从常量或者随机值中产生tensor数据用于初始化。

注意这些操作要求你指定tensor数据的形状。这个形状自动的成为变量的形状。变量的形状是固定的。不过，Tensorflow提供了一些高级机制用于改变变量的形状。

# 创建两个变量

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

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

#### 2.6.2 初始化

变量的初始化必须找模型的其他操作之前，而且必须显示的运行。最简单的方式是添加一个节点用于初始化所有的变量，然后在使用模型之前运行这个节点。

或者你可以选择从checkpoint文件中加载变量，之后将会介绍。

使用tf.global\_variables\_initializer()添加节点用于初始化所有的变量。在你构建完整个模型并在会话中加载模型后，运行这个节点。

# 创建两个变量

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

name**=**"weights"**)**

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

**...**

# 添加用于初始化变量的节点

init\_op **=** tf**.**global\_variables\_initializer**()**

# 然后，在加载模型的时候

with tf**.**Session**()** as sess**:**

# 运行初始化操作

sess**.**run**(**init\_op**)**

**...**

# 使用模型

**...**

tf.global\_variables\_initializer()能够将所有的变量一步到位的初始化，非常的方便。你也可以将指定的列表传递给它，只初始化列表中的变量。 更多的选项请查看Variables Documentation，包括检查变量是否初始化。

#### 2.6.3 从别的变量中初始化

有时候你需要利用另一个变量来初始化当前变量。由于tf.global\_variables\_initializer()添加的节点适用于并行的初始化所有变量，所有如果你有这个需求，你得小心谨慎。

为了从另一个变量中初始化一个新的变量，使用变量的另一个方法initialized\_value()。你可以直接将旧变量的初始值作为新变量的初始值，或者你可以将旧变量的初始值进行一些运算后再作为新变量的初始值。

# 使用随机数创建一个变量

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

name**=**"weights"**)**

# 创建另一个变量，它与weights拥有相同的初始值

w2 **=** tf**.**Variable**(**weights**.**initialized\_value**(),** name**=**"w2"**)**

# 创建另一个变量，它的初始值是weights的两倍

w\_twice **=** tf**.**Variable**(**weights**.**initialized\_value**()** **\*** 2.0**,** name**=**"w\_twice"**)**

#### 2.6.4 保存和加载

使用tf.train.Saver()创建一个Saver对象，然后用它来管理模型中的所有变量。

# 创建一些变量

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

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

**...**

# 添加用于初始化变量的节点

init\_op **=** tf**.**global\_variables\_initializer**()**

# 添加用于保存和加载所有变量的节点

saver **=** tf**.**train**.**Saver**()**

# 然后，加载模型，初始化所有变量，完成一些操作后，把变量保存到磁盘上

with tf**.**Session**()** as sess**:**

sess**.**run**(**init\_op**)**

# 进行一些操作

**..**

# 将变量保存到磁盘上

save\_path **=** saver**.**save**(**sess**,** "/tmp/model.ckpt"**)**

print**(**"Model saved in file: %s" **%** save\_path**)**

Saver对象还可以用于加载变量。注意当你从文件中加载变量是，你不用实现初始化它们。

# 创建两个变量

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

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

**...**

# 添加用于保存和加载所有变量的节点

saver **=** tf**.**train**.**Saver**()**

# 然后，加载模型，使用saver对象从磁盘上加载变量，之后再使用模型进行一些操作

with tf**.**Session**()** as sess**:**

# 从磁盘上加载对象

saver**.**restore**(**sess**,** "/tmp/model.ckpt"**)**

print**(**"Model restored."**)**

# 使用模型进行一些操作

**...**

#### 2.6.5 用tensorflow实现计数器

# 创建变量，初始化为0

state **=** tf**.**Variable**(**0**,** name**=**"counter"**)**

# 创建一个 op , 其作用是时 state 增加 1

one **=** tf**.**constant**(**1**)** # 直接用 1 也就行了

new\_value **=** tf**.**add**(**state**,** 1**)**

update **=** tf**.**assign**(**state**,** new\_value**)**

# 启动图之后， 运行 update op

with tf**.**Session**()** as sess**:**

# 创建好图之后，变量必须经过‘初始化’

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

# 查看state的初始化值

print sess**.**run**(**state**)**

**for** \_ in range**(**3**):**

sess**.**run**(**update**)** # 这样子每一次运行state 都还是1

print sess**.**run**(**state**)**

### 2.7 placeholder

tensorflow中又一保存数据的利器，placeholder（type,strucuct…)它的第一个参数是你要保存的数据的数据类型，大多数是tensorflow中的float32数据类型，后面的参数就是要保存数据的结构，比如要保存一个1×2的矩阵，则struct=[1 2]。它在使用的时候和前面的variable不同的是在session运行阶段，需要给placeholder提供数据，利用feed\_dict的字典结构给placeholdr变量“喂数据”，具体使用如下：

# -\*- coding: utf-8 -\*-

import tensorflow as tf

a**=**tf**.**placeholder**(**tf**.**float32**)**

b**=**tf**.**placeholder**(**tf**.**float32**)**

c**=**tf**.**add**(**a**,**b**)**

with tf**.**Session**()** as sess**:**

print**(**sess**.**run**(**c**,**feed\_dict**={**a**:**10**,**b**:**30**}))** #把10赋给a，30赋给b

运行结果为40

### 2.8 TensorBoard

TensorFlow自带的一个强大的可视化工具，主要用途如下：

Event**:** 展示训练过程中的统计数据（最值，均值等）变化情况

Image**:** 展示训练过程中记录的图像

Audio**:** 展示训练过程中记录的音频

Histogram**:** 展示训练过程中记录的数据的分布图

大致原理为：

在运行过程中，记录结构化的数据

运行一个本地服务器，监听6006端口

请求时，分析记录的数据，绘制

#### 2.8.1 一个简单的例子

在使用简单网络实现mnist手写识别的基础上，添加tensorboard相关代码，用粗体表示。

#coding:utf-8

import tensorflow as tf

import numpy as np

from tensorflow**.**examples**.**tutorials**.**mnist import input\_data

######prepare data

mnist **=** input\_data**.**read\_data\_sets**(**'MNIST\_data/'**,**one\_hot **=** True**)**

#######create the graph

x **=** tf**.**placeholder**(**tf**.**float32**,**shape **=** **[**None**,**784**],**name **=** 'x'**)**

y\_ **=** tf**.**placeholder**(**tf**.**float32**,**shape **=** **[**None**,**10**],**name **=** 'y\_'**)**

W **=** tf**.**Variable**(**tf**.**zeros**([**784**,**10**]),**name **=** 'w'**)**

b **=** tf**.**Variable**(**tf**.**zeros**([**10**]),**name **=** 'b'**)**

y **=** tf**.**nn**.**softmax**(**tf**.**matmul**(**x**,**W**)** **+** b**)**

**tf.summary.histogram('y',y)**

cross\_entropy **=** **-**tf**.**reduce\_sum**(**y\_**\***tf**.**log**(**y**))**

**tf.summary.scalar('loss\_function', cross\_entropy)**

train\_step **=** tf**.**train**.**GradientDescentOptimizer**(**0.01**).**minimize**(**cross\_entropy**)**

correct\_prediction **=** tf**.**equal**(**tf**.**argmax**(**y**,** 1**),** tf**.**argmax**(**y\_**,** 1**))**

accuracy **=** tf**.**reduce\_mean**(**tf**.**cast**(**correct\_prediction**,** "float32"**))**

**tf.summary.scalar('accuracy', accuracy)**

**merged\_summary\_op = tf.summary.merge\_all()**

sess **=** tf**.**InteractiveSession**()**

init **=** tf**.**global\_variables\_initializer**()**

sess**.**run**(**init**)**

**if tf.gfile.Exists("/tmp/mnist\_logs"):**

**tf.gfile.DeleteRecursively("/tmp/mnist\_logs")**

**summary\_writer = tf.summary.FileWriter('/tmp/mnist\_logs',sess.graph)**

**for** i in range**(**2000**):**

batch **=** mnist**.**train**.**next\_batch**(**100**)**

sess**.**run**(**train\_step**,**feed\_dict**={**x**:** batch**[**0**],** y\_**:** batch**[**1**]})**

**summary\_str = sess.run(merged\_summary\_op,feed\_dict={x: batch[0], y\_: batch[1]})**

**summary\_writer.add\_summary(summary\_str, i)**

**if** i **%** 50 **==** 0**:**

print "Setp: "**,** i**,** "Accuracy: "**,**sess**.**run**(**accuracy**,** feed\_dict**={**x**:** mnist**.**test**.**images**,** y\_**:** mnist**.**test**.**labels**})**

print**(**accuracy**.**eval**(**session**=**sess**,**feed\_dict**={**x**:** mnist**.**test**.**images**,** y\_**:** mnist**.**test**.**labels**}))**

然后运行上面的代码，可以在运行的时候或者运行完成后打开tensorboard。

接下来需要启动tensorboard，启动方法如下：

**tensorboard --logdir='/tmp/mnist\_logs'**

Starting TensorBoard b'54' at http://emeetAI:6006

**(**Press CTRL**+**C to quit**)**

然后便可以使用浏览器打开<http://emeetAI:600>访问了。

#### 2.8.2 如何使用

Tensorboard Web前端有7个栏目，都一一对应着我们程序中定义信息的类型。



##### 2.8.2.1 GRAPHS

典型的 TensorFlow 可以有数以千计的节点，如此多而难以一下全部看到，甚至无法使用标准图表工具来展示。为简单起见，我们为变量名划定范围，并且可视化把该信息用于在图表中的节点上定义一个层级。默认情况下， 只有顶层节点会显示。下面这个例子使用tf.name\_scope在hidden命名域下定义了三个操作：

import tensorflow as tf

with tf**.**name\_scope**(**'hidden'**)** as scope**:**

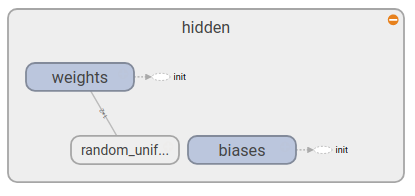
a **=** tf**.**constant**(**5**,** name**=**'alpha'**)**

W **=** tf**.**Variable**(**tf**.**random\_uniform**([**1**,** 2**],** **-**1.0**,** 1.0**),** name**=**'weights'**)**

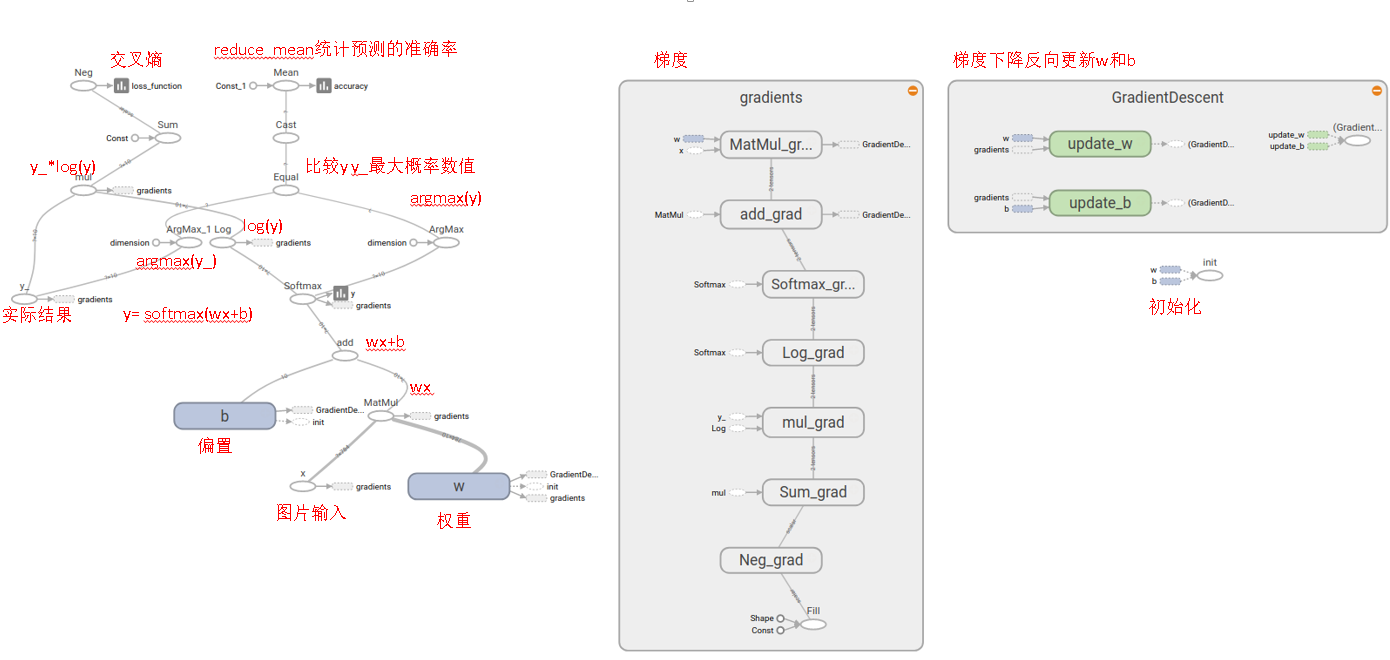
b **=** tf**.**Variable**(**tf**.**zeros**([**1**]),** name**=**'biases'**)**

默认地，三个操作名会折叠为一个节点并标注为hidden。其额外细节并没有丢失，你可以双击，或点击右上方橙色的+来展开节点，然后就会看到三个子节点alpha，weights和biases了。





在上面的例子的中，GRAPHS展示的内容如下：

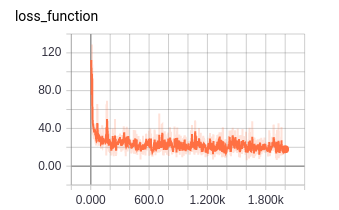


##### 2.8.2.2 SCALARS

展示的是标量的信息，我程序中用tf.summary.scalars()定义的信息都会在这个窗口。

上面的例子中，accuracy和loss随着训练的次数的变化如下图显示：





##### 2.8.2.3 IMAGES

我们也可以保存处理过程中的图片信息，例如：

with tf**.**name\_scope**(**'input\_reshape'**):**

image\_shaped\_input **=** tf**.**reshape**(**x**,** **[-**1**,** 28**,** 28**,** 1**])**

tf**.**summary**.**image**(**'input'**,** image\_shaped\_input**,** 10**)**

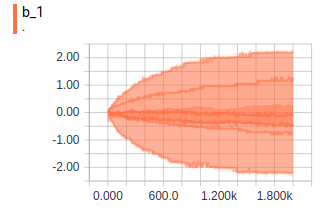
将输入的数据转换成[28 \* 28 \* 1]的shape，存储成另一个tensor，命名为image\_shaped\_input。为了能使图片在tensorbord上展示出来，使用tf.summary.image将图片数据汇总给tensorbord。tf.summary.image（）中传入的第一个参数是命名，第二个是图片数据，第三个是最多展示的张数，此处为10张。

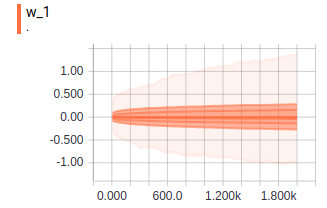
##### 2.8.2.4 AUDIO

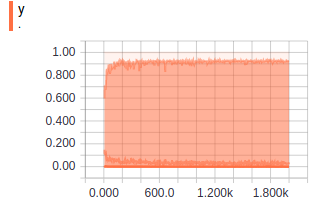
这里展示的是声音的信息，但本案例中没有涉及到声音的。

##### 2.8.2.5 DISTRIBUTION

展示训练过程中记录的数据的分布图。

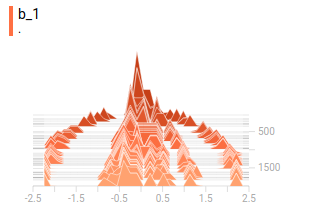


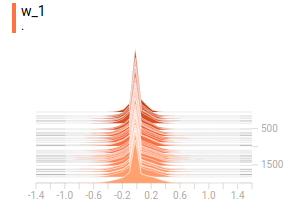
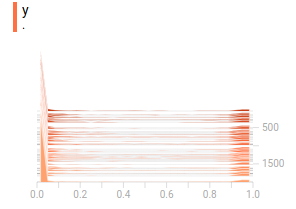




##### 2.8.2.6 HISTOGRAMS

展示训练过程中记录的数据的柱状图。



##### 2.8.2.7 EMBEDDINGS

展示词向量（如Word2vec）后的投影分布。

##### 2.8.2.8 展示

最后需要将上面的数据整合并存储到SummaryWriter。

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

summary\_writer **=** tf**.**train**.**SummaryWriter**(**'/tmp/mnist\_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**)**

2.9

tensorflow调试方法