

人工智能之深度学习

TensorFlow基础

上海育创网络科技有限公司

主讲人：Steven Tang

TensorFlow介绍

- **Tagline:** An open-source software library for Machine Intelligence.
- **Definition:** TensorFlow™ is an open source software library for numerical computation using data flow graphs.
- **GitHub:** <https://github.com/tensorflow/tensorflow>
- **Website:** <https://tensorflow.org/>
<https://tensorflow.google.cn/>

TensorFlow介绍

- TensorFlow™ 是一个采用数据流图（data flow graphs），用于数值计算的开源软件库。TensorFlow 最初由Google大脑小组（隶属于Google机器学习研究机构）的研究员和工程师们开发出来，用于机器学习和深度神经网络方面的研究，但这个系统的通用性使其也可广泛用于其他计算领域。它是谷歌基于DistBelief进行研发的第二代人工智能学习系统。2015年11月9日，Google发布人工智能系统TensorFlow并宣布开源。
- 其命名来源于本身的原理，Tensor（张量）意味着N维数组，Flow（流）意味着基于数据流图的计算。Tensorflow运行过程就是张量从图的一端流动到另一端的计算过程。张量从图中流过的直观图像是其取名为“TensorFlow”的原因。

TensorFlow

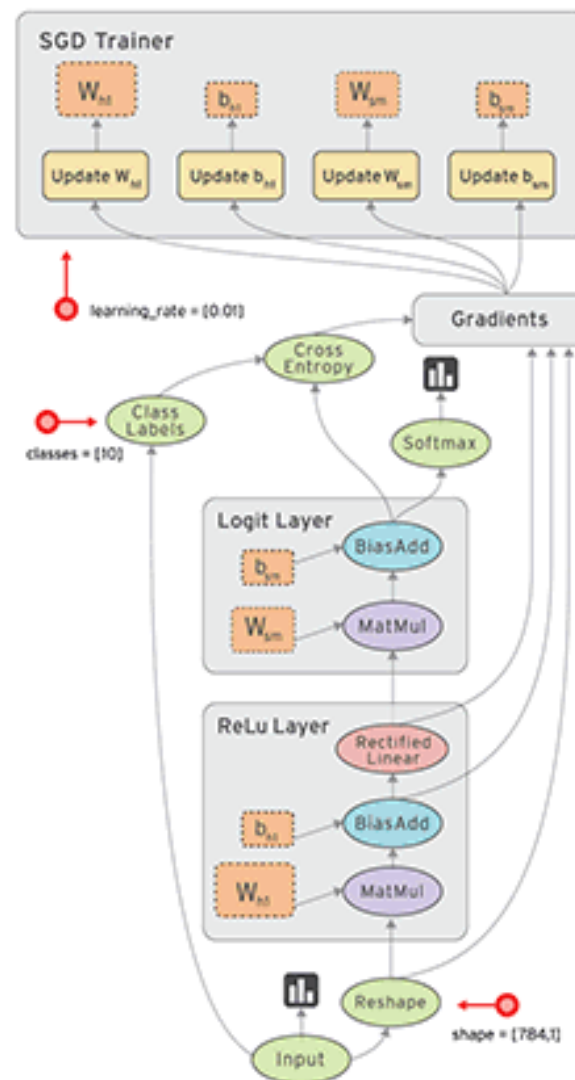
- TensorFlow的关键点是：“**Data Flow Graphs**”，表示TensorFlow是一种基于图的计算框架，其中节点（**Nodes**）在图中表示数学操作，线（**Edges**）则表示在节点间相互联系的多维数据数组，即张量（**Tensor**），这种基于流的架构让TensorFlow具有非常高的灵活性，该灵活性也让TensorFlow框架可以在多个平台上进行计算，例如：台式计算机、服务器、移动设备等。
- **备注**：TensorFlow的开发过程中，重点在于构建执行流图。

机器学习定义

- **Machine Learning**(ML) is a scientific discipline that deals with the construction and study of algorithms that can learn from data.
- 机器学习是一门从数据中研究算法的科学学科。
- 机器学习直白来讲，是根据已有的数据，进行算法选择，并基于算法和数据构建模型，最终对未来进行预测

What is Data Flow Graphs?

- 数据流图使用节点 (Node) 和线 (Edges) 的有向图描述数学计算；节点一般用来表示施加的数学操作，也可以表示数据输入 (feed in) 的起点和输出 (push out) 的终点，或者是读取/写入持久变量 (persistent variable) 的终点。线表示的是节点之间的输入/输出关系，这些线可以输运 “size可动态调整” 的多维数组，即张量 (Tensor)。
- 一旦输入端的所有张量准备好，节点将被分配到各种计算设备完成异步并行地执行运算。



TensorFlow的特性

- **高度的灵活性**：只要能够将计算表示成为一个数据流图，那么就可以使用TensorFlow。
- **可移植性**：TensorFlow支持CPU和GPU的运算，并且可以运行在台式机、服务器、手机移动端设备等等。
- **自动求微分**：TensorFlow内部实现了自动对于各种给定目标函数求导的方式。
- **多种语言支持**：Python、C++
- **性能高度优化**

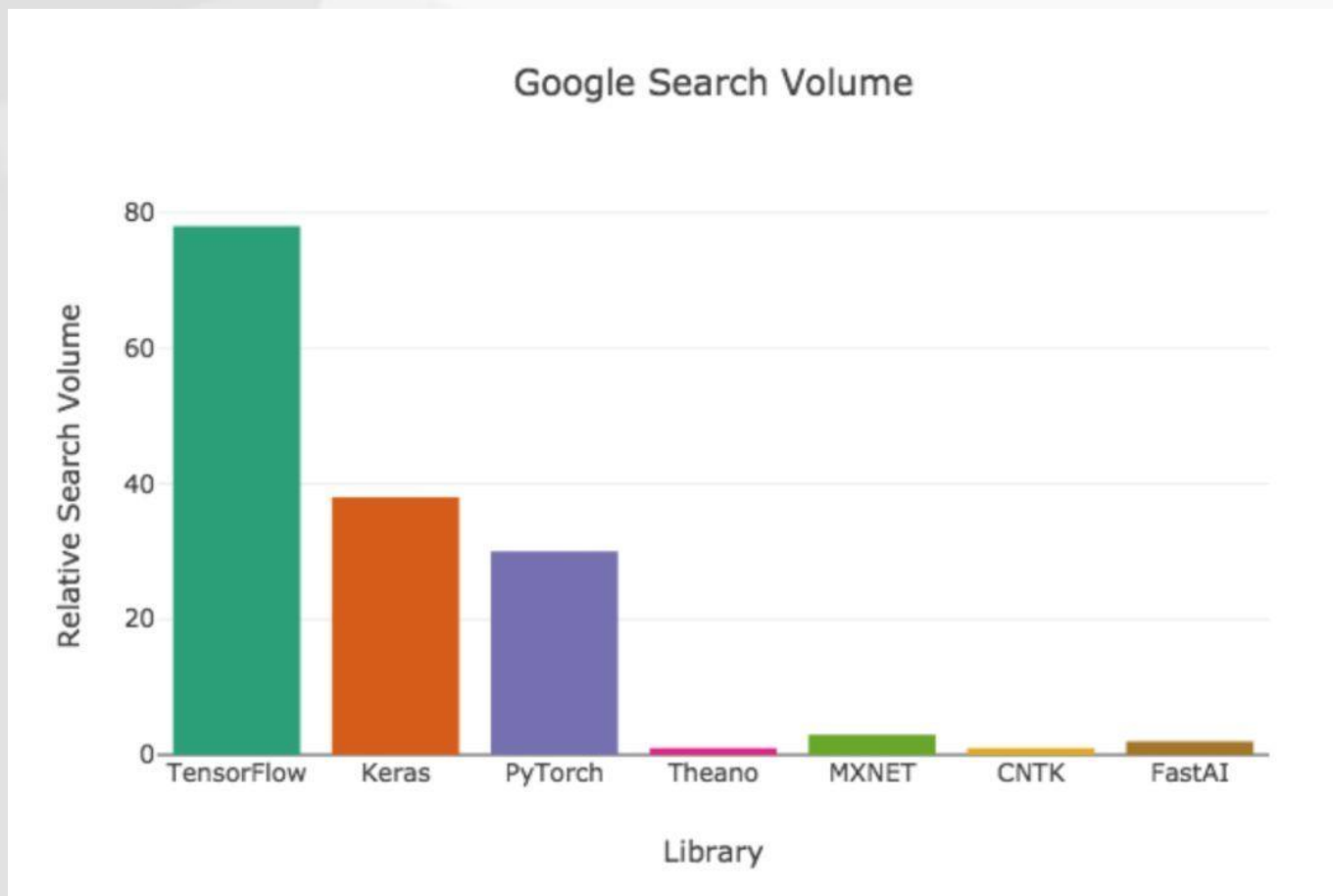
主流框架一览（WHY Tensorflow？）

库名	发布者	支持语言	支持系统
TensorFlow	Google	Python/C++/ Java/Go	Linux/Mac/Windows OS/Android/iOS
Caffe	UC Berkeley	Python/C++/ Matlab	Linux/Mac OS/Windows
CNTK	Microsoft	Python/C++/ BrainScript	Linux/Windows
MXNet	DMLC (分布式机器学习社区)	Python/C++/Matlab/ Julia/Go/R/Scala	Linux/Mac OS/ Windows/Android/iOS
Pytorch	Facebook	Python	Linux/Mac OS/ Windows/Android/iOS
Theano	蒙特利尔大学	Python	Linux/Mac OS/Windows
Keras	François Chollet	Python	Linux/Windows

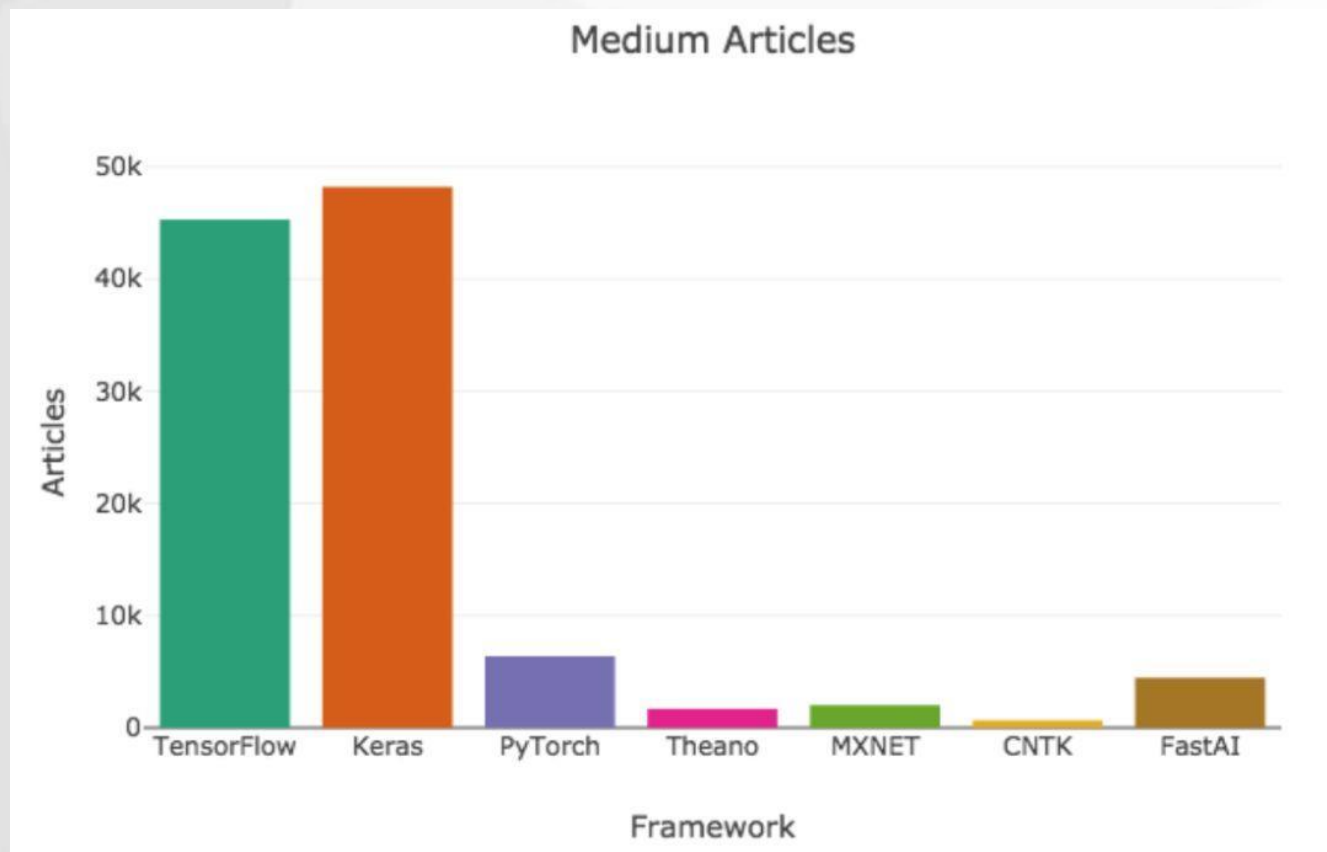
主流机器学习框架

库名	学习材料 丰富程度	CNN建模 能力	RNN建模 能力	易用程度	运行速度	多GPU支持 程度
TensorFlow	★★★★	★★★★	★★	★★★★	★★	★★
Caffe	★	★★	★	★	★	★
CNTK	★	★★★★	★★★★	★	★★	★
MXNet	★★	★★	★	★★	★★	★★★★
Pytorch	★★	★★★★	★★	★★	★★★★	★★
Theano	★★	★★	★★	★	★★	★★
Keras	★★★★	★★	★★★★	★★★★	★	★★

主流机器学习框架



主流机器学习框架



TensorFlow安装

- 要求：Python必须是64位
- 根据TensorFlow的计算方式，TensorFlow的安装分为CPU版本和GPU版本
- 对于Python3.5或者Python3.6，可以使用pip install tensorflow（安装CPU版本）和pip install tensorflow-gpu（安装GPU版本）
- 对于Python2.7，只能通过源码编译来安装TensorFlow(Windows操作系统)
- 备注：TensorFlow-GPU要求机器的显卡必须是NVidia的显卡。
- 备注：授课TensorFlow版本选择1.4.0

TensorFlow安装

- TensorFlow CPU版本安装:

- 环境: Python 3.6

- 安装命令:

- `pip install tensorflow-gpu==1.4.0`

```
C:\Users\ibf>pip install tensorflow==1.4.0
Collecting tensorflow==1.4.0
  Using cached https://files.pythonhosted.org/packages/76/7b/2048b4ecd8
.4.0-cp36-cp36m-win_amd64.whl
Collecting tensorflow-tensorboard<0.5.0,>=0.4.0rc1 (from tensorflow==1.
  Using cached https://files.pythonhosted.org/packages/e9/9f/5845c18f9d
ensorboard-0.4.0-py3-none-any.whl
Requirement already satisfied: numpy>=1.12.1 in c:\anaconda3\lib\site-p
Requirement already satisfied: wheel>=0.26 in c:\anaconda3\lib\site-pac
Requirement already satisfied: enum34>=1.1.6 in c:\anaconda3\lib\site-p
Requirement already satisfied: protobuf>=3.3.0 in c:\anaconda3\lib\site
Requirement already satisfied: six>=1.10.0 in c:\anaconda3\lib\site-pac
Requirement already satisfied: markdown>=2.6.8 in c:\anaconda3\lib\site
flow==1.4.0) (2.6.11)
Requirement already satisfied: html5lib==0.9999999 in c:\anaconda3\lib\
nsorflow==1.4.0) (0.9999999)
Requirement already satisfied: bleach==1.5.0 in c:\anaconda3\lib\site-p
ow==1.4.0) (1.5.0)
Requirement already satisfied: werkzeug>=0.11.10 in c:\anaconda3\lib\si
orflow==1.4.0) (0.11.15)
Requirement already satisfied: setuptools in c:\anaconda3\lib\site-pack
low==1.4.0) (27.2.0)
Installing collected packages: tensorflow-tensorboard, tensorflow
Successfully installed tensorflow-1.4.0 tensorflow-tensorboard-0.4.0
```

```
C:\Users\ibf>python
Python 3.6.0 |Anaconda 4.3.1 (64-bit)| (default, Dec 23 2016, 11:57:41)
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow
>>> _
```

GPU安装中CUDA, Cudnn, Python对应关系

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow_gpu-1.14.0	3.5-3.7	MSVC 2017	Bazel 0.24.1-0.25.2	7.4	10
tensorflow_gpu-1.13.0	3.5-3.7	MSVC 2015 update 3	Bazel 0.19.0-0.21.0	7.4	10
tensorflow_gpu-1.12.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.11.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.9.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.8.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.7.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.6.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.5.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.4.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	6	8
tensorflow_gpu-1.3.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	6	8
tensorflow_gpu-1.2.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	5.1	8
tensorflow_gpu-1.1.0	3.5	MSVC 2015 update 3	Cmake v3.6.3	5.1	8
tensorflow_gpu-1.0.0	3.5	MSVC 2015 update 3	Cmake v3.6.3	5.1	8

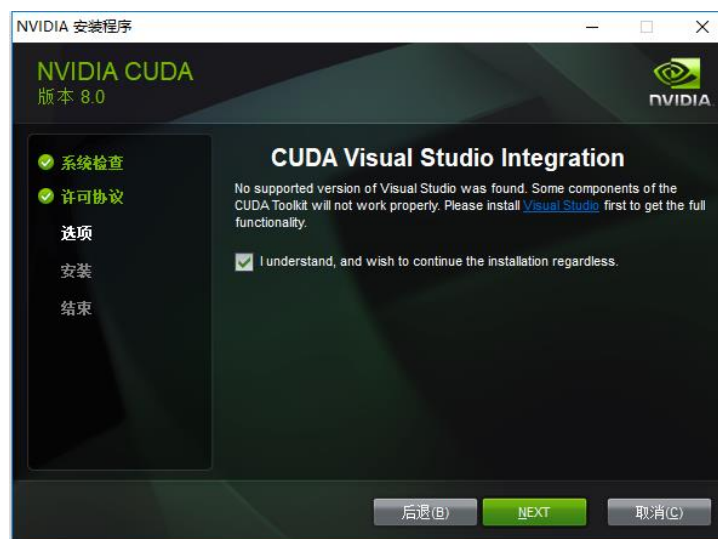
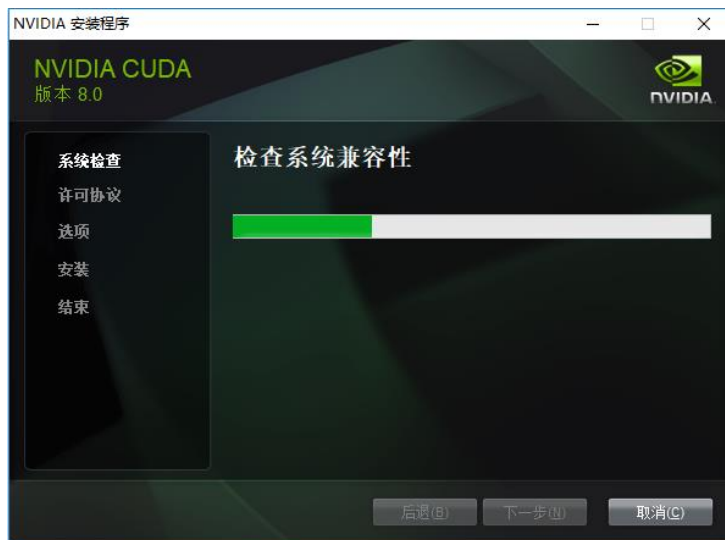
```
x=tf.placeholder(tf.int32)
y=tf.placeholder(tf.int32)
q=tf.placeholder(tf.int32)
u=tf.add(x,y)
z=tf.subtract(u,tf.constant(1))
# 需要你编程：从session中打印 z
with tf.Session() as sess:
    output=sess.run(z,feed_dict={x:10,y:2})
    print(output)
```

TensorFlow-GPU安装-CUDA

- CUDA下载安装链接: <https://developer.nvidia.com/cuda-toolkit>
- cuDNN下载安装链接: <https://developer.nvidia.com/rdp/cudnn-archive>
- 备注: 具体的CUDA版本根据导入tensorflow时提示的异常来选择; 即先安装tensorflow-gpu, 然后在python的命令行执行: `import tensorflow`, 会出现如下异常, 则表示我们需要安装的是CUDA 8.0版本, 至于cuDNN选择和CUDA对应版本即可。

```
ImportError: Could not find 'cudart6480.dll'. TensorFlow requires that this DLL be installed in a directory that is named in your %PATH% environment variable. Download and install CUDA 8.0 from this URL: https://developer.nvidia.com/cuda-toolkit
```


TensorFlow-GPU安装-CUDA



设置环境变量

计算机上点右键，打开属性->高级系统设置->环境变量，可以看到系统中多了CUDA_PATH和CUDA_PATH_V8_0两个环境变量，接下来，还要在系统中添加以下几个环境变量

:

CUDA_SDK_PATH = C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0(这是默认安装位置的路径)

CUDA_LIB_PATH = %CUDA_PATH%\lib\x64

CUDA_BIN_PATH = %CUDA_PATH%\bin

CUDA_SDK_BIN_PATH =
%CUDA_SDK_PATH%\bin\win64

CUDA_SDK_LIB_PATH =
%CUDA_SDK_PATH%\common\lib\x64

100%

- ```
C:\Users\ibf>set path
Path=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\bin;C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\libnvvp;
aconda3;C:\Anaconda3\Scripts;C:\Anaconda3\Library\bin;C:\Program Files\Java\jdk1.8.0_144\bin;C:\Program Files\scala-2.10.4\bin;C:
ows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\NVIDIA Corpora
\PhysX\Common;C:\ProgramData\Oracle\Java\javapath;C:\ffmpeg\bin;C:\Program Files (x86)\Windows Kits\8.1\Windows Performance Toolk
:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\cmd.exe;C:\WIND
t\WindowsApps;C:\Program Files\Microsoft VS Code\bin;
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
```

The screenshot shows the file explorer window for the path 'Program Files > NVIDIA GPU Computing Toolkit > CUDA > v8.0'. The search bar at the top right contains 'v8.0'. The file list below shows various folders and files. The folders 'bin' and 'lib' are circled in red. The file list is as follows:

| 名称                             | 修改日期            | 类型   | 大小    |
|--------------------------------|-----------------|------|-------|
| bin                            | 2018/4/21 16:10 | 文件夹  |       |
| doc                            | 2018/4/21 16:03 | 文件夹  |       |
| extras                         | 2018/4/21 16:03 | 文件夹  |       |
| include                        | 2018/4/21 16:10 | 文件夹  |       |
| jre                            | 2018/4/21 16:03 | 文件夹  |       |
| lib                            | 2018/4/21 16:03 | 文件夹  |       |
| libnvvp                        | 2018/4/21 16:03 | 文件夹  |       |
| nvml                           | 2018/4/21 16:03 | 文件夹  |       |
| nvvm                           | 2018/4/21 16:03 | 文件夹  |       |
| src                            | 2018/4/21 16:03 | 文件夹  |       |
| tools                          | 2018/4/21 16:03 | 文件夹  |       |
| CUDA_Toolkit_Release_Notes.txt | 2017/1/12 4:39  | 文本文档 | 38 KB |
| EULA.txt                       | 2017/1/12 4:39  | 文本文档 | 97 KB |
| version.txt                    | 2017/1/12 4:39  | 文本文档 | 1 KB  |

# TensorFlow基本概念

- **图 (Graph)** : 图描述了计算的过程, TensorFlow使用图来表示计算任务。
- **张量 (Tensor)** : TensorFlow使用tensor表示数据。每个Tensor是一个类型化的多维数组。
- **操作 (op)** : 图中的节点被称为op (operation的缩写), 一个op获得/输入0个或多个Tensor, 执行计算, 产生0个或多个Tensor。
- **会话 (Session)** : 图必须在称之为“会话”的上下文中执行。会话将图的op分发到诸如CPU或GPU之类的设备上执行。
- **变量 (Variable)** : 运行过程中可以被改变, 用于维护状态。

- TensorFlow的边即有两种连接关系：
  - 数据依赖
  - 控制依赖
- 实线边表示数据依赖，代表数据，即张量。任意维度的数据统称为张量。在机器学习算法中，张量在数据流图中从前往后流动一遍就完成一次前向传播，而残差从后向前流动一遍就完成一次反向传播。
- 虚线边表示控制依赖，可以用于控制操作的运行，这被用来确保happens-before关系，这类边上没有数据流过，但源节点必须在目的节点开始执行前完成。

## 数据属性

| 数据类型         | Python类型     | 描述                        |
|--------------|--------------|---------------------------|
| DT_FLOAT     | tf.float32   | 32位浮点型                    |
| DT_DOUBLE    | tf.float64   | 64位浮点型                    |
| DT_INT64     | tf.int64     | 64位有符号整型                  |
| DT_INT32     | tf.int32     | 32位有符号整型                  |
| DT_INT16     | tf.int16     | 16位有符号整型                  |
| DT_INT8      | tf.int8      | 8位有符号整型                   |
| DT_UINT8     | tf.uint8     | 8位无符号整型                   |
| DT_STRING    | tf.string    | 可变长度的字节数组，每一个张量元素都是一个字节数组 |
| DT_BOOL      | tf.bool      | 布尔型                       |
| DT_COMPLEX64 | tf.complex64 | 由两个32位浮点数组成的复数：实数和虚数      |
| DT_QINT32    | tf.qint32    | 用于量化操作的32位有符号整型           |
| DT_QINT8     | tf.qint8     | 用于量化操作的8位有符号整型            |
| DT_QUINT8    | tf.quint8    | 用于量化操作的8位无符号整型            |

- **节点**又称为**算子**，它代表一个操作，一般用来表示施加的数字运算，也可以表示数据输入的起点以及输出的重点，或者是读取/写出持久化变量的终点。

| 类别        | 示例                                                        |
|-----------|-----------------------------------------------------------|
| 数学运算操作    | Add、Subtract、Multiply、Div、Exp、Log、Greater、Less、Equal..... |
| 数组运算操作    | Concat, Slice, Split, Constant, Rank, Shape, Shuffle..... |
| 矩阵运算操作    | MatMul, MatrixInverse, MatrixDeterminant.....             |
| 有状态的操作    | Variable、Assign、AssignAdd.....                            |
| 神经网络构建操作  | SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool.....       |
| 检查点操作     | Save, Restore.....                                        |
| 队列和同步操作   | Enqueue, Dequeue, MutexAcquire, MutexRelease.....         |
| 控制张量流动的操作 | Merge, Switch, Enter, Leave, NextIteration.....           |

```
import tensorflow as tf
需要你编程：将下面转换成tensorflow
x = 10
y = 2
z = x/y - 1

x=tf.placeholder(tf.int32)
y=tf.placeholder(tf.int32)
z=tf.subtract(tf.div(x,y),tf.constant(1))
需要你编程：从session中打印 z
with tf.Session() as sess:
 output=sess.run(z,feed_dict={x:10,y:2})
 print(output)
```



# TensorFlow基本用法

- TensorFlow可以认为是一种编程工具，使用TensorFlow来实现具体的业务需求，所以我们可以认为TensorFlow就是一个“工具箱”，然后我们使用TensorFlow这个“工具箱”中的各种“工具”（方法/API)来实现各种功能，比如使用TensorFlow实现基本的数值计算、机器学习、深度学习等；使用TensorFlow必须理解下列概念：
  - 使用图(graph)来表示计算任务；
  - 在会话(session)的上下文中执行图；
  - 使用tensor表示数据；
  - 通过变量(Variable)来维护状态；
  - 使用feed和fetch可以为任意的操作(Operation/op)赋值或者从其中获取数据。

## *TensorFlow*程序结构

- TensorFlow的程序一般分为两个阶段：构建阶段和执行阶段；
- 构建阶段：op的执行步骤被描述称为一个图，然后使用TensorFlow提供的API构建这个图。
- 执行阶段：将构建好的执行图(Operation Graph)在给定的会话中执行，并得到执行结果。

- TensorFlow编程的重点是根据业务需求，使用TensorFlow的API将业务转换为执行图（有向无环图）；图中的节点是Tensor，节点之间的连线是节点之间的操作，连线前的节点可以认为是操作的输入，连线后的节点可以认为操作的输出；根据节点的特性（是否有输入输出），可以将节点分为源节点、中间节点和最终的结果节点。
- 图构建的第一步就是创建源op(source op); 源op不需要任何的输入。op构造器的返回值代表被构造出的op的输出，这些返回值可以传递给其它op构造器作为输入或者直接获取结果。
- TensorFlow库中有一个默认图(default graph)，op构造器可以直接为其添加节点，一般情况下，使用默认的Graph即可完成程序代码的实现。不过TensorFlow也支持通过Graph类管理多个图。

- 默认图现在有三个节点，两个constant op和一个matmul op。

```
import tensorflow as tf

]# 创建一个常量op， 产生一个1x2的矩阵， 这个op被作为一个节点
默认是加入到默认图中
]# 构造器的返回值代表该常量op的返回值
mat1 = tf.constant([3, 3], dtype=tf.float32, shape=[1, 2])

创建另一个常量op， 产生一个2x1的矩阵
mat2 = tf.constant([[2.], [2.]])

]# 创建一个矩阵乘法op， 将mat1和mat2作为输入
]# 返回值代表乘法op的结果
product = tf.matmul(mat1, mat2)
```

- 不使用默认图(Graph), 使用多个图来进行编程; 但是注意: 操作必须属于同一个图, 不同图中的节点不能相连。

查看添加的操作是否是添加到默认图中

```
a = tf.constant(4.0)
print("变量a定义在默认图上: {}".format(a.graph is tf.get_default_graph()))
```

明确指定一个新图

```
g = tf.Graph()
with g.as_default():
 # 定义一个新的操作在图g上
 b = tf.constant(3.0)
 c = tf.constant(5.0)
 print("变量b定义在图g上: {}".format(b.graph is g))
print("变量c定义在图g上: {}".format(c.graph is g))
```

定义一个变量, 定义在默认图上

```
d = tf.constant(5.0)
print("变量e定义在默认图上: {}".format(d.graph is tf.get_default_graph()))
```

再定义在一个新图

```
with tf.Graph().as_default() as g2:
 e = tf.constant(6.0)
 print("变量e定义在图g2上: {}".format(e.graph is g2))
```

*# NOTE: 合并多个图的值(这一段代码是错误的, 只能是同一个图的op之间进行操作)*

```
e = tf.add(tf.add(a, b), tf.add(c, d))
print(e.graph is tf.get_default_graph())
```

- 当执行图构建完成后，才能给启动图，进入到执行阶段；启动图的第一步就是创建一个Session对象，如果无任何参数的情况下，会话构造器将启动默认图。

# 启动默认图

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

# 调用sess的run方法来执行矩阵乘法op，传入product作为方法参数

# 在图中的构建中，可以发现product代表的是矩阵的输出，传入它表示希望取出矩阵乘法op的结果

# 结果的执行是自动化的，会话负责传递op的所有输入。op通常是并发执行的

# 函数调用run(product)触发图中的三个op的执行

# 返回值result是一个numpy的ndarray对象。

```
result = sess.run(product)
```

```
print("type: {}, value: {}".format(type(result), result))
```

# 任务完成，关闭会话

```
sess.close()
```

# 使用with代码块，自动完成关闭操作

```
with tf.Session() as sess2:
```

```
 result2 = sess2.run([product])
```

```
 print(result2)
```

- tf.Session在构建会话的时候，如果不给定任何参数，那么构建出来Session对应的内部的Graph其实就是默认Graph，不过我们可以通过参数给定具体对应的是那一个Graph以及当前Session对应的配合参数。Session的构造主要有三个参数，作用如下：
  - target: 给定连接的url，只有当分布式运行的时候需要给定(后面分布式运行讲);
  - graph: 给定当前Session对应的图，默认为TensorFlow中的默认图;
  - config: 给定当前Session的相关参数，参数详见：  
<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto>中的[ConfigProto]

- 通过Session的config参数可以对TensorFlow的应用的执行进行一些优化调整，主要涉及到的参数如下：

| 属性                   | 作用                                                                                                                 |
|----------------------|--------------------------------------------------------------------------------------------------------------------|
| gpu_options          | GPU相关参数，主要参数：per_process_gpu_memory_fraction和allow_growth                                                          |
| allow_soft_placement | 是否允许动态使用CPU和GPU，默认为False；当我们的安装方式为GPU的时候，建议该参数设置为True，因为TensorFlow中的部分op只能在CPU上运行。                                 |
| log_device_placement | 是否打印日志，默认为False，不打印日志                                                                                              |
| graph_options        | Graph优化相关参数，一般不需要给定，默认即可，主要参数：optimizer_options(do_common_subexpression_elimination、do_constant_folding和opt_level) |



# TensorFlow会话

# 构建一个图

```
a = tf.constant('10', tf.string, name='a_const')
b = tf.string_to_number(a, out_type=tf.float64, name='str_2_double')
c = tf.to_double(5.0, name='to_double')
d = tf.add(b, c, name='add')
```

]# 构建Session并执行图

]# 1. 构建GPU相关参数

```
gpu_options = tf.GPUOptions()
```

]# per\_process\_gpu\_memory\_fraction: 给定对于每一个进程, 分配多少的GPU内存, 默认为1

]# 设置为0.5表示分配50%的GPU内存

```
gpu_options.per_process_gpu_memory_fraction = 0.5
```

]# allow\_growth: 设置为True表示在进行GPU内存分配的时候, 采用动态分配方式, 默认为False

# 动态分配的意思是指, 在启动之前, 不分配全部内存, 根据需要, 后面动态的进行内存分配

]# 在开启动态分配后, GPU内存不会自动的释放(故: 复杂、长时间运行的任务不建议)

```
gpu_options.allow_growth = True
```

# 2. 构建Graph优化的相关参数

```
optimizer = tf.OptimizerOptions(
 do_common_subexpression_elimination=True, # 设置为True表示开启公共执行子句优化
 do_constant_folding=True, # 设置为True表示开始常数折叠优化
 opt_level=0 # 设置为0表示开始上述两个优化, 默认就是0
)
graph_options = tf.GraphOptions(optimizer_options=optimizer)
```

# 3. 构建Session的Config相关参数

```
config_proto = tf.ConfigProto(allow_soft_placement=True, log_device_placement=True,
 graph_options=graph_options, gpu_options=gpu_options)
```

# 4. 构建Session并运行

```
with tf.Session(config=config_proto) as sess:
 print(sess.run(d))
```

## TensorFlow会话

- 在TensorFlow中，除了可以使用Session表示这个会话外，还可以通过InteractiveSession来表示会话，InteractiveSession的意思是：交互式会话，使用交互式会话可以降低代码的复杂度，使用Tensor.eval()或者Operation.run()来代替Session.run()方法，这样可以避免一个变量来维持会话；备注：Session也可以使用Tensor.eval()和Operation.run()获取数据/执行操作(只要明确当前会话)。

```
构建图
a = tf.constant(4)
b = tf.constant(3)
c = tf.multiply(a, b)

运行
with tf.Session():
 print(c.eval())
```

```
进入交互式会话
sess = tf.InteractiveSession()

定义变量和常量
x = tf.constant([1.0, 2.0])
a = tf.constant([2.0, 4.0])

进行减操作
sub = tf.subtract(x, a)

输出结果
print(sub.eval())
print(sess.run(sub))
```

## Tensor张量

- TensorFlow使用Tensor数据结构来代表所有数据，计算图中，操作间传递的数据都是Tensor。Tensor可以看作是一个n维的数组或者列表，一个Tensor主要由一个静态数据类型和动态类型的维数(Rank、Shape)组成。Tensor可以在图中的节点之间流通。

| 阶 | 形状               | 维数  | 实例                          |
|---|------------------|-----|-----------------------------|
| 0 | []               | 0-D | 一个 0维张量. 一个纯量.              |
| 1 | [D0]             | 1-D | 一个1维张量的形式[5].               |
| 2 | [D0, D1]         | 2-D | 一个2维张量的形式[3, 4].            |
| 3 | [D0, D1, D2]     | 3-D | 一个3维张量的形式 [1, 4, 3].        |
| n | [D0, D1, ... Dn] | n-D | 一个n维张量的形式 [D0, D1, ... Dn]. |

## TensorFlow变量

- 变量(Variables)是维护图执行过程中的状态信息。在训练模型过程中，可以通过变量来存储和更新参数。变量包含张量(Tensor)存放于内存的缓存区。建模的时候变量必须被明确的初始化，模型训练后变量必须被存储到磁盘。这些变量的值可以在之后的模型训练和分析中被加载。
- 在构建变量的时候，必须将一个张量或者可以转化为张量的Python对象作为初始值传入构造函数Variable中。
- 特别注意：变量的全局初始化(tf.global\_variables\_initializer())是并行执行的，如果存在变量之间的依赖关系的时候，再进行初始化的时候要特别注意。(1.4版本之后该问题就不存在了)

# TensorFlow变量

- 一个简单的使用变量的案例

```
⌋# 创建一个变量，初始化值为标量3.0
```

```
a = tf.Variable(3.0)
```

```
创建一个常量
```

```
b = tf.constant(2.0)
```

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

```
⌋# 启动图后，变量必须先经过初始化操作
```

```
增加一个初始化变量的op到图中
```

```
⌋# tf.initialize_all_variables: 初始化全局所有变量
```

```
init_op = tf.initialize_all_variables()
```

```
启动图
```

```
⌋with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
```

```
 # 运行init op
```

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

```
 # 获取值
```

```
 print("a={}".format(sess.run(a)))
```

```
⌋ print("c={}".format(c.eval()))
```

# TensorFlow变量

- 进行变量依赖的案例创建

```
创建一个变量
```

```
w1 = tf.Variable(tf.random_normal([10], stddev=0.5, dtype=tf.float32), name='w1')
```

```
基于第一个变量创建第二个变量
```

```
a = tf.constant(2, dtype=tf.float32)
```

```
w2 = tf.Variable(w1.initialized_value() * a, name='w2')
```

```
进行全局初始化
```

```
init_op = tf.initialize_all_variables()
```

```
启动图
```

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
```

```
 # 运行init op
```

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

```
 # 获取值
```

```
 result = sess.run([w1, w2])
```

```
 print("w1={} \nw2={}".format(result[0], result[1]))
```

## TensorFlow Fetch

- 为了取回操作的输出内容，可以在使用Session对象的run方法调用执行图的时候，传入一些tensor，通过run方法就可以获取这些tensor对应的结果值。
- 如果需要获取多个tensor的值，那么尽量一次运行就获取所有的结果值，而不是采用逐个获取的方式。

```
启动图
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
 # 运行init op
 sess.run(init_op)
 # 获取值（对于多个值的获取建立获取方式）
 result = sess.run([w1, w2])
 print("w1={} \nw2={}".format(result[0], result[1]))
```

## *TensorFlow Feed*

- Tensorflow还提供了填充机制(feed)，可以在构建图时使用placeholder类型的API临时替代任意操作的张量(占位符)，在调用Session对象的run()方法去执行图时，使用填充数据作为调用的参数，调用结束后，填充数据就消失了。
- feed使用一个tensor值临时替换一个操作的输出结果，在获取数据的时候必须给定对应的feed数据作为参数。feed只有在调用它的方法内有效，方法结束，feed就消失了。
- feed可以使用placeholder类型的API创建占位符，常见API：tf.placeholder、tf.placeholder\_with\_default



## TensorFlow Feed

```
创建占位符, 创建图
m1 = tf.placeholder(tf.float32)
m2 = tf.placeholder(tf.float32)
m3 = tf.placeholder_with_default(4.0, shape=None)
output = tf.multiply(m1, m2)
ot1 = tf.add(m1, m3)

运行图
with tf.Session() as sess:
 print(sess.run(output, feed_dict={m1: 3, m2: 4}))
 print(output.eval(feed_dict={m1: 8, m2: 10}))
 print(sess.run(ot1, feed_dict={m1: 3, m3: 3}))
 print(sess.run(ot1, feed_dict={m1: 3}))
```



```
features = tf.placeholder(tf.float32)
labels = tf.placeholder(tf.float32)
w = get_weights(n_features, n_labels)
b = get_biases(n_labels)

线性函数 $xW + b$
logits = linear(features, w, b)

train_features, train_labels =
mnist_features_labels(n_labels)
需要你编程：设置初始化器2
init=tf.global_variables_initializer()
prediction = tf.nn.softmax(logits)
交叉熵，后面会学到
cross_entropy = -tf.reduce_sum(labels * tf.log(prediction),
 reduction_indices=1)
loss = tf.reduce_mean(cross_entropy)
```



# TensorFlow



GANYMEDE AT  
HALF PHASE

YEAR 1996  
MISSION GALILEO

## TensorFlow案例一

- 使用已介绍的相关TensorFlow相关知识点，实现以下三个功能(变量更新)
  - 1. 实现一个累加器，并且每一步均输出累加器的结果值。
  - 2. 编写一段代码，实现动态的更新变量的维度数目

## TensorFlow控制依赖

- 我们可以通过Variable和assign完成变量的定义和更新，但是如果在更新变量之前需要更新其它变量，那么会导致一个比较严重的问题：也就是需要多次调用sess.run方法来进行变量的更新。通过这种方式，代码复杂程度上升，同时也没有执行效率。
- 解决该问题的方案就是：控制依赖。通过TensorFlow中提供的一组函数来处理不完全依赖的情况下的操作排序问题(即给定哪个操作先执行的问题)，通过tf.control\_dependencies API完成。

```
tmp_sum = sum * i
assign_op = tf.assign(sum, tmp_sum)
with tf.control_dependencies([assign_op]):
 sum = tf.Print(sum, data=[sum, sum.read_value()], message='sum, sum_read:')
```

# TensorFlow设备

- **设备**是指一块可以用来运算并且拥有自己的地址空间的硬件，如CPU和GPU。Tensorflow为了在执行操作的时候，充分利用计算资源，可以明确指定操作在哪个设备上执行。
- 一般情况下，不需要显示指定使用CPU还是GPU，TensorFlow会自动检测。如果检测到GPU，TensorFlow会尽可能地利用第一个GPU来执行操作。注意：如果机器上有超过一个可用的GPU，那么除了第一个外其它GPU默认是不参与计算的。所以，在实际TensorFlow编程中，经常需要明确给定使用的CPU和GPU。

- `"/cpu:0"`：表示使用机器CPU运算
- `"/gpu:0"`：表示使用第一个GPU运算，如果有的话
- `"/gpu:1"`：表示使用第二个GPU运算，以此类推

```
with tf.device('/gpu:0'):
 a = tf.constant([1, 2, 3], name='a')
 b = tf.constant(2, name='b')
 c = a * b

新建session with log_device_placement并设置为True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
运行这个 op.
print(sess.run(c))
```

## TensorFlow变量作用域

- 通过tf.Variable我们可以创建变量，但是当模型复杂的时候，需要构建大量的变量集，这样会导致我们对于变量管理的复杂性，而且没法共享变量(存在多个相似的变量)。针对这个问题，可以通过TensorFlow提供的变量作用域机制来解决，在构建一个图的时候，就可以非常容易的使用共享命名过的变量。
- Tensorflow中有两个作用域，一个是name\_scope，另一个是variable\_scope。
- 变量作用域机制在TensorFlow中主要通过两部分组成：
  - tf.get\_variable：通过所给定的名字创建或者返回一个对应的变量
  - tf.variable\_scope：为通过创建的变量或者操作Operation指定命名空间



# TensorFlow变量作用域

- 使用TensorFlow的变量作用域，体现简化变量数量的案例代码

```
def my_func(x):
 weight = tf.get_variable('weight', [1], initializer=tf.random_normal_initializer())[0]
 bias = tf.get_variable('bias', [1], initializer=tf.random_normal_initializer())[0]
 result = weight * x + bias
 return result, weight, bias

def my_func(x):
 weight1 = tf.Variable(tf.random_normal([1]))[0]
 bias1 = tf.Variable(tf.random_normal([1]))[0]
 result1 = weight1 * x + bias1

 weight2 = tf.Variable(tf.random_normal([1]))[0]
 bias2 = tf.Variable(tf.random_normal([1]))[0]
 result2 = weight2 * x + bias2

 return result1, weight1, bias1, result2, weight2, bias2

def func(x):
 with tf.variable_scope("op1"):
 r1 = my_func(x)
 with tf.variable_scope("op2"):
 r2 = my_func(x)
 return r1, r2
```

## TensorFlow变量作用域

- `tf.get_variable`方法在调用的时候，主要需要给定参数名称`name`，形状`shape`，数据类型`dtype`以及初始化方式`initializer`四个参数。该API底层执行的时候，根据`variable scope`的属性`reuse`的值决定采用何种方式来获取变量。当`reuse`值为`False`的时候(不允许设置)，作用域就是创建新变量设置的，此时要求对应的变量不存在，否则报错；当`reuse`值为`True`的时候，作用域就是为重用变量所设置的，此时要求对应的变量必须存在，否则报错。当`reuse`的值为`tf.AUTO_REUSE`的时候，表示如果变量存在就重用变量，如果变量不存在，就创建新变量返回。(备注：`reuse`一般设置在`variable scope`对象上)
- TF底层使用'变量作用域/变量名称:0'的方式标志变量(eg: `func/op1/weight:0`)。

## TensorFlow变量作用域

- `tf.get_variable`常用的initializer初始化器:

| 初始化器                                                    | 描述                                       |
|---------------------------------------------------------|------------------------------------------|
| <code>tf.constant_initializer(value)</code>             | 初始化为给定的常数值value                          |
| <code>tf.random_uniform_initializer(a, b)</code>        | 初始化为从a到b的均匀分布的随机值                        |
| <code>tf.random_normal_initializer(mean, stddev)</code> | 初始化为均值为mean、方差为stddev的服从高斯分布的随机值         |
| <code>tf.orthogonal_initializer(gini=1.0)</code>        | 初始化一个正交矩阵，gini参数作用是最终返回的矩阵是随机矩阵乘以gini的结果 |
| <code>tf.identity_initializer(gini=1.0)</code>          | 初始化一个单位矩阵，gini参数作用是最终返回的矩阵是随机矩阵乘以gini的结果 |

## TensorFlow变量作用域

- `tf.variable_scope`方法的作用就是定义一个作用域，定义在`variable_scope`作用域中的变量和操作，会将`variable_scope`的名称作为前缀添加到变量/操作名称前，支持嵌套的作用域，添加前缀规则和文件目录路径的规则类似。
- `tf.variable_scope`参数如果给定的是一个已经存在的作用域对象的时候，那么构建变量的时候表示直接跳过当前作用域前缀，直接成为一个完全不同与现在的作用域(直接创建给定作用域下的变量)。但是构建操作的时候，还是和嵌套的方式一样，直接添加子作用域。
- `tf.variable_scope`参数中，可以给定当前作用域中默认的初始化器`initializer`，并且子作用域会直接继承父作用域的相关参数(是否重用、默认初始化器等)

# TensorFlow变量作用域

- 测试嵌套域

```
}with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
 with tf.variable_scope('foo', initializer=tf.constant_initializer(0.4)) as foo_scope:
 v = tf.get_variable("v", [1])
 w = tf.get_variable("w", [1], initializer=tf.constant_initializer(0.3))
 with tf.variable_scope('bar'):
 l = tf.get_variable("l", [2])

 with tf.variable_scope(foo_scope):
 h = tf.get_variable("h", [1])
 g = v + w + l[0] + h
```

```
foo/v:0, [0.4]
foo/w:0, [0.3]
foo/bar/l:0, [0.4 0.4]
foo/h:0, [0.4]
foo/bar/foo/add_2, [1.5]
```

```
sess.run(tf.global_variables_initializer())
print("{} {}".format(v.name, v.eval()))
print("{} {}".format(w.name, w.eval()))
print("{} {}".format(l.name, l.eval()))
print("{} {}".format(h.name, h.eval()))
print("{} {}".format(g.op.name, g.eval()))
```

## TensorFlow变量作用域

- TensorFlow中的`name_scope`和`variable_scope`是两个不同的东西，`name_scope`的主要作用是为`op_name`前加前缀，`variable_scope`是为`get_variable`创建的变量的名字加前缀。简单来讲：使用`tf.Variable`创建的变量受`name_scope`和`variable_scope`的效果，会给变量添加前缀，但是使用`tf.get_variable`创建变量只受`variable_scope`的效果。
- `name_scope`的主要作用就是：Tensorflow中常常会有数以千计的节点，在可视化的过程中很难一下子展示出来，因此用`name_scope`为变量划分范围，在可视化中，这表示在计算图中的一个层级。`name_scope`会影响`op_name`，不会影响用`get_variable()`创建的变量，而会影响通过`Variable()`创建的变量。

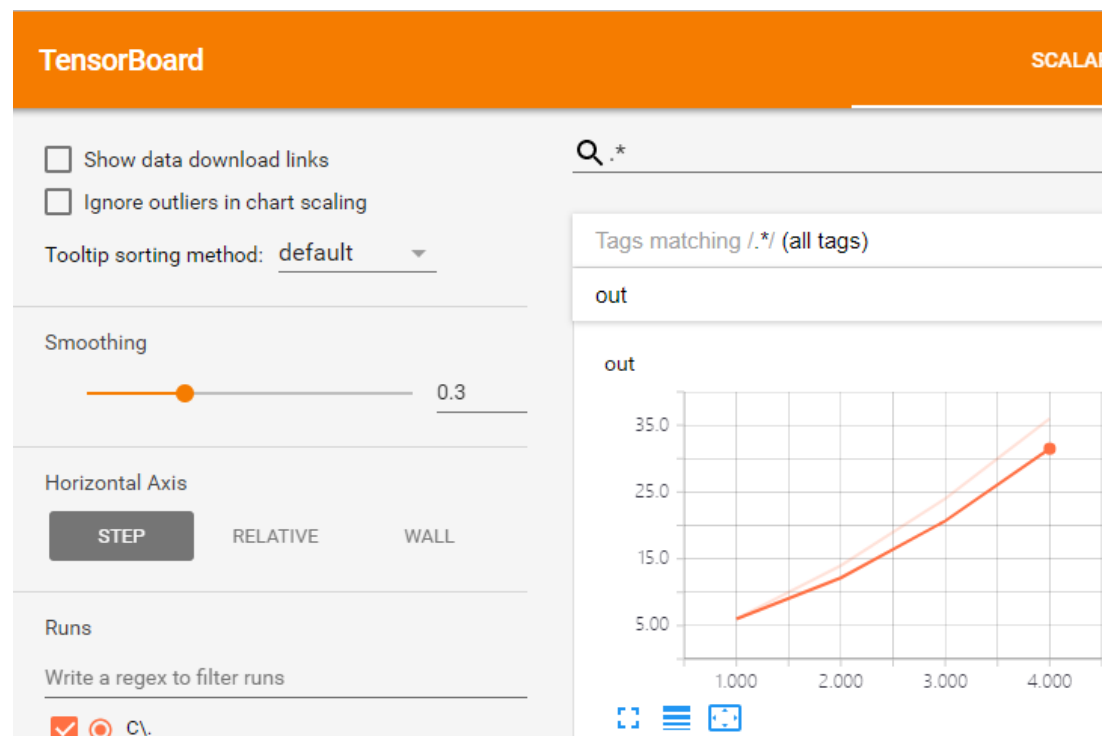
```
name1/variable1/v:0, [1.0]
variable1/w:0, [2.]
name1/variable1/add, [3.]
variable2/name2/v2:0, 2.0
variable2/w2:0, [2.]
variable2/name2/add, [4.]
```

```
with tf.Session() as sess:
 with tf.name_scope('name1'):
 with tf.variable_scope('variable1'):
 v = tf.Variable(1.0, name='v')
 w = tf.get_variable(name='w', shape=[1], initializer=tf.constant_initializer(2.0))
 h = v + w

 with tf.variable_scope('variable2'):
 with tf.name_scope('name2'):
 v2 = tf.Variable(2.0, name='v2')
 w2 = tf.get_variable(name='w2', shape=[1], initializer=tf.constant_initializer(2.0))
 h2 = v2 + w2
```

# TensorFlow可视化

- TensorFlow提供了一套可视化工具：TensorBoard，在通过pip安装TensorFlow的情况下，默认也会安装TensorBoard。通过TensorBoard可以展示TensorFlow的图像、绘制图像生成的定量指标以及附加数据等信息。



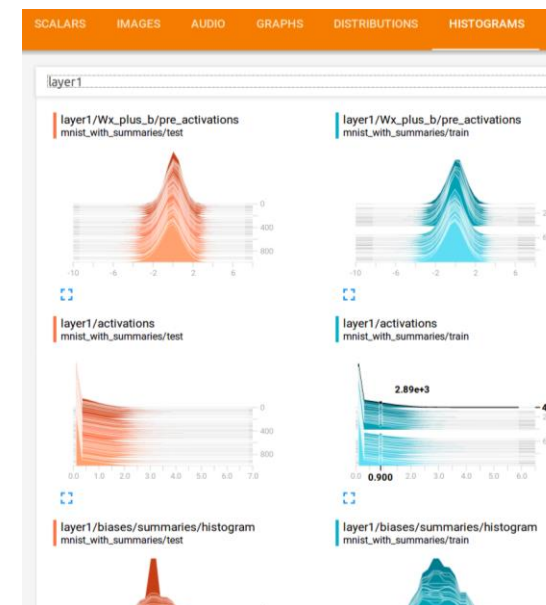
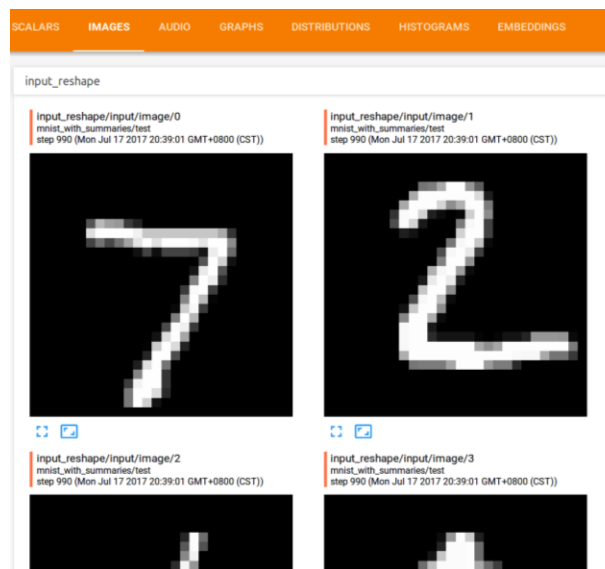
## TensorFlow可视化

- TensorBoard通过读取TensorFlow的事件文件来运行，TensorFlow的事件文件包括了在TensorFlow运行中涉及到的主要数据，比如：scalar、image、audio、histogram和graph等。
- 通过tf.summary相关API，将数据添加summary中，然后在Session中执行这些操作得到一个序列化Summary protobuf对象，然后使用FileWriter对象将汇总的序列数据写入到磁盘，然后使用tensorboard命令进行图标展示，默认访问端口是：6006
- TensorBoard中支持结构视图和设备视图。



# TensorFlow可视化

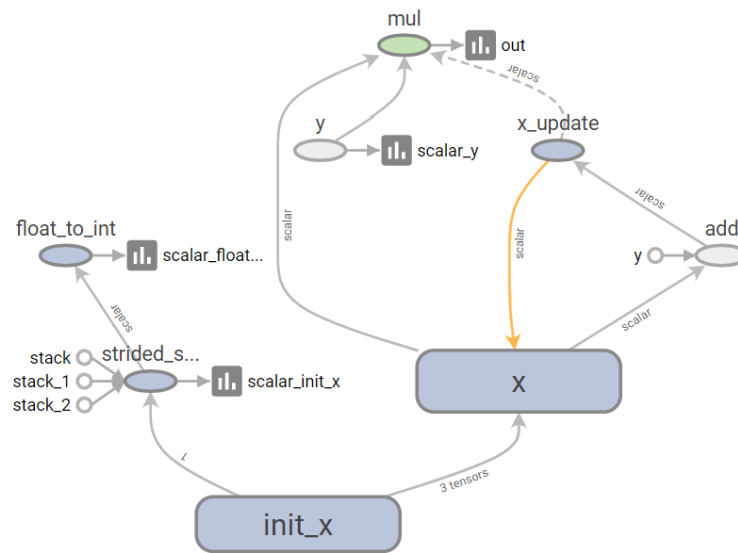
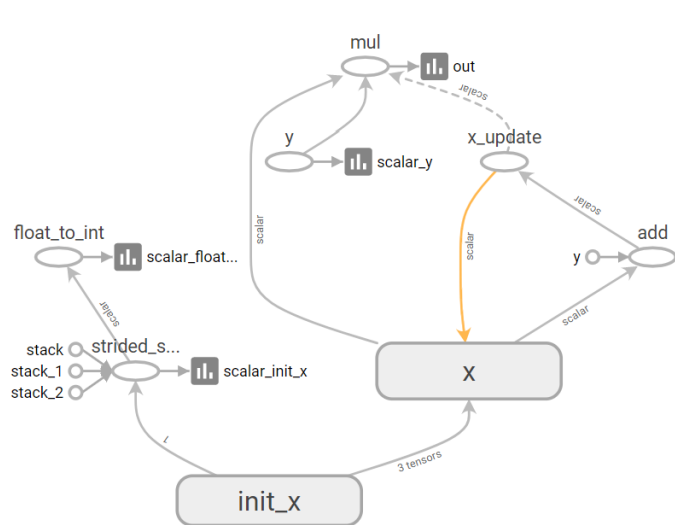
| API                  | 描述                  |
|----------------------|---------------------|
| tf.summary.scalar    | 添加一个标量              |
| tf.summary.audio     | 添加一个音频变量            |
| tf.summary.image     | 添加一个图片变量            |
| tf.summary.histogram | 添加一个直方图变量           |
| tf.summary.text      | 添加一个字符串类型的变量(一般很少用) |



# TensorFlow可视化

```
C:\Users\ibf>tensorboard --logdir C:\workspace\python\tensorflow_demo\result
TensorBoard 0.4.0 at http://DESKTOP-F32424:6006 (Press CTRL+C to quit)
```

- 案例：编写一个累加器；定义一个变量 $x$ (初值随机给定)，定义一个占位符 $y$ ，迭代4次，每个迭代返回 $x*y$ 的值，并且在计算 $x*y$ 乘积前，先对 $x$ 进行累加操作。并且将这个程序信息输出到文件以TensorBoard展示。




# TensorFlow可视化


| 符号                                                                                  | 意义                               |
|-------------------------------------------------------------------------------------|----------------------------------|
|    | High-level节点代表一个名称域，双击则展开一个高层节点。 |
|    | 彼此之间不连接的有限个节点序列。                 |
|    | 彼此之间相连的有限个节点序列。                  |
|    | 一个单独的操作节点。                       |
|    | 一个常量结点。                          |
|    | 一个摘要节点。                          |
|  | 显示各操作间的数据流边。                     |
|  | 显示各操作间的控制依赖边。                    |
|  | 引用边，表示出度操作节点可以使入度tensor发生变化。     |


**mul**  
Operation: Mul


**Attributes (1)**  
T {"type": "DT\_INT32"}  
Device /device:GPU:0

**Inputs (3)**  


 x/read scalar

 y

 Control dependencies

 x\_update

**Outputs (1)**  

 out

Remove from main graph

## TensorFlow线程和队列

- QueueRunner类可以创建一组线程，这些线程可以重复的执行enqueue操作，同时使用Coordinator来处理线程同步终止，此外QueueRunner内部维护了一个closer thread，当Coordinator收到异常报告时，这个线程会自动关闭队列。

```
Create a queue runner that will run 4 threads in parallel to enqueue
examples.
qr = tf.train.QueueRunner(queue, [enqueue_op] * 4)

Launch the graph.
sess = tf.Session()
Create a coordinator, launch the queue runner threads.
coord = tf.train.Coordinator()
enqueue_threads = qr.create_threads(sess, coord=coord, start=True)
Run the training loop, controlling termination with the coordinator.
for step in xrange(1000000):
 if coord.should_stop():
 break
 sess.run(train_op)
When done, ask the threads to stop.
coord.request_stop()
And wait for them to actually do it.
coord.join(threads)
```

# TensorFlow数据读取

- TensorFlow提供了三种方式来读取数据：
  - 供给数据(Feeding): 在TensorFlow程序运行的每一步, 让Python代码来供给数据
  - 从文件读取数据(深度学习中主要应用): 在TensorFlow图的起始, 让一个输入管线从文件中读取数据
  - 预加载数据: 在TensorFlow图中定义常量或变量来保存所有数据(仅适合小规模数据集的情况)

## TensorFlow模型保存、提取

- TensorFlow使用`tf.train.Saver`类实现模型的保存和提取。Saver对象的`save`方法将TensorFlow模型保存到指定路径中。
- 通过Saver对象的`restore`方法可以加载模型，并通过保存好的模型变量相关值重新加载完全加载进来。
- 如果不希望重复定义计算图上的运算，可以直接加载已经持久化的图，通过`tf.train.import_meta_graph`方法直接加载。
- 备注：在加载的时候，可以在Saver对象构建的时候，明确给定变量名之间的映射关系。

# 激活函数

- 激活函数运行时激活神经网络中某一部分神经元，将激活信息向后传入下一层的神经网络。神经网络之所以能解决非线性问题，本质上就是激活函数加入非线性因素，弥补了线性模型的表达力，把“激活的神经元的特征”通过函数保留并映射到下一层。
- 因为神经网络的数学基础是处处可微的，所以选取的激活函数要把保证数据输入与输出也是可微的。
- 常见的**激活函数**:
  - sigmoid
  - tanh
  - relu
  - dropout

## 分类函数

- 常见的分类函数：
  - `sigmoid_cross_entropy_with_logits`
  - `softmax_cross_entropy_with_logits`
  - `sparse_softmax_cross_entropy_with_logits`
  - `weighted_cross_entropy_with_logits`
  - `softmax`
  - `log_softmax`



# 优化方法

- 如何加速神经网络的训练呢？
- 目前加速训练的优化方法基本都基于梯度下降的。只是细节上有差异。
- 优化方法：
  - 梯度下降法 (BGD、SGD)
  - Adadelta
  - Adagrad (Adagrad、AdagradDAO)
  - Momentum (Momentum\Nesterov Momentum)
  - Adam
  - Ftrl
  - RMSprop

- 运行方法：
  - 1. 加载数据及定义超参数
  - 2. 构建网络
  - 3. 训练模型
  - 4. 评估模型和进行预测
- Linear线性回归案例
- Softmax回归案例（手写数字）

## TensorFlow常见API

- 不同版本之间的TensorFlow API的调用方式存在小的差别点，但是整体而言，API基本上没有太大的区别：
- 参考文档：[https://tensorflow.google.cn/versions/r1.4/api\\_docs/python/](https://tensorflow.google.cn/versions/r1.4/api_docs/python/)

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

# TensorFlow常见API

- 数据类型转换相关API

- Casting

- `tf.string_to_number(string_tensor, out_type=None, name=None)`
- `tf.to_double(x, name='ToDouble')`
- `tf.to_float(x, name='ToFloat')`
- `tf.to_bfloat16(x, name='ToBFloat16')`
- `tf.to_int32(x, name='ToInt32')`
- `tf.to_int64(x, name='ToInt64')`
- `tf.cast(x, dtype, name=None)`

# TensorFlow常见API

- Tensor Shape获取以及设置相关API

- Shapes and Shaping

- `tf.shape(input, name=None)`
- `tf.size(input, name=None)`
- `tf.rank(input, name=None)`
- `tf.reshape(tensor, shape, name=None)`
- `tf.squeeze(input, squeeze_dims=None, name=None)`
- `tf.expand_dims(input, dim, name=None)`

# TensorFlow常见API

- Tensor合并、分割相关API

- Slicing and Joining

- `tf.slice(input_, begin, size, name=None)`
    - `tf.split(split_dim, num_split, value, name='split')`
    - `tf.tile(input, multiples, name=None)`
    - `tf.pad(input, paddings, name=None)`
    - `tf.concat(concat_dim, values, name='concat')`
    - `tf.pack(values, name='pack')`
    - `tf.unpack(value, num=None, name='unpack')`
    - `tf.reverse_sequence(input, seq_lengths, seq_dim, name=None)`
    - `tf.reverse(tensor, dims, name=None)`
    - `tf.transpose(a, perm=None, name='transpose')`
    - `tf.gather(params, indices, name=None)`
    - `tf.dynamic_partition(data, partitions, num_partitions, name=None)`
    - `tf.dynamic_stitch(indices, data, name=None)`

# TensorFlow常见API

- Error相关类API

- Error classes

- `class tf.OpError`
- `class tf.errors.CancelledError`
- `class tf.errors.UnknownError`
- `class tf.errors.InvalidArgumentError`
- `class tf.errors.DeadlineExceededError`
- `class tf.errors.NotFoundError`
- `class tf.errors.AlreadyExistsError`
- `class tf.errors.PermissionDeniedError`
- `class tf.errors.UnauthenticatedError`
- `class tf.errors.ResourceExhaustedError`
- `class tf.errors.FailedPreconditionError`
- `class tf.errors.AbortedError`
- `class tf.errors.OutOfRangeError`
- `class tf.errors.UnimplementedError`
- `class tf.errors.InternalError`
- `class tf.errors.UnavailableError`
- `class tf.errors.DataLossError`

# TensorFlow常见API

- 常量类型的Tensor对象相关API

- Constant Value Tensors

- `tf.zeros(shape, dtype=tf.float32, name=None)`
    - `tf.zeros_like(tensor, dtype=None, name=None)`
    - `tf.ones(shape, dtype=tf.float32, name=None)`
    - `tf.ones_like(tensor, dtype=None, name=None)`
    - `tf.fill(dims, value, name=None)`
    - `tf.constant(value, dtype=None, shape=None, name='Const')`



## *TensorFlow*中的条件判断

- `tf.cond`

```
import tensorflow as tf
```

```
a=tf.constant(2)
```

```
b=tf.constant(3)
```

```
x=tf.constant(4)
```

```
y=tf.constant(5)
```

```
z = tf.multiply(a, b)
```

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

```
with tf.Session() as session:
```

```
 print(result.eval())
```

## TensorFlow中的条件判断

- tf.case

```
import tensorflow as tf
def f1():
 return tf.constant(17)
def f2():
 return tf.constant(23)
def f3():
 return tf.constant(-1)
x = 2
y = 3
r = tf.case({tf.less(x, y): f1, tf.greater(x, y): f2}, default=f3, exclusive=True)
sess = tf.Session()
print(sess.run(r))#17
```

# TensorFlow中的循环

```
loop = []
while cond(loop):
 loop = body(loop)
```

```
import tensorflow as tf
a = tf.constant(10)
n = tf.constant(10)
def cond(a, n):
 return a < n
def body(a, n):
 a = a + 1
 return a, n
a, n = tf.nn.loop(cond, body, [a, n])
with tf.Session() as sess:
 tf.global_variables_initializer().run()
 res = sess.run([a, n])
print(res)
```

```
i = 0
n = 10
while(i < n):
 i = i + 1
```

# 作业：用tf实现下列代码

```
a= 0
b =a*a
c=0
if b>4:
 n=10
else:
 n=4
while(c< n):
 c = c +1
```

# TensorFlow常见API

- 基础数学运算相关API

- Basic Math Functions

- `tf.add_n(inputs, name=None)`
- `tf.abs(x, name=None)`
- `tf.neg(x, name=None)`
- `tf.sign(x, name=None)`
- `tf.inv(x, name=None)`
- `tf.square(x, name=None)`
- `tf.round(x, name=None)`
- `tf.sqrt(x, name=None)`
- `tf.rsqrt(x, name=None)`
- `tf.pow(x, y, name=None)`
- `tf.exp(x, name=None)`
- `tf.log(x, name=None)`
- `tf.ceil(x, name=None)`
- `tf.floor(x, name=None)`
- `tf.maximum(x, y, name=None)`
- `tf.minimum(x, y, name=None)`
- `tf.cos(x, name=None)`
- `tf.sin(x, name=None)`

# TensorFlow常见API

- 矩阵乘法相关API

- Matrix Math Functions

- `tf.diag(diagonal, name=None)`
    - `tf.transpose(a, perm=None, name='transpose')`
    - `tf.matmul(a, b, transpose_a=False, transpose_b=False, a_is_sparse=False, b_is_sparse=False, name=None)`
    - `tf.batch_matmul(x, y, adj_x=None, adj_y=None, name=None)`
    - `tf.matrix_determinant(input, name=None)`
    - `tf.batch_matrix_determinant(input, name=None)`
    - `tf.matrix_inverse(input, name=None)`
    - `tf.batch_matrix_inverse(input, name=None)`
    - `tf.cholesky(input, name=None)`
    - `tf.batch_cholesky(input, name=None)`



谢谢