

目 录

致谢

介绍

起步

介绍

下载及安装

基本用法

教程

总览

MNIST 机器学习入门

深入 MNIST

TensorFlow 运作方式入门

卷积神经网络

字词的向量表示

递归神经网络

曼德布洛特(Mandelbrot)集合

偏微分方程

MNIST数据下载

运作方式

总览

变量:创建、初始化、保存和加载

TensorBoard:可视化学习

TensorBoard:图表可视化

读取数据

线程和队列

添加新的Op

自定义数据读取

使用gpu

共享变量

资源

总览

BibTex 引用

示例使用

FAQ

[术语表](#)

[Tensor排名、形状和类型](#)

[其他](#)

[常见问题汇总](#)

[相关资源](#)

[个人学习心得](#)

致谢

当前文档《TensorFlow 官方文档中文版》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-25。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/TensorFlow>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

- [TensorFlow 官方文档中文版](#)
 - 你正在阅读的项目可能会比 [Android](#) 系统更加深远地影响着世界！
- [缘起](#)
- [内容来源](#)
- [参与者（按认领章节排序）](#)
 - [翻译](#)
 - [校对](#)
- [进度记录](#)
- [花絮](#)
- [持续改进](#)
- [感谢支持](#)
- [离线版本](#)
- [Tex-PDF 修订版](#)

TensorFlow 官方文档中文版



你正在阅读的项目可能会比 [Android](#) 系统更加深远地影响着世界！

缘起

2015年11月9日，Google发布人工智能系统TensorFlow并宣布开源，同日，极客学院组织在线TensorFlow中文文档翻译。

机器学习作为人工智能的一种类型，可以让软件根据大量的数据来对未来的情况进行阐述或预判。如今，领先的科技巨头无不在机器学习下予以极大投入。Facebook、苹果、微软，甚至国内的百度。Google 自然也在其中。「TensorFlow」是 Google 多年以来内部的机器学习系统。如今，Google 正在将此系统成为开源系统，并将此系统的参数公布给业界工程师、学者和拥有大量编程能力的技术人员，这意味着什么呢？

打个不太恰当的比喻，如今 Google 对待 TensorFlow 系统，有点类似于该公司对待旗下移动操作系统 Android。如果更多的数据科学家开始使用 Google 的系统来从事机器学习方面的研究，那么这有利于 Google 对日益发展的机器学习行业拥有更多的主导权。

为了让国内的技术人员在最短的时间内迅速掌握这一世界领先的 AI 系统，极客学院 Wiki 团队发起对 TensorFlow 官方文档的中文协同翻译，一周之内，全部翻译认领完成，一个月后，全部30章节翻译校对完成，上线极客学院Wiki平台并提供下载。

Google TensorFlow项目负责人Jeff Dean为该中文翻译项目回信称：“看到能够将TensorFlow翻译成中文我非常激动，我们将TensorFlow开源的主要原因之一是为了让全世界的人们能够从机器学习与人工智能中获益，类似这样的协作翻译能够让更多的人更容易地接触到TensorFlow项目，很期待接下来该项目在全球范围内的应用！”

Jeff回信原文：



Jeff Dean

发送至 我 ▾

12月10日 (6天前) ☆



英文 ▾

>

中文 ▾

翻译邮件

对英文停用 ×

Hi Floyd,

Sure, I'm very excited by your project. Thanks for organizing these efforts. Feel free to use the quote below, if that would work:

"

I'm very excited to see the efforts of the Mandarin translation project for TensorFlow at <https://github.com/jikexueyuanwiki/tensorflow-zh>. One of the main reasons we released TensorFlow as an open-source project was to allow people all over the world to benefit from the power of machine learning and artificial intelligence, and projects like this translation effort make the project accessible to more people and developers. I'm very excited to see how TensorFlow is used all over the world!

Jeff Dean

Google Senior Fellow

<http://research.google.com/people/jeff>

"



再次衷心感谢每一位为该翻译项目做出贡献的同学，我们会持续关注TensorFlow、AI领域以及其它最新技术的发展、持续维护该协作翻译、持续提供更多更优质的内容，为广大IT学习者服务！

内容来源

英文官方网站：

<http://tensorflow.org/>

官方GitHub仓库：

<https://github.com/tensorflow/tensorflow>

中文版 GitHub 仓库：

<https://github.com/jikexueyuanwiki/tensorflow-zh>

参与者（按认领章节排序）

翻译

- @PFZheng
- @Tony Jin
- @chenweican
- @bingjin
- @oskycar
- @btpeter
- @War1n
- @ericxk
- @wangaicc
- @Terence Cooper
- @zhyhooo
- @thylaco1eo
- @volvet
- @zhangkom
- @derekshang
- @lianghyv
- @nb312
- @Jim-Zenn
- @andyiac
- @Terence Cooper
- @leege100

校对

- @yangtze
- @ericxk
- @HongyangWang
- @LichAmnesia
- @zhyhooo
- @waiwaizheng
- @HongyangWang

- @tensorflowly
- @lonlonago
- @jishaoming
- @lucky521
- @allensummer
- @volvet
- @ZHNathanielLee
- @pengfoo
- @qiaohaijun
- @Seika

进度记录

- 2015-11-10, 谷歌发布全新人工智能系统TensorFlow并宣布开源, 极客学院Wiki启动协同翻译, 创建 GitHub 仓库, 制定协同规范
- 2015-11-18, 所有章节认领完毕, 翻译完成18章, 校对认领7章, Star数361, fork数100, 协同翻译QQ群及技术交流群的TF爱好者将近300人, GitHub搜索TensorFlow排名第二
- 2015-12-10, Star数超过500
- 2015-12-15, 项目正式上线

花絮

在组织翻译的过程中, 有些事情令人印象深刻, 记录下来, 希望以后来学习文档的同学能够明了到手中这份文档的由来:

- 参加翻译的有学生, 也有老师; 有专门研究AI/ML的, 也有对此感兴趣的; 有国内的, 也有远在美国的; 有工程技术人员也有博士、专家
- 其中一位, 恩泽同学, 为了翻译一篇文档, 在前一天没有睡觉的情况下坚持翻完, 20个小时没有合眼
- 还有一位老师, 刚从讲台上讲完课, 就立即给我们的翻译提修改意见
- 很多同学自发的将搭建环境中遇到的问题总结到FAQ里帮助他人
- 为了一个翻译细节, 经常是来回几次, 和其他人讨论完善

持续改进

这样的一个小高技术领域文档, 我们在翻译的过程中, 难免会有不完善的地方, 希望请大家一起帮助我们持续改进文档的翻译质量, 帮助更多的人, 方法:

- 在GitHub上提Issue或Pull Request, 地址为:
<https://github.com/jikexueyuanwiki/tensorflow-zh>

- 加入我们的QQ群提建议-协同翻译群：248320884，技术交流群：551830261
- 给我们写邮件： wiki@jikexueyuan.com

感谢支持

- [极客学院 Wiki](#) 提供图文教程托管服务

离线版本

目前，离线版本(PDF、ePub)可正常下载、使用

Tex-PDF 修订版

[Tex-PDF 修订版](#) 目前正在编订中，欢迎加入进来一起修订。您可以在[此查看预览版](#)目前最新状态。

起步

起步

- [介绍](#)
- [下载及安装](#)
- [基本用法](#)

介绍

- [简介](#)

简介

本章的目的是让你了解和运行 TensorFlow!

在开始之前, 让我们先看一段使用 Python API 撰写的 TensorFlow 示例代码, 让你对将要学习的内容有初步的印象.

这段很短的 Python 程序生成了一些三维数据, 然后用一个平面拟合它.

```
1. import tensorflow as tf
2. import numpy as np
3.
4. # 使用 NumPy 生成假数据(phony data), 总共 100 个点.
5. x_data = np.float32(np.random.rand(2, 100)) # 随机输入
6. y_data = np.dot([0.100, 0.200], x_data) + 0.300
7.
8. # 构造一个线性模型
9. #
10. b = tf.Variable(tf.zeros([1]))
11. W = tf.Variable(tf.random_uniform([1, 2], -1.0, 1.0))
12. y = tf.matmul(W, x_data) + b
13.
14. # 最小化方差
15. loss = tf.reduce_mean(tf.square(y - y_data))
16. optimizer = tf.train.GradientDescentOptimizer(0.5)
17. train = optimizer.minimize(loss)
18.
19. # 初始化变量
20. init = tf.initialize_all_variables()
21.
22. # 启动图 (graph)
23. sess = tf.Session()
24. sess.run(init)
25.
26. # 拟合平面
27. for step in xrange(0, 201):
28.     sess.run(train)
29.     if step % 20 == 0:
30.         print step, sess.run(W), sess.run(b)
31.
32. # 得到最佳拟合结果 W: [[0.100 0.200]], b: [0.300]
```

为了进一步激发你的学习欲望，我们想让你先看一下 TensorFlow 是如何解决一个经典的机器学习问题的。在神经网络领域，最为经典的问题莫过于 MNIST 手写数字分类问题。我们准备了两篇不同的教程，分别面向机器学习领域的初学者和专家。如果你已经使用其它软件训练过许多 MNIST 模型，请阅读高级教程（红色药丸链接）。如果你以前从未听说过 MNIST，请阅读初级教程（蓝色药丸链接）。如果你的水平介于这两类人之间，我们建议你先快速浏览初级教程，然后再阅读高级教程。



图片由 CC BY-SA 4.0 授权；原作者 W. Carter

如果你已经下定决心，准备学习和安装 TensorFlow，你可以略过这些文字，直接阅读后面的章节。不用担心，你仍然会看到 MNIST — 在阐述 TensorFlow 的特性时，我们还会使用 MNIST 作为一个样例。

推荐随后阅读

[下载与安装 基本使用](#)

* [TensorFlow 技术指南](#)

原文: [Introduction](#) 翻译: [@doc001](#) 校对: [@yangtze](#)

下载及安装

- [下载与安装](#)
 - [Pip 安装](#)
 - [基于 Docker 的安装](#)
 - [其它镜像](#)
 - [基于 VirtualEnv 的安装](#)
 - [基于 Anaconda 的安装](#)
 - [尝试你的第一个 TensorFlow 程序](#)
 - [\(可选\) 启用 GPU 支持](#)
 - [运行 TensorFlow](#)
 - [从源码安装](#)
 - [克隆 TensorFlow 仓库](#)
 - [Linux 安装](#)
 - [安装 Bazel](#)
 - [安装其他依赖](#)
 - [可选: 安装 CUDA \(在 Linux 上开启 GPU 支持\)](#)
 - [下载并安装 Cuda Toolkit 7.0](#)
 - [下载并安装 CUDNN Toolkit 6.5](#)
 - [配置 TensorFlow 的 Cuda 选项](#)
 - [编译目标程序, 开启 GPU 支持](#)
 - [已知问题](#)
 - [Mac OS X 安装](#)
 - [Bazel](#)
 - [SWIG](#)
 - [Numpy](#)
 - [创建 pip 包并安装](#)
 - [训练你的第一个 TensorFlow 神经网络模型](#)
 - [常见问题](#)
 - [GPU 相关问题](#)
 - [在 Linux 上](#)
 - [在 Mac OS X 上](#)

下载与安装

你可以使用我们提供的 [Pip](#), [Docker](#), [Virtualenv](#), [Anaconda](#) 或 [源码编译](#) 的方法安装 TensorFlow.

Pip 安装

[Pip](#)) 是一个 Python 的软件包安装与管理工具。

在安装 TensorFlow 过程中要涉及安装或升级的包详见 [列表](#)

首先安装 pip (或 Python3 的 pip3):

```
1. # Ubuntu/Linux 64-bit
2. $ sudo apt-get install python-pip python-dev
3.
4. # Mac OS X
5. $ sudo easy_install pip
```

安装 TensorFlow :

```
1. # Ubuntu/Linux 64-bit, CPU only, Python 2.7:
2. $ sudo pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0-cp27-none-linux_x86_64.whl
3.
4. # Ubuntu/Linux 64-bit, GPU enabled, Python 2.7. Requires CUDA toolkit 7.5 and CuDNN v4.
5. # For other versions, see "Install from sources" below.
6. $ sudo pip install --upgrade https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.8.0-cp27-none-linux_x86_64.whl
7.
8. # Mac OS X, CPU only:
9. $ sudo easy_install --upgrade six
10. $ sudo pip install --upgrade https://storage.googleapis.com/tensorflow/mac/tensorflow-0.8.0-py2-none-any.whl
```

如果是 Python3 :

```
1. # Ubuntu/Linux 64-bit, CPU only, Python 3.4:
2. $ sudo pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0-cp34-cp34m-linux_x86_64.whl
3.
4. # Ubuntu/Linux 64-bit, GPU enabled, Python 3.4. Requires CUDA toolkit 7.5 and CuDNN v4.
5. # For other versions, see "Install from sources" below.
6. $ sudo pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.8.0-cp34-cp34m-linux_x86_64.whl
7.
8. # Mac OS X, CPU only:
9. $ sudo easy_install --upgrade six
10. $ sudo pip3 install --upgrade https://storage.googleapis.com/tensorflow/mac/tensorflow-0.8.0-py3-none-any.whl
```

备注: 如果之前安装过 TensorFlow < 0.7.1 的版本, 应该先使用 `pip uninstall` 卸载 TensorFlow 和 protobuf , 保证获取的是一个最新 protobuf 依赖下的安装包。

之后可以[测试](#)一下。

基于 Docker 的安装

我们也支持通过 [Docker](#) 运行 TensorFlow。

该方式的优点是不用操心软件依赖问题。

首先，[安装 Docker](#)。一旦 Docker 已经启动运行，可以通过命令启动一个容器：

```
1. $ docker run -it b.gcr.io/tensorflow/tensorflow
```

该命令将启动一个已经安装好 TensorFlow 及相关依赖的容器。

其它镜像

默认的 Docker 镜像只包含启动和运行 TensorFlow 所需依赖库的一个最小集。我们额外提供了下面的容器，该容器同样可以通过上述 `docker run` 命令安装：

- `b.gcr.io/tensorflow/tensorflow-full`：镜像中的 TensorFlow 是从源代码完整安装的，包含了编译和运行 TensorFlow 所需的全部工具。在该镜像上，可以直接使用源代码进行实验，而不再需要再安装上述的任何依赖。

基于 VirtualEnv 的安装

我们推荐使用 [virtualenv](#) 创建一个隔离的容器，来安装 TensorFlow。这是可选的，但是这样做能使排查安装问题变得更容易。

首先，安装所有必备工具：

```
1. # 在 Linux 上:
2. $ sudo apt-get install python-pip python-dev python-virtualenv
3.
4. # 在 Mac 上:
5. $ sudo easy_install pip # 如果还没有安装 pip
6. $ sudo pip install --upgrade virtualenv
```

接下来，建立一个全新的 virtualenv 环境。为了将环境建在 `~/tensorflow` 目录下，执行：

```
1. $ virtualenv --system-site-packages ~/tensorflow
2. $ cd ~/tensorflow
```

然后, 激活 virtualenv:

```
1. $ source bin/activate # 如果使用 bash
2. $ source bin/activate.csh # 如果使用 csh
3. (tensorflow)$ # 终端提示符应该发生变化
```

在 virtualenv 内, 安装 TensorFlow:

```
1. (tensorflow)$ pip install --upgrade <$url_to_binary.whl>
```

接下来, 使用类似命令运行 TensorFlow 程序:

```
1. (tensorflow)$ cd tensorflow/models/image/mnist
2. (tensorflow)$ python convolutional.py
3.
4. # 当使用完 TensorFlow
5. (tensorflow)$ deactivate # 停用 virtualenv
6.
7. $ # 你的命令提示符会恢复原样
```

基于 Anaconda 的安装

[Anaconda](#) 是一个集成许多第三方科学计算库的 Python 科学计算环境, Anaconda 使用 conda 作为自己的包管理工具, 同时具有自己的[计算环境](#), 类似 Virtualenv.

和 Virtualenv 一样, 不同 Python 工程需要的依赖包, conda 将他们存储在不同的地方。TensorFlow 上安装的 Anaconda 不会对之前安装的 Python 包进行覆盖.

- 安装 Anaconda
- 建立一个 conda 计算环境
- 激活环境, 使用 conda 安装 TensorFlow
- 安装成功后, 每次使用 TensorFlow 的时候需要激活 conda 环境

安装 Anaconda :

参考 Anaconda 的下载页面的[指导](#)

建立一个 conda 计算环境名字叫 `tensorflow` :

```

1. # Python 2.7
2. $ conda create -n tensorflow python=2.7
3.
4. # Python 3.4
5. $ conda create -n tensorflow python=3.4

```

激活 `tensorflow` 环境, 然后使用其中的 `pip` 安装 TensorFlow. 当使用 `easy_install` 使用 `--ignore-installed` 标记防止错误的产生。

```

1. $ source activate tensorflow
2. (tensorflow)$ # Your prompt should change
3.
4. # Ubuntu/Linux 64-bit, CPU only, Python 2.7:
5. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0rc0-cp27-none-
   linux_x86_64.whl
6.
7. # Ubuntu/Linux 64-bit, GPU enabled, Python 2.7. Requires CUDA toolkit 7.5 and CuDNN v4.
8. # For other versions, see "Install from sources" below.
9. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.8.0rc0-cp27-none-
   linux_x86_64.whl
10.
11. # Mac OS X, CPU only:
12. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/mac/tensorflow-0.8.0rc0-py2-none-any.whl

```

对于 Python 3.x :

```

1. $ source activate tensorflow
2. (tensorflow)$ # Your prompt should change
3.
4. # Ubuntu/Linux 64-bit, CPU only, Python 3.4:
5. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0rc0-cp34-cp34m-
   linux_x86_64.whl
6.
7. # Ubuntu/Linux 64-bit, GPU enabled, Python 3.4. Requires CUDA toolkit 7.5 and CuDNN v4.
8. # For other versions, see "Install from sources" below.
9. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.8.0rc0-cp34-cp34m-
   linux_x86_64.whl
10.
11. # Mac OS X, CPU only:
12. (tensorflow)$ pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/mac/tensorflow-0.8.0rc0-py3-none-any.whl

```

conda 环境激活后, 你可以[测试](#)

当你不用 TensorFlow 的时候, 关闭环境:

```
1. (tensorflow)$ source deactivate
2.
3. $ # Your prompt should change back
```

再次使用的时候再激活 :-)

```
1. $ source activate tensorflow
2. (tensorflow)$ # Your prompt should change.
3. # Run Python programs that use TensorFlow.
4. ...
5. # When you are done using TensorFlow, deactivate the environment.
6. (tensorflow)$ source deactivate
```

尝试你的第一个 TensorFlow 程序

(可选) 启用 GPU 支持

如果你使用 pip 二进制包安装了开启 GPU 支持的 TensorFlow, 你必须确保系统里安装了正确的 CUDA sdk 和 CUDNN 版本. 请参阅 [CUDA 安装教程](#)

你还需要设置 `LD_LIBRARY_PATH` 和 `CUDA_HOME` 环境变量. 可以考虑将下面的命令添加到 `~/.bash_profile` 文件中, 这样每次登陆后自动生效. 注意, 下面的命令假定 CUDA 安装目录为 `/usr/local/cuda` :

```
1. export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64"
2. export CUDA_HOME=/usr/local/cuda
```

运行 TensorFlow

打开一个 python 终端:

```
1. $ python
2.
3. >>> import tensorflow as tf
4. >>> hello = tf.constant('Hello, TensorFlow!')
5. >>> sess = tf.Session()
6. >>> print sess.run(hello)
7. Hello, TensorFlow!
8. >>> a = tf.constant(10)
9. >>> b = tf.constant(32)
```

```
10. >>> print sess.run(a+b)
11. 42
12. >>>
```

从源码安装

克隆 TensorFlow 仓库

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

`--recurse-submodules` 参数是必须得，用于获取 TensorFlow 依赖的 protobuf 库。

Linux 安装

安装 Bazel

首先依照 [教程](#) 安装 Bazel 的依赖。

然后在 [链接](#) 中下载适合你的操作系统的最新稳定版，
最后按照下面脚本执行：

```
1. $ chmod +x PATH_TO_INSTALL.SH
2. $ ./PATH_TO_INSTALL.SH --user
```

注意把 `PATH_TO_INSTALL.SH` 替换为你下载的安装包的文件路径。

将执行路径 `output/bazel` 添加到 `$PATH` 环境变量中。

安装其他依赖

```
1. # For Python 2.7:
2. $ sudo apt-get install python-numpy swig python-dev python-wheel
3. # For Python 3.x:
4. $ sudo apt-get install python3-numpy swig python3-dev python3-wheel
```

可选：安装 CUDA（在 Linux 上开启 GPU 支持）

为了编译并运行能够使用 GPU 的 TensorFlow，需要先安装 NVIDIA 提供的 Cuda Toolkit 7.0 和 CUDNN 6.5 V2。

TensorFlow 的 GPU 特性只支持 NVidia Compute Capability ≥ 3.5 的显卡。被支持的显卡

包括但不限于：

- NVidia Titan
- NVidia Titan X
- NVidia K20
- NVidia K40

下载并安装 Cuda Toolkit 7.0

[下载地址](#)

将工具安装到诸如 `/usr/local/cuda` 之类的路径。

下载并安装 CUDNN Toolkit 6.5

[下载地址](#)

解压并拷贝 CUDNN 文件到 Cuda Toolkit 7.0 安装路径下。假设 Cuda Toolkit 7.0 安装在 `/usr/local/cuda`，执行以下命令：

```
1. tar xvzf cudnn-6.5-linux-x64-v2.tgz
2. sudo cp cudnn-6.5-linux-x64-v2/cudnn.h /usr/local/cuda/include
3. sudo cp cudnn-6.5-linux-x64-v2/libcudnn* /usr/local/cuda/lib64
```

配置 TensorFlow 的 Cuda 选项

从源码树的根路径执行：

```
1. $ ./configure
2. Do you wish to build TensorFlow with GPU support? [y/n] y
3. GPU support will be enabled for TensorFlow
4.
5. Please specify the location where CUDA 7.0 toolkit is installed. Refer to
6. README.md for more details. [default is: /usr/local/cuda]: /usr/local/cuda
7.
8. Please specify the location where CUDNN 6.5 V2 library is installed. Refer to
9. README.md for more details. [default is: /usr/local/cuda]: /usr/local/cuda
10.
11. Setting up Cuda include
12. Setting up Cuda lib64
13. Setting up Cuda bin
14. Setting up Cuda nvvm
15. Configuration finished
```

这些配置将建立到系统 Cuda 库的符号链接。每当 Cuda 库的路径发生变更时，必须重新执行上述步骤，否则无法调用 bazel 编译命令。

编译目标程序，开启 GPU 支持

从源码树的根路径执行：

```
1. $ bazel build -c opt --config=cuda //tensorflow/cc:tutorials_example_trainer
2.
3. $ bazel-bin/tensorflow/cc/tutorials_example_trainer --use_gpu
4. # 大量的输出信息。这个例子用 GPU 迭代计算一个 2x2 矩阵的主特征值 (major eigenvalue)。
5. # 最后几行输出和下面的信息类似。
6. 000009/000005 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
7. 000006/000001 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
8. 000009/000009 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
```

注意，GPU 支持需通过编译选项 “--config=cuda” 开启。

已知问题

- 尽管可以在同一个源码树下编译开启 Cuda 支持和禁用 Cuda 支持的版本，我们还是推荐在切换这两种不同的编译配置时，使用 “bazel clean” 清理环境。
- 在执行 bazel 编译前必须先运行 configure，否则编译会失败并提示错误信息。未来，我们可能考虑将 configure 步骤包含在编译过程中，以简化整个过程，前提是 bazel 能够提供新的特性支持这样。

Mac OS X 安装

Mac 和 Linux 需要的软件依赖完全一样，但是安装过程区别很大。以下链接用于帮助你在 Mac OS X 上安装这些依赖：

Bazel

参见[本网页](#)的 Mac OS X 安装指南。

SWIG

[Mac OS X 安装教程](#)。

注意：你需要安装[PCRE](#)，
而不是 PCRE2。

Numpy

参见[安装教程](#)。

创建 pip 包并安装

```
1. $ bazel build -c opt //tensorflow/tools/pip_package:build_pip_package
2.
3. $ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
4.
5. # .whl 文件的实际名字与你所使用的平台有关
6. $ pip install /tmp/tensorflow_pkg/tensorflow-0.5.0-cp27-none-linux_x86_64.whl
```

训练你的第一个 TensorFlow 神经网络模型

从源代码树的根路径执行：

```
1. $ cd tensorflow/models/image/mnist
2. $ python convolutional.py
3. Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
4. Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
5. Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
6. Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
7. Extracting data/train-images-idx3-ubyte.gz
8. Extracting data/train-labels-idx1-ubyte.gz
9. Extracting data/t10k-images-idx3-ubyte.gz
10. Extracting data/t10k-labels-idx1-ubyte.gz
11. Initialized!
12. Epoch 0.00
13. Minibatch loss: 12.054, learning rate: 0.010000
14. Minibatch error: 90.6%
15. Validation error: 84.6%
16. Epoch 0.12
17. Minibatch loss: 3.285, learning rate: 0.010000
18. Minibatch error: 6.2%
19. Validation error: 7.0%
20. ...
21. ...
```

常见问题

GPU 相关问题

如果在尝试运行一个 TensorFlow 程序时出现以下错误：

```
1. ImportError: libcudart.so.7.0: cannot open shared object file: No such file or directory
```

请确认你正确安装了 GPU 支持, 参见 [相关章节](#)。

在 Linux 上

如果出现错误:

```
1. ...
2.  "__add__", "__radd__",
3.         ^
4. SyntaxError: invalid syntax
```

解决方案: 确认正在使用的 Python 版本为 Python 2.7.

在 Mac OS X 上

如果出现错误:

```
1. import six.moves.copyreg as copyreg
2.
3. ImportError: No module named copyreg
```

解决方案: TensorFlow 使用的 protobuf 依赖 `six-1.10.0`。但是, Apple 的默认 python 环境

已经安装了 `six-1.4.1`, 该版本可能很难升级。这里提供几种方法来解决该问题:

1. 升级全系统的 `six`:

```
1. sudo easy_install -U six
```

2. 通过 homebrew 安装一个隔离的 python 副本:

```
1. brew install python
```

3. 在 `virtualenv` 内编译或使用 TensorFlow.

如果出现错误:

```
1. >>> import tensorflow as tf
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
```

```

4.   File "/usr/local/lib/python2.7/site-packages/tensorflow/__init__.py", line 4, in <module>
5.       from tensorflow.python import *
6.   File "/usr/local/lib/python2.7/site-packages/tensorflow/python/__init__.py", line 13, in
    <module>
7.       from tensorflow.core.framework.graph_pb2 import *
8.   ...
9.   File "/usr/local/lib/python2.7/site-packages/tensorflow/core/framework/tensor_shape_pb2.py",
    line 22, in <module>
10.
    serialized_pb=_b('\n\tensorflow/core/framework/tensor_shape.proto\x12\tensorflow\"d\n\x10TensorSh
    \n\x03\x64im\x18\x02 \x03(\x0b\x32
    .tensorflow.TensorShapeProto.Dim\x1a!\n\x03\x44im\x12\x0c\n\x04size\x18\x01
    \x01(\x03\x12\x0c\n\x04name\x18\x02 \x01(\tb\x06proto3')
11. TypeError: __init__() got an unexpected keyword argument 'syntax'

```

这是由于安装了冲突的 `protobuf` 版本引起的，TensorFlow 需要的是 `protobuf 3.0.0`。当前最好的解决方案是确保没有安装旧版本的 `protobuf`，可以使用以下命令重新安装 `protobuf` 来解决冲突：

```
1. brew reinstall --devel protobuf
```

原文: [Download and Setup](#) 翻译: @doc001 校对: @yangtze

基本用法

- [基本使用](#)
 - [综述](#)
 - [计算图](#)
 - [构建图](#)
 - [在一个会话中启动图](#)
 - [交互式使用](#)
 - [Tensor](#)
 - [变量](#)
 - [Fetch](#)
 - [Feed](#)

基本使用

使用 TensorFlow, 你必须明白 TensorFlow:

- 使用图 (graph) 来表示计算任务.
- 在被称之为 `会话 (Session)` 的上下文 (context) 中执行图.
- 使用 `tensor` 表示数据.
- 通过 `变量 (Variable)` 维护状态.
- 使用 `feed` 和 `fetch` 可以为任意的操作(arbitrary operation) 赋值或者从其中获取数据.

综述

TensorFlow 是一个编程系统, 使用图来表示计算任务. 图中的节点被称之为 *op* (operation 的缩写). 一个 *op* 获得 0 个或多个 `Tensor`, 执行计算, 产生 0 个或多个 `Tensor`. 每个 `Tensor` 是一个类型化的多维数组. 例如, 你可以将一小组图像集表示为一个四维浮点数数组, 这四个维度分别是 `[batch, height, width, channels]`.

一个 TensorFlow 图描述了计算的过程. 为了进行计算, 图必须在 `会话` 里被启动. `会话` 将图的 *op* 分发到诸如 CPU 或 GPU 之类的 `设备` 上, 同时提供执行 *op* 的方法. 这些方法执行后, 将产生的 *tensor* 返回. 在 Python 语言中, 返回的 *tensor* 是 `numpy ndarray` 对象; 在 C 和 C++ 语言中, 返回的 *tensor* 是 `tensorflow::Tensor` 实例.

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。

例如，通常在构建阶段创建一个图来表示和训练神经网络，然后在执行阶段反复执行图中的训练 op。

TensorFlow 支持 C, C++, Python 编程语言。目前，TensorFlow 的 Python 库更加易用，它提供了大量的辅助函数来简化构建图的工作，这些函数尚未被 C 和 C++ 库支持。

三种语言的会话库 (session libraries) 是一致的。

构建图

构建图的第一步，是创建源 op (source op)。源 op 不需要任何输入，例如 `tf.constant()`。源 op 的输出被传递给其它 op 做运算。

Python 库中，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。

TensorFlow Python 库有一个默认图 (*default graph*)，op 构造器可以为其增加节点。这个默认图对

许多程序来说已经足够用了。阅读 [Graph 类](#) 文档来了解如何管理多个图。

```
1. import tensorflow as tf
2.
3. # 创建一个常量 op，产生一个 1x2 矩阵。这个 op 被作为一个节点
4. # 加到默认图中。
5. #
6. # 构造器的返回值代表该常量 op 的返回值。
7. matrix1 = tf.constant([[3., 3.]])
8.
9. # 创建另外一个常量 op，产生一个 2x1 矩阵。
10. matrix2 = tf.constant([[2.],[2.]])
11.
12. # 创建一个矩阵乘法 matmul op，把 'matrix1' 和 'matrix2' 作为输入。
13. # 返回值 'product' 代表矩阵乘法的结果。
14. product = tf.matmul(matrix1, matrix2)
```

默认图现在有三个节点，两个 `tf.constant()` op，和一个 `tf.matmul()` op。为了真正进行矩阵相乘运算，并得到矩阵乘法的结果，你必须在会话里启动这个图。

在一个会话中启动图

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

欲了解完整的会话 API，请阅读[Session 类](#)。

```
1. # 启动默认图。
2. sess = tf.Session()
3.
4. # 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。
5. # 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
6. # 矩阵乘法 op 的输出。
7. #
8. # 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。
9. #
10. # 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。
11. #
12. # 返回值 'result' 是一个 numpy `ndarray` 对象。
13. result = sess.run(product)
14. print result
15. # ==> [[ 12.]]
16.
17. # 任务完成，关闭会话。
18. sess.close()
```

`Session` 对象在使用完后需要关闭以释放资源。除了显式调用 `close` 外，也可以使用“with”代码块来自动完成关闭动作。

```
1. with tf.Session() as sess:
2.     result = sess.run([product])
3.     print result
```

在实现上，TensorFlow 将图形定义转换成分布式执行的操作，以充分利用可用的计算资源（如 CPU 或 GPU）。一般你不需要显式指定使用 CPU 还是 GPU，TensorFlow 能自动检测。如果检测到 GPU，TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作。

如果机器上有超过一个可用的 GPU，除第一个外的其它 GPU 默认是不参与计算的。为了让 TensorFlow

使用这些 GPU，你必须将 op 明确指派给它们执行。`with...Device` 语句用来指派特定的 CPU 或 GPU 执行操作：

```

1. with tf.Session() as sess:
2.     with tf.device("/gpu:1"):
3.         matrix1 = tf.constant([[3., 3.]])
4.         matrix2 = tf.constant([[2.],[2.]])
5.         product = tf.matmul(matrix1, matrix2)
6.         ...

```

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。

阅读[使用GPU](#)章节，了解 TensorFlow GPU 使用的更多信息。

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。

```

1. # 进入一个交互式 TensorFlow 会话。
2. import tensorflow as tf
3. sess = tf.InteractiveSession()
4.
5. x = tf.Variable([1.0, 2.0])
6. a = tf.constant([3.0, 3.0])
7.
8. # 使用初始化器 initializer op 的 run() 方法初始化 'x'
9. x.initializer.run()
10.
11. # 增加一个减法 sub op，从 'x' 减去 'a'。运行减法 op，输出结果
12. sub = tf.sub(x, a)
13. print sub.eval()
14. # ==> [-2. -1.]

```

Tensor

TensorFlow 程序使用 `tensor` 数据结构来代表所有的数据，计算图中，操作间传递的数据都是 `tensor`。

你可以把 TensorFlow `tensor` 看作是一个 n 维的数组或列表。一个 `tensor` 包含一个静态类型 `rank`，和

一个 `shape`。想了解 TensorFlow 是如何处理这些概念的，参见

[Rank](#)，[Shape](#)，和 [Type](#)。

变量

[Variables](#) for more details.

变量维护图执行过程中的状态信息。下面的例子演示了如何使用变量实现一个简单的计数器。参见 [变量](#) 章节了解更多细节。

```
1. # 创建一个变量，初始化为标量 0.
2. state = tf.Variable(0, name="counter")
3.
4. # 创建一个 op，其作用是使 state 增加 1
5.
6. one = tf.constant(1)
7. new_value = tf.add(state, one)
8. update = tf.assign(state, new_value)
9.
10. # 启动图后，变量必须先经过`初始化` (init) op 初始化，
11. # 首先必须增加一个`初始化` op 到图中。
12. init_op = tf.initialize_all_variables()
13.
14. # 启动图，运行 op
15. with tf.Session() as sess:
16.     # 运行 'init' op
17.     sess.run(init_op)
18.     # 打印 'state' 的初始值
19.     print sess.run(state)
20.     # 运行 op，更新 'state'，并打印 'state'
21.     for _ in range(3):
22.         sess.run(update)
23.         print sess.run(state)
24.
25. # 输出：
26.
27. # 0
28. # 1
29. # 2
30. # 3
```

代码中 `assign()` 操作是图所描绘的表达式的一部分，正如 `add()` 操作一样。所以在调用 `run()` 执行表达式之前，它并不会真正执行赋值操作。

通常会将一个统计模型中的参数表示为一组变量。例如，你可以将一个神经网络的权重作为某个变量存储在一个 `tensor` 中。

在训练过程中，通过重复运行训练图，更新这个 `tensor`。

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`：

```
1. input1 = tf.constant(3.0)
2. input2 = tf.constant(2.0)
3. input3 = tf.constant(5.0)
4. intermed = tf.add(input2, input3)
5. mul = tf.mul(input1, intermed)
6.
7. with tf.Session() as sess:
8.     result = sess.run([mul, intermed])
9.     print result
10.
11. # 输出:
12. # [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

需要获取的多个 `tensor` 值，在 `op` 的一次运行中一起获得（而不是逐个去获取 `tensor`）。

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。

```
1.  
2. input1 = tf.placeholder(tf.float32)  
3. input2 = tf.placeholder(tf.float32)  
4. output = tf.mul(input1, input2)  
5.  
6. with tf.Session() as sess:  
7.     print sess.run([output], feed_dict={input1:[7.], input2:[2.]})  
8.  
9. # 输出:  
10. # [array([ 14.], dtype=float32)]
```

for a larger-scale example of feeds.

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

MNIST 全连通 [feed 教程](#)

([source code](#))

给出了一个更大规模的使用 feed 的例子。

原文: [Basic Usage](#) 翻译: [@doc001](#) 校对: [@yangtze](#)

教程

教程

- [总览](#)
- [MNIST 机器学习入门](#)
- [深入 MNIST](#)
- [TensorFlow 运作方式入门](#)
- [卷积神经网络](#)
- [字词的向量表示](#)
- [递归神经网络](#)
- [曼德布洛特\(Mandelbrot\)集合](#)
- [偏微分方程](#)
- [MNIST数据下载](#)

总览

- [综述](#)
 - [面向机器学习初学者的 MNIST 初级教程](#)
 - [面向机器学习专家的 MNIST 高级教程](#)
 - [TensorFlow 使用指南](#)
 - [卷积神经网络](#)
 - [单词的向量表示](#)
 - [循环神经网络 \(Recurrent Neural Network, 简称 RNN\)](#)
 - [序列到序列模型 \(Sequence-to-Sequence Model\)](#)
 - [Mandelbrot 集合](#)
 - [偏微分方程](#)
 - [MNIST 数据下载](#)
 - [视觉物体识别 \(Visual Object Recognition\)](#)
 - [Deep Dream 视幻觉软件](#)

综述

面向机器学习初学者的 MNIST 初级教程

如果你是机器学习领域的新手，我们推荐你从本文开始阅读。本文通过讲述一个经典的问题，手写数字识别 (MNIST)，让你对多类分类 (multiclass classification) 问题有直观的了解。

[阅读教程](#)

面向机器学习专家的 MNIST 高级教程

如果你已经对其它深度学习软件比较熟悉，并且也对 MNIST 很熟悉，这篇教程能够引导你对 TensorFlow 有初步了解。

[阅读教程](#)

TensorFlow 使用指南

这是一篇技术教程，详细介绍了如何使用 TensorFlow 架构训练大规模模型。本文继续使用 MNIST 作为例子。

[阅读教程](#)

卷积神经网络

这篇文章介绍了如何使用 TensorFlow 在 CIFAR-10 数据集上训练卷积神经网络。卷积神经网络是为图像识别量身定做的一个模型。相比其它模型，该模型利用了平移不变性(translation invariance)，从而能够更更简洁有效地表示视觉内容。

[阅读教程](#)

单词的向量表示

本文让你了解为什么学会使用向量来表示单词，即单词嵌套 (word embedding)，是一件很有用的事情。文章中介绍的 word2vec 模型，是一种高效学习嵌套的方法。本文还涉及了对比噪声 (noise-contrastive) 训练方法的一些高级细节，该训练方法是训练嵌套领域最近最大的进展。

[阅读教程](#)

循环神经网络 (Recurrent Neural Network, 简称 RNN)

一篇 RNN 的介绍文章，文章中训练了一个 LSTM 网络来预测一个英文句子的下一个单词(该任务有时候被称作语言建模)。

[阅读教程](#)

序列到序列模型 (Sequence-to-Sequence Model)

RNN 教程的后续，该教程采用序列到序列模型进行机器翻译。你将学会构建一个完全基于机器学习，端到端的 英语-法语 翻译器。

[阅读教程](#)

Mandelbrot 集合

TensorFlow 可以用于与机器学习完全无关的其它计算领域。这里实现了一个原生的 Mandelbrot 集合的可视化程序。

[阅读教程](#)

偏微分方程

这是另外一个非机器学习计算的例子，我们利用一个原生实现的偏微分方程，对雨滴落在池塘上的过程进行仿真。

[阅读教程](#)

MNIST 数据下载

一篇关于下载 MNIST 手写识别数据集的详细教程。

[阅读教程](#)

视觉物体识别 (Visual Object Recognition)

我们将毫无保留地发布已经选训练好的，目前最先进的 Inception 物体识别模型。

敬请期待...

Deep Dream 视幻觉软件

我们将发布一个 TensorFlow 版本的 [Deep Dream](#)，这是一款基于 Inception 识别模型的神经网络视幻觉软件。

敬请期待...

原文: [Overview](#) 翻译: [@doc001](#) 校对: [@eric_xu](#)

MNIST 机器学习入门

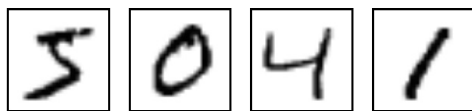
- [MNIST机器学习入门](#)
 - [MNIST数据集](#)
 - [Softmax回归介绍](#)
 - [实现回归模型](#)
 - [训练模型](#)
 - [评估我们的模型](#)

MNIST机器学习入门

这个教程的目标读者是对机器学习和`TensorFlow`都不太了解的新手。如果你已经了解`MNIST`和`softmax`回归(`softmax regression`)的相关知识，你可以阅读这个[快速上手教程](#)。

当我们开始学习编程的时候，第一件事往往是学习打印“Hello World”。就好比编程入门有Hello World，机器学习入门有MNIST。

MNIST是一个入门级的计算机视觉数据集，它包含各种手写数字图片：



它也包含每一张图片对应的标签，告诉我们这个是数字几。比如，上面这四张图片的标签分别是5，0，4，1。

在此教程中，我们将训练一个机器学习模型用于预测图片里面的数字。我们的目的不是要设计一个世界一流的复杂模型 — 尽管我们会在之后给你源代码去实现一流的预测模型 — 而是要介绍下如何使用`TensorFlow`。所以，我们这里会从一个很简单的数学模型开始，它叫做`Softmax Regression`。

对应这个教程的实现代码很短，而且真正有意思的内容只包含在三行代码里面。但是，去理解包含在这些代码里面的设计思想是非常重要的：`TensorFlow`工作流程和机器学习的基本概念。因此，这个教程会很详细地介绍这些代码的实现原理。

MNIST数据集

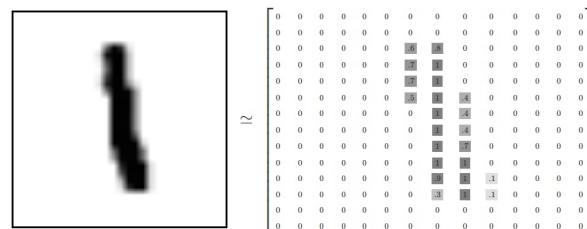
MNIST数据集的官网是[Yann LeCun's website](#)。在这里，我们提供了一份python源代码用于自动下载和安装这个数据集。你可以下载[这份代码](#)，然后用下面的代码导入到你的项目里面，也可以直接复制粘贴到你的代码文件里面。

```
1. import tensorflow.examples.tutorials.mnist.input_data
2. mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

下载下来的数据集被分成两部分：60000行的训练数据集（`mnist.train`）和10000行的测试数据集（`mnist.test`）。这样的切分很重要，在机器学习模型设计时必须有一个单独的测试数据集不用于训练而是用来评估这个模型的性能，从而更加容易把设计的模型推广到其他数据集上（泛化）。

正如前面提到的一样，每一个MNIST数据单元有两部分组成：一张包含手写数字的图片和一个对应的标签。我们把这些图片设为“xs”，把这些标签设为“ys”。训练数据集和测试数据集都包含xs和ys，比如训练数据集的图片是 `mnist.train.images`，训练数据集的标签是 `mnist.train.labels`。

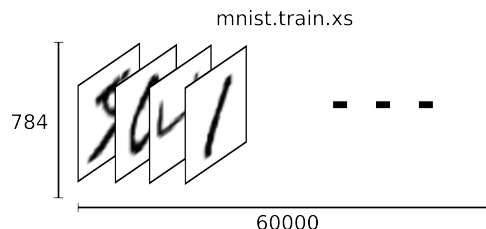
每一张图片包含28X28个像素点。我们可以用一个数字数组来表示这张图片：



我们把这个数组展开成一个向量，长度是 $28 \times 28 = 784$ 。如何展开这个数组（数字间的顺序）不重要，只要保持各个图片采用相同的方式展开。从这个角度来看，MNIST数据集的图片就是在784维向量空间里面的点，并且拥有比较**复杂**的结构（提醒：此类数据的可视化是计算密集型的）。

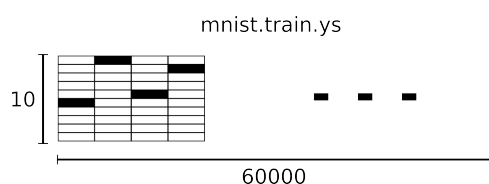
展平图片的数字数组会丢失图片的二维结构信息。这显然是不理想的，最优秀的计算机视觉方法会挖掘并利用这些结构信息，我们会在后续教程中介绍。但是在这个教程中我们忽略这些结构，所介绍的简单数学模型，softmax回归(softmax regression)，不会利用这些结构信息。

因此，在MNIST训练数据集中，`mnist.train.images` 是一个形状为 `[60000, 784]` 的张量，第一个维度数字用来索引图片，第二个维度数字用来索引每张图片中的像素点。在此张量里的每一个元素，都表示某张图片里的某个像素的强度值，值介于0和1之间。



相对应的MNIST数据集的标签是介于0到9的数字，用来描述给定图片里表示的数字。为了用于这个教程，我们使标签数据是“one-hot vectors”。一个one-hot向量除了某一位的数字是1以外其余各维度数字都是0。所以在此教程中，数字n将表示成一个只有在第n维度（从0开始）数字为1的10维向

量。比如，标签0将表示成 $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ 。因此，`mnist.train.labels` 是一个 `[60000, 10]` 的数字矩阵。



现在，我们准备好可以开始构建我们的模型啦！

Softmax回归介绍

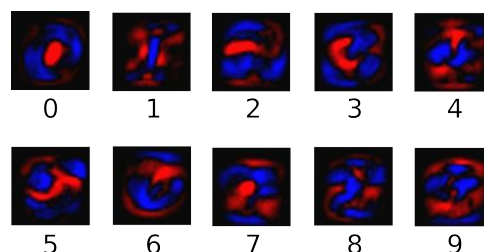
我们知道MNIST的每一张图片都表示一个数字，从0到9。我们希望得到给定图片代表每个数字的概率。比如说，我们的模型可能推测一张包含9的图片代表数字9的概率是80%但是判断它是8的概率是5%（因为8和9都有上半部分的小圆），然后给予它代表其他数字的概率更小的值。

这是一个使用softmax回归（softmax regression）模型的经典案例。softmax模型可以用来给不同的对象分配概率。即使在之后，我们训练更加精细的模型时，最后一步也需要用softmax来分配概率。

softmax回归（softmax regression）分两步：第一步

为了得到一张给定图片属于某个特定数字类的证据（evidence），我们对图片像素值进行加权求和。如果这个像素具有很强的证据说明这张图片不属于该类，那么相应的权值为负数，相反如果这个像素拥有有利的证据支持这张图片属于这个类，那么权值是正数。

下面的图片显示了一个模型学习到的图片上每个像素对于特定数字类的权值。红色代表负数权值，蓝色代表正数权值。



我们也需要加入一个额外的偏置量（bias），因为输入往往会带有一些无关的干扰量。因此对于给定的输入图片 x 它代表的是数字 i 的证据可以表示为

$$\text{evidence}_i = \sum_j W_{i,j} x_j + b_i$$

其中 W_i 代表权重, b_i 代表数字 i 类的偏置量, j 代表给定图片 x 的像素索引用于像素求和。然后用softmax函数可以把这些证据转换成概率 y :

$$y = \text{softmax}(\text{evidence})$$

这里的softmax可以看成是一个激励 (activation) 函数或者链接 (link) 函数, 把我们定义的线性函数的输出转换成我们想要的格式, 也就是关于10个数字类的概率分布。因此, 给定一张图片, 它对于每一个数字的吻合度可以被softmax函数转换成为一个概率值。softmax函数可以定义为:

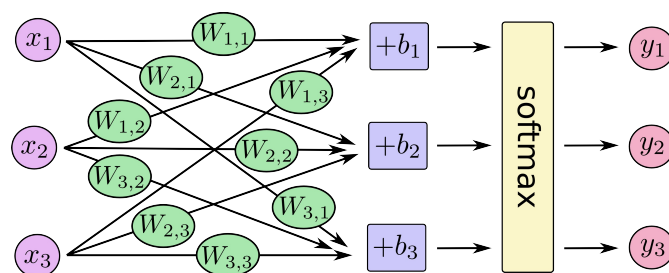
$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

展开等式右边的子式, 可以得到:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

但是更多的时候把softmax模型函数定义为前一种形式: 把输入值当成幂指数求值, 再正则化这些结果值。这个幂运算表示, 更大的证据对应更大的假设模型 (hypothesis) 里面的乘数权重值。反之, 拥有更少的证据意味着在假设模型里面拥有更小的乘数系数。假设模型里的权值不可以是0值或者负值。Softmax然后会正则化这些权重值, 使它们的总和等于1, 以此构造一个有效的概率分布。(更多的关于Softmax函数的信息, 可以参考Michael Nieslen的书里面的[这个部分](#), 其中有关于softmax的可交互式的可视化解释。)

对于softmax回归模型可以用下面的图解释, 对于输入的 x_s 加权求和, 再分别加上一个偏置量, 最后再输入到softmax函数中:



如果把它写成一个等式，我们可以得到：

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix}$$

我们也可以用向量表示这个计算过程：用矩阵乘法和向量相加。这有助于提高计算效率。（也是一种更有效的思考方式）

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

更进一步，可以写成更加紧凑的方式：

$$y = \text{softmax}(Wx + b)$$

实现回归模型

为了用python实现高效的数值计算，我们通常会使用函数库，比如NumPy，会把类似矩阵乘法这样的复杂运算使用其他外部语言实现。不幸的是，从外部计算切换回Python的每一个操作，仍然是一个很大的开销。如果你用GPU来进行外部计算，这样的开销会更大。用分布式的计算方式，也会花费更多的资源用来传输数据。

TensorFlow也把复杂的计算放在python之外完成，但是为了避免前面说的那些开销，它做了进一步完善。Tensorflow不单独地运行单一的复杂计算，而是让我们可以先用图描述一系列可交互的计算操作，然后全部一起在Python之外运行。（这样类似的运行方式，可以在不少的机器学习库中看

到。)

使用TensorFlow之前，首先导入它：

```
1. import tensorflow as tf
```

我们通过操作符号变量来描述这些可交互的操作单元，可以用下面的方式创建一个：

```
1. x = tf.placeholder(tf.float32, [None, 784])
```

`x` 不是一个特定的值，而是一个占位符 `placeholder`，我们在TensorFlow运行计算时输入这个值。我们希望能够输入任意数量的MNIST图像，每一张图展平成784维的向量。我们用2维的浮点数张量来表示这些图，这个张量的形状是 `[None, 784]`。（这里的 `None` 表示此张量的第一个维度可以是任何长度的。）

我们的模型也需要权重值和偏置量，当然我们可以把它们当做是另外的输入（使用占位符），但TensorFlow有一个更好的方法来表示它们：`Variable`。

一个 `Variable` 代表一个可修改的张量，存在在TensorFlow的用于描述交互性操作的图中。它们可以用于计算输入值，也可以在计算中被修改。对于各种机器学习应用，一般都会有模型参数，可以用 `Variable` 表示。

```
1. W = tf.Variable(tf.zeros([784,10]))
2. b = tf.Variable(tf.zeros([10]))
```

我们赋予 `tf.Variable` 不同的初值来创建不同的 `Variable`：在这里，我们都用全为零的张量来初始化 `W` 和 `b`。因为我们要学习 `W` 和 `b` 的值，它们的初值可以随意设置。

注意，`W` 的维度是 `[784, 10]`，因为我们想要用784维的图片向量乘以它得到一个10维的证据值向量，每一位对应不同数字类。`b` 的形状是 `[10]`，所以我们可以直接把它加到输出上面。

现在，我们可以实现我们的模型啦。只需要一行代码！

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

首先，我们用 `tf.matmul(X, W)` 表示 `x` 乘以 `W`，对应之前等式里面的 Wx ，这里 `x` 是一个2维张量拥有多个输入。然后再加上 `b`，把和输入到 `tf.nn.softmax` 函数里面。

至此，我们先用了几行简短的代码来设置变量，然后只用了一行代码来定义我们的模型。

TensorFlow不仅仅可以使softmax回归模型计算变得特别简单，它也用这种非常灵活的方式来描述其他各种数值计算，从机器学习模型对物理学模拟仿真模型。一旦被定义好之后，我们的模型可以在不同的设备上运行：计算机的CPU，GPU，甚至是手机！

训练模型

为了训练我们的模型，我们首先需要定义一个指标来评估这个模型是好的。其实，在机器学习，我们通常定义指标来表示一个模型是坏的，这个指标称为成本（cost）或损失（loss），然后尽量最小化这个指标。但是，这两种方式是相同的。

一个非常常见的，非常漂亮的成本函数是“交叉熵”（cross-entropy）。交叉熵产生于信息论里面的信息压缩编码技术，但是它后来演变成为从博弈论到机器学习等其他领域里的重要技术手段。它的定义如下：

$$H_{y'}(y) = - \sum_i y'_i \log(y_i)$$

y 是我们预测的概率分布， y' 是实际的分布（我们输入的one-hot vector）。比较粗糙的理解是，交叉熵是用来衡量我们的预测用于描述真相的低效性。更详细的关于交叉熵的解释超出本教程的范畴，但是你很有必要好好[理解它](#)。

为了计算交叉熵，我们首先需要添加一个新的占位符用于输入正确值：

```
1. y_ = tf.placeholder("float", [None,10])
```

$$- \sum y' \log(y):$$

然后我们可以用

计算交叉熵：

```
1. cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

首先，用 `tf.log` 计算 `y` 的每个元素的`对数`。接下来，我们把 `y_` 的每一个元素和 `tf.log(y)` 的对应元素相乘。最后，用 `tf.reduce_sum` 计算张量的所有元素的总和。（注意，这里的交叉熵不仅仅用来衡量单一的一对预测和真实值，而是所有100幅图片的交叉熵的总和。对于100个数据点的预测表现比单一数据点的表现能更好地描述我们的模型的性能。

现在我们知道我们需要我们的模型做什么啦，用TensorFlow来训练它是非常容易的。因为TensorFlow拥有一张描述你各个计算单元的图，它可以自动地使用[反向传播算法](#)（`backpropagation algorithm`）来有效地确定你的变量是如何影响你想要最小化的那个成本值的。然后，TensorFlow会用你选择的优化算法来不断地修改变量以降低成本。

```
1. train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

在这里，我们要求TensorFlow用梯度下降算法 (gradient descent algorithm) 以0.01的学习速率最小化交叉熵。梯度下降算法 (gradient descent algorithm) 是一个简单的学习过程，TensorFlow只需将每个变量一点点地往使成本不断降低的方向移动。当然TensorFlow也提供了[其他许多优化算法](#)：只要简单地调整一行代码就可以使用其他的算法。

TensorFlow在这里实际上所做的是，它会在后台给描述你的计算的那张图里面增加一系列新的计算操作单元用于实现反向传播算法和梯度下降算法。然后，它返回给你的只是一个单一的操作，当运行这个操作时，它用梯度下降算法训练你的模型，微调你的变量，不断减少成本。

现在，我们已经设置好了我们的模型。在运行计算之前，我们需要添加一个操作来初始化我们创建的变量：

```
1. init = tf.initialize_all_variables()
```

现在我们可以 在一个 `Session` 里面启动我们的模型，并且初始化变量：

```
1. sess = tf.Session()
2. sess.run(init)
```

然后开始训练模型，这里我们让模型循环训练1000次！

```
1. for i in range(1000):
2.     batch_xs, batch_ys = mnist.train.next_batch(100)
3.     sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

该循环的每个步骤中，我们都会随机抓取训练数据中的100个批处理数据点，然后用这些数据点作为参数替换之前的占位符来运行 `train_step`。

使用一小部分的随机数据来进行训练被称为随机训练 (stochastic training) - 在这里更确切的说随机梯度下降训练。在理想情况下，我们希望用我们所有的数据来进行每一步的训练，因为这能给我们更好的训练结果，但显然这需要很大的计算开销。所以，每一次训练我们可以使用不同的数据子集，这样做既可以减少计算开销，又可以最大化地学习到数据集的总体特性。

评估我们的模型

那么我们的模型性能如何呢？

首先让我们找出那些预测正确的标签。`tf.argmax` 是一个非常有用的函数，它能给出某个tensor对象在某一维上的其数据最大值所在的索引值。由于标签向量是由0,1组成，因此最大值1所在的索引位置就是类别标签，比如 `tf.argmax(y,1)` 返回的是模型对于任一输入x预测到的标签值，而 `tf.argmax(y_,1)` 代表正确的标签，我们可以用 `tf.equal` 来检测我们的预测是否真实标签匹配 (索引位置一样表示匹配)。

```
1. correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

这行代码会给我们一组布尔值。为了确定正确预测项的比例，我们可以把布尔值转换成浮点数，然后取平均值。例如，`[True, False, True, True]` 会变成 `[1,0,1,1]`，取平均值后得到 `0.75`。

```
1. accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

最后，我们计算所学习到的模型在测试数据集上面的正确率。

```
1. print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

这个最终结果值应该大约是91%。

这个结果好吗？嗯，并不太好。事实上，这个结果是很差的。这是因为我们仅仅使用了一个非常简单的模型。不过，做一些小小的改进，我们就可以得到97%的正确率。最好的模型甚至可以获得超过99.7%的准确率！（想了解更多信息，可以看看这个关于各种模型的[性能对比列表](#)。）

比结果更重要的是，我们从这个模型中学习到的设计思想。不过，如果你仍然对这里的结果有点失望，可以查看[下一个教程](#)，在那里你可以学习如何用TensorFlow构建更加复杂的模型以获得更好的性能！

原文地址：[MNIST For ML Beginners](#) 翻译：[linbojin](#) 校对：

深入 MNIST

- [深入MNIST](#)
 - [安装](#)
 - [加载MNIST数据](#)
 - [运行TensorFlow的InteractiveSession](#)
 - [计算图](#)
 - [构建Softmax 回归模型](#)
 - [占位符](#)
 - [变量](#)
 - [类别预测与损失函数](#)
 - [训练模型](#)
 - [评估模型](#)
 - [构建一个多层卷积网络](#)
 - [权重初始化](#)
 - [卷积和池化](#)
 - [第一层卷积](#)
 - [第二层卷积](#)
 - [密集连接层](#)
 - [Dropout](#)
 - [输出层](#)
 - [训练和评估模型](#)

深入MNIST

TensorFlow是一个非常强大的用来做大规模数值计算的库。其所擅长的任务之一就是实现以及训练深度神经网络。

在本教程中，我们将学到构建一个TensorFlow模型的基本步骤，并将通过这些步骤为MNIST构建一个深度卷积神经网络。

这个教程假设你已经熟悉神经网络和MNIST数据集。如果你尚未了解，请查看[新手指南](#)。

安装

在创建模型之前，我们会先加载MNIST数据集，然后启动一个TensorFlow的session。

加载MNIST数据

为了方便起见，我们已经准备了一个[脚本](#)来自动下载和导入MNIST数据集。它会自动创建一

个 `'MNIST_data'` 的目录来存储数据。

```
1. import input_data
2. mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

这里，`mnist` 是一个轻量级的类。它以Numpy数组的形式存储着训练、校验和测试数据集。同时提供了一个函数，用于在迭代中获得minibatch，后面我们将会用到。

运行TensorFlow的InteractiveSession

Tensorflow依赖于一个高效的C++后端来进行计算。与后端的这个连接叫做session。一般而言，使用TensorFlow程序的流程是先创建一个图，然后在session中启动它。

这里，我们使用更加方便的 `InteractiveSession` 类。通过它，你可以更加灵活地构建你的代码。它能让你在运行图的时候，插入一些[计算图](#)，这些计算图是由某些操作(operations)构成的。这对于工作在交互式环境中的人们来说非常便利，比如使用IPython。如果你没有使用 `InteractiveSession`，那么你需要在启动session之前构建整个计算图，然后[启动该计算图](#)。

```
1. import tensorflow as tf
2. sess = tf.InteractiveSession()
```

计算图

为了在Python中进行高效的数值计算，我们通常会使用像NumPy一类的库，将一些诸如矩阵乘法的耗时操作在Python环境的外部来计算，这些计算通常会通过其它语言并用更为高效的代码来实现。

但遗憾的是，每一个操作切换回Python环境时仍需要不小的开销。如果你想在GPU或者分布式环境中计算时，这一开销更加可怖，这一开销主要可能是用来进行数据迁移。

TensorFlow也是在Python外部完成其主要工作，但是进行了改进以避免这种开销。其并没有采用在Python外部独立运行某个耗时操作的方式，而是先让我们描述一个交互操作图，然后完全将其运行在Python外部。这与Theano或Torch的做法类似。

因此Python代码的目的是用来构建这个可以在外部运行的计算图，以及安排计算图的哪一部分应该被运行。详情请查看[基本用法](#)中的[计算图表](#)一节。

构建Softmax 回归模型

在这一节中我们将建立一个拥有一个线性层的softmax回归模型。在下一节，我们会将其扩展为一个拥有多层卷积网络的softmax回归模型。

占位符

我们通过为输入图像和目标输出类别创建节点，来开始构建计算图。

```
1. x = tf.placeholder("float", shape=[None, 784])
2. y_ = tf.placeholder("float", shape=[None, 10])
```

这里的 `x` 和 `y` 并不是特定的值，相反，他们都只是一个 `占位符`，可以在TensorFlow运行某一计算时根据该占位符输入具体的值。

输入图片 `x` 是一个2维的浮点数张量。这里，分配给它的 `shape` 为 `[None, 784]`，其中 `784` 是一张展平的MNIST图片的维度。`None` 表示其值大小不定，在这里作为第一个维度值，用以指代batch的大小，意即 `x` 的数量不定。输出类别值 `y_` 也是一个2维张量，其中每一行为一个10维的one-hot向量，用于代表对应某一MNIST图片的类别。

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

变量

我们现在为模型定义权重 `w` 和偏置 `b`。可以将它们当作额外的输入量，但是TensorFlow有一个更好的处理方式：`变量`。一个 `变量` 代表着TensorFlow计算图中的一个值，能够在计算过程中使用，甚至进行修改。在机器学习的应用过程中，模型参数一般用 `Variable` 来表示。

```
1. w = tf.Variable(tf.zeros([784,10]))
2. b = tf.Variable(tf.zeros([10]))
```

我们在调用 `tf.Variable` 的时候传入初始值。在这个例子里，我们把 `w` 和 `b` 都初始化为零向量。`w` 是一个784x10的矩阵（因为我们有784个特征和10个输出值）。`b` 是一个10维的向量（因为我们有10个分类）。

Before `Variable`s can be used within a session, they must be initialized using that session.

This step takes the initial values (in this case tensors full of zeros) that

have already been specified, and assigns them to each `Variable`. This can be

done for all `Variables` at once.

`变量` 需要通过session初始化后，才能在session中使用。这一初始化步骤为，为初始值指定具体值（本例当中是全为零），并将其分配给每个 `变量`，可以一次性为所有 `变量` 完成此操作。

```
1. sess.run(tf.initialize_all_variables())
```

类别预测与损失函数

现在我们可以实现我们的回归模型了。这只需要一行！我们把向量化后的图片 `x` 和权重矩阵 `W` 相乘，加上偏置 `b`，然后计算每个分类的softmax概率值。

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

可以很容易的为训练过程指定最小化误差用的损失函数，我们的损失函数是目标类别和预测类别之间的交叉熵。

```
1. cross_entropy = -tf.reduce_sum(y*tf.log(y))
```

注意，`tf.reduce_sum` 把minibatch里的每张图片的交叉熵值都加起来了。我们计算的交叉熵是指整个minibatch的。

训练模型

我们已经定义好模型和训练用的损失函数，那么用TensorFlow进行训练就很简单了。因为TensorFlow知道整个计算图，它可以使用自动微分法找到对于各个变量的损失的梯度值。TensorFlow有[大量内置的优化算法](#) 这个例子中，我们用最速下降法让交叉熵下降，步长为0.01。

```
1. train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

这一行代码实际上是用来往计算图上添加一个新操作，其中包括计算梯度，计算每个参数的步长变化，并且计算出新的参数值。

返回的 `train_step` 操作对象，在运行时会使用梯度下降来更新参数。因此，整个模型的训练可以通过反复地运行 `train_step` 来完成。

```
1. for i in range(1000):
2.     batch = mnist.train.next_batch(50)
3.     train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

每一步迭代，我们都会加载50个训练样本，然后执行一次 `train_step`，并通过 `feed_dict` 将 `x` 和 `y_` 张量 `占位符` 用训练数据替代。

注意，在计算图中，你可以用 `feed_dict` 来替代任何张量，并不仅限于替换 `占位符`。

评估模型

那么我们的模型性能如何呢？

首先让我们找出那些预测正确的标签。`tf.argmax` 是一个非常有用的函数，它能给出某个tensor对象在某一维上的其数据最大值所在的索引值。由于标签向量是由0, 1组成，因此最大值1所在的索引位置就是类别标签，比如 `tf.argmax(y, 1)` 返回的是模型对于任一输入x预测到的标签值，而 `tf.argmax(y_, 1)` 代表正确的标签，我们可以用 `tf.equal` 来检测我们的预测是否真实标签匹配（索引位置一样表示匹配）。

```
1. correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

这里返回一个布尔数组。为了计算我们分类的准确率，我们将布尔值转换为浮点数来代表对、错，然后取平均值。例如：`[True, False, True, True]` 变为 `[1, 0, 1, 1]`，计算出平均值为 `0.75`。

```
1. accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

最后，我们可以计算出在测试数据上的准确率，大概是91%。

```
1. print accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

构建一个多层卷积网络

在MNIST上只有91%正确率，实在太糟糕。在这个小节里，我们用一个稍微复杂的模型：卷积神经网络来改善效果。这会达到大概99.2%的准确率。虽然不是最高，但是还是比较让人满意。

权重初始化

为了创建这个模型，我们需要创建大量的权重和偏置项。这个模型中的权重在初始化时应该加入少量的噪声来打破对称性以及避免0梯度。由于我们使用的是ReLU神经元，因此比较好的做法是用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为0的问题（dead neurons）。为了不在建立模型的时候反复做初始化操作，我们定义两个函数用于初始化。

```
1. def weight_variable(shape):
2.     initial = tf.truncated_normal(shape, stddev=0.1)
3.     return tf.Variable(initial)
4.
5. def bias_variable(shape):
6.     initial = tf.constant(0.1, shape=shape)
7.     return tf.Variable(initial)
```

卷积和池化

TensorFlow在卷积和池化上有很强的灵活性。我们怎么处理边界？步长应该设多大？在这个实例里，我们会一直使用vanilla版本。我们的卷积使用1步长（stride size），0边距（padding size）的模板，保证输出和输入是同一个大小。我们的池化用简单传统的2x2大小的模板做max pooling。为了代码更简洁，我们把这部分抽象成一个函数。

```
1. def conv2d(x, W):
2.     return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
3.
4. def max_pool_2x2(x):
5.     return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
6.                             strides=[1, 2, 2, 1], padding='SAME')
```

第一层卷积

现在我们可以开始实现第一层了。它由一个卷积接一个max pooling完成。卷积在每个5x5的patch中算出32个特征。卷积的权重张量形状是 `[5, 5, 1, 32]`，前两个维度是patch的大小，接着是输入的通道数目，最后是输出的通道数目。而对于每一个输出通道都有一个对应的偏置量。

```
1. W_conv1 = weight_variable([5, 5, 1, 32])
2. b_conv1 = bias_variable([32])
```

为了用这一层，我们把 `x` 变成一个4d向量，其第2、第3维对应图片的宽、高，最后一维代表图片的颜色通道数（因为是灰度图所以这里的通道数为1，如果是rgb彩色图，则为3）。

```
1. x_image = tf.reshape(x, [-1, 28, 28, 1])
```

We then convolve `x_image` with the weight tensor, add the bias, apply the ReLU function, and finally max pool.

我们把 `x_image` 和权值向量进行卷积，加上偏置项，然后应用ReLU激活函数，最后进行max pooling。

```
1. h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
2. h_pool1 = max_pool_2x2(h_conv1)
```

第二层卷积

为了构建一个更深的网络，我们会把几个类似的层堆叠起来。第二层中，每个5x5的patch会得到64个特征。

```

1. W_conv2 = weight_variable([5, 5, 32, 64])
2. b_conv2 = bias_variable([64])
3.
4. h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
5. h_pool2 = max_pool_2x2(h_conv2)

```

密集连接层

现在，图片尺寸减小到7x7，我们加入一个有1024个神经元的全连接层，用于处理整个图片。我们把池化层输出的张量reshape成一些向量，乘上权重矩阵，加上偏置，然后对其使用ReLU。

```

1. W_fc1 = weight_variable([7 * 7 * 64, 1024])
2. b_fc1 = bias_variable([1024])
3.
4. h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
5. h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

```

Dropout

为了减少过拟合，我们在输出层之前加入dropout。我们用一个 `placeholder` 来代表一个神经元的输出在dropout中保持不变的概率。这样我们可以在训练过程中启用dropout，在测试过程中关闭dropout。

TensorFlow的 `tf.nn.dropout` 操作除了可以屏蔽神经元的输出外，还会自动处理神经元输出值的scale。所以用dropout的时候可以不用考虑scale。

```

1. keep_prob = tf.placeholder("float")
2. h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

```

输出层

最后，我们添加一个softmax层，就像前面的单层softmax regression一样。

```

1. W_fc2 = weight_variable([1024, 10])
2. b_fc2 = bias_variable([10])
3.
4. y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

```

训练和评估模型

这个模型的效果如何呢？

为了进行训练和评估，我们使用与之前简单的单层SoftMax神经网络模型几乎相同的一套代码，只是我们会用更加复杂的ADAM优化器来做梯度最速下降，在 `feed_dict` 中加入额外的参数 `keep_prob` 来控制dropout比例。然后每100次迭代输出一次日志。

```
1. cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
2. train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
3. correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y,1))
4. accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
5. sess.run(tf.initialize_all_variables())
6. for i in range(20000):
7.     batch = mnist.train.next_batch(50)
8.     if i%100 == 0:
9.         train_accuracy = accuracy.eval(feed_dict={
10.             x:batch[0], y_: batch[1], keep_prob: 1.0})
11.         print "step %d, training accuracy %g"%(i, train_accuracy)
12.         train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
13.
14. print "test accuracy %g"%accuracy.eval(feed_dict={
15.     x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
```

以上代码，在最终测试集上的准确率大概是99.2%。

目前为止，我们已经学会了用TensorFlow快捷地搭建、训练和评估一个复杂一点儿的深度学习模型。

原文地址: [Deep MNIST for Experts](#) 翻译: [chenweican](#) 校对: [HongyangWang](#)

TensorFlow 运作方式入门

- TensorFlow运作方式入门
 - 教程使用的文件
 - 准备数据
 - 下载
 - 输入与占位符 (Inputs and Placeholders)
 - 构建图表 (Build the Graph)
 - 推理 (Inference)
 - 损失 (Loss)
 - 训练
 - 训练模型
 - 图表
 - 会话
 - 训练循环
 - 向图表提供反馈
 - 检查状态
 - 状态可视化
 - 保存检查点 (checkpoint)
 - 评估模型
 - 构建评估图表 (Eval Graph)
 - 评估图表的输出 (Eval Output)

TensorFlow运作方式入门

代码: [tensorflow/g3doc/tutorials/mnist/](https://www.tensorflow.org/g3doc/tutorials/mnist/)

本篇教程的目的，是向大家展示如何利用TensorFlow使用（经典）MNIST数据集训练并评估一个用于识别手写数字的简易前馈神经网络（feed-forward neural network）。我们的目标读者，是感兴趣使用TensorFlow的资深机器学习人士。

因此，撰写该系列教程并不是为了教大家机器学习领域的基础知识。

在学习本教程之前，请确保您已按照[安装TensorFlow](#)教程中的要求，完成了安装。

教程使用的文件

本教程引用如下文件：

文件	目的

<code>mnist.py</code>	构建一个完全连接 (fully connected) 的MNIST模型所需的代码。
<code>fully_connected_feed.py</code>	利用下载的数据集训练构建好的MNIST模型的主要代码，以数据反馈字典 (feed dictionary) 的形式作为输入模型。

只需要直接运行 `fully_connected_feed.py` 文件，就可以开始训练：

```
python fully_connected_feed.py
```

准备数据

MNIST是机器学习领域的一个经典问题，指的是让机器查看一系列大小为28x28像素的手写数字灰度图像，并判断这些图像代表0-9中的哪一个数字。



更多相关信息，请查阅Yann LeCun网站中关于MNIST的介绍或者Chris Olah对MNIST的可视化探索。

下载

在 `run_training()` 方法的一开始，`input_data.read_data_sets()` 函数会确保你的本地训练文件夹中，已经下载了正确的数据，然后将这些数据解压并返回一个含有 `DataSet` 实例的字典。

```
1. data_sets = input_data.read_data_sets(FLAGS.train_dir, FLAGS.fake_data)
```

注意：`fake_data` 标记是用于单元测试的，读者可以不必理会。

数据集	目的
<code>data_sets.train</code>	55000个图像和标签 (labels)，作为主要训练集。
<code>data_sets.validation</code>	5000个图像和标签，用于迭代验证训练准确度。
<code>data_sets.test</code>	10000个图像和标签，用于最终测试训练准确度 (trained accuracy)。

了解更多数据有关信息，请查阅此系列教程的[数据下载](#)部分。

输入与占位符 (Inputs and Placeholders)

`placeholder_inputs()` 函数将生成两个 `tf.placeholder` 操作，定义传入图表中的shape参数，shape参数中包括 `batch_size` 值，后续还会将实际的训练用例传入图表。

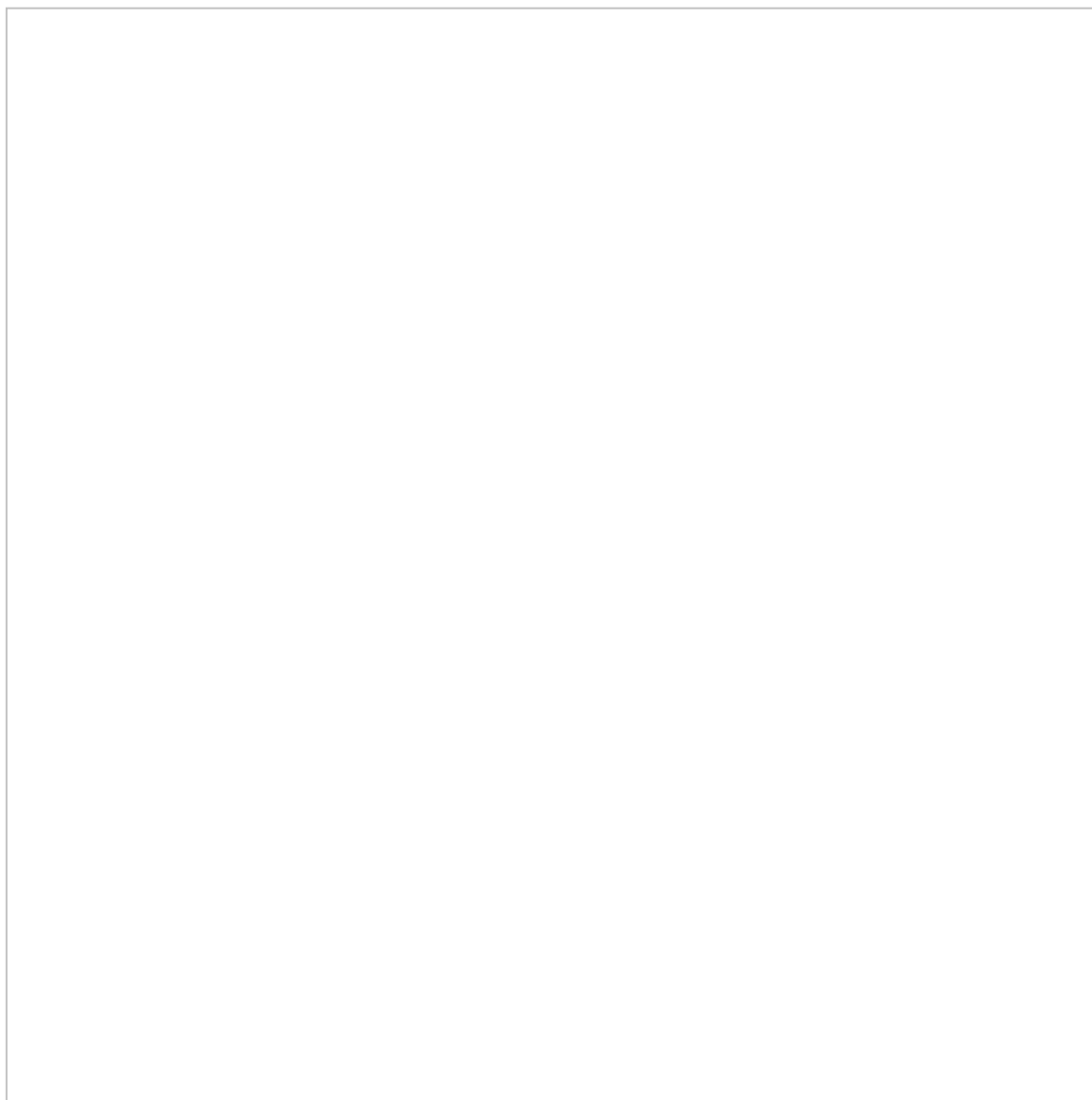
```
1. images_placeholder = tf.placeholder(tf.float32, shape=(batch_size,
2.                                     IMAGE_PIXELS))
3. labels_placeholder = tf.placeholder(tf.int32, shape=(batch_size))
```

在训练循环 (training loop) 的后续步骤中，传入的整个图像和标签数据集会被切片，以符合每一个操作所设置的 `batch_size` 值，占位符操作将会填补以符合这个 `batch_size` 值。然后使用 `feed_dict` 参数，将数据传入 `sess.run()` 函数。

构建图表 (Build the Graph)

在为数据创建占位符之后，就可以运行 `mnist.py` 文件，经过三阶段的模式函数操作：`inference()`，`loss()`，和 `training()`。图表就构建完成了。

1. `inference()` — 尽可能地构建好图表，满足促使神经网络向前反馈并做出预测的要求。
2. `loss()` — 往 `inference` 图表中添加生成损失 (loss) 所需要的操作 (ops)。
3. `training()` — 往损失图表中添加计算并应用梯度 (gradients) 所需的操作。



推理（Inference）

`inference()` 函数会尽可能地构建图表，做到返回包含了预测结果（output prediction）的 Tensor。

它接受图像占位符为输入，在此基础上借助ReLU(Rectified Linear Units)激活函数，构建一对完全连接层（layers），以及一个有着十个节点（node）、指明了输出logits模型的线性层。

每一层都创建于一个唯一的 `tf.name_scope` 之下，创建于该作用域之下的所有元素都将带有其前缀。

```
1. with tf.name_scope('hidden1') as scope:
```

在定义的作用域中，每一层所使用的权重和偏差都在 `tf.Variable` 实例中生成，并且包含了各自期望的shape。

```

1. weights = tf.Variable(
2.     tf.truncated_normal([IMAGE_PIXELS, hidden1_units],
3.                         stddev=1.0 / math.sqrt(float(IMAGE_PIXELS))),
4.     name='weights')
5. biases = tf.Variable(tf.zeros([hidden1_units]),
6.                      name='biases')

```

例如，当这些层是在 `hidden1` 作用域下生成时，赋予权重变量的独特名称将会是“ `hidden1/weights` ”。

每个变量在构建时，都会获得初始化操作（`initializer ops`）。

在这种最常见的情况下，通过 `tf.truncated_normal` 函数初始化权重变量，给赋予的shape则是一个二维tensor，其中第一个维度代表该层中权重变量所连接（`connect from`）的单元数量，第二个维度代表该层中权重变量所连接到的（`connect to`）单元数量。对于名叫 `hidden1` 的第一层，相应的维度则是 `[IMAGE_PIXELS, hidden1_units]`，因为权重变量将图像输入连接到 `hidden1` 层。`tf.truncated_normal` 初始函数将根据所得到的均值和标准差，生成一个随机分布。

然后，通过 `tf.zeros` 函数初始化偏差变量（`biases`），确保所有偏差的起始值都是0，而它们的shape则是其在该层中所接到的（`connect to`）单元数量。

图表的三个主要操作，分别是两个 `tf.nn.relu` 操作，它们中嵌入了隐藏层所需的 `tf.matmul`；以及 logits模型所需的另外一个 `tf.matmul`。三者依次生成，各自的 `tf.Variable` 实例则与输入占位符或下一层的输出tensor所连接。

```

1. hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)

```

```

1. hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)

```

```

1. logits = tf.matmul(hidden2, weights) + biases

```

最后，程序会返回包含了输出结果的 `logits` Tensor。

损失（Loss）

`loss()` 函数通过添加所需的损失操作，进一步构建图表。

首先，`labels_placeholder` 中的值，将被编码为一个含有1-hot values的Tensor。例如，如果类标识符为“3”，那么该值就会被转换为：

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

```

1. batch_size = tf.size(labels)

```

```

2. labels = tf.expand_dims(labels, 1)
3. indices = tf.expand_dims(tf.range(0, batch_size, 1), 1)
4. concated = tf.concat(1, [indices, labels])
5. onehot_labels = tf.sparse_to_dense(
6.     concated, tf.pack([batch_size, NUM_CLASSES]), 1.0, 0.0)

```

之后，又添加一个 `tf.nn.softmax_cross_entropy_with_logits` 操作，用来比较 `inference()` 函数与1-hot标签所输出的logits Tensor。

```

1. cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits,
2.                                                         onehot_labels,
3.                                                         name='xentropy')

```

然后，使用 `tf.reduce_mean` 函数，计算batch维度（第一维度）下交叉熵（cross entropy）的平均值，将该值作为总损失。

```

1. loss = tf.reduce_mean(cross_entropy, name='xentropy_mean')

```

最后，程序会返回包含了损失值的Tensor。

注意：交叉熵是信息理论中的概念，可以让我们描述如果基于已有事实，相信神经网络所做的推测最坏会导致什么结果。更多详情，请查阅博文《可视化信息理论》(<http://colah.github.io/posts/2015-09-Visual-Information/>)

训练

`training()` 函数添加了通过梯度下降（gradient descent）将损失最小化所需的操作。

首先，该函数从 `loss()` 函数中获取损失Tensor，将其交给 `tf.scalar_summary`，后者在与 `SummaryWriter`（见下文）配合使用时，可以向事件文件（events file）中生成汇总值（summary values）。在本篇教程中，每次写入汇总值时，它都会释放损失Tensor的当前值（snapshot value）。

```

1. tf.scalar_summary(loss.op.name, loss)

```

接下来，我们实例化一个 `tf.train.GradientDescentOptimizer`，负责按照所要求的学习效率（learning rate）应用梯度下降法（gradients）。

```

1. optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)

```

之后，我们生成一个变量用于保存全局训练步骤（global training step）的数值，并使用 `minimize()` 函数更新系统中的三角权重（triangle weights）、增加全局步骤的操作。根据惯例，这个操作被称为 `train_op`，是TensorFlow会话（session）诱发一个完整训练步骤所必须运行的操作（见下文）。

```
1. global_step = tf.Variable(0, name='global_step', trainable=False)
2. train_op = optimizer.minimize(loss, global_step=global_step)
```

最后，程序返回包含了训练操作（`training op`）输出结果的Tensor。

训练模型

一旦图表构建完毕，就通过 `fully_connected_feed.py` 文件中的用户代码进行循环地迭代式训练和评估。

图表

在 `run_training()` 这个函数的一开始，是一个Python语言中的 `with` 命令，这个命令表明所有已经构建的操作都要与默认的 `tf.Graph` 全局实例关联起来。

```
1. with tf.Graph().as_default():
```

`tf.Graph` 实例是一系列可以作为整体执行的操作。TensorFlow的大部分场景只需要依赖默认图表一个实例即可。

利用多个图表的更加复杂的使用场景也是可能的，但是超出了本教程的范围。

会话

完成全部的构建准备、生成全部所需的操作之后，我们就可以创建一个 `tf.Session`，用于运行图表。

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

另外，也可以利用 `with` 代码块生成 `Session`，限制作用域：

```
1. with tf.Session() as sess:
```

`Session` 函数中没有传入参数，表明该代码将会依附于（如果还没有创建会话，则会创建新的会话）默认的本地会话。

生成会话之后，所有 `tf.Variable` 实例都会立即通过调用各自初始化操作中的 `sess.run()` 函数进行初始化。

```
1. init = tf.initialize_all_variables()
```

```
2. sess.run(init)
```

`sess.run()` 方法将会运行图表中与作为参数传入的操作相对应的完整子集。在初次调用时，`init` 操作只包含了变量初始化程序 `tf.group`。图表的其他部分不会在这里，而是在下面的训练循环运行。

训练循环

完成会话中变量的初始化之后，就可以开始训练了。

训练的每一步都是通过用户代码控制，而能有效训练的最简单循环就是：

```
1. for step in xrange(max_steps):
2.     sess.run(train_op)
```

但是，本教程中的例子要更为复杂一点，原因是我们必须把输入的数据根据每一步的情况进行切分，以匹配之前生成的占位符。

向图表提供反馈

执行每一步时，我们的代码会生成一个反馈字典（feed dictionary），其中包含对应步骤中训练所要使用的例子，这些例子的哈希键就是其所代表的占位符操作。

`fill_feed_dict` 函数会查询给定的 `DataSet`，索要下一批次 `batch_size` 的图像和标签，与占位符相匹配的Tensor则会包含下一批次的图像和标签。

```
1. images_feed, labels_feed = data_set.next_batch(FLAGS.batch_size)
```

然后，以占位符为哈希键，创建一个Python字典对象，键值则是其代表的反馈Tensor。

```
1. feed_dict = {
2.     images_placeholder: images_feed,
3.     labels_placeholder: labels_feed,
4. }
```

这个字典随后作为 `feed_dict` 参数，传入 `sess.run()` 函数中，为这一步的训练提供输入样例。

检查状态

在运行 `sess.run` 函数时，要在代码中明确其需要获取的两个值：`[train_op, loss]`。

```
1. for step in xrange(FLAGS.max_steps):
2.     feed_dict = fill_feed_dict(data_sets.train,
```

```

3.             images_placeholder,
4.             labels_placeholder)
5.     _, loss_value = sess.run([train_op, loss],
6.                             feed_dict=feed_dict)

```

因为要获取这两个值，`sess.run()` 会返回一个有两个元素的元组。其中每一个 `Tensor` 对象，对应了返回的元组中的numpy数组，而这些数组中包含了当前这步训练中对应Tensor的值。由于 `train_op` 并不会产生输出，其在返回的元组中的对应元素就是 `None`，所以会被抛弃。但是，如果模型在训练中出现偏差，`loss` Tensor的值可能会变成NaN，所以我们要获取它的值，并记录下来。

假设训练一切正常，没有出现NaN，训练循环会每隔100个训练步骤，就打印一行简单的状态文本，告知用户当前的训练状态。

```

1. if step % 100 == 0:
2.     print 'Step %d: loss = %.2f (%.3f sec)' % (step, loss_value, duration)

```

状态可视化

为了释放TensorBoard所使用的事件文件（events file），所有的即时数据（在这里只有一个）都要在图表构建阶段合并至一个操作（op）中。

```

1. summary_op = tf.merge_all_summaries()

```

在创建好会话（session）之后，可以实例化一个 `tf.train.SummaryWriter`，用于写入包含了图表本身和即时数据具体值的事件文件。

```

1. summary_writer = tf.train.SummaryWriter(FLAGS.train_dir,
2.                                         graph_def=sess.graph_def)

```

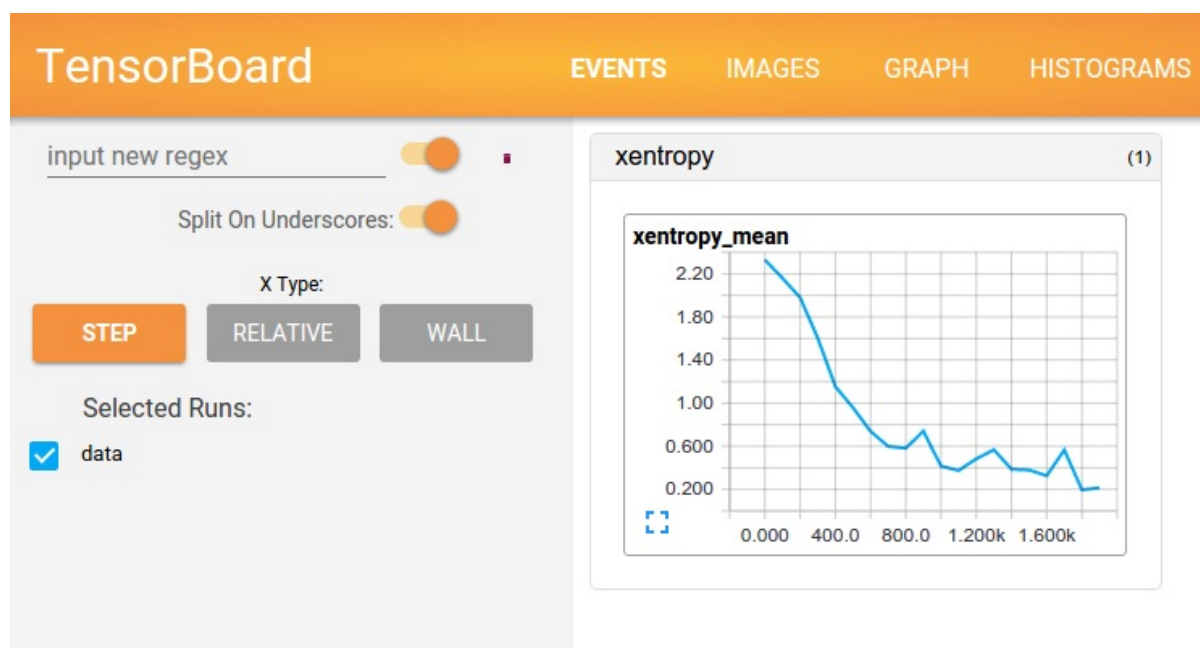
最后，每次运行 `summary_op` 时，都会往事件文件中写入最新的即时数据，函数的输出会传入事件文件读写器（writer）的 `add_summary()` 函数。。

```

1. summary_str = sess.run(summary_op, feed_dict=feed_dict)
2. summary_writer.add_summary(summary_str, step)

```

事件文件写入完毕之后，可以就训练文件夹打开一个TensorBoard，查看即时数据的情况。



注意：了解更多如何构建并运行TensorBoard的信息，请查看相关教程[Tensorboard：训练过程可视化](#)。

保存检查点（checkpoint）

为了得到可以用来后续恢复模型以进一步训练或评估的检查点文件（checkpoint file），我们实例化一个 `tf.train.Saver`。

```
1. saver = tf.train.Saver()
```

在训练循环中，将定期调用 `saver.save()` 方法，向训练文件夹中写入包含了当前所有可训练变量值得检查点文件。

```
1. saver.save(sess, FLAGS.train_dir, global_step=step)
```

这样，我们以后就可以使用 `saver.restore()` 方法，重载模型的参数，继续训练。

```
1. saver.restore(sess, FLAGS.train_dir)
```

评估模型

每隔一千个训练步骤，我们的代码会尝试使用训练数据集与测试数据集，对模型进行评估。`do_eval` 函数会被调用三次，分别使用训练数据集、验证数据集和测试数据集。

```

1. print 'Training Data Eval:'
2. do_eval(sess,
3.         eval_correct,
4.         images_placeholder,
5.         labels_placeholder,
6.         data_sets.train)
7. print 'Validation Data Eval:'
8. do_eval(sess,
9.         eval_correct,
10.        images_placeholder,
11.        labels_placeholder,
12.        data_sets.validation)
13. print 'Test Data Eval:'
14. do_eval(sess,
15.         eval_correct,
16.         images_placeholder,
17.         labels_placeholder,
18.         data_sets.test)

```

注意，更复杂的使用场景通常是，先隔绝 `data_sets.test` 测试数据集，只有在大量的超参数优化调整 (*hyperparameter tuning*) 之后才进行检查。但是，由于MNIST问题比较简单，我们在这里一次性评估所有的数据。

构建评估图表 (Eval Graph)

在打开默认图表 (Graph) 之前，我们应该先调用 `get_data(train=False)` 函数，抓取测试数据集。

```
1. test_all_images, test_all_labels = get_data(train=False)
```

在进入训练循环之前，我们应该先调用 `mnist.py` 文件中的 `evaluation` 函数，传入的logits和标签参数要与 `loss` 函数的一致。这样做事为了先构建Eval操作。

```
1. eval_correct = mnist.evaluation(logits, labels_placeholder)
```

`evaluation` 函数会生成 `tf.nn.in_top_k`

操作，如果在K个最有可能的预测中可以发现真的标签，那么这个操作就会将模型输出标记为正确。在本文中，我们把K的值设置为1，也就是只有在预测是真的标签时，才判定它是正确的。

```
1. eval_correct = tf.nn.in_top_k(logits, labels, 1)
```

评估图表的输出 (Eval Output)

之后，我们可以创建一个循环，往其中添加 `feed_dict`，并在调用 `sess.run()` 函数时传入 `eval_correct` 操作，目的就是用给定的数据集评估模型。

```
1. for step in xrange(steps_per_epoch):
2.     feed_dict = fill_feed_dict(data_set,
3.                                 images_placeholder,
4.                                 labels_placeholder)
5.     true_count += sess.run(eval_correct, feed_dict=feed_dict)
```

`true_count` 变量会累加所有 `in_top_k` 操作判定为正确的预测之和。接下来，只需要将正确测试的总数，除以例子总数，就可以得出准确率了。

```
1. precision = float(true_count) / float(num_examples)
2. print ' Num examples: %d Num correct: %d Precision @ 1: %0.02f' % (
3.     num_examples, true_count, precision)
```

原文: [TensorFlow Mechanics 101](#)

翻译: [bingjin](#)

校对: [LichAmnesia](#)

卷积神经网络

- [卷积神经网络](#)
 - [概述](#)
 - [目标](#)
 - [本教程的重点](#)
 - [模型架构](#)
 - [代码组织](#)
 - [CIFAR-10 模型](#)
 - [模型输入](#)
 - [模型预测](#)
 - [模型训练](#)
 - [开始执行并训练模型](#)
 - [评估模型](#)
 - [在多个GPU板卡上训练模型](#)
 - [在多个设备中设置变量和操作](#)
 - [启动并在多个GPU上训练模型](#)
 - [下一步](#)

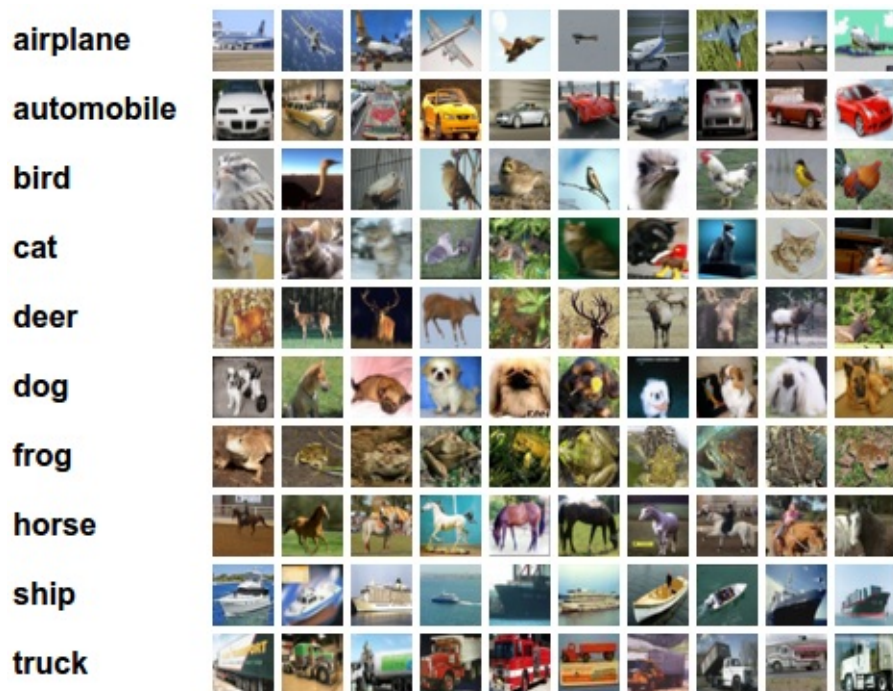
卷积神经网络

注意：本教程适用于对`Tensorflow`有丰富经验的用户，并假定用户有机器学习相关领域的专业知识和经验。

概述

对CIFAR-10 数据集的分类是机器学习中一个公开的基准测试问题，其任务是对一组大小为32x32的RGB图像进行分类，这些图像涵盖了10个类别：

飞机， 汽车， 鸟， 猫， 鹿， 狗， 青蛙， 马， 船以及卡车。



想了解更多信息请参考[CIFAR-10 page](#)，以及Alex Krizhevsky写的[技术报告](#)

目标

本教程的目标是建立一个用于识别图像的相对较小的卷积神经网络，在这一过程中，本教程会：

1. 着重于建立一个规范的网络组织结构，训练并进行评估；
2. 为建立更大规模更加复杂的模型提供一个范例

选择CIFAR-10是因为它的复杂程度足以用来检验TensorFlow中的大部分功能，并可将其扩展为更大的模型。与此同时由于模型较小所以训练速度很快，比较适合用来测试新的想法，检验新的技术。

本教程的重点

CIFAR-10 教程演示了在TensorFlow上构建更大更复杂模型的几个种重要内容：

- 相关核心数学对象，如[卷积](#)、[修正线性激活](#)、[最大池化](#)以及[局部响应归一化](#)；
- 训练过程中一些网络行为的[可视化](#)，这些行为包括输入图像、损失情况、网络行为的分布情况以及梯度；
- 算法学习参数的[移动平均值](#)的计算函数，以及在评估阶段使用这些平均值提高预测性能；
- 实现了一种机制，使得[学习率](#)随着时间的推移而递减；
- 为输入数据设计预存取[队列](#)，将磁盘延迟和高开销的图像预处理操作与模型分离开来处理；

我们也提供了模型的多GPU版本，用以表明：

- 可以配置模型后使其在多个GPU上并行的训练
- 可以在多个GPU之间共享和更新变量值

我们希望本教程给大家开了个头，使得在Tensorflow上可以为视觉相关工作建立更大型的CNN模型

模型架构

本教程中的模型是一个多层架构，由卷积层和非线性层(nonlinearities)交替多次排列后构成。这些层最终通过全连通层对接到softmax分类器上。这一模型除了最顶部的几层外，基本跟Alex Krizhevsky提出的模型一致。

在一个GPU上经过几个小时的训练后，该模型最高可以达到86%的精度。细节请查看[下面](#)的描述以及代码。模型中包含了1,068,298个学习参数，对一副图像进行分类大概需要19.5M个乘加操作。

代码组织

本教程的代码位于 `tensorflow/models/image/cifar10/` 。

文件	作用
<code>cifar10_input.py</code>	读取本地CIFAR-10的二进制文件格式的内容。
<code>cifar10.py</code>	建立CIFAR-10的模型。
<code>cifar10_train.py</code>	在CPU或GPU上训练CIFAR-10的模型。
<code>cifar10_multi_gpu_train.py</code>	在多GPU上训练CIFAR-10的模型。
<code>cifar10_eval.py</code>	评估CIFAR-10模型的预测性能。

CIFAR-10 模型

CIFAR-10 网络模型部分的代码位于

`cifar10.py` 。

完整的训练图中包含约765个操作。但是我们发现通过下面的模块来构造训练图可以最大限度的提高代码复用率：

1. **模型输入：**包括 `inputs()` 、 `distorted_inputs()` 等一些操作，分别用于读取CIFAR的图像并进行预处理，做为后续评估和训练的输入；
2. **模型预测：**包括 `inference()` 等一些操作，用于进行统计计算，比如在提供的图像进行分类；
adds operations that perform inference, i.e. classification, on supplied images.
3. **模型训练：**包括 `loss()` and `train()` 等一些操作，用于计算损失、计算梯度、进行变量更新以及呈现最终结果。

模型输入

输入模型是通过 `inputs()` 和 `distorted_inputs()` 函数建立起来的，这2个函数会从CIFAR-10二进制文件中读取图片文件，由于每个图片的存储字节数是固定的，因此可以使用 `tf.FixedLengthRecordReader` 函数。更多的关于 `Reader` 类的功能可以查看[Reading Data](#)。

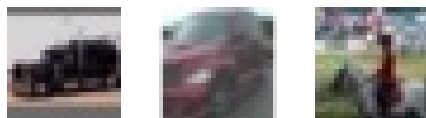
图片文件的处理流程如下：

- 图片会被统一裁剪到24x24像素大小，裁剪中央区域用于评估或[随机](#)裁剪用于训练；
- 图片会进行[近似的白化处理](#)，使得模型对图片的动态范围变化不敏感。

对于训练，我们另外采取了一系列随机变换的方法来人为的增加数据集的大小：

- 对图像进行[随机的左右翻转](#)；
- 随机变换[图像的亮度](#)；
- 随机变换[图像的对比度](#)；

可以在[Images](#)页的列表中查看所有可用的变换，对于每个原始图我们还附带了一个 `image_summary`，以便于在TensorBoard中查看。这对于检查输入图像是否正确十分有用。



从磁盘上加载图像并进行变换需要花费不少的处理时间。为了避免这些操作减慢训练过程，我们在16个独立的线程中并行进行这些操作，这16个线程被连续的安排在一个TensorFlow[队列](#)中。

模型预测

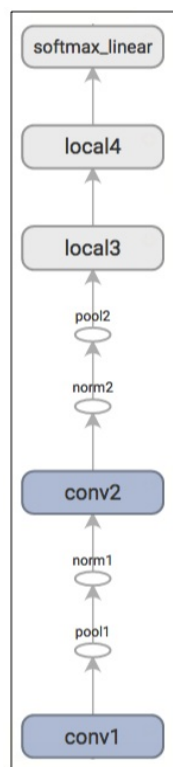
模型的预测流程由 `inference()` 构造，该函数会添加必要的操作步骤用于计算预测值的 *logits*，其对应的模型组织方式如下所示：

Layer 名称	描述
<code>conv1</code>	实现 卷积 以及 rectified linear activation 。
<code>pool1</code>	max pooling 。
<code>norm1</code>	局部响应归一化 。
<code>conv2</code>	卷积 and rectified linear activation 。
<code>norm2</code>	局部响应归一化 。
<code>pool2</code>	max pooling 。
<code>local13</code>	基于修正线性激活的全连接层 。
<code>local14</code>	基于修正线性激活的全连接层 。

softmax_linear

进行线性变换以输出 logits。

这里有一个由TensorBoard绘制的图形，用于描述模型建立过程中经过的步骤：



练习： `inference` 的输出是未归一化的logits，尝试使用 `tf.softmax()` 修改网络架构后返回归一化的预测值。

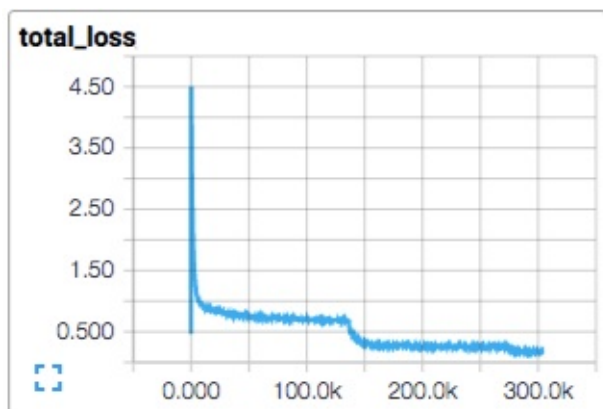
`inputs()` 和 `inference()` 函数提供了评估模型时所需的所有构件，现在我们把讲解的重点从构建一个模型转向训练一个模型。

练习： `inference()` 中的模型跟 *cuda-convnet* 中描述的 *CIFAR-10* 模型有些许不同，其差异主要在于其顶层不是全连接层而是局部连接层，可以尝试修改网络架构来准确的复制全连接模型。

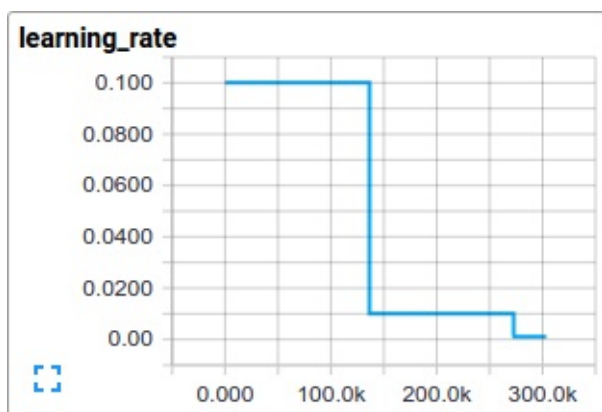
模型训练

训练一个可进行N维分类的网络的常用方法是使用[多项式逻辑回归](#)，又被叫做softmax 回归。Softmax 回归在网络的输出层上附加了一个softmax nonlinearity，并且计算归一化的预测值和label的1-hot encoding的交叉熵。在正则化过程中，我们会对所有学习变量应用[权重衰减损失](#)。模型的目标函数是求交叉熵损失和所有权重衰减项的和， `loss()` 函数的返回值就是这个值。

在TensorBoard中使用[scalar_summary](#)来查看该值的变化情况：



我们使用标准的梯度下降算法来训练模型（也可以在[Training](#)中看看其他方法），其学习率随时间以指数形式衰减。



`train()` 函数会添加一些操作使得目标函数最小化，这些操作包括计算梯度、更新学习变量（详细信息请查看 [GradientDescentOptimizer](#) ）。`train()` 函数最终会返回一个用以对一批图像执行所有计算的操作步骤，以便训练并更新模型。

开始执行并训练模型

我们已经把模型建立好了，现在通过执行脚本 `cifar10_train.py` 来启动训练过程。

```
1. python cifar10_train.py
```

注意：当第一次在CIFAR-10教程上启动任何任务时，会自动下载CIFAR-10数据集，该数据集大约有160M大小，因此第一次运行时泡杯咖啡小栖一会吧。

你应该可以看到如下类似的输出：

```
1. Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
2. 2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)
```

```

3. 2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)
4. 2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)
5. 2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)
6. 2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)
7. 2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)
8. ...

```

脚本会在每10步训练过程后打印出总损失值，以及最后一批数据的处理速度。下面是几点注释：

- 第一批数据会非常的慢（大概要几分钟时间），因为预处理线程要把20,000个待处理的CIFAR图像填充到重排队列中；
- 打印出来的损失值是最近一批数据的损失值的均值。请记住损失值是交叉熵和权重衰减项的和；
- 上面打印结果中关于一批数据的处理速度是在Tesla K40C上统计出来的，如果你运行在CPU上，性能会比此要低；

练习：当实验时，第一阶段的训练时间有时会非常的长，长到足以让人生厌。可以尝试减少初始化时初始填充到队列中图片数量来改变这种情况。在 `cifar10.py` 中搜索 `NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN` 并修改之。

`cifar10_train.py` 会周期性的在[检查点文件](#)中保存模型中的所有参数，但是不会对模型进行评估。`cifar10_eval.py` 会使用该检查点文件来测试预测性能（详见下面的描述：[评估模型](#)）。

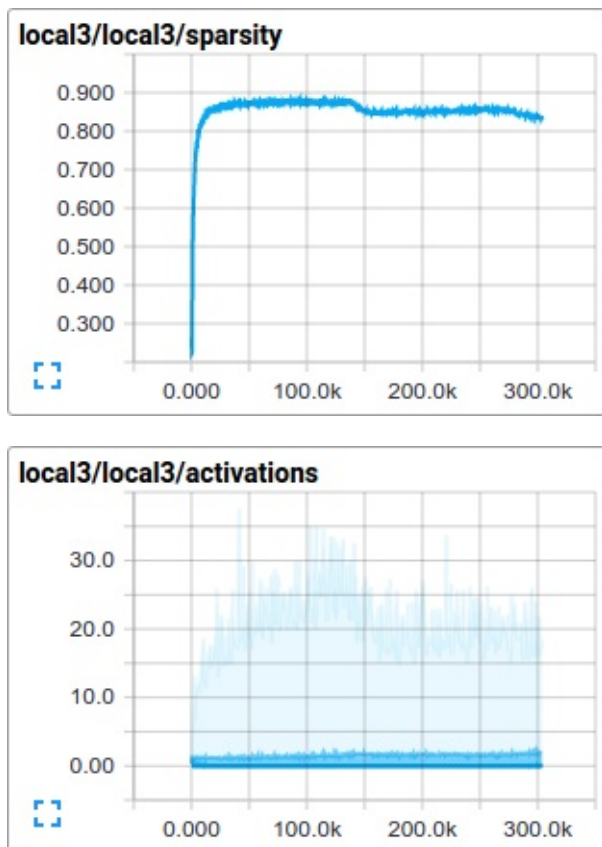
如果按照上面的步骤做下来，你应该已经开始训练一个CIFAR-10模型了。[恭喜你！](#)

`cifar10_train.py` 输出的终端信息中提供了关于模型如何训练的一些信息，但是我们可能希望了解更多关于模型训练时的信息，比如：

- 损失是真的在减小还是看到的只是噪声数据？
- 为模型提供的图片是否合适？
- 梯度、激活、权重的值是否合理？
- 当前的学习率是多少？

[TensorBoard](#)提供了该功能，可以通过 `cifar10_train.py` 中的 `SummaryWriter` 周期性的获取并显示这些数据。

比如我们可以在训练过程中查看 `local3` 的激活情况，以及其特征维度的稀疏情况：



相比于总损失，在训练过程中的单项损失尤其值得人们的注意。但是由于训练中使用的数据批量比较小，损失值中夹杂了相当多的噪声。在实践过程中，我们也发现相比于原始值，损失值的移动平均值显得更为有意义。请参阅脚本 `ExponentialMovingAverage` 了解如何实现。

评估模型

现在可以在另一部分数据集上来评估训练模型的性能。脚本文件 `cifar10_eval.py` 对模型进行了评估，利用 `inference()` 函数重构模型，并使用了在评估数据集所有10,000张CIFAR-10图片进行测试。最终计算出的精度为 $1:N$ ， N =预测值中置信度最高的一项与图片真实label匹配的频次。(It calculates the *precision at 1*: how often the top prediction matches the true label of the image)。

为了监控模型在训练过程中的改进情况，评估用的脚本文件会周期性的在最新的检查点文件上运行，这些检查点文件是由 `cifar10_train.py` 产生。

```
1. python cifar10_eval.py
```

注意：不要在同一块GPU上同时运行训练程序和评估程序，因为可能会导致内存耗尽。尽可能的在其它单独的GPU上运行评估程序，或者在同一块GPU上运行评估程序时先挂起训练程序。

你可能会看到如下所示输出：

```
1. 2015-11-06 08:30:44.391206: precision @ 1 = 0.860
2. ...
```

评估脚本只是周期性的返回`precision@1` (The script merely returns the `precision @ 1` periodically)–在该例中返回的准确率是86%。 `cifar10_eval.py` 同时也返回其它一些可以在TensorBoard中进行可视化的简要信息。可以通过这些简要信息在评估过程中进一步的了解模型。

训练脚本会为所有学习变量计算其**移动均值**, 评估脚本则直接将所有学习到的模型参数替换成对应的移动均值。这一替代方式可以在评估过程中提升模型的性能。

练习: 通过`precision @ 1`测试发现, 使用均值参数可以将预测性能提高约3%, 在 `cifar10_eval.py` 中尝试修改为不采用均值参数的方式, 并确认由此带来的预测性能下降。

在多个GPU板卡上训练模型

现代的工作站可能包含多个GPU进行科学计算。TensorFlow可以利用这一环境在多个GPU卡上运行训练程序。

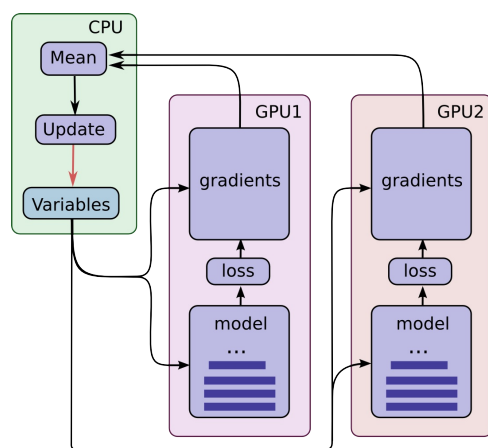
在并行、分布式的环境中进行训练, 需要对训练程序进行协调。对于接下来的描述, 术语**模型拷贝** (*model replica*) 特指在一个数据子集中训练出来的模型的一份拷贝。

如果天真的对模型参数的采用异步方式更新将会导致次优的训练性能, 这是因为我们可能会基于一个旧的模型参数的拷贝去训练一个模型。但与此相反采用完全同步更新的方式, 其速度将会变得和最慢的模型一样慢(Conversely, employing fully synchronous updates will be as slow as the slowest model replica.)。

在具有多个GPU的工作站中, 每个GPU的速度基本接近, 并且都含有足够的内存来运行整个CIFAR-10模型。因此我们选择以下方式来设计我们的训练系统:

- 在每个GPU上放置单独的模型副本;
- 等所有GPU处理完一批数据后再同步更新模型的参数;

下图示意了该模型的结构: :



可以看到，每一个GPU会用一批独立的数据计算梯度和估计值。这种设置可以非常有效的将一大批数据分割到各个GPU上。

这一机制要求所有GPU能够共享模型参数。但是众所周知在GPU之间传输数据非常的慢，因此我们决定在CPU上存储和更新所有模型的参数(对应图中绿色矩形的位置)。这样一来，GPU在处理一批新的数据之前会更新一遍的参数。

图中所有的GPU是同步运行的。所有GPU中的梯度会累积并求平均值(绿色方框部分)。模型参数会利用所有模型副本梯度的均值来更新。

在多个设备中设置变量和操作

在多个设备中设置变量和操作时需要做一些特殊的抽象。

我们首先需要把在单个模型拷贝中计算估计值和梯度的行为抽象到一个函数中。在代码中，我们称这个抽象对象为“tower”。对于每一个“tower”我们都需要设置它的两个属性：

- 在一个tower中为所有操作设定一个唯一的名称。 `tf.name_scope()` 通过添加一个范围前缀来提供该唯一名称。比如，第一个tower中的所有操作都会附带一个前缀 `tower_0`，示例：`tower_0/conv1/Conv2D`；
- 在一个tower中运行操作的优先硬件设备。 `tf.device()` 提供该信息。比如，在第一个tower中的所有操作都位于 `device('/gpu:0')` 范围中，暗含的意思是这些操作应该运行在第一块GPU上；

为了在多个GPU上共享变量，所有的变量都绑定在CPU上，并通过 `tf.get_variable()` 访问。可以查看[Sharing Variables](#)以了解如何共享变量。

启动并在多个GPU上训练模型

如果你的机器上安装有多块GPU，你可以通过使用 `cifar10_multi_gpu_train.py` 脚本来加速模型训练。该脚本是训练脚本的一个变种，使用多个GPU实现模型并行训练。

```
1. python cifar10_multi_gpu_train.py --num_gpus=2
```

训练脚本的输出如下所示：

```
1. Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
2. 2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)
3. 2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)
4. 2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)
5. 2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)
6. 2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)
7. 2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)
8. ...
```

需要注意的是默认的GPU使用数是1，此外，如果你的机器上只有一个GPU，那么所有的计算都只会在一个GPU上运行，即便你可能设置的是N个。

练习： `cifar10_train.py` 中的批处理大小默认配置是128。尝试在2个GPU上运行 `cifar10_multi_gpu_train.py` 脚本，并且设定批处理大小为64，然后比较2种方式的训练速度。

下一步

恭喜你！ 你已经完成了CIFAR-10教程。

如果你对开发和训练自己的图像分类系统感兴趣，我们推荐你新建一个基于该教程的分支，并修改其中的内容以建立解决您问题的图像分类系统。

练习：下载[Street View House Numbers \(SVHN\)](#) 数据集。新建一个CIFAR-10教程的分支，并将输入数据替换成SVHN。尝试改变网络结构以提高预测性能。

原文：[Convolutional Neural Networks](#) 翻译：[oskycar](#) 校对：[KK4SBB](#)

字词的向量表示

- [Vector Representations of Words](#)
 - [亮点](#)
 - [动机：为什么需要学习 Word Embeddings?](#)
 - [处理噪声对比训练](#)
 - [Skip-gram 模型](#)
 - [建立图形](#)
 - [训练模型](#)
 - [嵌套学习结果可视化](#)
 - [嵌套学习的评估：类比推理](#)
 - [优化实现](#)
 - [总结](#)

Vector Representations of Words

在本教程我们来看一下[Mikolov et al](#)中提到的word2vec模型。该模型是用于学习文字的向量表示，称之为“word embedding”。

亮点

本教程意在展现出在TensorFlow中构建word2vec模型有趣、本质的部分。

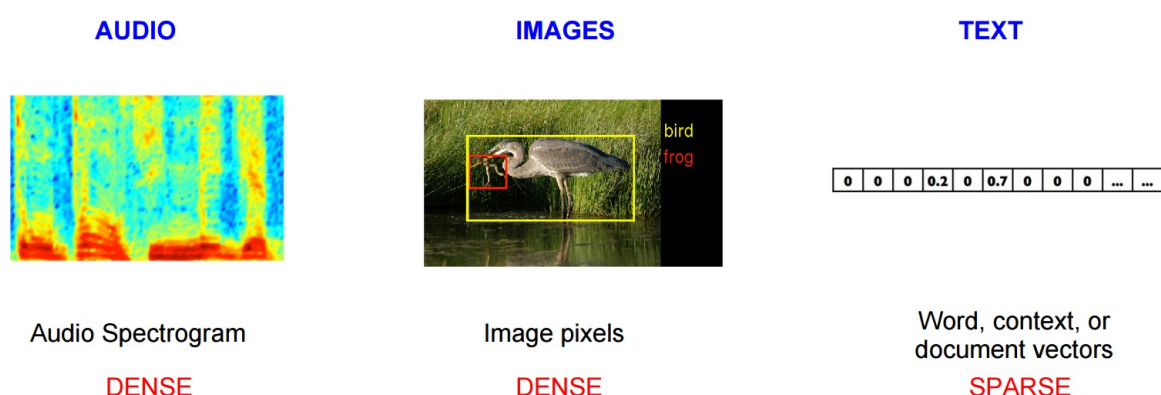
- 我们从我们为何需要使用向量表示文字开始。
- 我们通过直观地例子观察模型背后的本质，以及它是如何训练的（通过一些数学方法评估）。
- 同时我们也展示了TensorFlow对该模型的简单实现。
- 最后，我们着眼于让给这个简单版本的模型表现更好。

我们会在教程的推进中循序渐进地解释代码，但是如果你更希望直入主题，可以在[tensorflow/g3doc/tutorials/word2vec/word2vec_basic.py](#)查看到一个最简单的实现。这个基本的例子提供的代码可以完成下载一些数据，简单训练后展示结果。一旦你觉得已经完全掌握了这个简单版本，你可以查看[tensorflow/models/embedding/word2vec.py](#)，这里提供了一些更复杂的实现，同时也展示了TensorFlow的一些更进阶的特性，比如如何更高效地使用线程将数据送入文本模型，再比如如何在训练中设置检查点等等。

但是首先，让我们来看一下为何需要学习word embeddings。如果你对word embeddings相关内容已经是专家了，那么请安心跳过本节内容，直接深入细节干一些脏活吧。

动机：为什么需要学习 Word Embeddings?

通常图像或音频系统处理的是由图片中所有单个原始像素点强度值或者音频中功率谱密度的强度值，把它们编码成丰富、高纬度的向量数据集。对于物体或语音识别这一类的任务，我们所需的全部信息已经都存储在原始数据中（显然人类本身就是依赖原始数据进行日常的物体或语音识别的）。然后，自然语言处理系统通常将词汇作为离散的单一符号，例如“cat”一词或可表示为 `Id537`，而“dog”一词或可表示为 `Id143`。这些符号编码毫无规律，无法提供不同词汇之间可能存在的关联信息。换句话说，在处理关于“dogs”一词的信息时，模型将无法利用已知的关于“cats”的信息（例如，它们都是动物，有四条腿，可作为宠物等等）。可见，将词汇表达为上述的独立离散符号将进一步导致数据稀疏，使我们在训练统计模型时不得不寻求更多的数据。而词汇的向量表示将克服上述的难题。



向量空间模型 (VSMs) 将词汇表达（嵌套）于一个连续的向量空间中，语义近似的词汇被映射为相邻的数据点。向量空间模型在自然语言处理领域中有着漫长且丰富的历史，不过几乎所有利用这一模型的方法都依赖于

分布式假设，其核心思想为出现于上下文情景中的词汇都有相类似的语义。采用这一假设的研究方法大致分为以下两类：基于计数的方法（e.g.

潜在语义分析），

和 预测方法（e.g.

神经概率化语言模型）。

其中它们的区别在如下论文中又详细阐述

Baroni et al.，不过简而言之：基于计数的方法计算某词汇与其邻近词汇在一个大型语料库中共同出现的频率及其他统计量，然后将这些统计量映射到一个小型且稠密的向量中。预测方法则试图直接从某词汇的邻近词汇对其进行预测，在此过程中利用已经学习到的小型且稠密的嵌套向量。

Word2vec是一种可以进行高效率词嵌套学习的预测模型。其两种变体分别为：连续词袋模型（CBOW）及Skip-Gram模型。从算法角度看，这两种方法非常相似，其区别为CBOW根据源词上下文词汇（‘the cat sits on the’）来预测目标词汇（例如，‘mat’），而Skip-Gram模型做法相反，它通过目标词汇来预测源词汇。Skip-Gram模型采取CBOW的逆过程的动机在于：CBOW算法对于很多分布式信息进行了平滑处理（例如将一整段上下文信息视为一个单一观察量）。很多情况下，对

于小型的数据集，这一处理是有帮助的。相形之下，Skip-Gram模型将每个“上下文-目标词汇”的组合视为一个新观察量，这种做法在大型数据集中会更为有效。本教程余下部分将着重讲解Skip-Gram模型。

处理噪声对比训练

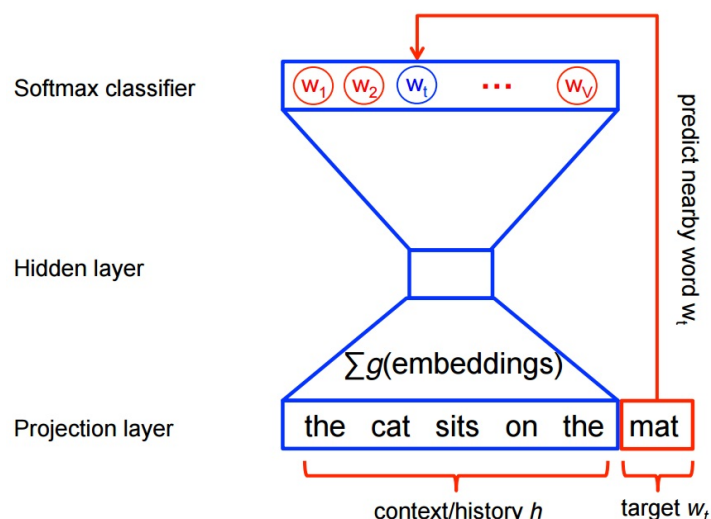
神经概率化语言模型通常使用[极大似然法](#)（ML）进行训练，其中通过 [softmax function](#) 来最大化当提供前一个单词 h （代表 “history”），后一个单词的概率 w_t （代表 “target”），

$$P(w_t|h) = \text{softmax}(\text{score}(w_t, h)) \\ = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}}.$$

当 $\text{score}(w_t, h)$ 计算了文字 w_t 和 上下文 h 的相容性（通常使用向量积）。我们使用对数似然函数来训练训练集的最大值，比如通过：

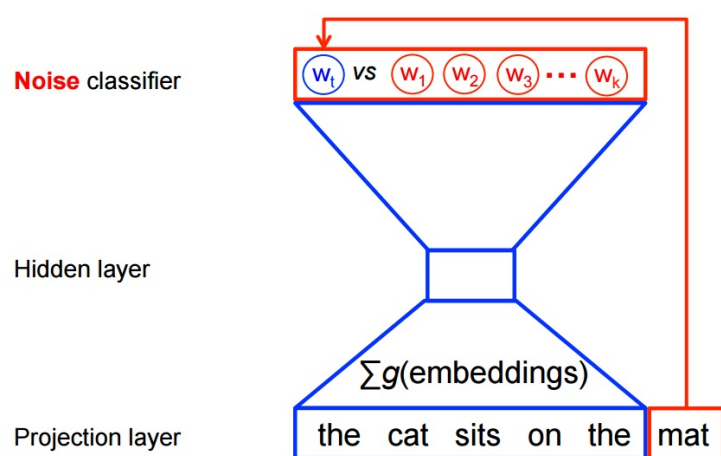
$$J_{\text{ML}} = \log P(w_t|h) \\ = \text{score}(w_t, h) - \log \left(\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\} \right)$$

这里提出了一个解决语言概率模型的合适的通用方法。然而这个方法实际执行起来开销非常大，因为我们需要去计算并正则化当前上下文环境 h 中所有其他 v 单词 w' 的概率得分，在每一步训练迭代中。



从另一个角度来说，当使用word2vec模型时，我们并不需要对概率模型中的所有特征进行学习。而CBOW模型和Skip-Gram模型为了避免这种情况发生，使用一个二分类器（逻辑回归）在同一个上下文

文环境里从 k 虚构的（噪声）单词 $\tilde{w}$ 区分出真正的目标单词 w_t 。我们下面详细阐述一下CBOW模型，对于Skip-Gram模型只要简单地做相反的操作即可。



从数学角度来说，我们的目标是对每个样本最大化：

$$J_{\text{NEG}} = \log Q_{\theta}(D = 1|w_t, h) + k \mathbb{E}_{\tilde{w} \sim P_{\text{noise}}} [\log Q_{\theta}(D = 0|\tilde{w}, h)]$$

其中 $Q_{\theta}(D = 1|w, h)$ 代表的是数据集在当前上下文 h ，根据所学习的嵌套向量 θ ，目标单词 w 使用二分类逻辑回归计算得出的概率。在实践中，我们通过在噪声分布中绘制比对文字来获得

近似的期望值（通过计算蒙特卡洛平均值）。

当真实地目标单词被分配到较高的概率，同时噪声单词的概率很低时，目标函数也就达到最大值了。

从技术层面来说，这种方法叫做

负抽样，而且使用这个损失函数在数学层面上也有很好的解释：这个更新过程也近似于softmax函数的更新。这在计算上将会有很大的优势，因为当计算这个损失函数时，只是有我们挑选出来的 k 个噪声单词，而没有使用整个语料库 V 。这使得训练变得非常快。我们实际上使用了与noise-contrastive estimation (NCE)介绍的非常相似的方法，这在TensorFlow中已经封装了一个很便捷的函数 `tf.nn.nce_loss()`。

让我们在实践中来直观地体会它是如何运作的！

Skip-gram 模型

下面来看一下这个数据集

```
the quick brown fox jumped over the lazy dog
```

我们首先对一些单词以及它们的上下文环境建立一个数据集。我们可以以任何合理的方式定义‘上下文’，而通常上这个方式是根据文字的句法语境的（使用语法原理的方式处理当前目标单词可以看一下这篇文献 [Levy et al.](#)，比如说把目标单词左边的内容当做一个‘上下文’，或者以目标单词右边的内容，等等。现在我们把目标单词的左右单词视作一个上下文，使用大小为1的窗口，这样就得到这样一个由（上下文，目标单词）组成的数据集：

```
([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...
```

前文提到Skip-Gram模型是把目标单词和上下文颠倒过来，所以在这个问题中，举个例子，就是用‘quick’来预测‘the’和‘brown’，用‘brown’预测‘quick’和‘fox’。因此这个数据集就变成由（输入，输出）组成的：

```
(quick, the), (quick, brown), (brown, quick), (brown, fox), ...
```

目标函数通常是对整个数据集建立的，但是本问题中要对每一个样本（或者是一个 `batch_size` 很小的样本集，通常设置为 `16 <= batch_size <= 512`）在同一时间执行特别的操作，称之为随机梯度下降 (SGD)。我们来看一下训练过程中每一步的执行。

假设用 t 表示上面这个例子中 `quick` 来预测 `the` 的训练的单个循环。用 `num_noise` 定义从噪声分布中挑选出来的噪声（相反的）单词的个数，通常使用一元分布 $P(w)$ 。为了简单起见，我们就定 `num_noise=1`，用 `sheep` 选作噪声词。接下来就可以计算每一对观察值和噪声值的损失函数了，每一个执行步骤就可表示为：

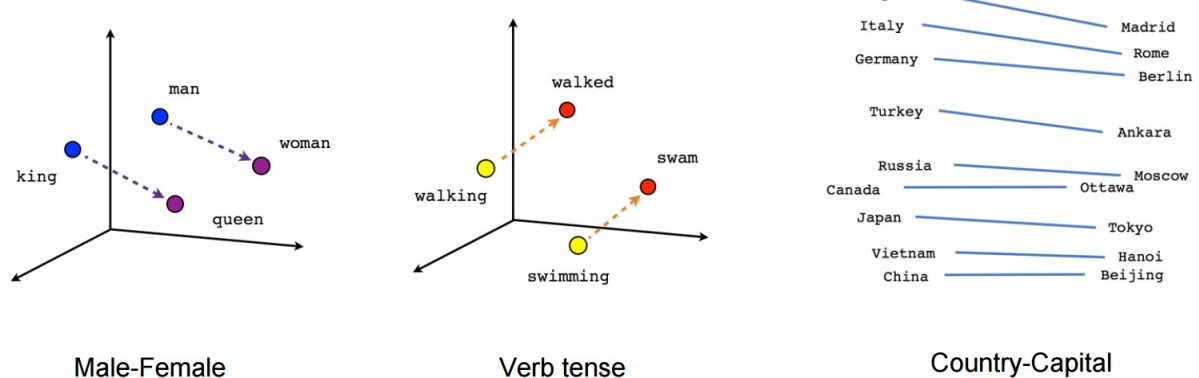
$$J_{\text{NEG}}^{(t)} = \log Q_{\theta}(D = 1 | \text{the, quick}) + \log(Q_{\theta}(D = 0 | \text{sheep, quick})).$$

整个计算过程的目标是通过更新嵌套参数 θ 来逼近目标函数（这个例子中就是使目标函数最大化）。为此我们要计算损失函数中嵌套参数 θ 的梯度，比如，

$$\frac{\partial}{\partial \theta} J_{\text{NEG}}$$

（幸好TensorFlow封装了工具函数可以简单调用！）。对于整个数据集，当梯度下降的过程中不断地更新参数，对应产生的效果就是不断地移动每个单词的嵌套向量，直到可以把真实单词和噪声单词很好得区分开。

我们可以把学习向量映射到2维中以便我们观察，其中用到的技术可以参考 [t-SNE 降维技术](#)。当我们用可视化的方式来观察这些向量，就可以很明显的获取单词之间语义信息的关系，这实际上是非常有用的。当我们第一次发现这样的诱导向量空间中，展示了一些特定的语义关系，这是非常有趣的，比如文字中 *male-female*, *gender* 甚至还有 *country-capital* 的关系，如下方的图所示（也可以参考 [Mikolov et al., 2013](#) 论文中的例子）。



这也解释了为什么这些向量在传统的NLP问题中可作为特性使用，比如用在对一个演讲章节打个标签，或者对一个专有名词的识别

（看看如下这个例子

[Collobert et al.](#) 或者

[Turian et al.](#)）。

不过现在让我们用它们来画漂亮的图表吧！

建立图形

这里谈得都是嵌套，那么先来定义一个嵌套参数矩阵。我们用唯一的随机值来初始化这个大矩阵。

```
1. embeddings = tf.Variable(
2.     tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

对噪声-比对的损失计算就使用一个逻辑回归模型。对此，我们需要对语料库中的每个单词定义一个权重值和偏差值。(也可称之为 `输出权重` 与之对应的 `输入嵌套值`)。定义如下。

```
1. nce_weights = tf.Variable(
2.     tf.truncated_normal([vocabulary_size, embedding_size],
3.                           stddev=1.0 / math.sqrt(embedding_size)))
4. nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

我们有了这些参数之后，就可以定义Skip-Gram模型了。简单起见，假设我们已经把语料库中的文字整型化了，这样每个整型代表一个单词(细节请查看 tensorflow/g3doc/tutorials/word2vec/word2vec_basic.py)。Skip-Gram模型有两个输入。一个是一组用整型表示的上下文单词，另一个是目标单词。给这些输入建立占位符节点，之后就可以填入数据了。

```
1. # 建立输入占位符
2. train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
3. train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

然后我们需要对批数据中的单词建立嵌套向量，TensorFlow提供了方便的工具函数。

```
1. embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

好了，现在我们有每个单词的嵌套向量，接下来就是使用噪声-比对的训练方式来预测目标单词。

```
1. # 计算 NCE 损失函数，每次使用负标签的样本。
2. loss = tf.reduce_mean(
3.     tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels,
4.                    num_sampled, vocabulary_size))
```

我们对损失函数建立了图形节点，然后我们需要计算相应梯度和更新参数的节点，比如说在这里我们会使用随机梯度下降法，TensorFlow也已经封装好了该过程。

```
1. # 使用 SGD 控制器。
2. optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
```

训练模型

训练的过程很简单，只要在循环中使用 `feed_dict` 不断给占位符填充数据，同时调用 `session.run` 即可。

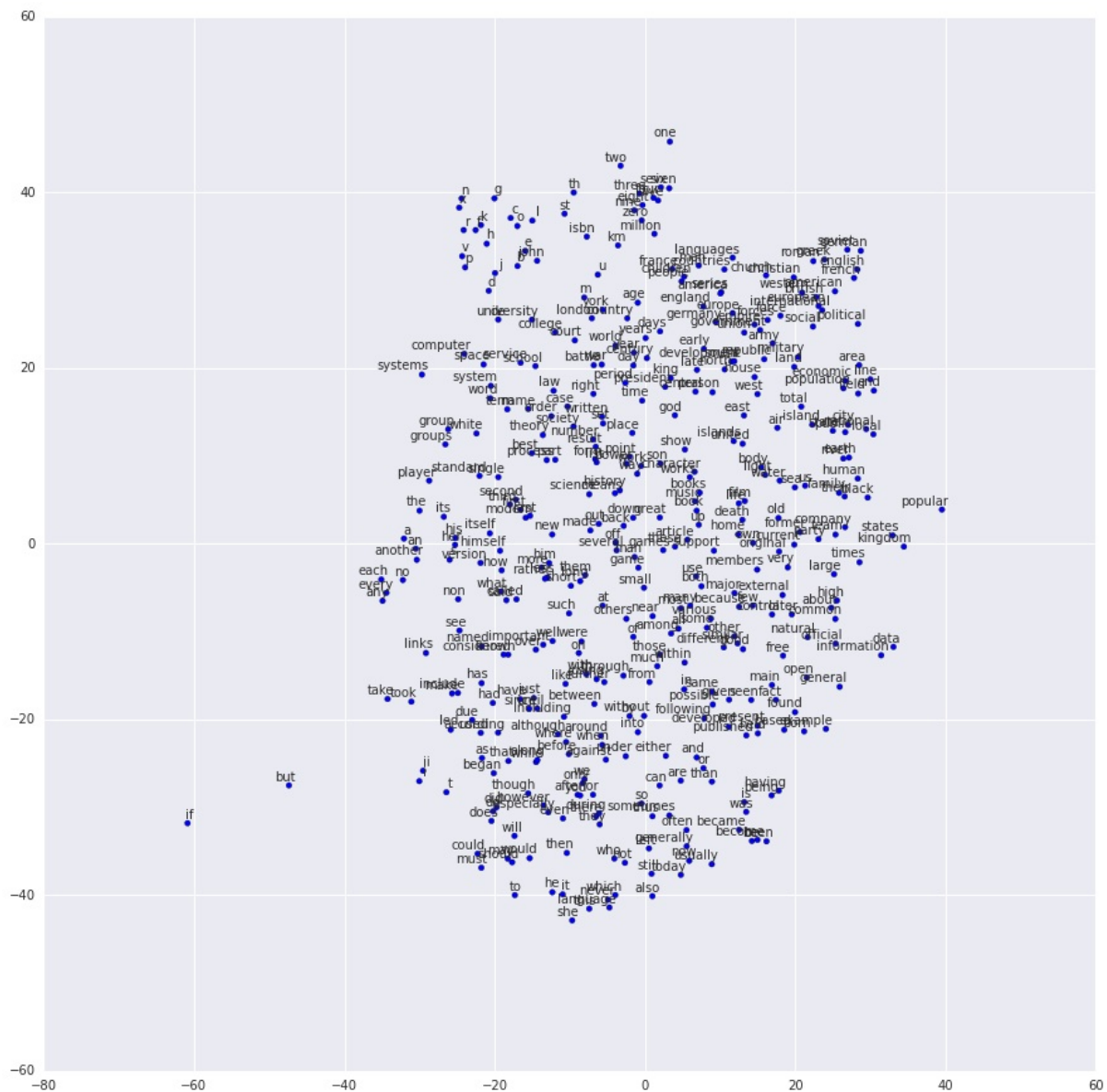
```
1. for inputs, labels in generate_batch(...):
2.     feed_dict = {training_inputs: inputs, training_labels: labels}
3.     _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)
```

完整地例子可参考

tensorflow/g3doc/tutorials/word2vec/word2vec_basic.py.

嵌套学习结果可视化

使用t-SNE来看一下嵌套学习完成的结果。



Et voila! 与预期的一样，相似的单词被聚类在一起。对word2vec模型更复杂的实现需要用到

TensorFlow一些更高级的特性，具体是实现可以参考[tensorflow/models/embedding/word2vec.py](https://www.tensorflow.org/models/embedding/word2vec.py)。

嵌套学习的评估： 类比推理

词嵌套在NLP的预测问题中是非常有用且使用广泛地。如果要检测一个模型是否可以成熟地区分词性或者区分专有名词的模型，最简单的办法就是直接检验它的预测词性、语义关系的能力，比如让它解决形如 `king is to queen as father is to ?` 这样的问题。这种方法叫做类比推理，可参

考Mikolov and colleagues，数据集下载地址为：

<https://word2vec.googlecode.com/svn/trunk/questions-words.txt>。

To see how we do this evaluation如何执行这样的评估，可以看 `build_eval_graph()` 和 `eval()` 这两个函数在下面源码中的使用

[tensorflow/models/embedding/word2vec.py](https://www.tensorflow.org/models/embedding/word2vec.py)。

超参数的选择对该问题解决的准确性有巨大的影响。想要模型具有很好的表现，需要有一个巨大的训练数据集，同时仔细调整参数的选择并且使用例如二次抽样的一些技巧。不过这些问题已经超出了本教程的范围。

优化实现

以上简单的例子展示了TensorFlow的灵活性。比如说，我们可以很轻松地用现成的 `tf.nn.sampled_softmax_loss()` 来代替 `tf.nn.nce_loss()` 构成目标函数。如果你对损失函数想做新的尝试，你可以用TensorFlow手动编写新的目标函数的表达式，然后用控制器执行计算。这种灵活性的价值体现在，当我们探索一个机器学习模型时，我们可以很快地遍历这些尝试，从中选出最优。

一旦你有了一个满意的模型结构，或许它就可以使实现运行地更高效（在短时间内覆盖更多的数据）。比如说，在本教程中使用的简单代码，实际运行速度都不错，因为我们使用Python来读取和填充数据，而这些在TensorFlow后台只需执行非常少的工作。如果你发现你的模型在输入数据时存在严重的瓶颈，你可以根据自己的实际问题自行实现一个数据阅读器，参考

[新的数据格式](#)。对于Skip-Gram

模型，我们已经完成了如下这个例子

[tensorflow/models/embedding/word2vec.py](https://www.tensorflow.org/models/embedding/word2vec.py)。

如果I/O问题对你的模型已经不再是个问题，并且想进一步地优化性能，或许你可以自行编写TensorFlow操作单元，详见

[添加一个新的操作](#)。相应的，我们也提供了Skip-Gram模型的例子

[tensorflow/models/embedding/word2vec_optimized.py](https://www.tensorflow.org/models/embedding/word2vec_optimized.py)。请自行调节以上几个过程的标准，使模型在每个运行阶段有更好地性能。

总结

在本教程中我们介绍了word2vec模型,它在解决词嵌套问题中具有良好的性能。我们解释了使用词嵌套模型的实用性,并且讨论了如何使用TensorFlow实现该模型的高效训练。总的来说,我们希望这个例子能够让你展示TensorFlow可以提供实验初期的灵活性,以及在后期优化模型时对模型内部的可操控性。

原文地址: [Vector Representation of Words](#) 翻译: [btpeter](#) 校对: waiwaizheng

递归神经网络

- [递归神经网络](#)
 - [介绍](#)
 - [语言模型](#)
 - [教程文件](#)
 - [下载及准备数据](#)
 - [模型](#)
 - [LSTM](#)
 - [截断反向传播](#)
 - [输入](#)
 - [损失函数](#)
 - [多个 LSTM 层堆叠](#)
 - [编译并运行代码](#)
 - [除此之外？](#)

递归神经网络

介绍

可以在 [this great article](#) 查看循环神经网络(RNN)以及 LSTM 的介绍。

语言模型

此教程将展示如何在高难度的语言模型中训练循环神经网络。该问题的目标是获得一个能确定语句概率的概率模型。为了做到这一点，通过之前已经给出的词语来预测后面的词语。我们将使用 PTB(Penn Tree Bank) 数据集，这是一种常用来衡量模型的基准，同时它比较小而且训练起来相对快速。

语言模型是很多有趣难题的关键所在，比如语音识别，机器翻译，图像字幕等。它很有意思—可以参看 [here](#)。

本教程的目的是重现 [Zaremba et al., 2014](#) 的成果，他们在 PTB 数据集上得到了很棒的结果。

教程文件

本教程使用的下面文件的目录是 `models/rnn/ptb`：

文件	作用
<code>ptb_word_lm.py</code>	在 PTB 数据集上训练一个语言模型。
<code>reader.py</code>	读取数据集。

下载及准备数据

本教程需要的数据在 `data/` 路径下，来源于 Tomas Mikolov 网站上的 PTB 数据集 `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`。

该数据集已经预先处理过并且包含了全部的 10000 个不同的词语，其中包括语句结束标记符，以及标记稀有词语的特殊符号 `(<unk>)`。我们在 `reader.py` 中转换所有的词语，让他们各自有唯一的整型标识符，便于神经网络处理。

模型

LSTM

模型的核心由一个 LSTM 单元组成，其可以在某时刻处理一个词语，以及计算语句可能的延续性的概率。网络的存储状态由一个零矢量初始化并在读取每一个词语后更新。而且，由于计算上的原因，我们将以 `batch_size` 为最小批量来处理数据。

基础的伪代码就像下面这样：

```
1. lstm = rnn_cell.BasicLSTMCell(lstm_size)
2. # 初始化 LSTM 存储状态.
3. state = tf.zeros([batch_size, lstm.state_size])
4.
5. loss = 0.0
6. for current_batch_of_words in words_in_dataset:
7.     # 每次处理一批词语后更新状态值.
8.     output, state = lstm(current_batch_of_words, state)
9.
10.    # LSTM 输出可用于产生下一个词语的预测
11.    logits = tf.matmul(output, softmax_w) + softmax_b
12.    probabilities = tf.nn.softmax(logits)
13.    loss += loss_function(probabilities, target_words)
```

截断反向传播

为使学习过程易于处理，通常的做法是将反向传播的梯度在（按时间）展开的步骤上照一个固定长度

(`num_steps`)截断。

通过在一次迭代中的每个时刻上提供长度为 `num_steps` 的输入和每次迭代完成之后反向传导，这会很容易实现。

一个简化版的用于计算图创建的截断反向传播代码：

```
1. # 一次给定的迭代中的输入占位符。
2. words = tf.placeholder(tf.int32, [batch_size, num_steps])
3.
4. lstm = rnn_cell.BasicLSTMCell(lstm_size)
5. # 初始化 LSTM 存储状态。
6. initial_state = state = tf.zeros([batch_size, lstm.state_size])
7.
8. for i in range(len(num_steps)):
9.     # 每处理一批词语后更新状态值。
10.    output, state = lstm(words[:, i], state)
11.
12.    # 其余的代码。
13.    # ...
14.
15. final_state = state
```

下面展现如何实现迭代整个数据集：

```
1. # 一个 numpy 数组，保存每一批词语之后的 LSTM 状态。
2. numpy_state = initial_state.eval()
3. total_loss = 0.0
4. for current_batch_of_words in words_in_dataset:
5.     numpy_state, current_loss = session.run([final_state, loss],
6.         # 通过上一次迭代结果初始化 LSTM 状态。
7.         feed_dict={initial_state: numpy_state, words: current_batch_of_words})
8.     total_loss += current_loss
```

输入

在输入 LSTM 前，词语 ID 被嵌入到了一个密集表示中(查看 [矢量表示教程](#))。这种方式允许模型高效地表示词语，也便于写代码：

```
1. # embedding_matrix 张量的形状是： [vocabulary_size, embedding_size]
2. word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

嵌入的矩阵会被随机地初始化，模型会学会通过数据分辨不同词语的意思。

损失函数

我们想使目标词语的平均负对数概率最小

$$\text{loss} = -\frac{1}{N} \sum_{i=1}^N \ln p_{\text{target}_i}$$

实现起来并非很难，而且函数 `sequence_loss_by_example` 已经有了，可以直接使用。

论文中的典型衡量标准是每个词语的平均困惑度 (perplexity)，计算式为

$$e^{-\frac{1}{N} \sum_{i=1}^N \ln p_{\text{target}_i}} = e^{\text{loss}}$$

同时我们会观察训练过程中的困惑度值 (perplexity)。

多个 LSTM 层堆叠

要想给模型更强的表达能力，可以添加多层 LSTM 来处理数据。第一层的输出作为第二层的输入，以此类推。

类 `MultiRNNCell` 可以无缝的将其实现：

```
1. lstm = rnn_cell.BasicLSTMCell(lstm_size)
2. stacked_lstm = rnn_cell.MultiRNNCell([lstm] * number_of_layers)
3.
4. initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
5. for i in range(len(num_steps)):
6.     # 每次处理一批词语后更新状态值。
7.     output, state = stacked_lstm(words[:, i], state)
8.
9.     # 其余的代码。
10.    # ...
11.
12. final_state = state
```

编译并运行代码

首先需要构建库，在 CPU 上编译：

```
1. bazel build -c opt tensorflow/models/rnn/ptb:ptb_word_lm
```

如果你有一个强大的 GPU，可以运行：

```
1. bazel build -c opt --config=cuda tensorflow/models/rnn/ptb:ptb_word_lm
```

运行模型：

```
1. bazel-bin/tensorflow/models/rnn/ptb/ptb_word_lm \  
2. --data_path=/tmp/simple-examples/data/ --alsologtostderr --model small
```

教程代码中有 3 个支持的模型配置参数：“small”，“medium”和“large”。它们指的是 LSTM 的大小，以及用于训练的超参数集。

模型越大，得到的结果应该更好。在测试集中 `small` 模型应该可以达到低于 120 的困惑度（perplexity），`large` 模型则是低于 80，但它可能花费数小时来训练。

除此之外？

还有几个优化模型的技巧没有提到，包括：

- 随时间降低学习率，
- LSTM 层间 dropout。

继续学习和更改代码以进一步改善模型吧。

原文：[Recurrent Neural Networks](#)

翻译：[WarIn](#)

校对：[HongyangWang](#)

曼德布洛特(Mandelbrot)集合

- 曼德布洛特(Mandelbrot)集合
 - 基本步骤
 - 会话 (session) 和变量 (variable) 初始化
 - 定义并运行计算

曼德布洛特 (Mandelbrot) 集合

虽然可视化曼德布洛特(Mandelbrot)集合与机器学习没有任何关系，但这对于将TensorFlow应用在数学更广泛的领域是一个有趣的例子。实际上，这是tensorflow一个非常直截了当的可视化运用。（我们最终也许会提供一种更加精心设计的运用方式来生成真正更加美丽的图像。）

说明：本教程使用了IPython的notebook。

基本步骤

首先，我们需要导入一些库。

```
1. # 导入仿真库
2. import tensorflow as tf
3. import numpy as np
4.
5. # 导入可视化库
6. import PIL.Image
7. from cStringIO import StringIO
8. from IPython.display import clear_output, Image, display
9. import scipy.ndimage as nd
```

现在我们将定义一个函数来显示迭代计算出的图像。

```
1. def DisplayFractal(a, fmt='jpeg'):
2.     """显示迭代计算出的彩色分形图像。"""
3.     a_cyclic = (6.28*a/20.0).reshape(list(a.shape)+[1])
4.     img = np.concatenate([10+20*np.cos(a_cyclic),
5.                           30+50*np.sin(a_cyclic),
6.                           155-80*np.cos(a_cyclic)], 2)
7.     img[a==a.max()] = 0
8.     a = img
9.     a = np.uint8(np.clip(a, 0, 255))
10.    f = StringIO()
11.    PIL.Image.fromarray(a).save(f, fmt)
```

```
12.     display(Image(data=f.getvalue()))
```

会话 (session) 和变量 (variable) 初始化

为了操作的方便，我们常常使用交互式会话 (interactive session)，但普通会话 (regular session) 也能正常使用。

```
1.     sess = tf.InteractiveSession()
```

我们可以自由的混合使用NumPy和TensorFlow，这一点非常方便。

```
1. # 使用NumPy创建一个在[-2,2]x[-2,2]范围内的2维复数数组
2.
3. Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]
4. Z = X+1j*Y
```

现在我们定义并初始化一组TensorFlow的张量 (tensors)。

```
1. xs = tf.constant(Z.astype("complex64"))
2. zs = tf.Variable(xs)
3. ns = tf.Variable(tf.zeros_like(xs, "float32"))
```

TensorFlow在使用之前需要你明确给定变量的初始值。

```
1. tf.initialize_all_variables().run()
```

定义并运行计算

现在我们指定更多的计算...

```
1. # 计算一个新值z: z^2 + x
2. zs_ = zs*zs + xs
3.
4. # 这个新值会发散吗？
5. not_diverged = tf.complex_abs(zs_) < 4
6.
7. # 更新zs并且迭代计算。
8. #
9. # 说明：在这些值发散之后，我们仍然在计算zs，这个计算消耗特别大！
10. #     如果稍微简单点，这里有更好的方法来处理。
11. #
```

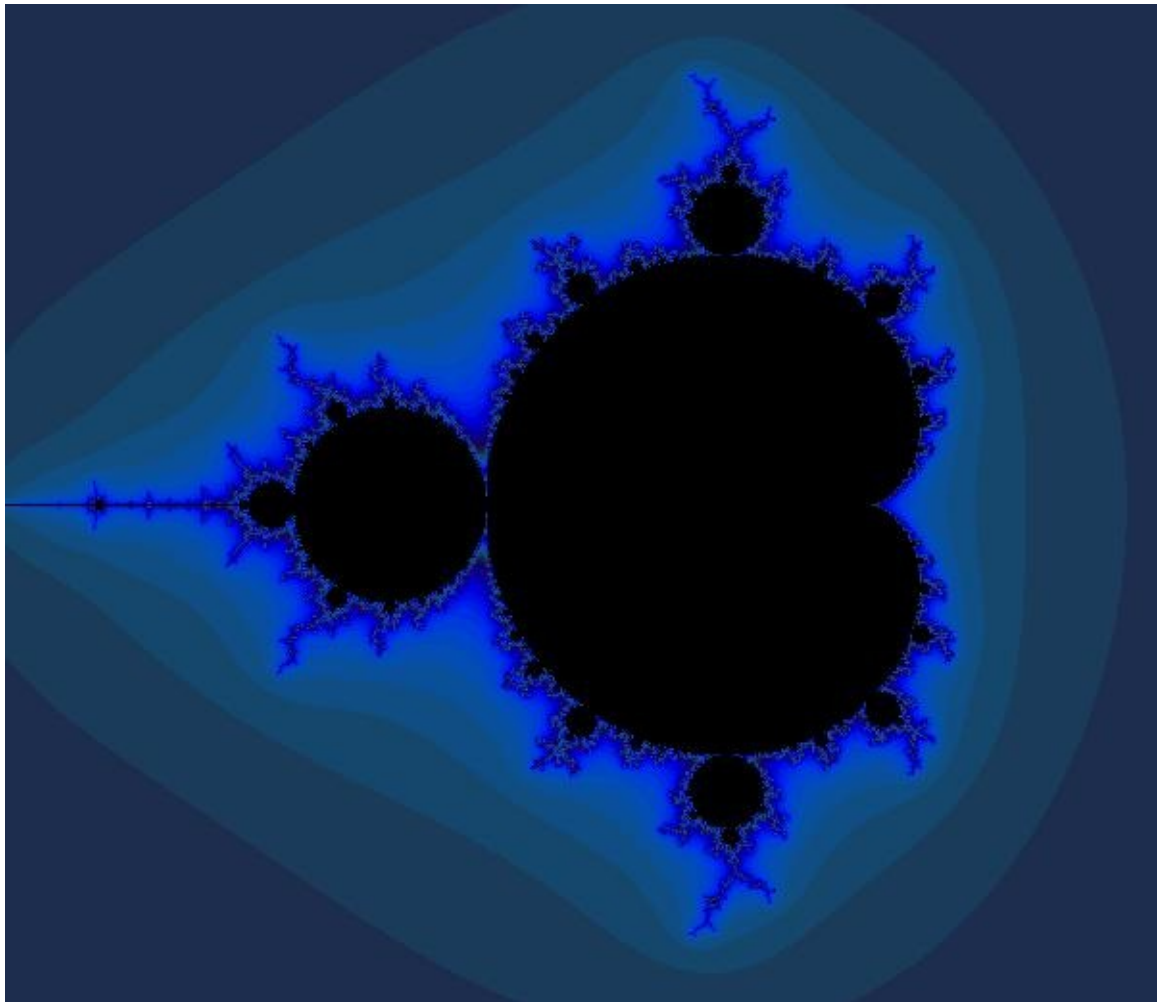
```
12. step = tf.group(  
13.     zs.assign(zs_),  
14.     ns.assign_add(tf.cast(not_diverged, "float32"))  
15. )
```

...继续执行几百个步骤

```
1. for i in range(200): step.run()
```

让我们看看我们得到了什么。

```
1. DisplayFractal(ns.eval())
```



结果不错！

原文: [Mandelbrot Set](#) 翻译: [ericxk](#) 校对: [tensorflowly](#)

偏微分方程

- 偏微分方程
 - 基本设置
 - 定义计算函数
 - 定义偏微分方程
 - 开始仿真

偏微分方程

TensorFlow 不仅仅是用来机器学习，它更可以用来模拟仿真。在这里，我们将通过模拟仿真几滴落入一块方形水池的雨点的例子，来引导您如何使用 **TensorFlow** 中的偏微分方程来模拟仿真的基本使用方法。

注：本教程最初是准备做为一个 **IPython** 的手册。

译者注：关于偏微分方程的相关知识，译者推荐读者查看 [网易公开课](#) 上的《麻省理工学院公开课：多变量微积分》课程。

基本设置

首先, 我们需要导入一些必要的引用。

```
1. #导入模拟仿真需要的库
2. import tensorflow as tf
3. import numpy as np
4.
5. #导入可视化需要的库
6. import PIL.Image
7. from cStringIO import StringIO
8. from IPython.display import clear_output, Image, display
```

然后, 我们还需要一个用于表示池塘表面状态的函数。

```
1. def DisplayArray(a, fmt='jpeg', rng=[0,1]):
2.     """Display an array as a picture."""
3.     a = (a - rng[0])/float(rng[1] - rng[0])*255
4.     a = np.uint8(np.clip(a, 0, 255))
5.     f = StringIO()
6.     PIL.Image.fromarray(a).save(f, fmt)
7.     display(Image(data=f.getvalue()))
```

最后, 为了方便演示, 这里我们需要打开一个 **TensorFlow** 的交互会话 (interactive

session)。当然为了以后能方便调用，我们可以把相关代码写到一个可以执行的**Python**文件中。

```
1. sess = tf.InteractiveSession()
```

定义计算函数

```
1. def make_kernel(a):
2.     """Transform a 2D array into a convolution kernel"""
3.     a = np.asarray(a)
4.     a = a.reshape(list(a.shape) + [1,1])
5.     return tf.constant(a, dtype=1)
6.
7. def simple_conv(x, k):
8.     """A simplified 2D convolution operation"""
9.     x = tf.expand_dims(tf.expand_dims(x, 0), -1)
10.    y = tf.nn.depthwise_conv2d(x, k, [1, 1, 1, 1], padding='SAME')
11.    return y[0, :, :, 0]
12.
13. def laplace(x):
14.     """Compute the 2D laplacian of an array"""
15.     laplace_k = make_kernel([[0.5, 1.0, 0.5],
16.                               [1.0, -6., 1.0],
17.                               [0.5, 1.0, 0.5]])
18.     return simple_conv(x, laplace_k)
```

定义偏微分方程

首先, 我们需要创建一个完美的 500×500 的正方形池塘, 就像是在现实中找到的一样。

```
1. N = 500
```

然后, 我们需要创建了一个池塘和几滴将要坠入池塘的雨滴。

```
1. # Initial Conditions -- some rain drops hit a pond
2.
3. # Set everything to zero
4. u_init = np.zeros([N, N], dtype="float32")
5. ut_init = np.zeros([N, N], dtype="float32")
6.
7. # Some rain drops hit a pond at random points
8. for n in range(40):
9.     a,b = np.random.randint(0, N, 2)
```

```
10. u_init[a,b] = np.random.uniform()  
11.  
12. DisplayArray(u_init, rng=[-0.1, 0.1])
```



现在，让我们来指定该微分方程的一些详细参数。

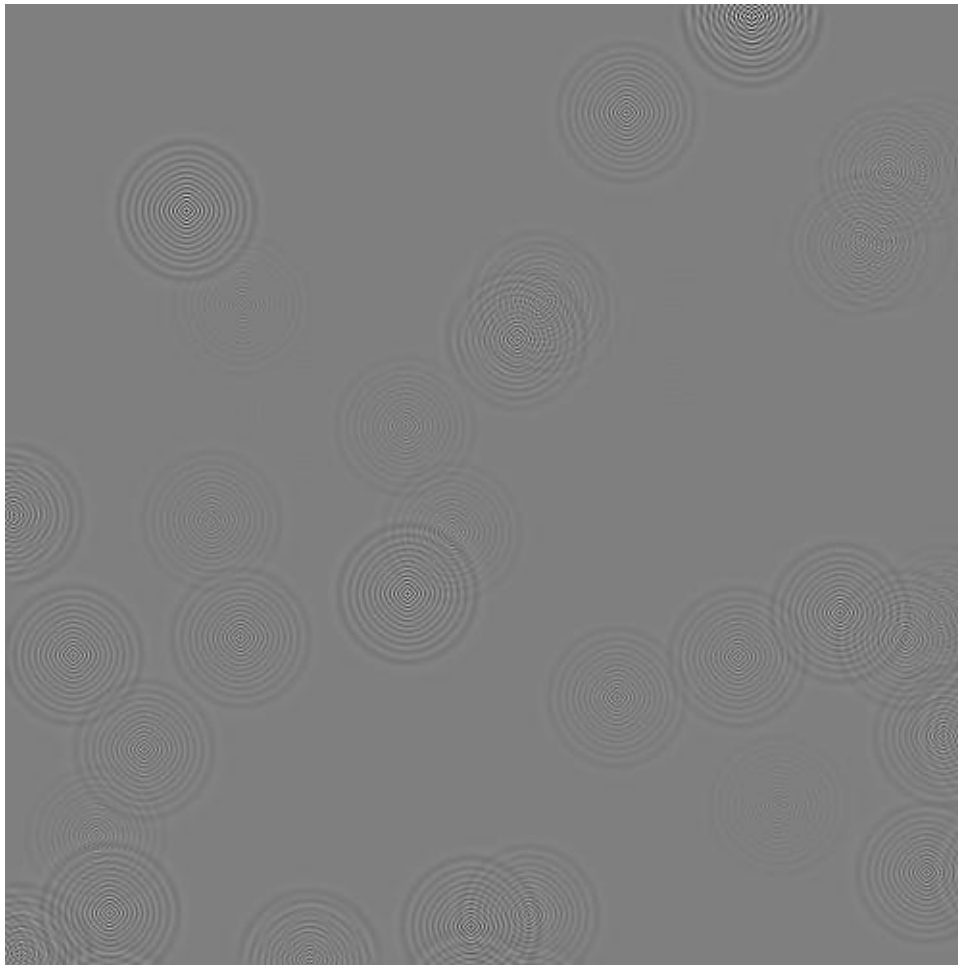
```
1. # Parameters:  
2. # eps -- time resolution  
3. # damping -- wave damping  
4. eps = tf.placeholder(tf.float32, shape=())  
5. damping = tf.placeholder(tf.float32, shape=())  
6.  
7. # Create variables for simulation state  
8. U = tf.Variable(u_init)  
9. Ut = tf.Variable(ut_init)  
10.  
11. # Discretized PDE update rules  
12. U_ = U + eps * Ut  
13. Ut_ = Ut + eps * (laplace(U) - damping * Ut)  
14.  
15. # Operation to update the state
```

```
16. step = tf.group(  
17.     U.assign(U_),  
18.     Ut.assign(Ut_))
```

开始仿真

为了能看清仿真效果，我们可以用一个简单的 **for** 循环来运行我们的仿真程序。

```
1. # Initialize state to initial conditions  
2. tf.initialize_all_variables().run()  
3.  
4. # Run 1000 steps of PDE  
5. for i in range(1000):  
6.     # Step simulation  
7.     step.run({eps: 0.03, damping: 0.04})  
8.     # Visualize every 50 steps  
9.     if i % 50 == 0:  
10.         clear_output()  
11.         DisplayArray(U.eval(), rng=[-0.1, 0.1])
```



看！！ 雨点落在池塘中,和现实中一样的泛起了涟漪。

原文链接:<http://tensorflow.org/tutorials/pdes/index.md> 翻译:@wangaicc 校对:@tensorfly

MNIST数据下载

- [MNIST 数据下载](#)
 - [教程 文件](#)
 - [准备数据](#)
 - [下载](#)
 - [解压 与 重构](#)
 - [数据集 对象](#)

MNIST 数据下载

源码: tensorflow/g3doc/tutorials/mnist/

本教程的目标是展示如何下载用于手写数字分类问题所要用到的（经典）MNIST数据集。

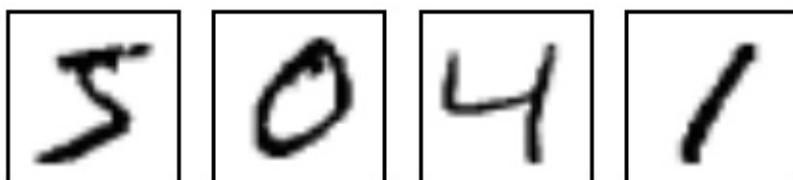
教程 文件

本教程需要使用以下文件：

文件	目的
<code>input_data.py</code>	下载用于训练和测试的MNIST数据集的源码

准备数据

MNIST是在机器学习领域中的一个经典问题。该问题解决的是把28x28像素的灰度手写数字图片识别为相应的数字，其中数字的范围从0到9。



更多详情，请参考 [Yann LeCun's MNIST page](#)
或 [Chris Olah's visualizations of MNIST](#).

下载

Yann LeCun's MNIST page

也提供了训练集与测试集数据的下载。

文件	内容
<code>train-images-idx3-ubyte.gz</code>	训练集图片 - 55000 张 训练图片, 5000 张 验证图片
<code>train-labels-idx1-ubyte.gz</code>	训练集图片对应的数字标签
<code>t10k-images-idx3-ubyte.gz</code>	测试集图片 - 10000 张 图片
<code>t10k-labels-idx1-ubyte.gz</code>	测试集图片对应的数字标签

在 `input_data.py` 文件中, `maybe_download()` 函数可以确保这些训练数据下载到本地文件夹中。

文件夹的名字在

`fully_connected_feed.py` 文件的顶部由一个标记变量指定, 你可以根据自己的需要进行修改。

解压 与 重构

这些文件本身并没有使用标准的图片格式储存, 并且需要使用 `input_data.py` 文件中 `extract_images()` 和 `extract_labels()` 函数来手动解压 (页面中有相关说明)。

图片数据将被解压成2维的tensor: `[image index, pixel index]`

其中每一项表示某一图片中特定像素的强度值, 范围从 `[0, 255]` 到 `[-0.5, 0.5]`。"image index"代表数据集中图片的编号, 从0到数据集的上限值。"pixel index"代表该图片中像素点得个数, 从0到图片的像素上限值。

以 `train-*` 开头的文件中包括60000个样本, 其中分割出55000个样本作为训练集, 其余的5000个样本作为验证集。因为所有数据集中28x28像素的灰度图片的尺寸为784, 所以训练集输出的tensor格式为 `[55000, 784]`。

数字标签数据被解压称1维的tensor: `[image index]`, 它定义了每个样本数值的类别分类。对于训练集的标签来说, 这个数据规模就是: `[55000]`。

数据集 对象

底层的源码将会执行下载、解压、重构图片和标签数据来组成以下的数据集对象:

数据集	目的
<code>data_sets.train</code>	55000 组 图片和标签, 用于训练。
<code>data_sets.validation</code>	5000 组 图片和标签, 用于迭代验证训练的准确性。
<code>data_sets.test</code>	10000 组 图片和标签, 用于最终测试训练的准确性。

执行 `read_data_sets()` 函数将会返回一个 `DataSet` 实例, 其中包含了以上三个数据集。函

数 `DataSet.next_batch()` 是用于获取以 `batch_size` 为大小的一个元组，其中包含了一组图片和标签，该元组会被用于当前的TensorFlow运算会话中。

```
1. images_feed, labels_feed = data_set.next_batch(FLAGS.batch_size)
```

原文地址: [MNIST Data Download](#) 翻译: [btpeter](#) 校对: waiwaizheng

运作方式

操作方式

- [总览](#)
- [变量: 创建、初始化、保存和加载](#)
- [TensorBoard: 可视化学习](#)
- [TensorBoard: 图表可视化](#)
- [读取数据](#)
- [线程和队列](#)
- [添加新的Op](#)
- [自定义数据读取](#)
- [使用gpu](#)
- [共享变量](#)

总览

- [综述 Overview](#)
 - [Variables: 创建, 初始化, 保存, 和恢复](#)
 - [TensorFlow 机制 101](#)
 - [TensorBoard: 学习过程的可视化](#)
 - [TensorBoard: 图的可视化](#)
 - [数据读入](#)
 - [线程和队列](#)
 - [添加新的 Op](#)
 - [自定义数据的 Readers](#)
 - [使用 GPUs](#)
 - [共享变量 Sharing Variables](#)

综述 Overview

Variables: 创建, 初始化, 保存, 和恢复

TensorFlow Variables 是内存中的容纳 tensor 的缓存。这一小节介绍了用它们在模型训练时 (during training) 创建、保存和更新模型参数(model parameters) 的方法。

[参看教程](#)

TensorFlow 机制 101

用 MNIST 手写数字识别作为一个小例子, 一步一步的将使用 TensorFlow 基础架构 (infrastructure) 训练大规模模型的细节做详细介绍。

[参看教程](#)

TensorBoard: 学习过程的可视化

对模型进行训练和评估时, TensorBoard 是一个很有用的可视化工具。此教程解释了创建和运行 TensorBoard 的方法, 和使用摘要操作(Summary ops)的方法, 通过添加摘要操作(Summary ops), 可以自动把数据传输到 TensorBoard 所使用的事件文件。

[参看教程](#)

TensorBoard：图的可视化

此教程介绍了在 TensorBoard 中使用可视化工具的方法，它可以帮助你理解张量流图的过程并 debug。

[参看教程](#)

数据读入

此教程介绍了把数据传入 TensorFlow 程序的三种主要的方法：Feeding, Reading 和 Preloading.

[参看教程](#)

线程和队列

此教程介绍 TensorFlow 中为了更容易进行异步和并发训练的各种不同结构(constructs)。

[参看教程](#)

添加新的 Op

TensorFlow 已经提供一整套节点操作(operation)，你可以在你的 graph 中随意使用它们，不过这里有关于添加自定义操作(custom op)的细节。

[参看教程。](#)

自定义数据的 Readers

如果你有相当大量的自定义数据集，可能你想要对 TensorFlow 的 Data Readers 进行扩展，使它能直接以数据自身的格式将其读入。

[参看教程。](#)

使用 GPUs

此教程描述了用多个 GPU 构建和运行模型的方法。

[参看教程](#)

共享变量 Sharing Variables

当在多 GPU 上部署大型的模型，或展开复杂的 LSTMs 或 RNNs 时，在模型构建代码的不同位置对许多相同的变量（Variable）进行读写常常是必须的。设计变量作用域（Variable Scope）机制的目的就是为了帮助上述任务的实现。

[参看教程](#)。

原文：[How-to](#)

翻译：[Terence Cooper](#)

校对：[lonlonago](#)

变量:创建、初始化、保存和加载

- [变量:创建、初始化、保存和加载](#)
 - [创建](#)
 - [初始化](#)
 - [由另一个变量初始化](#)
 - [自定义初始化](#)
 - [保存和加载](#)
 - [检查点文件](#)
 - [保存变量](#)
 - [恢复变量](#)
 - [选择存储和恢复哪些变量](#)

变量:创建、初始化、保存和加载

当训练模型时，用[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。建模时它们需要被明确地初始化，模型训练后它们必须被存储到磁盘。这些变量的值可在之后模型训练和分析是被加载。

本文档描述以下两个TensorFlow类。点击以下链接可查看完整的API文档：

- `tf.Variable` 类
- `tf.train.Saver` 类

创建

当创建一个[变量](#)时，你将一个 `张量` 作为初始值传入构造函数 `Variable()`。TensorFlow提供了一系列操作符来初始化张量，初始值是[常量或是随机值](#)。

注意，所有这些操作符都需要你指定张量的shape。那个形状自动成为变量的shape。变量的shape通常是固定的，但TensorFlow提供了高级的机制来重新调整其行列数。

```
1. # Create two variables.
2. weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
3.                        name="weights")
4. biases = tf.Variable(tf.zeros([200]), name="biases")
```

调用 `tf.Variable()` 添加一些操作(Op, operation)到graph:

- 一个 `Variable` 操作存放变量的值。
- 一个初始化op将变量设置为初始值。这事实上是一个 `tf.assign` 操作。

- 初始值的操作，例如示例中对 `biases` 变量的 `zeros` 操作也被加入了graph。

`tf.Variable` 的返回值是Python的 `tf.Variable` 类的一个实例。

初始化

变量的初始化必须在模型的其它操作运行之前先明确地完成。最简单的方法就是添加一个给所有变量初始化的操作，并在使用模型之前首先运行那个操作。

你或者可以从检查点文件中重新获取变量值，详见下文。

使用 `tf.initialize_all_variables()` 添加一个操作对变量做初始化。记得在完全构建好模型并加载之后再运行那个操作。

```
1. # Create two variables.
2. weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
3.                        name="weights")
4. biases = tf.Variable(tf.zeros([200]), name="biases")
5. ...
6. # Add an op to initialize the variables.
7. init_op = tf.initialize_all_variables()
8.
9. # Later, when launching the model
10. with tf.Session() as sess:
11.     # Run the init operation.
12.     sess.run(init_op)
13.     ...
14.     # Use the model
15.     ...
```

由另一个变量初始化

你有时候会需要用另一个变量的初始化值给当前变量初始化。由于 `tf.initialize_all_variables()` 是并行地初始化所有变量，所以在有这种需求的情况下需要小心。

用其它变量的值初始化一个新的变量时，使用其它变量的 `initialized_value()` 属性。你可以直接把已初始化的值作为新变量的初始值，或者把它当做tensor计算得到一个值赋予新变量。

```
1. # Create a variable with a random value.
2. weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
3.                        name="weights")
4. # Create another variable with the same value as 'weights'.
5. w2 = tf.Variable(weights.initialized_value(), name="w2")
6. # Create another variable with twice the value of 'weights'
```

```
7. w_twice = tf.Variable(weights.initialized_value() * 0.2, name="w_twice")
```

自定义初始化

`tf.initialize_all_variables()` 函数便捷地添加一个op来初始化模型的所有变量。你也可以给它传入一组变量进行初始化。详情请见[Variables Documentation](#)，包括检查变量是否被初始化。

保存和加载

最简单的保存和恢复模型的方法是使用 `tf.train.Saver` 对象。构造器给graph的所有变量，或是定义在列表里的变量，添加 `save` 和 `restore` ops。saver对象提供了方法来运行这些ops，定义检查点文件的读写路径。

检查点文件

变量存储在二进制文件里，主要包含从变量名到tensor值的映射关系。

当你创建一个 `Saver` 对象时，你可以选择性地为检查点文件中的变量挑选变量名。默认情况下，将每个变量 `Variable.name` 属性的值。

保存变量

用 `tf.train.Saver()` 创建一个 `Saver` 来管理模型中的所有变量。

```
1. # Create some variables.
2. v1 = tf.Variable(..., name="v1")
3. v2 = tf.Variable(..., name="v2")
4. ...
5. # Add an op to initialize the variables.
6. init_op = tf.initialize_all_variables()
7.
8. # Add ops to save and restore all the variables.
9. saver = tf.train.Saver()
10.
11. # Later, launch the model, initialize the variables, do some work, save the
12. # variables to disk.
13. with tf.Session() as sess:
14.     sess.run(init_op)
15.     # Do some work with the model.
16.     ..
17.     # Save the variables to disk.
18.     save_path = saver.save(sess, "/tmp/model.ckpt")
19.     print "Model saved in file: ", save_path
```

恢复变量

用同一个 `Saver` 对象来恢复变量。注意，当你从文件中恢复变量时，不需要事先对它们做初始化。

```
1. # Create some variables.
2. v1 = tf.Variable(..., name="v1")
3. v2 = tf.Variable(..., name="v2")
4. ...
5. # Add ops to save and restore all the variables.
6. saver = tf.train.Saver()
7.
8. # Later, launch the model, use the saver to restore variables from disk, and
9. # do some work with the model.
10. with tf.Session() as sess:
11.     # Restore variables from disk.
12.     saver.restore(sess, "/tmp/model.ckpt")
13.     print "Model restored."
14.     # Do some work with the model
15.     ...
```

选择存储和恢复哪些变量

如果你不给 `tf.train.Saver()` 传入任何参数，那么saver将处理graph中的所有变量。其中每一个变量都以变量创建时传入的名称被保存。

有时候在检查点文件中明确定义变量的名称很有用。举个例子，你也许已经训练得到了一个模型，其中有个变量命名为 `"weights"`，你想把它的值恢复到一个新的变量 `"params"` 中。

有时候仅保存和恢复模型的一部分变量很有用。再举个例子，你也许训练得到了一个5层神经网络，现在想训练一个6层的新模型，可以将之前5层模型的参数导入到新模型的前5层中。

你可以通过给 `tf.train.Saver()` 构造函数传入Python字典，很容易地定义需要保持的变量及对应名称：键对应使用的名称，值对应被管理的变量。

注意：

- 如果需要保存和恢复模型变量的不同子集，可以创建任意多个saver对象。同一个变量可被列入多个saver对象中，只有当saver的 `restore()` 函数被运行时，它的值才会发生改变。
- 如果你仅在session开始时恢复模型变量的一个子集，你需要对剩下的变量执行初始化op。详情请见 `tf.initialize_variables()`。

```
1. # Create some variables.
2. v1 = tf.Variable(..., name="v1")
3. v2 = tf.Variable(..., name="v2")
```

```
4. ...  
5. # Add ops to save and restore only 'v2' using the name "my_v2"  
6. saver = tf.train.Saver({"my_v2": v2})  
7. # Use the saver object normally after that.  
8. ...
```

原文链接: http://tensorflow.org/how_tos/variables/index.html

翻译: [赵屹华](#) 校对: [Wiki](#)

TensorBoard:可视化学习

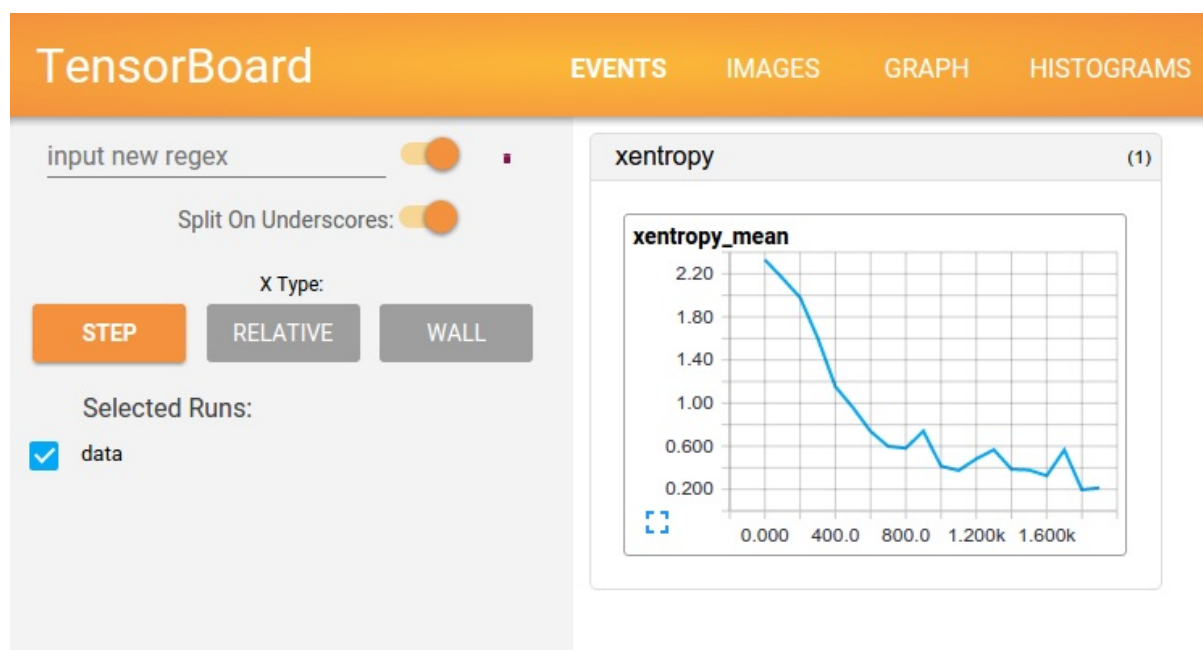
- [TensorBoard:可视化学习](#)
 - [数据序列化](#)
 - [启动TensorBoard](#)

TensorBoard:可视化学习

TensorBoard 涉及到的运算，通常是在训练庞大的深度神经网络中出现的复杂而又难以理解的运算。

为了方便 TensorFlow 程序的理解、调试与优化，我们发布了一套叫做 TensorBoard 的可视化工具。你可以用 TensorBoard 来展现你的 TensorFlow 图像，绘制图像生成的定量指标图以及附加数据。

当 TensorBoard 设置完成后，它应该是这样子的：



数据序列化

TensorBoard 通过读取 TensorFlow 的事件文件来运行。TensorFlow 的事件文件包括了你会在 TensorFlow 运行中涉及到的主要数据。下面是 TensorBoard 中汇总数据 (Summary data) 的大体生命周期。

首先，创建你想汇总数据的 TensorFlow 图，然后再选择你想在哪个节点进行[汇总\(summary\)](#)操

作。

比如，假设你正在训练一个卷积神经网络，用于识别 MNIST 标签。你可能希望记录学习速度 (learning rate) 的如何变化，以及目标函数如何变化。通过向节点附加 `scalar_summary` 操作来分别输出学习速度和期望误差。然后你可以给每个 `scalar_summary` 分配一个有意义的 `标签`，比如 `'learning rate'` 和 `'loss function'`。

或者你还希望显示一个特殊层中激活的分布，或者梯度权重的分布。可以通过分别附加 `histogram_summary` 运算来收集权重变量和梯度输出。

所有可用的 `summary` 操作详细信息，可以查看 [summary_operation](#) 文档。

在TensorFlow中，所有的操作只有当你执行，或者另一个操作依赖于它的输出时才会运行。我们刚才创建的这些节点 (`summary nodes`) 都围绕着你的图像：没有任何操作依赖于它们的结果。因此，为了生成汇总信息，我们需要运行所有这些节点。这样的手动工作是很乏味的，因此可以使用 `tf.merge_all_summaries` 来将他们合并为一个操作。

然后你可以执行合并命令，它会依据特点步骤将所有数据生成一个序列化的 `Summary` `protobuf` 对象。最后，为了将汇总数据写入磁盘，需要将汇总的 `protobuf` 对象传递给 `tf.train.SummaryWriter`。

`SummaryWriter` 的构造函数中包含了参数 `logdir`。这个 `logdir` 非常重要，所有事件都会写到它所指的目录下。此外，`SummaryWriter` 中还包含了一个可选的参数 `GraphDef`。如果输入了该参数，那么 `TensorBoard` 也会显示你的图像。

现在已经修改了你的图，也有了 `SummaryWriter`，现在就可以运行你的神经网络了！如果你愿意的话，你可以每一步执行一次合并汇总，这样你会得到一大堆训练数据。这很有可能超过了你想要的数据量。你也可以每一百步执行一次合并汇总，或者如下面代码里示范的这样。

```
1. merged_summary_op = tf.merge_all_summaries()
2. summary_writer = tf.train.SummaryWriter('/tmp/mnist_logs', sess.graph)
3. total_step = 0
4. while training:
5.     total_step += 1
6.     session.run(training_op)
7.     if total_step % 100 == 0:
8.         summary_str = session.run(merged_summary_op)
9.         summary_writer.add_summary(summary_str, total_step)
```

现在已经准备好用 `TensorBoard` 来可视化这些数据了。

启动TensorBoard

输入下面的指令来启动TensorBoard

```
1. python tensorflow/tensorboard/tensorboard.py --logdir=path/to/log-directory
```

这里的参数 `logdir` 指向 `SummaryWriter` 序列化数据的存储路径。如果 `logdir` 目录的子目录中包含另一次运行时的数据，那么 TensorBoard 会展示所有运行的数据。一旦 TensorBoard 开始运行，你可以通过在浏览器中输入 `localhost:6006` 来查看 TensorBoard。

如果你已经通过pip安装了 TensorBoard，你可以通过执行更为简单地命令来访问 TensorBoard

```
1. tensorboard --logdir=/path/to/log-directory
```

进入 TensorBoard 的界面时，你会在右上角看到导航选项卡，每一个选项卡将展现一组可视化的序列化数据集。对于你查看的每一个选项卡，如果 TensorBoard 中没有数据与这个选项卡相关的话，则会显示一条提示信息指示你如何序列化相关数据。

更多更详细的关于如何使用 `graph` 选项来显示你的图像的信息。参见 [TensorBoard:图表可视化](#)

原文地址: [TensorBoard:Visualizing Learning](#) 翻译: [thylaco1eo](#) 校对: [lucky521](#)

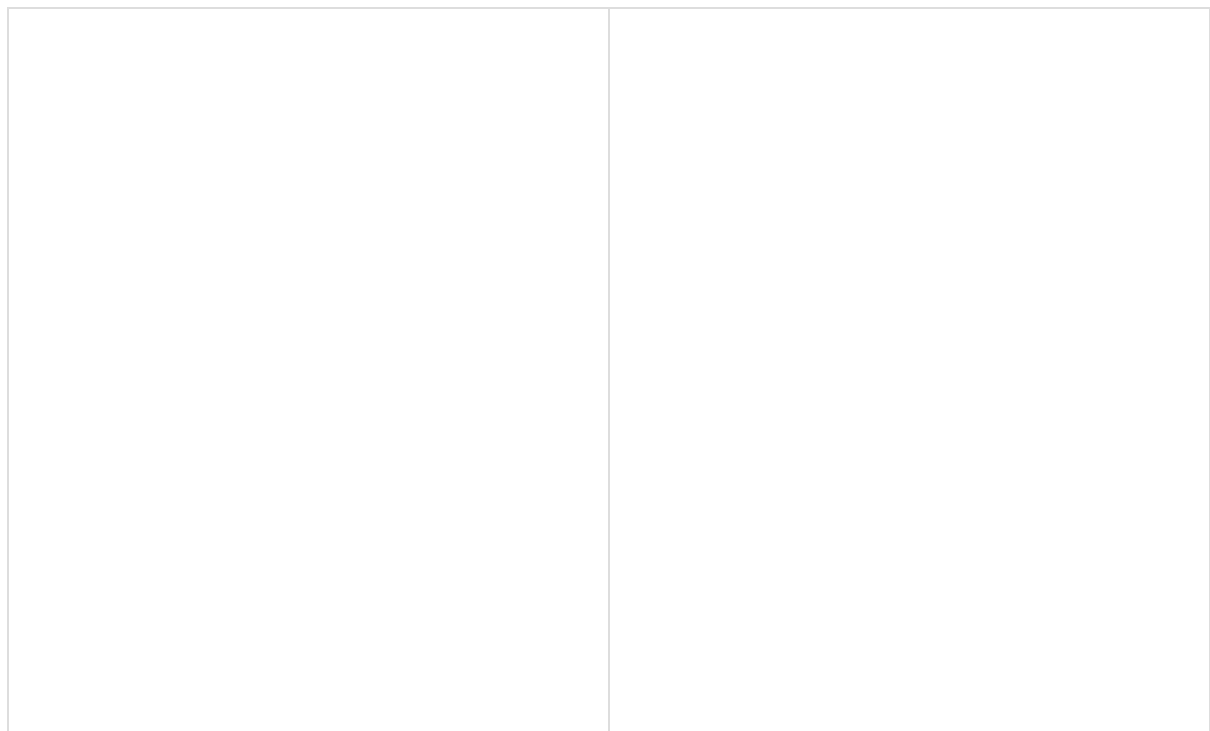

```
1. import tensorflow as tf
2.
3. with tf.name_scope('hidden') as scope:
4.     a = tf.constant(5, name='alpha')
5.     W = tf.Variable(tf.random_uniform([1, 2], -1.0, 1.0), name='weights')
6.     b = tf.Variable(tf.zeros([1]), name='biases')
```

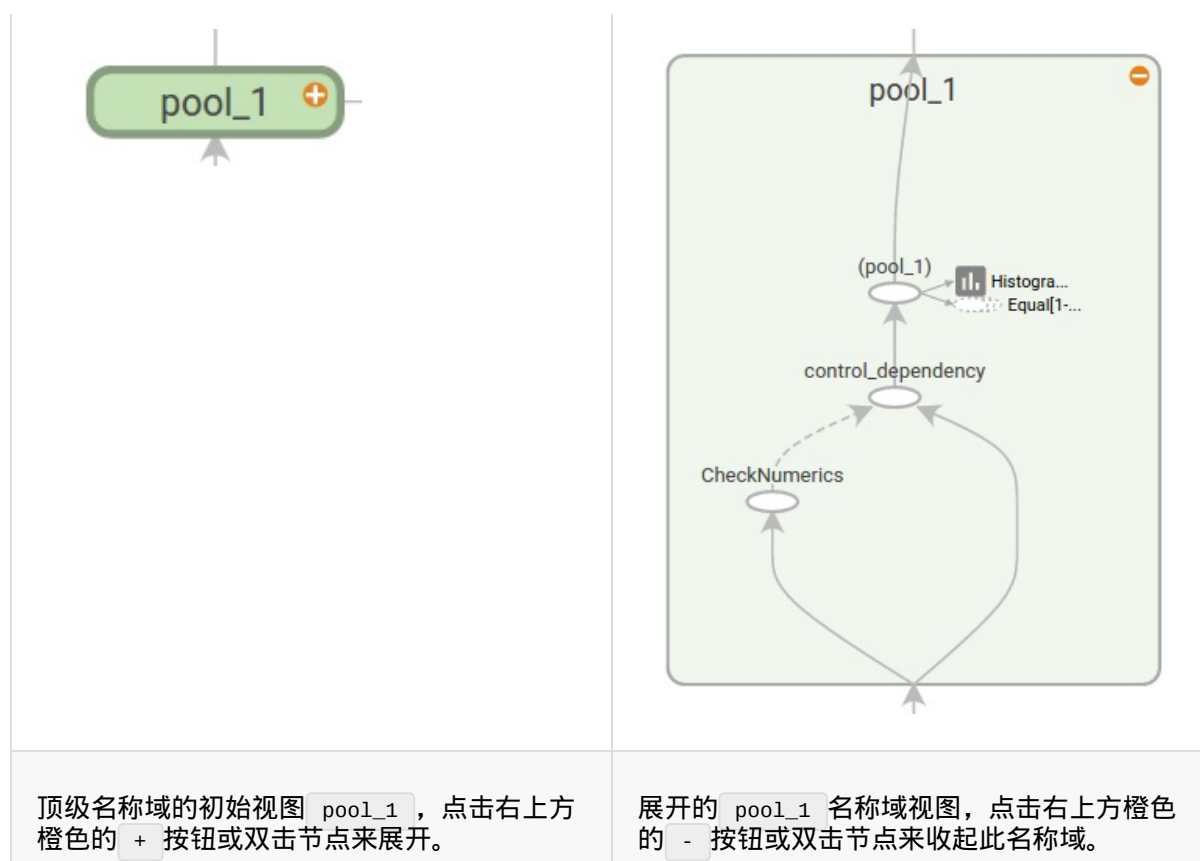
结果是得到了下面三个操作名：

- *hidden/alpha*
- *hidden/weights*
- *hidden/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。

这有一个生动的例子，例中有一个更复杂的节点，节点处于其初始和展开状态。



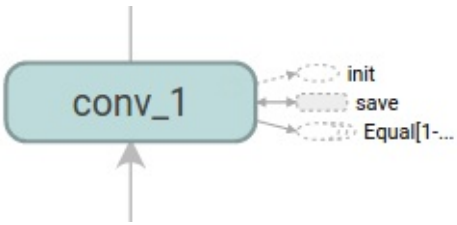
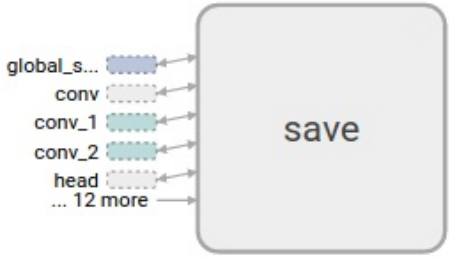


通过名称域把节点分组来得到可读性高的图表很关键的。如果你在构建一个模型，名称域就可以用来控制可视化结果。你的名称域越好，可视性就越好。

上面的图像例子说明了可视化的另一方面，TensorFlow 图表有两种连接关系：数据依赖和控制依赖。数据依赖显示两个操作之间的tensor流程，用实心箭头指示，而控制依赖用点线表示。在已展开的视图(上面的右图)中，除了用点线连接的 `CheckNumerics` 和 `control_dependency` 之外，所有连接都是数据依赖的。

还有一种手段用来简化布局。大多数 TensorFlow 图表有一部分节点，这部分节点和其他节点之间有很多连接。比如，许多节点在初始化阶段可能会有一个控制依赖，而绘制所有 `init` 节点的边缘和其依赖可能会创建一个混乱的视图。

为了减少混乱，可视化把所有 high-degree 节点分离到右边的一个从属区域，而不会绘制线条来表示他们的边缘。线条也不用来表示连接了，我们绘制了小节点图标来指示这些连接关系。分离出从属节点通常不会把关键信息删除掉，因为这些节点和内构功能是相关的。

	
节点 <code>conv_1</code> 被连接到 <code>save</code>，注意其右边 <code>save</code> 节点图标。	<code>save</code> has a high degree, 并会作为从属节点出现, 与 <code>conv_1</code> 的连接作为一个节点图标显示在其左边。为了继续减少杂乱, 既然 <code>save</code> 有很多连接, 我们则只显示前5个, 而把其余的缩略为 <code>... 12 more</code>。

最后一个结构上的简化法叫做序列折叠 (*series collapsing*)。序列基序 (Sequential motifs) 是拥有相同结构并且其名称结尾的数字不同的节点, 它们被折叠进一个单独的节点块 (stack) 中。对长序列网络来说, 序列折叠极大地简化了视图, 对于已层叠的节点, 双击会展开序列。

	
一个节点序列的折叠视图。	视图的一小块, 双击后展开。

最后, 针对易读性的最后一点要说到的是, 可视化为常节点和摘要节点使用了特别的图标, 总结起来有下面这些节点符号:

符号	意义
	<i>High-level</i> 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。

交互

通过平移和缩放来导航图表，点击和拖动用于平移，滚动手势用于缩放。双击一个节点或点击其  按钮来展开代表一组操作的名称域。右下角有一个小地图可以在缩放和平移时方便的改变当前视角。

要关闭一个打开的节点，再次双击它或点击它的  按钮，你也可以只点击一次来选中一个节点，节点的颜色会加深，并且会看到节点的详情，其连接到的节点会在可视化右上角的详情卡片显现。

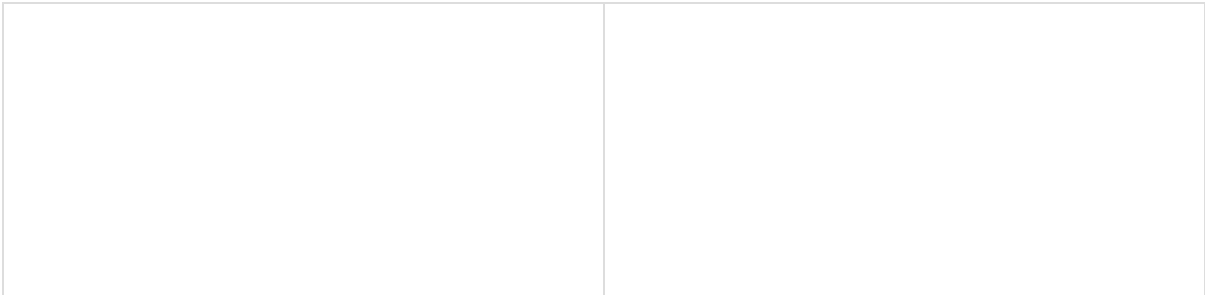
详情卡片展示 <code>conv2</code> 名称域的详细信息，名称域中操作节点的输入和输出被结合在一起，适用于不显示属性的名称域。	详情卡片展示 <code>DecodeRaw</code> 操作节点，除了输入和输出，卡片也会展示与当前节点相关的设备和属性。

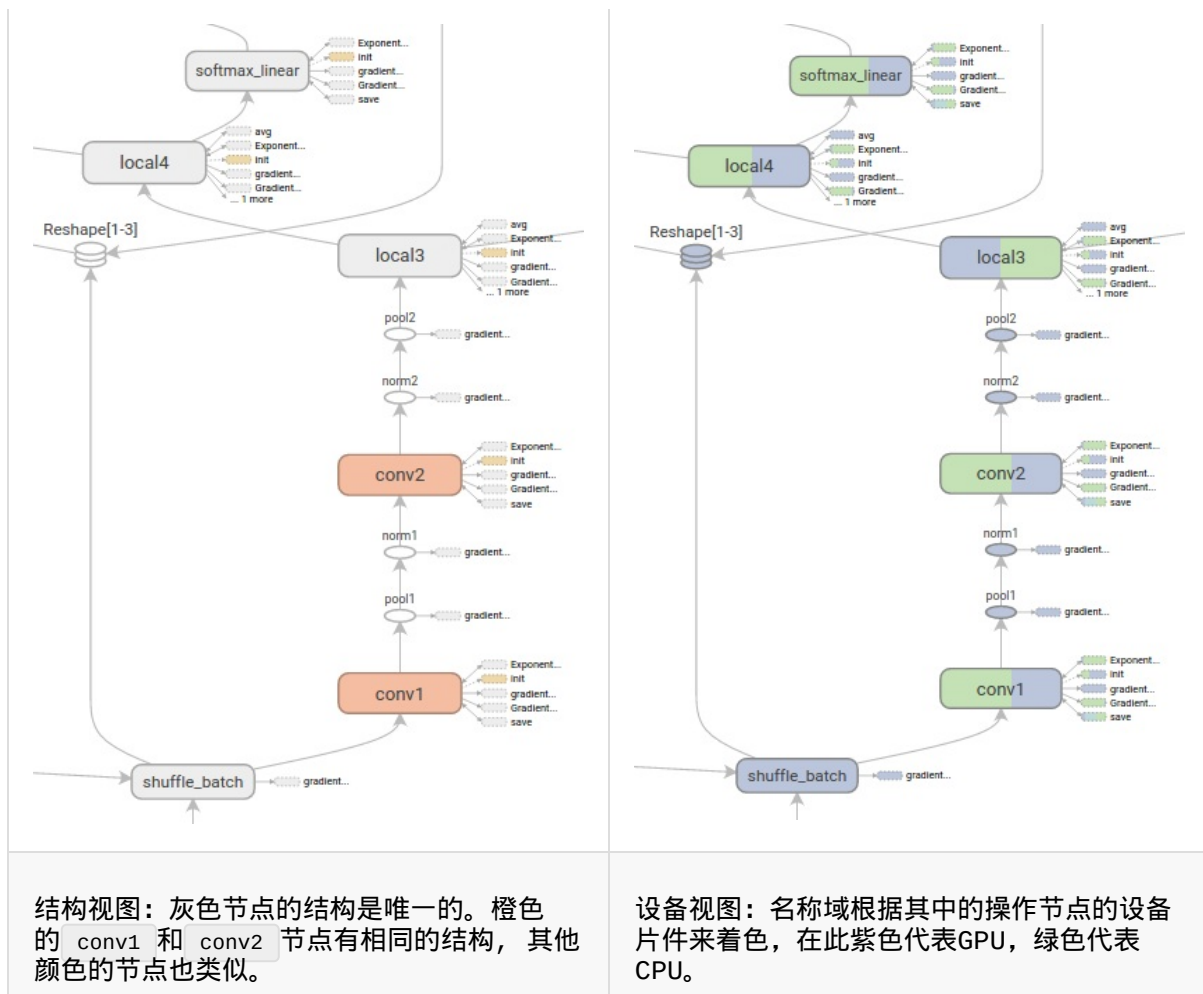
选择对于 `high-degree` 节点的理解也很有帮助，选择任意节点，则与它的其余连接相应的节点也会选中，这使得在进行例如查看哪一个节点是否已保存等操作时非常容易。

点击详情卡片中的一个节点名称时会选中该节点，必要的话，视角会自动平移以使该节点可见。

最后，使用图例上方的颜色菜单，你可以给你的图表选择两个颜色方案。默认的结构视图下，当两个 `high-level` 节点颜色一样时，其会以相同的彩虹色彩出现，而结构唯一的节点颜色是灰色。还有一个视图则展示了不同的操作运行于什么设备之上。名称域被恰当的根据其中的操作节点的设备片来着色。

下图是一张真实图表的图解：





原文： [TensorBoard: Graph Visualization](#)

翻译：@War1n 校对：lucky521

读取数据

- 数据读取
 - 目录
 - 数据读取
 - 供给数据
 - 从文件读取数据
 - 文件名, 乱序(shuffling), 和最大训练迭代数(epoch limits)
 - 文件格式
 - CSV 文件
 - 固定长度的记录
 - 标准TensorFlow格式
 - 预处理
 - 批处理
 - 创建线程并使用 `QueueRunner` 对象来预取
 - 疑问: 这是怎么回事?
 - 疑问: 在达到最大训练迭代数的时候如何清理关闭线程?
 - 筛选记录或产生每个记录的多个样本
 - 稀疏输入数据
 - 预取数据
 - 多输入管道

数据读取

TensorFlow程序读取数据一共有3种方法:

- 供给数据(Feeding): 在TensorFlow程序运行的每一步, 让Python代码来供给数据。
- 从文件读取数据: 在TensorFlow图的起始, 让一个输入管线从文件中读取数据。
- 预加载数据: 在TensorFlow图中定义常量或变量来保存所有数据(仅适用于数据量比较小的情况)。

目录

数据读取

- 供给数据(Feeding)
- 从文件读取数据
 - 文件名, 乱序(shuffling), 和最大训练迭代数(epoch limits)
 - 文件格式

- 预处理
- 批处理
- 使用 `QueueRunner` 创建预读线程
- 对记录进行过滤或者为每个纪录创建多个样本
- 序列化输入数据(Sparse input data)
- 预加载数据
- 多管线输入

供给数据

TensorFlow的数据供给机制允许你在TensorFlow运算图中将数据注入到任一张量中。因此，python运算可以把数据直接设置到TensorFlow图中。

通过给`run()`或者`eval()`函数输入 `feed_dict` 参数， 可以启动运算过程。

```
1. with tf.Session():
2.     input = tf.placeholder(tf.float32)
3.     classifier = ...
4.     print classifier.eval(feed_dict={input: my_python_preprocessing_fn()})
```

虽然你可以使用常量和变量来替换任何一个张量， 但是最好的做法应该是使用 `placeholder` `op`节点。设计 `placeholder` 节点的唯一的意图就是为了提供数据供给(feeding)的方法。`placeholder` 节点被声明的时候是未初始化的， 也不包含数据， 如果没有为它供给数据， 则TensorFlow运算的时候会产生错误， 所以千万不要忘了为 `placeholder` 提供数据。

可以在 `tensorflow/g3doc/tutorials/mnist/fully_connected_feed.py` 找到使用 `placeholder` 和MNIST训练的例子，[MNIST tutorial](#)也讲述了这一例子。

从文件读取数据

一共典型的文件读取管线会包含下面这些步骤：

1. 文件名列表
2. 可配置的 文件名乱序(shuffling)
3. 可配置的 最大训练迭代数(epoch limit)
4. 文件名队列
5. 针对输入文件格式的阅读器
6. 纪录解析器
7. 可配置的预处理器
8. 样本队列

文件名, 乱序(shuffling), 和最大训练迭代数(epoch limits)

可以使用字符串张量(比如 `["file0", "file1"]` , `[("file%d" % i) for i in range(2)]` , `[("file%d" % i) for i in range(2)]`) 或者 `tf.train.match_filenames_once` 函数来产生文件名列表。

将文件名列表交给 `tf.train.string_input_producer` 函数。 `string_input_producer` 来生成一个先入先出的队列, 文件阅读器会需要它来读取数据。

`string_input_producer` 提供的可配置参数来设置文件名乱序和最大的训练迭代数, `QueueRunner` 会为每次迭代(epoch)将所有的文件名加入文件名队列中, 如果 `shuffle=True` 的话, 会对文件名进行乱序处理。这一过程是比较均匀的, 因此它可以产生均衡的文件名列表。

这个 `QueueRunner` 的工作线程是独立于文件阅读器的线程, 因此乱序和将文件名推入到文件名队列这些过程不会阻塞文件阅读器运行。

文件格式

根据你的文件格式, 选择对应的文件阅读器, 然后将文件名队列提供给阅读器的 `read` 方法。阅读器的 `read` 方法会输出一个key来表征输入的文件和其中的纪录(对于调试非常有用), 同时得到一个字符串标量, 这个字符串标量可以被一个或多个解析器, 或者转换操作将其解码为张量并且构造成为样本。

CSV 文件

从CSV文件中读取数据, 需要使用 `TextLineReader` 和 `decode_csv` 操作, 如下面的例子所示:

```
1. filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])
2.
3. reader = tf.TextLineReader()
4. key, value = reader.read(filename_queue)
5.
6. # Default values, in case of empty columns. Also specifies the type of the
7. # decoded result.
8. record_defaults = [[1], [1], [1], [1], [1]]
9. col1, col2, col3, col4, col5 = tf.decode_csv(
10.     value, record_defaults=record_defaults)
11. features = tf.concat(0, [col1, col2, col3, col4])
12.
13. with tf.Session() as sess:
14.     # Start populating the filename queue.
15.     coord = tf.train.Coordinator()
16.     threads = tf.train.start_queue_runners(coord=coord)
```

```

17.
18.     for i in range(1200):
19.         # Retrieve a single instance:
20.         example, label = sess.run([features, col5])
21.
22.     coord.request_stop()
23.     coord.join(threads)

```

每次 `read` 的执行都会从文件中读取一行内容，`decode_csv` 操作会解析这一行内容并将其转为张量列表。如果输入的参数有缺失，`record_default` 参数可以根据张量的类型来设置默认值。

在调用 `run` 或者 `eval` 去执行 `read` 之前，你必须调用 `tf.train.start_queue_runners` 来将文件名填充到队列。否则 `read` 操作会被阻塞到文件名队列中有值为止。

固定长度的记录

从二进制文件中读取固定长度纪录，可以使用 `tf.FixedLengthRecordReader` 的 `tf.decode_raw` 操作。`decode_raw` 操作可以讲一个字符串转换为一个uint8的张量。

举例来说，[the CIFAR-10 dataset](#)的文件格式定义是：每条记录的长度都是固定的，一个字节的标签，后面是3072字节的图像数据。uint8的张量的标准操作就可以从中获取图像片并且根据需要进行重组。例子代码可以在 [tensorflow/models/image/cifar10/cifar10_input.py](#) 找到，具体讲述可参见[教程](#)。

标准TensorFlow格式

另一种保存记录的方法可以允许你讲任意的数据转换为TensorFlow所支持的格式，这种方法可以使TensorFlow的数据集更容易与网络应用架构相匹配。这种建议的方法就是使用TFRecords文件，TFRecords文件包含了 `tf.train.Example` 协议内存块(protocol buffer) (协议内存块包含了字段

`Features`)。你可以写一段代码获取你的数据，将数据填入到 `Example` 协议内存块(protocol buffer)，将协议内存块序列化为一个字符串，并且通过 `tf.python_io.TFRecordWriter` class写入到TFRecords文件。[tensorflow/g3doc/how_tos/reading_data/convert_to_records.py](#) 就是这样的一个例子。

从TFRecords文件中读取数据，可以使用 `tf.TFRecordReader` 的 `tf.parse_single_example` 解析器。这个 `parse_single_example` 操作可以将 `Example` 协议内存块(protocol buffer)解析为张量。MNIST的例子就使用了 `convert_to_records` 所构建的数据。请参看 [tensorflow/g3doc/how_tos/reading_data/fully_connected_reader.py](#)，您也可以将这个例子跟 `fully_connected_feed` 的版本加以比较。

预处理

你可以对输入的样本进行任意的预处理，这些预处理不依赖于训练参数，你可以

在 `tensorflow/models/image/cifar10/cifar10.py` 找到数据归一化， 提取随机数据片， 增加噪声或失真等等预处理的例子。

批处理

在数据输入管线的末端， 我们需要有另一个队列来执行输入样本的训练， 评价和推理。因此我们使用 `tf.train.shuffle_batch` 函数来对队列中的样本进行乱序处理

示例：

```
1. def read_my_file_format(filename_queue):
2.     reader = tf.SomeReader()
3.     key, record_string = reader.read(filename_queue)
4.     example, label = tf.some_decoder(record_string)
5.     processed_example = some_processing(example)
6.     return processed_example, label
7.
8. def input_pipeline(filenamees, batch_size, num_epochs=None):
9.     filename_queue = tf.train.string_input_producer(
10.         filenamees, num_epochs=num_epochs, shuffle=True)
11.     example, label = read_my_file_format(filename_queue)
12.     # min_after_dequeue defines how big a buffer we will randomly sample
13.     # from -- bigger means better shuffling but slower start up and more
14.     # memory used.
15.     # capacity must be larger than min_after_dequeue and the amount larger
16.     # determines the maximum we will prefetch. Recommendation:
17.     # min_after_dequeue + (num_threads + a small safety margin) * batch_size
18.     min_after_dequeue = 10000
19.     capacity = min_after_dequeue + 3 * batch_size
20.     example_batch, label_batch = tf.train.shuffle_batch(
21.         [example, label], batch_size=batch_size, capacity=capacity,
22.         min_after_dequeue=min_after_dequeue)
23.     return example_batch, label_batch
```

如果你需要对不同文件中的样子有更强的乱序和并行处理，可以使用 `tf.train.shuffle_batch_join` 函数。

示例：

```
1. def read_my_file_format(filename_queue):
2.     # Same as above
3.
4. def input_pipeline(filenamees, batch_size, read_threads, num_epochs=None):
5.     filename_queue = tf.train.string_input_producer(
6.         filenamees, num_epochs=num_epochs, shuffle=True)
7.     example_list = [read_my_file_format(filename_queue)
8.                     for _ in range(read_threads)]
```

```

9.     min_after_dequeue = 10000
10.    capacity = min_after_dequeue + 3 * batch_size
11.    example_batch, label_batch = tf.train.shuffle_batch_join(
12.        example_list, batch_size=batch_size, capacity=capacity,
13.        min_after_dequeue=min_after_dequeue)
14.    return example_batch, label_batch

```

在这个例子中，你虽然只使用了一个文件名队列，但是TensorFlow依然能保证多个文件阅读器从同一次迭代(epoch)的不同文件中读取数据，知道这次迭代的所有文件都被开始读取为止。（通常来说一个线程来对文件名队列进行填充的效率是足够的）

另一种替代方案是：使用 `tf.train.shuffle_batch` 函数, 设置 `num_threads` 的值大于1。这种方案可以保证同一时刻只在一个文件中进行读取操作(但是读取速度依然优于单线程)，而不是之前的同时读取多个文件。这种方案的优点是：

- 避免了两个不同的线程从同一个文件中读取同一个样本。
- 避免了过多的磁盘搜索操作。

你一共需要多少个读取线程呢？函数 `tf.train.shuffle_batch*` 为TensorFlow图提供了获取文件名队列中的元素个数之和的方法。如果你有足够多的读取线程，文件名队列中的元素个数之和应该一直是一个略高于0的数。具体可以参考[TensorBoard:可视化学习](#)。

创建线程并使用 `QueueRunner` 对象来预取

简单来说：使用上面列出的许多 `tf.train` 函数添加 `QueueRunner` 到你的数据流图中。在你运行任何训练步骤之前，需要调用 `tf.train.start_queue_runners` 函数，否则数据流图将一直挂起。`tf.train.start_queue_runners` 这个函数将会启动输入管道的线程，填充样本到队列中，以便出队操作可以从队列中拿到样本。这种情况下最好配合使用一个 `tf.train.Coordinator`，这样可以在发生错误的情况下正确地关闭这些线程。如果你对训练迭代数做了限制，那么需要使用一个训练迭代数计数器，并且需要被初始化。推荐的代码模板如下：

```

1. # Create the graph, etc.
2. init_op = tf.initialize_all_variables()
3.
4. # Create a session for running operations in the Graph.
5. sess = tf.Session()
6.
7. # Initialize the variables (like the epoch counter).
8. sess.run(init_op)
9.
10. # Start input enqueue threads.
11. coord = tf.train.Coordinator()
12. threads = tf.train.start_queue_runners(sess=sess, coord=coord)
13.
14. try:

```

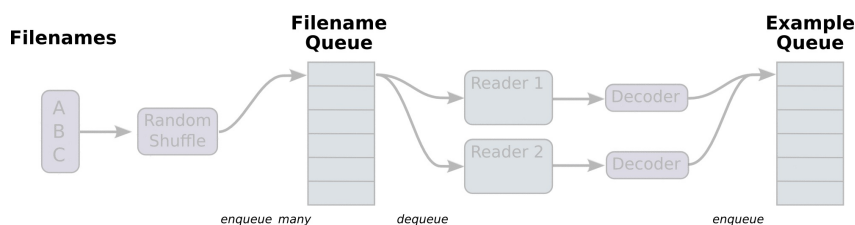
```

15.     while not coord.should_stop():
16.         # Run training steps or whatever
17.         sess.run(train_op)
18.
19. except tf.errors.OutOfRangeError:
20.     print 'Done training -- epoch limit reached'
21. finally:
22.     # When done, ask the threads to stop.
23.     coord.request_stop()
24.
25. # Wait for threads to finish.
26. coord.join(threads)
27. sess.close()

```

疑问：这是怎么回事？

首先，我们先创建数据流图，这个数据流图由一些流水线的阶段组成，阶段间用队列连接在一起。第一阶段将生成文件名，我们读取这些文件名并且把他们排到文件名队列中。第二阶段从文件中读取数据（使用 `Reader` ），产生样本，而且把样本放在一个样本队列中。根据你的设置，实际上也可以拷贝第二阶段的样本，使得他们相互独立，这样就可以从多个文件中并行读取。在第二阶段的最后是一个排队操作，就是入队到队列中去，在下一阶段出队。因为我们要开始运行这些入队操作的线程，所以我们的训练循环会使得样本队列中的样本不断地出队。



在 `tf.train` 中要创建这些队列和执行入队操作，就要添加 `tf.train.QueueRunner` 到一个使用 `tf.train.add_queue_runner` 函数的数据流图中。每个 `QueueRunner` 负责一个阶段，处理那些需要在线程中运行的入队操作的列表。一旦数据流图构造成功，`tf.train.start_queue_runners` 函数就会要求数据流图中每个 `QueueRunner` 去开始它的线程运行入队操作。

如果一切顺利的话，你现在可以执行你的训练步骤，同时队列也会被后台线程来填充。如果您设置了最大训练迭代数，在某些时候，样本出队的操作可能会得到一个 `tf.OutOfRangeError` 的错误。这其实是TensorFlow的“文件结束”（EOF）—— 这就意味着已经达到了最大训练迭代数，已经没有更多可用的样本了。

最后一个因素是 `Coordinator`。这是负责在收到任何关闭信号的时候，让所有的线程都知道。最常用的是在发生异常时这种情况就会呈现出来，比如说其中一个线程在运行某些操作时出现错误（或一个普通的Python异常）。

想要了解更多的关于threading, queues, QueueRunners, and Coordinators的内容可以[看这里](#).

疑问：在达到最大训练迭代数的时候如何清理关闭线程？

想象一下，你有一个模型并且设置了最大训练迭代数。这意味着，生成文件的那个线程将只会在产生 `OutOfRange` 错误之前运行许多次。该 `QueueRunner` 会捕获该错误，并且关闭文件名的队列，最后退出线程。关闭队列做了两件事情：

- 如果还试着对文件名队列执行入队操作时将发生错误。任何线程不应该尝试去这样做，但是当队列因为其他错误而关闭时，这就会有用了。
- 任何当前或将来出队操作要么成功（如果队列中还有足够的元素）或立即失败（发生 `OutOfRange` 错误）。它们不会防止等待更多的元素被添加到队列中，因为上面的一点已经保证了这种情况不会发生。

关键是，当在文件名队列被关闭时候，有可能还有许多文件名在该队列中，这样下一阶段的流水线（包括reader和其它预处理）还可以继续运行一段时间。一旦文件名队列空了之后，如果后面的流水线还要尝试从文件名队列中取出一个文件名（例如，从一个已经处理完文件的reader中），这将会触发 `OutOfRange` 错误。在这种情况下，即使你可能有一个QueueRunner关联着多个线程。如果这不是在QueueRunner中的最后那个线程，`OutOfRange` 错误仅仅只会使得一个线程退出。这使得其他那些正处理自己的最后一个文件的线程继续运行，直至他们完成为止。（但如果假设你使用的是 `tf.train.Coordinator`，其他类型的错误将导致所有线程停止）。一旦所有的reader线程触发 `OutOfRange` 错误，然后才是下一个队列，再是样本队列被关闭。

同样，样本队列中会有一些已经入队的元素，所以样本训练将一直持续直到样本队列中再没有样本为止。如果样本队列是一个 `RandomShuffleQueue`，因为你使用了 `shuffle_batch` 或者 `shuffle_batch_join`，所以通常不会出现以往那种队列中的元素会比 `min_after_dequeue` 定义的更少的情况。然而，一旦该队列被关闭，`min_after_dequeue` 设置的限定值将失效，最终队列将为空。在这一点来说，当实际训练线程尝试从样本队列中取出数据时，将会触发 `OutOfRange` 错误，然后训练线程会退出。一旦所有的培训线程完成，`tf.train.Coordinator.join` 会返回，你就可以正常退出了。

筛选记录或产生每个记录的多个样本

举个例子，有形式为 `[x, y, z]` 的样本，我们可以生成一批形式为 `[batch, x, y, z]` 的样本。如果你想滤除这个记录（或许不需要这样的设置），那么可以设置batch的大小为0；但如果你需要每个记录产生多个样本，那么batch的值可以大于1。然后很简单，只需调用批处理函数（比如：

`shuffle_batch` OR `shuffle_batch_join`）去设置 `enqueue_many=True` 就可以实现。

稀疏输入数据

SparseTensors这种数据类型使用队列来处理不是太好。如果要使用SparseTensors你就必须在批处理之后使用 `tf.parse_example` 去解析字符串记录（而不是在批处理之前使用

```
tf.parse_single_example ) 。
```

预取数据

这仅用于可以完全加载到存储器中的小的数据集。有两种方法：

- 存储在常数中。
- 存储在变量中，初始化后，永远不要改变它的值。

使用常数更简单一些，但是会使用更多的内存（因为常数会内联的存储在数据流图数据结构中，这个结构体可能会被复制几次）。

```
1. training_data = ...
2. training_labels = ...
3. with tf.Session():
4.     input_data = tf.constant(training_data)
5.     input_labels = tf.constant(training_labels)
6.     ...
```

要改为使用变量的方式，您就需要在数据流图建立后初始化这个变量。

```
1. training_data = ...
2. training_labels = ...
3. with tf.Session() as sess:
4.     data_initializer = tf.placeholder(dtype=training_data.dtype,
5.                                     shape=training_data.shape)
6.     label_initializer = tf.placeholder(dtype=training_labels.dtype,
7.                                     shape=training_labels.shape)
8.     input_data = tf.Variable(data_initializer, trainable=False, collections=[])
9.     input_labels = tf.Variable(label_initializer, trainable=False, collections=[])
10.    ...
11.    sess.run(input_data.initializer,
12.             feed_dict={data_initializer: training_data})
13.    sess.run(input_labels.initializer,
14.             feed_dict={label_initializer: training_labels})
```

设定 `trainable=False` 可以防止该变量被数据流图的 `GraphKeys.TRAINABLE_VARIABLES` 收集，这样我们就不会在训练的时候尝试更新它的值； 设定 `collections=[]` 可以防止 `GraphKeys.VARIABLES` 收集后做为保存和恢复的中断点。

无论哪种方式， `tf.train.slice_input_producer function` 函数可以被用来每次产生一个切片。这样就会让样本在整个迭代中被打乱，所以在使用批处理的时候不需要再次打乱样本。所以我们不使用 `shuffle_batch` 函数，取而代之的是纯 `tf.train.batch` 函数。 如果要使用多个线程进行预处理，需要将 `num_threads` 参数设置为大于1的数字。

在 `tensorflow/g3doc/how_tos/reading_data/fully_connected_preloaded.py` 中可以找到一个MNIST例子，使用常数来预加载。 另外使用变量来预加载的例子

在 `tensorflow/g3doc/how_tos/reading_data/fully_connected_preloaded_var.py`，你可以用上面 `fully_connected_feed` 和 `fully_connected_reader` 的描述来进行比较。

多输入管道

通常你会在一个数据集上面训练，然后在另外一个数据集上做评估计算(或称为 “eval”)。 这样做的一种方法是，实际上包含两个独立的进程：

- 训练过程中读取输入数据，并定期将所有的训练的变量写入还原点文件)。
- 在计算过程中恢复还原点文件到一个推理模型中，读取有效的输入数据。

这两个进程在下面的例子中已经完成了：[the example CIFAR-10 model](#)，有以下几个好处：

- eval被当做训练后变量的一个简单映射。
- 你甚至可以在训练完成和退出后执行eval。

您可以在同一个进程的相同的数据流图中有训练和eval，并分享他们的训练后的变量。参考[the shared variables tutorial](#)。

原文地址：[Reading data](#) 翻译：[volvet](#) and [zhangkom](#) 校对：

线程和队列

- [线程和队列](#)
 - [队列使用概述](#)
 - [Coordinator](#)
 - [QueueRunner](#)
 - [异常处理](#)

线程和队列

在使用TensorFlow进行异步计算时，队列是一种强大的机制。

正如TensorFlow中的其他组件一样，队列就是TensorFlow图中的节点。这是一种有状态的节点，就像变量一样：其他节点可以修改它的内容。具体来说，其他节点可以把新元素插入到队列后端(rear)，也可以把队列前端(front)的元素删除。

为了感受一下队列，让我们来看一个简单的例子。我们先创建一个“先入先出”的队列(FIFOQueue)，并将其内部所有元素初始化为零。然后，我们构建一个TensorFlow图，它从队列前端取走一个元素，加上1之后，放回队列的后端。慢慢地，队列的元素的值就会增加。

Client

```
q = tf.FIFOQueue(3, "float")
init = q.enqueue_many([[0.,0.,0.]])

x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()
```

`Enqueue`、`EnqueueMany` 和 `Dequeue` 都是特殊的节点。他们需要获取队列指针，而非普通的值，如此才能修改队列内容。我们建议您将它们看作队列的方法。事实上，在Python API中，它们就是队列对象的方法（例如 `q.enqueue(...)`）。

现在你已经对队列有了一定的了解，让我们深入到细节...

队列使用概述

队列，如 `FIFOQueue` 和 `RandomShuffleQueue`，在TensorFlow的张量异步计算时都非常重要。

例如，一个典型的输入结构：是使用一个 `RandomShuffleQueue` 来作为模型训练的输入：

- 多个线程准备训练样本，并且把这些样本推入队列。
- 一个训练线程执行一个训练操作，此操作会从队列中移除最小批次的样本（mini-batches）。

这种结构具有许多优点，正如在[Reading data how to](#)中强调的，同时，[Reading data how to](#)也概括地描述了如何简化输入管道的构造过程。

TensorFlow的 `Session` 对象是可以支持多线程的，因此多个线程可以很方便地使用同一个会话（Session）并且并行地执行操作。然而，在Python程序实现这样的并行运算却并不容易。所有线程都必须能被同步终止，异常必须能被正确捕获并报告，会话终止的时候，队列必须能被正确地关闭。

所幸TensorFlow提供了两个类来帮助多线程的实现：`tf.Coordinator`和 `tf.QueueRunner`。从设计上这两个类必须被一起使用。`Coordinator` 类可以用来同时停止多个工作线程并且向那个在等待所有工作线程终止的程序报告异常。`QueueRunner` 类用来协调多个工作线程同时将多个张量推入同一个队列中。

Coordinator

`Coordinator`类用来帮助多个线程协同工作，多个线程同步终止。

其主要方法有：

- `should_stop()`：如果线程应该停止则返回True。
- `request_stop(<exception>)`：请求该线程停止。
- `join(<list of threads>)`：等待被指定的线程终止。

首先创建一个 `Coordinator` 对象，然后建立一些使用 `Coordinator` 对象的线程。这些线程通常一直循环运行，一直到 `should_stop()` 返回True时停止。

任何线程都可以决定计算什么时候应该停止。它只需要调用 `request_stop()`，同时其他线程的 `should_stop()` 将会返回 `True`，然后都停下来。

```
1. # 线程体：循环执行，直到`Coordinator`收到了停止请求。
2. # 如果某些条件为真，请求`Coordinator`去停止其他线程。
3. def MyLoop(coord):
4.     while not coord.should_stop():
5.         ...do something...
6.         if ...some condition...:
7.             coord.request_stop()
8.
9. # Main code: create a coordinator.
10. coord = Coordinator()
11.
12. # Create 10 threads that run 'MyLoop()'
```

```

13. threads = [threading.Thread(target=MyLoop, args=(coord)) for i in xrange(10)]
14.
15. # Start the threads and wait for all of them to stop.
16. for t in threads: t.start()
17. coord.join(threads)

```

显然，Coordinator可以管理线程去做不同的事情。上面的代码只是一个简单的例子，在设计实现的时候不必完全照搬。Coordinator还支持捕捉和报告异常，具体可以参考[Coordinator class](#)的文档。

QueueRunner

`QueueRunner` 类会创建一组线程，这些线程可以重复的执行Enqueue操作，他们使用同一个Coordinator来处理线程同步终止。此外，一个QueueRunner会运行一个*closer thread*，当Coordinator收到异常报告时，这个*closer thread*会自动关闭队列。

您可以使用一个queue runner，来实现上述结构。

首先建立一个TensorFlow图表，这个图表使用队列来输入样本。增加处理样本并将样本推入队列中的操作。增加training操作来移除队列中的样本。

```

1. example = ...ops to create one example...
2. # Create a queue, and an op that enqueues examples one at a time in the queue.
3. queue = tf.RandomShuffleQueue(...)
4. enqueue_op = queue.enqueue(example)
5. # Create a training graph that starts by dequeuing a batch of examples.
6. inputs = queue.dequeue_many(batch_size)
7. train_op = ...use 'inputs' to build the training part of the graph...

```

在Python的训练程序中，创建一个 `QueueRunner` 来运行几个线程，这几个线程处理样本，并且将样本推入队列。创建一个 `Coordinator`，让queue runner使用 `Coordinator` 来启动这些线程，创建一个训练的循环，并且使用 `Coordinator` 来控制 `QueueRunner` 的线程们的终止。

```

1. # Create a queue runner that will run 4 threads in parallel to enqueue
2. # examples.
3. qr = tf.train.QueueRunner(queue, [enqueue_op] * 4)
4.
5. # Launch the graph.
6. sess = tf.Session()
7. # Create a coordinator, launch the queue runner threads.
8. coord = tf.train.Coordinator()
9. enqueue_threads = qr.create_threads(sess, coord=coord, start=True)
10. # Run the training loop, controlling termination with the coordinator.
11. for step in xrange(1000000):
12.     if coord.should_stop():

```

```
13.         break
14.         sess.run(train_op)
15.     # When done, ask the threads to stop.
16.     coord.request_stop()
17.     # And wait for them to actually do it.
18.     coord.join(threads)
```

异常处理

通过queue runners启动的线程不仅仅只处理推送样本到队列。他们还捕捉和处理由队列产生的异常，包括 `OutOfRangeError` 异常，这个异常是用于报告队列被关闭。

使用 `Coordinator` 的训练程序在主循环中必须同时捕捉和报告异常。

下面是对上面训练循环的改进版本。

```
1. try:
2.     for step in xrange(1000000):
3.         if coord.should_stop():
4.             break
5.         sess.run(train_op)
6. except Exception, e:
7.     # Report exceptions to the coordinator.
8.     coord.request_stop(e)
9.
10. # Terminate as usual. It is innocuous to request stop twice.
11. coord.request_stop()
12. coord.join(threads)
```

原文地址: [Threading and Queues](#) 翻译: zhangkom 校对: volvet

添加新的Op

- 增加一个新 Op
 - 内容
 - 增加一个新 Op
 - 定义 Op 的接口
 - 为 Op 实现 kernel
 - 生成客户端包装器
 - Python Op 包装器
 - C++ Op 包装器
 - 检查 Op 能否正常工作
 - 验证条件
 - Op 注册
 - 属性
 - 属性类型
 - 默认值和约束条件
 - 多态
 - Type Polymorphism
 - 列表输入和输出
 - 输入和输出
 - 向后兼容性
 - GPU 支持
 - 使用 Python 实现梯度
 - 在 Python 中实现一个形状函数

增加一个新 Op

预备知识：

- 对 C++ 有一定了解。
- 已经[下载 TensorFlow 源代码](#)并有能力编译它。

如果现有的库没有涵盖你想要的操作，你可以自己定制一个。为了使定制的 Op 能够兼容原有的库，你必须做以下工作：

- 在一个 C++ 文件中注册新 Op。Op 的注册与实现是相互独立的。在其注册时描述了 Op 该如何执行。
例如，注册 Op 时定义了 Op 的名字，并指定了它的输入和输出。
- 使用 C++ 实现 Op。每一个实现称之为一个 “kernel”，可以存在多个 kernel，以适配不同的架构
(CPU, GPU 等)或不同的输入/输出类型。

- 创建一个 Python 包装器 (wrapper)。这个包装器是创建 Op 的公开 API。当注册 Op 时，会自动生成一个默认
默认的包装器。既可以直接使用默认包装器，也可以添加一个新的包装器。
- (可选) 写一个函数计算 Op 的梯度。
- (可选) 写一个函数，描述 Op 的输入和输出 shape。该函数能够允许从 Op 推断 shape。
- 测试 Op，通常使用 Python。如果你定义了梯度，你可以使用Python的[GradientChecker](#)来测试它。

内容

增加一个新 Op

- 定义 Op 的接口
- 为 Op 实现 kernel
- 生成客户端包装器
 - Python Op 包装器
 - C++ Op 包装器
- 检查 Op 能否正常工作
- 验证条件
- Op 注册
 - 属性
 - 属性类型
 - 多态
 - 输入和输出
 - 向后兼容性
- GPU 支持
- 使用 Python 实现梯度
- 使用 Python 实现 shape 函数

定义 Op 的接口

向 TensorFlow 系统注册来定义 Op 的接口。在注册时，指定 Op 的名称，它的输入(类型和名称)

和输出(类型和名称)，和所需要任何 [属性](#)的文档说明。

为了让你有直观的认识，创建一个简单的 Op 作为例子。该 Op 接受一个 `int32` 类型 tensor 作为

输入，输出这个 tensor 的一个副本，副本与原 tensor 唯一的区别在于第一个元素被置为 0。

创建

文件 `tensorflow/core/user_ops/zero_out.cc`，并调用 `REGISTER_OP` 宏来定义 Op 的接口。

```

1. #include "tensorflow/core/framework/op.h"
2. REGISTER_OP("ZeroOut")
3.   .Input("to_zero: int32")
4.   .Output("zeroed: int32");

```

`ZeroOut` Op 接受 32 位整型的 tensor `to_zero` 作为输入，输出 32 位整型的 tensor `zeroed`。

命名的注意事项：`Op` 的名称必须是为唯一的，并使用驼峰命名法。以下划线 `_` 开始的名称保留为内部使用。

为 Op 实现 kernel

在定义接口之后，提供一个或多个 Op 的实现。为这些 kernel 的每一个创建一个对应的类，继承

`OpKernel`，覆盖 `Compute` 方法。`Compute` 方法提供一个类型为 `OpKernelContext*` 的参数 `context`，用于访问一些有用的信息，例如输入和输出的 tensor。

将 kernel 添加到刚才创建的文件中，kernel 看起来和下面的代码类似：

```

1. #include "tensorflow/core/framework/op_kernel.h"
2. using namespace tensorflow;
3. class ZeroOutOp : public OpKernel {
4. public:
5.   explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
6.   void Compute(OpKernelContext* context) override {
7.     // 获取输入 tensor.
8.     const Tensor& input_tensor = context->input(0);
9.     auto input = input_tensor.flat<int32>();
10.    // 创建一个输出 tensor.
11.    Tensor* output_tensor = NULL;
12.    OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),
13.                                                    &output_tensor));
14.    auto output = output_tensor->template flat<int32>();
15.    // 设置 tensor 除第一个之外的元素均设为 0.
16.    const int N = input.size();
17.    for (int i = 1; i < N; i++) {
18.      output(i) = 0;
19.    }
20.    // 尽可能地保留第一个元素的值.
21.    if (N > 0) output(0) = input(0);
22.  }
23. };

```

实现 kernel 后，将其注册到 TensorFlow 系统中。注册时，可以指定该 kernel 运行时的多

个约束

条件。例如可以指定一个 kernel 在 CPU 上运行，另一个在 GPU 上运行。

将下列代码加入到 `zero_out.cc` 中，注册 `ZeroOut` op:

```
1. REGISTER_KERNEL_BUILDER(Name("ZeroOut").Device(DEVICE_CPU), ZeroOutOp);
```

一旦[创建和重新安装了 TensorFlow](#)，Tensorflow 系统可以在需要时引用和使用该 Op。

生成客户端包装器

Python Op 包装器

当编译 TensorFlow 时，所有放在 `tensorflow/core/user_ops` 目录下的 Op 会自动在 `bazel-genfiles/tensorflow/python/ops/gen_user_ops.py` 文件中生成 Python Op 包装器。通过以下声明，把那些 Op 引入到 `tensorflow/python/user_ops/user_ops.py` 中：

```
1. from tensorflow.python.ops.gen_user_ops import *
```

你可以选择性将部分函数替换为自己的实现。为此，首先要隐藏自动生成的代码，在 `tensorflow/python/BUILD` 文件中，将其名字添加到 `"user_ops"` 的 `hidden` 列表。

```
1. tf_gen_op_wrapper_py(
2.     name = "user_ops",
3.     hidden = [
4.         "Fact",
5.     ],
6.     require_shape_functions = False,
7. )
```

紧接着 `"Fact"` 列出自己的 Op。然后，在 `tensorflow/python/user_ops/user_ops.py` 中添加你的替代实现函数。通常，替代实现函数也会调用自动生成函数来真正把 Op 添加到图中。被隐藏的自动生成函数位于 `gen_user_ops` 包中，名称多了一个下划线前缀 (`"_"`)。例如：

```
1. def my_fact():
2.     """覆盖一个 Op 自动生成代码的示例."""
```

```
3.     return gen_user_ops._fact()
```

C++ Op 包装器

当编译 TensorFlow 时，所有 `tensorflow/core/user_ops` 文件夹下的 Op 会自动创建 C++ Op 包装器。例如，`tensorflow/core/user_ops/zero_out.cc` 中的 Op 会自动在 `bazel-genfiles/tensorflow/cc/ops/user_ops.{h,cc}` 中生成包装器。

`tensorflow/cc/ops/standard_ops.h` 通过下述申明，导入用户自定义 Op 自动生成的包装器。

```
1. #include "tensorflow/cc/ops/user_ops.h"
```

检查 Op 能否正常工作

验证已经成功实现 Op 的方式是编写测试程序。创建文件

```
tensorflow/python/kernel_tests/zero_out_op_test.py
```

包含以下内容：

```
1. import tensorflow as tf
2. class ZeroOutTest(tf.test.TestCase):
3.     def testZeroOut(self):
4.         with self.test_session():
5.             result = tf.user_ops.zero_out([5, 4, 3, 2, 1])
6.             self.assertEqual(result.eval(), [5, 0, 0, 0, 0])
```

然后运行测试：

```
1. $ bazel test tensorflow/python:zero_out_op_test
```

验证条件

上述示例假定 Op 能够应用在任何 shape 的 tensor 上。如果只想应用到 vector 上呢？

这意味需要在上述 OpKernel 实现中添加相关的检查。

```
1. void Compute(OpKernelContext* context) override {
2.     // 获取输入 tensor
3.     const Tensor& input_tensor = context->input(0);
```

```

4.     OP_REQUIRES(context, TensorShapeUtils::IsVector(input_tensor.shape()),
5.                 errors::InvalidArgument("ZeroOut expects a 1-D vector."));
6.     // ...
7. }

```

OP_REQUIRES 断言的输入是一个 vector，如果不是 vector，将设置 `InvalidArgument` 状态并返回。

`OP_REQUIRES` 宏 有三个参数：

- `context`：可以是一个 `OpKernelContext` 或 `OpKernelConstruction` 指针（参见 `tensorflow/core/framework/op_kernel.h`），其 `SetStatus()` 方法将被使用到。
- 检查条件：`tensorflow/core/public/tensor_shape.h` 中有一些验证 tensor shape 的函数。
- 条件不满足时产生的错误：错误用一个 `Status` 对象表示，参见 `tensorflow/core/public/status.h`。
`Status` 包含一个类型（通常是 `InvalidArgument`，但也可以是任何类型）和一个消息。构造一个错误的函数位于 `tensorflow/core/lib/core/errors.h` 中。

如果想要测试一个函数返回的 `Status` 对象是否是一个错误，可以使用 `OP_REQUIRES_OK`。这些宏如果检测到错误，会直接跳出函数，终止函数执行。

Op 注册

属性

Op 可以有属性，属性的值在 Op 添加到图中时被设置。属性值用于配置 Op，在 kernel 实现中，Op 注册的输入和输出类型中，均可访问这些属性值。尽可能地使用输入代替属性，因为输入的灵活性更高，例如可以在执行步骤中被更改，可以使用 feed 等等。属性可用于实现一些输入无法做到的事情，例如影响 Op 签名（即输入输出的数量和类型）的配置或只读配置可以通过属性实现。

注册 Op 时可以用 `Attr` 方法指定属性的名称和类型，以此来定义一个属性，形式如下：

```
1. <name>: <attr-type-expr>
```

`<name>` 必须以字母开头，可以由数字，字母，下划线组成。`<attr-type-expr>` 是一个类型表达式，形式如下：

例如，如果想要 `ZeroOut` Op 保存一个用户索引，指示该 Op 不仅仅只有一个元素，你可以注册

Op 如下:

```
1. REGISTER_OP("ZeroOut")
2.   .Attr("preserve_index: int")
3.   .Input("to_zero: int32")
4.   .Output("zeroed: int32");
```

你的 kernel 可以在构造函数里, 通过 `context` 参数访问这个属性:

```
1. class ZeroOutOp : public OpKernel {
2. public:
3.   explicit ZeroOutOp(OpKernelConstruction * context) : OpKernel(context) {
4.     // 获取欲保存的索引值
5.     OP_REQUIRES_OK(context,
6.                     context->GetAttr("preserve_index", &preserve_index_));
7.     // 检查 preserve_index 是否为正
8.     OP_REQUIRES(context, preserve_index_ >= 0,
9.                 errors::InvalidArgument("Need preserve_index >= 0, got ",
10.                                         preserve_index_));
11.   }
12.   void Compute(OpKernelContext* context) override {
13.     // ...
14.   }
15. private:
16.   int preserve_index_;
17. };
```

该值可以在 `Compute` 方法中被使用:

```
1. void Compute(OpKernelContext* context) override {
2.   // ...
3.   // 检查 preserve_index 范围是否合法
4.   OP_REQUIRES(context, preserve_index_ < input.dimension(0),
5.               errors::InvalidArgument("preserve_index out of range"));
6.   // 设置输出 tensor 所有的元素值为 0
7.   const int N = input.size();
8.   for (int i = 0; i < N; i++) {
9.     output_flat(i) = 0;
10.  }
11.   // 保存请求的输入值
12.   output_flat(preserve_index_) = input(preserve_index_);
13. }
```

为了维持向后兼容性, 将一个属性添加到一个已有的 Op 时, 必须指定一个默认值:

```

1. REGISTER_OP("ZeroOut")
2.     .Attr("preserve_index: int = 0")
3.     .Input("to_zero: int32")
4.     .Output("zeroed: int32");

```

属性类型

属性可以使用下面的类型：

- `string`：任何二进制字节流（UTF8 不是必须的）。
- `int`：一个有型整数。
- `float`：一个浮点数。
- `bool`：真或假。
- `type`：`DataType` 非引用类型之一。
- `shape`：一个 `TensorShapeProto`。
- `tensor`：一个 `TensorProto`。
- `list(<type>)`：`<type>` 列表，其中 `<type>` 是上述类型之一。
注意 `list(list(<type>))` 是无效的。

权威的列表以 `op_def_builder.cc:FinalizeAttr` 为准。

默认值和约束条件

属性可能有默认值，一些类型的属性可以有约束条件。为了定义一个有约束条件的属性，你可以使用下列的

`<attr-type-expr>` 形式：

- `{'<string1>', '<string2>'}`：属性值必须是一个字符串，取值可以为 `<string1>` 或 `<string2>`。

值的语法已经暗示了值的类型为 `string`，已经暗示了。下述语句模拟了一个枚举值：

```

1. REGISTER_OP("EnumExample")
2.     .Attr("e: {'apple', 'orange'}");

```

- `{<type1>, <type2>}`：值是 `type` 类型，且必须为 `<type1>` 或 `<type2>` 之一，当然 `<type1>` 和 `<type2>` 必须都是有效的 `tensor` 类型。

你无须指定属性的类型为 `type`，而是通过 `{...}` 语句给出一个类型列表。例如，在下面的例子里，

属性 `t` 的类型必须为 `int32`，`float`，或 `bool`：

```

1. REGISTER_OP("RestrictedTypeExample")
2.     .Attr("t: {int32, float, bool}");

```

- 这里有一些常见类型约束条件的快捷方式：

- `numbertype`：限制类型为数字类型，即非 `string` 非 `bool` 的类型。
- `realnumbertype`：与 `numbertype` 区别是不支持复杂类型。
- `quantizedtype`：与 `numbertype` 区别是只支持量化数值 (`quantized number type`)。

这些类型的列表在 `tensorflow/core/framework/types.h`

文件中通过函数定义（如 `NumberTypes()`）。

本例中属性 `t` 必须为某种数字类型：

```
1. REGISTER_OP("NumberType")
2.     .Attr("t: numbertype");
```

对于这个 Op：

```
1. tf.number_type(t=tf.int32) # 有效
2. tf.number_type(t=tf.bool) # 无效
```

- `int >= <n>`：值必须是一个整数，且取值大于等于 `<n>`，`<n>` 是一个自然数。

例如，下列 Op 注册操作指定了属性 `a` 的取值至少为 `2`。

```
1. REGISTER_OP("MinIntExample")
2.     .Attr("a: int >= 2");
```

- `list(<type>) >= <n>`：一个 `<type>` 类型列表，列表长度必须大于等于 `<n>`。

例如，下面的 Op 注册操作指定属性 `a` 是一个列表，列表中的元素类型是 `int32` 或 `float` 列表长度至少为3。

```
1. REGISTER_OP("TypeListExample")
2.     .Attr("a: list({int32, float}) >= 3");
```

通过添加 `= <default>` 到约束条件末尾，给一个属性设置默认值（使其在自动生成的代码里变成可选属性），如下：

```
1. REGISTER_OP("AttrDefaultExample")
2.     .Attr("i: int = 0");
```

默认值支持的语法将在最终 `GraphDef` 定义的 `protobuf` 表示中被使用。

下面是给所有类型赋予默认值的例子：

```

1. REGISTER_OP("AttrDefaultExampleForAllTypes")
2.     .Attr("s: string = 'foo'")
3.     .Attr("i: int = 0")
4.     .Attr("f: float = 1.0")
5.     .Attr("b: bool = true")
6.     .Attr("ty: type = DT_INT32")
7.     .Attr("sh: shape = { dim { size: 1 } dim { size: 2 } }")
8.     .Attr("te: tensor = { dtype: DT_INT32 int_val: 5 }")
9.     .Attr("l_empty: list(int) = []")
10.    .Attr("l_int: list(int) = [2, 3, 5, 7]");

```

请特别注意那些类型值里面包含的 `DT_*` 名称。

多态

Type Polymorphism

对于那些可以使用不同类型输入或产生不同类型输出的 Op，可以注册 Op 时为输入/输出类型里指定一个属性。

一般紧接着，会为每一个支持的类型注册一个 `OpKernel`。

例如，除了 `int32` 外，想要 `ZeroOut` Op 支持 `float`，注册代码如下：

```

1. REGISTER_OP("ZeroOut")
2.     .Attr("T: {float, int32}")
3.     .Input("to_zero: <b>T</b>")
4.     .Output("zeroed: <b>T</b>");

```

这段 Op 注册代码现在指定了输入的类型必须为 `float` 或 `int32`，而且既然输入和输出制定了同样的类型 `T`，输出也同样如此。

一个命名建议：`{#naming}` 输入，输出，和属性通常使用 `snake_case` 命名法。唯一的例外是属性被用作输入类型或是输入类型的一部分。当添加到图中时，这些属性可以被推断出来，因此不会出现在 Op 的函数里。例如，最后一个 `ZeroOut` 定义生成的 Python 函数如下：

```

1. def zero_out(to_zero, name=None):
2.     """...
3.     参数:
4.         to_zero: 一个 `Tensor`。必须为下列类型之一:
5.             `float32`, `int32`.
6.         name: 操作的名字 (可选).
7.
8.     返回值:
9.         一个 `Tensor`，类型和 `to_zero` 一样。

```

```
10. """
```

如果输入的 `to_zero` 是一个 `int32` 的 *tensor*, 然后 `T` 将被自动设置为 `int32` (实际上是 `DT_INT32`)。那些推导出的属性的名称字母全大写或采用驼峰命名法。

下面是一个输出类型自动推断的例子, 读者可以对比一下:

```
1. REGISTER_OP("StringToNumber")
2.   .Input("string_tensor: string")
3.   .Output("output: out_type")
4.   .Attr("out_type: {float, int32}");
5.   .Doc(R"doc(
6.     Converts each string in the input Tensor to the specified numeric type.
7.   )doc");
```

在这种情况下, 用户需要在生成的 *Python* 代码中指定输出类型。

```
1. def string_to_number(string_tensor, out_type=None, name=None):
2.     """将输入 Tensor 中的每一个字符串转化成指定的数字类型
3.
4.     参数:
5.         string_tensor: 一个 `string` 类型的 `Tensor`.
6.         out_type: 一个可选的 `tf.DType`, 取值为 `tf.float32, tf.int32`.
7.         默认值是 `tf.float32`.
8.         name: 操作的名称 (可选).
9.
10.    返回值:
11.        一个 `out_type` 类型的 `Tensor`.
12.    """
```

```
1. #include "tensorflow/core/framework/op_kernel.h"
2. class ZeroOutInt32Op : public OpKernel {
3.     // 和之前一样
4. };
5. class ZeroOutFloatOp : public OpKernel {
6. public:
7.     explicit ZeroOutFloatOp(OpKernelConstruction * context)
8.         : OpKernel(context) {}
9.     void Compute(OpKernelContext * context) override {
10.        // 获取输入 tensor
11.        const Tensor& input_tensor = context->input(0);
12.        auto input = input_tensor.flat<float>();
13.        // 创建一个输出 tensor
14.        Tensor * output = NULL;
15.        OP_REQUIRES_OK(context,
16.            context->allocate_output(0, input_tensor.shape(), &output));
```

```

17.     auto output_flat = output->template flat<float>();
18.     // 设置输出 tensor 的所有元素为 0
19.     const int N = input.size();
20.     for (int i = 0; i < N; i++) {
21.         output_flat(i) = 0;
22.     }<br/>
23.     // 保留第一个输入值
24.     if (N > 0) output_flat(0) = input(0);
25. }
26. };
27. // 注意, TypeConstraint<int32>("T") 意味着属性 "T" (在上面 Op 注册代码中
28. // 定义的) 必须是 "int32", 才能实例化.
29. REGISTER_KERNEL_BUILDER(
30.     Name("ZeroOut")
31.     .Device(DEVICE_CPU)
32.     .TypeConstraint<int32>("T"),
33.     ZeroOutOpInt32);
34. REGISTER_KERNEL_BUILDER(
35.     Name("ZeroOut")
36.     .Device(DEVICE_CPU)
37.     .TypeConstraint<float>("T"),
38.     ZeroOutFloatOp);

```

为了保持[向后兼容性](#)，你在为一个已有的 op 添加属性时，必须指定一个[默认值](#)：

```

1. REGISTER_OP("ZeroOut")
2.   .Attr("T: {float, int32} = DT_INT32")
3.   .Input("to_zero: T")
4.   .Output("zeroed: T")

```

如果需要添加更多类型，例如 `double`：

```

1. REGISTER_OP("ZeroOut")
2.   .Attr("T: {float, double, int32}")
3.   .Input("to_zero: T")
4.   .Output("zeroed: T");

```

为了避免为新增的类型写冗余的 `OpKernel` 代码，通常可以写一个 C++ 模板作为替代。当然，仍然需要为每一个重载版本定义一个 kernel 注册（`REGISTER_KERNEL_BUILDER` 调用）。

```

1. template <typename T>
2. class ZeroOutOp : public OpKernel {
3. public:
4.     explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
5.     void Compute(OpKernelContext* context) override {

```

```

6.    // 获取输入 tensor
7.    const Tensor& input_tensor = context->input(0);
8.    auto input = input_tensor.flat<T>();
9.    // 创建一个输出 tensor
10.   Tensor* output = NULL;
11.   OP_REQUIRES_OK(context,
12.                   context->allocate_output(0, input_tensor.shape(), &output));
13.   auto output_flat = output->template flat<T>();
14.   // 设置输出 tensor 的所有元素为 0
15.   const int N = input.size();
16.   for (int i = 0; i < N; i++) {
17.       output_flat(i) = 0;
18.   }
19.   // Preserve the first input value
20.   if (N > 0) output_flat(0) = input(0);
21. }
22. };
23. };<br/>
24. // 注意, TypeConstraint<int32>("T") 意味着属性 "T" (在上面 Op 注册代码中
25. // 定义的) 必须是 "int32", 才能实例化. </b>
26. REGISTER_KERNEL_BUILDER(
27.     Name("ZeroOut")
28.     .Device(DEVICE_CPU)
29.     .TypeConstraint<int32>("T"),
30.     ZeroOutOp<int32>);
31. REGISTER_KERNEL_BUILDER(
32.     Name("ZeroOut")
33.     .Device(DEVICE_CPU)
34.     .TypeConstraint<float>("T"),
35.     ZeroOutOp<float>);
36. REGISTER_KERNEL_BUILDER(
37.     Name("ZeroOut")
38.     .Device(DEVICE_CPU)
39.     .TypeConstraint<double>("T"),
40.     ZeroOutOp<double>);

```

如果有很多重载版本, 可以将注册操作通过一个宏来实现.

```

1.  #include "tensorflow/core/framework/op_kernel.h"
2.  #define REGISTER_KERNEL(type) \
3.      REGISTER_KERNEL_BUILDER( \
4.          Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
5.          ZeroOutOp<type>)
6.  REGISTER_KERNEL(int32);
7.  REGISTER_KERNEL(float);
8.  REGISTER_KERNEL(double);
9.  #undef REGISTER_KERNEL

```

取决于注册 kernel 使用哪些类型，你可能可以使用 `tensorflow/core/framework/register_types.h` 提供的宏：

```
1. #include "tensorflow/core/framework/op_kernel.h"
2. #include "tensorflow/core/framework/register_types.h"
3. REGISTER_OP("ZeroOut")
4.   .Attr("T: realnumbertype")
5.   .Input("to_zero: T")
6.   .Output("zeroed: T");
7. template <typename T>
8. class ZeroOutOp : public OpKernel { ... };
9. #define REGISTER_KERNEL(type) \
10.   REGISTER_KERNEL_BUILDER( \
11.     Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
12.     ZeroOutOp<type>)
13. TF_CALL_REAL_NUMBER_TYPES(REGISTER_KERNEL);
14. #undef REGISTER_KERNEL
```

列表输入和输出

除了能够使用不同类型的 tensor 作为输入或输出，Op 还支持使用多个 tensor 作为输入或输出。

在接下来的例子里，属性 `T` 存储了一个类型列表，并同时作为输入 `in` 和输出 `out` 的类型。

输入和输出均为指定类型的 tensor 列表。既然输入和输出的类型均为 `T`，它们的 tensor 数量和类型是一致的。

```
1. REGISTER_OP("PolymorphicListExample")
2.   .Attr("T: list(type)")
3.   .Input("in: T")
4.   .Output("out: T");
```

可以为列表中可存放的类型设置约束条件。在下一个例子中，输入是 `float` 和 `double` 类型的 tensor 列表。例如，这个 Op 可接受的输入类型为 `(float, double, float)` 的数据，且在此情况下，输出类型同样为 `(float, double, float)`。

```
1. REGISTER_OP("ListTypeRestrictionExample")
2.   .Attr("T: list({float, double})")
3.   .Input("in: T")
4.   .Output("out: T");
```

如果想要一个列表中的所有 `tensor` 是同一类型，你需要写下列代码：

```
1. REGISTER_OP("IntListInputExample")
2.   .Attr("N: int")
3.   .Input("in: N * int32")
4.   .Output("out: int32");
```

这段代码接受 `int32` `tensor` 列表，并用一个 `int` 属性 `N` 来指定列表的长度。

这也可用于[类型推断](#)。在下一个例子中，输入是一个 `tensor` 列表，长度为 `"N"`，类型为 `"T"`，输出是单个 `"T"` 的 `tensor`：

```
1. REGISTER_OP("SameListInputExample")
2.   .Attr("N: int")
3.   .Attr("T: type")
4.   .Input("in: N * T")
5.   .Output("out: T");
```

默认情况下，`tensor` 列表的最小长度为1。这个约束条件可以通过[为指定的属性增加一个](#) `">="` [约束](#)来变更：

```
1. REGISTER_OP("MinLengthIntListExample")
2.   .Attr("N: int >= 2")
3.   .Input("in: N * int32")
4.   .Output("out: int32");
```

同样的语法也适用于 `"list(type)"` 属性：

```
1. REGISTER_OP("MinimumLengthPolymorphicListExample")
2.   .Attr("T: list(type) >= 3")
3.   .Input("in: T")
4.   .Output("out: T");
```

输入和输出

总结一下上述内容，一个 `Op` 注册操作可以指定多个输入和输出：

```
1. REGISTER_OP("MultipleInsAndOuts")
2.   .Input("y: int32")
3.   .Input("z: float")
4.   .Output("a: string")
5.   .Output("b: int32");
```

每一个输入或输出形式如下：

```
1. <name>: <io-type-expr>
```

其中，`<name>` 以字母打头，且只能由数字，字母和下划线组成。`<io-type-expr>` 可以是下列类型表达式之一：

- `<type>`，一个合法的输入类型，如 `float`，`int32`，`string`。这可用于指定给定类型的单个 tensor。

参见[合法 Tensor 类型列表](#)。

```
1. REGISTER_OP("BuiltInTypesExample")
2.   .Input("integers: int32")
3.   .Input("complex_numbers: scomplex64");
```

- `<attr-type>`，一个[属性](#)和一个类型 `type` 或类型列表 `list(type)`（可能包含类型限制）。该语法可实现[多态 Op](#)。

```
1. REGISTER_OP("PolymorphicSingleInput")
2.   .Attr("T: type")
3.   .Input("in: T");
4. REGISTER_OP("RestrictedPolymorphicSingleInput")
5.   .Attr("T: {int32, int64}")
6.   .Input("in: T");
```

将属性的类型设置为 `list(type)` 将允许你接受一个序列的 tensor。

```
1. REGISTER_OP("ArbitraryTensorSequenceExample")
2.   .Attr("T: list(type)")
3.   .Input("in: T")
4.   .Output("out: T");
5. REGISTER_OP("RestrictedTensorSequenceExample")
6.   .Attr("T: list({int32, int64})")
7.   .Input("in: T")
8.   .Output("out: T");
```

注意，输入和输出均为 `T`，意味着输入和输出的类型与数量均相同。

- `<number> * <type>`，一组拥有相同类型的 tensor，`<number>` 是一个 `int` 类型属性的名称。
`<type>` 可以是一个类似于 `int32` 和 `float` 的特定类型，
 或者一个 `type` 类型属性的名字。前者的例子如下，该例子接受一个 `int32` tensor 列表

作为 Op 输入：

```
1. REGISTER_OP("Int32SequenceExample")
2.     .Attr("NumTensors: int")
3.     .Input("in: NumTensors * int32")
```

后者的例子如下，该例子接受一个泛型 tensor 列表作为 Op 输入：

```
1. REGISTER_OP("SameTypeSequenceExample")
2.     .Attr("NumTensors: int")
3.     .Attr("T: type")
4.     .Input("in: NumTensors * T")
```

- Tensor 的引用表示为 `Ref(<type>)`，其中 `<type>` 是上述类型之一。

一个命名建议：当使用属性表示一个输入的类型时，该类型可以被推断出来。实现该特性，将需要推断的类型用大写名称表示（如 `T` 或 `N`），其它的输入，输出，和属性像使用函数参数一样使用这些大写名称。参见之前的[命名建议](#)章节查看更多细节。

更多细节参见 [tensorflow/core/framework/op_def_builder.h](https://www.tensorflow.org/api_guides/python/op_def_builder)。

向后兼容性

通常，对规范的改变必须保持向后兼容性：Op 使用新规范后，需保证使用旧规范构造的序列化 GraphDef 仍能正确工作。

下面是几种保持向后兼容性的方式：

- 任何添加到 Op 的新属性必须有默认值，且默认值下的行为有明确定义。将一个非多态的操作变为多态操作，你必须为新的类型属性赋予默认值，以保持原始的函数签名。例如，有如下操作：

```
1. REGISTER_OP("MyGeneralUnaryOp")
2.     .Input("in: float")
3.     .Output("out: float");
```

可以通过下述方式将其变为多态，且保持向后兼容性：

```
1. REGISTER_OP("MyGeneralUnaryOp")
2.     .Input("in: T")
3.     .Output("out: T")
4.     .Attr("T: numerictype = float");
```

- 放宽一个属性的约束条件是安全的。例如，你可以将 `{int32, int64}` 变为 `{int32, int64,`

`float}` ,
或者, 将 `{"apple", "orange"}` 变为 `{"apple", "banana", "orange"}` .

2. 通过给 Op 名称添加一些项目中唯一的标识作为前缀, 来为新建的 Op 添加命名空间. 命名空间可以预防你的 Op 与 TensorFlow 未来版本里的内置 Op 产生命名冲突.

3. 超前计划! 尝试着去预测 Op 未来的用途, 超前设计, 毕竟, 一些签名的变更无法保证兼容性 (例如, 增加新的输入, 或将原来的单元输入变成一个列表).

如果不能以兼容的方式改变一个操作, 那就创建一个全新的操作, 来实现所需功能.

GPU 支持

你可以实现不同的 `OpKernel`, 将其中之一注册到 GPU, 另一个注册到 CPU, 正如[为不同的类型注册 kernel](#) 一样.

[tensorflow/core/kernels/](#) 中有一些 GPU 支持的例子.

注意, 一些 kernel 的 CPU 版本位于 `.cc` 文件, GPU 版本位于 `_gpu.cu.cc` 文件, 共享的代码位于 `.h` 文件.

例如, `pad` op 除了 GPU kernel 外的其它代码

均在 [tensorflow/core/kernels/pad_op.cc](#) 中. GPU kernel 位于

[tensorflow/core/kernels/pad_op_gpu.cu.cc](#) ,

共享的一个模板类代码定义在 [tensorflow/core/kernels/pad_op.h](#) .

需要注意的事情是, 即使使用 `pad` 的 GPU 版本时, 仍然需要将 `"paddings"` 输入放置到内存中.

为了实现这一点, 将输入或输出标记为必须保存在内存中, 为 kernel 注册一个 `HostMemory()` 调用.

如下:

```
1. #define REGISTER_GPU_KERNEL(T) \
2. REGISTER_KERNEL_BUILDER(Name("Pad") \
3.                             .Device(DEVICE_GPU) \
4.                             .TypeConstraint<T>("T") \
5.                             .HostMemory("paddings"), \
6.                             PadOp<GPUDevice, T>)
```

使用 Python 实现梯度

给定一个 Op 组成的图, TensorFlow 使用自动微分 (反向传播) 来添加新的 Op 以表示梯度运算, 同时

不影响已有的 Op (参见[梯度运算](#)).

为了使自动微分能够与新的 Op 协同工作，必须注册一个梯度函数，从 Op 的输入计算梯度，并返回代表梯度值的输出。

数学上，如果一个 Op 计算 $y = f(x)$ ，注册的梯度 Op 通过以下链式法则，将 $\frac{\partial}{\partial y}$ 的梯度运算转化为 $\frac{\partial}{\partial x}$ 的梯度运算。

$$\begin{aligned} \frac{\partial}{\partial x} &= \frac{\partial}{\partial y} \frac{\partial y}{\partial x} \\ &= \frac{\partial}{\partial y} \frac{\partial f}{\partial x} \end{aligned}$$

在 `ZeroOut` 的例子中，输入中只有一个项会影响输出，所以，代表输入的梯度值的 tensor 也只有一个输入项。如下所示：

```
1. from tensorflow.python.framework import ops
2. from tensorflow.python.ops import array_ops
3. from tensorflow.python.ops import sparse_ops
4.
5. @ops.RegisterGradient("ZeroOut")
6. def _zero_out_grad(op, grad):
7.     """`zero_out` 的梯度。
8.
9.     参数:
10.         op: 欲进行微分的 `zero_out` 操作，可以用于获取原始 Op 的输入和输出。
11.         grad: 代表 `zero_out` 输出的梯度 Op。
12.
13.     返回:
14.         代表输入 `zero_out` 的微分。
15.     """
16.     to_zero = op.inputs[0]
17.     shape = array_ops.shape(to_zero)
18.     index = array_ops.zeros_like(shape)
19.     first_grad = array_ops.reshape(grad, [-1])[0]
20.     to_zero_grad = sparse_ops.sparse_to_dense(index, shape, first_grad, 0)
21.     return [to_zero_grad] # 单个 Tensor 的列表，既然只有一个输入
```

使用 `ops.RegisterGradient`

注册梯度函数需要注意的一些细节：

- 对于仅有一个输出的 Op，梯度函数使用 `Operation` 和 `op` 作为一个 `Tensor` `grad` 作为参数，并从 `op.inputs[i]` , `op.outputs[i]` ,

和 `grad` 构建新的 Op。属性的信息可以通过 `op.get_attr` 获取。

- 如果 Op 有多个输出，梯度函数将使用 `op` 和 `grads` 作为参数，其中，`grads` 是一个梯度 Op 的列表，为每一个输出计算梯度。梯度函数的输出必须是一个 `Tensor` 对象列表，对应到每一个输入的梯度。
- 如果没有为一些输入定义梯度，譬如用作索引的整型，这些输入返回的梯度为 `None`。举个例子，如果一个 Op 的输入为一个浮点数 tensor `x` 和一个整型索引 `i`，那么梯度函数将返回 `[x_grad, None]`。
- 如果梯度对于一个 Op 来说毫无意义，使用 `ops.NoGradient("OpName")` 禁用自动差分。

注意当梯度函数被调用时，作用的对象是数据流图中的 Op，而不是 tensor 数据本身。因此，只有在图运行时，梯度运算才会被其它 tensorflow Op 的执行动作所触发。

在 Python 中实现一个形状函数

TensorFlow Python API 有一个“形状推断”功能，可以不执行图就获取 tensor 的形状信息。

形状推断功能藉由每一个 Op 类型注册的“形状函数”来支持，该函数有两个规则：假设所有输入的

形状必须是兼容的，以及指定输出的形状。一个形状函数以一个 `Operation` 作为输入，返回一个 `TensorShape`

对象列表（每一个输出一个对象）。使用 `tf.RegisterShape` 装饰器注册形状函数。例如，上文定义的 `ZeroOut` Op 的形状函数如下：

```
1. @tf.RegisterShape("ZeroOut"):
2. def _zero_out_shape(op):
3.     """ZeroOut Op 的形状函数.
4.
5.     这是 ZeroOut 形状函数的无约束版本，为每一个输出产生的形状和对应的输入一样.
6.     """
7.     return [op.inputs[0].get_shape()]
```

一个形状函数也可以约束输入的形状。下面是 `ZeroOut` 形状函数的 `vector` 输入约束版本：

```
1. @tf.RegisterShape("ZeroOut"):
2. def _zero_out_shape(op):
```

```

3.  """ZeroOut Op 的形状函数.
4.
5.  这是 ZeroOut 形状函数的约束版本, 要输入的 rank 必须是 1 (即使一个 vector).
6.  """
7.  input_shape = op.inputs[0].get_shape().with_rank(1)
8.  return [input_shape]

```

如果 Op 是多输入的多态 Op, 使用操作的属性来决定需要检查的形状数量:

```

1. @tf.RegisterShape("IntListInputExample")
2. def _int_list_input_example_shape(op):
3.     """ "IntListInputExample" Op 的形状函数.
4.
5.     所有的输入和输出是同大小的矩阵.
6.     """
7.     output_shape = tf.TensorShape(None)
8.     for input in op.inputs:
9.         output_shape = output_shape.merge_with(input.get_shape().with_rank(2))
10.    return [output_shape]

```

既然形状推断是一个可选的特性, 且 tensor 的形状可能动态变化, 形状函数必须足够健壮, 能够处理任意

输入形状信息缺失的情形. `merge_with` 方法能够帮助

调用者判断两个形状是否是一样的, 即使两个形状的信息不全, 该函数同样有效. 所有的[标准 Python Op](#)

的形状函数都已经定义好了, 并且已经有很多不同的使用示例.

原文: [Adding a New Op](#) 翻译: @doc001 校对: @ZHNathanielLee

自定义数据读取

- [自定义数据读取](#)
 - [主要内容](#)
 - [自定义数据读取](#)
 - [编写一个文件格式读写器](#)
 - [编写一个记录格式Op](#)

自定义数据读取

基本要求：

- 熟悉 C++ 编程。
- 确保[下载 TensorFlow 源文件](#)，并可编译使用。

我们将支持文件格式的任务分成两部分：

- 文件格式： 我们使用 *Reader* Op来从文件中读取一个 *record*（可以使任意字符串）。
- 记录格式： 我们使用解码器或者解析运算将一个字符串记录转换为TensorFlow可以使用的张量。

例如， 读取一个

[CSV 文件](#)，我们使用

[一个文本读写器](#)，

然后是从一行文本中解析CSV数据的运算。

主要内容

自定义数据读取

- [编写一个文件格式读写器](#)
- [编写一个记录格式Op](#)

编写一个文件格式读写器

Reader 是专门用来读取文件中的记录的。TensorFlow中内建了一些读写器Op的实例：

- [tf.TFRecordReader](#)
(代码位于[kernels/tf_record_reader_op.cc](#))
- [tf.FixedLengthRecordReader](#)

(代码位于 `kernels/fixed_length_record_reader_op.cc`)

- `tf.TextLineReader`

(代码位于 `kernels/text_line_reader_op.cc`)

你可以看到这些读写器的界面是一样的，唯一的差异是在它们的构造函数中。最重要的方法是 `Read`。

它需要一个行列参数，通过这个行列参数，可以在需要的时候随时读取文件名（例如：当 `Read Op` 首次运行，或者前一个 `Read`` 从一个文件中读取最后一条记录时）。它将会生成两个标量张量：一个字符串和一个字符串键值。

新创建一个名为 `SomeReader` 的读写器，需要以下步骤：

1. 在 C++ 中，定义一个 `tensorflow::ReaderBase` 的子类，命名为 “SomeReader”。
2. 在 C++ 中，注册一个新的读写器 `Op` 和 `Kernel`，命名为 “SomeReader”。
3. 在 Python 中，定义一个 `tf.ReaderBase` 的子类，命名为 “SomeReader”。

你可以把所有的 C++ 代码放在

`tensorflow/core/user_ops/some_reader_op.cc` 文件中。读取文件的代码将被嵌入到 C++ 的

`ReaderBase` 类的迭代中。这个 `ReaderBase` 类是在

`tensorflow/core/kernels/reader_base.h` 中定义的。

你需要执行以下的方法：

- `OnWorkStartedLocked`：打开下一个文件
- `ReadLocked`：读取一个记录或报告 EOF/error
- `OnWorkFinishedLocked`：关闭当前文件
- `ResetLocked`：清空记录，例如：一个错误记录

以上这些方法的名字后面都带有 “Locked”，表示 `ReaderBase` 在调用任何一个方法之前确保获得互斥锁，这样就不用担心线程安全（虽然只保护了该类中的元素而不是全局的）。

对于 `OnWorkStartedLocked`，需要打开的文件名是 `current_work()` 函数的返回值。此时的 `ReadLocked` 的数字签名如下：

```
1. Status ReadLocked(string* key, string* value, bool* produced, bool* at_end)
```

如果 `ReadLocked` 从文件中成功读取了一条记录，它将更新为：

- `*key`：记录的标志位，通过该标志位可以重新定位到该记录。可以包含从 `current_work()` 返回值获得的文件名，并追加一个记录号或其他信息。
- `*value`：包含记录的内容。
- `*produced`：设置为 `true`。

当你在文件 (EOF) 末尾, 设置 `*at_end` 为 `true`, 在任何情况下, 都将返回 `Status::OK()`。当出现错误的时候, 只需要使用

[tensorflow/core/lib/core/errors.h](#) 中的一个辅助功能就可以简单地返回, 不需要做任何参数修改。

接下来你讲创建一个实际的读写器Op。 如果你已经熟悉了[添加新的Op](#) 那会很有帮助。 主要步骤如下:

- 注册Op。
- 定义并注册 `OpKernel`。

要注册Op, 你需要用到一个调用指令定义在

[tensorflow/core/framework/op.h](#)中的`REGISTER_OP`。

读写器 Op 没有输入, 只有 `Ref(string)` 类型的单输出。它们调用 `SetIsStateful()`, 并有一个

`container` 字符串和 `shared_name` 属性。你可以在一个 `Doc` 中定义配置或包含文档的额外属性。 例如: 详见

[tensorflow/core/ops/io_ops.cc](#)等:

```
1. #include "tensorflow/core/framework/op.h"
2. REGISTER_OP("TextLineReader")
3.   .Output("reader_handle: Ref(string)")
4.   .Attr("skip_header_lines: int = 0")
5.   .Attr("container: string = ''")
6.   .Attr("shared_name: string = ''")
7.   .SetIsStateful()
8.   .Doc(R"doc(
9. A Reader that outputs the lines of a file delimited by '\n'.
10. )doc");
```

要定义一个 `OpKernel`, 读写器可以使用定义

在[tensorflow/core/framework/reader_op_kernel.h](#)中的 `ReaderOpKernel` 的递减快捷方式, 并运行一个叫 `SetReaderFactory` 的构造函数。

定义所需要的类之后, 你需要通过 `REGISTER_KERNEL_BUILDER(...)` 注册这个类。

一个没有属性的例子:

```
1. #include "tensorflow/core/framework/reader_op_kernel.h"
2. class TFRecordReaderOp : public ReaderOpKernel {
3. public:
4.   explicit TFRecordReaderOp(OpKernelConstruction* context)
5.     : ReaderOpKernel(context) {
6.     Env* env = context->env();
7.     SetReaderFactory([this, env]() { return new TFRecordReader(name(), env); });
```

```

8.     }
9. };
10. REGISTER_KERNEL_BUILDER(Name("TFRecordReader").Device(DEVICE_CPU),
11.                           TFRecordReaderOp);

```

一个带有属性的例子：

```

1. #include "tensorflow/core/framework/reader_op_kernel.h"
2. class TextLineReaderOp : public ReaderOpKernel {
3. public:
4.     explicit TextLineReaderOp(OpKernelConstruction* context)
5.         : ReaderOpKernel(context) {
6.         int skip_header_lines = -1;
7.         OP_REQUIRES_OK(context,
8.                         context->GetAttr("skip_header_lines", &skip_header_lines));
9.         OP_REQUIRES(context, skip_header_lines >= 0,
10.                     errors::InvalidArgument("skip_header_lines must be >= 0 not ",
11.                                              skip_header_lines));
12.         Env* env = context->env();
13.         SetReaderFactory([this, skip_header_lines, env]() {
14.             return new TextLineReader(name(), skip_header_lines, env);
15.         });
16.     }
17. };
18. REGISTER_KERNEL_BUILDER(Name("TextLineReader").Device(DEVICE_CPU),
19.                           TextLineReaderOp);

```

最后一步是添加 Python 包装器，你需要将 `tensorflow.python.ops.io_ops` 导入到 `tensorflow/python/user_ops/user_ops.py`，并添加一个 `io_ops.ReaderBase` 的衍生函数。

```

1. from tensorflow.python.framework import ops
2. from tensorflow.python.ops import common_shapes
3. from tensorflow.python.ops import io_ops
4. class SomeReader(io_ops.ReaderBase):
5.     def __init__(self, name=None):
6.         rr = gen_user_ops.some_reader(name=name)
7.         super(SomeReader, self).__init__(rr)
8.     ops.NoGradient("SomeReader")
9.     ops.RegisterShape("SomeReader")(common_shapes.scalar_shape)

```

你可以在 `tensorflow/python/ops/io_ops.py` 中查看一些范例。

编写一个记录格式Op

一般来说，这是一个普通的Op， 需要一个标量字符串记录作为输入， 因此遵循 [添加Op的说明](#)。你可以选择一个标量字符串作为输入， 并包含在错误消息中报告不正确的格式化数据。

用于解码记录的运算实例：

- `tf.parse_single_example`
(and `tf.parse_example`)
- `tf.decode_csv`
- `tf.decode_raw`

请注意，使用多个Op 来解码某个特定的记录格式也是有效的。 例如，你有一张以字符串格式保存在 `tf.train.Example` 协议缓冲区的图像文件。

根据该图像的格式， 你可能从

`tf.parse_single_example` 的Op 读取响应输出并调用 `tf.decode_jpeg`，
`tf.decode_png`， 或者
`tf.decode_raw`。通过读取 `tf.decode_raw` 的响应输出并使用`tf.slice` 和
`tf.reshape` 来提取数据是通用的方法。

原文： [Custom Data Readers](#) 翻译： [@derekshang](#) 校对： [Wiki](#)

使用gpu

- 使用 GPUs
 - 支持的设备
 - 记录设备指派情况
 - 手工指派设备
 - 在多GPU系统里使用单一GPU
 - 使用多个 GPU

使用 GPUs

支持的设备

在一套标准的系统上通常有多个计算设备。TensorFlow 支持 CPU 和 GPU 这两种设备。我们用指定字符串

`strings` 来标识这些设备。比如：

- `"/cpu:0"`：机器中的 CPU
- `"/gpu:0"`：机器中的 GPU，如果你有一个的话。
- `"/gpu:1"`：机器中的第二个 GPU，以此类推...

如果一个 TensorFlow 的 operation 中兼有 CPU 和 GPU 的实现，当这个算子被指派设备时，GPU 有优先权。比如 `matmul` 中 CPU 和 GPU kernel 函数都存在。那么在 `cpu:0` 和 `gpu:0` 中，`matmul` operation 会被指派给 `gpu:0`。

记录设备指派情况

为了获取你的 operations 和 Tensor 被指派到哪个设备上运行，用 `log_device_placement` 新建一个 `session`，并设置为 `True`。

```
1. # 新建一个 graph.
2. a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
3. b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
4. c = tf.matmul(a, b)
5. # 新建session with log_device_placement并设置为True.
6. sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
7. # 运行这个 op.
8. print sess.run(c)
```

你应该能看见以下输出：

```
1. Device mapping:
2. /job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus
3. id: 0000:05:00.0
4. b: /job:localhost/replica:0/task:0/gpu:0
5. a: /job:localhost/replica:0/task:0/gpu:0
6. MatMul: /job:localhost/replica:0/task:0/gpu:0
7. [[ 22.  28.]
8.  [ 49.  64.]]
```

手工指派设备

如果你不想使用系统来为 operation 指派设备，而是手工指派设备，你可以用 `with tf.device` 创建一个设备环境，这个环境下的 operation 都统一运行在环境指定的设备上。

```
1. # 新建一个graph.
2. with tf.device('/cpu:0'):
3.     a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
4.     b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
5.     c = tf.matmul(a, b)
6. # 新建session with log_device_placement并设置为True.
7. sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
8. # 运行这个op.
9. print sess.run(c)
```

你会发现现在 `a` 和 `b` 操作都被指派给了 `cpu:0`。

```
1. Device mapping:
2. /job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus
3. id: 0000:05:00.0
4. b: /job:localhost/replica:0/task:0/cpu:0
5. a: /job:localhost/replica:0/task:0/cpu:0
6. MatMul: /job:localhost/replica:0/task:0/gpu:0
7. [[ 22.  28.]
8.  [ 49.  64.]]
```

在多GPU系统里使用单一GPU

如果你的系统里有多 GPU，那么 ID 最小的 GPU 会默认使用。如果你想用别的 GPU，可以用下面的方法显式的声明你的偏好：

```

1. # 新建一个 graph.
2. with tf.device('/gpu:2'):
3.     a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
4.     b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
5.     c = tf.matmul(a, b)
6. # 新建 session with log_device_placement 并设置为 True.
7. sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
8. # 运行这个 op.
9. print sess.run(c)

```

如果你指定的设备不存在，你会收到 `InvalidArgumentError` 错误提示：

```

1. InvalidArgumentError: Invalid argument: Cannot assign a device to node 'b':
2. Could not satisfy explicit device specification '/gpu:2'
3. [[Node: b = Const[dtype=DT_FLOAT, value=Tensor<type: float shape: [3,2]
4.   values: 1 2 3...>, _device="/gpu:2"]()]]

```

为了避免出现你指定的设备不存在这种情况，你可以在创建的 `session` 里把参数 `allow_soft_placement` 设置为 `True`，这样 `tensorflow` 会自动选择一个存在并且支持的设备来运行 operation.

```

1. # 新建一个 graph.
2. with tf.device('/gpu:2'):
3.     a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
4.     b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
5.     c = tf.matmul(a, b)
6. # 新建 session with log_device_placement 并设置为 True.
7. sess = tf.Session(config=tf.ConfigProto(
8.     allow_soft_placement=True, log_device_placement=True))
9. # 运行这个 op.
10. print sess.run(c)

```

使用多个 GPU

如果你想让 `TensorFlow` 在多个 GPU 上运行，你可以建立 `multi-tower` 结构，在这个结构里每个 `tower` 分别被指配给不同的 GPU 运行。比如：

```

1. # 新建一个 graph.
2. c = []
3. for d in ['/gpu:2', '/gpu:3']:
4.     with tf.device(d):
5.         a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
6.         b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])

```

```

7.     c.append(tf.matmul(a, b))
8.     with tf.device('/cpu:0'):
9.         sum = tf.add_n(c)
10.    # 新建session with log_device_placement并设置为True.
11.    sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
12.    # 运行这个op.
13.    print sess.run(sum)

```

你会看到如下输出：

```

1. Device mapping:
2. /job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K20m, pci bus
3. id: 0000:02:00.0
4. /job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K20m, pci bus
5. id: 0000:03:00.0
6. /job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K20m, pci bus
7. id: 0000:83:00.0
8. /job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K20m, pci bus
9. id: 0000:84:00.0
10. Const_3: /job:localhost/replica:0/task:0/gpu:3
11. Const_2: /job:localhost/replica:0/task:0/gpu:3
12. MatMul_1: /job:localhost/replica:0/task:0/gpu:3
13. Const_1: /job:localhost/replica:0/task:0/gpu:2
14. Const: /job:localhost/replica:0/task:0/gpu:2
15. MatMul: /job:localhost/replica:0/task:0/gpu:2
16. AddN: /job:localhost/replica:0/task:0/cpu:0
17. [[ 44.  56.]
18. [ 98. 128.]]

```

[cifar10 tutorial](#) 这个例子很好的演示了怎样用GPU集群训练。

原文:[using_gpu](#)

翻译:[@lianghyv](#) 校对:[Wiki](#)

共享变量

- 共享变量
 - 问题
 - 变量作用域实例
 - 变量作用域是怎么工作的？
 - 理解 `tf.get_variable()`
 - `tf.variable_scope()` 基础
 - 获取变量作用域
 - 变量作用域中的初始化器
 - 在 `tf.variable_scope()` 中ops的名称
 - 使用实例

共享变量

你可以在[怎么使用变量](#)中所描述的方式来创建，初始化，保存及加载单一的变量。但是当创建复杂的模块时，通常你需要共享大量变量集并且如果你还想在同一个地方初始化这所有的变量，我们又该怎么做呢。本教程就是演示如何使用 `tf.variable_scope()` 和 `tf.get_variable()` 两个方法来实现这一点。

问题

假设你为图片过滤器创建了一个简单的模块，和我们的[卷积神经网络教程](#)模块相似，但是这里包括两个卷积（为了简化实例这里只有两个）。如果你仅使用 `tf.Variable` 变量，那么你的模块就如[怎么使用变量](#)里面所解释的是一样的模块。

```
1. def my_image_filter(input_images):
2.     conv1_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
3.                                 name="conv1_weights")
4.     conv1_biases = tf.Variable(tf.zeros([32]), name="conv1_biases")
5.     conv1 = tf.nn.conv2d(input_images, conv1_weights,
6.                           strides=[1, 1, 1, 1], padding='SAME')
7.     relu1 = tf.nn.relu(conv1 + conv1_biases)
8.
9.     conv2_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
10.                                name="conv2_weights")
11.    conv2_biases = tf.Variable(tf.zeros([32]), name="conv2_biases")
12.    conv2 = tf.nn.conv2d(relu1, conv2_weights,
13.                          strides=[1, 1, 1, 1], padding='SAME')
14.    return tf.nn.relu(conv2 + conv2_biases)
```

你很容易想到，模块集很快就比一个模块变得更为复杂，仅在这里我们就有了四个不同的变

量: `conv1_weights` , `conv1_biases` , `conv2_weights` , 和 `conv2_biases` .

当我们想重用这个模块时问题还在增多. 假设你想把你的图片过滤器运用到两张不同的图片, `image1` 和 `image2` . 你想通过拥有同一个参数的同一个过滤器来过滤两张图片, 你可以调用 `my_image_filter()` 两次, 但是这会产生两组变量.

```
1. # First call creates one set of variables.
2. result1 = my_image_filter(image1)
3. # Another set is created in the second call.
4. result2 = my_image_filter(image2)
```

通常共享变量的方法就是在单独的代码块中来创建他们并且通过使用他们的函数. 如使用字典的例子:

```
1. variables_dict = {
2.     "conv1_weights": tf.Variable(tf.random_normal([5, 5, 32, 32]),
3.         name="conv1_weights")
4.     "conv1_biases": tf.Variable(tf.zeros([32]), name="conv1_biases")
5.     ... etc. ...
6. }
7.
8. def my_image_filter(input_images, variables_dict):
9.     conv1 = tf.nn.conv2d(input_images, variables_dict["conv1_weights"],
10.         strides=[1, 1, 1, 1], padding='SAME')
11.     relu1 = tf.nn.relu(conv1 + variables_dict["conv1_biases"])
12.
13.     conv2 = tf.nn.conv2d(relu1, variables_dict["conv2_weights"],
14.         strides=[1, 1, 1, 1], padding='SAME')
15.     return tf.nn.relu(conv2 + variables_dict["conv2_biases"])
16.
17. # The 2 calls to my_image_filter() now use the same variables
18. result1 = my_image_filter(image1, variables_dict)
19. result2 = my_image_filter(image2, variables_dict)
```

虽然使用上面的方式创建变量是很方便的, 但是在这个模块代码之外却破坏了其封装性:

- 在构建试图的代码中标明变量的名字, 类型, 形状来创建.
- 当代码改变了, 调用的地方也许就会产生或多或少或不同类型的变量.

解决此类问题的方法之一就是使用类来创建模块, 在需要的地方使用类来小心地管理他们需要的变量. 一个更高明的做法, 不用调用类, 而是利用TensorFlow 提供了变量作用域 机制, 当构建一个视图时, 很容易就可以共享命名过的变量.

变量作用域实例

变量作用域机制在TensorFlow中主要由两部分组成:

- `tf.get_variable(<name>, <shape>, <initializer>)` :
通过所给的名字创建或是返回一个变量。
- `tf.variable_scope(<scope_name>)` :
通过 `tf.get_variable()` 为变量名指定命名空间。

方法 `tf.get_variable()` 用来获取或创建一个变量，而不是直接调用 `tf.Variable`。它采用的不是像 `tf.Variable` 这样直接获取值来初始化的方法。一个初始化就是一个方法，创建其形状并且为这个形状提供一个张量。这里有一些在TensorFlow中使用的初始化变量：

- `tf.constant_initializer(value)` 初始化一切所提供的值，
- `tf.random_uniform_initializer(a, b)` 从a到b均匀初始化，
- `tf.random_normal_initializer(mean, stddev)` 用所给平均值和标准差初始化均匀分布。

为了了解 `tf.get_variable()` 怎么解决前面所讨论的问题，让我们在单独的方法里面创建一个卷积来重构一下代码，命名为 `conv_relu`：

```
1. def conv_relu(input, kernel_shape, bias_shape):
2.     # Create variable named "weights".
3.     weights = tf.get_variable("weights", kernel_shape,
4.                               initializer=tf.random_normal_initializer())
5.     # Create variable named "biases".
6.     biases = tf.get_variable("biases", bias_shape,
7.                              initializer=tf.constant_initializer(0.0))
8.     conv = tf.nn.conv2d(input, weights,
9.                          strides=[1, 1, 1, 1], padding='SAME')
10.    return tf.nn.relu(conv + biases)
```

这个方法中用了 `"weights"` 和 `"biases"` 两个简称。而我们更偏向于用 `conv1` 和 `conv2` 这两个变量的写法，但是不同的变量需要不同的名字。这就是 `tf.variable_scope()` 变量起作用的地方。他为变量指定了相应的命名空间。

```
1. def my_image_filter(input_images):
2.     with tf.variable_scope("conv1"):
3.         # Variables created here will be named "conv1/weights", "conv1/biases".
4.         relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
5.     with tf.variable_scope("conv2"):
6.         # Variables created here will be named "conv2/weights", "conv2/biases".
7.         return conv_relu(relu1, [5, 5, 32, 32], [32])
```

现在，让我们看看当我们调用 `my_image_filter()` 两次时究竟会发生了什么。

```
1. result1 = my_image_filter(image1)
2. result2 = my_image_filter(image2)
3. # Raises ValueError(... conv1/weights already exists ...)
```

就像你看见的一样, `tf.get_variable()` 会检测已经存在的变量是否已经共享. 如果你想共享他们, 你需要像下面使用的一样, 通过 `reuse_variables()` 这个方法指定.

```
1. with tf.variable_scope("image_filters") as scope:
2.     result1 = my_image_filter(image1)
3.     scope.reuse_variables()
4.     result2 = my_image_filter(image2)
```

用这种方式来共享变量是非常好的, 轻量级而且安全.

变量作用域是怎么工作的？

理解

`tf.get_variable()`

为了理解变量作用域, 首先完全理解 `tf.get_variable()` 是怎么工作的是很有必要的. 通常我们就是这样调用 `tf.get_variable` 的.

```
1. v = tf.get_variable(name, shape, dtype, initializer)
```

此调用做了有关作用域的两件事中的其中之一, 方法调入. 总的有两种情况.

- 情况1: 当 `tf.get_variable_scope().reuse == False` 时, 作用域就是为创建新变量所设置的.

这种情况下, `v` 将通过 `tf.Variable` 所提供的形状和数据类型来重新创建. 创建变量的全称将会由当前变量作用域名+所提供的 `名字` 所组成, 并且还会检查来确保没有任何变量使用这个全称. 如果这个全称已经有一个变量使用了, 那么方法将会抛出 `ValueError` 错误. 如果一个变量被创建, 他将会用 `initializer(shape)` 进行初始化. 比如:

```
1. with tf.variable_scope("foo"):
2.     v = tf.get_variable("v", [1])
3.     assert v.name == "foo/v:0"
```

- 情况1: 当 `tf.get_variable_scope().reuse == True` 时, 作用域是为重用变量所设置

这种情况下, 调用就会搜索一个已经存在的变量, 他的全称和当前变量的作用域名+所提供的 `名字` 是否相等. 如果不存在相应的变量, 就会抛出 `ValueError` 错误. 如果变量找到了, 就返回这个变量. 如下:

```
1. with tf.variable_scope("foo"):
2.     v = tf.get_variable("v", [1])
3.     with tf.variable_scope("foo", reuse=True):
4.         v1 = tf.get_variable("v", [1])
```

```
5. assert v1 == v
```

```
tf.variable_scope()
```

基础

知道 `tf.get_variable()` 是怎么工作的, 使得理解变量作用域变得很容易. 变量作用域的主方法带有一个名称, 它将会作为前缀用于变量名, 并且带有一个重用标签来区分以上的两种情况. 嵌套的作用域附加名字所用的规则和文件目录的规则很类似:

```
1. with tf.variable_scope("foo"):
2.     with tf.variable_scope("bar"):
3.         v = tf.get_variable("v", [1])
4. assert v.name == "foo/bar/v:0"
```

当前变量作用域可以用 `tf.get_variable_scope()` 进行检索并且 `reuse` 标签可以通过调用 `tf.get_variable_scope().reuse_variables()` 设置为 `True` .

```
1. with tf.variable_scope("foo"):
2.     v = tf.get_variable("v", [1])
3.     tf.get_variable_scope().reuse_variables()
4.     v1 = tf.get_variable("v", [1])
5. assert v1 == v
```

注意你不能设置 `reuse` 标签为 `False` . 其中的原因就是允许改写创建模块的方法. 想一下你前面写得方法 `my_image_filter(inputs)` . 有人在变量作用域内调用 `reuse=True` 是希望所有内部变量都被重用. 如果允许在方法体内强制执行 `reuse=False` , 将会打破内部结构并且用这种方法使得很难再共享参数.

即使你不能直接设置 `reuse` 为 `False` , 但是你可以输入一个重用变量作用域, 然后就释放掉, 就成为非重用的变量. 当打开一个变量作用域时, 使用 `reuse=True` 作为参数是可以的. 但也要注意, 同一个原因, `reuse` 参数是不可继承. 所以当你打开一个重用变量作用域, 那么所有的子作用域也将会被重用.

```
1. with tf.variable_scope("root"):
2.     # At start, the scope is not reusing.
3.     assert tf.get_variable_scope().reuse == False
4.     with tf.variable_scope("foo"):
5.         # Opened a sub-scope, still not reusing.
6.         assert tf.get_variable_scope().reuse == False
7.         with tf.variable_scope("foo", reuse=True):
8.             # Explicitly opened a reusing scope.
9.             assert tf.get_variable_scope().reuse == True
10.            with tf.variable_scope("bar"):
11.                # Now sub-scope inherits the reuse flag.
12.                assert tf.get_variable_scope().reuse == True
```

```
13.     # Exited the reusing scope, back to a non-reusing one.
14.     assert tf.get_variable_scope().reuse == False
```

获取变量作用域

在上面的所有例子中，我们共享参数只因为他们的名字是一致的，那是因为我们开启一个变量作用域重用刚好用了同一个字符串。在更复杂的情况，他可以通过变量作用域对象来使用，而不是通过依赖于右边的名字来使用。为此，变量作用域可以被获取并使用，而不是仅作为当开启一个新的变量作用域的名字。

```
1. with tf.variable_scope("foo") as foo_scope:
2.     v = tf.get_variable("v", [1])
3. with tf.variable_scope(foo_scope)
4.     w = tf.get_variable("w", [1])
5. with tf.variable_scope(foo_scope, reuse=True)
6.     v1 = tf.get_variable("v", [1])
7.     w1 = tf.get_variable("w", [1])
8. assert v1 == v
9. assert w1 == w
```

当开启一个变量作用域，使用一个预先已经存在的作用域时，我们会跳过当前变量作用域的前缀而直接成为一个完全不同的作用域。这就是我们做得完全独立的地方。

```
1. with tf.variable_scope("foo") as foo_scope:
2.     assert foo_scope.name == "foo"
3. with tf.variable_scope("bar")
4.     with tf.variable_scope("baz") as other_scope:
5.         assert other_scope.name == "bar/baz"
6.         with tf.variable_scope(foo_scope) as foo_scope2:
7.             assert foo_scope2.name == "foo" # Not changed.
```

变量作用域中的初始化器

使用 `tf.get_variable()` 允许你重写方法来创建或者重用变量，并且可以被外部透明调用。但是如果我们想改变创建变量的初始化器那要怎么做呢？是否我们需要为所有的创建变量方法传递一个额外的参数呢？那在大多数情况下，当我们想在一个地方并且为所有的方法的所有变量设置一个默认初始化器，那又改怎么做呢？为了解决这些问题，变量作用域可以携带一个默认的初始化器。他可以被子作用域继承并传递给 `tf.get_variable()` 调用。但是如果其他初始化器被明确地指定，那么他将会被重写。

```
1. with tf.variable_scope("foo", initializer=tf.constant_initializer(0.4)):
2.     v = tf.get_variable("v", [1])
3.     assert v.eval() == 0.4 # Default initializer as set above.
4.     w = tf.get_variable("w", [1], initializer=tf.constant_initializer(0.3)):
```

```

5.     assert w.eval() == 0.3 # Specific initializer overrides the default.
6.     with tf.variable_scope("bar"):
7.         v = tf.get_variable("v", [1])
8.         assert v.eval() == 0.4 # Inherited default initializer.
9.     with tf.variable_scope("baz", initializer=tf.constant_initializer(0.2)):
10.        v = tf.get_variable("v", [1])
11.        assert v.eval() == 0.2 # Changed default initializer.

```

在 `tf.variable_scope()` 中 ops 的名称

我们讨论 `tf.variable_scope` 怎么处理变量的名字. 但是又是如何在作用域中影响到其他 ops 的名字呢? ops 在一个变量作用域的内部创建, 那么他应该是共享他的名字, 这是很自然的想法. 出于这样的原因, 当我们用 `with tf.variable_scope("name")` 时, 这就间接地开启了一个 `tf.name_scope("name")`. 比如:

```

1. with tf.variable_scope("foo"):
2.     x = 1.0 + tf.get_variable("v", [1])
3.     assert x.op.name == "foo/add"

```

名称作用域可以被开启并添加到一个变量作用域中, 然后他们只会影响到 ops 的名称, 而不会影响到变量.

```

1. with tf.variable_scope("foo"):
2.     with tf.name_scope("bar"):
3.         v = tf.get_variable("v", [1])
4.         x = 1.0 + v
5.     assert v.name == "foo/v:0"
6.     assert x.op.name == "foo/bar/add"

```

当用一个引用对象而不是一个字符串去开启一个变量作用域时, 我们就不会为 ops 改变当前的名称作用域.

使用实例

这里有一些指向怎么使用变量作用域的文件. 特别是, 他被大量用于 [时间递归神经网络](#) 和 `sequence-to-sequence` 模型,

File	What's in it?
<code>models/image/cifar10.py</code>	图像中检测对象的模型.
<code>models/rnn/rnn_cell.py</code>	时间递归神经网络的元方法集.
<code>models/rnn/seq2seq.py</code>	为创建 <code>sequence-to-sequence</code> 模型的方法集.

原文: [Sharing Variables](#) 翻译: [nb312](#)校对: [Wiki](#)

资源

资源

- [总览](#)
- [BibTex 引用](#)
- [示例使用](#)
- [FAQ](#)
- [术语表](#)
- [Tensor排名、形状和类型](#)

总览

- [其他资源](#)
 - [TensorFlow 白皮书](#)
 - [引用](#)
 - [社区](#)
 - [讨论](#)
 - [报告 Issues](#)
 - [开发](#)

其他资源

TensorFlow 白皮书

在这份白皮书里，你可以找到关于 TensorFlow 编程模型的更多详情和 TensorFlow 的实现原理。

- [TensorFlow: Large-scale machine learning on heterogeneous systems](#)

引用

如果你在你的研究中使用了 TensorFlow，并且希望在引用中注记 TensorFlow，我们建议你引用上面这篇论文。你可以使用这个 [BibTeX 目录](#)。随着项目的继续发展，我们还会在这个建议引用列表添加新的论文。

社区

讨论

- [GitHub](#)
- [Stack Overflow](#)
- [TensorFlow 讨论邮件列表](#)

报告 Issues

- [TensorFlow issues](#)

开发

- 如果你有兴趣为 TensorFlow 贡献代码请[阅读这篇代码贡献指南](#).

原文: <http://tensorflow.org/resources/bib.md>

翻译: [Jim-Zenn](#)

校对: [lonlonago](#)

BibTex 引用

- BibTex 引用

BibTex 引用

如果你在研究中使用了 TensorFlow，并且希望引用 TensorFlow系统。我们建议你引用一下白皮书。

```
1. @misc{tensorflow2015-whitepaper,  
2. title={{TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems},  
3. url={http://tensorflow.org/},  
4. note={Software available from tensorflow.org},  
5. author={  
6.     Mart'\{i\}n~Abadi and  
7.     Ashish~Agarwal and  
8.     Paul~Barham and  
9.     Eugene~Brevdo and  
10.    Zhifeng~Chen and  
11.    Craig~Citro and  
12.    Greg~S.~Corrado and  
13.    Andy~Davis and  
14.    Jeffrey~Dean and  
15.    Matthieu~Devin and  
16.    Sanjay~Ghemawat and  
17.    Ian~Goodfellow and  
18.    Andrew~Harp and  
19.    Geoffrey~Irving and  
20.    Michael~Isard and  
21.    Yangqing Jia and  
22.    Rafal~Jozefowicz and  
23.    Lukasz~Kaiser and  
24.    Manjunath~Kudlur and  
25.    Josh~Levenberg and  
26.    Dan~Man'\{e\} and  
27.    Rajat~Monga and  
28.    Sherry~Moore and  
29.    Derek~Murray and  
30.    Chris~Olah and  
31.    Mike~Schuster and  
32.    Jonathon~Shlens and  
33.    Benoit~Steiner and  
34.    Ilya~Sutskever and  
35.    Kunal~Talwar and  
36.    Paul~Tucker and
```

```

37. Vincent~Vanhoucke and
38. Vijay~Vasudevan and
39. Fernanda~Vi\{e}gas and
40. Oriol~Vinyals and
41. Pete~Warden and
42. Martin~Wattenberg and
43. Martin~Wicke and
44. Yuan~Yu and
45. Xiaoqiang~Zheng},
46. year={2015},
47. }

```

文本形式如下：

```

1. Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo,
2. Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis,
3. Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
4. Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia,
5. Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster,
6. Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens,
7. Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker,
8. Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas,
9. Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke,
10. Yuan Yu, and Xiaoqiang Zheng.
11. TensorFlow: Large-scale machine learning on heterogeneous systems,
12. 2015. Software available from tensorflow.org.

```

原文: <http://tensorflow.org/resources/bib.md> 翻译: Jim-Zenn 校对: Wiki

示例使用

- [应用实例](#)

应用实例

本页介绍了一些 TensorFlow 系统当前在实际中的应用。

如果您在做研究、教育、或在某些产品中正在使用 *TensorFlow*，我们非常乐意在这里添加一些有关您的使用情况。请随时给我们发电子邮件简要说明您是如何使用*TensorFlow*的，或者给我们发 *pull request*来添加一个条目到本文件。

下面列出了一些 TensorFlow 的用途。

- **RankBrain**

- 组织：Google
- 域名：Information Retrieval
- 描述：对www.google.com搜索排名大规模部署的深层神经网络。
- 更多信息：[“Google Turning Over Its Lucrative Search to AI Machines”](#)

- **Inception Image Classification Model**

- 组织：Google
- 描述：研究高精确的计算机视觉模型，赢得了2014年Imagenet图像分类的挑战（ILSVRC 2014）
- 更多信息：关于 Baseline model 的描述 [Arxiv paper](#)

- **SmartReply**

- 组织：Google
- 描述：基于深度 LSTM 模型的自动生成电子邮件回复
- 更多信息：[Google research blog post](#)

- **Massively Multitask Networks for Drug Discovery**

- 组织：Google and Stanford University
- 域名：Drug discovery
- 描述：基于深度神经网络模型的药物探索
- 更多信息：[Arxiv paper](#)

- **On-Device Computer Vision for OCR**

- 组织：Google

- 描述：用设备内置的计算机视觉模型来做‘光学字符识别’（OCR）以实现实时翻译。
- 更多信息：[Google Research blog post](#)

原文:<http://tensorflow.org/resources/uses.md>

翻 译:[andyiac](#)

校 对:[lonlonago](#)

FAQ

- 常见问题
 - 内容
 - 常见问题
 - 建立 TensorFlow graph
 - 为什么 `c = tf.matmul(a, b)` 不立即执行矩阵相乘？
 - 设备是如何命名的？
 - 如何在指定的设备上运行操作(operations)？
 - 可用的 tensor 有哪些不同的类型？
- " level="2">运行 TensorFlow 计算过程。
 - 请详细解释 feeding 和 placeholders？
 - `Session.run()` 和 `Tensor.eval()` 有什么区别？
 - Sessions 有生存期吗？调用时产生的 tensors 呢？
 - 我可以在多个计算机上运行分布式的训练吗？
 - 运行时会并行计算图的执行的各个部分(parts of graph execution)吗？
 - TensorFlow 支持哪些客户端编程语言？
 - TensorFlow 会利用我计算机上所有可用的设备 (GPUs 和 CPUs) 吗？
 - 当使用一个 reader 或 一个 queue 时，为什么 `Session.run()` 会挂起？
- Variables
 - 变量的生存期是？
 - 并发读取或存入变量时会是什么情况？
- Tensor shapes
 - 在 Python 中我怎么判断一个 tensor 的 shape ？
 - `x.set_shape()` 和 `x = tf.reshape(x)` 有什么区别？
 - 我怎么创建这样一个 graph ，它在批次大小可变(variable batch sizes)的情形下也可以正常运作？
- TensorBoard
 - 我怎样视觉化一个 TensorFlow graph ？
 - 向 TensorBoard 发送数据的最简单的方法是什么？
- 扩展 TensorFlow
 - 我的数据是自定义格式，要怎样用 TensorFlow 来读取它？
 - 我如何定义操作(operation)使得它能够接受可变数量的输入？
- 其他问题
 - TensorFlow 能使用 Python 3 吗？
 - TensorFlow 的代码风格有什么规则？

常见问题

此文档对关于TensorFlow的一些常见问题提供了答案，如果这里没有你问题的答案，你可能会在[社](#)

[区资源](#)中找到它。

内容

常见问题

- [建立 TensorFlow graph](#)
- [运行 TensorFlow 计算过程](#)
- [Variables](#)
- [Tensor shapes](#)
- [TensorBoard](#)
- [扩展 TensorFlow](#)
- [其他问题](#)

建立 TensorFlow graph

参看

[建立 graph 的 API 文档](#).

为什么 `c = tf.matmul(a, b)` 不立即执行矩阵相乘？

在 TensorFlow 的 Python API 中, `a`, `b`, and `c` 都是

`Tensor` 对象. 一个 `Tensor` 对象是一个操作(operation)结果的字符别名, 它实际上并不储存操作(operation)输出结果的值。

TensorFlow 鼓励用户去建立复杂的表达式 (如整个神经网络及其梯度) 来形成 data flow graph。

然后你可以将整个 data flow graph 的计算过程交给一个 TensorFlow 的 `Session`, 此 `Session` 可以运行整个计算过程, 比起操作(operations)一条一条的执行效率高的多。

设备是如何命名的？

对CPU设备而言, 支持的设备名是 `"/device:CPU:0"` (或 `"/cup:0"`), 对第 i 个 GPU 设备是 `"/device:GPU:i"` (或 `"/gpu:i"`)

如何在指定的设备上运行操作(operations)？

在 `with tf.device(name):` context 中创建操作(operation), 这样可以在指定的设备上运行操作(operation)。

关于 TensorFlow 怎样将操作(operations)分配给设备的细节, 参看 [TensorFlow使用 GPU](#); 使用多 GPU 的示范实例参看 [CIFAR-10 教程](#)。

可用的 tensor 有哪些不同的类型？

TensorFlow 支持许多不同的数据类型和 tensor shape，更多细节请参看 [ranks, shapes, and type reference](#)

" class="reference-link">运行 TensorFlow 计算过程。

参看

[运行 graph 的 API 文档](#).

请详细解释 feeding 和 placeholders？

Feeding 是 TensorFlow Session API 的一种机制，它允许你在运行时用不同的值替换一个或多个 tensor 的值。

`Session.run()` 的参数 `feed_dict` 是一个字典，它将 `Tensor` 对象映射为 numpy 的数组（和一些其他类型）。在执行 step 时，这些数组就是 tensor 的值。

你常会碰到某些 tensor 总是有值的，比如 inputs。`tf.placeholder()` 操作(operation)允许你定义一种必须提供值的 tensor，你也可以随意限定它们的 shape。关于如何使用 placeholders 和 feeding 为神经网络提供训练数据的例子，请参看[初学者的 MNIST 教程](#)

`Session.run()` 和 `Tensor.eval()` 有什么区别？

如果 `t` 是一个 `Tensor` 对象，`t.eval()` 就是 `sess.run(t)`（`sess` 是当前默认 `session`）的简写。

以下两段小程序是等效的：

```
1. # 使用 `Session.run()``.
2. sess = tf.Session()
3. c = tf.constant(5.0)
4. print sess.run(c)
5.
6. # 使用 `Tensor.eval()``.
7. c = tf.constant(5.0)
8. with tf.Session():
9.     print c.eval()
```

在第二个例子中，`session` 的作用就象 `context manager`，`context manager` 在 `with` 块的生存期，将 `session` 作为默认的 `session`。对简单应用的情形（如单元测试），`context manager` 的方法可以得到更简洁的代码；如果你的代码要处理多个 graph 和

`session`，更直白的方式可能是显式调用 `Session.run()`。

Sessions 有生存期吗？调用时产生的 tensors 呢？

`Session` 能够占有资源，例如 `variables`，`queues`，和 `readers`；这些资源会使用相当大量的内存。当调用 `Session.close()` 关闭 `session` 后，这些资源（和相关的内存）就被释放了。

作为调用 `Session.run()` 过程的一部分所创建的 `tensors`，会在调用时或调用结束前释放。

我可以在多个计算机上运行分布式的训练吗？

最初的 TensorFlow 开源版本支持单一计算机内的多设备（CPUs 和 GPUs）。我们也在致力于一个分布式的版本：如果你有兴趣，请告知我们，这样我们可以做相应的调整。

运行时会并行计算图的执行的各个部分(parts of graph execution)吗？

TensorFlow 运行时会在许多不同的层面(dimensions)并行图的执行(graph execution)：

- 在一个CPU中用多核或是一个GPU中用多线程来并行许多单独的操作(operation)。
- 在 TensorFlow graph 中各个独立的节点可以在多个设备上并行，这样就提供了加速的可能。[CIFAR-10 用多 GPU 训练](#)。
- `Session` API 允许并行执行多并发的 steps（如调用 `Session.run()`）。如果单一的 step 不使用你计算机中所有的资源，这种方法可以使运行时有更高的吞吐量。

TensorFlow 支持哪些客户端编程语言？

TensorFlow 被设计成为支持多种客户端语言。当前支持最好的客户端语言是 `Python`。`C++ 客户端 API` 提供了启动 graph 和运行 steps 的接口；我们还有一个 [用 C++ 建立 graph 的 API](#)，此 API 是实验性的。

从社区的利益出发，我们想要支持更多的客户端语言。TensorFlow 有一个 [基于 C 的客户端 API](#)，它使得用许多不同的语言创建客户端变得很容易。我们请大家在新语言绑定上做出努力。

TensorFlow 会利用我计算机上所有可用的设备（GPUs 和 CPUs）吗？

TensorFlow 支持多 GPU 和 CPU。有关 TensorFlow 如何将操作(operations)分配到设备的细节请参看 [TensorFlow 如何使用 GPUs](#) 的文档，有关使用多 GPU 的示范实例请参看 [CIFAR-10 教程](#)

请注意，TensorFlow 只使用计算能力(compute capability)大于 3.5 的 GPU 设备。

当使用一个 reader 或 一个 queue 时，为什么 `Session.run()` 会挂起？

`reader` 类和 `queue` 类提供特殊的操作(operations)，这些操作(operations)在有可用的输入(对有界队列则是空闲空间)前会阻塞。使用这些操作(operations)你可以创建复杂的输入流水线(input pipelines)，不过，这会使 TensorFlow 的计算过程更复杂。有关如何使用这些操作(operations)的更多信息请参看 how-to 文档中的使用 `QueueRunner` 对象来控制 queues 和 readers。

Variables

参看 [Variables](#)

和 [变量作用域](#) 的 how-to 文档，还有[关于变量的 API 文档](#)

变量的生存期是？

在某一 session 中，当你一开始运行 `tf.Variable.initializer` 操作(operation)时，变量就会被创建。此 `session` 关闭后 它就被摧毁(destroyed)了。

并发读取或存入变量时会是什么情况？

变量可以进行并发的读和写操作(operation)。由于变量是并发(concurrently)更新的，所以从一个变量中读出的值可能会改变。在不互斥的条件下，对一个变量的并发的许多赋值操作(operation)是默认允许运行的。在对一个变量赋值时，如果想要加锁，可以将

`use_locking=True` 传递给 `Variable.assign()`。

Tensor shapes

参看

`TensorShape` [API 文档](#)。

在 Python 中我怎么判断一个 tensor 的 shape ？

在 TensorFlow 中，一个 tensor 具备静态和动态两种 shape。静态的 shape 可以用 `tf.Tensor.get_shape()` 方法读出：这种 shape 是由此 tensor 在创建时使用的操作(operations)推导得出的，可能是 `partially complete` 的。如果静态 shape 没有完整定义(not fully defined)的话，则一个 tensor 的动态 shape 可通过求 `tf.shape(t)` 的值得到。

`x.set_shape()` 和 `x = tf.reshape(x)` 有什么区别？

`tf.Tensor.set_shape()` 方法(method)会更新(updates)一个 `Tensor` 对象的静态 `shape` , 当静态 `shape` 信息不能够直接推导得出的时候, 此方法常用来提供额外的 `shape` 信息。它不改变此 `tensor` 动态 `shape` 的信息。

`tf.reshape()` 操作(operation)会以不同的动态 `shape` 创建一个新的 `tensor`。

我怎么创建这样一个 graph , 它在批次大小可变(variable batch sizes)的情形下也可以正常运作 ?

如果能够创建一个 graph , 在批次大小可变(variable batch sizes)的情形下也可以正常运作将会是十分有用的, 例如可以使用相同的代码完成 (小) 批量训练((mini-)batch training)和单例推导(single-instance inference)。这样生成的 graph 可以[保存起来当作协议缓存\(protocol buffer\)](#), 也可以[导入至其他的程序](#)。

创建一个可变大小的 graph 时, 要记住最重要的事情是不要将批次大小(batch size)编码成为 Python 常数, 而是用一个字符性(symbolic)的 `Tensor` 来表示。下面的提示可能会有用:

- 用 `batch_size = tf.shape(input)[0]` 从一个叫 `input` 的 `Tensor` 提取批次的维度(batch dimation), 再将其存入一个名为 `batch_size` 的 `Tensor` 。
- 用 `tf.reduce_mean()` 而不是 `tf.reduce_sum(...) / batch_size` 。
- 如果你使用 [placeholders for feeding input](#), 你就可以用 `tf.placeholder(..., shape=[None, ...])` 通过创建 placeholder 来具体指定一个可变的批次维度(variable batch dimation)。shape 的 `None` 元素与可变大小的维度(a variable-sized dimension)相对应。

TensorBoard

我怎样视觉化一个 TensorFlow graph ?

参看 [graph 的视觉化教程](#)。

向 TensorBoard 发送数据的最简单的方法是什么 ?

给你的 TensorFlow graph 增加 summary 操作(ops), 接着用 `SummaryWriter` 将这些 summaries 写入一个 log directory。然后用以下命令启动 TensorBoard 。

```
1. `python tensorflow/tensorboard/tensorboard.py --logdir=path/to/log-directory`
```

更多细节请参看 [Summaries](#) 和 [TensorBoard 教程](#)。

扩展 TensorFlow

参看有关[向 TensorFlow 添加新操作\(operations\)](#) 的 how-to 文档。

我的数据是自定义格式，要怎样用 TensorFlow 来读取它？

有两种主要的操作(operation)来处理自定义格式的数据。

较简单的方法：用 Python 编写一段分词的代码(parsing code)，将数据转换成为 numpy array，然后用此数据把一个 [`tf.placeholder()`] ([../api_docs/python/io_ops.md#placeholder](#)) 传送给一个 tensor 。更多的细节参见[使用 placeholders 进行输入](#) 的相关文档。这个方法比较容易实现，不过分词的部分会成为性能的瓶颈。

更高效的方法是添加一个用 C++ 编写的操作(op)，用这个操作(operation)来对你的数据格式进行分词(parse)。

[新数据格式处理指南](#)中有更多相关步骤的信息。

我如何定义操作(operation)使得它能够接受可变数量的输入？

TensorFlow 的操作(operation)注册机制允许你定义几种输入：单独的 tensor，一系列相同类型的 tensors (例如把一个可变长列表中的 tensors 相加)， 一系列不同类型的 tensors (例如将一个 tuple 中的 tensors 入队(enqueue))。有关怎样定义这些不同的输入类型的更多细节，请参看[添加具有一列输入或输出的操作\(op\)](#)的相关文档。

其他问题

TensorFlow 能使用 Python 3 吗？

我们只用 Python 2.7 进行了测试。我们了解对 Python 3 的兼容性来说，还需要有一些修改，欢迎大家朝这个方向多努力。

TensorFlow 的代码风格有什么规则？

TensorFlow Python API 遵循 [PEP8](#) 惯例。

* 特别的，我们使用 `CamelCase` 格式作为类名， `snake_case` 格式作为方程名， 方法名， 和 属性名。我们也遵循 [Google Python style guide](#)。

TensorFlow C++ 代码遵循 [Google C++ style guide](#)。

(* 有一条例外：我们使用 2 空格缩进而不是 4 空格缩进)

原文: [Frequently Asked Questions](#) 翻译: [Terence Cooper](#) 校对: [Wiki](#)

术语表

- 术语表
 - 广播操作(Broadcasting operation)
 - Devices
 - eval
 - Feed
 - Fetch
 - Graph(图)
 - IndexedSlices (索引化切片)
 - Node (节点)
 - 操作 (Op/operation)
 - Run
 - Session(会话)
 - Shape
 - SparseTensor
 - Tensor

术语表

广播操作(Broadcasting operation)

一种用[numpy-style broadcasting](#)来保证tensor参数的形态兼容的操作。

Devices

一块可以用来运算并且拥有自己的地址空间的硬件，比如GPU和CPU。

eval

Tensor 的一个方法，返回 Tensor 的值。触发任意一个图计算都需要计算出这个值。只能在一个已经启动的会话的图中才能调用该 Tensor 值。

Feed

TensorFlow 的一个概念：把一个 Tensor 直接连接到一个会话图表中的任意节点。feed 不是在构建图(graph)的时候创建，而是在触发图的执行操作时去申请。一个 feed 临时替代一个带有 Tensor 值的节点。把feed数据作为run()方法和eval()方法的参数来初始化运算。方法运行结束后，替换的 feed 就会消失，而最初的节点定义仍然还在。可以通过tf.placeholder()把特定

的节点指定为 `feed` 节点来创建它们。详见[Basic Usage](#)。

Fetch

TensorFlow中的一个概念：为了取回运算操作的输出结果。取回的申请发生在触发执行图操作的时候，而不是发生在建立图的时候。如果要取回一个或多个节点（node）的 Tensor 值，可以通过在 Session 对象上调用`run()`方法并将待取回节点（node）的列表作为参数来执行图表(graph)。详见[Basic Usage](#)。

Graph(图)

把运算任务描述成一个直接的无环图形（DAG），图表中的节点（node）代表必须要实现的一些操作。图中的边代表数据或者可控的依赖。GratheDef 是系统中描述一个图表的协议(api)，它由一个 NodeDefs 集合组成。一个GraphDef可以转化成一个更容易操作的图表对象。

IndexedSlices（索引化切片）

在 Python API 中，TensorFlow 仅仅在第一维上对 Tensor 有所体现。如果一个 Tensor 有 k维，那么一个 IndexedSlices 实例在逻辑上代表一个沿着这个 Tensor 第一维的(k-1)维切片的集合。切片的索引被连续储存在一个单独的一维向量中，而对应的切片则被拼接成一个单独的k维 Tensor。如果 sparsity 不是受限于一维空间，请用 SparseTensor。

Node（节点）

图中的一个元素。

把启动一个特定操作的方式称为特定运算图表的一个节点，包括任何用来配置这个操作的属性的值。对于那些多形态的操作，这些属性包括能完全决定这个节点（Node）签名的充分信息。详见 `graph.proto`。

操作（Op/operation）

在 TensorFlow 的运行时中，它是一种类似 `add` 或 `matmul` 或 `concat`的运算。可以用[how to add an op](#)中的方法来向运行时添加新的操作。

在 Python 的API中，它是图中的一个节点。在[tf.Operation](#)类中列举出了这些操作。一个操作 (Operation)的 `type` 属性决定这个节点（node）的操作类型，比如`add`和`matmul`。

Run

在一个运行的图中执行某种操作的行为。要求图必须运行在会话中。

在 Python 的 API 中，它是 `Session` 类的一个方法 `tf.Session.run`。可以通过 `Tensors` 来订阅或获取 `run()` 操作。

在 C++ 的 API 中，它是 `tensorflow::Session` 类的一个方法。

Session(会话)

启动图的第一步是创建一个 `Session` 对象。`Session` 提供在图中执行操作的一些方法。

在 Python API 中，使用 `tf.Session`。

在 C++ 的 API 中，`tensorflow::Session` 是用来创建一个图并运行操作的类：

Shape

`Tensor` 的维度和它们的大小。

在一个已经启动的图中，它表示流动在节点 (node) 之间的 `Tensor` 的属性。一些操作对 `shape` 有比较强的要求，如果没有 `Shape` 属性则会报告错误。

在 Python API 中，用创建图的 API 来说明 `Tensor` 的 `Shape` 属性。`Tensor` 的 `Shape` 属性要么只有部分已知，要么全部未知。详见 `tf.TensorShape`

在 C++ 中，`Shape` 类用来表示 `Tensor` 的维度。`tensorflow::TensorShape`。

SparseTensor

在 Python API 中，它用来表示在 `TensorFlow` 中稀疏散落在任意地方的 `Tensor`。
`SparseTensor` 以字典-值格式来储存那些沿着索引的非空值。换言之， m 个非空值，就包含一个长度为 m 的值向量和一个由 m 列索引 (indices) 组成的矩阵。为了提升效率，`SparseTensor` 需要将 `indices` (索引) 按维度的增加来按序存储，比如行主序。如果稀疏值仅沿着第一维度，就用 `IndexedSlices`。

Tensor

`Tensor` 是一种特定的多维数组。比如，一个浮点型的四维数组表示一小批由 `[batch, height, width, channel]` 组成的图片。

在一个运行的图 (graph) 中，它是一种流动在节点 (node) 之间的数据。

在 Python 中，`Tensor` 类表示添加到图的操作中的输入和输出，见 `tf.Tensor`，这样的类不持有数据。

在 C++ 中，`Tensor` 是方法 `Session::Run()` 的返回值，见 `tensorflow::Tensor`，这样的

Tensor 持有数据。

原文: [Glossary](#)

翻译: [leege100](#)

校对: [lonlonago](#)

Tensor排名、形状和类型

- 张量的阶、形状、数据类型
 - 阶
 - 形状
 - 数据类型

张量的阶、形状、数据类型

TensorFlow用张量这种数据结构来表示所有的数据. 你可以把一个张量想象成一个n维的数组或列表. 一个张量有一个静态类型和动态类型的维数. 张量可以在图中的节点之间流通.

阶

在TensorFlow系统中, 张量的维数来被描述为阶. 但是张量的阶和矩阵的阶并不是同一个概念. 张量的阶 (有时是关于如顺序或度数或者是 n 维) 是张量维数的一个数量描述. 比如, 下面的张量 (使用Python中list定义的) 就是2阶.

```
1.      t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

你可以认为一个二阶张量就是我们平常所说的矩阵, 一阶张量可以认为是一个向量. 对于一个二阶张量你可以用语句 `t[i, j]` 来访问其中的任何元素. 而对于三阶张量你可以用 `t[i, j, k]` 来访问其中的任何元素.

阶	数学实例	Python 例子
0	纯量 (只有大小)	<code>s = 483</code>
1	向量 (大小和方向)	<code>v = [1.1, 2.2, 3.3]</code>
2	矩阵 (数据表)	<code>m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]</code>
3	3阶张量 (数据立体)	<code>t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]</code>
n	n阶 (自己想想看)	<code>....</code>

形状

TensorFlow文档中使用了三种记号来方便地描述张量的维度: 阶, 形状以及维数. 下表展示了他们之间的关系:

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] 。

形状可以通过Python中的整数列表或元祖 (`int list`或`tuples`) 来表示, 也或者用 `TensorShape` [class](#) 。

数据类型

除了维度, `Tensors`有一个数据类型属性. 你可以为一个张量指定下列数据类型中的任意一个类型:

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

原文: [Tensor Ranks, Shapes, and Types](#)

翻译: [nb312](#)

校对: [lonlonago](#)

其他

其他

- [常见问题汇总](#)
- [相关资源](#)
- [个人学习心得](#)

常见问题汇总

- 常见问题汇总
 - 说明
 - 常见问题及解答
 - (示例)官网地址是哪里？
 - 如何安装 pip ？
 - `docker run -it b.gcr.io/tensorflow/tensorflow` 失败
 - `SSLError: SSL_VERIFY_FAILED`

常见问题汇总

说明

本章非官方文档翻译，是由众多TensorFlow爱好者将安装和使用TF过程中的问题总结而成的。

常见问题及解答

(示例)官网地址是哪里？

<http://www.tensorflow.org/>

如何安装 pip ？

- Ubuntu (14.04)

```
1. sudo apt-get update && sudo apt-get install -y python python-dev python-pip
```

- CentOS 7

```
1. yum update -y && yum install -y python python-devel epel-release.noarch python-pip
```

- MACOS

```
1. sudo easy_install pip
```

`docker run -it b.gcr.io/tensorflow/tensorflow` 失败

该镜像所在仓库被墙，需要梯子。

[这里](#)（密码：v9ts）有镜像的导出包。

使用方法

```
1. docker load < tensorflow.tar.gz
```

SSLError: SSL_VERIFY_FAILED

在用url通过pip的方式安装的时候，可能会遇到的错误如下：

```
1.      SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:590)
```

解决办法：先下载wheel到本地，之后本地pip install

相关资源

- [TensorFlow 相关资源](#)

TensorFlow 相关资源

- [Google官方Blog宣布TensorFlow开源](#)
- [TensorFlow WhitePaper\(PDF下载\)](#)
- [Jeff Dean 介绍 TensorFlow \(视频\)](#)
- [TensorFlow 简化版接口 Scikit Flow](#)
- [TensorFlow 使用样例](#)
- [TensorFlow 与 mxnet, caffe 对比 @chenrudan](#)
- [TensorFlow 与 torch7, caffe, mxnet 在内存使用量以及性能对比 @mli](#)

个人学习心得

- [TensorFlow 个人学习心得](#)

TensorFlow 个人学习心得

说明：本章所列链接为个人学习TensorFlow的心得，很多是博客地址，从实践的角度帮助大家更好的理解官方文档的内容

示例：

- [我的TensorFlow学习博客](#)