TensorFlow 指南

2015 年 12 月 30 日

目录

第一章　起步

1.1 简介

本章的目的是让你了解和运行 TensorFlow!

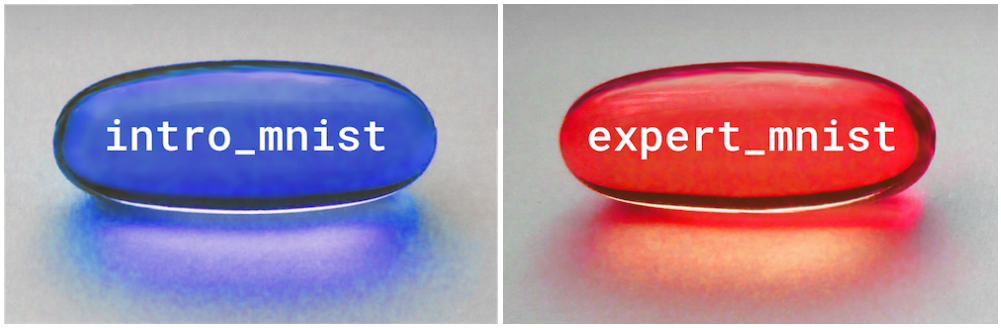
在开始之前, 让我们先看一段使用 Python API 撰写的 TensorFlow 示例代码, 让你对将要学习的内容有初步的印象.

下面这段短小的 Python 程序将把一些数据放入二维空间，再用一条线来拟合这些数据。

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype("float32")
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but Tensorflow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this
    first.
init = tf.initialize_all_variables()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in xrange(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]
```

以上代码的第一部分构建了数据的流向图 (flow graph)。在一个 session 被建立并且`run()`函数被运行前，TensorFlow 不会进行任何实质的计算。

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。为此，我们准备了两篇不同的教程, 分别面向初学者和专家。如果你已经使用其它软件训练过许多 MNIST 模型, 请参阅高级教程 (红色药丸)。如果你以前从未听说过 MNIST, 请先阅读初级教程 (蓝色药丸)。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。



如果你已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[1]。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。

[1]**推荐随后阅读内容：** 1 下载与安装, 2 基本使用, 3 TensorFlow 101.

1.2 使用基础

使用 TensorFlow 之前你需要了解关于 TensorFlow 的以下基础知识:

- 使用图(graphs)来表示计算.
- 在会话(Session)中执行图.
- 使用张量(tensors)来代表数据.
- 通过变量(Variables)维护状态.
- 使用 feeds 和 fetches 将数据传入或传出 arbitrary operations.

1.2.1 Overview

TensorFlow is a programming system in which you represent computations as graphs. Nodes in the graph are called ops (short for operations). An op takes zero or more Tensors, performs some computation, and produces zero or more Tensors. A Tensor is a typed multi-dimensional array. For example, you can represent a mini-batch of images as a 4-D array of floating point numbers with dimensions [batch, height, width, channels].

TensorFlow 是一个以图(graphs)来表示计算的编程系统, 图中的节点被称之为 op (operation 的缩写). 一个 op 获得零或多个张量(tensors)执行计算, 产生零或多个张量(tensors)。张量是一个按类型划分的多维数组。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]。

A TensorFlow graph is a description of computations. To compute anything, a graph must be launched in a Session. A Session places the graph ops onto Devices, such as CPUs or GPUs, and provides methods to execute them. These methods return tensors produced by ops as numpy ndarray objects in Python, and as tensorflow::Tensor instances in C and C++.

TensorFlow 的图描述了计算的流程。在计算开始前, 图必须在会话()Session中被启动.会话()Session将图的 op 分发到如 CPU 或 GPU 之类的设备(Devices)上, 同时提供执行 op 的方法。这些方法执行后, 将产生的张量 (tensor) 返回。在 Python 语言中, 返回numpy的ndarray 对象; 在 C 和 C++ 语言中, 返回tensorflow::Tensor实例。

1.2.2 图的构建

TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph.

For example, it is common to create a graph to represent and train a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.

TensorFlow can be used from C, C++, and Python programs. It is presently much easier to use the Python library to assemble graphs, as it provides a large set of helper functions not available in the C and C++ libraries.

The session libraries have equivalent functionalities for the three languages.

通常，TensorFlow 编程可按两个阶段组织起来: **构建阶段**和**执行阶段**。构建阶段用于组织计算用的图，而执行阶段利用 session 中执行 op 操作来计算图。

例如, 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一组 op 来实现图中的训练。

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, 而这些函数在 C 和 C++ 库中尚不被支持。

这三种语言的会话库 (session libraries) 是一致的.

构建图

To build a graph start with ops that do not need any input (source ops), such as Constant, and pass their output to other ops that do computation.

The ops constructors in the Python library return objects that stand for the output of the constructed ops. You can pass these to other ops constructors to use as inputs.

The TensorFlow Python library has a default graph to which ops constructors add nodes. The default graph is sufficient for many applications. See the Graph class documentation for how to explicitly manage multiple graphs.

刚开始基于 op 建立图的时候一般不需要任何的输入源 (source op)，例如输入常量 (`Constance`)，再将它们传递给其它 op 执行运算。

Python 库中的 op 构造函数返回代表已被组织好的 op 作为输出对象，这些对象可以传递给其它 op 构造函数作为输入。

TensorFlow Python 库有一个可被 op 构造函数加入计算结点的默认图 (default graph)。对大多数应用来说，这个默认图已经足够用了。阅读 Graph 类文档来了解如何明晰的管理多个图.

```
import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])
```

```
# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)
```

The default graph now has three nodes: two constant() ops and one matmul() op. To actually multiply the matrices, and get the result of the multiplication, you must launch the graph in a session.

默认图现在拥有三个节点，两个`constant()` op，一个`matmul()` op. 为了真正进行矩阵乘法运算，得到乘法结果, 你必须在一个会话 (session) 中载入动这个图。

在会话 (session) 中载入图 (graph)

Launching follows construction. To launch a graph, create a Session object. Without arguments the session constructor launches the default graph.

See the Session class for the complete session API.

构建过程完成后就可运行执行过程。为了载入之前所构建的图，必须先创建一个会话对象 (`Session` object)。会话构建器在未指明参数时会载入默认的图。

完整的会话 API 资料，请参见**会话类 (Session object)**。

```
# Launch the default graph.
sess = tf.Session()

# To run the matmul op we call the session 'run()' method, passing '
    product'
# which represents the output of the matmul op.  This indicates to
    the call
# that we want to get the output of the matmul op back.
#
# All inputs needed by the op are run automatically by the session.
    They
# typically are run in parallel.
#
# The call 'run(product)' thus causes the execution of threes ops in
    the
# graph: the two constants and matmul.
#
# The output of the op is returned in 'result' as a numpy `ndarray`
    object.
result = sess.run(product)
print(result)
# ==> [[ 12.]]

# Close the Session when we're done.
sess.close()
```

Sessions should be closed to release resources. You can also enter a Session with a "with" block. The Session closes automatically at the end of the with block.

会话在完成后必须关闭以释放资源。你也可以使用`"with"`句块开始一个会话，该会话将在`"with"`句块结束时自动关闭。

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

```
  print(result)
```

The TensorFlow implementation translates the graph definition into executable operations distributed across available compute resources, such as the CPU or one of your computer's GPU cards. In general you do not have to specify CPUs or GPUs explicitly. TensorFlow uses your first GPU, if you have one, for as many operations as possible.

TensorFlow 事实上通过一个“翻译”过程，将定义的图转化为不同的可用计算资源间实现分布计算的操作，如 CPU 或是显卡 GPU。通常不需要用户指定具体使用的 CPU 或 GPU，TensorFlow 能自动检测并尽可能的充分利用找到的第一个 GPU 进行运算。

If you have more than one GPU available on your machine, to use a GPU beyond the first you must assign ops to it explicitly. Use with...Device statements to specify which CPU or GPU to use for operations:

如果你的设备上有不止一个 GPU，你需要明确指定 op 操作到不同的运算设备以调用它们。使用`with...Device`语句明确指定哪个 CPU 或 GPU 将被调用。

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

Devices are specified with strings. The currently supported devices are:

"/cpu:0": The CPU of your machine.

"/gpu:0": The GPU of your machine, if you have one.

"/gpu:1": The second GPU of your machine, etc.

See Using GPUs for more information about GPUs and TensorFlow.

使用字符串指定设备，目前支持的设备包括:

`"/cpu:0"`：计算机的 CPU；

`"/gpu:0"`：计算机的第一个 GPU，如果可用；

`"/gpu:1"`：计算机的第二个 GPU，以此类推。

关于使用 GPU 的更多信息，请参阅 **GPU 使用**。

1.2.3 交互式使用

The Python examples in the documentation launch the graph with a `Session` and use the `Session.run()` method to execute operations.

For ease of use in interactive Python environments, such as `IPython` you can instead use the `InteractiveSession` class, and the `Tensor.eval()` and `Operation.run()` methods. This avoids having to keep a variable holding the session.

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。

为了方便使用如交互式 Python 环境, 可以使用 `InteractiveSession`(../api_docs/python/client.mdInteractiv 代替`Session`类, 使用`Tensor.eval()`(../api_docs/python/framework.mdTensor.eval) 和 `Operation.run()`(../api_docs/python/framework.mdOperation.run) 方法代替`Session.run()`. 这样可以避免使用一个变量来持有会话.

```
# Enter an interactive TensorFlow Session.
import tensorflow as tf
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# Initialize 'x' using the run() method of its initializer op.
x.initializer.run()

# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = tf.sub(x, a)
print(sub.eval())
# ==> [-2. -1.]

# Close the Session when we're done.
sess.close()
```

1.2.4 张量 (Tensors)

TensorFlow programs use a tensor data structure to represent all data – only tensors are passed between operations in the computation graph. You can think of a TensorFlow tensor as an n-dimensional array or list. A tensor has a static type, a rank, and a shape. To learn more about how TensorFlow handles these concepts, see the Rank, Shape, and Type reference.

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](../resources/dims_types.md).

1.2.5 变量

Variables maintain state across executions of the graph. The following example shows a variable serving as a simple counter. See Variables for more details.

变量 (../how_tos/variables/index.md) 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,更多细节详见 [变量](tensorflow-zh/how_tos/variables.md)。

```
# Create a Variable, that will be initialized to the scalar value 0.
state = tf.Variable(0, name="counter")

# Create an Op to add one to `state`.

one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

```

```
# Variables must be initialized by running an `init` Op after having
# launched the graph.  We first have to add the `init` Op to the
    graph.
init_op = tf.initialize_all_variables()

# Launch the graph and run the ops.
with tf.Session() as sess:
  # Run the 'init' op
  sess.run(init_op)
  # Print the initial value of 'state'
  print(sess.run(state))
  # Run the op that updates 'state' and print 'state'.
  for _ in range(3):
    sess.run(update)
    print(sess.run(state))

# output:

# 0
# 1
# 2
# 3
```

The `assign()` operation in this code is a part of the expression graph just like the `add()` operation, so it does not actually perform the assignment until `run()` executes the expression.

You typically represent the parameters of a statistical model as a set of Variables. For example, you would store the weights for a neural network as a tensor in a Variable. During training you update this tensor by running a training graph repeatedly.

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.

1.2.6 Fetches

To fetch the outputs of operations, execute the graph with a `run()` call on the `Session` object and pass in the tensors to retrieve. In the previous example we fetched the single node `state`, but you can also fetch multiple tensors:

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

```
input1 = tf.constant(3.0)
input2 = tf.constant(2.0)
input3 = tf.constant(5.0)
intermed = tf.add(input2, input3)
mul = tf.mul(input1, intermed)

with tf.Session() as sess:
  result = sess.run([mul, intermed])
```

```
  print(result)

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

All the ops needed to produce the values of the requested tensors are run once (not once per requested tensor).

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）。

1.2.7 Feeds

The examples above introduce tensors into the computation graph by storing them in `Constants` and `Variables`. TensorFlow also provides a feed mechanism for patching a tensor directly into any operation in the graph.

A feed temporarily replaces the output of an operation with a tensor value. You supply feed data as an argument to a `run()` call. The feed is only used for the run call to which it is passed. The most common use case involves designating specific operations to be "feed" operations by using `tf.placeholder()` to create them:

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

feed 使用一个 tensor 值临时替换一个操作的输出结果. 你可以提供 feed 数据作为 `run()` 调用的参数.feed 只在调用它的方法内有效, 方法结束, feed 就会消失. 最常见的用例是将某些特殊的操作指定为"feed" 操作, 标记的方法是使用`tf.placeholder()`为这些操作创建占位符.

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)

with tf.Session() as sess:
  print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

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

A `placeholder()` operation generates an error if you do not supply a feed for it. See the MNIST fully-connected feed tutorial (source code) for a larger-scale example of feeds.

如果没有正确提供 feed, `placeholder()` 操作将会产生一个错误提示. 关于 feed 的规模更大的案例，参见MNIST 全连通 feed 教程以及其源代码。

原文:[Basic Usage](http://tensorflow.org/get_started/basic_usage.md)[@doc001](https://github.com/PFZheng)[@yangtze](https://github.com/sstruct)

第二章　基础教程

综述

MNIST For ML Beginners

If you're new to machine learning, we recommend starting here. You'll learn about a classic problem, handwritten digit classification (MNIST), and get a gentle introduction to multiclass classification.

View Tutorial

Deep MNIST for Experts

If you're already familiar with other deep learning software packages, and are already familiar with MNIST, this tutorial with give you a very brief primer on TensorFlow.

View Tutorial

TensorFlow Mechanics 101

This is a technical tutorial, where we walk you through the details of using TensorFlow infrastructure to train models at scale. We use again MNIST as the example.

View Tutorial

Convolutional Neural Networks

An introduction to convolutional neural networks using the CIFAR-10 data set. Convolutional neural nets are particularly tailored to images, since they exploit translation invariance to yield more compact and effective representations of visual content.

View Tutorial

Vector Representations of Words

This tutorial motivates why it is useful to learn to represent words as vectors (called word embeddings). It introduces the word2vec model as an efficient method for learning embeddings. It also covers the high-level details behind noise-contrastive training methods (the biggest recent advance in training embeddings).

View Tutorial

Recurrent Neural Networks

An introduction to RNNs, wherein we train an LSTM network to predict the next word in an English sentence. (A task sometimes called language modeling.)

View Tutorial

Sequence-to-Sequence Models

A follow on to the RNN tutorial, where we assemble a sequence-to-sequence model for machine translation. You will learn to build your own English-to-French translator, entirely machine learned, end-to-end.

View Tutorial

Mandelbrot Set

TensorFlow can be used for computation that has nothing to do with machine learning. Here's a naive implementation of Mandelbrot set visualization.

View Tutorial

Partial Differential Equations

As another example of non-machine learning computation, we offer an example of a naive PDE simulation of raindrops landing on a pond.

View Tutorial

MNIST Data Download

Details about downloading the MNIST handwritten digits data set. Exciting stuff.

View Tutorial

Image Recognition

How to run object recognition using a convolutional neural network trained on ImageNet Challenge data and label set.

View Tutorial

We will soon be releasing code for training a state-of-the-art Inception model.

Deep Dream Visual Hallucinations

Building on the Inception recognition model, we will release a TensorFlow version of the Deep Dream neural network visual hallucination software.

COMING SOON

2.1 MNIST 之机器学习入门

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。

2.1.1 MNIST 数据集

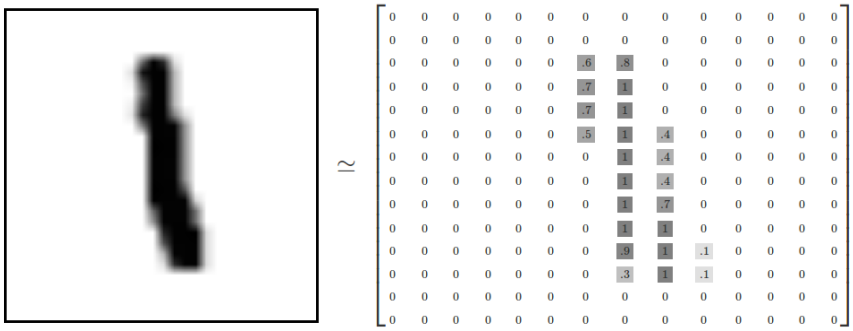
MNIST 数据集的官网是Yann LeCun's website。在这里，我们提供了一份 python 源代码用于自动下载和安装这个数据集。你可以下载这段代码，然后用下面的代码导入到你的项目里面，也可以直接复制粘贴到你的代码文件里面。

```
import input_data
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`。

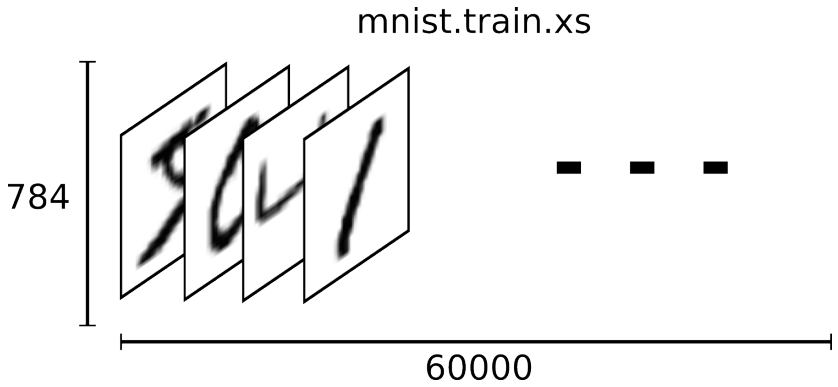
每一张图片包含 28×28 像素。我们可以用一个数字数组来表示这张图片：



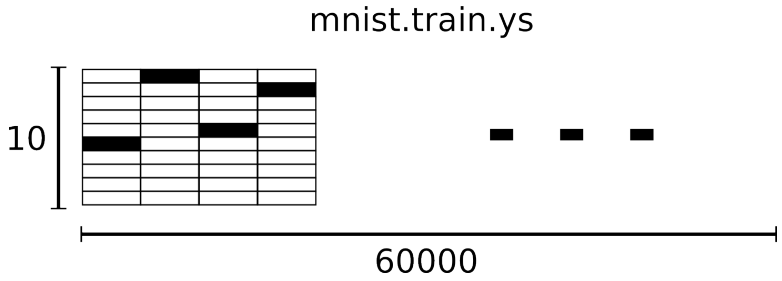
我们把这个数组展开成一个向量，长度是 $28 \times 28 = 784$。如何展开这个数组（数字间的顺序）不重要，只要保持各个图片采用相同的方式展开。从这个角度来看，MNIST 数据集的图片就是在 784 维向量空间里面的点, 并且拥有比较复杂的结构 (提醒: 此类数据的可视化是计算密集型的)。

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])。因此，`mnist.train.labels`是一个 [60000, 10] 的数字矩阵。



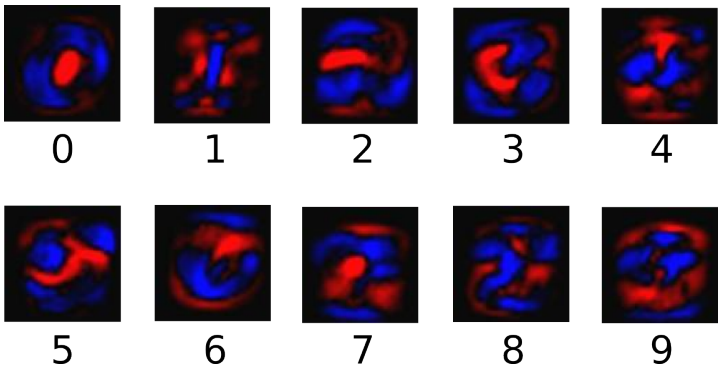
现在，我们准备开始真正的建模之旅啦！

2.1.2 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：首先，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 x 的证据可以表示为

$$evidence_i = \sum_j W_{i,j} x_j + b_i \tag{2.1}$$

其中，W_i 代表权重，b_i 代表第 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:

$$y = softmax(evidence) \tag{2.2}$$

这里的 softmax 可以看成是一个激励（activation）函数或是链接（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：

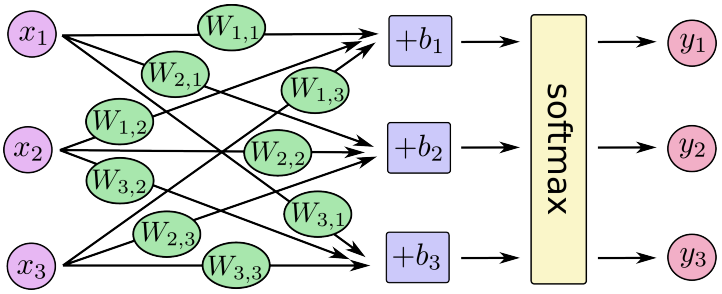
$$softmax(x) = normalize(exp(x)) \tag{2.3}$$

展开等式右边的子式，可以得到：

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.4}$$

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 xs 加权求和，再分别加上一个偏置量，最后再输入到 softmax 函数中：



如果把它写成一个方程，可以得到：

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

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：

$$y = softmax(W_x + b) \tag{2.5}$$

2.1.3 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：

```
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：

```
x = tf.placeholder("float", [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。

```
W = tf.Variable(tf.zeros([784,10]))
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以得到一个 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，可以实现我们的模型了，只需以下一行代码：

```
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 W_x，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，我们的模型就可以在不同的设备上运行：计算机的 CPU，GPU，甚至是手机！

2.1.4 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：

$$H_{y'}(u) = -\sum_i y_{i'} log(y_i) \tag{2.6}$$

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：

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

然后我们可以用

$$-\sum y' log(y) \tag{2.7}$$

计算交叉熵:

```
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。

```
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：

```
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：

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

然后开始训练模型，这里我们让模型循环训练 1000 次！

```
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    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，然后我们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）- 在这里更确切的说是随机梯度下降训练。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。

2.1.5 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，它能给你在一个张量里沿着某条轴的最高条目的索引值。比如，`tf.argmax(y,1)`是模型认为每个输入最有可能对应的那些标签，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。

```
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 .

```
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。

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

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，在那里你将学到如何用FensorFlow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：

2.2 深入 MNIST

TensorFlow 是一个做大规模数值计算的强大库。其中一个特点就是它能够实现和训练深度神经网络。在这一小节里，我们将会学习在 MNIST 上构建深度卷积分类器的基本步骤。

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.

2.2.1 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 MINIST 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。

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

开始 TensorFlow 的交互会话

Tensorflow 基于一个高效的 C++ 模块进行运算。与这个模块的连接叫做 session。一般而言，使用 TensorFlow 程序的流程是先创建一个图，然后在 session 中加载它。

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，插入一些构建计算图的操作。这能给使用交互式文本 shell 如 iPython 带来便利。如果你没有使用 InteractiveSession 的话，你需要在开始 session 和加载图之前，构建整个计算图。

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

计算图

传统的计算行为中，为了更高效地在 Python 里进行数值计算，我们一般会使用像 NumPy 这样用其他语言编写的 lib，在 Python 外完成这些费时的操作（例如矩阵运算）。可是，每一步操作依然会经常在 Python 和第三方 lib 之间切换。这些操作很糟糕，特别是当你想在 GPU 上进行计算，又或者想使用分布式的做法的时候。这些情况下数据传输代价高昂。

在 TensorFlow 中，也有 Python 与外界的频繁操作。但是它在这一方面，做了进一步的改良。TensorFlow 构建一个交互操作的图，作为一个整体在 Python 外运行，而不

是以代价高昂的单个交互操为单位在 Python 外运行。这与 Theano、Torch 的做法很相似。

所以，这部分 Python 代码，目的是构建这个在外部运行的计算图，并安排这个计算图的哪一部分应该被运行。详细请阅读计算图部分的基本用法。

2.2.2 构建 Softmax Regression 模型

在这小节里，我们将会构建一个一层线性的 softmax regression 模型。下一节里，我们会扩展到多层卷积网络。

占位符

我们先来创建计算图的输入（图片）和输出（类别）。

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

这里的x和y并不是具体值，他们是一个placeholder，是一个变量，在 TensorFlow 运行计算的时候使用。

输入图片 x 是浮点数 2 维张量。这里，定义它的shape为[None, 784]，其中 784 是单张展开的 MNIST 图片的维度数。shape的第一维输入指代一个 batch 的大小，None，可为任意值。输出值y_也是一个 2 维张量，其中每一行为一个 10 维向量代表对应 MNIST 图片的分类。

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。

Variables

我们现在为模型定义权重W和偏置b。它们可以被视作是额外的输入量，但是 TensorFlow 有一个更好的方式来处理：Variable。一个Variable代表着在 TensorFlow 计算图中的一个值，它是能在计算过程中被读取和修改的。在机器学习的应用过程中，模型参数一般用Variable来表示。

```
W = tf.Variable(tf.zeros([784,10]))
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 784×10 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）。

Variable需要在session之前初始化，才能在session中使用。初始化需要初始值（本例当中是全为零）传入并赋值给每一个Variable。这个操作可以一次性完成。

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

预测分类与损失函数

现在我们可以实现我们的 regression 模型了。这只需要一行！我们把图片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。

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

在训练中最小化损失函数同样很简单。我们这里的损失函数用目标分类和模型预测分类之间的交叉熵。

```
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。

2.2.3 训练模型

我们已经定义好了模型和训练的时候用的损失函数，接下来使用 TensorFlow 来训练。因为 TensorFlow 知道整个计算图，它会用自动微分法来找到损失函数对于各个变量的梯度。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。

```
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。

train_step这个操作，用梯度下降来更新权值。因此，整个模型的训练可以通过反复地运行train_step来完成。

```
for i in range(1000):
    batch = mnist.train.next_batch(50)
    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，使用 feed_dict，用训练数据替换 placeholder 向量 x 和 y_。

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 placeholder。

评估模型

我们的模型效果怎样？

首先，要先知道我们哪些 label 是预测正确了。tf.argmax 是一个非常有用的函数。它会返回一个张量某个维度中的最大值的索引。例如，tf.argmax(y,1) 表示我们模型对每个输入的最大概率分类的分类值。而 tf.argmax(y_,1) 表示真实分类值。我们可以用 tf.equal 来判断我们的预测是否与真实分类一致。

```
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。

```
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%。

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

2.2.4 构建多层卷积网络模型

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

在创建模型之前，我们先来创建权重和偏置。一般来说，初始化时应加入轻微噪声，来打破对称性，防止零梯度的问题。因为我们用的是 ReLU，所以用稍大于 0 的值来初始化偏置能够避免节点输出恒为 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。

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    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。

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], 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。权重是一个 [5, 5, 1, 32] 的张量，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。输出对应一个同样大小的偏置向量。

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

为了用这一层，我们把 x 变成一个 4d 向量，第 2、3 维对应图片的宽高，最后一维代表颜色通道。

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

我们把 x_image 和权值向量进行卷积相乘，加上偏置，使用 ReLU 激活函数，最后 max pooling。

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
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。

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

密集连接层

现在，图片降维到 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，使用 ReLU 激活。

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
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 来代表一个神经元在 dropout 中被保留的概率。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。

```
keep_prob = tf.placeholder("float")
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。

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

训练和评估模型

这次效果又有多好呢？我们用前面几乎一样的代码来测测看。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。

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print "step %d, training accuracy %g"%(i, train_accuracy)
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob:
    0.5})

print "test accuracy %g"%accuracy.eval(feed_dict={
    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 来快速和简易地搭建、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

2.3 TensorFlow Mechanics 101

代码地址: tensorflow/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）。我们的目标读者是有兴趣使用 TensorFlow 的机器学习资深人士。

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。

2.3.1 教程使用的文件

本教程引用如下文件：

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

2.3.2 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。

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

1

Edit table here[2]

[1] ‘fake_data‘标记是用于单元测试的，读者可以不必理会。

[2] 了解更多数据有关信息，请查阅此系列教程的 [数据下载](mnist/download/index.md) 部分.

输入与占位符

`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。

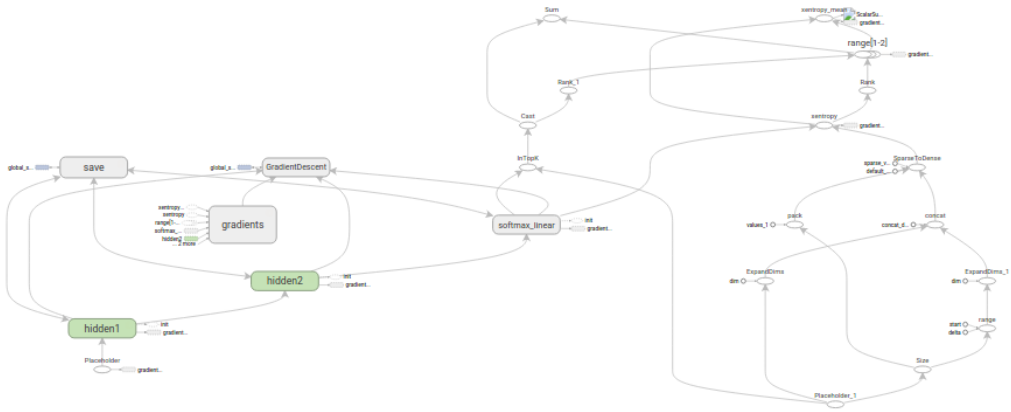
```
images_placeholder = tf.placeholder(tf.float32, shape=(batch_size,
    IMAGE_PIXELS))
labels_placeholder = tf.placeholder(tf.int32, shape=(batch_size))
```

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。

2.3.3 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()\lstinline`，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 logtis 模型的线性层。

每一层都创建于一个唯一的 [‘tf.name$_s cope$‘](../$api_d ocs/python/framework.mdname_s cope$)

```
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‘](../api_docs/python/state_ops.md#Vari

```
weights = tf.Variable(tf.truncated_normal([IMAGE_PIXELS,
    hidden1_units], stddev=1.0 / math.sqrt(float(IMAGE_PIXELS))), name
    ='weights')
biases = tf.Variable(tf.zeros([hidden1_units]), 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‘](../api_docs/python/constant_op.md#truncated_norm

然后,通过 [‘tf.zeros‘](../api_docs/python/constant_op.md#zeros)biases0shapeconnectto

图表的三个主要操作,分别是两个 [‘tf.nn.relu‘](../api_docs/python/nn.md#relu)[‘tf.matmul‘](../api_docs/

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

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

```
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_placeholer`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]`

```
batch_size = tf.size(labels)
labels = tf.expand_dims(labels, 1)
indices = tf.expand_dims(tf.range(0, batch_size, 1), 1)
concated = tf.concat(1, [indices, labels])
onehot_labels = tf.sparse_to_dense(
    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[3],用来比较`inference()`函数与 1-hot 标签所输出的 logits Tensor。

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits,
    onehot_labels, 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，将将该值作为总损失。

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

最后，程序会返回包含了损失值的 Tensor。

[3]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）。

```
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）。

```
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）。

```
global_step = tf.Variable(0, name='global_step', trainable=False)
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。

2.3.4 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 (The Graph)

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。

```
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 (The Session)

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。

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

另外，也可以利用`with`代码块生成`Session`，限制作用域：

```
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。

```
init = tf.initialize_all_variables()
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，而能实现有效训练的最简单循环就是：

```
for step in xrange(max_steps):
    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

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。

```
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。

```
feed_dict = {
    images_placeholder: images_feed,
    labels_placeholder: labels_feed,
}
```

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]`。

```
for step in xrange(FLAGS.max_steps):
    feed_dict = fill_feed_dict(data_sets.train, images_placeholder,
    labels_placeholder)
    _, loss_value = sess.run([train_op, loss], 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，其在返回的元祖中的对应元素就是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。

```
if step % 100 == 0:
    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。

```
summary_op = tf.merge_all_summaries()
```

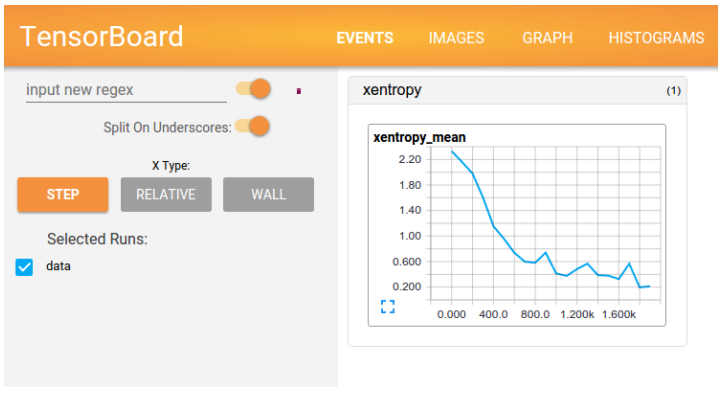
在创建好会话（session）之后，可以实例化一个[`tf.train.SummaryWriter`]，用于写入包含了图表本身和即时数据具体值的事件文件。

```
summary_writer = tf.train.SummaryWriter(FLAGS.train_dir, 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。。

```
summary_str = sess.run(summary_op, feed_dict=feed_dict)
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](../how_tos/summaries_and_tensorboard.md)

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`]

```
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。

```
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。

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

2.3.5 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、验证数据集合测试数据集。

```
print 'Training␣Data␣Eval:'
do_eval(sess, eval_correct, images_placeholder, labels_placeholder,
    data_sets.train)
print 'Validation␣Data␣Eval:'
do_eval(sess, eval_correct, images_placeholder, labels_placeholder,
    data_sets.validation)
print 'Test␣Data␣Eval:'
do_eval(sess, eval_correct, images_placeholder, labels_placeholder,
    data_sets.test)
```

>注意,更复杂的使用场景通常是,先隔绝‘data$_sets.test‘hyperparametertuningMNIST$

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$_data(train = False)$‘

```
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。

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

‘evaluation‘函数会生成[‘tf.nn.in$_top_k$‘](../$api_docs/python/nn.md in_top_k)$KK1$

```
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$_dict‘‘sess.run()‘‘eval_correct‘$

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

‘true$_count‘‘in_top_k‘$

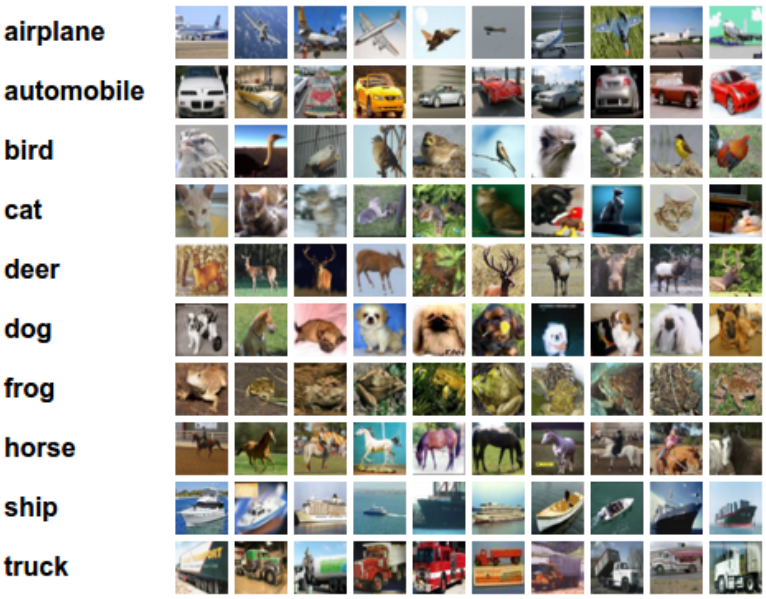
```
precision = float(true_count) / float(num_examples)
print '␣␣Num␣examples:␣%d␣␣Num␣correct:␣%d␣␣Precision␣@␣1:␣%0.02f' %
    (
    num_examples, true_count, precision)
```

> 原文：[TensorFlow Mechanics 101](http://www.tensorflow.org/tutorials/mnist/tf/index.md)
> 翻译：[bingjin](https://github.com/bingjin) > 校对：[LichAmnesia](https://github.com/LichAmnesia)

2.4 卷积神经网络

2.4.1 Overview

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 32x32RGB 的图像进行分类，这些图像涵盖了 10 个类别：`airplane`, `automobile`, `bird`, `cat`, `deer`, `dog`, `frog`, `horse`, `ship`, 和 `truck`.[4]



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：

- 着重于建立一个规范的网络组织结构，训练并进行评估；
- 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：

[4] This tutorial is intended for advanced users of TensorFlow and assumes expertise and experience in machine learning

- 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 GUP 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 Cnns 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，该模型达到了最高 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，分类一副图像需要大概 19.5M 个乘加操作。

2.4.2 Code Organization

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

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

2.4.3 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
conv1	实现卷积以及 rectified linear activation.
pool1	max pooling.
norm1	局部响应归一化.
conv2	卷积 and rectified linear activation.
norm2	局部响应归一化.
pool2	max pooling.
local3	基于修正线性激活的全连接层.
local4	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：

图 2.1: CIFAR-10 Loss

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。

图 2.2: CIFAR-10 Learning Rate Decay

`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。

2.4.4 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。

```
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:

```
Filling queue with 20000 CIFAR images before starting to train. This will ta
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 se
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315
...
```

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。

2.4.5 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。

```
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:

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

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。

2.4.6 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。

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

训练脚本的输出如下所示：

```
Filling queue with 20000 CIFAR images before starting to train. This will take
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 se
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 se
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 se
```

```
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315
...
```

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。

2.4.7 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

2.5 Vector Representations of Words

In this tutorial we look at the word2vec model by Mikolov et al. This model is used for learning vector representations of words, called "word embeddings".

在本教程我们来看一下Mikolovet 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"wordembedding"。

2.5.1 亮点

本教程旨在意在展现出在 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 相关内容已经是个专家了，那么请安心跳过本节内容，直接深入细节干一些脏活吧。

2.5.2 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：*基于技术的方法* (如，潜在语义分析)，和 *预测方法* (如，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 etal.，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。

2.5.3 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”)，

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

图 2.3:

当 **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：

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

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

图 2.4:

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：

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

图 2.5:

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-contrastiveestimation(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！

2.5.4 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 'brown'。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：

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

图 2.6:

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 降纬技术。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！

2.5.5 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。

[] embeddings = tf.Variable(tf.random$_u niform$([$vocabulary_s ize, embedding_s ize$], −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。

[] nce$_w eights$=$tf.Variable$($tf.truncated_n ormal$([$vocabulary_s ize, embedding_s ize$],$stddev$=1.0/$mat$

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/wor 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。

[] 建立输入占位符 train$_i nputs$=$tf.placeholder$($tf.int$32,$shape$=[$batch_s ize$])$train_l abels$=$tf.placeho$

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。

[] embed = tf.nn.embedding$_l ookup$($embeddings, train_i nputs$)

好了，现在我们有了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。

[] 计算 NCE 损失函数, 每次使用负标签的样本. loss = tf.reduce$_m ean$($tf.nn.nce_l oss$($nce_w eights, nce_b iase$

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。

[] 使用 SGD 控制器. optimizer = tf.train.GradientDescentOptimizer(learning$_r ate$=1.0).$minimize$($loss$)

2.5.6 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。

[] for inputs, labels in generate$_b atch$(...) : $feed_d ict$={$training_i nputs$: $inputs, training_l abels$: $labels$},cu

完整地例子可参考 tensorflow/g3doc/tutorials/word2vec/word2vec_basic.py.

2.5.7 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。

2.5.8 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是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.

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。

2.5.9 优化实现

以上简单的例子展示了 TensorFlow 的灵活性。比如说，我们可以很轻松得用现成的 `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 来读取和填装数据，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。

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。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。

2.5.10 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向你展示 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

2.6 循环神经网络

2.6.1 介绍

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

2.6.2 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。

2.6.3 教程文件

本教程使用的下面文件的目录是 `models/rnn/ptb`:

[c]@ll@ 文件 作用 `ptb_word_lm.py` 在 PTB 数据集上训练一个语言模型.`reader.py` 读取数据集.

2.6.4 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。

2.6.5 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：

[] lstm = rnn$_{c}ell.BasicLSTMCell(lstm_{s}ize)LSTM.state{=}tf.zeros([batch_{s}ize, lstm.state_{s}ize])$

loss = 0.0 for current$_{b}atch_{o}f_{w}ordsinwords_{i}n_{d}ataset$:$.output, state{=}lstm(current_{b}atch_{o}f_{w}ords, st$

LSTM 输出可用于产生下一个词语的预测 logits = tf.matmul(output, softmax$_w$)+$softmax_b probabilities=$

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：

[] 一次给定的迭代中的输入占位符. words = tf.placeholder(tf.int32, [batch$_size, num_steps$])

lstm = rnn$_cell.BasicLSTMCell(lstm_size)LSTM.initial_state=state=tf.zeros([batch_size, lstm.state_s i$

for i in range(len(num$_steps)) : .output, state=lstm(words[:, i], state)$

其余的代码. ...

final$_state=state$

下面展现如何实现迭代整个数据集：

[] 一个 numpy 数组，保存每一批词语之后的 LSTM 状态. numpy$_state=initial_state.eval()total_loss=0.0f$

输入

在输入 LSTM 前，词语 ID 被嵌入到了一个密集的表示中 (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：

[] embedding$_matrix[vocabulary_size, embedding_size]word_embeddings=tf.nn.embedding_lookup(e$

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

图 2.7:

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

图 2.8:

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：

[] lstm = rnn$_cell.BasicLSTMCell(lstm_size)stacked_lstm=rnn_cell.MultiRNNCell([lstm]*number_o$

initial$_state=state=stacked_lstm.zero_state(batch_size, tf.float32)foriinrange(len(num_steps)):.ou$

其余的代码. ...

final$_state=state$

2.6.6 编译并运行代码

首先需要构建库，在 CPU 上编译：

```
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：

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

运行模型：

```
bazel-bin/tensorflow/models/rnn/ptb/ptb_word_lm \
  --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。

2.6.7 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：Warln 校对：HongyangWang

2.7 Sequence-to-Sequence Models

Recurrent neural networks can learn to model language, as already discussed in the RNN Tutorial (if you did not read it, please go through it before proceeding with this one). This raises an interesting question: could we condition the generated words on some input and generate a meaningful response? For example, could we train a neural network to translate from English to French? It turns out that the answer is *yes*.

This tutorial will show you how to build and train such a system end-to-end. You can start by running this binary.

```
bazel run -c opt <...>/models/rnn/translate/translate.py
  --data_dir [your_data_directory]
```

It will download English-to-French translation data from the WMT'15 Website prepare it for training and train. It takes about 20GB of disk space, and a while to download and prepare (see later for details), so you can start and leave it running while reading this tutorial.

This tutorial references the following files from `models/rnn`.

File	What's in it?
`seq2seq.py`	Library for building sequence-to-sequence models.
`translate/seq2seq_model.py`	Neural translation sequence-to-sequence model.
`translate/data_utils.py`	Helper functions for preparing translation data.
`translate/translate.py`	Binary that trains and runs the translation model.

2.7.1 Sequence-to-Sequence Basics

A basic sequence-to-sequence model, as introduced in Cho et al., 2014, consists of two recurrent neural networks (RNNs): an *encoder* that processes the input and a *decoder* that generates the output. This basic architecture is depicted below.

Each box in the picture above represents a cell of the RNN, most commonly a GRU cell or an LSTM cell (see the RNN Tutorial for an explanation of those). Encoder and decoder can share weights or, as is more common, use a different set of parameters. Mutli-layer cells have been successfully used in sequence-to-sequence models too, e.g. for translation Sutskever et al., 2014.

In the basic model depicted above, every input has to be encoded into a fixed-size state vector, as that is the only thing passed to the decoder. To allow the decoder more direct access to the input, an *attention* mechanism was introduced in Bahdanu et al., 2014. We will not go into the details of the attention mechanism (see the paper), suffice it to say that it allows the decoder to peek into the input at every decoding step. A multi-layer sequence-to-sequence network with LSTM cells and attention mechanism in the decoder looks like this.

2.7.2 TensorFlow seq2seq Library

As you can see above, there are many different sequence-to-sequence models. Each of these models can use different RNN cells, but all of them accept encoder inputs and decoder inputs. This motivates the interfaces in the TensorFlow seq2seq library (`models/rnn/seq2seq.py`). The basic RNN encoder-decoder sequence-to-sequence model works as follows.

[] outputs, states = basic$_rnn_seq2seq(encoder_inputs, decoder_inputs, cell)$

In the above call, `encoder_inputs` are a list of tensors representing inputs to the encoder, i.e., corresponding to the letters *A, B, C* in the first picture above. Similarly, `decoder_inputs` are tensors representing inputs to the decoder, *GO, W, X, Y, Z* on the first picture.

The `cell` argument is an instance of the `models.rnn.rnn_cell.RNNCell` class that determines which cell will be used inside the model. You can use an existing cell, such as `GRUCell` or `LSTMCell`, or you can write your own. Moreover, `rnn_cell` provides wrappers to construct multi-layer cells, add dropout to cell inputs or outputs, or to do other transformations, see the RNN Tutorial for examples.

The call to `basic_rnn_seq2seq` returns two arguments: `outputs` and `states`. Both of them are lists of tensors of the same length as `decoder_inputs`. Naturally, `outputs` correspond to the outputs of the decoder in each time-step, in the first picture above that would be *W, X, Y, Z, EOS*. The returned `states` represent the internal state of the decoder at every time-step.

In many applications of sequence-to-sequence models, the output of the decoder at time t is fed back and becomes the input of the decoder at time t+1. At test time, when decoding a sequence, this is how the sequence is constructed. During training, on the other hand, it is common to provide the correct input to the decoder at every time-step, even if the decoder made a mistake before. Functions in `seq2seq.py` support both modes using the `feed_previous` argument. For example, let's analyze the following use of an embedding RNN model.

[] outputs, states = embedding$_rnn_seq2seq(encoder_inputs, decoder_inputs, cell, num_encoder_symbols$

In the `embedding_rnn_seq2seq` model, all inputs (both `encoder_inputs` and `decoder_inputs`) are integer-tensors that represent discrete values. They will be embed-

ded into a dense representation (see the Vectors Representations Tutorial for more details on embeddings), but to construct these embeddings we need to specify the maximum number of discrete symbols that will appear: `num_encoder_symbols` on the encoder side, and `num_decoder_symbols` on the decoder side.

In the above invocation, we set `feed_previous` to False. This means that the decoder will use `decoder_inputs` tensors as provided. If we set `feed_previous` to True, the decoder would only use the first element of `decoder_inputs`. All other tensors from this list would be ignored, and instead the previous output of the encoder would be used. This is used for decoding translations in our translation model, but it can also be used during training, to make the model more robust to its own mistakes, similar to Bengio et al., 2015.

One more important argument used above is `output_projection`. If not specified, the outputs of the embedding model will be tensors of shape batch-size by `num_decoder_symbols` as they represent the logits for each generated symbol. When training models with large output vocabularies, i.e., when `num_decoder_symbols` is large, it is not practical to store these large tensors. Instead, it is better to return smaller output tensors, which will later be projected onto a large output tensor using `output_projection`. This allows to use our seq2seq models with a sampled softmax loss, as described in Jean et. al., 2015.

In addition to `basic_rnn_seq2seq` and `embedding_rnn_seq2seq` there are a few more sequence-to-sequence models in `seq2seq.py`, take a look there. They all have similar interfaces, so we will not describe them in detail. We will use `embedding_attention_seq2seq` for our translation model below.

2.7.3 Neural Translation Model

While the core of the sequence-to-sequence model is constructed by the functions in `models/rnn/seq2seq.py`, there are still a few tricks that are worth mentioning that are used in our translation model in `models/rnn/translate/seq2seq_model.py`.

Sampled softmax and output projection

For one, as already mentioned above, we want to use sampled softmax to handle large output vocabulary. To decode from it, we need to keep track of the output projection. Both the sampled softmax loss and the output projections are constructed by the following code in `seq2seq_model.py`.

[] if num$_s amples$>0$and num_s amples$<$self.target_v ocab_s ize$:$w=tf.get_v ariable("proj_w",[size,self.ta

def sampled$_l oss(inputs, labels)$: $labels=tf.reshape(labels,[-1,1])returntf.nn.sampled_s oftmax_l oss$

First, note that we only construct a sampled softmax if the number of samples (512 by

default) is smaller that the target vocabulary size. For vocabularies smaller than 512 it might be a better idea to just use a standard softmax loss.

Then, as you can see, we construct an output projection. It is a pair, consisting of a weight matrix and a bias vector. If used, the rnn cell will return vectors of shape batch-size by `size`, rather than batch-size by `target_vocab_size`. To recover logits, we need to multiply by the weight matrix and add the biases, as is done in lines 124-126 in `seq2seq_model.py`.

[] if output$_p rojection is not None: self.outputs[b] = [tf.matmul(output, output_p rojection[0]) + out$

Bucketing and padding

In addition to sampled softmax, our translation model also makes use of *bucketing*, which is a method to efficiently handle sentences of different lengths. Let us first clarify the problem. When translating English to French, we will have English sentences of different lengths L1 on input, and French sentences of different lengths L2 on output. Since the English sentence is passed as `encoder_inputs`, and the French sentence comes as `decoder_inputs` (prefixed by a GO symbol), we should in principle create a seq2seq model for every pair (L1, L2+1) of lengths of an English and French sentence. This would result in an enormous graph consisting of many very similar subgraphs. On the other hand, we could just pad every sentence with a special PAD symbol. Then we'd need only one seq2seq model, for the padded lengths. But on shorter sentence our model would be inefficient, encoding and decoding many PAD symbols that are useless.

As a compromise between contructing a graph for every pair of lengths and padding to a single length, we use a number of *buckets* and pad each sentence to the length of the bucket above it. In `translate.py` we use the following default buckets.

[] buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]

This means that if the input is an English sentence with 3 tokens, and the corresponding output is a French sentence with 6 tokens, then they will be put in the first bucket and padded to length 5 for encoder inputs, and length 10 for decoder inputs. If we have an English sentence with 8 tokens and the corresponding French sentence has 18 tokens, then they will not fit into the (10, 15) bucket, and so the (20, 25) bucket will be used, i.e. the English sentence will be padded to 20, and the French one to 25.

Remember that when constructing decoder inputs we prepend the special `GO` symbol to the input data. This is done in the `get_batch()` function in `seq2seq_model.py`, which also reverses the input English sentence. Reversing the inputs was shown to improve results for the neural translation model in Sutskever et al., 2014. To put it all together, imagine we have the sentence "I go.", tokenized as `["I", "go", "."]` as input and the sentence "Je vais." as output, tokenized `["Je", "vais", "."]`. It will be put in the (5, 10) bucket, with encoder inputs representing `[PAD PAD "." "go" "I"]` and decoder

inputs `[GO "Je" "vais" "." EOS PAD PAD PAD PAD PAD]`.

2.7.4 Let's Run It

To train the model described above, we need to a large English-French corpus. We will use the *10^9-French-English corpus* from the WMT'15 Website for training, and the 2013 news test from the same site as development set. Both data-sets will be downloaded to `data_dir` and training will start, saving checkpoints in `train_dir`, when this command is run.

```
bazel run -c opt <...>/models/rnn/translate:translate
  --data_dir [your_data_directory] --train_dir [checkpoints_directory]
  --en_vocab_size=40000 --fr_vocab_size=40000
```

It takes about 18GB of disk space and several hours to prepare the training corpus. It is unpacked, vocabulary files are created in `data_dir`, and then the corpus is tokenized and converted to integer ids. Note the parameters that determine vocabulary sizes. In the example above, all words outside the 40K most common ones will be converted to an `UNK` token representing unknown words. So if you change vocabulary size, the binary will re-map the corpus to token-ids again.

After the data is prepared, training starts. Default parameters in `translate` are set to quite large values. Large models trained over a long time give good results, but it might take too long or use too much memory for your GPU. You can request to train a smaller model as in the following example.

```
bazel run -c opt <...>/models/rnn/translate:translate
  --data_dir [your_data_directory] --train_dir [checkpoints_directory]
  --size=256 --num_layers=2 --steps_per_checkpoint=50
```

The above command will train a model with 2 layers (the default is 3), each layer with 256 units (default is 1024), and will save a checkpoint every 50 steps (the default is 200). You can play with these parameters to find out how large a model can be to fit into the memory of your GPU.

During training, every `steps_per_checkpoint` steps the binary will print out statistics from recent steps. With the default parameters (3 layers of size 1024), first messages look like this.

```
global step 200 learning rate 0.5000 step-time 1.39 perplexity 1720.62
  eval: bucket 0 perplexity 184.97
  eval: bucket 1 perplexity 248.81
  eval: bucket 2 perplexity 341.64
```

```
  eval: bucket 3 perplexity 469.04
global step 400 learning rate 0.5000 step-time 1.38 perplexity 379.89
  eval: bucket 0 perplexity 151.32
  eval: bucket 1 perplexity 190.36
  eval: bucket 2 perplexity 227.46
  eval: bucket 3 perplexity 238.66
```

You can see that each step takes just under 1.4 seconds, the perplexity on the training set and the perplexities on the development set for each bucket. After about 30K steps, we see perplexities on short sentences (bucket 0 and 1) going into single digits. Since the training corpus contains ~22M sentences, one epoch (going through the training data once) takes about 340K steps with batch-size of 64. At this point the model can be used for translating English sentences to French using the `--decode` option.

```
bazel run -c opt <...>/models/rnn/translate:translate --decode
  --data_dir [your_data_directory] --train_dir [checkpoints_directory]

Reading model parameters from /tmp/translate.ckpt-340000
>  Who is the president of the United States?
 Qui est le président des États-Unis ?
```

2.7.5 What Next?

The example above shows how you can build your own English-to-French translator, end-to-end. Run it and see how the model performs for yourself. While it has reasonable quality, the default parameters will not give you the best translation model. Here are a few things you can improve.

First of all, we use a very promitive tokenizer, the `basic_tokenizer` function in `data_utils`. A better tokenizer can be found on the WMT'15 Website. Using that tokenizer, and a larger vocabulary, should improve your translations.

Also, the default parameters of the translation model are not tuned. You can try changing the learning rate, decay, or initializing the weights of your model in a different way. You can also change the default `GradientDescentOptimizer` in `seq2seq_model.py` to a more advanced one, such as `AdagradOptimizer`. Try these things and see how they improve your results!

Finally, the model presented above can be used for any sequence-to-sequence task, not only for translation. Even if you want to transform a sequence to a tree, for example to generate a parsing tree, the same model as above can give state-of-the-art results, as demonstrated in Vinyals & Kaiser et al., 2015. So you can not only build your own translator, you can also build a parser, a chat-bot, or any program that comes to your mind. Experiment!

2.8 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。

2.8.1 基本设置

首先, 我们需要导入一些必要的引用。

[] 导入模拟仿真需要的库 import tensorflow as tf import numpy as np

导入可视化需要的库 import PIL.Image from cStringIO import StringIO 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。

[] def DisplayArray(a, fmt='jpeg', rng=[0,1]): """Display an array as a picture.""" a = (a - rng[0])/float(rng[1] - rng[0])*255 a = np.uint8(np.clip(a, 0, 255)) f = StringIO() PIL.Image.fromarray(a).save(f, fmt) 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。

[] sess = tf.InteractiveSession()

2.8.2 定义计算函数

[] def make$_kernel(a)$: """$Transforma2Darrayintoaconvolutionkernel$"""$a=np.asarray(a)a=a.resh$

def simple$_conv(x,k)$: """$Asimplified2Dconvolutionoperation$"""$x=tf.expand_dims(tf.expand_dims$

def laplace(x): """Compute the 2D laplacian of an array""" laplace$_k=make_kernel$([[0.5,1.0,0.5],[1.0,−6.,1.0],[0

2.8.3 定义偏微分方程

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

[] 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。

[] Initial Conditions – some rain drops hit a pond

Set everything to zero u$_init=np.zeros$([N,N],$dtype$="$float32$")$ut_init=np.zeros$([N,N],$dtype$="$float3$

Some rain drops hit a pond at random points for n in range(40): a,b = np.random.randint(0, N, 2) u$_init[a, b]$=$np.random.uniform()$

DisplayArray(u$_init, rng$=[−0.1,0.1])

图 2.9: jpeg

现在，让我们来指定该微分方程的一些详细参数。

[] Parameters: eps – time resolution damping – wave damping eps = tf.placeholder(tf.float32, shape=()) damping = tf.placeholder(tf.float32, shape=())

Create variables for simulation state U = tf.Variable(u$_init)Ut=tf.Variable(ut_init)$

Discretized PDE update rules U=$U+eps*UtUt=Ut+eps*(laplace(U)-damping*Ut)$

Operation to update the state step = tf.group(U.assign(U$_)$, $Ut.assign(Ut_)$)

2.8.4 开始仿真

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

[] Initialize state to initial conditions tf.initialize$_all_variables().run()$

Run 1000 steps of PDE for i in range(1000): Step simulation step.run({eps: 0.03, damping: 0.04}) Visualize every 50 steps if i clear$_output()DisplayArray(U.eval(), rng$=[−0.1,0.1])

图 2.10: jpeg

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](https://github.com/wangaicc) 校对:[@tensorfly](https://github.com/tensorfly)

2.9 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。

2.9.1 教程文件

本教程需要使用以下文件：

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

2.9.2 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.

图 2.11: MNIST Digits

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
[c]@ll@	
train-images-idx3-ubyte.gz	训练集图片 - 55000 张训练图片, 5000 张验证图片
train-labels-idx1-ubyte.gz	训练集图片对应的数字标签
t10k-images-idx3-ubyte.gz	测试集图片 - 10000 张图片
t10k-labels-idx1-ubyte.gz	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:

[c]@ll@ 数据集目的 `data_sets.train` 55000 组图片和标签，用于训练。`data_sets.validation` 5000 组图片和标签，用于迭代验证训练的准确性。`data_sets.test` 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。

[] images$_feed, labels_feed = data_set.next_batch(FLAGS.batch_size)$

原文地址：MNIST Data Download 翻译：btpeter 校对：waiwaizheng

第三章　运作方式

综述 Overview

3.0.1 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

3.0.2 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

3.0.3 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

3.0.4 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

3.0.5 数据读入

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

参看教程

3.0.6 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

3.0.7 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。

3.0.8 自定义数据的 Readers

如果你有相当大量的自定义数据集合，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。

3.0.9 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

3.0.10 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

3.1 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

3.1.1 变量创建

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。

```
# Create two variables.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name
    ="weights")
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。

3.1.2 变量初始化

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。

```
# Create two variables.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
...
# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()

# Later, when launching the model
with tf.Session() as sess:
  # Run the init operation.
  sess.run(init_op)
  ...
```

```
  # Use the model
  ...
```

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。

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

3.1.3 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。

Checkpoint Files

Variables are saved in binary files that, roughly, contain a map from variable names to tensor values.

When you create a Saver object, you can optionally choose names for the variables in the checkpoint files. By default, it uses the value of the `Variable.name` property for each variable.

保存变量

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

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

```
...
# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()

# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, initialize the variables, do some work,
    save the
# variables to disk.
with tf.Session() as sess:
  sess.run(init_op)
  # Do some work with the model.
  ..
  # Save the variables to disk.
  save_path = saver.save(sess, "/tmp/model.ckpt")
  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。

```
# Create some variables.
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, use the saver to restore variables from
    disk, and
# do some work with the model.
with tf.Session() as sess:
  # Restore variables from disk.
  saver.restore(sess, "/tmp/model.ckpt")
  print "Model␣restored."
  # Do some work with the model
  ...
```

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:

> You can create as many saver objects as you want if you need to save and restore different subsets of the model variables. The same variable can be listed in multiple saver objects, its value is only changed when the saver restore() method is run.
>
> If you only restore a subset of the model variables at the start of a session, you have to run an initialize op for the other variables. See `tf.initialize_variables()` for more information.

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()`。

```
# Create some variables.
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
# Add ops to save and restore only 'v2' using the name "my_v2"
saver = tf.train.Saver({"my_v2": v2})
# Use the saver object normally after that.
...
```

3.2 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.

3.2.1 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.

[] def my$_{i}mage_{f}ilter(input_{i}mages):conv1_{w}eights=tf.Variable(tf.random_{n}ormal$([5,5,32,32]),$nam$

conv2$_{w}eights=tf.Variable(tf.random_{n}ormal$([5,5,32,32]),$name="conv2_{w}eights")conv2_{b}iases=tf.$V

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.

[] First call creates one set of variables. result1 = my$_{i}mage_{f}ilter(image1)Anothersetiscreatedintheseco$

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：

[] variables$_{d}ict$={"$conv1_{w}eights$": $tf.Variable(tf.random_{n}ormal$([5,5,32,32]),$name="conv1_{w}eights"$

def my$_{i}mage_{f}ilter(input_{i}mages,variables_{d}ict):conv1=tf.nn.conv2d(input_{i}mages,variables_{d}ict$

conv2 = tf.nn.conv2d(relu1, variables$_{d}ict$["$conv2_{w}eights$"],$strides$=[1,1,1,1],$padding='SAME')returnt$

The 2 calls to my$_{i}mage_{f}ilter()nowusethesamevariablesresult1=my_{i}mage_{f}ilter(image1,variables_{d}$

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.

3.2.2 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`：

[] def conv$_r elu(input, kernel_s hape, bias_s hape)$:$Createvariablenamed"weights".weights = tf.get$

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.

[] def my$_i mage_f ilter(input_i mages)$: $withtf.variable_s cope("conv1")$:$Variablescreatedherewill$

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.

```
result1 = my_image_filter(image1)
result2 = my_image_filter(image2)
# 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()` 这个方法来指定.

```
with tf.variable_scope("image_filters") as scope:
    result1 = my_image_filter(image1)
    scope.reuse_variables()
    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.

3.2.3 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.

[] v = tf.get$_{v}ariable(name, shape, dtype, initializer)$

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：

[] with tf.variable$_{s}cope("foo"):v=tf.get_{v}ariable("v",[1])assert v.name=="foo/v:0"$

- 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：

[] with tf.variable$_{s}cope("foo"):v=tf.get_{v}ariable("v",[1])with tf.variable_{s}cope("foo",reuse=True):v$

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：

[] with tf.variable$_{s}cope("foo"):with tf.variable_{s}cope("bar"):v=tf.get_{v}ariable("v",[1])assert v.name=$

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 .

[] with tf.variable$_scope$("foo"):v=$tf.get_variable$("v",[1])$tf.get_variable_scope$().$reuse_variables$()$v1$=

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.

[] with tf.variable$_scope$("$root$"):$At start, the scope is not reusing.assert tf.get_variable_scope().reuse$=

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时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.

[] with tf.variable$_scope$("foo")$as foo_scope$:v=$tf.get_variable$("v",[1])$with tf.variable_scope(foo_sco$

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.

[] with tf.variable$_scope$("foo")$as foo_scope$:$assert foo_scope.name$=="foo"$with tf.variable_scope$("b

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的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.

[] with tf.variable$_scope$("foo",$initializer$=$tf.constant_initializer$(0.4)):v=$tf.get_variable$("v",[1])a

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 的名字的呢？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：

[] with tf.variable$_scope$("foo") :x=1.0+$tf.get_variable$("v",[1])$assert x.op.name$=="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.

[] with tf.variable$_scope$("foo") :$with tf.name_scope$("bar") :$v=tf.get_variable$("v",[1])x=1.0+$v assert v.n$

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.

3.2.4 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? — | — models/image/cifar10.py | 图像中检测对象的模型. models/rnn/rnn_cell.py|时间递归神经网络的元方法集. models/rnn/seq2seq.py |为创建 sequence-to-sequence 模型的方法集. 原文:Sharing Variables 翻译:nb312校对：Wiki

3.3 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。

为了更方便 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：

图 3.1: MNIST TensorBoard

3.3.1 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 scalary_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择的参数 `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。

[] merged_summary_op=tf.merge_all_summaries()summary_writer=tf.train.SummaryWriter('/tmp/

现在已经准备好用 TensorBoard 来可视化这些数据了。

3.3.2 启动 TensorBoard

输入下面的指令来启动 TensorBoard

```
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

```
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

3.4 TensorBoard: 图表可视化

TensorFlow 图表计算强大而又复杂，图表可视化在理解和调试时显得非常有帮助。下面是一个运作时的可式化例子。

“一个 TensorFlow 图表的可视化”) 一个 *TensorFlow* 图表的可视化。

为了显示自己的图表，需将 TensorBoard 指向此工作的日志目录并运行，点击图表顶部窗格的标签页，然后在左上角的菜单中选择合适的运行。想要深入学习关于如何运行 TensorBoard 以及如何保证所有必要信息被记录下来，请查看 Summaries 和 TensorBoard.

3.4.1 名称域（Name scoping）和节点（Node）

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

[] import tensorflow as tf

with tf.name$_s cope('hidden') as scope: a = tf.constant(5, name='alpha') W = tf.Variable(tf.random

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。

```
<td style="width: 50%;">
  <img src="../images/pool1_collapsed.png" alt="未展开的名称域" title="未展开
</td>
<td style="width: 50%;">
  <img src="../images/pool1_expanded.png" alt="展开的名称域" title="展开的名
</td>

<td style="width: 50%;">
  顶级名称域的初始视图<code>pool_1</code>，点击右上方橙色的<code>+</code>按钮
</td>
<td style="width: 50%;">
```

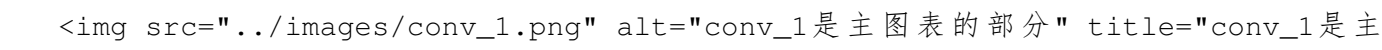
展开的<code>pool_1</code>名称域视图，点击右上方橙色的<code>-</code>按钮或双击
</td>

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造出一个混乱的视图。

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。

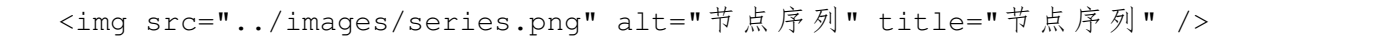
<td style="width: 50%;">


</td>
<td style="width: 50%;">


</td>

<td style="width: 50%;">
节点<code>conv_1</code>被连接到<code>save</code>，注意其右边<code>save</code>
</td>
<td style="width: 50%;">
<code>save</code> has a high degree, 并会作为从属节点出现，与<code>conv_1</code>
</td>

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。

<td style="width: 50%;">


</td>
<td style="width: 50%;">
</td>

```
  <img src="../images/series_expanded.png" alt="展开的节点序列" title="展开的
</td>
```

```
<td style="width: 50%;">
  一个节点序列的折叠视图。
</td>
<td style="width: 50%;">
  视图的一小块，双击后展开。
</td>
```

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
	High-level 节点代表一个名称域，双击则展开一个高层节点。
	彼此之间不连接的有限个节点序列。
	彼此之间相连的有限个节点序列。
	一个单独的操作节点。
	一个常量结点。
	一个摘要节点。

显示各操作间的数据流边。

显示各操作间的控制依赖边。

引用边，表示出度操作节点可以使入度 tensor 发生变化。

3.4.2 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。

```
<td style="width: 50%;">
  <img src="../images/infocard.png" alt="一个名称域的详情卡片" title="一个名称
</td>
<td style="width: 50%;">
  <img src="../images/infocard_op.png" alt="操作节点的详情卡片" title="操作节点
</td>

<td style="width: 50%;">
```

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
</td>
<td style="width: 50%;">
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
</td>

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件来着色。

下图是一张真实图表的图解：

<td style="width: 50%;">
<img src="../images/colorby_structure.png" alt="按结构着色" title="按结构着
</td>
<td style="width: 50%;">
<img src="../images/colorby_device.png" alt="按设备着色" title="按设备着色
</td>

<td style="width: 50%;">
结构视图：灰色节点的结构是唯一的。橙色的<code>conv1</code>和<code>conv2</c
</td>
<td style="width: 50%;">
设备视图：名称域根据其中的操作节点的设备片件来着色，在此紫色代表GPU，绿色
</td>

原文: TensorBoard: Graph Visualization 翻译: [@Warln](https://github.com/Warln) 校对: lucky521

3.5 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)。

3.5.1 目录

数据读取

3.5.2 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。

[] with tf.Session(): input = tf.placeholder(tf.float32) classifier = ... print classifier.eval(feed$_dict$={$input$: my_p

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。

3.5.3 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_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队列。

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：

[] filename$_{q}ueue$=$tf.train.string_{i}nput_{p}roducer$(["$file$0.$csv$","$file$1.$csv$"])

reader = tf.TextLineReader() key, value = reader.read(filename$_{q}ueue$)

Default values, in case of empty columns. Also specifies the type of the decoded result.

record$_{d}efaults$=[[1],[1],[1],[1],[1]]col1,col2,col3,col4,col5=$tf.decode_{c}sv(value,record_{d}efaults$=$record_{d}$

with tf.Session() as sess: Start populating the filename queue. coord = tf.train.Coordinator()

threads = tf.train.start$_{q}ueue_{r}unners(coord$=$coord)$

for i in range(1200): Retrieve a single instance: example, label = sess.run([features, col5])

coord.request$_{s}top()coord.join(threads)$

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.de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.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:

```
def read_my_file_format(filename_queue):
  reader = tf.SomeReader()
  key, record_string = reader.read(filename_queue)
  example, label = tf.some_decoder(record_string)
  processed_example = some_processing(example)
  return processed_example, label

def input_pipeline(filenames, batch_size, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example, label = read_my_file_format(filename_queue)
  # min_after_dequeue defines how big a buffer we will randomly sample
  #   from -- bigger means better shuffling but slower start up and more
  #   memory used.
  # capacity must be larger than min_after_dequeue and the amount larger
  #   determines the maximum we will prefetch.  Recommendation:
  #   min_after_dequeue + (num_threads + a small safety margin) * batch_size
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch(
      [example, label], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  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_函数. 示例:

```
def read_my_file_format(filename_queue):
  # Same as above

def input_pipeline(filenames, batch_size, read_threads, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example_list = [read_my_file_format(filename_queue)
                  for _ in range(read_threads)]
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch_join(
      example_list, batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  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：

[] Create the graph, etc. $init_op = tf.initialize_all_variables()$

Create a session for running operations in the Graph. sess = tf.Session()

Initialize the variables (like the epoch counter). sess.run($init_op$)

Start input enqueue threads. coord = tf.train.Coordinator() threads = tf.train.start$_{q}ueue_{r}unners(sess=ses$

try: while not coord.should$_{s}top() : Runtrainingstepsorwhateversess.run(train_{o}p)$

except tf.errors.OutOfRangeError: print 'Done training – epoch limit reached' finally:
When done, ask the threads to stop. coord.request$_{s}top()$

Wait for threads to finish. coord.join(threads) 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。因为我们是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`) 。

3.5.4 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）。

[] training$_data$=...$training_labels$=...$withtf.Session()$:$input_data$=$tf.constant(training_data)inp$

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。

[] training$_data$=...$training_labels$=...$withtf.Session()assess$:$data_initializer$=$tf.placeholder(dty$

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/`你可以用上面 `fully_connected_feed` 和 `fully_connected_reader` 的描述来进行比较。

3.5.5 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：

3.6 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。

`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...

3.6.1 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，回话终止的时候，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。

3.6.2 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。

[] 线程体：循环执行，直到 'Coordinator'收到了停止请求。 如果某些条件为真，请求 'Coordinator'去停止其他线程。 def MyLoop(coord): while not coord.should$_stop()$: ...$dosomething...if...somecondition...:coord.request_stop()$

Main code: create a coordinator. coord = Coordinator()

Create 10 threads that run 'MyLoop()' threads = [threading.Thread(target=MyLoop, args=(coord)) for i in xrange(10)]

Start the threads and wait for all of them to stop. for t in threads: t.start() 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。

3.6.3 QueueRunner

`QueueRunner` 类会创建一组线程，这些线程可以重复的执行 Enquene 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。

[] example = ...ops to create one example... Create a queue, and an op that enqueues examples one at a time in the queue. queue = tf.RandomShuffleQueue(...) enqueue$_op=queue.enqueue(exam$

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。

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

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

3.6.4 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。

[] try: for step in xrange(1000000): if coord.should$_stop()$: $breaksess.run(train_op)exceptException, e$:Rep

Terminate as usual. It is innocuous to request stop twice. coord.request$_stop()coord.join(threads)$

原文地址：Threading and Queues 翻译：zhangkom 校对：volvet

3.7 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 Pyhton。如果你定义了梯度，你可以使用 Python 的GradientChecker来测试它。

3.7.1 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

3.7.2 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.

```
#include "tensorflow/core/framework/op.h"
REGISTER_OP("ZeroOut")
    .Input("to_zero: int32")
    .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.

3.7.3 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:

```
 #include "tensorflow/core/framework/op_kernel.h"
using namespace tensorflow;
class ZeroOutOp : public OpKernel {
 public:
  explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
  void Compute(OpKernelContext* context) override {
    // 获取输入 tensor.
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<int32>();
   // 创建一个输出 tensor.
    Tensor* output_tensor = NULL;
    OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape()
                                                     &output_tensor));
    auto output = output_tensor->template flat<int32>();
    // 设置 tensor 除第一个之外的元素均设为 0.
    const int N = input.size();
    for (int i = 1; i < N; i++) {
      output(i) = 0;
    }
    // 尽可能地保留第一个元素的值.
    if (N > 0) output(0) = input(0);
  }
};
```

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:

```
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.

3.7.4 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` 中:

[] from tensorflow.python.ops.gen$_{u}ser_{o}psimport*

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.

[] tf$_{g}en_{o}p_{w}rapper_{p}y(name="user_{o}ps",hidden=["Fact",],require_{s}hape_{f}unctions=False,)

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:

[] def my$_{f}act()$: """Op."""$return gen_{u}ser_{o}ps._{f}act()$

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.

```
#include "tensorflow/cc/ops/user_ops.h"
```

3.7.5 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 包含以下内容:

[] import tensorflow as tf class ZeroOutTest(tf.test.TestCase): def testZeroOut(self): with self.test$_{s}ession()$: $result=tf.user_{o}ps.zero_{o}ut([5,4,3,2,1])self.assertAllEqual(result.eval(),[5,0,0,0,0])$

然后运行测试:

```
$ bazel test tensorflow/python:zero_out_op_test
```

3.7.6 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.

```
  void Compute(OpKernelContext* context) override {
   // 获取输入 tensor
```

```
    const Tensor& input_tensor = context->input(0);
    OP_REQUIRES(context, TensorShapeUtils::IsVector(input_tensor.shape()),
                errors::InvalidArgument("ZeroOut expects a 1-D vector."));
    // ...
  }
```

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/stat` `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.

3.7.7 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:

```
<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:

```
REGISTER_OP("ZeroOut")
    .Attr("preserve_index: int")
    .Input("to_zero: int32")
    .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:

```
class ZeroOutOp : public OpKernel {
 public:
  explicit ZeroOutOp(OpKernelConstruction * context) : OpKernel(context) {
   // 获取欲保存的索引值
    OP_REQUIRES_OK(context,
                   context->GetAttr("preserve_index", &preserve_index_));
    // 检查 preserve_index 是否为正
    OP_REQUIRES(context, preserve_index_ >= 0,
                errors::InvalidArgument("Need preserve_index >= 0, got ",
                                        preserve_index_));
  }
  void Compute(OpKernelContext* context) override {
    // ...
}
 private:
  int preserve_index_;
};
```

该值可以在 Compute 方法中被使用:

```
void Compute(OpKernelContext* context) override {
    // ...
   // 检查 preserve_index 范围是否合法
OP_REQUIRES(context, preserve_index_ < input.dimension(0),
                errors::InvalidArgument("preserve_index out of range"));
    // 设置输出 tensor 所有的元素值为 0
   const int N = input.size();
    for (int i = 0; i < N; i++) {
      output_flat(i) = 0;
    }
    // 保存请求的输入值
   output_flat(preserve_index_) = input(preserve_index_);
  }
```

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:

```
REGISTER_OP("ZeroOut")
    .Attr("preserve_index: int = 0")
    .Input("to_zero: int32")
    .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:

```
REGISTER_OP("EnumExample")
    .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`:

```
REGISTER_OP("RestrictedTypeExample")
      .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:

```
REGISTER_OP("NumberType")
        .Attr("t: numbertype");
```

对于这个 Op:

[] tf.number$_type(t=tf.int32)tf.number_type(t=tf.bool)$

- `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.

```
REGISTER_OP("MinIntExample")
      .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.

```
REGISTER_OP("TypeListExample")
      .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:

```
REGISTER_OP("AttrDefaultExample")
    .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:

```
REGISTER_OP("AttrDefaultExampleForAllTypes")
    .Attr("s: string = 'foo'")
    .Attr("i: int = 0")
    .Attr("f: float = 1.0")
    .Attr("b: bool = true")
    .Attr("ty: type = DT_INT32")
    .Attr("sh: shape = { dim { size: 1 } dim { size: 2 } }")
    .Attr("te: tensor = { dtype: DT_INT32 int_val: 5 }")
    .Attr("l_empty: list(int) = []")
    .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:

```
REGISTER_OP("ZeroOut")
     .Attr("T: {float, int32}")
     .Input("to_zero: <b>T</b>")
     .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:
>
> [] def zero$_out(to_zero, name=None)$:"""...:to_zero:‘$Tensor$‘.:‘$float32$‘,‘$int32$‘.$name$:().
> 返回值: 一个 ‘Tensor‘, 类型和 ‘to$_zero$‘."""
>
> 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:

```
REGISTER_OP("StringToNumber")
     .Input("string_tensor: string")
     .Output("output: out_type")
     .Attr("out_type: {float, int32}");
     .Doc(R"doc(
 Converts each string in the input Tensor to the specified numeric type.
 )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.

[] def string$_to_number(string_tensor, out_type=None, name=None):"""Tensor
参数: string$_tensor: `string``Tensor`.out$_type: `tf.DType`,`tf.float32, tf.int32`.`tf.float32`.name: ().
返回值: 一个 `out$_type``Tensor`."""

```
 #include "tensorflow/core/framework/op_kernel.h"
class ZeroOutInt32Op : public OpKernel {
  // 和之前一样
};
class ZeroOutFloatOp : public OpKernel {
 public:
  explicit ZeroOutFloatOp(OpKernelConstruction * context)
      : OpKernel(context) {}
  void Compute(OpKernelContext * context) override {
    // 获取输入 tensor
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<float>();
    // 创建一个输出 tensor
    Tensor * output = NULL;
    OP_REQUIRES_OK(context,
                   context->allocate_output(0, input_tensor.shape(), &output)
    auto output_flat = output->template flat<float>();
    // 设置输出 tensor 的所有元素为 0
    const int N = input.size();
    for (int i = 0; i &lt; N; i++) {
      output_flat(i) = 0;
    }<br/>
    // 保留第一个输入值
    if (N &gt; 0) output_flat(0) = input(0);
  }
};
```

```
// 注意, TypeConstraint<int32>("T") 意味着属性 "T" (在上面 Op 注册代码中
// 定义的) 必须是 "int32", 才能实例化.
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint&lt;int32&gt;("T"),
    ZeroOutOpInt32);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<float>("T"),
    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:

```
REGISTER_OP("ZeroOut")
  .Attr("T: {float, int32} = DT_INT32")
  .Input("to_zero: T")
  .Output("zeroed: T")
```

如果需要添加更多类型, 例如 double:

```
REGISTER_OP("ZeroOut")
    .Attr("T: {float, double, int32}")
    .Input("to_zero: T")
    .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 keneral 注册 (REGISTER_KERNEL_BUILDER 调用).

```
template <typename T>;
class ZeroOutOp : public OpKernel {
 public:
    explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
  void Compute(OpKernelContext* context) override {
    // 获取输入 tensor
     const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<T>();
    // 创建一个输出 tensor
```

```
      Tensor* output = NULL;
    OP_REQUIRES_OK(context,
                   context->allocate_output(0, input_tensor.shape(), &output));
    auto output_flat = output->template flat<T>();
    // 设置输出 tensor 的所有元素为 0
   const int N = input.size();
    for (int i = 0; i < N; i++) {
      output_flat(i) = 0;
    }
    // Preserve the first input value
    if (N > 0) output_flat(0) = input(0);
  }
};
};<br/>
// 注意, TypeConstraint<int32>("T") 意味着属性 "T" (在上面 Op 注册代码中
// 定义的) 必须是 "int32", 才能实例化. </b>
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<int32>("T"),
    ZeroOutOp<int32>);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<float>("T"),
    ZeroOutOp<float>);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<double>("T"),
    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.

```
#include "tensorflow/core/framework/op_kernel.h"
#define REGISTER_KERNEL(type)                                          \
 REGISTER_KERNEL_BUILDER(                                             \
     Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
     ZeroOutOp<type>)
```

```
REGISTER_KERNEL(int32);
REGISTER_KERNEL(float);
REGISTER_KERNEL(double);
 #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_ty
提供的宏:

```
 #include "tensorflow/core/framework/op_kernel.h"
 #include "tensorflow/core/framework/register_types.h"
REGISTER_OP("ZeroOut")
    .Attr("T: realnumbertype")
    .Input("to_zero: T")
    .Output("zeroed: T");
template <typename T>
class ZeroOutOp : public OpKernel { ... };
 #define REGISTER_KERNEL(type)                                          \
  REGISTER_KERNEL_BUILDER(                                             \
      Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      ZeroOutOp<type>)
TF_CALL_REAL_NUMBER_TYPES(REGISTER_KERNEL);
 #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.

```
REGISTER_OP("PolymorphicListExample")
    .Attr("T: list(type)")
    .Input("in: T")
    .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)`.

```
REGISTER_OP("ListTypeRestrictionExample")
    .Attr("T: list({float, double})")
```

```
    .Input("in: T")
    .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:

```
REGISTER_OP("IntListInputExample")
    .Attr("N: int")
    .Input("in: N * int32")
    .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:

```
REGISTER_OP("SameListInputExample")
    .Attr("N: int")
    .Attr("T: type")
    .Input("in: N * T")
    .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:

```
REGISTER_OP("MinLengthIntListExample")
    .Attr("N: int >= 2")
    .Input("in: N * int32")
    .Output("out: int32");
```

同样的语法也适用于 `"list(type)"` 属性:

```
REGISTER_OP("MinimumLengthPolymorphicListExample")
    .Attr("T: list(type) >= 3")
    .Input("in: T")
    .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:

```
REGISTER_OP("MultipleInsAndOuts")
    .Input("y: int32")
    .Input("z: float")
    .Output("a: string")
    .Output("b: int32");
```

每一个输入或输出形式如下:

```
<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.

```
REGISTER_OP("BuiltInTypesExample")
      .Input("integers: int32")
      .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.

```
REGISTER_OP("PolymorphicSingleInput")
      .Attr("T: type")
      .Input("in: T);
REGISTER_OP("RestrictedPolymorphicSingleInput")
      .Attr("T: {int32, int64}")
      .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.

```
REGISTER_OP("ArbitraryTensorSequenceExample")
      .Attr("T: list(type)")
      .Input("in: T")
      .Output("out: T");
REGISTER_OP("RestrictedTensorSequenceExample")
      .Attr("T: list({int32, int64})")
      .Input("in: T")
      .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:

```
REGISTER_OP("Int32SequenceExample")
    .Attr("NumTensors: int")
    .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:

```
REGISTER_OP("SameTypeSequenceExample")
    .Attr("NumTensors: int")
    .Attr("T: type")
    .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.

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:

1. 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:

```
REGISTER_OP("MyGeneralUnaryOp")
    .Input("in: float")
    .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:

```
REGISTER_OP("MyGeneralUnaryOp")
    .Input("in: T")
    .Output("out: T")
    .Attr("T: numerictype = float");
```

1. 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, 或将原来的单元素输入变成一个列表).

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.

3.7.8 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 GPU, 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:

```
 #define REGISTER_GPU_KERNEL(T)                                \
REGISTER_KERNEL_BUILDER(Name("Pad")                          \
                            .Device(DEVICE_GPU)       \
                            .TypeConstraint<T>("T")   \
                            .HostMemory("paddings"), \
                        PadOp<GPUDevice, T>)
```

3.7.9 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 \(\partial/\partial y\)\(\partial/\partial x\).

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial}{\partial y}\frac{\partial f}{\partial x}.$$

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:

[] from tensorflow.python.framework import ops from tensorflow.python.ops import array$_ops from tensorflow.python.ops import sparse$_ops

@ops.RegisterGradient("ZeroOut") def $_zero_out_grad(op, grad):"""‘zero$_out‘.

参数: op: 欲进行微分的 ‘zero$_out“‘, $Op.grad :‘zero_out‘Op.

返回: 代表输入 ‘zero$_out‘."""$to$_zero=op.inputs[0]shape=array$_ops.shape(to_zero)index=array$_ops.zer

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` `op` 和一个 `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`. 举一个例子, 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.

3.7.10 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:

[] @tf.RegisterShape("ZeroOut"): def $_zero_out_shape(op):"""$ZeroOutOp.

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. """ 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:

[] @tf.RegisterShape("ZeroOut"): def $_zero_out_shape(op):"""$ZeroOutOp.

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). """ input$_shape$=$op.inputs$[0].get_shape().$with_rank$(1)$return$[$input_shape$]

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:

```
@tf.RegisterShape("IntListInputExample")
def _int_list_input_example_shape(op):
  """ "IntListInputExample" Op 的形状函数.

  所有的输入和输出是同大小的矩阵.
  """
  output_shape = tf.TensorShape(None)
  for input in op.inputs:
    output_shape = output_shape.merge_with(input.get_shape().with_rank(2))
  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](https://github.com/PFZheng) 校对：[@ZHNathanielLee](https://github.com/ZHNathanielLee)

3.8 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。

3.8.1 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

3.8.2 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关键值。

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:

```
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：

```
 #include "tensorflow/core/framework/op.h"
REGISTER_OP("TextLineReader")
    .Output("reader_handle: Ref(string)")
    .Attr("skip_header_lines: int = 0")
    .Attr("container: string = ''")
    .Attr("shared_name: string = ''")
    .SetIsStateful()
    .Doc(R"doc(
A Reader that outputs the lines of a file delimited by '\n'.
)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：

```
 #include "tensorflow/core/framework/reader_op_kernel.h"
class TFRecordReaderOp : public ReaderOpKernel {
 public:
  explicit TFRecordReaderOp(OpKernelConstruction* context)
      : ReaderOpKernel(context) {
    Env* env = context->env();
    SetReaderFactory([this, env]() { return new TFRecordReader(name(), env); });
  }
};
REGISTER_KERNEL_BUILDER(Name("TFRecordReader").Device(DEVICE_CPU),
                        TFRecordReaderOp);
```

一个带有属性的例子：

```
 #include "tensorflow/core/framework/reader_op_kernel.h"
class TextLineReaderOp : public ReaderOpKernel {
 public:
  explicit TextLineReaderOp(OpKernelConstruction* context)
      : ReaderOpKernel(context) {
    int skip_header_lines = -1;
    OP_REQUIRES_OK(context,
                   context->GetAttr("skip_header_lines", &skip_header_lines));
    OP_REQUIRES(context, skip_header_lines >= 0,
                errors::InvalidArgument("skip_header_lines must be >= 0 not ",
```

```
                                                skip_header_lines));
    Env* env = context->env();
    SetReaderFactory([this, skip_header_lines, env]() {
      return new TextLineReader(name(), skip_header_lines, env);
    });
  }
};
REGISTER_KERNEL_BUILDER(Name("TextLineReader").Device(DEVICE_CPU),
                        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。

```
from tensorflow.python.framework import ops
from tensorflow.python.ops import common_shapes
from tensorflow.python.ops import io_ops
class SomeReader(io_ops.ReaderBase):
    def __init__(self, name=None):
        rr = gen_user_ops.some_reader(name=name)
        super(SomeReader, self).__init__(rr)
ops.NoGradient("SomeReader")
ops.RegisterShape("SomeReader")(common_shapes.scalar_shape)
```

你可以在 tensorflow/python/ops/io_ops.py中查看一些范例。

3.8.3 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](https://github.com/derekshang)
校对： Wiki

3.9 使用 GPUs

3.9.1 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` .

3.9.2 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`.

[] 新建一个 graph. a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a') b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b') c = tf.matmul(a, b) 新建 session with $\log_device_placementTrue.sess=tf.Session(config=tf.ConfigProto(log_device_placement=Tr$

你应该能看见以下输出:

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bu
id: 0000:05:00.0
b: /job:localhost/replica:0/task:0/gpu:0
a: /job:localhost/replica:0/task:0/gpu:0
MatMul: /job:localhost/replica:0/task:0/gpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

3.9.3 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.

[] 新建一个 graph. with tf.device('/cpu:0'): a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a') b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b') c =

tf.matmul(a, b) 新建 session with $\log_{d}evice_{p}lacementTrue.sess=tf.Session(config=tf.ConfigProto(log_{d}$

你会发现现在 a 和 b 操作都被指派给了 cpu:0.

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus
id: 0000:05:00.0
b: /job:localhost/replica:0/task:0/cpu:0
a: /job:localhost/replica:0/task:0/cpu:0
MatMul: /job:localhost/replica:0/task:0/gpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

3.9.4 在多 GPU 系统里使用单一 GPU

如果你的系统里有多个 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:

[] 新建一个 graph. with tf.device('/gpu:2'): a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a') b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b') c = tf.matmul(a, b) 新建 session with $\log_{d}evice_{p}lacementTrue.sess=tf.Session(config=tf.ConfigProto(log_{d}$

如果你指定的设备不存在, 你会收到 InvalidArgumentError 错误提示:

```
InvalidArgumentError: Invalid argument: Cannot assign a device to node 'b':
Could not satisfy explicit device specification '/gpu:2'
   [[Node: b = Const[dtype=DT_FLOAT, value=Tensor<type: float shape: [3,2]
   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.

[] 新建一个 graph. with tf.device('/gpu:2'): a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a') b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b') c = tf.matmul(a, b) 新建 session with $\log_{d}evice_{p}lacementTrue.sess=tf.Session(config=tf.ConfigProto(allo$

3.9.5 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:

```
# 新建一个 graph.
c = []
for d in ['/gpu:2', '/gpu:3']:
  with tf.device(d):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
    c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
  sum = tf.add_n(c)
# 新建session with log_device_placement并设置为True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# 运行这个op.
print sess.run(sum)
```

你会看到如下输出:

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K20m, pci bu
id: 0000:02:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K20m, pci bu
id: 0000:03:00.0
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K20m, pci bu
id: 0000:83:00.0
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K20m, pci bu
id: 0000:84:00.0
Const_3: /job:localhost/replica:0/task:0/gpu:3
Const_2: /job:localhost/replica:0/task:0/gpu:3
MatMul_1: /job:localhost/replica:0/task:0/gpu:3
Const_1: /job:localhost/replica:0/task:0/gpu:2
Const: /job:localhost/replica:0/task:0/gpu:2
MatMul: /job:localhost/replica:0/task:0/gpu:2
AddN: /job:localhost/replica:0/task:0/cpu:0
[[  44.   56.]
 [  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](https://github.com/lianghyv) 校对:Wiki

第四章　资源

第五章　其他