Tensorflow 入门

深度学习中的框架特点及介绍
下面我们来对深度学习中的各个框架的特点进行介绍

1.TensorFlow 框架

tensorflow 是用 c++语言开发的，同时支持 C，java，python 等多种语言多的调用，目前主流的方式通常会使用 python 语言进行驱动应用。利用 c++语言可以保证其运行效率，python 语言作为其上层应用语言，可以为研究人员节省大量的时间。
Tensorflow 与 CNTK，MXNET，theano 同属于符号计算架构，允许用户在不需要使用低级语言实现的情况下，开发出新的复杂层类型。基于图运算是其最主要的特点，通过图上的节点变量可以控制训练中各个环节的变量，尤其在需要进行底层操作时，Tensorflow 要比其他的框架更容易些。虽然 Tensorflow 在大型计算机集群中的并行处理，运算性能略低于 CNTK，但是在个人机器使用场景下，可以根据机器的配置自动选择 CPU 或者 GPU 来进行计算。

2.Theano
Theano 是一个十余年的 python 深度学习和机器学习框架，用来定义，优化和模拟数学表达式计算，用于高效的解决多为数组的计算问题，有较好的扩展性。

3.Torch
Torch 同样具有很好的扩展性，但是有些接口不够全面，比如 WGAN-OP 这样的网络需要手动计算来修改梯度没有对应的接口。其最大的缺点是，需要 LuauJIT 的支持，用于 Lu 语言，在 python 流行的今天，通用性能方面比较差。

4.Keras
keras 可以理解为一个 Theano 框架和 tensorflow 前端的一和个组合。其构建模型的 api 调用方式渐渐成为了主流，包括 Tensorflow，CNTK，MXNet 等知名框架，都提供对 keras 调用语法的支持。使用 keras 编写的代码，会有更好的可移植性。

5.DeepLearning4j
DeepLearning4j 是基于 java 语言和 Scala 语言开发的，应用在 Hadoop 和 spark 系统之上的深度学习软件。

6.Caffe
最初 caffe 是一个强大的图像分类框架，是最容易测试和评估性能的标准深度学习框架，并且提供了很多训练好的模型，尤其是该模型的复用价值在其他的框架中都会出现，大大提升了现有模型的训练时间。但是 Caffe 更新缓慢。

7.MXNet
MXNet 是一个可移植的，可伸缩的深度学习库，具有 Torch，Theano 和 caffe 的部分特性。在不同程度上面支持 Python，R，Scala，Julia 和 C++语言，也是目前比热门的主流框架之一。

8.CNTK
CNTK 是一个微软开发的深度学习软件包，以速度块儿著称，运用独特的神经网络配置语言 Brain Script，大大的降低了学习成本。有微软作为后盾，CNTK 成为最具有潜力的深度学习框架之一。目前的成熟度和 Tensorflow 相比有较大的差距，但是其与 Visual Studio 的耦合，以及特定的 MS 编程风格，使得熟悉 VS 的小伙伴极容易上手。

作为机器学习最重要的一个分支，深度学习近年来发展迅猛，在国内外都引起了广泛的关注。然而深度学习的火热也不是一时兴起的，而是经历了一段漫长的发展史。接下来我们了解一下深度学习的发展历程。

1. 深度学习的起源阶段

图 1（略） “AI 之父”马文·明斯基

1943 年，心里学家麦卡洛克和数学逻辑学家皮兹发表论文《神经活动中内在思想的逻辑演算》，提出了 MP 模型。MP 模型是模仿神经元的结构和工作原理，构成出的一个基于神经网络的数学模型，本质上是一种“模拟人类大脑”的神经元模型。MP 模型作为人工神经网络的起源，开创了人工神经网络的新时代，也奠定了神经网络模型的基础。

1949 年，加拿大著名心理学家唐纳德·赫布在《行为的组织》中提出了一种基于无监督学习的规则——海布学习规则(Hebb Rule)。海布规则模仿人类认知世界的过程建立一种“网络模型”，该网络模型针对训练集进行大量的训练并提取训练集的统计特征，然后按照样本的相似程度进行分类，把相互之间联系密切的样本分为一类，这样就把样本分成了若干类。海布学习规则与“条件反射”机理一致，为以后的神经网络学习算法奠定了基础，具有重大的历史意义。

20 世纪 50 年代末，在 MP 模型和海布学习规则的研究基础上，美国科学家罗森布拉特发现了一种类似于人类学习过程的学习算法——感知机学习。并于 1958 年，正式提出了由两层神经元组成的神经网络，称之为“感知器”。感知器本质上是一种线性模型，可以对输入的训练集数据进行二分类，且能够在训练集中自动更新权值。感知器的提出吸引了大量科学家对人工神经网络研究的兴趣，对神经网络的发展具有里程碑式的意义。

但随着研究的深入，在 1969 年，“AI 之父”马文·明斯基和 LOGO 语言的创始人西蒙·派珀特共同编写了一本书籍《感知器》，在书中他们证明了单层感知器无法解决线性不可分问题（例如：异或问题）。由于这个致命的缺陷以及没有及时推广感知器到多层神经网络中，在 20 世纪 70 年代，人工神经网络进入了第一个寒冬期，人们对神经网络的研究也停滞了将近 20 年。

2. 深度学习的发展阶段

图 2（略） 深度学习之父杰弗里·辛顿

1982 年，著名物理学家约翰·霍普菲尔德发明了 Hopfield 神经网络。Hopfield 神经网络是一种结合存储系统和二元系统的循环神经网络。Hopfield 网络也可以模拟人类的记忆，根据激活函数的选取不同，有连续型和离散型两种类型，分别用于优化计算和联想记忆。但由于容易陷入局部最小值的缺陷，该算法并未在当时引起很大的轰动。

直到 1986 年，深度学习之父杰弗里·辛顿提出了一种适用于多层感知器的反向传播算法——BP 算法。BP 算法在传统神经网络正向传播的基础上，增加了误差的反向传播过程。反向传播过程不断地调整神经元之间的权值和阈值，直到输出的误差达到减小到允许的范围之内，或达到预先设定的训练次数为止。BP 算法完美的解决了非线性分类问题，让人工神经网络再次的引起了人们广泛的关注。

图 2 深度学习之父杰弗里·辛顿

但是由于八十年代计算机的硬件水平有限，如：运算能力跟不上，这就导致当神经网络的规模增大时，再使用 BP 算法会出现“梯度消失”的问题。这使得 BP 算法的发展受到了很大的限制。再加上 90 年代中期，以 SVM 为代表的其它浅层机器学习算法被提出，并在分类、回归问题上均取得了很好的效果，其原理又明显不同于神经网络模型，所以人工神经网络的发展再次进入了瓶颈期。

3. 深度学习的爆发阶段

图 3 （略）AlphaGo 大战李世石

2006 年，杰弗里·辛顿以及他的学生鲁斯兰·萨拉赫丁诺夫正式提出了深度学习的概念。他们在世界顶级学术期刊《科学》发表的一篇文章中详细的给出了“梯度消失”问题的解决方案——通过无监督的学习方法逐层训练算法，再使用有监督的反向传播算法进行调优。该深度学习方法的提出，立即在学术圈引起了巨大的反响，以斯坦福大学、多伦多大学为代表的众多世界知名高校纷纷投入巨大的人力、财力进行深度学习领域的相关研究。而后又在迅速蔓延到工业界中。
2006 年，杰弗里·辛顿以及他的学生鲁斯兰·萨拉赫丁诺夫正式提出了深度学习的概念。他们在世界顶级学术期刊《科学》发表的一篇文章中详细的给出了“梯度消失”问题的解决方案——通过无监督的学习方法逐层训练算法，再使用有监督的反向传播算法进行调优。该深度学习方法的提出，立即在学术圈引起了巨大的反响，以斯坦福大学、多伦多大学为代表的众多世界知名高校纷纷投入巨大的人力、财力进行深度学习领域的相关研究。而后又在迅速蔓延到工业界中。
2012 年，在著名的 ImageNet 图像识别大赛中，杰弗里·辛顿领导的小组采用深度学习模型 AlexNet 一举夺冠。AlexNet 采用 ReLU 激活函数，从根本上解决了梯度消失问题，并采用 GPU 极大的提高了模型的运算速度。同年，由斯坦福大学著名的吴恩达教授和世界顶尖计算机专家 Jeff Dean 共同主导的深度神经网络——DNN 技术在图像识别领域取得了惊人的成绩，在 ImageNet 评测中成功的把错误率从 26%降低到了 15%。深度学习算法在世界大赛的脱颖而出，也再一次吸引了学术界和工业界对于深度学习领域的关注。
随着深度学习技术的不断进步以及数据处理能力的不断提升，2014 年，Facebook 基于深度学习技术的 DeepFace 项目，在人脸识别方面的准确率已经能达到 97%以上，跟人类识别的准确率几乎没有差别。这样的结果也再一次证明了深度学习算法在图像识别方面的一骑绝尘。
2016 年，随着谷歌公司基于深度学习开发的 AlphaGo 以 4:1 的比分战胜了国际顶尖围棋高手李世石，深度学习的热度一时无两。后来，AlphaGo 又接连和众多世界级围棋高手过招，均取得了完胜。这也证明了在围棋界，基于深度学习技术的机器人已经超越了人类。
2017 年，基于强化学习算法的 AlphaGo 升级版 AlphaGo Zero 横空出世。其采用“从零开始”、“无师自通”的学习模式，以 100:0 的比分轻而易举打败了之前的 AlphaGo。除了围棋，它还精通国际象棋等其它棋类游戏，可以说是真正的棋类“天才”。此外在这一年，深度学习的相关算法在医疗、金融、艺术、无人驾驶等多个领域均取得了显著的成果。所以，也有专家把 2017 年看作是深度学习甚至是人工智能发展最为突飞猛进的一年。
所以在深度学习的浪潮之下，不管是 AI 的相关从业者还是其他各行各业的工作者，都应该以开放、学习的心态关注深度学习、人工智能的热点动态。人工智能正在悄无声息的改变着我们的生活！

[卷积神经网络概述-机器学习集训营手把手教你从入门到精通卷积神经网络]

卷积神经网络

图像识别问题和数据集

> 计算机视觉中有哪些问题？典型问题：经典数据集。
在 2012 年的 ILSVRC 比赛中 Hinton 的学生 Alex Krizhevsky 使用深度卷积神经网络模型 AlexNet 以显著的优势赢得了比赛，top-5 的错误率降低至了 16.4% ，相比第二名的成绩 26.2% 错误率有了巨大的提升。AlexNet 再一次吸引了广大研究人员对于卷积神经网络的兴趣，激发了卷积神经网络在研究和工业中更为广泛的应用。现在基于卷积神经网络计算机视觉还广泛的应用于医学图像处理，人脸识别，自动驾驶等领域。越来越多的人开始了解卷积神经网络相关的技术，并且希望学习和掌握相关技术。因为卷积神经网络需要大量的标记数据集，有一些经典的数据集可以用来学习，同时解决一些常见的计算机视觉问题。

- 卷积神经网络的具体应用，经典数据集。

比如最常用的 mnist 手写数字数据集，这个数据集有 60000 个训练样本，10000 个测试样本；cfair 10 数据集包含 60000 个 32x32 像素 的彩色图片，它们分别属于 10 个类别，每一个类别有 6000 个图片，其中 50000 个作为训练集，10000 个作为测试集。

- 卷积神经网络在这些应用上取得的成果。

针对 mnist 手写数字数据集，现在已经达到了 99% 以上的识别率，在稍后的学习中，也会实现一个准确率达到 99% 以上的模型。

卷积神经网络简介

> 卷积神经网络是什么，以及卷积神经网络将如何解决计算机视觉的相关问题。

图像数据集的特点，对于神经网络的设计提出了一些新的挑战。

维度比较高

因为图像的维度普遍比较高，例如 MNIST 数据集，每一个图片是 28 * 28 的图片。

如果直接用神经网络，假设采用 2 个 1000 个神经元的隐藏层加 1 个 10 个神经元的隐藏层，最后使用 softmax 分类层，输出 10 个数字对应的概率。

参数的数量有：

786 * 1000 * 1000 * 10

如果是更大一点的图片，网络的规模还会进一步快速的增长。为了应对这种问题，

Yann LeCun 在贝尔实验室做研究员的时候提出了卷积网络技术，并展示如何使用它来大幅度提高手写识别能力。接下来将介绍卷积和池化以及卷积神经网络。

卷积介绍

我们尝试用一个简单的神经网络，来探讨如何解决这个问题。假设有 4 个输入节点和 4 个隐藏层节点的神经网络，如图所示：

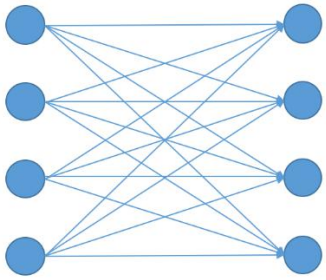


图 1 全连接神经网络

每一个输入节点都要和隐藏层的 4 个节点连接，每一个连接需要一个权重参数 w：

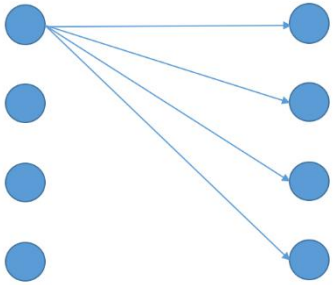


图 2 一个输入节点向下一层传播

一共有 4 个输入节点，，所以一共需要 4*4=16 个参数。

相应的每一个隐藏层节点，都会接收所有输入层节点：

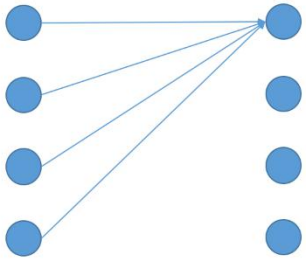


图 3 每个隐藏层节点接收所有输入层节点输入

这是一个简化版的模型，例如手写数据集 MNIST 28 * 28 的图片，输入节点有 784 个，假如也只要一个隐藏层有 784 个节点，那么参数的个数都会是：784 * 784=614656，很明显参数的个数随着输入维度指数级增长。

因为神经网络中的参数过多，会造成训练中的困难，所以降低神经网络中参数的规模，是图像处理问题中的一个重要问题。

有两个思路可以进行尝试：

1.隐藏层的节点并不需要连接所有输入层节点，而只需要连接部分输入层。

如图所示：

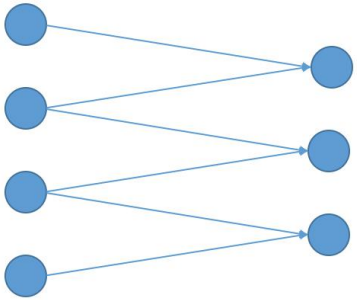


图 4 改为局部连接之后的网络结构

每个隐藏层节点，只接受两个输入层节点的输入，那么，这个网络只需要 3 * 2 =6 个连接。使用局部连接之后，单个输出层节点虽然没有连接到所有的隐藏层节点，但是隐藏层汇总之后所有的输出节点的值都对网络有影响。

2.局部连接的权重参数，如果可以共享，那么网络中参数规模又会明显的下降。如果把局部连接的权重参数当做是一个特征提取器的话，可以尝试将这个特征提取器用在其他的地方。那么这个网络最后只需要 2 个参数，就可以完成输入层节点和隐藏层节点的连接。

这两个思路就是卷积神经网络中的稀疏交互和权值共享，下一篇文章将会详细讲解卷积神经网络的原理以及使用 TensorFlow 实现。

Tensorflow 初级教程

初步介绍

Google 于 2011 年推出人工深度学习系统——DistBelief。通过 DistBelief，Google 能够扫描数据中心数以千计的核心，并建立更大的神经网络。Google 的这个系统将 Google 应用中的语音识别率提高了 25%，以及在 Google Photos 中建立了图片搜索，并驱动了 Google 的图片字幕匹配实验。但它很难被设置，没有开源。

2015 年 11 月，第二代分布式机器学习系统 Tensorflow 在 github 上开源！

2016 年 4 月，发布了分布式版本！

2017 年 1 月，发布了 1.0 预览版。API 接口趋于稳定。同年 3 月发布了可用于工业级场景的 1.0 正式版！

Tensorflow 优点：

1.既是一个实行机器学习算法的接口，同时也是执行机器学习算法的框架

2.Tensorflow 前端支持 python,C++,GO,java 等多种开发语言

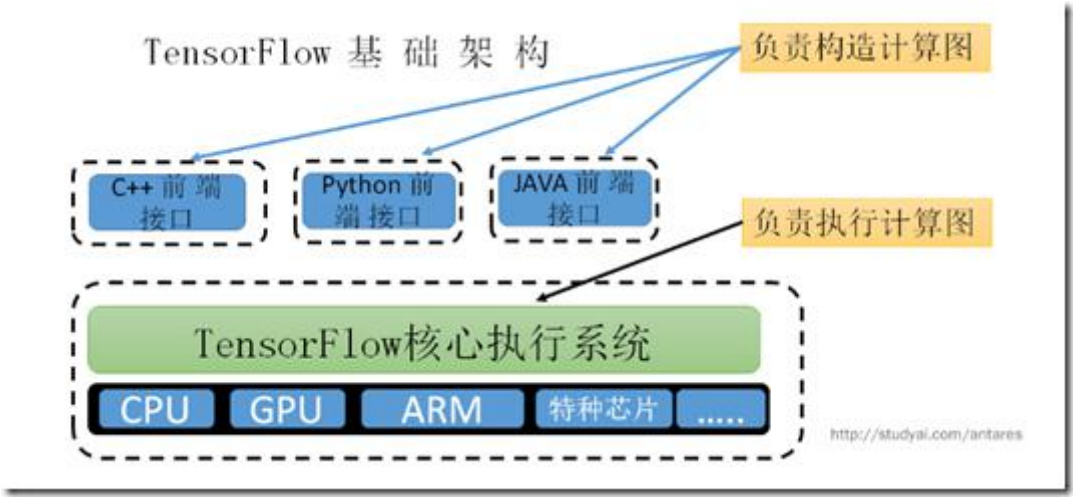
3.Tensorflow 后端使用 C++,CUDA 等写成

4.Tensorflow 可以在众多异构系统平台上移植：Android，iphone,TV，普通的 CPU 服务器以及大规模 GPU 集群等

5.Tensorflow 除了可以执行深度学习算法，还可以用来实现很多其他算法：线性回归，随机森林，支持向量机

6.Tensorflow 可以建立大规模深度学习模型的应用场景：语音识别，自然语言处理，机器视觉，机器人控制，信息抽取，医药研发等等。

Tensorflow 基础架构



前端负责构造计算图，后端负责执行计算图。就像是前端是建筑专业，后端是土木专业。

Tensorflow 核心概念

1.计算图

2.操作

3.变量

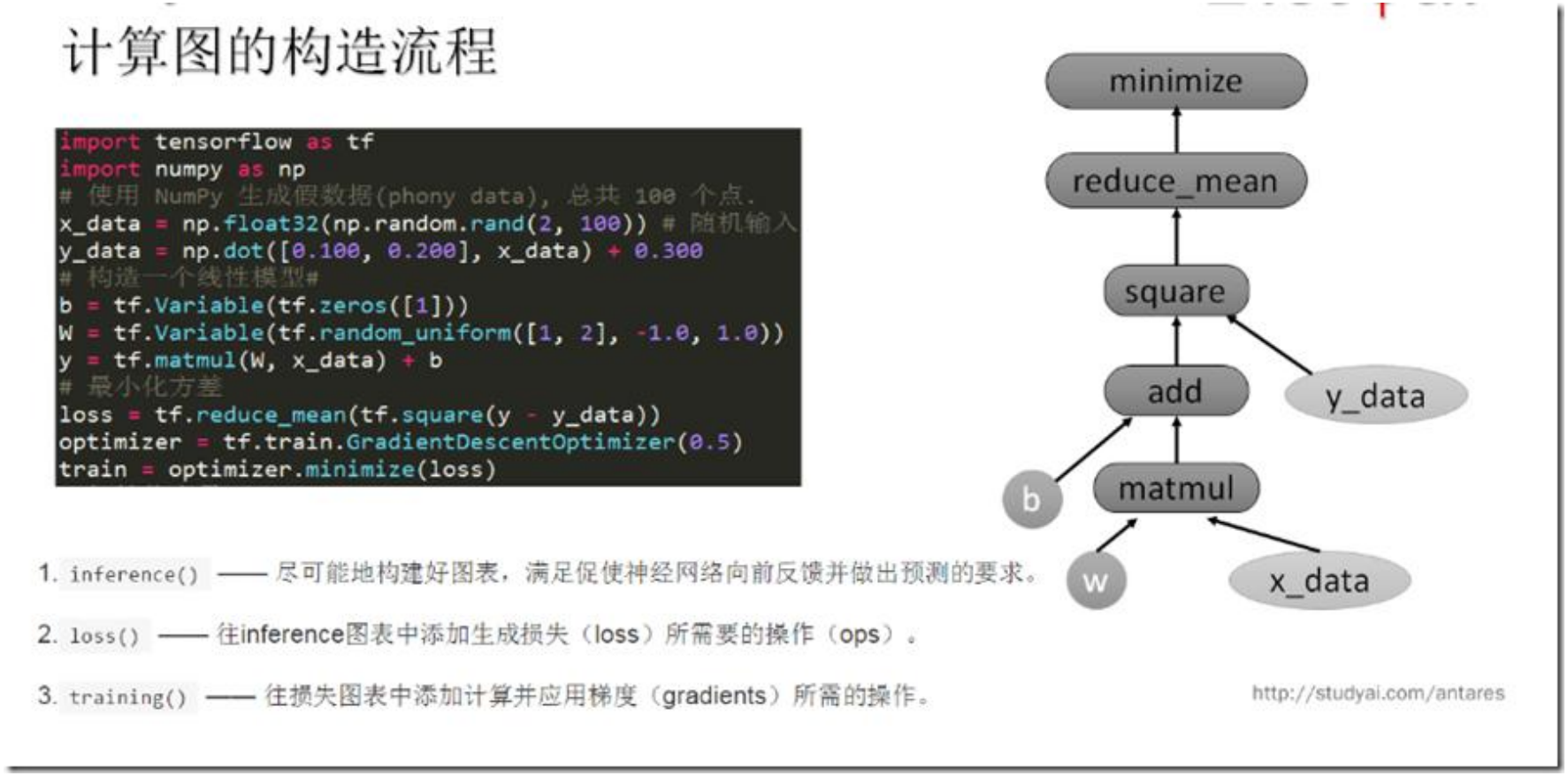
4 会话

一、计算图

计算图又被称为有向图，数据流图。数据流图用结点（nodes）和线（edges）的有向图来描述数学计算，结点一般用来表示施加的数学操作，但也可以表示数据输入的起点或者输出终点，线表示结点之间输入/输出的关系，这些数据“线”可以运输 size 可动态调整的多维数据数组，即张量（tensor）。

(注：有一类特殊边中没有数据流动，这种边被称为依赖控制，作用是控制结点的执行顺序，它可以让起始节点执行完毕再去执行目标节点，用户可以使用这样的边进行灵活控制，比如限制内存使用的最高峰值)

例：



二、操作

一个运算操作代表了一种类型的抽象运算，比如矩阵乘法或向量加法

一个运算操作可以有自己的属性，但是所有属性都必须被预先设置，或者能够在创建计算图时根据上下文推断出来

通过设置运算操作的属性可以用来支持不同的 tensor 元素类型，比如让向量加法支持浮点数或者整数

运算核（kernel）是一个运算操作在某个具体的硬件（cpu 或 gpu）的实现

在 Tensorflow 中可以通过注册机制加入新的运算操作或者为已有的运算操作添加新的计算核

TensorFlow 内建的运算操作

运算类型	运算示例
标量运算	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal
向量运算	Concat, Slice, Split, Constant, Rank, Shape, Shuffle
矩阵运算	MatMul, MatrixInverse, MatrixDeterminant
带状态的运算	Variable, Assign, AssignAdd
神经网络组件	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPooling
储存，恢复	Save, Restore
队列以及同步运算	Enqueue, Dequeue, MutexAcquire, MutexRelease
控制流	Merge, Switch, Enter, Leave, NextIteration

三、变量

当训练模型时，用变量来存储和更新参数。变量包含张量（tensor）存放在内存的缓存区。建模时，他们需要明确的初始化，模型训练后他们必须要被存储到磁盘。这些变量的值可在之后模型训练和分析时被加载。

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，初始值是常量或者随机值

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

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

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

1.一个 variable 操作存放变量的值

2.一个初始化 op 将变量设置为初始值。这事实上是一个 tf.assign 操作

3.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

变量的初始化

一次性全部初始化

变量的初始化必须在模型的其他操作运行之前先明确地完成。最简单的方法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。

```
# 初始化变量
init = tf.initialize_all_variables()
# 启动图 (graph)
sess = tf.Session()
sess.run(init)
```

使用 tf.initialize_all_variables()台南佳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。

自定义初始化

tf.initialize_all_variables()函数便捷地添加一个 op 来初始化模型地所有变量。你也可以给它出入一组变量进行初始化。

由另一个变量初始化

你有时候会需要用另一个变量地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，所以再有这种需求地情况下需要小心。

用其他变量地值初始化一个新的变量时，使用其他变量的 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，或者把它当作 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")
```

四、会话

完成全部的构建准备、生成全部所需的操作之后，我们就可以建立一个 tf.Session，用于运行图标

sess=tf.Session()

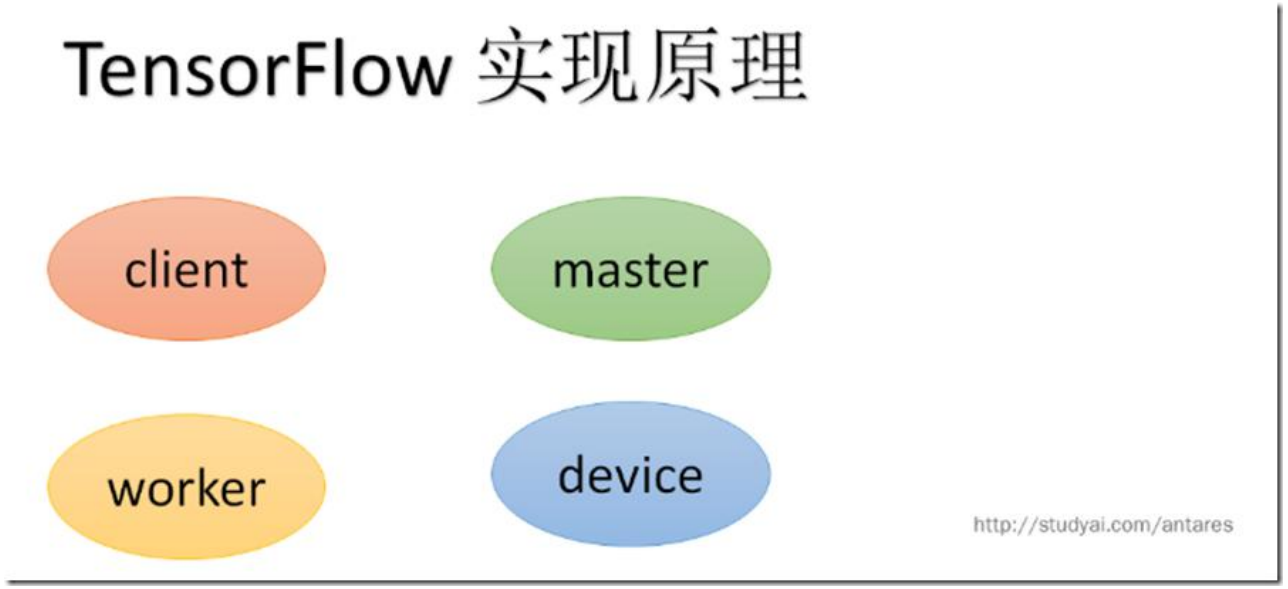
另外，也可以利用 with 代码生成 Session,限制作用域：

with tf.Session() as sess:

Session 函数没有传入参数，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。

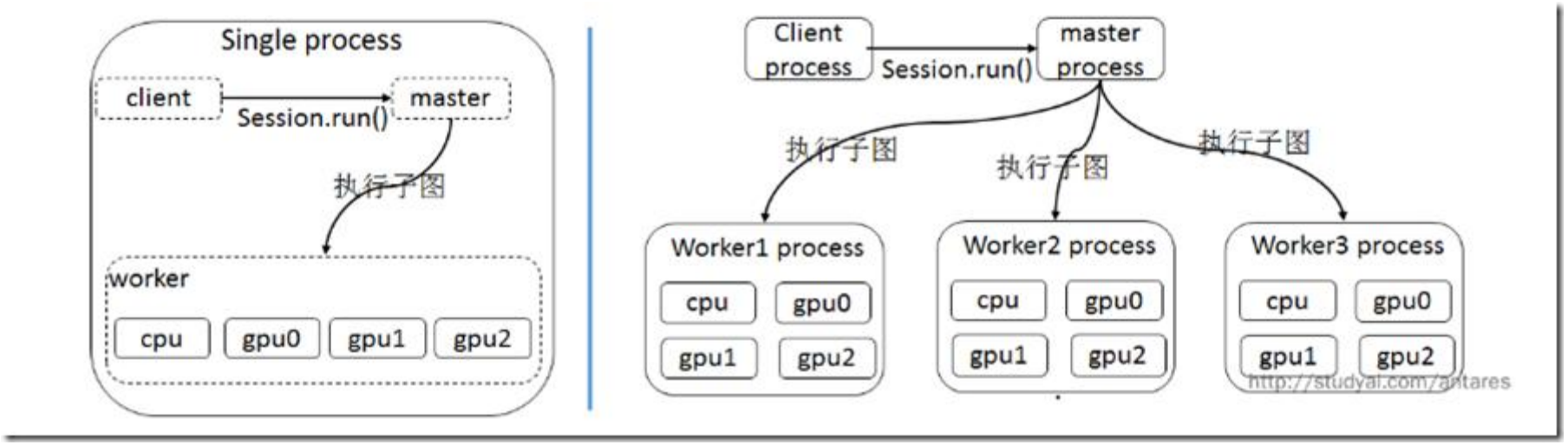
生成会话之后，所有 tf.Variable 实例都会立即通过调用各自初始化操作中的 sess.run()函数进行初始化

Tensorflow 实现原理



TensorFlow 有一个重要组件 client，也就是客户端，它通过 Session 的接口与 master 以及多个 worker 相连接。每一个 worker 可以与多个硬件设备连接，比如 cpu 和 GPU,并负责管理这些硬件。Master 则负责指导所有的 worker 按照流程执行计算图。

Tensorflow 有单机模式和分布式模式。单机模式下，client,master,worker 全部在同一台计算机上的同一个进程中。分布式模式允许 client,master,worker 在不同机器的不同进程中，同时由集群调度系统统一管理各项任务



TensorFlow 中每一个 worker 可以管理多个设备，每一个设备的 name 包含硬件类别、编号、任务号（单机版本没有）。

TensorFlow 为 CPU 和 GPU 提供管理设备的对象接口，每一个对象负责分配、释放设备内存，以及执行节点的运算核。TensorFlow 中的 Tensor 是多维数组，数据类型支持 8 位至 64 位的 int，以及 IEEE 标准的 float.double.complex.string。

在只有一个硬件设备的情况下，计算图会按照依赖关系被顺序执行。当一个节点的所有上游依赖全部执行完毕（依赖数==0）,这个节点就会被加入就绪队列（ready queen）以等待执行。同时这个节点的下游节点的依赖数自动减 1。

当多个设备的时候，情况就变得复杂了。主要有两大难点：

（1）每一个节点该让什么设备执行？

TensorFlow 设计了一套为节点分配设备的策略。这个策略首先需要计算一个代价模型。代价模型首先估算每一个节点的输入，输出 Tensor 的大小，以及所需要的计算时间。代价模型一部分由人工经验指定的启发式规则得到，另一部分则是对一小部分数据进行实际运算测量得到。

接下来，分配策略会模拟执行整个计算图，从起点开始，按照拓扑序执行。并在执行一个节点时，会把每一个能执行这个节点的设备都测试一遍，测试内容包括计算时间的估算以及数据传递所需要的通信时间。最后选择一个综合时间最短的设备计算相应的节点。这是一个简单的贪婪策略。

除了运算时间，内存的最高使用峰值也会被考虑进来。

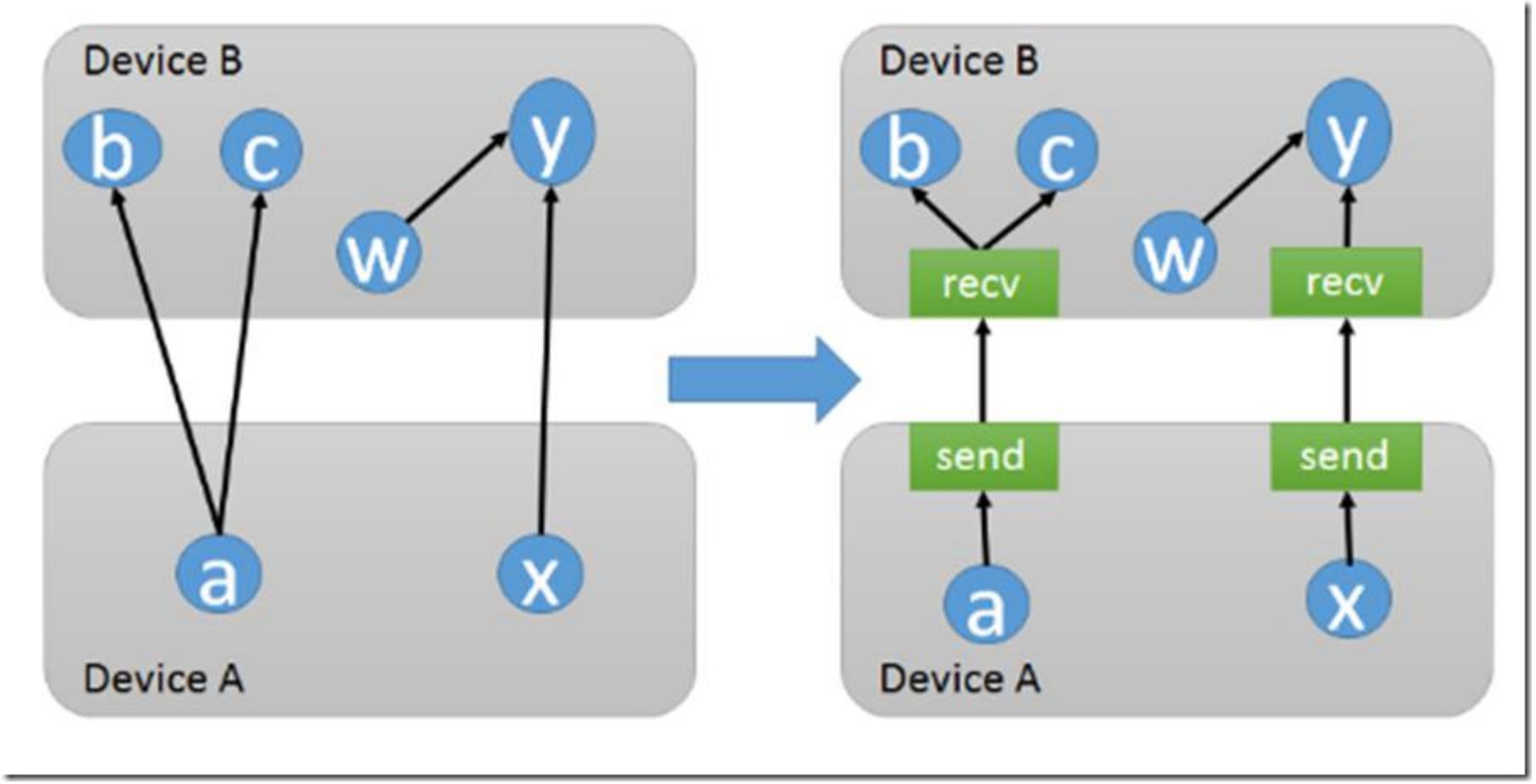
Tensor flow 的节点分配策略仍在不断优化改进。未来，可能会用一个强化学习的神经网络来辅助节点分配。同时，用户可以自定义某些分配限制条件。

（2）如何管理节点间的数据通信？

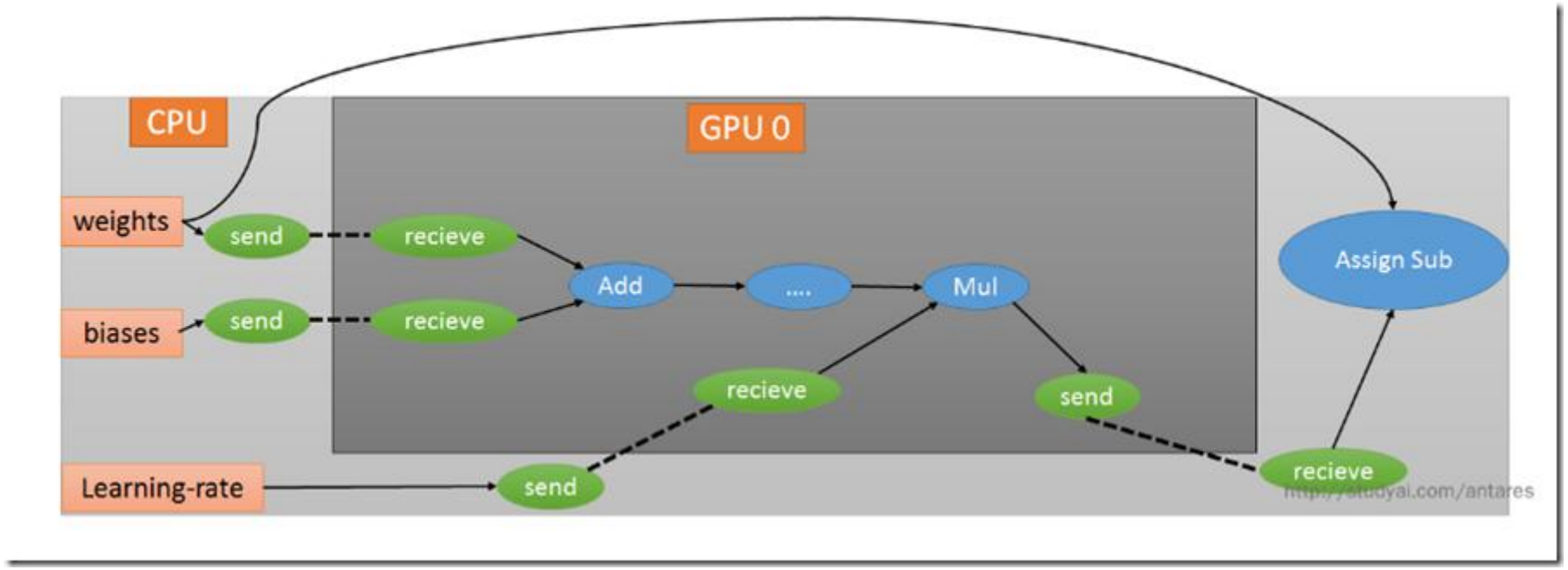
当给节点分配设备的方案被确定，整个计算图就会被划分为许多子图了，使用同一个设备并且相邻的节点会被划分到同一个子图。然后计算图中从 x 到 y 的边，会被取代为一个发送端的发送节点（send node）,一个接收端的接受节点（receive node），以及从发送节点到接受节点的边。

把数据通信的问题转换为发送节点和接受节点的实现问题，用户不需要为不同的硬件环境实现通信方法。

两个子图之间可能会有多个接受节点，如果这些接受节点接受的都是同一个 tensor，那么所有这些接受节点会被自动合并为一个，避免了数据的反复传递和设备内存占用。



发送节点和接受节点的设计简化了底层的通信模式，用户无需设计节点之间的通信流程，可以让同一套代码自动扩展到不同的硬件环境并处理复杂的通信流程。



从单机单设备的版本改造为单机多设备的版本也比较容易。下面的代码只添加了一行，就实现了从一块 GPU 到多块 GPU 训练的改造

```
for i in range(8):
    for d in range(4):
        with tf.device("/gpu:%d" % d):
            input = x[i] if d is 0 else m[d-1]
            m[d],c[d] = LSTMCell(input,mprev[d],cprev[d])
            mprev[d] = m[d]
            cprev[d] = c[d]
```

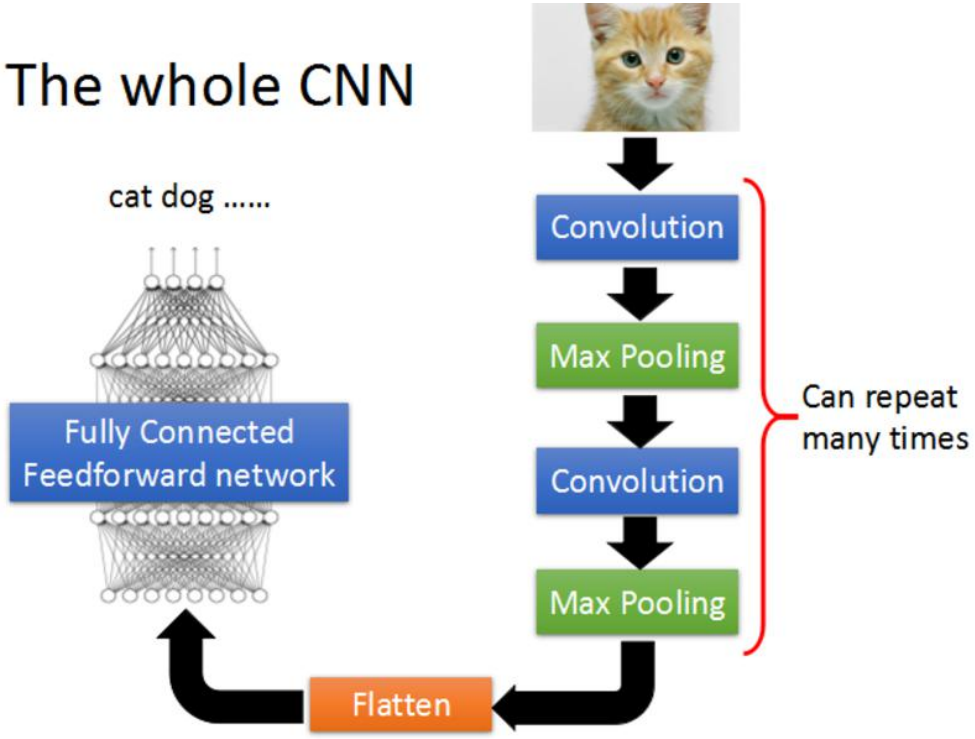
参考文献：
http://www.studyai.com/article/a187469c758d4a2884c0fa733fdfe899

分类：Tensorflow

TensorFlow 实现 CNN

TensorFlow 是目前深度学习最流行的框架，很有学习的必要，下面我们就来实际动手，使用 TensorFlow 搭建一个简单的 CNN，来对经典的 mnist 数据集进行数字识别。

如果对 CNN 还不是很熟悉的朋友，可以参考：Convolutional Neural Network。



下面就开始。

step 0 导入 TensorFlow

```
1 import tensorflow as tf
2 from tensorflow.examples.tutorials.mnist import input_data
```

step 1 加载数据集 mnist

声明两个 placeholder，用于存储神经网络的输入，输入包括 image 和 label。这里加载的 image 是(784,)的 shape。

```
mnist = input_data.read_data_sets('MNIST_data/', one_hot=True)
x = tf.placeholder(tf.float32,[None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
```

step 2 定义 weights 和 bias

为了使代码整洁，这里把 weight 和 bias 的初始化封装成函数。

```
#----Weight Initialization---#
#One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients
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)
```

step 3 定义卷积层和 maxpooling

同样，为了代码的整洁，将卷积层和 maxpooling 封装起来。padding='SAME'表示使用 padding，不改变图片的大小。

```
#Convolution and Pooling
#Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input.
#Our pooling is plain old max pooling over 2x2 blocks
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')
```

step 4 reshape image 数据

为了神经网络的 layer 可以使用 image 数据，我们要将其转化成 4d 的 tensor: (Number, width, height, channels)

```
#To apply the layer, we first reshape x to a 4d tensor, with the second and third dimensions corresponding to image width and height,
#and the final dimension corresponding to the number of color channels.
x_image = tf.reshape(x, [-1,28,28,1])
```

下面我们就要开始搭建 CNN 结构了。

step 5 搭建第一个卷积层

使用 32 个 5x5 的 filter，然后通过 maxpooling。

```
#----first convolution layer----#
#he convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of [5, 5, 1, 32].
#The first two dimensions are the patch size,
#the next is the number of input channels, and the last is the number of output channels.
W_conv1 = weight_variable([5,5,1,32])

#We will also have a bias vector with a component for each output channel.
b_conv1 = bias_variable([32])

#We then convolve x_image with the weight tensor, add the bias, apply the ReLU function, and finally max pool.
#The max_pool_2x2 method will reduce the image size to 14x14.
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

step 6 第二层卷积

使用 64 个 5x5 的 filter。

```
#----second convolution layer----#
#The second layer will have 64 features for each 5x5 patch and input size 32.
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)
```

step 7 构建全链接层

需要将上一层的输出，展开成 1d 的神经层。

```
#----fully connected layer----#
#Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image
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)
```

step 8 添加 Dropout

加入 Dropout 层，可以防止过拟合问题。注意，这里使用了另外一个 placeholder，可以控制在训练和预测时是否使用 Dropout。

```
#-----dropout------#
#To reduce overfitting, we will apply dropout before the readout layer.
#We create a placeholder for the probability that a neuron's output is kept during dropout.
#This allows us to turn dropout on during training, and turn it off during testing.
keep_prob = tf.placeholder(tf.float32)
h_fc1_dropout = tf.nn.dropout(h_fc1, keep_prob)
```

step 9 输入层

没有什么特别的，就是输出一个线性结果。

```
#----read out layer----#
W_fc2 = weight_variable([1024,10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1_dropout, W_fc2) + b_fc2
```

step 10 训练和评估

首先，需要指定一个 cost function --cross_entropy，在输出层使用 softmax。然后指定 optimizer--adam。需要特别指出的是，一定要记得

tf.global_variables_initializer().run()初始化变量

```
#------train and evaluate----#
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_, 1), tf.argmax(y_conv, 1)), tf.float32))
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(3000):
        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.})
            print('setp {},the train accuracy: {}'.format(i, train_accuracy))
        train_step.run(feed_dict = {x: batch[0], y_: batch[1], keep_prob: 0.5})
    test_accuracy = accuracy.eval(feed_dict = {x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.})
    print('the test accuracy :{}'.format(test_accuracy))
    saver = tf.train.Saver()
    path = saver.save(sess, './my_net/mnist_deep.ckpt')
    print('save path: {}'.format(path))
```

这是我训练的结果。

```
setp 0, the train accuracy: 0.05999999865889549
setp 100, the train accuracy: 0.8399999737739563
setp 200, the train accuracy: 0.8399999737739563
setp 300, the train accuracy: 0.9399999976158142
setp 400, the train accuracy: 0.8999999761581421
setp 500, the train accuracy: 0.9399999976158142
setp 600, the train accuracy: 0.9599999785423279
setp 700, the train accuracy: 0.9599999785423279
setp 800, the train accuracy: 0.8999999761581421
setp 900, the train accuracy: 0.9599999785423279
setp 1000, the train accuracy: 0.9800000190734863
setp 1100, the train accuracy: 0.9399999976158142
setp 1200, the train accuracy: 0.9599999785423279
setp 1300, the train accuracy: 0.9399999976158142
setp 1400, the train accuracy: 0.9800000190734863
setp 1500, the train accuracy: 0.9599999785423279
setp 1600, the train accuracy: 0.8999999761581421
setp 1700, the train accuracy: 0.9599999785423279
setp 1800, the train accuracy: 0.9800000190734863
setp 1900, the train accuracy: 0.9399999976158142
setp 2000, the train accuracy: 0.9800000190734863
setp 2100, the train accuracy: 0.9800000190734863
setp 2200, the train accuracy: 0.9599999785423279
setp 2300, the train accuracy: 1.0
setp 2400, the train accuracy: 0.9200000166893005
setp 2500, the train accuracy: 1.0
setp 2600, the train accuracy: 1.0
setp 2700, the train accuracy: 1.0
setp 2800, the train accuracy: 1.0
setp 2900, the train accuracy: 0.9800000190734863
the test accuracy :0.9824000000953674
save path: ./my_net/mnist_deep.ckpt
```

reference：

https://www.tensorflow.org/get_started/mnist/pros

Tensorflow 从入门到精通之——Tensorflow 基本操作

前边的章节介绍了什么是 Tensorflow，本节将带大家真正走进 Tensorflow 的世界，学习 Tensorflow 一些基本的操作及使用方法。同时也欢迎大家关注我们的网站和系列教程：http://www.tensorflownews.com/，学习更多的机器学习、深度学习的知识！

Tensorflow 是一种计算图模型，即用图的形式来表示运算过程的一种模型。Tensorflow 程序一般分为图的构建和图的执行两个阶段。图的构建阶段也称为图的定义阶段，该过程会在图模型中定义所需的运算，每次运算的的结果以及原始的输入数据都可称为一个节点（operation ，缩写为 op）。我们通过以下程序来说明图的构建过程：

程序 2-1：

```
import tensorflow as tf
m1 = tf.constant([3,5])
m2 = tf.constant([2,4])
result = tf.add(m1,m2)
print(result)

>> Tensor("Add_1:0", shape=(2,), dtype=int32)
```

程序 2-1 定义了图的构建过程，“import tensorflow as tf”，是在 python 中导入 tensorflow 模块,并另起名为“tf”；接着定义了两个常量 op，m1 和 m2，均为 1*2 的矩阵；最后将 m1 和 m2 的值作为输入创建一个矩阵加法 op，并输出最后的结果 result。
我们分析最终的输出结果可知，其并没有输出矩阵相加的结果，而是输出了一个包含三个属性的 Tensor(Tensor 的概念我们会在下一节中详细讲解，这里就不再赘述)。
以上过程便是图模型的构建阶段：只在图中定义所需要的运算，而没有去执行运算。我们可以用图 2-1 来表示：

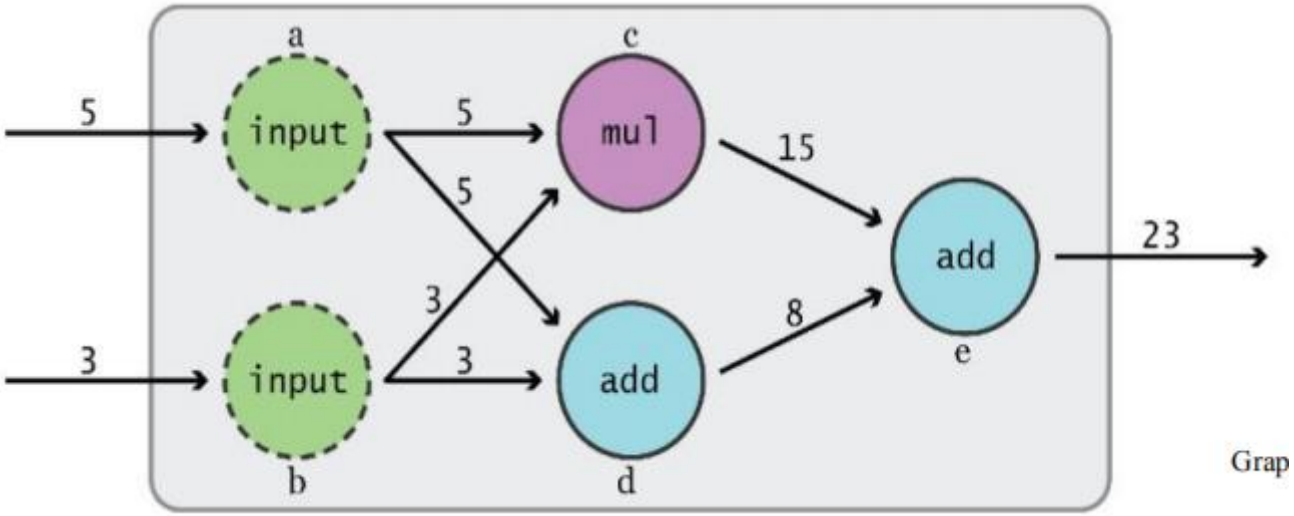


图 2-1 图的构建阶段

第二个阶段为图的执行阶段，也就是在会话（session）中执行图模型中定义好的运算。

我们通过程序 2-2 来解释图的执行阶段：

程序 2-2：

```
sess = tf.Session()
print(sess.run(result))
sess.close()


>> [5,9]
```

程序 2-2 描述了图的执行过程，首先通过“tf.session()”启动默认图模型，再调用 run()方法启动、运行图模型，传入上述参数 result，执行矩阵的加法，并打印出相加的结果，最后在任务完成时，要记得调用 close()方法，关闭会话。

除了上述的 session 写法外，我们更建议大家，把 session 写成如程序 2-4 所示“with”代码块的形式，这样就无需显示的调用 close 释放资源，而是自动地关闭会话。

程序 2-3：

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

此外，我们还可以利用 CPU 或 GPU 等计算资源分布式执行图的运算过程。一般我们无需显示的指定计算资源，Tensorflow 可以自动地进行识别，如果检测到我们的 GPU 环境，会优先的利用 GPU 环境执行我们的程序。但如果我们的计算机中有多于一个可用的 GPU，这就需要我们手动的指派 GPU 去执行特定的 op。如下程序 2-4 所示，Tensorflow 中使用 with...device 语句来指定 GPU 或 CPU 资源执行操作。

程序 2-4：

```
with tf.Session() as sess:
    with tf.device("/gpu:2"):
        m1 = tf.constant([3,5])
        m2 = tf.constant([2,4])
        result = tf.add(m1,m2)
```

上述程序中的“tf.device(“/gpu:2”)”是指定了第二个 GPU 资源来运行下面的 op。依次类推，我们还可以通过“/gpu:3”、“/gpu:4”、“/gpu:5”...来指定第 N 个 GPU 执行操作。

关于 GPU 的具体使用方法，我们会在下面的章节结合案例的形式具体描述。

Tensorflow 中还提供了默认会话的机制，如程序 2-5 所示，我们通过调用函数 as_default()生成默认会话。

程序 2-5：

```
import tensorflow as tf
m1 = tf.constant([3, 5])
m2 = tf.constant([2, 4])
result = tf.add(m1,m2)
sess = tf.Session()
with sess.as_default():
    print(result.eval())

>> [5,9]
```

我们可以看到程序 2-5 和程序 2-2 有相同的输出结果。我们在启动默认会话后，可以通过调用 eval()函数，直接输出变量的内容。
有时，我们需要在 Jupyter 或 IPython 等 python 交互式环境开发。Tensorflow 为了满足用户的这一需求，提供了一种专门针对交互式环境开发的方法 InteractiveSession(),具体用法如程序 2-6 所示：
程序 2-6：

```
import tensorflow as tf
m1 = tf.constant([3, 5])
m2 = tf.constant([2, 4])
result = tf.add(m1,m2)
sess = tf.InteractiveSession()
print(result.eval())

>>  [5,9]
```

程序 2-6 就是交互式环境中经常会使用的 InteractiveSession()方法，其创建 sess 对象后，可以直接输出运算结果。
综上所述，我们介绍了 Tensorflow 的核心概念——计算图模型，以及定义图模型和运行图模型的几种方式。接下来，我们思考一个问题，为什么 Tensorflow 要使用图模型？图模型有什么优势呢？
首先，图模型的最大好处是节约系统开销，提高资源的利用率，可以更加高效的进行运算。因为我们在图的执行阶段，只需要运行我们需要的 op,这样就大大的提高了资源的利用率；其次，这种结构有利于我们提取中间某些节点的结果，方便以后利用中间的节点去进行其它运算；还有就是这种结构对分布式运算更加友好，运算的过程可以分配给多个 CPU 或是 GPU 同时进行，提高运算效率；最后，因为图模型把运算分解成了很多个子环节，所以这种结构也让我们的求导变得更加方便。

2.3.2 Tensor 介绍

Tensor（张量）是 Tensorflow 中最重要的数据结构，用来表示 Tensorflow 程序中的所有数据。Tensor 本是广泛应用在物理、数学领域中的一个物理量。那么在 Tensorflow 中该如何理解 Tensor 的概念呢？

实际上，我们可以把 Tensor 理解成 N 维矩阵（N 维数组）。其中零维张量表示的是一个标量，也就是一个数；一维张量表示的是一个向量，也可以看作是一个一维数组；二维张量表示的是一个矩阵；同理，N 维张量也就是 N 维矩阵。

在计算图模型中，操作间所传递的数据都可以看做是 Tensor。那 Tensor 的结构到底是怎样的呢？我们可以通过程序 2-7 更深入的了解一下 Tensor。

程序 2-7：

```
#导入 tensorflow 模块
import tensorflow as tf
a = tf.constant([[2.0,3.0]] ,name="a")
b = tf.constant([[1.0],[4.0]] ,name="b")
result = tf.matmul(a,b,name="mul")
print(result)

>> Tensor("mul_3:0", shape=(1, 1), dtype=float32)
```

程序 2-7 的输出结果表明：构建图的运算过程输出的结果是一个 Tensor，且其主要由三个属性构成：Name、Shape 和 Type。Name 代表的是张量的名字，也是张量的唯一标识符，我们可以在每个 op 上添加 name 属性来对节点进行命名，Name 的值表示的是该张量来自于第几个输出结果（编号从 0 开始），上例中的"mul_3:0"说明是第一个结果的输出。Shape 代表的是张量的维度，上例中 shape 的输出结果(1,1)说明该张量 result 是一个二维数组，且每个维度数组的长度是 1。最后一个属性表示的是张量的类型，每个张量都会有唯一的类型，常见的张量类型如图 2-2 所示。

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_UINT16	tf.uint16	16 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

图 2-2 常用的张量类型

我们需要注意的是要保证参与运算的张量类型相一致，否则会出现类型不匹配的错误。如程序 2-8 所示，当参与运算的张量类型不同时，Tensorflow 会报类型不匹配的错误：

程序 2-8：

```
import tensorflow as tf
m1 = tf.constant([5,1])
m2 = tf.constant([2.0,4.0])
result = tf.add(m1,m2)

TypeError: Input 'y' of 'Add' Op has type float32 that does not match type int32 of arg
ument 'x'.
```

正如程序的报错所示：m1 是 int32 的数据类型，而 m2 是 float32 的数据类型，两者的数据类型不匹配，所以发生了错误。所以我们在实际编程时，一定注意参与运算的张量数据类型要相同。

2.3.3 常量、变量及占位符

Tensorflow 中对常量的初始化，不管是对数值、向量还是对矩阵的初始化，都是通过调用 constant()函数实现的。因为 constant()函数在 Tensorflow 中的使用非常频繁，经常被用于构建图模型中常量的定义，所以接下来，我们通过程序 2-9 了解一下 constant()的相关属性：

程序 2-9：

```
import tensorflow as tf
a = tf.constant([2.0,3.0],name="a",shape=(2,0),dtype="float64",verify_shape="true")
print(a)

>> Tensor("a_11:0", shape=(2, 0), dtype=float64)
```

如程序 2-9 所示，函数 constant 有五个参数，分别为 value，name，dtype，shape 和 verify_shape。其中 value 为必选参数，其它均为可选参数。Value 为常量的具体值，可以是一个数字，一维向量或是多维矩阵。Name 是常量的名字，用于区别其它常量。Dtype 是常量的类型，具体类型可参见图 2-2。Shape 是指常量的维度，我们可以自行定义常量的维度。

verify_shape 是验证 shape 是否正确，默认值为关闭状态(False)。也就是说当该参数 true 状态时，就会检测我们所写的参数 shape 是否与 value 的真实 shape 一致，若不一致就会报 TypeError 错误。如：上例中的实际 shape 为(2,0)，若我们将参数中的 shape 属性改为(2,1)，程序就会报如下错误：

TypeError: Expected Tensor's shape: (2, 1), got (2,).

Tensorflow 还提供了一些常见常量的初始化，如：tf.zeros、tf.ones、tf.fill、tf.linspace、tf.range 等，均可以快速初始化一些常量。例如：我们想要快速初始化 N 维全 0 的矩阵，我们可以利用 tf.zeros 进行初始化，如程序 2-10 所示：

程序 2-10：

```
import tensorflow as tf
a=tf.zeros([2,2],tf.float32)
b=tf.zeros_like(a,optimize=True)
with tf.Session() as sess:
    print(sess.run(a))
    print(sess.run(b))

    >> [[ 0.  0.]
     [ 0.  0.]]
    [[ 0.  0.]
     [ 0.  0.]]
```

程序 2-10 向我们展示了 tf.zeros 和 tf.zeros_like 的用法。其它常见常量的具体初始化用法可以参考 Tensorflow 官方手册：https://www.tensorflow.org/api_guides/python/constant_op。

此外，Tensorflow 还可以生成一些随机的张量，方便快速初始化一些随机值。如：tf.random_normal()、tf.truncated_normal()、tf.random_uniform()、tf.random_shuffle()等。如程序 2-11 所示，我们以 tf.random_normal()为例，来看一下随机张量的具体用法：

程序 2-11：

```
import tensorflow as tf
random_num=tf.random_normal([2, 3], mean=-1, stddev=4,
                            dtype=tf.float32,seed=None,name='rnum')
with tf.Session() as sess:
    print(sess.run(random_num))

>> [[-2.71897316  1.04246855 -3.12996817]
    [-1.34851456 -0.13599336  4.60532522]]
```

随机张量 random_normal()有 shape、mean、stddev、dtype、seed、name 六个属性。 shape 是指张量的形状，如上述程序是生成一个 2 行 3 列的 tensor；mean 是指正态分布的均值；stddev 是指正太分布的标准差；dtype 是指生成 tensor 的数据类型；seed 是分发创建的一个随机种子；而 name 是给生成的随机张量命名。

Tensorflow 中的其它随机张量的具体使用方法和属性介绍，可以参见 Tensorflow 官方手册：https://www.tensorflow.org/api_guides/python/constant_op。这里将不在一一赘述。

除了常量 constant()，变量 variable()也是在 Tensorflow 中经常会被用到的函数。变量的作用是保存和更新参数。执行图模型时，一定要对变量进行初始化，经过初始化后的变量才能拿来使用。变量的使用包括创建、初始化、保存、加载等操作。首先，我们通过程序 2-12 了解一下变量是如何被创建的：

程序 2-12：

```
import tensorflow as tf
A = tf.Variable(3, name="number")
B = tf.Variable([1,3], name="vector")
C = tf.Variable([[0,1],[2,3]], name="matrix")
D = tf.Variable(tf.zeros([100]), name="zero")
E = tf.Variable(tf.random_normal([2,3], mean=1, stddev=2, dtype=tf.float32))
```

程序 2-12 展示了创建变量的多种方式。我们可以把函数 variable()理解为构造函数，构造函数的使用需要初始值，而这个初始值是一个任何形状、类型的 Tensor。也就是说，我们既可以通过创建数字变量、一维向量、二维矩阵初始化 Tensor，也可以使用常量或是随机常量初始化 Tensor，来完成变量的创建。

当我们完成了变量的创建，接下来，我们要对变量进行初始化。变量在使用前一定要进行初始化，且变量的初始化必须在模型的其它操作运行之前完成。通常，变量的初始化有三种方式，如程序 2-13 所示：

程序 2-13：

```
#初始化全部变量:
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
#初始化变量的子集：
init_subset=tf.variables_initializer([b,c], name="init_subset")
with tf.Session() as sess:
    sess.run(init_subset)
#初始化单个变量：
init_var = tf.Variable(tf.zeros([2,5]))
with tf.Session() as sess:
    sess.run(init_var.initializer)
```

程序 2-13 说明了初始化变量的三种方式：初始化全部变量、初始化变量的子集以及初始化单个变量。首先，global_variables_initializer()方法是不管全局有多少个变量，全部进行初始化，是最简单也是最常用的一种方式；variables_initializer()是初始化变量的子集，相比于全部初始化化的方式更加节约内存；Variable()是初始化单个变量，函数的参数便是要初始化的变量内容。通过上述的三种方式，我们便可以实现变量的初始化，放心的使用变量了。

我们经常在训练模型后，希望保存训练的结果，以便下次再使用或是方便日后查看，这时就用到了 Tensorflow 变量的保存。变量的保存是通过 tf.train.Saver()方法创建一个 Saver 管理器，来保存计算图模型中的所有变量。

具体代码如程序 2-14 所示：

程序 2-14：

```
import tensorflow as tf
var1 = tf.Variable([1,3], name="v1")
var2 = tf.Variable([2,4], name="v2")
#对全部变量进行初始化
init = tf.initialize_all_variables()
#调用 Saver()存储器方法
saver = tf.train.Saver()
#执行图模型
with tf.Session() as sess:
    sess.run(init)
    #设置存储路径
    save_path = saver.save(sess,"test/save.ckpt")
```

我们要注意，我们的存储文件 save.ckpt 是一个二进制文件，Saver 存储器提供了向该二进制文件保存变量和恢复变量的方法。保存变量的方法就是程序中的 save()方法，保存的内容是从变量名到 tensor 值的映射关系。完成该存储操作后，会在对应目录下生成如图 2-3 所示的文件：

名称	修改日期	类型	大小
checkpoint	2018/1/31 19:38	文件	1 KB
model.ckpt.data-00000-of-00001	2018/1/31 19:38	DATA-00000-OF...	614 KB
model.ckpt.index	2018/1/31 19:38	INDEX 文件	1 KB
model.ckpt.meta	2018/1/31 19:38	META 文件	5 KB

图 2-3 保存变量生成的相应文件

Saver 提供了一个内置的计数器自动为 checkpoint 文件编号。这就支持训练模型在任意步骤多次保存。此外，还可以通过 global_step 参数自行对保存文件进行编号，例如：global_step=2，则保存变量的文件夹为 model.ckpt-2。

那如何才能恢复变量呢？首先，我们要知道一定要用和保存变量相同的 Saver 对象来恢复变量。其次，不需要事先对变量进行初始化。具体代码如程序 2-15 所示：

程序 2-15：

```
import tensorflow as tf
var1 = tf.Variable([0,0], name="v1")
var2 = tf.Variable([0,0], name="v2")
saver = tf.train.Saver()
module_file = tf.train.latest_checkpoint('test/')
with tf.Session() as sess:
    saver.restore(sess,module_file)
    print("Model restored.")
```

本程序示例中，我们要注意：变量的获取是通过 restore()方法，该方法有两个参数，分别是 session 和获取变量文件的位置。我们还可以通过 latest_checkpoint()方法，获取到该目录下最近一次保存的模型。

以上就是对变量创建、初始化、保存、加载等操作的介绍。此外，还有一些与变量相关的重要函数，如：eval()等。

认识了常量和变量，Tensorflow 中还有一个非常重要的常用函数——placeholder。placeholder 是一个数据初始化的容器，它与变量最大的不同在于 placeholder 定义的是一个模板，这样我们就可以 session 运行阶段，利用 feed_dict 的字典结构给 placeholder 填充具体的内容，而无需每次都提前定义好变量的值，大大提高了代码的利用率。Placeholder 的具体用法如程序 2-16 所示：

程序序 2-16：

```
import tensorflow as tf
a = tf.placeholder(tf.float32,shape=[2],name=None)
b = tf.constant([6,4],tf.float32)
c = tf.add(a,b)
with tf.Session() as sess:
    print(sess.run(c,feed_dict={a:[10,10]}))
```

程序 2-16 演示了 placeholder 占位符的使用过程。Placeholder()方法有 dtype，shape 和 name 三个参数构成。dtype 是必填参数，代表传入 value 的数据类型；shape 是选填参数，代表传入 value 的维度；name 也是选填参数，代表传入 value 的名字。我们可以把这三个参数看作为形参，在使用时传入具体的常量值。这也是 placeholder 不同于常量的地方，它不可以直接拿来使用，而是需要用户传递常数值。

最后，Tensorflow 中还有一个重要的概念——fetch。Fetch 的含义是指可以在一个会话中同时运行多个 op。这就方便我们在实际的建模过程中，输出一些中间的 op，取回多个 tensor。Fetch 的具体用法如程序 2-17 所示：

程序 2-17：

```
import tensorflow as tf
a = tf.constant(5)
b = tf.constant(6)
c = tf.constant(4)
add = tf.add(b, c)
mul = tf.multiply(a, add)
with tf.Session() as sess:
    result = sess.run([mul, add])
    print(result)
```

程序 2-17 展示了 fetch 的用法，即我们利用 session 的 run()方法同时取回多个 tensor 值，方便我们查看运行过程中每一步 op 的输出结果。

程序 2-18：

```
import tensorflow as tf
var1 = tf.Variable([0,0], name="v1")
var2 = tf.Variable([0,0], name="v2")
Saver = tf.train.Saver()
module_file = tf.train.latest_checkpoint('test/')
with tf.Session() as sess:
    saver.restore(sess,module_file)
    print("Model restored.")
```

小结：本节旨在让大家学会 Tensorflow 的基础知识，为后边实战的章节打下基础。主要讲了 Tensorflow 的核心——计算图模型，如何定义图模型和计算图模型；还介绍了 Tensor 的概念，以及 Tensorflow 中的常量、变量、占位符、feed 等知识点。大家都掌握了吗？