# Tensorflow基础

对于tensorflow，有以下基本的概念需要了解：

* 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；
* 通过**供给（feeds）**和**取回（fetches）**将数据闯入或传出任何操作。

## 1. 张量和图

### 1.1 图与图运算

TensorFlow 是一种采用数据流图（data flow graphs），用于数值计算的开源软件库。其中 Tensor 代表传递的数据为张量（多维数组），Flow 代表使用计算图进行运算。数据流图用「结点」（nodes）和「边」（edges）组成的有向图来描述数学运算。「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和输出的终点，或者是读取/写入持久变量（persistent variable）的终点。边表示结点之间的输入/输出关系。这些数据边可以传送维度可动态调整的多维数据数组，即张量（tensor）。

**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。**

**Graphs是对计算的抽象描述，计算开始前，graphs必须在会话（Session）中被启动。会话将图的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，将产生的张量返回。**

**在python中，将返回numpy的ndarray对象；在c++语言中，将返回tensorflow::Tensor实例。**

1.1.1 计算图

TensorFlow编程可以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图，执行阶段利用Session中执行计算图中的op操作。

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。

为了高效地在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库件，在Python 外用其它执行效率高的语言完成这些高运算开销操作（如矩阵运算）。但是，每一步操作依然会需要切换回Python 带来很大开销。特别的，这种开销会在GPU 运算或是分布式集群运算这类高数据传输需求的运算形式上非常高昂。

TensorFlow 将高运算量计算放在Python 外进行，同时更进一步设法避免上述的额外运算开销。不同于在Python 中独立运行运算开销昂贵的操作，TensorFlow 让我们可以独立于Python 以外以图的形式描述交互式操作。

1.1.2 构建计算图

初始建立的op比如常量，不需要任何输入源，再将它们传递给其他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他op构造函数作为输入。

TensorFlowPython 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.]])**  **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)**  #默认图现在拥有三个节点，两个constant() op，一个matmul() op. |

1.1.3 在会话中载入图

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。

|  |
| --- |
| # Launch the default graph. 在会话中载入默认的图  **sess = tf.Session()**  # The output of the op is returned in 'result' as a numpy `ndarray` object.  **result = sess.run(product)**  **print(result)**  # 会话完成后必须关闭以释放资源  **sess.close()**  #也可以使用” with ”句块开始会话，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)** |

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。

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)**  **...** |

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。

1.1.4 交互式使用

考虑到如IPython这样的交互式Python 环境的易用, 可以**使用InteractiveSession 代替Session类, 使用Tensor.eval()和Operation.run() 方法代替Session.run(). 这样可以避免使用一个变量来持有会话。**

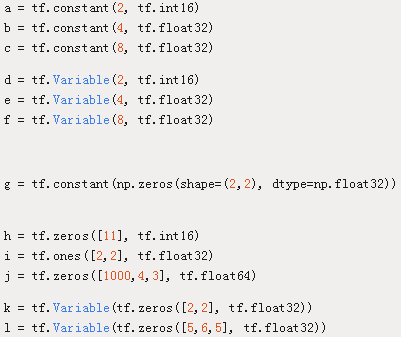
|  |
| --- |
| # Enter an interactive TensorFlow Session.  **import tensorflow as tf**  **sess = tf.InteractiveSession()**  **x = tf.Variable([1.0, 2.0])**  **a = tf.constant([3.0, 3.0])**  # Initialize 'x' using the run() method of its initializer op. **方法：operation.run()**  **x.initializer.run()**  # Add an op to subtract 'a' from 'x'. Run it and print the result **方法：tensor.eval()**  **sub = tf.sub(x, a)**  **print(sub.eval())**  # Close the Session when we're done.  **sess.close()** |

**交互会话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。**

### 1.2 常量constant和变量Variable

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 的张量看作是一个n 维的数组或列表。 一个tensor包含一个静态类型rank，和一个shape。

TensorFlow 中最基本的单位是常量（Constant）、变量（Variable）和占位符（Placeholder）。常量定义后值和维度不可变，变量定义后值可变而维度不可变。在神经网络中，变量一般可作为储存权重和其他信息的矩阵，而常量可作为储存超参数或其他结构信息的变量。

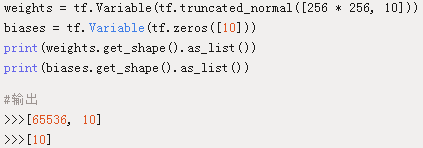


以下语句声明一个2行3列的变量矩阵，该变量的值服从标准差为1的正态分布，并随机生成。



TensorFlow还有tf.truncated\_normal()函数，即截断正态分布随机数，它只保留 [mean-2\*stddev, mean+2\*stddev] 范围内的随机数。

利用变量可以定义一层神经元个数为10，每个神经元输入为256\*256的神经网络的权重矩阵和偏置向量：



1.2.1 变量创建

下面主要描述TensorFlow的一个变量类：

**tf.Variable类**

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）到graph，返回值是Python的tf.Variable类的一个实例。

1.2.2 变量初始化、赋值初始化、自定义初始化

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。

**（1）构建好统一进行初始化**

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。

|  |
| --- |
| # Add an op to initialize the variables.  **init\_op = tf.initialize\_all\_variables()**  # Later, when launching the model # Run the init operation.  **with tf.Session() as sess:**  **sess.run(init\_op)**  ... |

**（2）由另一个变量初始化**

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() \* 2, name="w\_twice")** |

**（3）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。

1.2.3 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restoreops。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。

1. **保存**

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

系统的变量将被保存到checkpoints中，point我简单的理解成快照，其本质是tensorflow使用二进制保存的变量名到变量值的map映射文件。

新建Saver类，其init方法为：

tf.train.Saver(var\_list = None,reshape = False,shared = False,max\_to\_keep = 5,keep\_checkpoint\_every\_n\_hours = 10000.0,name = None,restore\_sequentially = False,saver\_def = None,builder = None)

简单的话，可以使用

tf.train.Saver({"v1":v1,"v2":v2})

tf.train.Saver([v1,v2])

tf.train.Saver{v.op.name for v in [v1,v2]}

从这个角度来看，Saver可以传入变量名set，变量名list，变量名：变量值的dict等类型。默认是将所有的变量都保存。

参数中重要的是max\_to\_keep 最大备份数量；keep\_checkpoint\_every\_n\_hours 每隔多少小时备份一次。

Saver类需要调用save才能真正的完成保存。调用方法为：

tf.train.Saver.save(sess,save\_path,global\_step=None,latest\_filename =None)

sess 和path 是必须传入的，global\_step 若标准，则会以此为checkpoint的文件名。latest\_filename将包含最新检查点文件名列表的协议缓冲区文件的可选名称。保存在与检查点文件相同的目录中的文件由保存程序自动管理以跟踪最近的检查点。默认为“checkpoint”。

|  |
| --- |
| # 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()**  # 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** |

**（2）加载**

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。

读取 tf.train.Saver.restore(sess, save\_path)，简单的使用sess 和path就可以读取。如果要读取数据，那么原来的变量要设置好，但是不必初始化。

|  |
| --- |
| # 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:**  **saver.restore(sess, "/tmp/model.ckpt")**  **print "Model restored."**  ... |

**（3）选择存储和恢复对应变量**

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

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

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

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

注意：

如果需要保存和恢复模型变量的不同子集，可以创建任意多个saver 对象。同一个变量可被列入多个saver 对象中，只有当saver 的restore()函数被运行时，它的值才会发生改变。

如果你仅在session 开始时恢复模型变量的一个子集，你需要对剩下的变量执行初始化op。详情请见tf.initialize\_variables()。

|  |
| --- |
| # 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. |

### 1.3 共享变量

对于一般的tf.Variable()变量，有如下定义：

|  |
| --- |
| **def my\_image\_filter(input\_images):**  **conv1\_weights = tf.Variable(tf.random\_normal([5, 5, 32, 32]),name="conv1\_weights")**  **conv1\_biases = tf.Variable(tf.zeros([32]), name="conv1\_biases")**  **conv1 = tf.nn.conv2d(input\_images, conv1\_weights,strides=[1, 1, 1, 1], padding='SAME')**  **relu1 = tf.nn.relu(conv1 + conv1\_biases)**  **conv2\_weights = tf.Variable(tf.random\_normal([5, 5, 32, 32]),name="conv2\_weights")**  **conv2\_biases = tf.Variable(tf.zeros([32]), name="conv2\_biases")**  **conv2 = tf.nn.conv2d(relu1, conv2\_weights,strides=[1, 1, 1, 1], padding='SAME')**  **return tf.nn.relu(conv2 + conv2\_biases)**  有四个变量(Variables)conv1\_weighs, conv1\_biases和几个个操作(Op)，如果你直接调用两次，不会出什么问题，但是会生成两套变量；  # First call creates one set of 4 variables.  **result1 = my\_image\_filter(image1)**  # Another set of 4 variables is created in the second call.  **result2 = my\_image\_filter(image2)**  **vs = tf.trainable\_variables()**  **print ('There are %d train\_able\_variables in the Graph: ' % len(vs))**  **for v in vs:**  **print (v)**  最后会打印有八个可训练的参数：  Conv1\_weights  Conv1\_biases  Conv1\_weights\_1  Conv1\_biases\_1  ... |

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。

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。

**Tf.variable\_scope()**

**Tf.get\_variable()**

1.3.0 名空间与变量作用域

要理解 name\_scope 和 variable\_scope， 首先必须明确二者的使用目的。我们都知道，和普通模型相比，神经网络的节点非常多，节点节点之间的连接（权值矩阵）也非常多。



为了解决这个问题，我们引入了 name\_scope 和 variable\_scope， 二者又分别承担着不同的责任：

\* name\_scope: 为了更好地管理变量的命名空间而提出的。比如在 tensorboard 中，因为引入了 name\_scope， 我们的 Graph 看起来才井然有序。

\* variable\_scope: 大部分情况下，跟 tf.get\_variable() 配合使用，实现变量共享的功能。

|  |
| --- |
| #理解名空间，普通变量名，共享变量名  **import tensorflow as tf**  **import numpy as np**  **with tf.name\_scope('nsc1'):**  **v1 = tf.Variable([1], name='v1')**  **with tf.variable\_scope('vsc1',reuse=tf.AUTO\_REUSE):**  **v2 = tf.Variable([1], name='v2')**  **v3 = tf.get\_variable(name='v3', shape=[])**  **print ('v1.name: ', v1.name)**  **print ('v2.name: ', v2.name)**  **print ('v3.name: ', v3.name)**  #输出结果  v1.name: nsc1/v1:0  v2.name: nsc1/vsc1/v2:0  v3.name: vsc1/v3:0  #可以发现：  **tf.name\_scope() 并不会对 tf.get\_variable() 创建的变量有任何影响。**  **tf.name\_scope() 主要是用来管理命名空间的，这样子让我们的整个模型更加有条理。而 tf.variable\_scope() 的作用是为了实现变量共享，它和 tf.get\_variable() 来完成变量共享的功能。** |

1.3.1 变量作用域----变量管理问题：

假设要写一个简单的层数很少的分类网络，可以如下方式设计：

|  |
| --- |
| def weight\_variable(shape):  return tf.Variable(tf.truncated\_normal(shape, stddev=0.1))  def bias\_variable(shape):  return tf.Variable(tf.constant(0.1, shape=shape))  W\_conv1 = weight\_variable([5, 5, 3, 32])  b\_conv1 = bias\_variable([32])  h\_conv1 = tf.nn.relu(tf.nn.conv2d(...))  h\_pool1 = tf.nn.max\_pool(h\_conv1，...)  W\_conv2 = weight\_variable([5, 5, 32, 64])  b\_conv2 = bias\_variable([64])  h\_conv2 = tf.nn.relu(tf.nn.conv2d(...))  h\_pool2 = tf.nn.max\_pool(h\_conv2，...)  W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])  b\_fc1 = bias\_variable([1024])  h\_flat = tf.reshape(...)  h\_fc1 = tf.nn.relu(tf.matmul(...)) |

那么问题来了，如果让你写一个19层的VGG网络，甚至是上百层的Resnet呢？这种定义方法显然是行不通的，等手动把[W\_conv1，b\_conv1，W\_conv2，b\_conv2，…]这些东西输完，估计也对TF丧失兴趣了。

或者你会想到利用循环的方式：

|  |
| --- |
| def layer(shape, ...):  w = tf.Variable(tf.truncated\_normal(shape, stddev=0.1))  b = tf.Variable(tf.constant(0.1, shape=shape))  return tf.nn.relu(tf.nn.conv2d(...))  for i in range(19):  ...  x = layer(shape, ...)  ... |

从某种程度上来看确实解放了双手。但是，如果我现在想读取第8个卷积层中w和b的数值，有没有什么简单的方法呢？再或者我想把这个网络中的参数转移到另一个完全相同的网络中使用呢？虽然你可以再定义一组列表var，在每次新定义变量后var.append(w)，但从管理变量和传输变量的角度来看依旧不是很方便。

TF早就想到了这一点，并且提供Variable Scope机制来帮助管理变量。有了这个工具，就再也不用为变量的定义和共享伤脑筋了。

变量作用域机制在TensorFlow 中主要由两部分组成：

**• tf.get\_variable(<name>, <shape>, <initializer>，<trainable>) 通过所给的名字创建或是返回一个变量.**

**name :变量名称**

**shape :**

**initializer :初始化值**

**trainable : 是否可训练**

**• tf.variable\_scope(<scope\_name>) 创建一个变量域，相当于在变量空间打开一个文件夹，一般与tf.get\_variable()组合使用**

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) 利用给定值初始化**

**• 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**

在构造CNN以及LSTM等模型的时候，需要重复使用共享权重。但是tensorflow默认是不允许重复使用变量的，会报一些InvalidArgumentError或者Attemping to use uninitialized value等错误。

解决方法是使用variable\_scope，并且设置该variable\_scope的reuse=True。设置方法有几种：

**• with tf.variable\_scope(scope\_name,reuse=True): 或 reuse=tf.AUTO\_REUSE**

**• with tf.variable\_scope(scope\_name) as scope:**

**scope.reuse\_variables()**

**• #或 tf.get\_variable\_scope().reuse\_variables()**

1.3.2 具体的使用案例

|  |
| --- |
| **#简单的使用方式**  import tensorflow as tf  with tf.variable\_scope('cnn'):  with tf.variable\_scope('conv1'):  w = tf.get\_variable(  initializer = tf.truncated\_normal([3,3,3,32], stddev=0.1),  trainable=True, name = 'w')  b = tf.get\_variable(  initializer = tf.zeros([32]),  trainable=True, name = 'b')  print(w.name)  print(b.name) |
| **#创建一个cnn模型**  **#单独写在一个cnnmodel.py的文件中**  import tensorflow as tf  #定义变量  def weight\_variable(shape):  initial = tf.truncated\_normal(shape, stddev=0.1)  return tf.get\_variable(initializer = initial, trainable=True, name = 'w')  def bias\_variable(shape):  initial = tf.zeros(shape)  return tf.get\_variable(initializer = initial, trainable=True, name = 'b')  #定义卷积、全连接、池化操作  def conv2d(x, W\_shape):  W = weight\_variable(W\_shape)  B = bias\_variable(W\_shape[-1])  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME') + B  def ann(x, W\_shape):  W = weight\_variable(W\_shape)  B = bias\_variable(W\_shape[-1])  return tf.matmul(x, W) + B  def max\_pool\_2x2(x):  return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')  #在函数里面定义变量。每次调用函数，创建的都是局部变量，即使同名也不会有冲突。  # 然后就是创建CNN模型了：  def cnnmodel(inp, keep\_prob):  with tf.variable\_scope('cnn'):  with tf.variable\_scope('conv1'):  h\_conv1 = tf.nn.relu(conv2d(inp, [5, 5, 1, 32]))  h\_pool1 = max\_pool\_2x2(h\_conv1)  with tf.variable\_scope('conv2'):  h\_conv2 = tf.nn.relu(conv2d(h\_pool1, [5, 5, 32, 64]))  h\_pool2 = max\_pool\_2x2(h\_conv2)  with tf.variable\_scope('fc1'):  h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])  h\_fc1 = tf.nn.relu(ann(h\_pool2\_flat, [7 \* 7 \* 64, 1024]))  h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)  with tf.variable\_scope('fc2'):  y\_conv = tf.nn.softmax(ann(h\_fc1\_drop, [1024, 10]))  var = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope='cnn')  return y\_conv, var  #这样写的结果就是：  在变量空间中有一个总文件夹叫“cnn”，下面有许多子文件夹“conv1”、“conv2”、“fc1”、“fc2”。每个子文件夹下都有“w”、“b”两个变量。  最后的tf.get\_collection（）就是为了把在“cnn”目录下的变量集合起来，一步到位。是不是比自己定义列表一个一个.append()方便多了。 |
| **#训练文件**  import input\_data  mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=True)  import tensorflow as tf  from cnnmodel import cnnmodel  sess = tf.InteractiveSession()  x = tf.placeholder("float", shape=[None, 784])  y\_ = tf.placeholder("float", shape=[None, 10])  keep\_prob = tf.placeholder("float")  x\_image = tf.reshape(x, [-1,28,28,1])  y\_conv, var = cnnmodel(x\_image, keep\_prob)  cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y\_conv))  train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy, var\_list = var)  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.global\_variables\_initializer())  for i in range(2000):  batch = mnist.train.next\_batch(50)  train\_step.run(feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})  if i % 500 == 0 or i == 1999:  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))  saver = tf.train.Saver()  saver.save(sess, 'backup/latest') |
| **#导入和测试**  import input\_data  mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=True)  import tensorflow as tf  from cnnmodel import cnnmodel  sess = tf.InteractiveSession()  x = tf.placeholder("float", shape=[None, 784])  y\_ = tf.placeholder("float", shape=[None, 10])  keep\_prob = tf.placeholder("float")  x\_image = tf.reshape(x, [-1,28,28,1])  y\_conv, var = cnnmodel(x\_image, keep\_prob)  var\_ = tf.global\_variables()  net\_var = [var for var in var\_ if "cnn" in var.name] #可以导入其中某一层参数  sess.run(tf.global\_variables\_initializer())  if tf.train.get\_checkpoint\_state('backup/'):  print('\nfound\n')  saver = tf.train.Saver(net\_var)  saver.restore(sess, 'backup/latest')  要提醒一点，导入参数一定要放在sess.run(tf.global\_variables\_initializer())之后，否则你刚把变量值导好，一个变量初始化过来又变成预设好的初始值了。 |

### 1.4 取回fetch

为了取回操作的输出内容, 可以在使用Session 对象的run() 调用执行图时, 传入一些tensor, 这些tensor 会帮助你取回结果.

|  |
| --- |
| **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)** |

### 1.5 占位符placeholder和feed\_dict（供给feed）

TensorFlow 还提供给(feed) 机制, 该机制可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.]}))** |

占位符placeholder并没有初始值，它只会分配必要的内存。在会话中，占位符可以使用 feed\_dict 馈送数据。

Placeholder需要两个输入参数，第一个是数据类型，第二个为tensor的shape，如果shape中包含None，则代表该维度不限制维数。

feed\_dict 是一个字典，在字典中需要给出每一个用到的占位符的取值。在训练神经网络时需要每次提供一个批量的训练样本，如果每次迭代选取的数据要通过常量表示，那么 TensorFlow 的计算图会非常大。因为每增加一个常量，TensorFlow 都会在计算图中增加一个结点。所以说拥有几百万次迭代的神经网络会拥有极其庞大的计算图，而占位符却可以解决这一点，它只会拥有占位符这一个结点。

下面搭建一个三层的全连接神经网络：



其中需要注意的几个函数如 tf.nn.relu() 代表调用 ReLU 激活函数，tf.matmul() 为矩阵乘法等。tf.clip\_by\_value(yhat,1e-10,1.0) 这一语句代表的是截断 yhat 的值，因为这一语句是嵌套在 tf.log() 函数内的，所以我们需要确保 yhat 的取值不会导致对数无穷大。

tf.train.AdamOptimizer(learning\_rate).minimize(cost\_function) 是进行训练的函数，其中我们采用的是 Adam 优化算法更新权重，并且需要提供学习速率和损失函数这两个参数。

### 1.6 线程与队列

　在TensorFlow中使这么描写队列的:Queues are a powerful mechanism for asynchronous(异步) computation using TensorFlow。队列其实也是一个节点，是个有点类似于variable的状态节点，通过入队（enqueue）和出队（dequeue）来维持一种状态。其中异步是一个很重要的内容，以数据读取为例：在输入的pipeline上，多线程可以有效地增加数据读取的效率，当我们用队列来准备用于训练的数据时可以：

　　\* 多线程的准备训练数据，然后把它们push到队列中

　　\* 训练线程生成一个一个训练operation来从数据队列中去除mini-batch数量的数据。

　　其实Session就是多线程的，所以TensorFlow可以并行运行，但是多线程有三个条件需要满足:

　　- 所有线程都需要能够同时停止

　　- 必须能够捕捉和报告异常

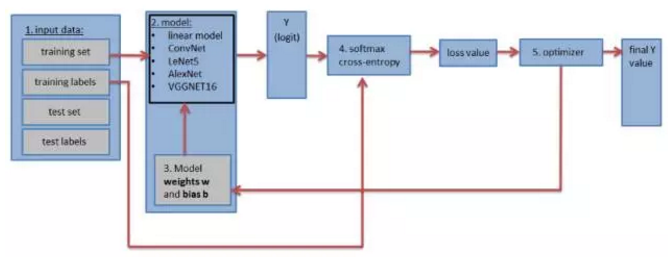
　　- 队列必须能在停止的时候适当关闭

　　TensorFlow给了我们两个类用于帮助我们协调线程：tf.Coordinator和tf.train.QueueRunner。这两个类实际上是一起使用的，其中tf.Coordinator用于多线程停止和异常捕捉，tf.train.QueueRunner用于操作Tensor入队。

### 1.7 数据处理

|  |
| --- |
| tf.constant(value,dtype = None,shape = None,name = "Constant")  这是最基础的一个，参数种dtype建议增加，若不增加，tensorflow会默认加一个，程序可能会有bug，shape对value进行一次形状变化，需要传入一个python列表；name为图中保存的名字。  tf.zeros(shape,dtype = tf.float32,name = None)/tf.ones(shape,dtype = tf.float32,name = None)  这两个函数差不多，一个生成全为0的常量，一个生成全为1的常量，默认的格式tf.float32.  tf.zeros\_like(tensor,dtype=None,name = None)/tf.ones\_like(tensor,dtype=None,name = None)  这俩函数基本一样，需要传入一个tensor，导出一个同结构全0/1的tensor，数据类型可能变化。  tf.linspace(start,stop,num,name = None )  生成一个一维tensor，以start为开始，以stop为结束，这两个数都必须是float，num为总个数（int）。 print(tf.linspace(0.0,10.0,11).eval()) 结果为 [ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]  tf.range(start,limit,delta = 1,name = "range")  生成一个一维tensor，以start为开始，最大不超过limit，间隔为delta。建议使用整数，浮点数可能会出现精度问题。  tf.random\_normal(shape,mean = 0,stddev = 1,dtype = tf.float32,seed = None,name = None)  tf.truncated\_normal(shape,mean = 0,stddev = 1,dtype = tf.float32,seed = None,name = None)  这两函数都是生成正态随机数，区别是truncated在生成随机数后，如果随机数在两倍标准差之外，就重新生成一个，直到生成最终结果。参数中shape为一个列表，表示形状，mean和stddev分别标书均值和标准差，seed为随机数种子。和其他python随机语法一样，固定seed以后，生成的随机数将不变。  tf.random.uniform(shape,minval = 0.0,maxval = 1.0,dtype = tf.float32,seed = None)  这个函数生成从0到1中平均分布的数值。  tf.random\_shuffle(value,seed = None,name = None)  传入一个tensor，然后将这个rensor进行随机洗牌  tf.gloable\_variables()  显示图中所有的变量。  tf.trainable\_variables()  显示图中可训练的变量。 |

## 2. TensorFlow中的神经网络



上图所描述的图像识别流程需要包含以下几步：

* 1）输入数据集，数据集分为训练数据集和标注、测试数据集和标注（包括验证数据集和标注）。测试和验证集能赋值到 tf.constant() 中，而训练集可以导入 tf.placeholder() 中，训练集只有导入占位符我们才能在随机梯度下降中成批量地进行训练。
* 2）确定神经网络模型，该模型可以是简单的一层全连接网络或 9 层、16 层的复杂卷积网络组成。
* 3）网络定义的权重矩阵和偏置向量后需要执行初始化，每一层需要一个权重矩阵和一个偏置向量。
* 4）构建损失函数，并计算训练损失。模型会输出一个预测向量，我们可以比较预测标签和真实标签并使用交叉熵函数和 softmax 回归来确定损失值。训练损失衡量预测值和真实值之间差距，并用于更新权重矩阵。
* 5）优化器，优化器将使用计算的损失值和反向传播算法更新权重和偏置项参数。

### 2.1 创建网络的函数

在使用 TensorFlow 构建卷积网络之前，我们需要了解一下 TensorFlow 中的函数。

TensorFlow 包含很多操作和函数，很多我们需要花费大量精力完成的过程可以直接调用已封装的函数，比如说「logits = tf.matmul(tf\_train\_dataset, weights) + biases」可以由函数「logits = tf.nn.xw\_plus\_b(train\_dataset, weights, biases)」代替。

还有很多函数可以让构建不同层级的神经网络变得十分简单。例如 conv\_2d() 和 fully\_connected() 函数分别构建了卷积层和全连接层。通过这些函数，层级的数量、滤波器的大小/深度、激活函数的类型等都可以明确地作为一个参数。权重矩阵和偏置向量能自动创建，附加激活函数和 dropout 正则化层同样也能轻松构建。

如下两种方式构建卷积层+池化层，都是等价的：

|  |
| --- |
| import tensorflow as tf    w1 = tf.Variable(tf.truncated\_normal([filter\_size, filter\_size, image\_depth, filter\_depth], stddev=0.1))  b1 = tf.Variable(tf.zeros([filter\_depth]))  layer1\_conv = tf.nn.conv2d(data, w1, [1, 1, 1, 1], padding='SAME')  layer1\_relu = tf.nn.relu(layer1\_conv + b1)  layer1\_pool = tf.nn.max\_pool(layer1\_pool, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME') |
| from tflearn.layers.conv import conv\_2d, max\_pool\_2d  layer1\_conv = conv\_2d(data, filter\_depth, filter\_size, activation='relu')  layer1\_pool = max\_pool\_2d(layer1\_conv\_relu, 2, strides=2) |

2.1.1 conv2d()函数和max\_pool函数

conv2d()函数有四个参数：

1）输入图像，即一个四维张量 [batch size, image\_width, image\_height, image\_depth]

2）卷积核权重矩阵，即一个四维张量 [filter\_size, filter\_size, image\_depth, filter\_depth]

3）卷积操作时每一个维度的步幅数

4）Padding (= 'SAME' / 'VALID')

前面两个参数都是四维张量，其包括了批量输入图像的信息和卷积滤波器的权值。

第三个参数为卷积的步幅（stride），即卷积滤波器在 4 个维度中的每一次移动的距离。四个中间的

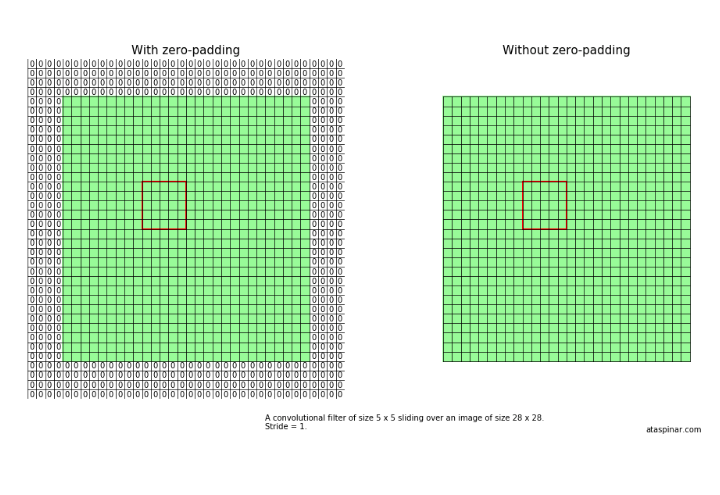
第一个维度代表着图像的批量数，这个维度肯定每次只能移动一张图片。

最后一个维度为图片深度（即色彩通道数，1 代表灰度图片，而 3 代表 RGB 图片），因为我们通常并不想跳过任何一个通道，所以这一个值也通常为 1。

第二个和第三个维度代表 X 和 Y 方向（图片宽度和高度）的步幅。如果我们希望能应用步幅参数，我们需要设定每个维度的移动步幅。例如设定步幅为 1，那么步幅参数就需要设定为 [1, 1, 1, 1]，如果我们希望在图像上移动的步幅设定为 2，步幅参数为 [1, 2, 2, 1]。

第四个参数表明 TensorFlow 是否需要使用 0 来填补图像周边，这样以确保图像输出尺寸在步幅参数设定为 1 的情况下保持不变。通过设置 padding = 'SAME'，图像会只使用 0 来填补周边（输出尺寸不变），而 padding = 'VALID'则不会使用 0。

在下图中，我们将看到两个使用卷积滤波器在图像上扫描的案例，其中滤波器的大小为 5 x 5、图像的大小为 28 x 28。左边的 Padding 参数设置为'SAME'，并且最后四行/列的信息也会包含在输出图像中。而右边 padding 设置为 'VALID'，最后四行/列是不包括在输出图像内的。



对于任意给定的步幅 S、滤波器尺寸 K、图像尺寸 W、padding 尺寸 P，输出的图像尺寸可以总结上表的规则如下：

IMG_256

Max\_pool()函数有四个参数：

第一个是输入的特征图；

第二个是池化操作的尺寸大小；

第三个是池化操作的步幅大小；

第四个是padding的方式。

其中第二个和第三个参数都是四维矩阵，其中第一维表示特征图的个数，所以一般都取1；第二维和第三维表示每个特征图的宽和高的维度，第四维表示每个特征图的通道数，一般也取1。

2.1.2 relu激活函数和偏置项

a=tf.nn.relu(tf.matmul(x,w1)+biases1)

2.1.3 softmax交叉熵函数

cross\_entropy=tf.nn.softmax\_cross\_entropy\_with\_logits(yhat,y)

2.1.4 均方误差损失函数mse

mse=tf.reduce\_mean(tf.square(y-yhat))

2.1.5 带L1/L2正则化的损失函数

w=tf.Variable(tf.random\_normal([2,1],stddev=1,seed=1))

yhat=tf.matmul(x,w)

Loss\_L1=tf.reduce\_mean(tf.square(y-yhat))+tf.contrib.layers.l1\_regularizer(lambda)(w)

Loss\_L2=tf.reduce\_mean(tf.square(y-yhat))+tf.contrib.layers.l2\_regularizer(lambda)(w)

2.1.6 逐步衰减学习率

#decayed\_learning\_rate=learning\_rate\*decay\_rate^(global\_steps/decay\_steps)，指数衰减函数的定义

global\_step=tf.Variable(0)

#使用exponential\_decay生成学习速率，因为staircase=tire，每100次迭代，学习率×0.96

learning\_rate=tf.train.exponential\_decay(0.1,global\_step,100,0.96,staircase=True)

#在minimize中导入global\_step将自动更新

#learning\_step=tf.train.GtadientDescentOptimizer(learning\_rate).minimize(loss\_function,global\_step=global\_step)

2.1.7 训练优化方法 SGD / Adam

（1）SGD

train\_step=tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss,global\_step=global\_step)

（2）Adam

train\_step=tf.train.AdamOptimizer(0.001).minimize(cross\_entropy)

### 2.2 常见API

tf.variable\_scope('layer1-conv1', reuse = True) 创建变量作用域

tf.name\_scope('layer2-pool1') 创建名空间

tf.greater()的输入是两个张量，比较两个张量中的每一个元素，并返回比较结果（true或false的向量）。

tf.select()有三个参数，第一个参数条件为真时选择第二个参数中的值，否则选择第三个参数的值。

TensorFlow内建的运算操作

|  |  |
| --- | --- |
| 类型 | 示例（tf.\*） |
| 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 |
| 存储、恢复 | Save、Restore |
| 队列及同步运算 | Enqueue、Dequeue、MutexAcquire、MutexRelease |
| 控制流 | Merge、Switch、Enter、Leave、NextIteration |

Tensorflow（简称tf）下的函数介绍

|  |  |  |
| --- | --- | --- |
| tf.nn | tf.nn.softmax | 求softmax函数  输入为[batchsize，feature]，输出为[batchsize，num\_classes] |
| tf.nn.softmax\_cross\_entropy\_with\_logits | 计算logits和labels之间的softmax cross entropy |
|  |  |
| Tf.nn.relu  Tf.nn.signoid  Tf.nn.tanh  Tf.nn.relu\_x | 常用激活函数 |
| Tf.nn.conv2d(input,  filter,  strides,  padding,  use\_cudnn\_on\_gpu,  data\_format=None,  Name=None) | 1)输入数据共四个维度，维度的构成有下面的data\_format决定  2)卷积核四维，维度构成总是[filter\_height高, filter\_width宽, in\_channels输入通道数, out\_channels输入通道数]  3)步长  4)补零格式  6)data\_format这是一个string格式的总共有："NHWC", "NCHW"两种，其中默认的是"NHWC"也就是数据按照[batch, height, width, channels]的格式输入的，而'NCHW'顾名思义是按照[batch, channels, height, width]输入的。默认情况下输入是[batch, in\_height, in\_width, in\_channels]，卷积核是[filter\_height, filter\_width, in\_channels,out\_channels] |
| Tf.nn.conv3d |  |
| Tf.nn.depthwise\_conv2d(  input,设输入是[batch,in\_height, in\_width, in\_channels]  filter,尺寸是[filter\_height, filter\_width, in\_channels, channel\_multiplier].  strides,[1, stride, stride, 1]  padding,  rate=None,  name=None,  data\_format=None  ) | 和tf.nn.conv2d的区别在于conv2d的卷积核的输入通道数与数据的通道数是一样的，也就是每个卷积核同时作用于所有层然后累加得出一个输出层的的结果，而这里的depthwise\_conv2d每个卷积核的输入维度是1也就是说不同的卷积核作用在每个输入通道上，按照输入的通道数一层一层处理。更通俗的说就是我们假设有4个输入通道数是1的卷积核，然后把这4个卷积核一层一层的作用在一个3通道的输入数据上，这样就得到了一个12个输出，把这12个输出并起来就可以了。 |
| Tf.nn.max\_pool(  value,尺寸如下[batch, height, width, channels]，而且要类型为tf.float32  ksize,池化的尺寸  strides,  padding,  data\_format='NHWC',  name=None  ) |  |
| Tf.nn.top\_k | 计算前k个最大值 |
| Tf.nn.in\_top\_k | 判断目标是不是前k个值中的一个 |
|  |  |
| tf.matmul | 矩阵乘法 | |
| tf.reduce\_mean | 求平均的作用 | |
| tf.reduce\_sum | 求和符号的作用 | |
| Tf.train.optimizer(优化器类) | tf.train.GradientDescentOptimizer().minimize() | 梯度下降求解器、学习率、优化目标 |
| AdagradOptimizer()  MomentumOptimizer()  .... |  |
| tf.train.exponential\_decay(starter\_learning\_rate, global\_step,  100000, 0.96, staircase=True) | | 学习率自动衰减，返回学习率变量 |
| tf.global\_variables | 全局参数初始化器 | |
| tf.argmax | 从一个tensor中寻找最大值的序号 | |
|  |  | |
|  |  | |
| Tf.contrib | tf.contrib.rnn | |
|  |  | |

### 2.3 设备选择与多GPU并行

2.3.1 指定运算使用的设备

With tf.device(“/gpu:0，2”) #使用第一块GPU和第三块GPU

With tf.device(“/cpu:0”) #使用cpu运算

由于有些操作只能在CPU上进行，在Session中的allow\_soft\_placement=True，可以避免错误。

2.3.2 使用多个GPU并行

在一个循环内可以循环使用每个GPU进行运算，调用tf.get\_variable\_scope().reuse\_variables()重用参数，让所有GPU共用一个模型以及完全相同的参数。

将单个GPU产生的梯度，添加到梯度列表，最后计算平均梯度，然后更新模型参数。

这样就完成多GPU的同步训练和更新。

## 3. Keras--TensorFlow外层模块化神经网络库

崇尚极简、高度模块化的神经网络库，基于python实现，可以运行在TensorFlow和theano上。用户只需要将高级模块拼装在一起，大大降低了编程开销。

它同时支持卷积网络和循环网络，支持级联模型或任意图结构模型。Keras训练模型相比TensorFlow和theano基本没有性能损耗，只是简化了编程复杂度。可以说模型越复杂，使用keras收益越大。尤其是在高度依赖权值共享、多模型组合、多任务学习模型上。