TensorFlow学习笔记

1、tensorflow基础知识

- 使用图（graph）来表示计算；
- 在会话（Session）中执行图；
- 使用张量（Tensor）来代表数据；
- 通过变量（Variable）维护状态；
- 使用供给（feeds）和取回（Fetches）将数据传入或传出任何操作；

TensorFlow中图（graph）表示计算的编程系统，图中的节点成为op（operator），一个op获得零活多个张量（Tensor）。在开始计算之前，图必须在会话（Session）中启动，会话将图的op分发到如CPU或GPU的设备上，同时提供执行op的方法，这些方法执行后将会产生张量（Tensor）返回。在Python中，将返回numpy的ndarray对象。

2、计算图（computation graph）

TensorFlow编程可按两个阶段组织起来：构建阶段和执行阶段；前者用于组织计算图，后者利用session中直行计算图的op操作。

1）构建计算图（Building The Graph）

刚开始基于op建立图的时候一般不需要任何输入源（source op），例如输入常量（Constant），再将他们传递给其他op执行运算。tensorflow有一个可被op构造函数加入计算节点的默认图，该默认图可以满足大多数应用。如下，构建了两个constant() op，一个matmul（）op：

```
# Create a Constant op that produces a 1*2 matrix. The op is added as a node to defaut graph
# The value returned by the constructor represents the output of the Constant op.
matrix1 = tf.constant([[3.0, 3.0]]);

# Create another Constant that produces a 2*1 matrix
matrix2 = tf.constnat([[2.0, 2.0]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
product = tf.matmul(matrix1, matrix2)
```

2）执行（在会话中载入图 Launching the graphs in session）

为了载入之前所构建的图，必须先创建一个会话对象（Session Object），会话构建器将在未指明参数时载入默认图，如下：

```
# Session (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 th the call that we want to get the output of the matmul op back.
result = sess.run(product)
print(result)

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

会话结束后，必须关闭以释放资源。

也可以使用'with'语句块开始一个会话，该会话将在with语句块结束时自动关闭，如：

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

3、交互式使用

考虑到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维持会话。如：

```
# 进入一个交互式回话
sess = tf.InteractiveSession()
x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# 使用初始化器initlizer op的run方法初始化'x'
x.initializer.run()

# 增加一个减法sub op， 从'x'减去'a'
sub = tf.subtract(x, a)
print(sub.eval())
```

4、tensor

TensorFlow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。可以把tensor看作一个n维的数组或列表。一个tensor包含一个静态类型的rank和一个shape。

5、变量（Variable）

变量维护图执行过程中的状态信息，如：

```
# 创建一个变量，初始化为标量0
state = tf.Variable(0, name='counter')
# 创建一个op，其作用是使state增加1
one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

# 启动图后，变量必须先经过初始化 （init）op
# 首先必须增加一个初始化op到图中
init_op = tf.global_variables_initializer()

# 启动图，运行op
with tf.Session() as sess:
    sess.run(init_op)
    # 打印初始值
    print(sess.run(state))
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))
```

6、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。如：

```
def demo_fetch():
    input1 = tf.constant(3.0)
    input2 = tf.constant(2.0)
    input3 = tf.constant(5.0)
    intermed = tf.add(input2, input3)
    mul = tf.multiply(input1, intermed)
    with tf.Session() as sess:
        result = sess.run([mul, intermed])
        print(result)
```

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）。

7、Feed

在Fetch的例子中，计算图引入了tensor，以常量或变量的形式储存。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不定，直接插入一个tensor。

Feed使用一个tensor值临时替换一个操作输出的结果，你可以提供feed数据作为run（）调用的参数。fedd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。如：

```
def demo_feed():
    input1 = tf.placeholder(tf.float32)
    input2 = tf.placeholder(tf.float32)
    output = tf.multiply(input1, input2)
    with tf.Session() as sess:
        result = sess.run([output], feed_dict={input1:[7.0], input2:[2.0]})
        print(result)
```

8、面向机器学习初学者的 MNIST 初级教程

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。

1）数据准备

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

```
import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

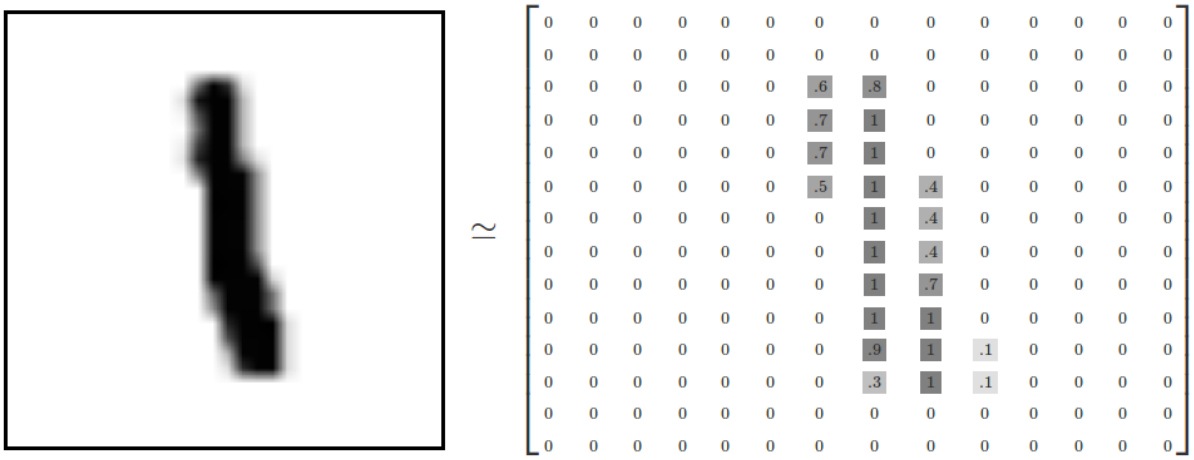
另外，tensorflow已经集成了对MNIST数据集的下载，代码如下：

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

通过上面的代码就可以将MNIST数据集下载到MNIST_data目录下。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）。

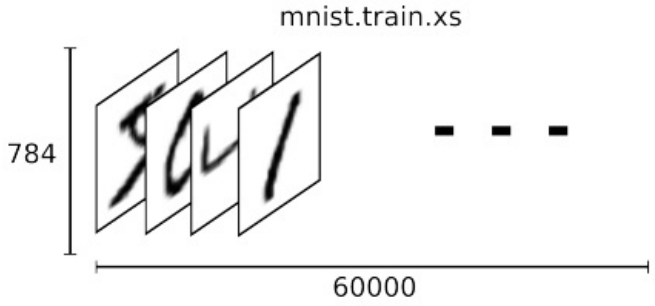
正如前面提到的一样，每一个MNIST数据单元有两部分组成：一张包含手写数字的图片和一个对应的标签。我们把这些图片设为“xs”，把这些标签设为“ys”。训练数据集和测试数据集都包含xs和ys，比如训练数据集的图片是 mnist.train.images ，训练数据集的标签是 mnist.train.labels。

每一张图片包含28像素X28像素。我们可以用一个数字数组来表示这张图片：

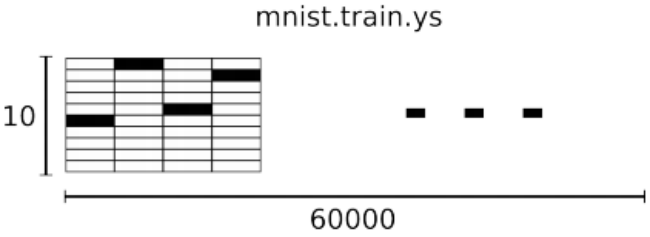


我们把这个数组展开成一个向量，长度是 28x28 = 784。如何展开这个数组（数字间的顺序）不重要，只要保持各个图片采用相同的方式展开。从这个角度来看，MNIST数据集的图片就是在784维向量空间里面的点, 并且拥有比较复杂的结构 (提醒: 此类数据的可视化是计算密集型的)。

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。



2）Softmax回归介绍

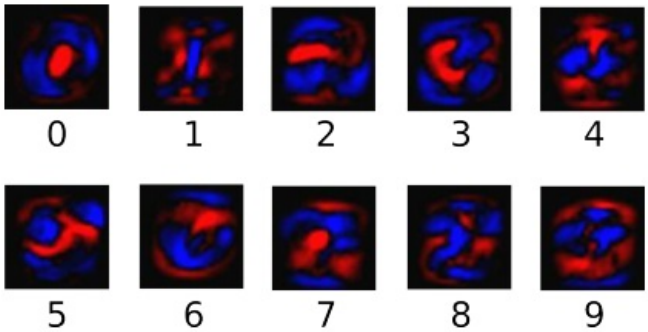
我们知道MNIST的每一张图片都表示一个数字，从0到9。我们希望得到给定图片代表每个数字的概率。比如说，我们的模型可能推测一张包含9的图片代表数字9的概率是80%但是判断它是8的概率是5%（因为8和9都有上半部分的小圆），然后给予它代表其他数字的概率更小的值。

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：

① 为了得到一张给定图片属于某个特定数字类的证据（evidence），我们对图片像素值进行加权求和。如果这个像素具有很强的证据说明这张图片不属于该类，那么相应的权值为负数，相反如果这个像素拥有有利的证据支持这张图片属于这个类，那么权值是正数。

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



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

证据可以表示为：

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

其中bi代表数字 i 类的偏置量，j 代表给定图片 x 的像素索引用于像素求和。

②用softmax函数可以把这些证据转换成概率 y：

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

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

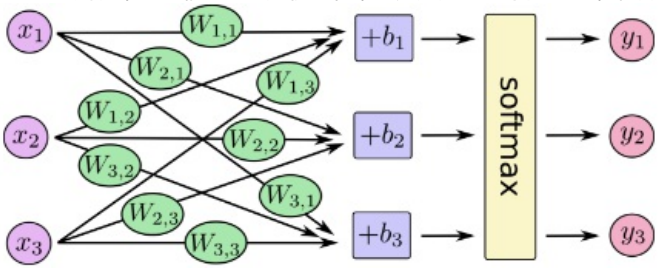
$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

展开等式右边的子式，可以得到：

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

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

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：



如果把它写成一个等式，我们可以得到：

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

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：

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

3）实现回归模型

首先导入tensorflow模块，如下：

```
from tensorflow.examples.tutorials.mnist import input_data
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(tf.float32, [None, 784], name = "X_input")
```

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]), name = 'W')
b = tf.Variable(tf.zeros([10]), name = 'b')
```

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。

现在，我们可以实现我们的模型啦。只需要一行代码，如下：

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

首先，我们用tf.matmul(X，W)表示x乘以W，对应之前等式里面的Wx，这里x是一个2维张量拥有多个输入。然后再加上b，把和输入到tf.nn.softmax函数里面。至此，我们先用了几行简短的代码来设置变量，然后只用了一行代码来定义我们的模型。TensorFlow不仅仅可以使softmax回归模型计算变得特别简单，它也用这种非常灵活的方式来描述其他各种数值计算，从机器学习模型对物理学模拟仿真模

型。一旦被定义好之后，我们的模型就可以在不同的设备上运行：计算机的CPU，GPU，甚至是手机！

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：

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

y 是我们预测的概率分布, y' 是实际的分布（我们输入的one-hot vector)。比较粗糙的理解是，交叉熵是用来衡量我们的预测用于描述真相的低效性。

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：

```
yactual = tf.placeholder(tf.float32, [None, 10])
```

然后我们可以用下式计算交叉熵:

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

```
entropy = -tf.reduce_sum(yactual*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 = tf.train.GradientDescentOptimizer(0.01).minimize(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。

然后开始训练模型，这里我们让模型循环训练1000次，如下：

```
for i in range(1000):
    batch_x, batch_y = mnist.train.next_batch(100)
    feed={x:batch_x, yactual:batch_y}
    sess.run(train, feed_dict = feed)
```

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。

5）评估模型

那么我们的模型性能如何呢？

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

```
pred = tf.equal(tf.argmax(y,1),tf.argmax(yactual,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(pred, tf.float32))
```

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, yactual:mnist.test.labels}))
```

这个最终结果值应该大约是91%。

这个结果好吗？嗯，并不太好。事实上，这个结果是很差的。这是因为我们仅仅使用了一个非常简单的模型。不过，做一些小小的改进，我们就可以得到97%的正确率。最好的模型甚至可以获得超过99.7%的准确率！（想了解更多信息，可以看看这个关于各种模型的性能对比列表)。不过，如果你仍然对这里的结果有点失望，可以查看下一个教程，在那里你可以学习如何用FensorFlow构建更加复杂的模型以获得更好的性能！

本文的示例代码下载：MNIST使用softmax的示例代码下载（tensorflow V1.0.）。

9、 MNIST 进阶

TensorFlow是一个非常强大的用来做大规模数值计算的库。其所擅长的任务之一就是实现以及训练深度神经网络。在本教程中，我们将学到构建一个TensorFlow模型的基本步骤，并将通过这些步骤为MNIST构建一个深度卷积神经网络。

1）加载MNIST数据

tensorflow已经集成了对MNIST数据集的下载，代码如下：

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

通过上面的代码就可以将MNIST数据集下载到MNIST_data目录下，mnist是一个轻量级的类。它以Numpy数组的形式存储着训练、校验和测试数据集。同时提供了一个函数，用于在迭代中获得minibatch。

2）运行TensorFlow的InteractiveSession

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

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

3）计算图

为了在Python中进行高效的数值计算，我们通常会使用像NumPy一类的库，将一些诸如矩阵乘法的耗时操作在Python环境的外部来计算，这些计算通常会通过其它语言并用更为高效的代码来实现。但遗憾的是，每一个操作切换回Python环境时仍需要不小的开销。如果你想在GPU或者分布式环境中计算时，这一开销更加可怖，这一开销主要可能是用来进行数据迁移。

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

因此Python代码的目的是用来构建这个可以在外部运行的计算图，以及安排计算图的哪一部分应该被运行。

4）构建一个多层卷积网络

① 权重初始化

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

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

def bias_variable(shape):
    inital = tf.constant(0.1, shape=shape)
    return tf.Variable(inital)
```

② 卷积核池化

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')
```

③ 第一层卷积

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

```
# First Convolution Layer - Relu + pooling
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，其第2、第3维对应图片的宽、高，最后一维代表图片的颜色通道数(因为是灰度图所以这里的通道数为1，如果是rgb彩色图，则为3)。

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

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

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

④ 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 = weight_variable([64])

W_conv2_hist = tf.summary.histogram("W_conv2", W_conv2)
b_conv2_hist = tf.summary.histogram("b_conv2", b_conv2)

# Second Convolution Layer - Relu + pooling
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2)+b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

⑤ 密集连接层

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

```
# Fully connected NN
h_pool2_flat = tf.reshape(h_pool2, [-1,(7*7*64)])

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

W_fc1_hist = tf.summary.histogram("W_fc1", W_fc1)
b_c1_hist = tf.summary.histogram("b_fc1", b_fc1)

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

⑥ Dropout

为了减少过拟合，我们在输出层之前加入dropout。我们用一个placeholder来代表一个神经元的输出在dropout中保持不变的概率。这样我们可以在训练过程中启用dropout，在测试过程中关闭dropout。 TensorFlow的tf.nn.dropout操作除了可以屏蔽神经元的输出外，还会自动处理神经元输出值的scale。所以用dropout的时候可以不用考虑scale。

```
# Dropout
prob = tf.placeholder(tf.float32)
h_fc1_dropout = tf.nn.dropout(h_fc1, prob)
```

⑦ 输出层

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

```
y_pred = tf.nn.softmax(tf.matmul(h_fc1_dropout, W_fc2)+b_fc2)
```

⑧ 训练和评估模型

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

```
# Train
entropy = -tf.reduce_sum(y_actual * tf.log(y_pred))
train_step = tf.train.AdamOptimizer(1e-4).minimize(entropy)

tf.summary.scalar("Cross_Entr", entropy)

# Eval
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_pred, 1), tf.argmax(y_actual, 1)), tf.float32))

# Session Start
init = tf.global_variables_initializer()
sess.run(init)

summary_op = tf.summary.merge_all()
summaryWriter = tf.summary.FileWriter("logs/MNIST_Activation/", sess.graph)

for i in range(12001):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x:batch[0], y_actual:batch[1], prob:0.5})

    if i%10 == 0:
        summary_str = sess.run(summary_op, feed_dict={x:batch[0], y_actual:batch[1], prob:0.5})
        summaryWriter.add_summary(summary_str, i)

    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_actual:batch[1], prob:1.0})
        print("step: %d, training accuracy:%g" %(i, train_accuracy))

test_accuracy = accuracy.eval(feed_dict={x: mnist.test.images, y_actual: mnist.test.labels, prob: 1.0})
print("************* test accuracy: %g ************" % test_accuracy)
```

以上代码，在最终测试集上的准确率大概是99.2%。

本文的示例代码下载：MNIST进阶-多层卷积网络示例代码下载（TensorFlow V1.01）。

10、 MNIST高级-TensorFlow运作方式入门

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）。

1）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。通过import tensorflow和mnist模块，

```
import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.examples.tutorials.mnist import mnist
```

使用如下代码下载数据：

```
# Get the sets of images and labels for training, validation and test on MNIST
data_sets = input_data.read_data_sets(FLAGS.input_data_dir, FLAGS.fake_data)
```

注意： `fake_data` 标记是用于单元测试的，读者可以不必理会。

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

2）输入占位符

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。

```
def placeholders_inputs(batch_size):
    """Generate placeholder variables to represent the input tensors.
       These placeholders are used as input by the rest of the model building code
       and will be fed from the downloaded data in the .run() loop, below.

       Args:
          batch_size: the batch size will be baked into both placeholder
       Returns:
          images_placeholders: Image placeholders
          labels_placeholders: Lbels placeholders
     """
    # Note that the shapes of the placeholders match the shapes of the full image and label tesnors"
    images_placeholders = tf.placeholder(tf.float32, shape=(batch_size, mnist.IMAGE_PIXELS))
    labels_placeholders = tf.placeholder(tf.int32, shape=(batch_size))
    return images_placeholders, labels_placeholders
```

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。

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()，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。

4）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_scope之下，创建于该作用域之下的所有元素都将带有其前缀。

```
def inference(images, hidden1_units, hidden2_units):
  """Build the MNIST model up to where it may be used for inference.

  Args:
    images: Images placeholder, from inputs().
    hidden1_units: Size of the first hidden layer.
    hidden2_units: Size of the second hidden layer.

  Returns:
    softmax_linear: Output tensor with the computed logits.
  """
  # Hidden 1
  with tf.name_scope('hidden1'):
    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')
    hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)
  # Hidden 2
  with tf.name_scope('hidden2'):
    weights = tf.Variable(
        tf.truncated_normal([hidden1_units, hidden2_units],
                            stddev=1.0 / math.sqrt(float(hidden1_units))),
        name='weights')
    biases = tf.Variable(tf.zeros([hidden2_units]),
                         name='biases')
    hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)
  # Linear
  with tf.name_scope('softmax_linear'):
    weights = tf.Variable(
        tf.truncated_normal([hidden2_units, NUM_CLASSES],
                            stddev=1.0 / math.sqrt(float(hidden2_units))),
        name='weights')
    biases = tf.Variable(tf.zeros([NUM_CLASSES]),
                         name='biases')
    logits = tf.matmul(hidden2, weights) + biases
  return logits
```

例如，当这些层是在hidden1作用域下生成时，赋予权重变量的独特名称将会是"hidden1/weights"。每个变量在构建时，都会获得初始化操作（initializer ops）。

在这种最常见的情况下，通过tf.truncated_normal函数初始化权重变量，给赋予的shape则是一个二维tensor，其中第一个维度代表该层中权重变量所连接（connect from）的单元数量，第二个维度代表该层中权重变量所连接到的（connect to）单元数量。对于名叫hidden1的第一层，相应的维度则是[IMAGE_PIXELS, hidden1_units]，因为权重变量将图像输入连接到了hidden1层。
tf.truncated_normal初始函数将根据所得到的均值和标准差，生成一个随机分布。然后，通过tf.zeros函数初始化偏差变量（biases），确保所有偏差的起始值都是0，而它们的shape则是其在该层中所接到的（connect to）单元数量。

图表的三个主要操作，分别是两个tf.nn.relu操作，它们中嵌入了隐藏层所需的tf.matmul；以及logits模型所需的另外一个

tf.matmul。三者依次生成，各自的tf.Variable实例则与输入占位符或下一层的输出tensor所连接。

5）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]，实现如下：

```
def loss(logits, labels):
  """Calculates the loss from the logits and the labels.

  Args:
    logits: Logits tensor, float - [batch_size, NUM_CLASSES].
    labels: Labels tensor, int32 - [batch_size].

  Returns:
    loss: Loss tensor of type float.
  """
  labels = tf.to_int64(labels)
  cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
      labels=labels, logits=logits, name='xentropy')
  return tf.reduce_mean(cross_entropy, name='xentropy_mean')
```

最后，程序会返回包含了损失值的Tensor。

6）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。实现如下：

```
def training(loss, learning_rate):
  """Sets up the training Ops.

  Creates a summarizer to track the loss over time in TensorBoard.

  Creates an optimizer and applies the gradients to all trainable variables.

  The Op returned by this function is what must be passed to the
  `sess.run()` call to cause the model to train.

  Args:
    loss: Loss tensor, from loss().
    learning_rate: The learning rate to use for gradient descent.

  Returns:
    train_op: The Op for training.
  """
  # Add a scalar summary for the snapshot loss.
  tf.summary.scalar('loss', loss)
  # Create the gradient descent optimizer with the given learning rate.
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  # Create a variable to track the global step.
  global_step = tf.Variable(0, name='global_step', trainable=False)
  # Use the optimizer to apply the gradients that minimize the loss
  # (and also increment the global step counter) as a single training step.
  train_op = optimizer.minimize(loss, global_step=global_step)
  return train_op
```

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）。

接下来，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）。

之后，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。

7）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。

```
if (step + 1) % 1000 == 0 or (step + 1) == FLAGS.max_steps:
    # Save a checkpoint and evaluate the model periodically.
    checkpoint_file = os.path.join(FLAGS.log_dir, 'model.ckpt')
    saver.save(sess, checkpoint_file, global_step=step)
    # Evaluate against the training set.
    print('Training Data Eval:')
    do_eval(sess, eval_correct, images_placeholder, labels_placeholder, data_sets.train)

    # Evaluate against the validation set.
    print('Validation Data Eval:')
    do_eval(sess, eval_correct,  images_placeholder, labels_placeholder, data_sets.validation)

    # Evaluate against the test set.
    print('Test Data Eval:')
    do_eval(sess,eval_correct,images_placeholder,labels_placeholder, data_sets.test)
```

本文的示例代码下载：MNIST高级-TensorFlow运作方式入门（TensorFlow V1.01）。

11、 卷积神经网络

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。

教程演示了在TensorFlow上构建更大更复杂模型的几个种重要内容：

- 相关核心数学对象，如卷积、修正线性激活、最大池化以及局部响应归一化；
- 训练过程中一些网络行为的可视化，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；

1）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。

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）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:

- 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；
- 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.
- 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。

3）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；

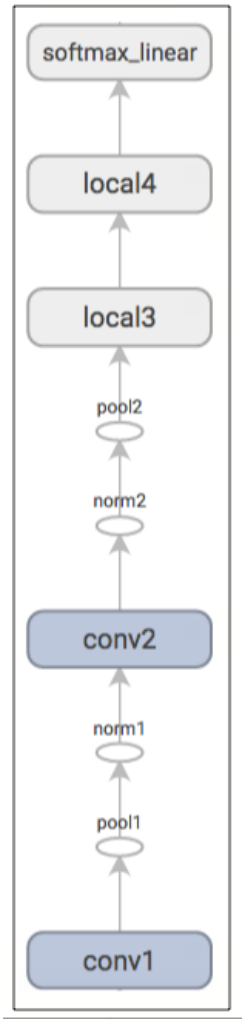
可以在Images页的列表中查看所有可用的变换，对于每个原始图我们还附带了一个image_summary，以便于在TensorBoard中查看。这对于检查输入图像是否正确十分有用。

4）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.

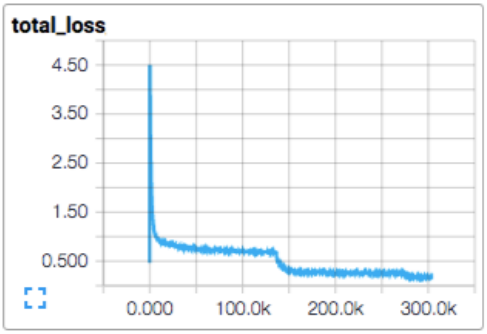
这里有一个由TensorBoard绘制的图形，用于描述模型建立过程中经过的步骤：



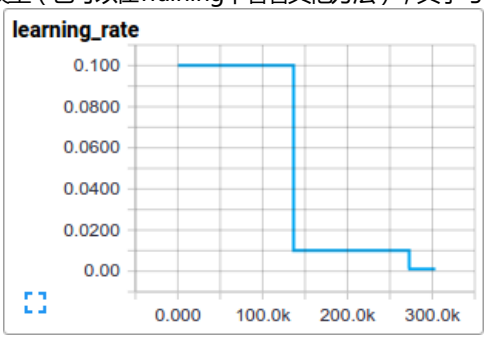
5）模型训练

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

在TensorBoard中使用scalar_summary来查看该值的变化情况：



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



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

6）开始执行并训练模型

至此，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
```

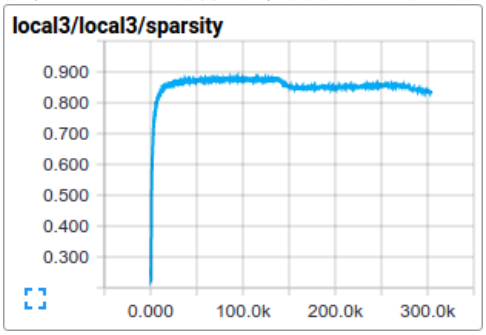
注意: 当第一次在CIFAR-10教程上启动任何任务时，会自动下载CIFAR-10数据集，该数据集大约有160M大小。

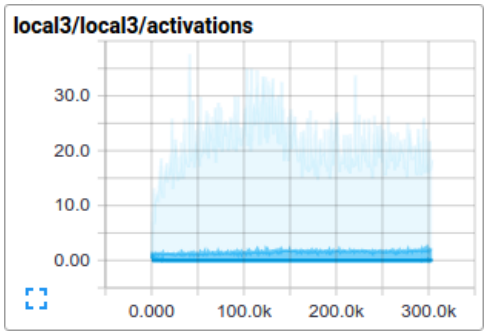
脚本会在每10步训练过程后打印出总损失值，以及最后一批数据的处理速度。下面是几点注释：

- 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。

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。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。

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。

7）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。

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
```

评估脚本只是周期性的返回precision@1 --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。

8）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。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。

Author：Belial Email: 312611432@qq.com