TensorFlow 零基础入门

一、写在前面的

本章中来讲一讲TenorFlow的零基础入门，本章中将会学到：

- TensorFlow 简介
- TensorFlow 一些常用的概念：张量、会话和图
- TensorFlow 求解方程与泛函、loss函数
- TensorFlow 实现全链接网络求解抑或和手写识别问题
- TensorBoard 查看计算图以及变量作用域

这个相比于前面一章会有更有动力一些。一直感觉做东西那种成就感才是学习兴趣的来源。所以说学好TenorFlow是学习机器学习的第一步，因为可以用它去完成现阶段大部分的机器学习任务，而这种直接做出产品的快感是难以言表的。当然官方宣称TenorFlow已经脱离了玩具的范畴，其是直接可以直接应用于工业产品的，如果说0.8版本以前来看的话，这个官宣显然像个笑话，单机速度慢，分布式不支持，但是之后的版本加入了各种支持甚至于手机上都可以运行了，此时再看这种官方说法还是比较靠谱的。

那么作为前言的东西，这里写下如何安装吧，首先是推荐一个python的集成环境Anaconda，懒人必备，包含了很多的现成的库。比如绘图库，比如numpy，比如spyder，还有一个包管理工具conda。在shell(win下power shell)里输入：


```
conda list
```

查看一下安装了什么包以及其版本，不过我想大部分人都不太会仔细看的，甚至于根本都不会看。可以试着用conda对sklearn进行一下升级：

```
conda update scikit-learn
```

python中还有一个常用的包管理工具pip，可以试着用它来安装TensorFlow的预编译版本：

```
pip install tensorflow
```

所谓无“显卡”不智能，gpu对于计算速度的提升是巨大的，因此作为一个靠谱的机器学习库是不能不提供GPU版本的：

```
pip install tensorflow-gpu
```

前提是得安装英伟达那套机器学习api，直白点说就是得有个核弹显卡。他家那套专门用于计算的GPU价格还是挺感人的。短时间使用的话可以买个云服务。恩，也不便宜。

二、该有的概念

每次写教程都得写一些概念，这些概念不是数学中的那些拗口的概念用来定义一些简单的东西，而是形成一种知识映射，将这个领域的知识映射到另外一个领域中，因为很多讨厌的人喜欢在自己的领域发明新的名词。比如**矩阵**用

的挺好他要叫成**张量**，比如**泛函**名字简洁明了，他要起个名叫**代价函数**（这两个概念还是有一定区别的）。

以前就说过，机器学习中进行的很多计算都是矩阵相关的运算，因此首先一个就是定义一个矩阵，那么TensorFlow中有几种形式的矩阵：

```
#引入库
import tensorflow as tf
import numpy as np
#定义几种形式的矩阵
A = tf.Variable([[1, 1], [-1, 1]], name="A")
B = tf.constant([[2, 2], [-2, 2]], name="B")
C = tf.placeholder(tf.float32, [2, 2], name="C")
```

第一个是代表一个变量，输入的**列表**[[1, 1], [-1, 1]]是对其初始化的值，初始化还可以**用numpy的矩阵**进行初始化。其之所以称之为变量是因为其是随着迭代进行不断变化的量，可以理解成机器学习中那个权值矩阵w，有一个好的习惯是对操作进行命名name="A"，这种命名使得在观察计算图的时候结构更加清晰。

第二个代表的是常量，初始化依然可以用变量的初始化方式，它代表不变的量。

第三个是placeholder，在训练神经网络的过程中需要不断的输入训练样本，而placeholder用于接收这些样本用于训练。

TensorFlow的python-API只是描述了一个计算，描述好之后的训练图是放到c/c++写成的内核中运行的，因此我们需要一个类似于placeholder的占位符，用于接收数据，而且不影响后续网络的搭建。前面说到TensorFlow的运行是基于数据流图的，这个"图"就是用python-API描述的，因此真正执行过程需要一个Session去运行这个计算图：

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


回想在c语言之中的过程，我们在定义矩阵变量之后只是定义了一个指针，并未分配内存，类似的，TensorFlow中也需要对Variable定义的变量进行初始化工作：

```
#变量初始化
init = tf.global_variables_initializer()
sess.run(init)
```

这里注意constant常量是不用进行初始化的，而在使用了初始化函数init之后，依然需要用sess.run去执行。

之后来看下矩阵的值：

```
print(sess.run(A.value()))
print(sess.run(B))
print(sess.run(C, feed_dict={C:[[3, 3], [3, -3]]}))
```

可以看到，A.value()是获取变量的值，但是这个值获取函数**并未执行**，依然需要run一下。第三个placeholder，用到其变量需要对placeholder填入数值，这就是说需要用到feed机制，如上面feed_dict就是输入变量的字典，C输入的是一个二维列表。这是训练中不断输入训练数据的过程。

TensorFlow中定义了很多矩阵相关的计算，比如矩阵的加法，矩阵乘法，矩阵连接，以及矩阵通过激活函数，后面这个矩阵通过激活函数是附加的，这种操作还是经常看到的：

$$\vec{y} = active function(\vec{x})$$

这种操作就是对矩阵中每个分量都进行激活函数的计算。

来看下TensorFlow的代码：

```
#格式转换
B2 = tf.cast(B, tf.float32)
#矩阵减法
D = C - B2
#矩阵乘法
E = tf.matmul(B2, C)
#矩阵连接
F = tf.concat([B2, C], axis=1)
#矩阵通过激活函数
G = tf.nn.relu(C)

sess =  tf.Session()
sess.run(tf.global_variables_initializer())

for itr in sess.run([D, E, F, G], feed_dict={C:[[3, 3], [3, -3]]}):
    print(itr)
```

前面定义的矩阵中初始化用到的是[[2, 2], [-2, 2]]，这就使得矩阵类型是int类型，显然之后再定义placeholder是float32类型，TensorFlow与python不同，其对于类型的检查是比较严格的，因此需要对其进行转化否则会报错，也就是cast函数。同时也是前面说的，在进行计算的时候如果有placeholder则需要在run的时候提供相应的数值。

可以看到，矩阵相乘和矩阵相加符合传统的运算方式，而矩阵通过激活函数是那个常提到的relu激活函数：

$$ReLU(x) = max(0, x)$$

大于0是线性变换，小于等于零是0，来看下最终输出：

```
[[ 1.  1.]
 [ 5. -5.]]
[[ 12.   0.]
 [  0. -12.]]
[[ 2.  2.  3.  3.]
 [-2.  2.  3. -3.]]
[[ 3.  3.]
 [ 3.  0.]]
```


这一系列的运算并**没有违反直觉**的地方。

三、第一个例子-解方程

第一篇文章其实一直在说**万物基于矩阵逆**，逆矩阵可以帮助完成几乎所有的**数值模拟**任务。那么首先来看下下面的这个方程：

$$2x + 2y = 1$$
$$-2x + 2y = 1$$

当然这种情况需要写成矩阵的形式，以便于与tensorflow的矩阵对应：

$$[x \quad y] \cdot \begin{bmatrix} 2 & 2 \\ -2 & 2 \end{bmatrix} = [1 \quad 1]$$
$$x \cdot A = b$$

上面这个方程可以通过TensorFlow，用**三种不同的方法**来求解，这个过程可以熟悉TensorFlow的用法，更重要的是领略不同loss函数的构建过程。

3.1 直接求解

首先是没有用到loss函数的，就是两边同时左乘以A^{-1}，TensorFlow是有矩阵逆的函数的：

```
import tensorflow as tf
A = tf.constant([[2., 2.], [-2., 2.]], name="B")
b = tf.constant([[1., 1.]], name='b')
invA = tf.matrix_inverse(A)
result = tf.matmul(b, invA)
sess = tf.Session()
print(sess.run(result))
```

输出结果为：

```
[[ 0.5 0. ]]
```

tensorflow中自带矩阵逆函数，这里处理还是比较方便的，但是更多的情况这个矩阵逆是用不到的，因为其算法复杂度是比较高的。还应注意到的一点就是由于均为常量操作，因此并不存在初始化的过程。

3.2定义损失函数求解

上面是第一种方式，第二种方式是**基于迭代的**，显然这个过程想求的是**数值解**，那么这个过程需要不断调整的就是变量x，使其乘以矩阵A后更好的接近于b，用程序来描述：

```
import tensorflow as tf
A = tf.constant([[2., 2.], [-2., 2.]], name="B")
b = tf.constant([[1., 1.]], name='b')
x = tf.Variable([[1., 1.]], name='x')
#x*A
M = tf.matmul(x, A)
#(b-x*A)^2
loss = tf.square(M-b)
step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

sess =  tf.Session()
sess.run(tf.global_variables_initializer())
#运行迭代
for itr in range(100):
    sess.run(step)
print(sess.run(x.value()))
```


首先是变量定义，可以看到除x外A,b均为常量，这是由于我们想调整不断改变的仅仅是x，而其他的量不发生变化。之后$\boldsymbol{x} \cdot \boldsymbol{A}$得到的向量与b做减法，并求平方当做损失函数。显然我们想通过不断的调整x使得其接近于b，而最接近的情况就是方程的解:

$$\begin{gathered} y = x \cdot A \\ loss = \sum (y-b)_i^2 \\ loss \to 0 \end{gathered}$$

这个接近的过程就是训练的过程：

```
tf.train.GradientDescentOptimizer(0.1).minimize(loss)
```

这里GradientDescentOptimizer就是常说的梯度下降法，后面的0.1是选择的步长，而需要进行最小化的部分就是loss函数，用到了minimize。在此将其定义为迭代的step，之后用sess去运行这个迭代过程，迭代持续了100步，最后print用来观察迭代迭代结果：

```
[[  5.00000000e-01   3.01552028e-09]]
```

显然**数值解是一个接近于真实解的数值。**

3.3 定义常规形式的泛函求解

前面说过，**代价函数可以称之为泛函**，那么在最优化算法中有一种定义泛函的方式，就是可以将求解方程问题转换为求解函数最小值问题，在3.2节中是一种最小值，但是与常规泛函还是有区别，这里定义一个常规的泛函：

$$f(\vec{x}) = \vec{x} \cdot A \cdot A^T \cdot \vec{x} - 2b \cdot A^T \cdot \vec{x}$$

这是我们最常见到的泛函形式，其最小值点就是方程

$$\vec{x} \cdot A = b$$

的解，这里泛函输出的是一个标量。

将泛函写成loss函数的形式，并完成程序的编写：

```
A = tf.constant([[2., 2.], [-2., 2.]], name="B")
b = tf.constant([[1., 1.]], name='b')
x = tf.Variable([[1., 1.]], name='x')
#xAAtx
xA = tf.matmul(x, A)
xt=tf.transpose(xA)
xAAx = tf.matmul(xA, xt)
#bx
bAx = tf.matmul(b, xt)
#loss function
fx = xAAx - 2 * bAx

step = tf.train.GradientDescentOptimizer(0.1).minimize(fx)

sess =  tf.Session()
sess.run(tf.global_variables_initializer())
for itr in range(100):
    sess.run(step)
    print(sess.run(fx))
print(sess.run(x.value()))
```

运行下看下输出:

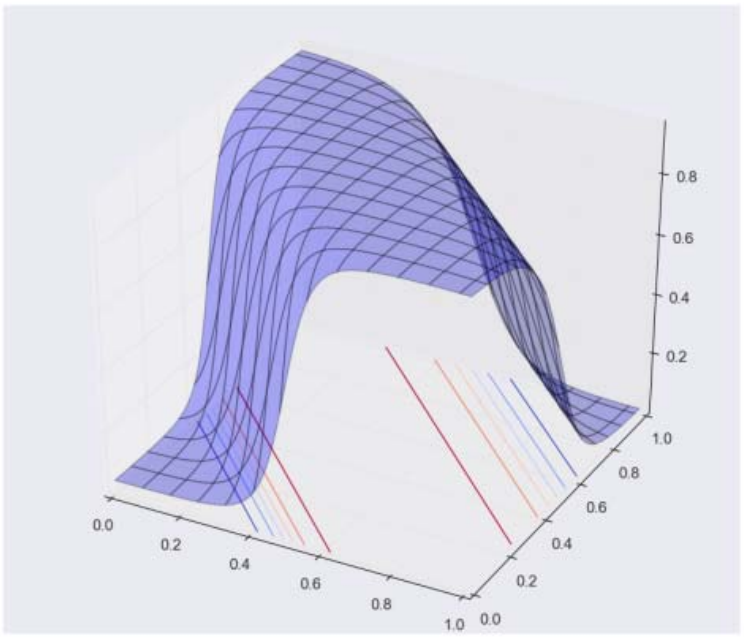
```
[[  5.00000000e-01   1.13164980e-08]]
```

可以看到，同样也得到了应得的结果。

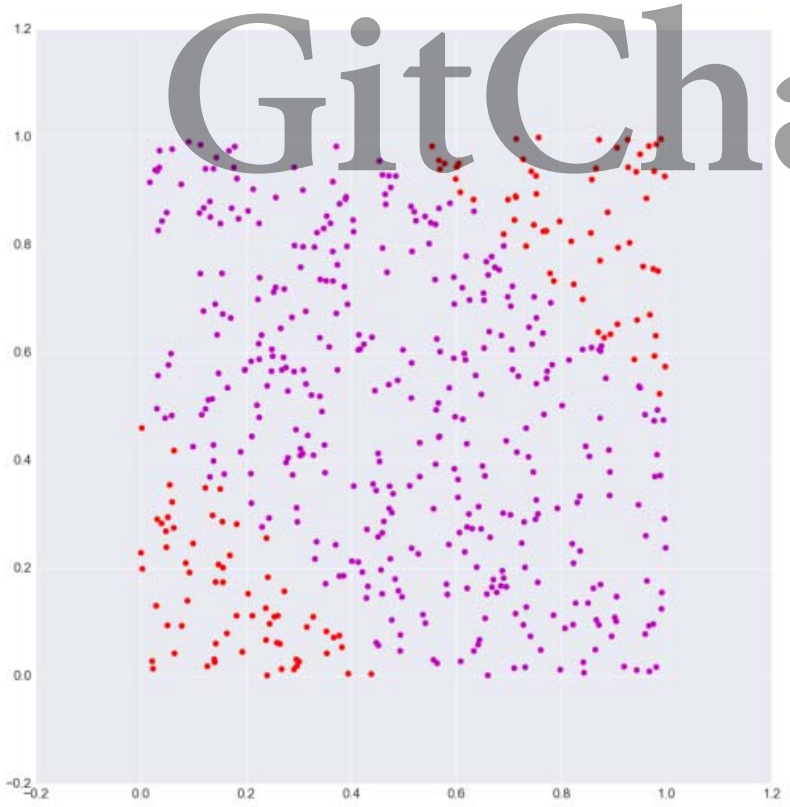
四、曲面拟合

4.1 抑或问题

我们前面分析过，很多神经网络其实都可以当成是曲线拟合问题。多维曲面的拟合其实是比较有意思的议题。我们来看下具体的曲面拟合的问题：



比如这种扭曲了两次的曲面，这种曲面其实是有现实意义的，比如可以解决线性不可分的抑或问题：

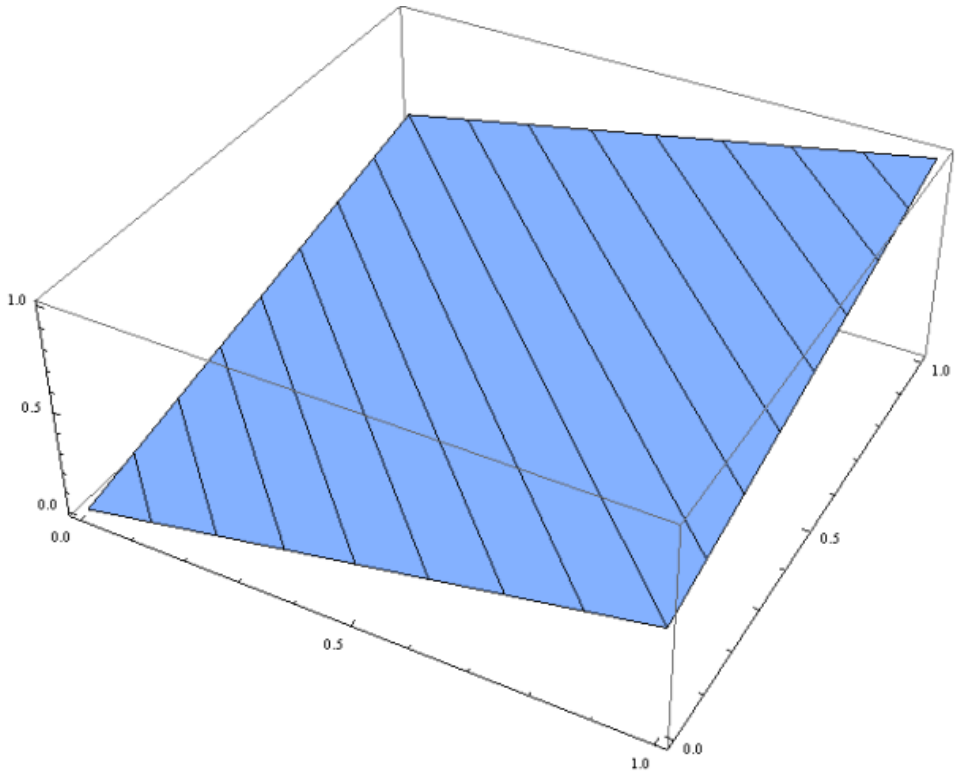


什么是线性可分，就是用一个平面，或者直线将下面两种不同的点划分为两类。显然上面那个马鞍形状的曲面如果以0.5为阈值的话，是可以对这种线性不可分问题进行求解的，这类线性不可分问题也可以称为“抑或问题”，再具体一些的“抑或问题”：两个开关，只有两个开关其中一个打开的情况时灯才会亮，有些像你死我活的问题。

那么下面来看下如何的构建这种网络，前面说过，这是一个曲面拟合的问题：

$$z = f(x, y)$$

这个曲面要拟合的就是本节中开始的那个曲面，显然进行单纯的仿射变换是无法把曲面进行扭曲的，不管多少次仿射变换，我们得到的曲面都是一个平面：

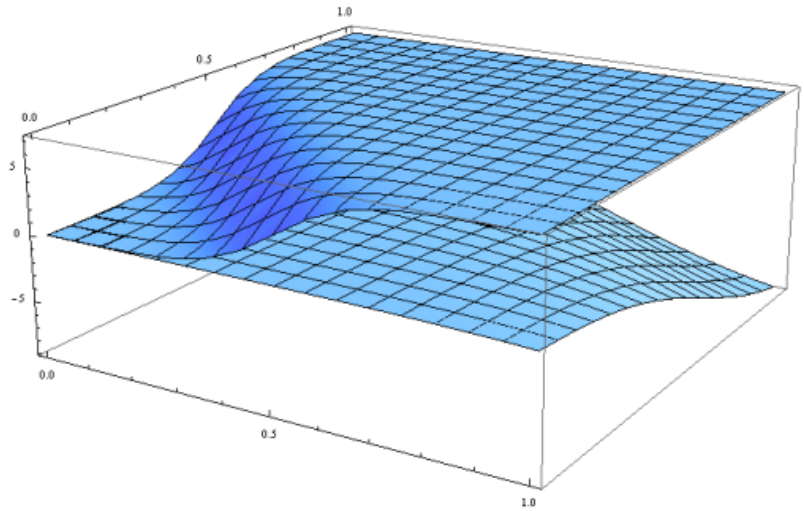


我们再来回顾一下什么是仿射变换：

$$y = A \cdot x$$

A是一个矩阵。

此时激活函数的作用就体现出来了：将曲面进行扭曲，在加入两层神经网络之后可以看成两个曲面的叠加：



我们用TensorFlow来实现一下上面所描述的过程：

```
import tensorflow as tf
#定义placeholder用来接收输入数据
#shape里有个None，也就是每次可以接收多个样本
x=tf.placeholder(tf.float32,shape=[None,2])
z=tf.placeholder(tf.float32,shape=[None,1])
#第一层全链接结构定义f(w1*x+b)
W1=tf.Variable(tf.truncated_normal([2,2],stddev=0.1))
b1=tf.Variable(tf.constant(0.1,shape=[2]))
fc1=tf.nn.sigmoid(tf.matmul(x,W1)+b1)
#第二层全链接结构定义f(w1*x+b)
```

```
W2=tf.Variable(tf.truncated_normal([2,1],stddev=0.1))
b2=tf.Variable(tf.constant(0.1,shape=[1]))
fc2=tf.nn.sigmoid(tf.matmul(fc1,W2)+b2)
#loss函数定义为简单的向量的二范数
ce=tf.reduce_mean(tf.square(fc2-y))
ts=tf.train.AdamOptimizer(1e-2).minimize(ce)

sess=tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
#训练过程
for i in range(10000):
    train_x, train_z =gen_data()
    sess.run(ts,feed_dict={x:train_x,z:trian_z})
```

这个过程就是对于抑或问题的两层全链接网络结构，那么简单来说全链接网络其实就是向量乘以矩阵，再套上一层激活函数。其实分类问题合理的角度应该有长度为2的输出，但是为了契合曲面拟合问题，输出为1。这里需要解释的一个问题就是代码placeholder里shape里有个None，这就代表一次可以输入多个样本，这个个数称之为BATCHSIZE，可以通过统计更好的估计梯度的方向。
而最终得到的曲面图形就是本节开始那个二次扭曲的曲面。

4.2 手写识别

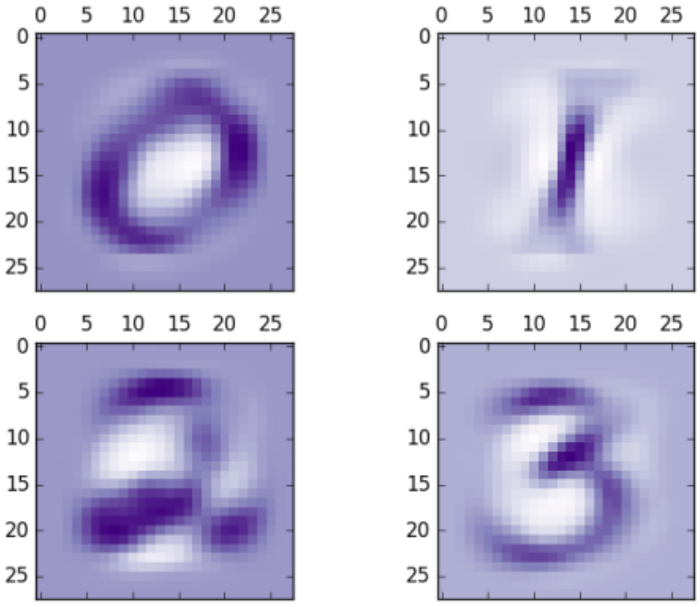
再来看一个全链接网络的例子,这是tensorflow自带的手写识别的例子，其值用到了一层神经网络：

```
#引入库
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
#获取数据
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
#构建网络模型
#x，label分别为图形数据和标签数据
x = tf.placeholder(tf.float32, [None, 784])
label = tf.placeholder(tf.float32, [None, 10])
#构建单层网络中的权值和偏置
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
#本例中无非线性激活函数
y = tf.matmul(x, W) + b
#定义损失函数为欧氏距离
loss = tf.reduce_mean(tf.square(y-label))
#用梯度迭代算法
train_step = tf.train.GradientDescentOptimizer(0.005).minimize(loss)
#用于验证
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(label, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
#定义会话
sess = tf.Session()
#初始化所有变量
sess.run(tf.global_variables_initializer())
#迭代过程
for itr in range(3000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, label: batch_ys})
    if itr % 10 == 0:
        print("step:%6d  accuracy:"%itr, sess.run(accuracy, feed_dict={x:
mnist.test.images,
                                        label: mnist.test.labels}))
```


可以看到，其是没有激活函数的，也就是一个线性变换就可以解决问题，其数学形式：

$$y = x \cdot W + b$$

可以挑一些权值绘制成图：



显然，这种所谓的权值不过就是对于空间数据的加权求和。

五、计算图

前面说到要对变量进行命名，也就是name，其为我们方便的查看计算图提供了基础。

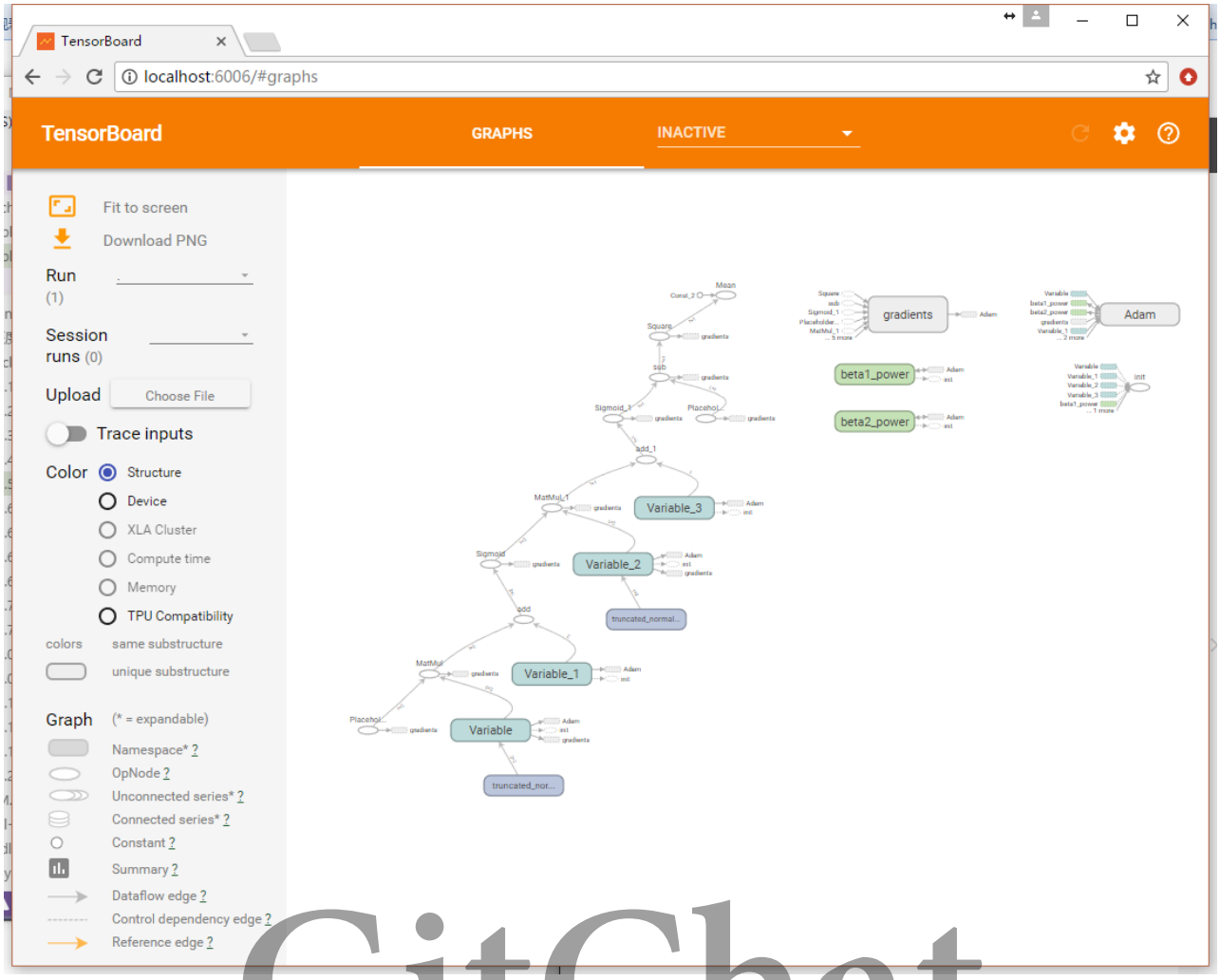
在前面的抑或问题的代码中加入：

```
train_writer = tf.summary.FileWriter("logdir-xor", sess.graph)
```

运行后再通过脚本运行：

```
tensorboard --logdir=logdir-xor
```

在浏览器中输入localhost:6006(这个port是可以改的)就可以查看所描述的计算图了：

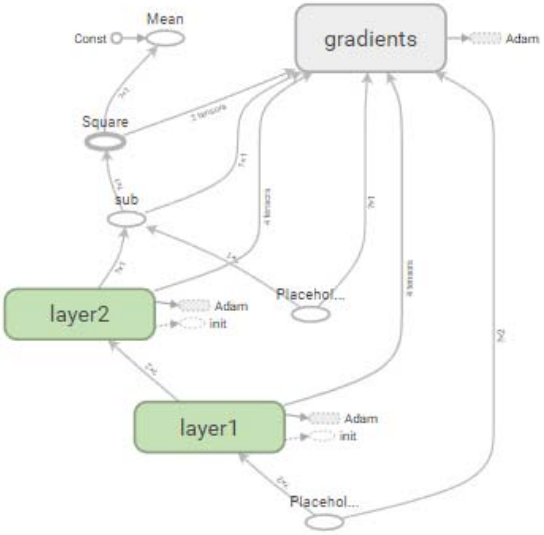


可以看到结构还是比较乱的，因此在定义的时候命名的作用就显现出来了，我们对每个变量进行命名：

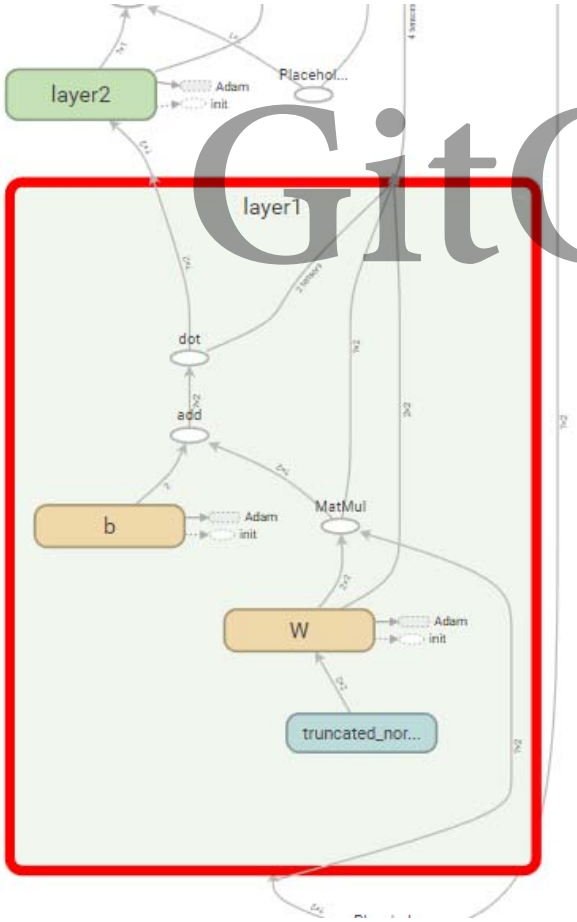
```
with tf.variable_scope("layer1"):
    W1=tf.Variable(tf.truncated_normal([2,2],stddev=0.1),name="W")
    b1=tf.Variable(tf.constant(0.1,shape=[2]),name="b")
    fc1=tf.nn.sigmoid(tf.matmul(x,W1)+b1,name="dot")
with tf.variable_scope("layer2"):
    W2=tf.Variable(tf.truncated_normal([2,1],stddev=0.1),name="W")
    b2=tf.Variable(tf.constant(0.1,shape=[1]),name="b")
    fc2=tf.nn.sigmoid(tf.matmul(fc1,W2)+b2,name="dot")
```

编辑器问题，前面需要有缩进。

此时运行一下再来看下运算图：



可以看到此时的结构是很清晰的，可以再点开看一下：



当然 `tf.variable_scope("layer1")` 的作用远不止于此，其最大的作用在于标识不同的变量，常常配合 `get_vairable` 使用：

```
improt tensorflow as tf
with tf.variable_scope("first-nn-layer") as scope:
    W = tf.get_variable("W", [784, 10])
    b = tf.get_variable("b", [10])
with tf.variable_scope("second-nn-layer") as scope:
```

```
        W = tf.get_variable("W", [784, 10])
        b = tf.get_variable("b", [10])
    with tf.variable_scope("second-nn-layer", reuse=True):
        W3 = tf.get_variable("W", [784, 10])
        b3 = tf.get_variable("b", [10])
    print(W.name)
    print(W3.name)
```

对于同一scope下的同名变量，代表是一个变量。在之后用到的时候需要将其改成复用的模式。