**AI**

**实 验 报 告**

**学生姓名** 吴语港

**学生学号** SA19225404

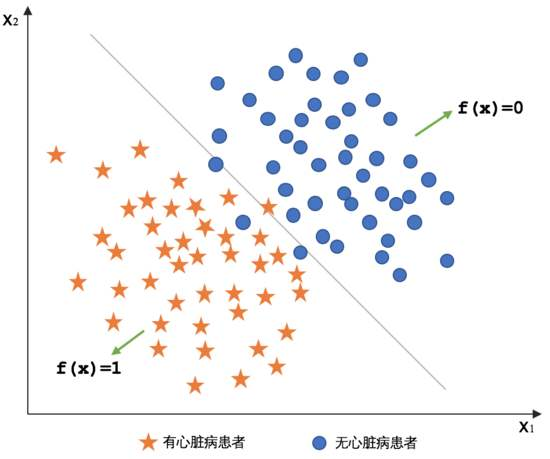
**实验日期** 2019/10/16

**逻辑回归工作原理**

机器学习的任务主要分为两种，一类是数值预测，另一类是分类问题。前者一般采用的是回归模型，一般是线性回归，而后者的方法则丰富了很多，比如SVM支持向量机，KNN K近邻算法，决策树，朴素贝叶斯模型等等。其实，两种问题从本质上讲是一样的：都是通过对已有数据的学习，构建模型，然后对未知的数据进行预测，若是连续的数值预测就是回归问题，若是离散的类标号预测，就是分类问题。

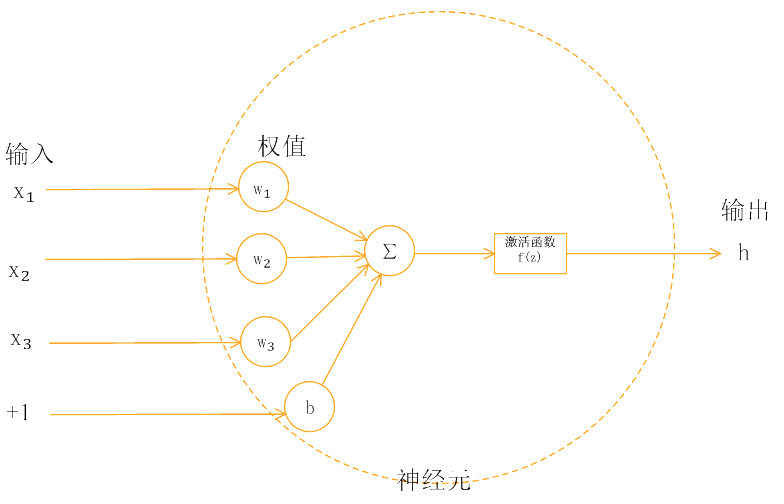
这里面有一类比较特殊的算法，就是逻辑回归（logistic regression）。它叫“回归”，可见基本思路还是回归，但是，逻辑回归又是标准的解决分类问题的模型。换句话说，逻辑回归是用与回归类似的思路解决了分类问题。

Logistic回归工作原理：根据现有数据对分类边界线建立回归公式，以此进行分类。‘回归’源于最佳拟合，表示要找到最佳拟合参数集。训练分类器时的做法就是寻找最佳拟合参数，使用的是最优化算法。见图1，Logistic回归就是为了找到这样的分类边界（Decision Boundary）备注：如果是二维情况，那么就是找到如图0中的分类曲线，如果是多维，那么就是找到可以分类的平面，比如下面这个例子



**神经元工作原理**

神经网络中的每个神经元节点接受上一层神经元的输出值作为本神经元的输入值，并将输入值传递给下一层，输入层神经元节点会将输入属性值直接传递给下一层（隐层或输出层）。在多层神经网络中，上层节点的输出和下层节点的输入之间具有一个函数关系，这个函数称为激活函数（又称激励函数）。



**给出至少2种常用的激活函数；**

早期研究神经网络主要采用sigmoid函数或者tanh函数，输出有界，很容易充当下一层的输入。

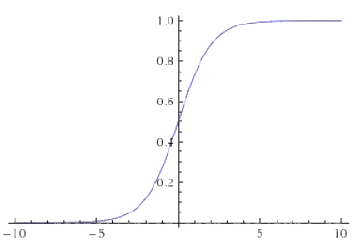
近些年Relu函数及其改进型（如Leaky-ReLU、P-ReLU、R-ReLU等）在多层神经网络中应用比较多。下面我们来总结下这些激活函数：

Sigmoid函数

Sigmoid 是常用的非线性的激活函数，它的数学形式如下：



Sigmoid的几何图像如下：



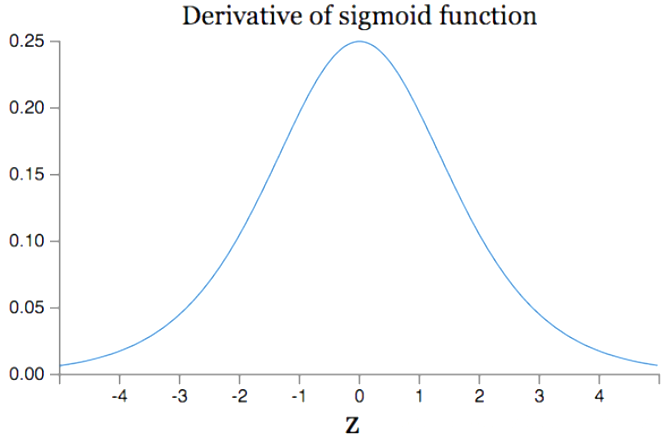
特点：

它能够把输入的连续实值变换为0和1之间的输出，特别的，如果是非常大的负数，那么输出就是0；如果是非常大的正数，输出就是1.

缺点：

sigmoid函数曾经被使用的很多，不过近年来，用它的人越来越少了。主要是因为它固有的一些 缺点。

缺点1：在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。首先来看Sigmoid函数的导数，如下图所示：



如果我们初始化神经网络的权值为 [0,1][0,1][0,1] 之间的随机值，由反向传播算法的数学推导可知，梯度从后向前传播时，每传递一层梯度值都会减小为原来的0.25倍，如果神经网络隐层特别多，那么梯度在穿过多层后将变得非常小接近于0，即出现梯度消失现象；当网络权值初始化为 (1,+∞)(1,+∞)(1,+∞) 区间内的值，则会出现梯度爆炸情况。

缺点2：Sigmoid 的 output 不是0均值（即zero-centered）。这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非0均值的信号作为输入。 产生的一个结果就是：如x>0, f=wTx+bx&gt;0, \ f= w^Tx+bx>0, f=wTx+b,那么对w求局部梯度则都为正，这样在反向传播的过程中w要么都往正方向更新，要么都往负方向更新，导致有一种捆绑的效果，使得收敛缓慢。 当然了，如果按batch去训练，那么那个batch可能得到不同的信号，所以这个问题还是可以缓解一下的。因此，非0均值这个问题虽然会产生一些不好的影响，不过跟上面提到的梯度消失问题相比还是要好很多的。

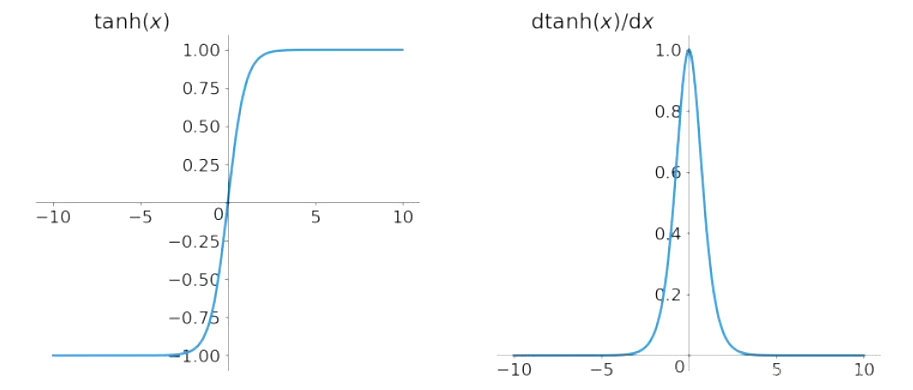
缺点3：其解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增加训练时间。

tanh函数

tanh函数解析式：



tanh函数及其导数的几何图像如下图：



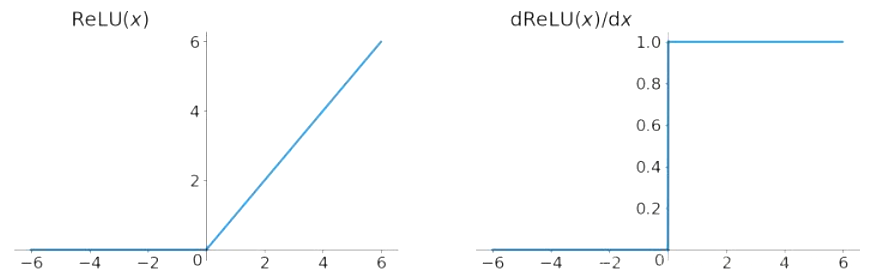
tanh读作Hyperbolic Tangent，它解决了Sigmoid函数的不是zero-centered输出问题，然而，梯度消失（gradient vanishing）的问题和幂运算的问题仍然存在。

Relu函数

Relu函数的解析式：



Relu函数及其导数的图像如下图所示：



ReLU函数其实就是一个取最大值函数，注意这并不是全区间可导的，但是我们可以取sub-gradient，如上图所示。ReLU虽然简单，但却是近几年的重要成果，有以下几大优点：

1） 解决了gradient vanishing问题 (在正区间)

2）计算速度非常快，只需要判断输入是否大于0

3）收敛速度远快于sigmoid和tanh

ReLU也有几个需要特别注意的问题：

1）ReLU的输出不是zero-centered

2）Dead ReLU Problem，指的是某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。有两个主要原因可能导致这种情况产生: (1) 非常不幸的参数初始化，这种情况比较少见 (2) learning rate太高导致在训练过程中参数更新太大，不幸使网络进入这种状态。解决方法是可以采用Xavier初始化方法，以及避免将learning rate设置太大或使用adagrad等自动调节learning rate的算法。

尽管存在这两个问题，ReLU目前仍是最常用的activation function

**简述卷积神经网络要素：**

**卷积核**

利用卷积操作可以从图像中提取特征，另外，卷积神经网络CNN，利用到了图像的局部相关性，这样可以减少全连接，减少需要训练的参数，卷积核相当于一个filter, 卷积核的大小一般采用 3x3 或 5x5，卷积操作, 相当于把卷积核作为一个 mask, 在像素矩阵上进行滑动, 然后进行加权求和作为特征值, 结果保存为新矩阵的对应像素点, 最后得到的新矩阵就是特征图，不同的卷积核, 可以提取图片的不同特征, 可以提升图片分类效果。

**滤波器**

CNN用滤波器来把图片分割成更小的 patch，patch 的大小跟滤波器大小相同，我们可以在水平方向，或者竖直方向滑动滤波器对图片的不同部分进行聚焦。

滤波器滑动的间隔被称作 stride（步长）。这是你可以调节的一个超参数。增大 stride 值后，会减少每层总 patch 数量，因此也减小了模型大小。通常这也会降低图像精度。这里最重要的是我们把相邻的像素聚在一起，把他们视作一个集合。

**池化**

池化也是一种提取特征的方式, 池化操作的过程和卷积很类似, 但是池化层还可以起到降维的作用。根据参考资料, 这样做的效果之一是增加特征的鲁棒性，减小过拟合，去除重复的特征，池化分为平均值池化和最大值池化, 平均池化就是取这个区域的平均值作为特征值, 最大池化则是取最大值作为特征值。池化层一般紧跟在卷积层后面。

**特征图**

每层有多少个卷积核，经过卷积就会产生多少个feature map，随着网络的加深，feature map的长宽尺寸缩小，本卷积层的每个map提取的特征越具有代表性（精华部分），所以后一层卷积层需要增加feature map的数量，才能更充分的提取出前一层的特征，一般是成倍增加，卷积特征的可视化，有助于我们更好地理解深度网络。卷积网络在学习过程中保持了图像的空间结构，也就是说最后一层的激活值（feature map）总和原始图像具有空间上的对应关系，具体对应的位置以及大小，可以用感受野来度量。

两种Padding方法：

same padding , 如果滑动窗口在经过某个 stride 后超出了图像范围, 则需要在外面补 0 以继续滑动提取特征值。输入输出窗口一样大

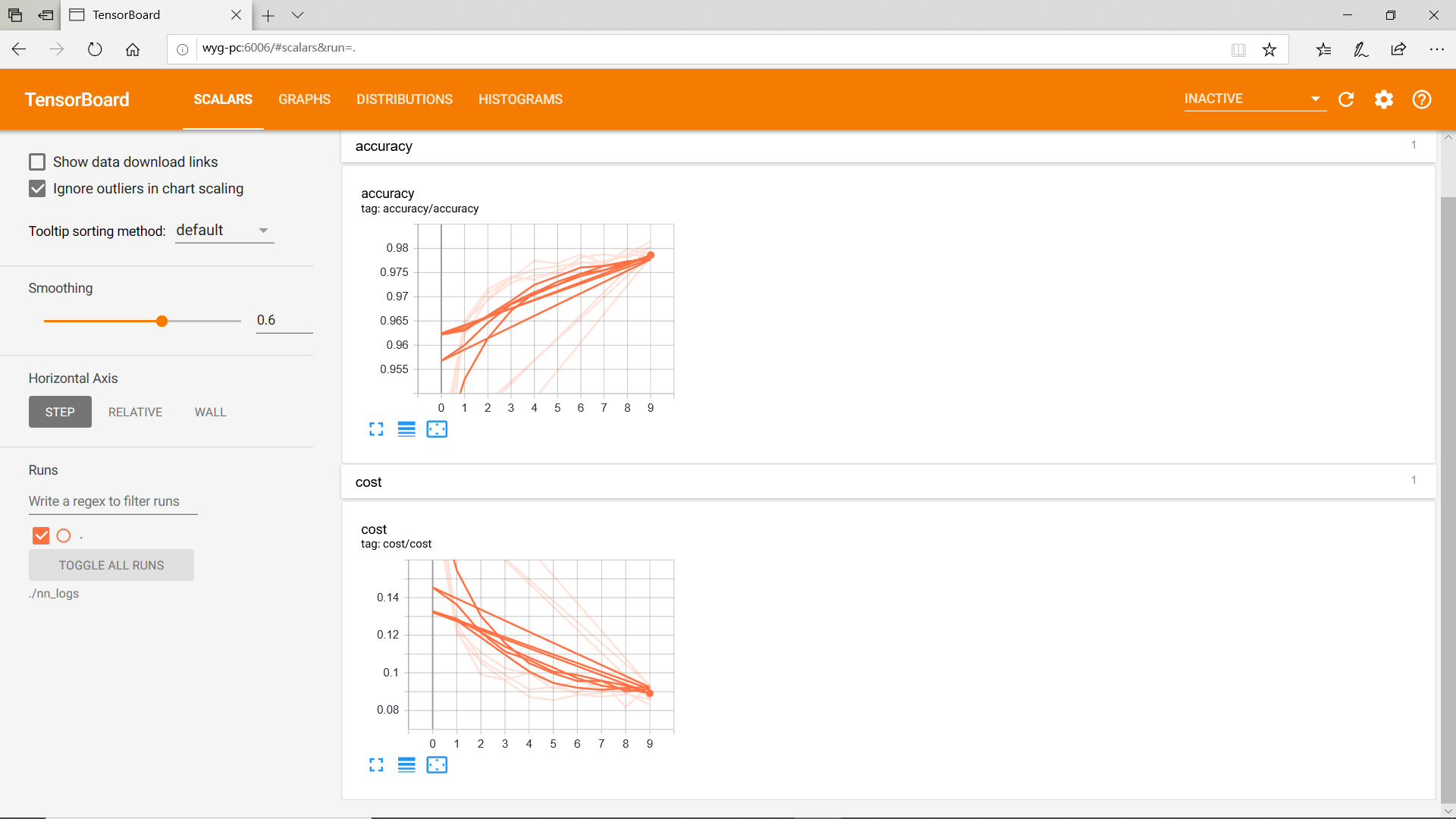
valid padding , 超出范围的就不要了。输入窗口大于输出。

**给出实验用到的模型图示**

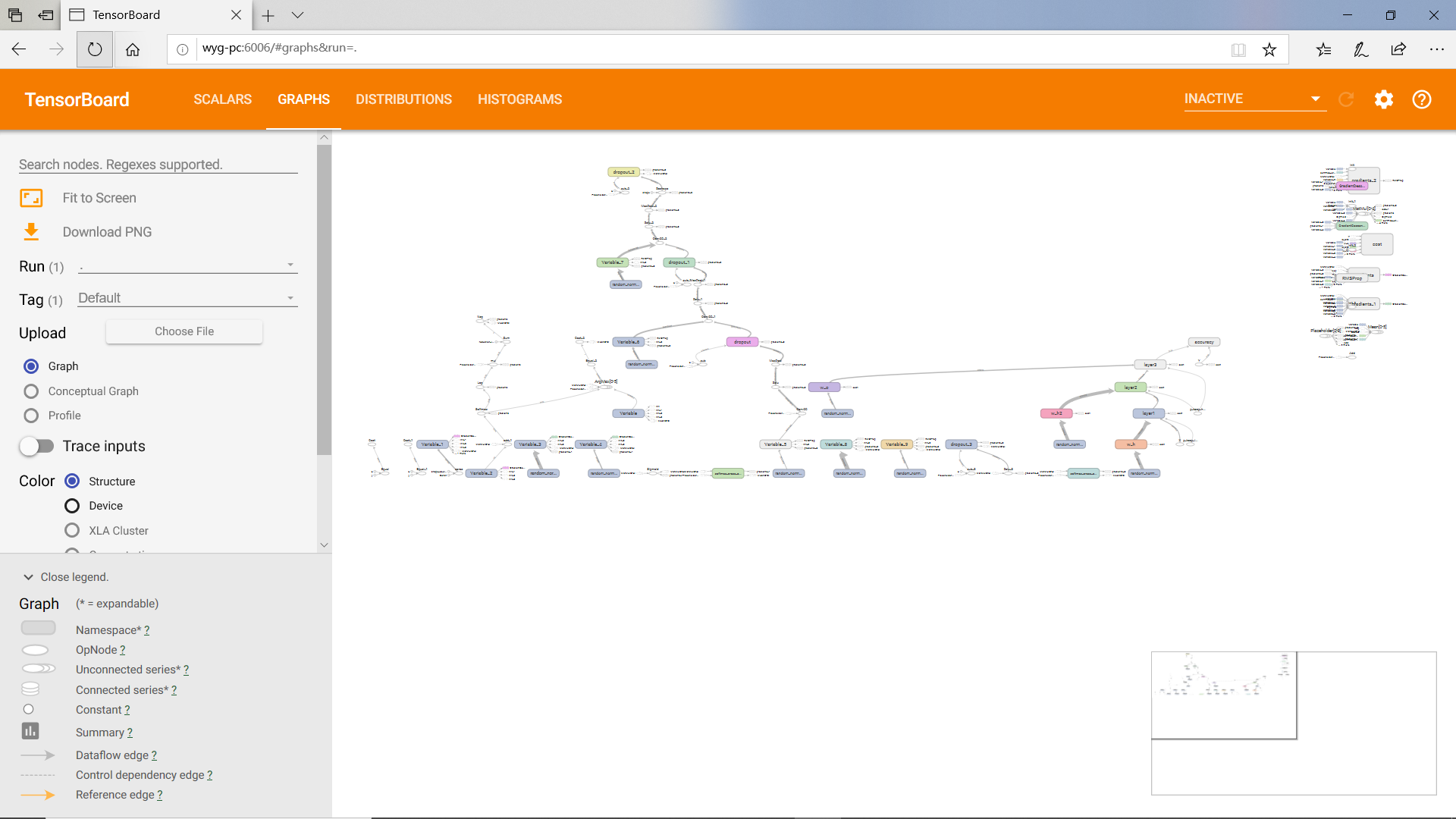
**启动TensorflowBoard**



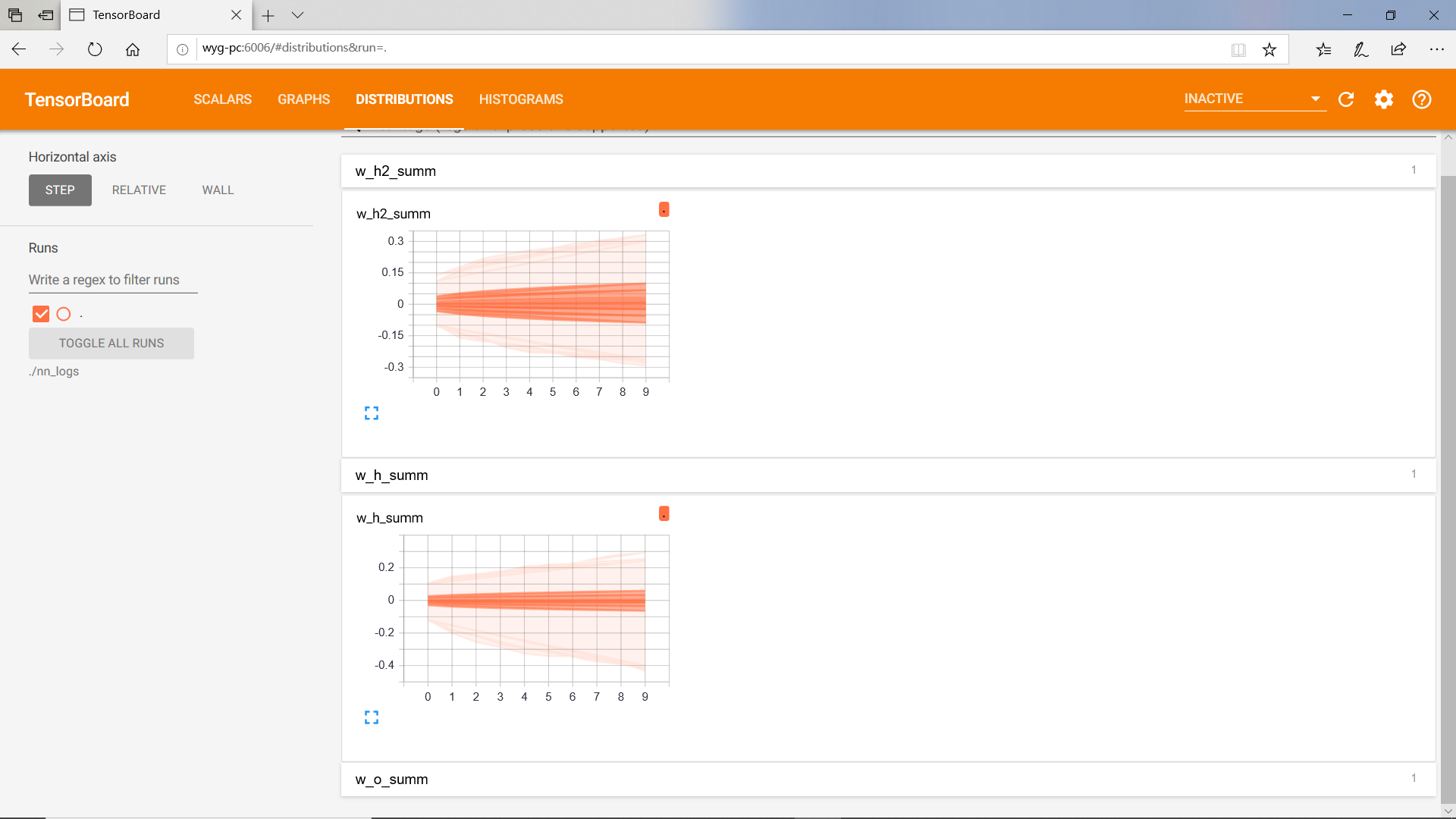
**标量**

****

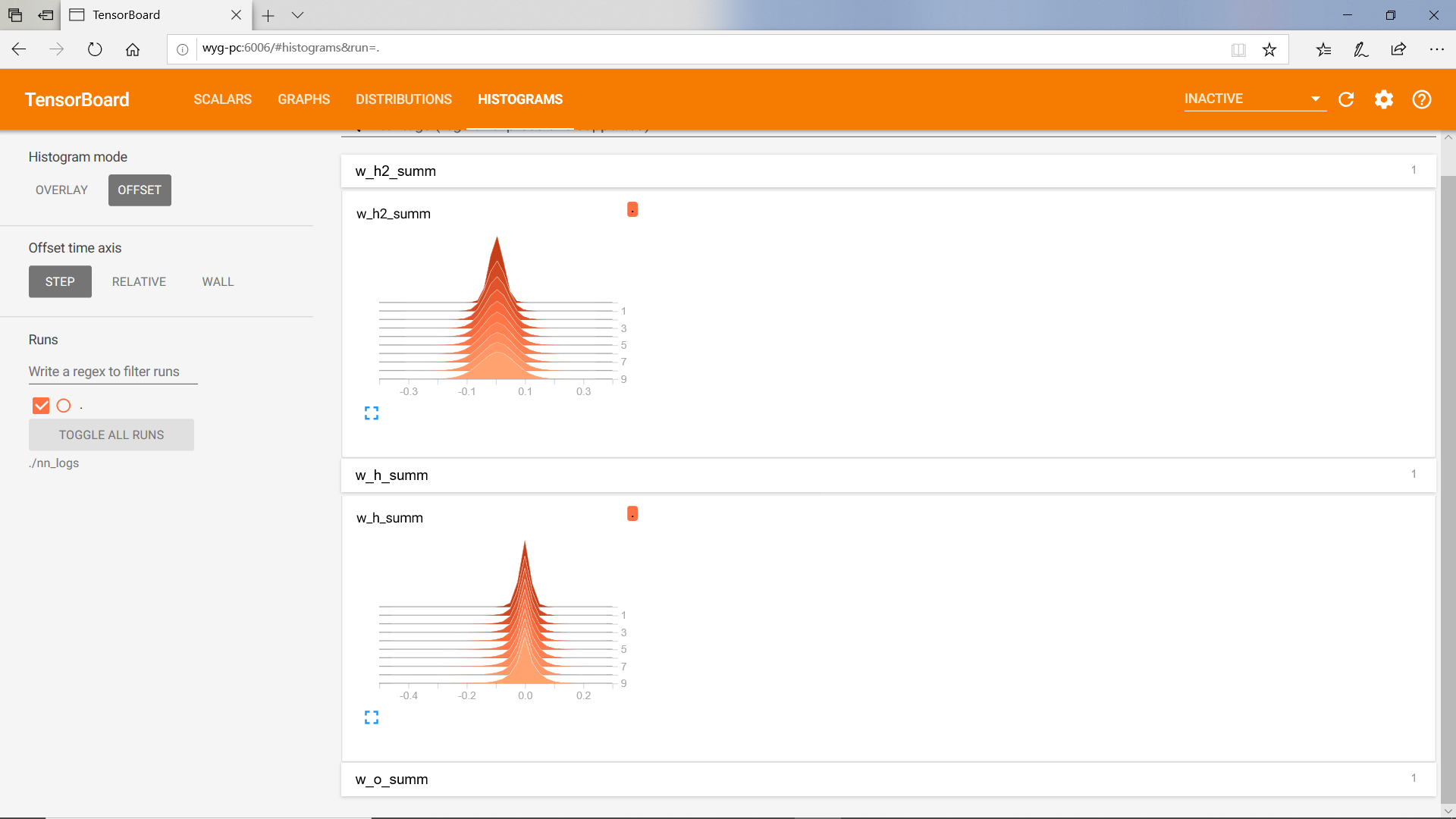
**图**

****

**分布**

****

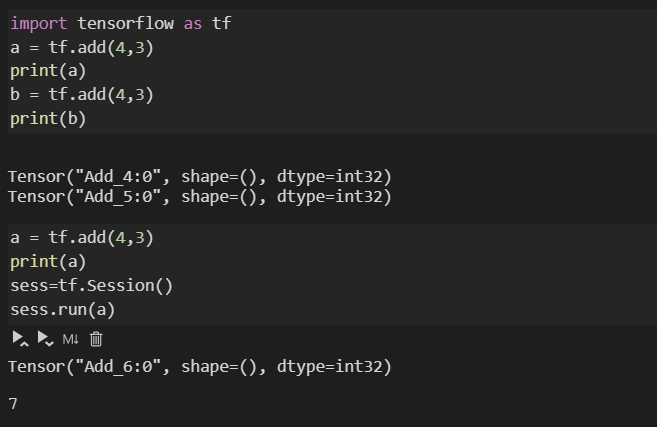
**直方图**

****

**给出实验中的关键代码并解释其实现原理**

**实验1**

**练习基本知识**



# 创建一个 常量 op, 返回值 'matrix1' 代表这个 1x2 矩阵.

matrix1 = tf.constant([[4., 3.]])

# 创建另外一个 常量 op, 返回值 'matrix2' 代表这个 2x1 矩阵.

matrix2 = tf.constant([[2.],[2.]])

# 创建一个矩阵乘法 matmul op , 把 'matrix1' 和 'matrix2' 作为输入.

# 返回值 'product' 代表矩阵乘法的结果.

product = tf.matmul(matrix1, matrix2)

# 启动默认图.

sess = tf.Session()

# 调用 sess 的 'run()' 方法, 传入 'product' 作为该方法的参数，

# 触发了图中三个 op (两个常量 op 和一个矩阵乘法 op)，

# 向方法表明, 我们希望取回矩阵乘法 op 的输出.

result = sess.run(product)

# 返回值 'result' 是一个 numpy `ndarray` 对象.

print(result)

# ==> [[ 14.]]

# 任务完成, 需要关闭会话以释放资源。

sess.close()

# 创建1个变量

W = tf.Variable(1, name="coef")

# 创建一个张量

a = tf.constant(3.0)

A = tf.Variable(tf.constant(0.0),dtype=tf.float32)

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

print(sess.run(A))

# 创建一个变量， 初始化为标量 Q. 初始化定义初值

A = tf.Variable(0,name="counter")

# 创建一个 oP} 其作用是使 A 增加 2

new\_value = tf.add(A, 2)

update = tf.assign(A,new\_value)

# 启动图变量初始化

init\_op = tf.global\_variables\_initializer()

# 启动默认亂运行 op

with tf.Session()as sess:

sess.run(init\_op)

#打印初始值

#with tfmSession()as sess:

print(sess.run(A))

# 重新赋值》 更新并打印

for \_ in range(3):

sess.run(update)

print(sess.run(A))

输出



**线性拟合**

import numpy as np

x\_data = np.float32(np.random.rand(2,100))

y\_data = np.dot([0.100,0.200],x\_data) + 0.300

b = tf.Variable(tf.zeros([1.0]))

W = tf.Variable(tf.random\_uniform([1,2],-1.0,1.0))

y = tf.matmul(W, x\_data) + b

# 最小化方差

loss = tf.reduce\_mean(tf.square(y - y\_data))

optimizer = tf.train.GradientDescentOptimizer(0.5)

train = optimizer.minimize(loss)

# 初始化变量

init = tf.global\_variables\_initializer()

# 启动图 (graph)

sess = tf.Session()

sess.run(init)

# 拟合平面

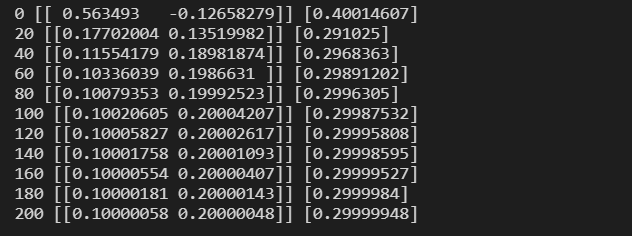
for step in range(0,201):

sess.run(train)

if step % 20 == 0:

print (step,sess.run(W),sess.run(b))

输出



**实验2**

**逻辑回归方法**

import tensorflow as tf

# 导入数据集

#from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

# 变量

batch\_size = 100

#训练的x(image),y(label)

# x = tf.Variable()

# y = tf.Variable()

x = tf.placeholder(tf.float32, [None, 784])

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

# 模型权重

#[55000,784] \* W = [55000,10]

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

b = tf.Variable(tf.zeros([10]))

# 用softmax构建逻辑回归模型

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

# 损失函数(交叉熵)

cost = tf.reduce\_mean(-tf.reduce\_sum(y\*tf.log(pred), 1))

# 低度下降

optimizer = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

# 初始化所有变量

init = tf.global\_variables\_initializer()

# 加载session图

with tf.Session() as sess:

sess.run(init)

# 开始训练

for epoch in range(25):

avg\_cost = 0.

total\_batch = int(mnist.train.num\_examples/batch\_size)

for i in range(total\_batch):

batch\_xs, batch\_ys = mnist.train.next\_batch(batch\_size)

sess.run(optimizer, {x: batch\_xs,y: batch\_ys})

#计算损失平均值

avg\_cost += sess.run(cost,{x: batch\_xs,y: batch\_ys}) / total\_batch

if (epoch+1) % 5 == 0:

print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg\_cost))

print("运行完成")

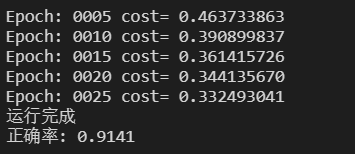
# 测试求正确率

correct = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

print("正确率:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))

输出



**人工神经网络方法**

import tensorflow as tf

import numpy as np

from tensorflow.examples.tutorials.mnist import input\_data

def init\_weights(shape):

return tf.Variable(tf.random\_normal(shape, stddev=0.01))

def model(X, w\_h, w\_o):

h = tf.nn.sigmoid(tf.matmul(X, w\_h)) # this is a basic mlp, think 2 stacked logistic regressions

return tf.matmul(h, w\_o) # note that we dont take the softmax at the end because our cost fn does that for us

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mnist.test.images, mnist.test.labels

X = tf.placeholder("float", [None, 784])

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

w\_h = init\_weights([784, 625]) # create symbolic variables

w\_o = init\_weights([625, 10])

py\_x = model(X, w\_h, w\_o)

cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=py\_x, labels=Y)) # compute costs

train\_op = tf.train.GradientDescentOptimizer(0.05).minimize(cost) # construct an optimizer

predict\_op = tf.argmax(py\_x, 1)

# Launch the graph in a session

with tf.Session() as sess:

# you need to initialize all variables

tf.global\_variables\_initializer().run()

for i in range(10):

for start, end in zip(range(0, len(trX), 128), range(128, len(trX)+1, 128)):

sess.run(train\_op, feed\_dict={X: trX[start:end], Y: trY[start:end]})

print(i, np.mean(np.argmax(teY, axis=1) ==

sess.run(predict\_op, feed\_dict={X: teX})))

输出



**卷积神经网络CNN方法**

# 加载TF 并加载数据

import tensorflow as tf

import numpy as np

from tensorflow.examples.tutorials.mnist import input\_data

#设置输入参数

batch\_size = 128

test\_size = 256

# 初始化权值与定义网络结构，建构一个3个卷积层和3个池化层

#一个全连接层和一个输出层的卷积神经网络

# 首先定义初始化权重函数

def init\_weights(shape):

return tf.Variable(tf.random\_normal(shape, stddev=0.01))

# 第一组卷积层以及池化层，最后　droupout是为了防止过拟合，在模型训练的时候丢掉一些神经元

# padding表示对边界的处理，SAME表示卷积的输入和输出保持同样尺寸

def model(X, w, w2, w3, w4, w\_o, p\_keep\_conv, p\_keep\_hidden):

l1a = tf.nn.relu(tf.nn.conv2d(X, w, # l1a shape=(?, 28, 28, 32)

strides=[1, 1, 1, 1], padding='SAME'))#水平滑动和竖直滑动的步长，填充输入输出图片大小不变

l1 = tf.nn.max\_pool(l1a, ksize=[1, 2, 2, 1], # l1 shape=(?, 14, 14, 32)

strides=[1, 2, 2, 1], padding='SAME')

l1 = tf.nn.dropout(l1, p\_keep\_conv)

# 第二组卷积层及池化层，最后dropout一些神经元

l2a = tf.nn.relu(tf.nn.conv2d(l1, w2, # l2a shape=(?, 14, 14, 64)

strides=[1, 1, 1, 1], padding='SAME'))

l2 = tf.nn.max\_pool(l2a, ksize=[1, 2, 2, 1], # l2 shape=(?, 7, 7, 64)

strides=[1, 2, 2, 1], padding='SAME')

l2 = tf.nn.dropout(l2, p\_keep\_conv)

# 第三组卷积神经网络及池化层，同样，最后dropout一些神经元

l3a = tf.nn.relu(tf.nn.conv2d(l2, w3, # l3a shape=(?, 7, 7, 128)

strides=[1, 1, 1, 1], padding='SAME'))

l3 = tf.nn.max\_pool(l3a, ksize=[1, 2, 2, 1], # l3 shape=(?, 4, 4, 128)

strides=[1, 2, 2, 1], padding='SAME')

l3 = tf.reshape(l3, [-1, w4.get\_shape().as\_list()[0]]) # reshape to (?, 2048)

l3 = tf.nn.dropout(l3, p\_keep\_conv)

# 全连接层

l4 = tf.nn.relu(tf.matmul(l3, w4))

l4 = tf.nn.dropout(l4, p\_keep\_hidden)

# 输出层

pyx = tf.matmul(l4, w\_o)

return pyx

# 导入数据

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

# 定义四个变量，分别为输入训练图像矩阵及其标签，输入测试图像矩阵及其标签

trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mnist.test.images, mnist.test.labels

# -1表示布考虑输入图片的数量，28\*28为图片的像素数，1是通道(channel)的数量，

# 因MNIST图片为黑白，彩色图片通道是3

trX = trX.reshape(-1, 28, 28, 1) # 28x28x1 input img

teX = teX.reshape(-1, 28, 28, 1) # 28x28x1 input img

# 10为识别图片的类别从0到9，共10个取值

X = tf.placeholder("float", [None, 28, 28, 1])

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

# 定义模型函数

# 神经网络模型的构建函数，传入以下参数

# X：输入数据

# w: 每一层权重

w = init\_weights([3, 3, 1, 32]) # 3x3x1 conv, 32 outputs

w2 = init\_weights([3, 3, 32, 64]) # 3x3x32 conv, 64 outputs

w3 = init\_weights([3, 3, 64, 128]) # 3x3x32 conv, 128 outputs

w4 = init\_weights([128 \* 4 \* 4, 625]) # FC 128 \* 4 \* 4 inputs, 625 outputs

w\_o = init\_weights([625, 10]) # FC 625 inputs, 10 outputs (labels)

# p\_keep\_conv,p\_keep\_hidden:dropout 保留神经元比例

# 定义dropout的占位符keep\_conv，表示一层中有多少比例的神经元被保留，生成网络模型，得到预测数据

# 在训练的时候把设定比例的节点改为0，避免过拟合

p\_keep\_conv = tf.placeholder("float")

p\_keep\_hidden = tf.placeholder("float")

py\_x = model(X, w, w2, w3, w4, w\_o, p\_keep\_conv, p\_keep\_hidden)

# 定义损失函数，采用tf.nn.softmax\_cross\_entropy\_with\_logists，作为比较预测值和真实值的差距

# 定义训练操作(train\_op) 采用RMSProp算法作为优化器,

cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=py\_x, labels=Y))

train\_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)

predict\_op = tf.argmax(py\_x, 1)

**启动图**

# Launch the graph in a session

with tf.Session() as sess:

# you need to initialize all variables

tf.global\_variables\_initializer().run()

for i in range(10):

training\_batch = zip(range(0, len(trX), batch\_size),

range(batch\_size, len(trX)+1, batch\_size))

for start, end in training\_batch:

sess.run(train\_op, feed\_dict={X: trX[start:end], Y: trY[start:end],

p\_keep\_conv: 0.8, p\_keep\_hidden: 0.5})

test\_indices = np.arange(len(teX)) # Get A Test Batch

np.random.shuffle(test\_indices)

test\_indices = test\_indices[0:test\_size]

print(i, np.mean(np.argmax(teY[test\_indices], axis=1) ==

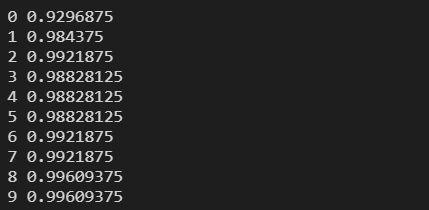
sess.run(predict\_op, feed\_dict={X: teX[test\_indices],

Y: teY[test\_indices],

p\_keep\_conv: 1.0,

p\_keep\_hidden: 1.0})))

**输出**



**TensorFlow可视化**

**模型**

def model(X, w\_h, w\_h2, w\_o, p\_keep\_input, p\_keep\_hidden):

# Add layer name scopes for better graph visualization

with tf.name\_scope("layer1"):

X = tf.nn.dropout(X, p\_keep\_input)

h = tf.nn.relu(tf.matmul(X, w\_h))

with tf.name\_scope("layer2"):

h = tf.nn.dropout(h, p\_keep\_hidden)

h2 = tf.nn.relu(tf.matmul(h, w\_h2))

with tf.name\_scope("layer3"):

h2 = tf.nn.dropout(h2, p\_keep\_hidden)

return tf.matmul(h2, w\_o)

**启动图**

with tf.Session() as sess:

# 创建log. 在终端运行 'tensorboard --logdir=./logs/nn\_logs'可以看到

writer = tf.summary.FileWriter("./logs/nn\_logs", sess.graph)

merged = tf.summary.merge\_all()

# 初始化变量

tf.global\_variables\_initializer().run()

for i in range(10):

for start, end in zip(range(0, len(trX), 128), range(128, len(trX)+1, 128)):

sess.run(train\_op, feed\_dict={X: trX[start:end], Y: trY[start:end],

p\_keep\_input: 0.8, p\_keep\_hidden: 0.5})

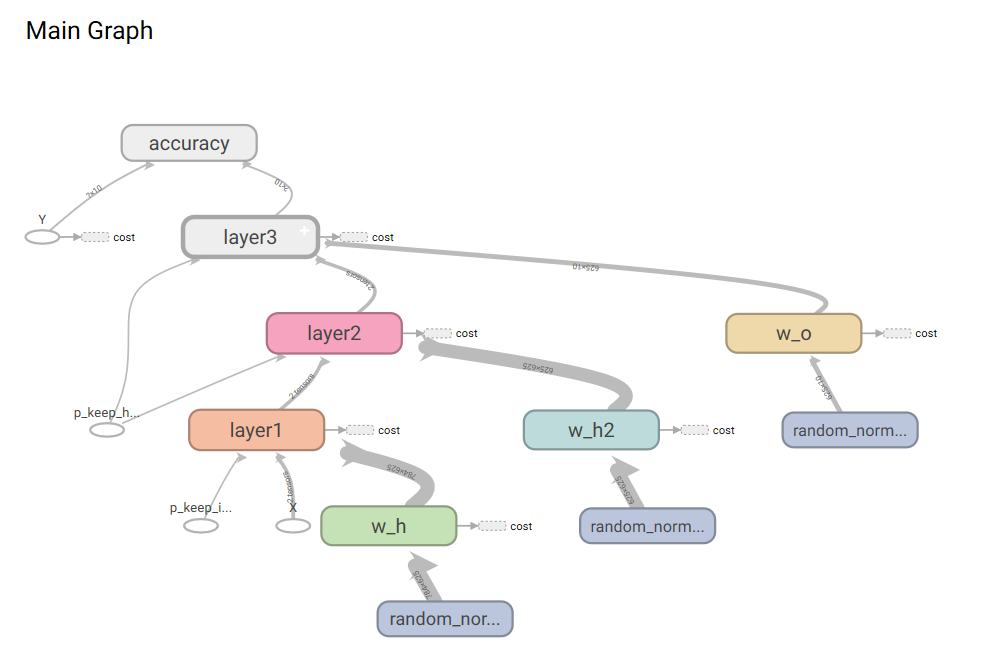
summary, acc = sess.run([merged, acc\_op], feed\_dict={X: teX, Y: teY,

p\_keep\_input: 1.0, p\_keep\_hidden: 1.0})

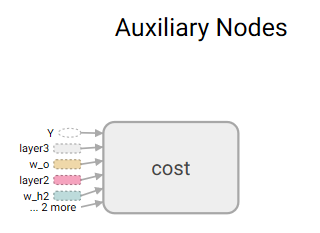
writer.add\_summary(summary, i) # 写入log

print(i, acc) #输出准确率

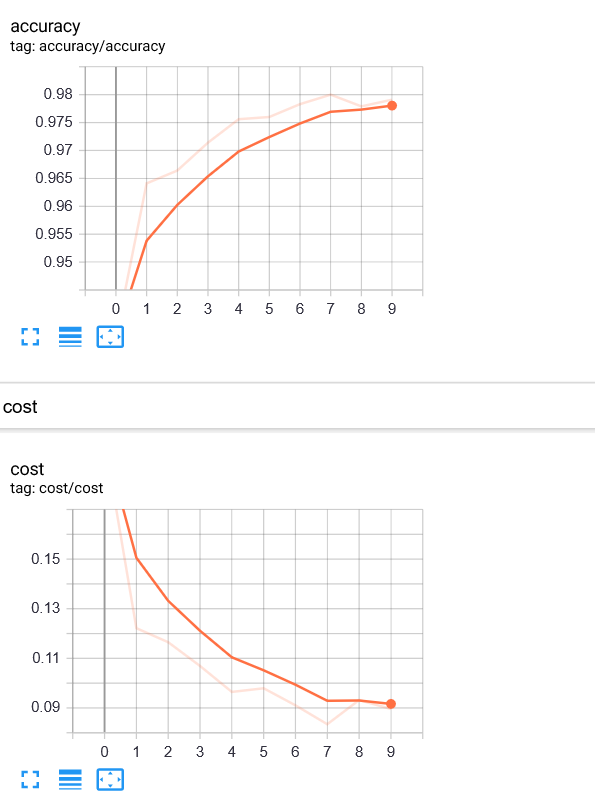
**主图**



**代价**



**准确度和代价函数**



**一些参数的直方图**

