# TensorFlow学习

## TensorFlow的执行时的基本对象及其具体理解：

分为Tensor、Variable和operation三大类，他们共同构成执行时的graph，执行可以执行graph中的一个子节点，子节点依照DAG顺序执行期上游全部父节点。

示例：下面讲解概念时有具体示例

## 常量操作和变量、另外也理解一下执行时的基本对象

**import**tensorflow**as**tf

m1 **=**tf.constant([[3, 3]]) *# 常量operation，生成一个Tensor*

m2 **=**tf.constant([[1], [2]]) *# 常量operation，生成一个Tensor*

product **=**tf.matmul(m1, m2) *# 加法operation，输入两个Tensor，生成一个Tensor*

​

**with**tf.Session() **as** session: *# 利用Session执行上述代码中生成的DAG*

res**=**session.run(product)

print(res)

[[9]]

In [2]:

**import**tensorflow**as**tf

state **=**tf.Variable([0, 0], "counter") *# 变量*

new\_value**=**tf.add(state, [1, 2]) *# 加法operation 生成一个Tensor*

upstate **=**tf.assign(state, new\_value) *# 赋值operation 生成一个Tensor*

​

init**=**tf.global\_variables\_initializer() *# 变量初始化*

​

**with**tf.Session() **as** session:

session.run(init) *# 变量初始化*

**for** \_ **in**range(10):

session.run(upstate) *# 执行更新赋值也就是“+=”10次*

print(session.run(state)) *# 查看每次执行的效果*

[1 2]

[2 4]

[3 6]

[4 8]

[ 5 10]

[ 6 12]

[ 7 14]

[ 8 16]

[ 9 18]

[10 20]

## Feed和fetch

*# fetch*

**import**tensorflow**as**tf

m1 **=**tf.constant([[3, 3]]) *# 常量operation，生成一个Tensor*

m2 **=**tf.constant([[1], [2]]) *# 常量operation，生成一个Tensor*

m3 **=**tf.constant([3])

product **=**tf.matmul(m1, m2) *# 加法operation，输入两个Tensor，生成一个Tensor*

add**=**tf.add(product, m3)

​

**with**tf.Session() **as** session: *# 利用Session执行上述代码中生成的DAG*

res **=**session.run([product, add]) *# 同一个run中运行多个operation，operation之间可以有依赖关系*

print(res)

[array([[9]]), array([[12]])]

[array([[9]]), array([[12]])]

*# feed*

**import**tensorflow**as**tf

input1 **=**tf.placeholder(tf.float32)

input2 **=**tf.placeholder(tf.float32)

add**=**tf.add(input1, input2)

​

**with**tf.Session() **as** session: *# 利用Session执行上述代码中生成的DAG*

res **=**session.run(add, feed\_dict**=** {input1: [32.1], input2: 78.2}) *# 同一个run中运行多个operation，operation之间可以有依赖关系*

print(res)

[110.299995]

## Tensorflow进行机器学习的一个简单案例——梯度下降求解线性回归模型

*# 构造一个线性回归的数据集*

**import**numpy**as**np

x\_data**=**np.random.rand(1000)

y\_data**=**x\_data**\***2.0**+**1.5

​

*# 利用tensorflow中的梯度下降法训练该线性回归*

**import**tensorflow**as**tf

​

*# 定义线性回归的两个变量：系数和常数项*

k **=**tf.Variable(0.0)

b **=**tf.Variable(0.0)

y **=**x\_data**\*** k **+** b

​

*# 梯度下降需要：学习率、损失函数 --梯度下降会自动根据损失函数求导，这一点很方便*

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

optimizer**=**tf.train.GradientDescentOptimizer(0.2)

train**=**optimizer.minimize(loss)

​

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

​

*# 运行梯度下降4000次并打印每次的结果和对应的损失函数*

**with**tf.Session() **as** session:

session.run(init)

**for** step **in**range(4000):

session.run(train)

print(session.run([k, b, loss]))

运行结果：

[0.5802862, 1.0091515, 1.6426241]

[0.87479293, 1.4959297, 0.4374238]

[1.0294542, 1.7277478, 0.14771365]

[1.1155472, 1.8351988, 0.076466516]

[1.1678884, 1.8820568, 0.057430845]

[1.203488, 1.8994639, 0.0509492]

………

[1.9999962, 1.5000021, 1.2278321e-12]

[1.9999963, 1.500002, 1.1365984e-12]

[1.9999964, 1.500002, 1.0793286e-12]

[1.9999965, 1.5000019, 9.928555e-13]

[1.9999967, 1.5000019, 9.46713e-13]

[1.9999968, 1.5000018, 8.66649e-13]

[1.9999969, 1.5000018, 8.3117866e-13]

[1.9999969, 1.5000017, 7.964047e-13]

[1.999997, 1.5000017, 7.523795e-13]

[1.9999971, 1.5000017, 7.226504e-13]

[1.9999971, 1.5000015, 6.7916517e-13]

[1.9999973, 1.5000015, 6.461818e-13]

[1.9999973, 1.5000015, 6.461818e-13]

[1.9999973, 1.5000015, 6.461818e-13]

[1.9999973, 1.5000015, 6.461818e-13]

[1.9999973, 1.5000015, 6.461818e-13]

最后稳定在6.461818e-13处

## 二层神经网络进行回归训练

**import**tensorflow**as**tf

**import**numpy**as**np

**import**matplotlib.pyplot**as**plt

​

*# 创建数据：二次曲线的分布 + 噪声*

*# ----*

*# 创建从-0.5到0.5的等差数列，长度为200*

line\_data**=**np.linspace(**-**0.5, 0.5, 200)

*# 为创建的数组再增加一个轴*

x\_data**=**line\_data[:, np.newaxis]

noise**=**np.random.normal(0, 0.02, x\_data.shape)

​

y\_data**=**np.square(x\_data) **+** noise

​

*# 看下数据的类型和shape 注意python中float位数变换会导致数组变长或变短，需要astype*

print(noise.shape, noise.dtype)

print(x\_data.shape)

print(y\_data.shape, y\_data.dtype)

​

*# 定义两个placeholder用来feed数据，分别代表x和y --1列*

x **=**tf.placeholder(tf.float32, [**None**, 1])

y **=**tf.placeholder(tf.float32, [**None**, 1])

​

*# ----*

*# 构建神经网络进行训练*

*# 假定：*

*# --一个隐含层L1*

*# --十个神经元*

*# ----*

​

*# 定义第一层的weight和bias, 都是1\*10的矩阵, 初始化分别为正态随机和0.0*

weight\_1 **=**tf.Variable(tf.random\_normal([1, 10]))

bias\_1 **=**tf.Variable(tf.zeros([1, 10]))

a\_L1 **=**tf.matmul(x, weight\_1) **+** bias\_1

z\_L1 **=**tf.nn.tanh(a\_L1) *# 这里用tanh更好一些*

​

*# 定义第二层的weight和bias，10\*1的矩阵和1\*1的矩阵，初始化分别为正态随机和0.0*

weight\_2 **=**tf.Variable(tf.random\_normal([10, 1]))

bias\_2 **=**tf.Variable(tf.zeros([1, 1]))

a\_L2 **=**tf.matmul(z\_L1, weight\_2) **+** bias\_2

y\_head**=**tf.nn.tanh(a\_L2) *# 这里也是用tanh更好一些*

*# 定义二次损失函数并依据梯度下降法进行训练 -- 这样梯度下降的train就变成了x和y的函数*

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

optimizer**=**tf.train.GradientDescentOptimizer(0.1)

train**=**optimizer.minimize(loss)

​

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

​

*# 运行梯度下降4000次并打印每次的结果和对应的损失函数*

**with**tf.Session() **as** session:

session.run(init)

i**=**0

**for** step **in**range(2000):

session.run(train, feed\_dict**=** {x: x\_data, y: y\_data}) *# 此处是最小化*

ls**=**session.run(loss, feed\_dict**=** {x: x\_data, y: y\_data})

*# 注意这里不是ls = session.run(loss), 因为loss也是DAG, 也是x和y的函数，因此同样需要feed，*

*#tf中常量和运行结果可以直接打印，其他都要run后打印*

**if**(i**%**20**==**0):

print(step, ls) *# 每隔二十次打印损失，观察收敛情况*

i**+=**1

*# 训练的神经网络模型中的两层的w和b*

print("weight\_1", session.run(weight\_1))

print("bias\_1", session.run(bias\_1))

print("weight\_2", session.run(weight\_2))

print("bias\_2", session.run(bias\_2))

*# 预测结果*

predict **=**session.run(y\_head, feed\_dict**=** {x: x\_data}) *# 这里y\_head即预测结果，注意同样是一个DAG，是x的函数，因此需要feed*

​

*# 利用matplot展示执行效果*

plt.figure()

plt.scatter(x\_data, y\_data) *# 训练数据用散点图展示*

plt.plot(x\_data, predict, 'r-', lw**=**3) *# 预测值用曲线展示，r表示红色，-表示实现，lw为曲线粗细*

plt.show()

​

​

(200, 1) float64

(200, 1)

(200, 1) float64

0 0.3595666

20 0.0062810197

40 0.005968654

60 0.0057035936

80 0.005450571

100 0.0052087354

120 0.0049773413

140 0.00475574

## 多分类逻辑回归进行手写数字识别

*# 数据集介绍*

*# MNIST数据集，100k的训练数据，10k的预测数据，数据由tensorflow中的examples.tutorials.mnist读取*

*# 数据集介绍：：YannLeCun's website*

*# 由28\*28的像素组成输入特征，输出特征为0-9的数字*

​

*# 可调节参数：*

*# --------*

*# batch\_size, initial\_weight,二次损失函数,learning\_rate,epoch\_n*

*# --------*

​

**import**tensorflow**as**tf

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

​

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

​

*# mini\_batch的大小*

batch\_size**=**100

batch\_n**=**mnist.train.num\_examples**//**batch\_size

​

*# # 定义两个placeholder用来feed数据，分别代表x和y --784列和10列(one-hot)*

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

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

​

*# # ----*

*# # 构建多分类回归*

​

*# # 定义weight和bias，初始化分别为正态随机和0.0*

initial\_weight**=**tf.random\_normal([784, 10])

weight**=**tf.Variable(initial\_weight)

bias**=**tf.Variable(tf.zeros([10]))

a **=**tf.matmul(x, weight) **+** bias

y\_head**=**tf.nn.softmax(a)

​

*# # 定义二次损失函数并依据梯度下降法进行训练 -- 这样梯度下降的train就变成了x和y的函数*

learning\_rate**=**0.1

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

optimizer**=**tf.train.GradientDescentOptimizer(learning\_rate)

train**=**optimizer.minimize(loss)

​

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

​

correct\_prediction**=**tf.equal(tf.argmax(y, 1), tf.argmax(y\_head, 1)) *# tf.argmax找到x中等于1的最大的id*

correction **=**tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32)) *# tf.cast转换类型，将bool转为float，从而求得准确率*

​

*# 迭代500次，进行mini\_batch梯度下降*

epoch\_n**=**500

**with**tf.Session() **as** session:

session.run(init)

**for** step **in**range(epoch\_n):

**for** batch **in**range(batch\_n):

batch\_x, batch\_y**=**mnist.train.next\_batch(batch\_size)

session.run(train, feed\_dict**=** {x: batch\_x, y: batch\_y}) *# 此处是最小化*

corr**=**session.run(correction, feed\_dict**=** {x: mnist.test.images, y: mnist.test.labels}) *# 基于测试集对准确率进行测试*

print("in iteration "**+**str(step) **+**"the accuracy is : "**+**str(corr)) *# 打印准确率*

*# 这里看似有问题，其实没问题，因为图没变，DAG对输入的batch依次执行梯度下降法，*

*# 并执行epoch\_n个周期，权重会更新epoch\_n \* batch\_n次*

​

Extracting MNIST\_data\train-images-idx3-ubyte.gz

Extracting MNIST\_data\train-labels-idx1-ubyte.gz

Extracting MNIST\_data\t10k-images-idx3-ubyte.gz

Extracting MNIST\_data\t10k-labels-idx1-ubyte.gz

in iteration 0the accuracy is : 0.1661

in iteration 1the accuracy is : 0.1918

## 交叉熵损失函数

由于分类并不适合二次损失函数，因此将多分类逻辑回归进行手写数字识别案例中的loss损失函数改成交叉熵损失函数，此时会更好。

loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(y - y\_head))

## dropout

dropout在训练时通过一定的随机丢弃部分神经元起到正则化的效果。

*# 通过随机化减少某些隐含层的神经元，达到正则化的目的*

​

*# 数据集介绍*

*# MNIST数据集，100k的训练数据，10k的预测数据，数据由tensorflow中的examples.tutorials.mnist读取*

*# 数据集介绍：：YannLeCun's website*

*# 由28\*28的像素组成输入特征，输出特征为0-9的数字*

​

*# 可调节参数：*

*# --------*

*# batch\_size, initial\_weight,交叉熵损失函数,learning\_rate,epoch\_n*

*# --------*

​

**import**tensorflow**as**tf

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

​

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

​

*# mini\_batch的大小*

batch\_size**=**100

batch\_n**=**mnist.train.num\_examples**//**batch\_size

​

*# # 定义两个placeholder用来feed数据，分别代表x和y --784列和10列(one-hot)*

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

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

​

*# # ----*

*# # 构建多分类回归*

*# # 定义weight和bias，初始化分别为截断正态随机和0.0*

​

*# 第一层*

num\_L1 **=**2000

keep\_prob1 **=**0.8

weight\_L1 **=**tf.Variable(tf.truncated\_normal([784, num\_L1], stddev**=**0.1))

bias\_L1 **=**tf.Variable(tf.zeros([num\_L1]) **+**0.1)

a\_L1 **=**tf.matmul(x, weight\_L1) **+** bias\_L1

z\_L1 **=**tf.nn.tanh(a\_L1)

z\_dropout\_L1 **=**tf.nn.dropout(z\_L1, keep\_prob1)

​

*# 第二层*

num\_L2 **=**2000

keep\_prob2 **=**0.8

weight\_L2 **=**tf.Variable(tf.truncated\_normal([num\_L1, num\_L2], stddev**=**0.1))

bias\_L2 **=**tf.Variable(tf.zeros([num\_L2]) **+**0.1)

a\_L2 **=**tf.matmul(z\_dropout\_L1, weight\_L2) **+** bias\_L2

z\_L2 **=**tf.nn.tanh(a\_L2)

z\_dropout\_L2 **=**tf.nn.dropout(z\_L2, keep\_prob2)

​

*# 第三层*

num\_L3 **=**1000

keep\_prob3 **=**0.8

weight\_L3 **=**tf.Variable(tf.truncated\_normal([num\_L2, num\_L3], stddev**=**0.1))

bias\_L3 **=**tf.Variable(tf.zeros([num\_L3]) **+**0.1)

a\_L3 **=**tf.matmul(z\_dropout\_L2, weight\_L3) **+** bias\_L3

z\_L3 **=**tf.nn.tanh(a\_L3)

z\_dropout\_L3 **=**tf.nn.dropout(z\_L3, keep\_prob3)

​

*# 输出层*

num\_y**=**10

keep\_prob3 **=**0.8

weight\_L4 **=**tf.Variable(tf.truncated\_normal([num\_L3, num\_y], stddev**=**0.1))

bias\_L4 **=**tf.Variable(tf.zeros([num\_y]) **+**0.1)

a **=**tf.matmul(z\_dropout\_L3, weight\_L4) **+** bias\_L4

y\_head**=**tf.nn.softmax(a)

​

*# # 定义交叉熵损失函数并依据梯度下降法进行训练 -- 这样梯度下降的train就变成了x和y的函数*

learning\_rate**=**0.2

loss**=**tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(labels **=** y, logits**=**y\_head))

optimizer**=**tf.train.GradientDescentOptimizer(learning\_rate)

train**=**optimizer.minimize(loss)

​

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

correct\_prediction**=**tf.equal(tf.argmax(y, 1), tf.argmax(y\_head, 1)) *# tf.argmax找到x中最大的id, 1为axis*

correction **=**tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32)) *# tf.cast转换类型，将bool转为float，从而求得准确率*

​

*# 迭代500次，进行mini\_batch梯度下降*

epoch\_n**=**50

**with**tf.Session() **as** session:

session.run(init)

**for** step **in**range(epoch\_n):

**for** batch **in**range(batch\_n):

batch\_x, batch\_y**=**mnist.train.next\_batch(batch\_size)

session.run(train, feed\_dict**=** {x: batch\_x, y: batch\_y}) *# 此处是最小化*

test\_corr**=**session.run(correction, feed\_dict**=** {x: mnist.test.images, y: mnist.test.labels}) *# 基于测试集对准确率进行测试*

train\_corr**=**session.run(correction, feed\_dict**=** {x: mnist.train.images, y: mnist.train.labels}) *# 基于训练集集对准确率进行测试*

print("in iteration "**+**str(step) **+**"the test accuracy is : "**+**str(test\_corr) **+**

"the train accuracy is : "**+**str(train\_corr)) *# 打印准确率*

## ​基于梯度下降的最优化方法

# tf.train.GradientDescentOptimizer

# tf.train.AdadeltaOptimizer

# tf.train.AdagradOptimizer

# tf.train.AdagradDAOptimizer

# tf.train.MomentumOptimizer

# tf.train.AdamOptimizer

# tf.train.FtrlOptimizer

# tf.train.ProximalGradientDescentOptimizer

# tf.train.ProximalAdagradOptimizer

# tf.train.RMSPropOptimizer

## TensorBoard之graph入门和可视化案例

### 通过FileWriter写入图

*# 数据集介绍*

*# MNIST数据集，100k的训练数据，10k的预测数据，数据由tensorflow中的examples.tutorials.mnist读取*

*# 数据集介绍：：YannLeCun's website*

*# 由28\*28的像素组成输入特征，输出特征为0-9的数字*

​

*# 可调节参数：*

*# --------*

*# batch\_size, initial\_weight,二次损失函数,learning\_rate,epoch\_n*

*# --------*

​

**import**tensorflow**as**tf

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

​

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

​

*# mini\_batch的大小*

batch\_size**=**100

batch\_n**=**mnist.train.num\_examples**//**batch\_size

​

*# 创建一个命名空间*

**with**tf.name\_scope("input"):

*# # 定义两个placeholder用来feed数据，分别代表x和y --784列和10列(one-hot)*

x**=**tf.placeholder(tf.float32, [**None**, 784], name**=**"x\_input")

y**=**tf.placeholder(tf.float32, [**None**, 10], name**=**"y\_input")

​

*# # ----*

*# # 构建多分类回归*

*# # 定义weight和bias，初始化分别为正态随机和0.0*

initial\_weight**=**tf.random\_normal([784, 10])

weight**=**tf.Variable(initial\_weight)

bias**=**tf.Variable(tf.zeros([10]))

a**=**tf.matmul(x, weight) **+**bias

y\_head**=**tf.nn.softmax(a)

​

*# # 定义二次损失函数并依据梯度下降法进行训练 -- 这样梯度下降的train就变成了x和y的函数*

learning\_rate**=**0.1

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

optimizer**=**tf.train.GradientDescentOptimizer(learning\_rate)

train**=**optimizer.minimize(loss)

​

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

​

correct\_prediction**=**tf.equal(tf.argmax(y, 1), tf.argmax(y\_head, 1)) *# tf.argmax找到x中等于1的最大的id*

correction**=**tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32)) *# tf.cast转换类型，将bool转为float，从而求得准确率*

​

*# 迭代500次，进行mini\_batch梯度下降*

epoch\_n**=**1

**with**tf.Session() **as**session:

session.run(init)

writer**=**tf.summary.FileWriter(logdir**=**"d://logs/", graph**=**session.graph) *# 最好不要放到带中文字符的路径*

**for**step**in**range(epoch\_n):

**for**batch**in**range(batch\_n):

batch\_x, batch\_y**=**mnist.train.next\_batch(batch\_size)

session.run(train, feed\_dict**=** {x: batch\_x, y: batch\_y}) *# 此处是最小化*

corr**=**session.run(correction, feed\_dict**=** {x: mnist.test.images, y: mnist.test.labels}) *# 基于测试集对准确率进行测试*

print("in iteration "**+**str(step) **+**"the accuracy is : "**+**str(corr)) *# 打印准确率*

*# 这里看似有问题，其实没问题，因为图没变，DAG对输入的batch依次执行梯度下降法，*

*# 并执行epoch\_n个周期，权重会更新epoch\_n \* batch\_n次*

### ​调用tensorboard生成服务

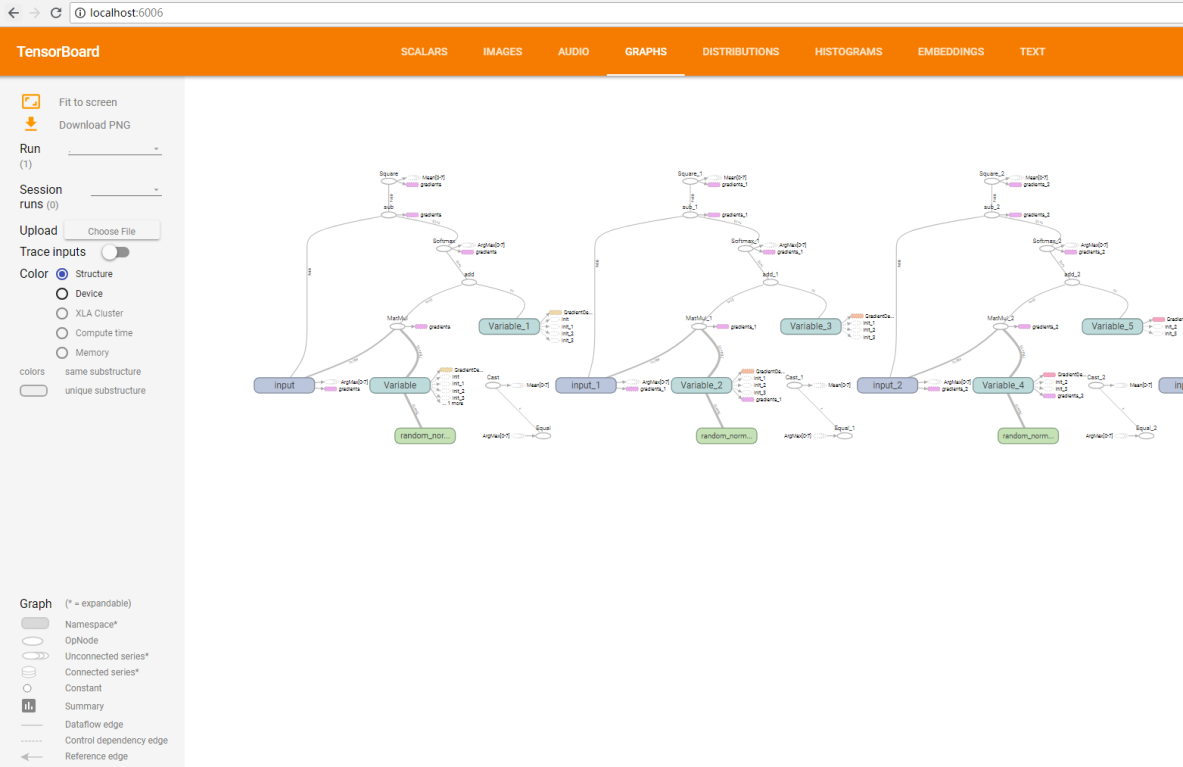
python–m tensorflow.tensorboard–logdir = d://logs

（如果端口被占用，上面命令变为python –m tensorflow.tensorboard–logdir = d://logs–port = 6007）

打开logcalhost：6006

### Tensorboard界面的效果

Tensorboard是可视化的图形界面，可以展示图中dag的依赖关系



### 注意事项

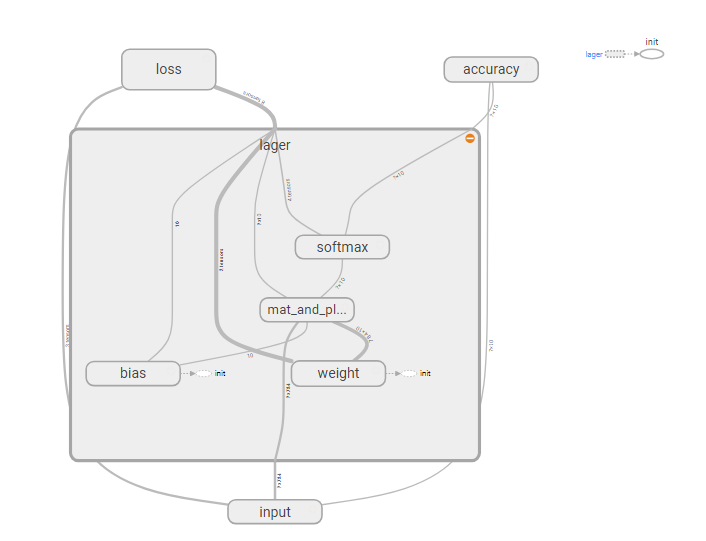
1. logdir路径不要有中文
2. tensorboard要和logdir在同一个盘符内运行
3. 最好在d盘的根目录里运行tensorboard，实验目录的/和\是一样
4. 会报告warning，没事：WARNING:tensorflow:path ../external/data/plugin/text/runs not found, sending 404
5. 用chrome浏览器最好
6. 有时候点击，可能看不到了，缩小，有可能跑到其他地方去了
7. 内存中图是append形式的（jupyter是这样）加入到内存的，如果同一个脚本执行多次，会发现图呈线性趋势越来越复杂。如果想避免多次执行图变得越来越复杂的情况，清空内存再执行，jupyter是restart&clear output或restart&run all，然后重新载入6006页面：

图6-1

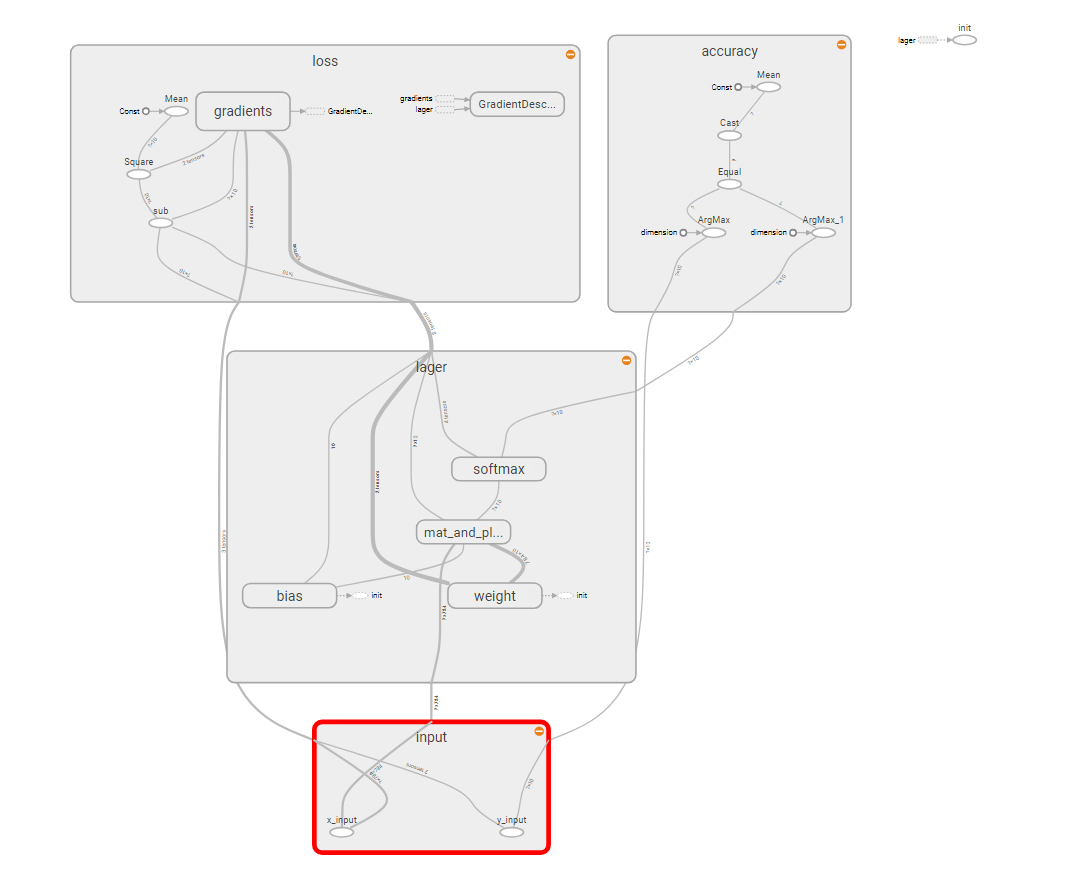


图6-2展开图

发现清晰了很多。

1. name\_scope的作用

可视化

获取参数（尤其是有同名参数时）

1. logs/下面继续新建两个子文件夹train和test，此时会生成两条曲线

tensorboard的logdir此时仍然是…./logs/

## 卷积神经网络

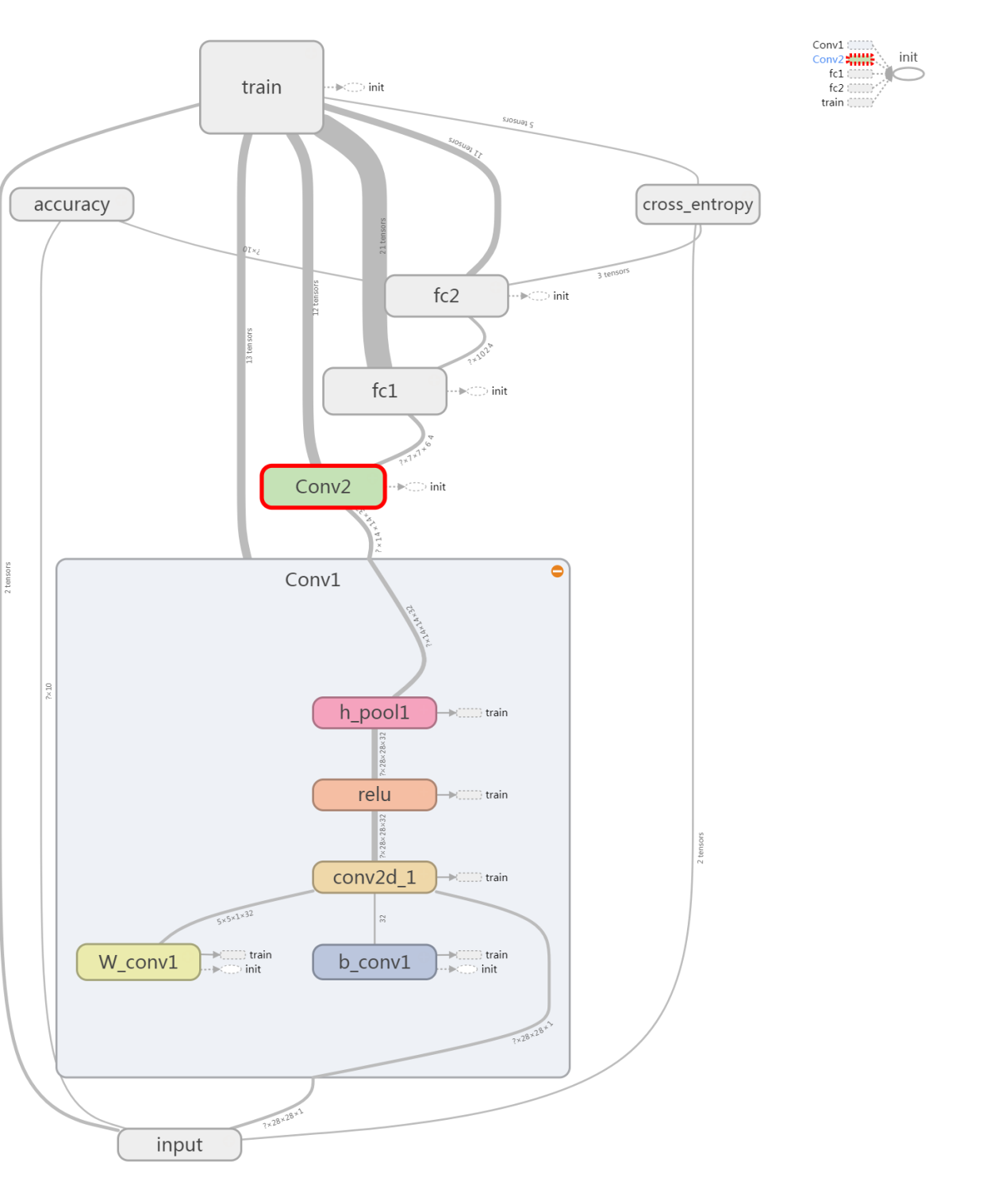
卷积和池化有两种形式，SAME\_PAD和VALID\_PAD

图形数据的信道

具体程序如下：

# coding: utf-8  
  
# In[1]:  
  
**import** tensorflow **as** tf  
**from** tensorflow.examples.tutorials.mnist **import** input\_data  
  
# In[2]:  
  
mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=**True**)  
  
# 每个批次的大小  
batch\_size = 100  
# 计算一共有多少个批次  
n\_batch = mnist.train.num\_examples // batch\_size  
  
  
# 参数概要  
**def variable\_summaries**(var):  
**with** tf.name\_scope('summaries'):  
 mean = tf.reduce\_mean(var)  
 tf.summary.scalar('mean', mean) # 平均值  
**with** tf.name\_scope('stddev'):  
 stddev = tf.sqrt(tf.reduce\_mean(tf.square(var - mean)))  
 tf.summary.scalar('stddev', stddev) # 标准差  
tf.summary.scalar('max', tf.reduce\_max(var)) # 最大值  
tf.summary.scalar('min', tf.reduce\_min(var)) # 最小值  
tf.summary.histogram('histogram', var) # 直方图  
  
  
# 初始化权值  
**def weight\_variable**(shape, name):  
 initial = tf.truncated\_normal(shape, stddev=0.1) # 生成一个截断的正态分布  
**return** tf.Variable(initial, name=name)  
  
  
# 初始化偏置  
**def bias\_variable**(shape, name):  
 initial = tf.constant(0.1, shape=shape)  
**return** tf.Variable(initial, name=name)  
  
  
# 卷积层  
**def conv2d**(x, W):  
# x input tensor of shape `[batch, in\_height, in\_width, in\_channels]`  
 # W filter / kernel tensor of shape [filter\_height, filter\_width, in\_channels, out\_channels]  
 # `strides[0] = strides[3] = 1`. strides[1]代表x方向的步长，strides[2]代表y方向的步长  
# padding: A `string` from: `"SAME", "VALID"`  
**return** tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')  
  
  
# 池化层  
**def max\_pool\_2x2**(x):  
# ksize [1,x,y,1]  
**return** tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')  
  
# logs 存储的根目录  
DIR = "d://tensorboardLogDir/sixWeek/"  
  
# 命名空间  
**with** tf.name\_scope('input'):  
# 定义两个placeholder  
x = tf.placeholder(tf.float32, [**None**, 784], name='x-input')  
 y = tf.placeholder(tf.float32, [**None**, 10], name='y-input')  
**with** tf.name\_scope('x\_image'):  
# 改变x的格式转为4D的向量[batch, in\_height, in\_width, in\_channels]`  
x\_image = tf.reshape(x, [-1, 28, 28, 1], name='x\_image')  
  
**with** tf.name\_scope('Conv1'):  
# 初始化第一个卷积层的权值和偏置  
**with** tf.name\_scope('W\_conv1'):  
 W\_conv1 = weight\_variable([5, 5, 1, 32], name='W\_conv1') # 5\*5的采样窗口，32个卷积核从1个平面抽取特征  
**with** tf.name\_scope('b\_conv1'):  
 b\_conv1 = bias\_variable([32], name='b\_conv1') # 每一个卷积核一个偏置值  
  
# 把x\_image和权值向量进行卷积，再加上偏置值，然后应用于relu激活函数  
**with** tf.name\_scope('conv2d\_1'):  
 conv2d\_1 = conv2d(x\_image, W\_conv1) + b\_conv1  
**with** tf.name\_scope('relu'):  
 h\_conv1 = tf.nn.relu(conv2d\_1)  
**with** tf.name\_scope('h\_pool1'):  
 h\_pool1 = max\_pool\_2x2(h\_conv1) # 进行max-pooling  
  
**with** tf.name\_scope('Conv2'):  
# 初始化第二个卷积层的权值和偏置  
**with** tf.name\_scope('W\_conv2'):  
 W\_conv2 = weight\_variable([5, 5, 32, 64], name='W\_conv2') # 5\*5的采样窗口，64个卷积核从32个平面抽取特征  
**with** tf.name\_scope('b\_conv2'):  
 b\_conv2 = bias\_variable([64], name='b\_conv2') # 每一个卷积核一个偏置值  
  
# 把h\_pool1和权值向量进行卷积，再加上偏置值，然后应用于relu激活函数  
**with** tf.name\_scope('conv2d\_2'):  
 conv2d\_2 = conv2d(h\_pool1, W\_conv2) + b\_conv2  
**with** tf.name\_scope('relu'):  
 h\_conv2 = tf.nn.relu(conv2d\_2)  
**with** tf.name\_scope('h\_pool2'):  
 h\_pool2 = max\_pool\_2x2(h\_conv2) # 进行max-pooling  
  
# 28\*28的图片第一次卷积后还是28\*28，第一次池化后变为14\*14  
# 第二次卷积后为14\*14，第二次池化后变为了7\*7  
# 进过上面操作后得到64张7\*7的平面  
  
**with** tf.name\_scope('fc1'):  
# 初始化第一个全连接层的权值  
**with** tf.name\_scope('W\_fc1'):  
 W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024], name='W\_fc1') # 上一场有7\*7\*64个神经元，全连接层有1024个神经元  
**with** tf.name\_scope('b\_fc1'):  
 b\_fc1 = bias\_variable([1024], name='b\_fc1') # 1024个节点  
  
# 把池化层2的输出扁平化为1维  
**with** tf.name\_scope('h\_pool2\_flat'):  
 h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7 \* 7 \* 64], name='h\_pool2\_flat')  
# 求第一个全连接层的输出  
**with** tf.name\_scope('wx\_plus\_b1'):  
 wx\_plus\_b1 = tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1  
**with** tf.name\_scope('relu'):  
 h\_fc1 = tf.nn.relu(wx\_plus\_b1)  
  
# keep\_prob用来表示神经元的输出概率  
**with** tf.name\_scope('keep\_prob'):  
 keep\_prob = tf.placeholder(tf.float32, name='keep\_prob')  
**with** tf.name\_scope('h\_fc1\_drop'):  
 h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob, name='h\_fc1\_drop')  
  
**with** tf.name\_scope('fc2'):  
# 初始化第二个全连接层  
**with** tf.name\_scope('W\_fc2'):  
 W\_fc2 = weight\_variable([1024, 10], name='W\_fc2')  
**with** tf.name\_scope('b\_fc2'):  
 b\_fc2 = bias\_variable([10], name='b\_fc2')  
**with** tf.name\_scope('wx\_plus\_b2'):  
 wx\_plus\_b2 = tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2  
**with** tf.name\_scope('softmax'):  
# 计算输出  
prediction = tf.nn.softmax(wx\_plus\_b2)  
  
# 交叉熵代价函数  
**with** tf.name\_scope('cross\_entropy'):  
 cross\_entropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y, logits=prediction),  
name='cross\_entropy')  
 tf.summary.scalar('cross\_entropy', cross\_entropy)  
  
# 使用AdamOptimizer进行优化  
**with** tf.name\_scope('train'):  
 train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)  
  
# 求准确率  
**with** tf.name\_scope('accuracy'):  
**with** tf.name\_scope('correct\_prediction'):  
# 结果存放在一个布尔列表中  
correct\_prediction = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1)) # argmax返回一维张量中最大的值所在的位置  
**with** tf.name\_scope('accuracy'):  
# 求准确率  
accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
 tf.summary.scalar('accuracy', accuracy)  
  
# 合并所有的summary  
merged = tf.summary.merge\_all()  
  
**with** tf.Session() **as** sess:  
 sess.run(tf.global\_variables\_initializer())  
 train\_writer = tf.summary.FileWriter(DIR + 'logs/train', sess.graph)  
 test\_writer = tf.summary.FileWriter(DIR + 'logs/test', sess.graph)  
**for** i **in** range(1001):  
# 训练模型  
batch\_xs, batch\_ys = mnist.train.next\_batch(batch\_size)  
 sess.run(train\_step, feed\_dict={x: batch\_xs, y: batch\_ys, keep\_prob: 0.5})  
# 记录训练集计算的参数  
summary = sess.run(merged, feed\_dict={x: batch\_xs, y: batch\_ys, keep\_prob: 1.0})  
 train\_writer.add\_summary(summary, i)  
# 记录测试集计算的参数  
batch\_xs, batch\_ys = mnist.test.next\_batch(batch\_size)  
 summary = sess.run(merged, feed\_dict={x: batch\_xs, y: batch\_ys, keep\_prob: 1.0})  
 test\_writer.add\_summary(summary, i)  
  
**if** i % 100 == 0:  
 test\_acc = sess.run(accuracy, feed\_dict={x: mnist.test.images, y: mnist.test.labels, keep\_prob: 1.0})  
 train\_acc = sess.run(accuracy, feed\_dict={x: mnist.train.images[:10000], y: mnist.train.labels[:10000],  
keep\_prob: 1.0})  
print("Iter " + str(i) + ", Testing Accuracy= " + str(test\_acc) + ", Training Accuracy= " + str(train\_acc))  
  
 sess.close()

程序在tensorboard的图如下：



## RNN神经网络

输入一个序列，x\_1, …, x\_t的训练过程不再是不相关的了，而是有一定的联结关系。

X\_1\_1, x\_1\_2, … x\_1\_t

X\_2\_1, x\_2\_2, … x\_2\_t

X\_n\_1, x\_n\_2, … x\_n\_t

常见的是LSTM

一个LSTM由输入控制、遗忘控制和输出控制组成，如果不加这些控制则信息不会随序列损失而损失，没有遗忘，如果加入非线性的激活函数，最终信号会衰减接近0

# coding: utf-8  
  
**import** tensorflow **as** tf  
**from** tensorflow.examples.tutorials.mnist **import** input\_data  
  
# 载入数据  
mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=**True**)  
  
# 认为某个图片的一行为一条词汇  
n\_puts = 28  
  
# 一个图片总共28个行  
max\_time = 28  
  
# lstm网络中隐藏层单元(broke)  
lstm\_size = 100  
  
# 分类数  
n\_classes = 10  
  
# 每个批次的大小  
batch\_size = 50  
# 计算一共有多少个批次  
n\_batch = mnist.train.num\_examples // batch\_size  
  
# 定义x和y的输入和输出  
**with** tf.name\_scope('input'):  
# 定义两个placeholder  
x = tf.placeholder(tf.float32, [**None**, 784], name='x-input')  
 y = tf.placeholder(tf.float32, [**None**, 10], name='y-input')  
  
# 定义权值  
**with** tf.name\_scope('layer1'):  
**with** tf.name\_scope('W1'):  
 weight = tf.Variable(tf.truncated\_normal([lstm\_size, n\_classes], stddev=0.1))  
**with** tf.name\_scope('b'):  
 bias = tf.Variable(tf.constant(0.1, shape=[n\_classes]))  
  
  
**def rnn**(inputs, weights, biases):  
# x为feed的张量，shape为batch\_size\*728, 在进入rnn时需要先进行下转换  
# 转换为三维的数据，其中-1代表未定，max\_time为行数，n\_puts为列数，即每行的数据数  
**with** tf.name\_scope("reshape"):  
 reshape\_inputs = tf.reshape(inputs, [-1, max\_time, n\_puts])  
# 定义一个基本的LSTM单元  
**with** tf.name\_scope("lstm\_cell"):  
 lstm\_cell = tf.contrib.rnn.core\_rnn\_cell.BasicLSTMCell(lstm\_size)  
 \_, final\_state = tf.nn.dynamic\_rnn(lstm\_cell, reshape\_inputs, dtype=tf.float32)  
**with** tf.name\_scope("output"):  
 results = tf.nn.softmax(tf.matmul(final\_state[1], weights) + biases)  
**return** results  
  
  
**with** tf.name\_scope("prediction"):  
 prediction = rnn(x, weight, bias)  
  
**with** tf.name\_scope("loss"):  
 loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=prediction, labels=y))  
  
**with** tf.name\_scope("train"):  
 train\_step = tf.train.AdamOptimizer(0.001).minimize(loss)  
  
**with** tf.name\_scope("performance"):  
# 结果存放在一个布尔列表中  
**with** tf.name\_scope("train\_accuracy"):  
 correct\_prediction = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1)) # argmax返回一维张量中最大的值所在的位置  
**with** tf.name\_scope('accuracy'):  
# 求准确率  
accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
 tf.summary.scalar('accuracy', accuracy)  
  
# 合并所有的summary  
merged = tf.summary.merge\_all()  
  
DIR = "D://tensorboardLogDir/sevenWeek/"  
**with** tf.Session() **as** sess:  
 sess.run(tf.global\_variables\_initializer())  
 train\_writer = tf.summary.FileWriter(DIR + 'logs/train', sess.graph)  
 test\_writer = tf.summary.FileWriter(DIR + 'logs/test', sess.graph)  
**for** i **in** range(1000):  
# 训练模型  
batch\_xs, batch\_ys = mnist.train.next\_batch(batch\_size)  
 sess.run(train\_step, feed\_dict={x: batch\_xs, y: batch\_ys})  
# 记录训练集计算的参数  
summary = sess.run(merged, feed\_dict={x: batch\_xs, y: batch\_ys})  
 train\_writer.add\_summary(summary, i)  
# 记录测试集计算的参数  
batch\_xs, batch\_ys = mnist.test.next\_batch(batch\_size)  
 summary = sess.run(merged, feed\_dict={x: batch\_xs, y: batch\_ys})  
 test\_writer.add\_summary(summary, i)  
  
**if** i % 10 == 0:  
 test\_acc = sess.run(accuracy, feed\_dict={x: mnist.test.images, y: mnist.test.labels})  
 train\_acc = sess.run(accuracy, feed\_dict={x: mnist.train.images[:10000], y: mnist.train.labels[:10000]})  
print("Iter " + str(i) + ", Testing Accuracy= " + str(test\_acc) + ", Training Accuracy= " + str(train\_acc))  
  
 sess.close()

## gooleNet --inception和图像识别

## GPU进行tensorflow运算

1. 电脑有GPU (英伟达）

Cuda支持的显卡系列：

<https://blog.csdn.net/iteye_3607/article/details/81577023>

独显和集显的区别：独显有独立的内存，不需要用cpu的内存。

2）安装cuda（相当于GPU的驱动）

3）安装cudnn（加速cuda的）