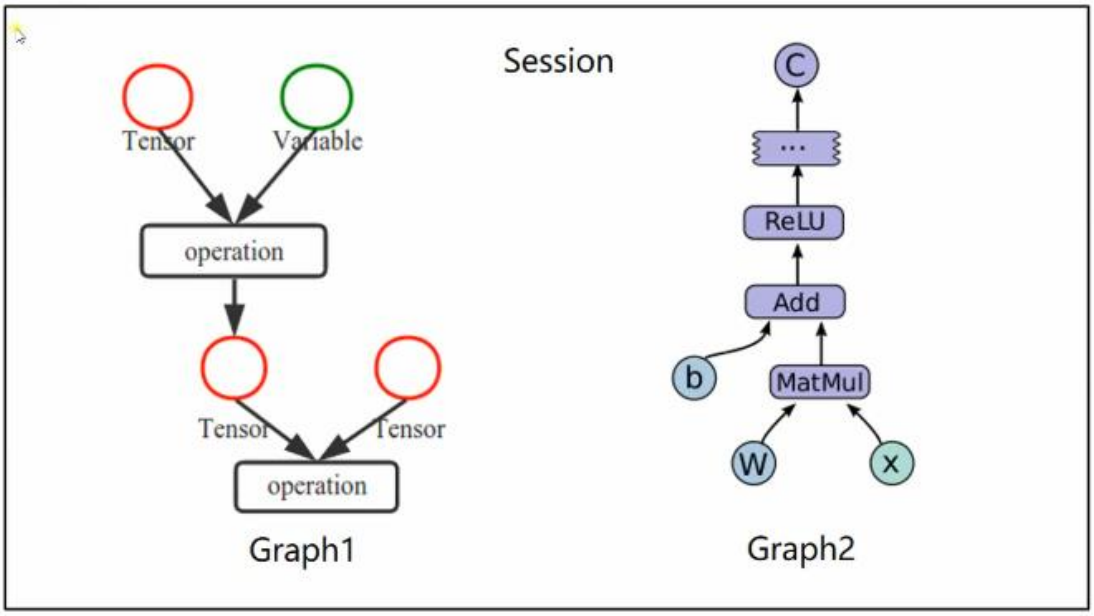
第二周　Tensorflow 的基本概念

1、基本概念

- 使用图（graphs）来表示计算任务
- 在被称之为会话（Session）的上下文（context）中执行图
- 使用 tensor 表示数据
- 通过变量 Variable 维护状态
- 使用 feed 和 fetch 可以为任意的操作赋值或者从其中获取数据

Tensorflow 是一个编程系统，使用图（graphs）来表示计算任务，图（graphs）中的节点称之为 operation，（add，mat 之类），一个 op 获得 0 个或者多个 tensor，执行计算，产生 0 个或多个 tensor，tensor 看作是一个 n 维的数组或列表。图必须在会话里被启动。



2、基本代码（变量、常量、会话、op 节点、图）

```
# 声明变量
x = tf.Variable([1,2])
# 声明常量
a = tf.constant([3,3])
# 增加一个减法 op
sub = tf.subtract(x,a)
# 增加一个加法 op
```

```
add = tf.add(a,sub)



# 全局变量初始化
init = tf.global_variables_initializer()

# 在图内运行代码
with tf.Session() as sess:
    sess.run(init)
    print(x.value)
    print(sess.run(sub))
print(sess.run(add))

# 赋值 op
#update = tf.assign(state,new_value)
```

3、Fetch and Feed

Fetch 可以在一个会话中同时执行多个 op,也就是说可以同事进行多个节点的计算

Feed 可以每次为网络传入不同的数据

代码示例:

```
# Fetch
input1 = tf.constant(3.0)
input2 = tf.constant(2.0)
input3 = tf.constant(5.0)

add = tf.add(input2,input3)
mul = tf.multiply(input1,add)

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

输出了两个数字

```
# Feed
#  创建占位符
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.multiply(input1, input2) #元素级别相乘

with tf.Session() as sess:
    # feed 数据以字典形式传入
    print(sess.run(output, feed_dict={input1:[2.0],input2:[6.0]}))
```

有了占位符和 feed，网络可以每次传入不同的数据

4、简单示例

```
import tensorflow as tf
import numpy as np

#  生成随机点
x_data = np.random.rand(100)
y_data = x_data*0.1 + 0.2

#  构建线性模型
b = tf.Variable(0.)
k = tf.Variable(0.)
y = k*x_data + b

#  定义损失函数
loss = tf.reduce_mean(tf.square(y_data-y))
#  定义一个梯度下降法进行训练的优化器,学习率是 0.2
optimizer = tf.train.GradientDescentOptimizer(0.2)
#  最小化代价函数
train = optimizer.minimize(loss)

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

```
with tf.Session() as sess:
    sess.run(init)
    for step in range(201):
        sess.run(train)
        if step%20 == 0:
            print(step+1, sess.run([k,b]))
```

第三周 Tensorflow 非线性回归以及分类的简单使用，softmax 介绍

1、非线性回归简单示例

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# 生成 200 个随机点，并改变其形状为 200*1
x_data = np.linspace(-0.5, 0.5, 200)[:,np.newaxis]
noise = np.random.normal(0,0.02,x_data.shape)
y_data = np.square(x_data) + noise

#查看一下数据形状
print(x_data.shape)
type(x_data)
print(noise.shape)

# 定义两个 placeholder
x = tf.placeholder(tf.float32, [None,1])
y = tf.placeholder(tf.float32, [None,1])

# 定义中间层
Weights_L1 = tf.Variable(tf.random_normal([1,10]))
bias_L1 = tf.Variable(tf.zeros([1,10]))
```

```
Wx_plus_b_L1 = tf.matmul(x, Weights_L1) + bias_L1
# 激活函数
L1 = tf.nn.tanh(Wx_plus_b_L1)

# 定义输出层
Weights_L2 = tf.Variable(tf.random_normal([10,1]))
bias_L2 = tf.Variable(tf.zeros([1,1]))
Wx_plus_b_L2 = tf.matmul(L1,Weights_L2) + bias_L2
prediction = tf.nn.tanh(Wx_plus_b_L2)

# 二次代价函数（损失函数）
loss = tf.reduce_mean(tf.square(y-prediction))
# 梯度下降法
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

with tf.Session() as sess:
    # 变量的初始化
    sess.run(tf.global_variables_initializer())
    for _ in range(2000):
        sess.run(train_step, feed_dict={x:x_data, y:y_data})

    # 获得预测值
    prediction_value = sess.run(prediction,feed_dict={x:x_data})

    plt.figure()
    plt.scatter(x_data, y_data)
    plt.plot(x_data, prediction_value,'r-', lw=5)
    plt.show()
```

第四周 交叉熵(cross-entropy)，过拟合，dropout 以及 Tensorflow 中各种优化器的介绍

1、代价函数的选择（关于交叉熵代价函数和对数释然代价函数

如果输出神经元是线性的，二次代价函数是合适的选择，

如果输出神经元是 S 型函数，适用交叉熵代价函数。（这里指的是输出层被 sigmoid 函数激活的情况）

如果输出层被 sigmoid 激活，可采用交叉熵代价函数，如果使用 softmax 作为网络最后一层，此时常用对数释然代价函数。

对数似然代价函数与 softmax 的组合，跟交叉熵与 sigmoid 函数的组合非常相似。在二分类问题中，对数释然代价函数简化为交叉熵代价函数的形势。

在 Tensorflow 中

Tf.sigmoid_cross_entropy_with_logits()表示跟 sigmoid 搭配使用的交叉熵

Tf.softmax_corss_entropy_with_logits()表示跟 softmax 搭配使用的交叉熵。

2、过拟合

解决过拟合的操作

增加数据集、

正则化方法、减小 loss，从而减小学习率

Dropout、随机丢弃某些神经节点，使网络不能过度依赖某些节点

3、Optimizer

各种优化器对比

标准梯度下降法:

计算所有样本汇总误差，根据总误差来更新权值;

缺点：太慢;

随机梯度下降法:

随机抽取一个样本来计算误差，然后更新权值;

缺点：对噪声过于敏感

批量梯度下降法:

算是一种折中方案, 从总样本中选取一个批次, 计算该批次数据误差, 来更新权值;

SGD Momentum

W：要训练的参数

$J(W)$：代价函数

$\nabla_W J(W)$：代价函数的梯度

η：学习率

SGD：

$$W = W - \eta \cdot \nabla_W J(W; x^{(i)}; y^{(i)})$$

Momentum：

γ：动力，通常设置为0.9

$$v_t = \gamma v_{t-1} + \eta \nabla_W J(W)$$
$$W = W - v_t$$

当前权值的改变会受到上一次权值改变的影响，类似于小球向下滚动的时候带上了惯性。这样可以加快小球的向下的速度。

NAG

NAG（Nesterov accelerated gradient）：

$$v_t = \gamma v_{t-1} + \eta \nabla_W J(W - \gamma v_{t-1})$$
$$W = W - v_t$$

NAG在TF中跟Momentum合并在同一个函数tf.train.MomentumOptimizer中，可以通过参数配置启用。在Momentun中小球会盲目地跟从下坡的梯度，容易发生错误，所以我们需要一个更聪明的小球，这个小球提前知道它要去哪里，它还要知道走到坡底的时候速度慢下来而不是又冲上另一个坡。γvt−1会用来修改W的值，计算W−γvt−1可以表示小球下一个位置大概在哪里。从而我们可以提前计算下一个位置的梯度，然后使用到当前位置。

Adagrad

Adagrad：

i：代表第i个分类

t：代表出现次数

ϵ：的作用是避免分母为0，取值一般为1e-8

η：取值一般为0.01

$$g_{t,i} = \nabla_W J(W_i)$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\sum_{t'=1}^{t}(g_{t',i})^2+\epsilon}} \odot g_t$$

它是基于SGD的一种算法，它的核心思想是对比较常见的数据给予它比较小的学习率去调整参数，对于比较罕见的数据给予它比较大的学习率去调整参数。它很适合应用于数据稀疏的数据集（比如一个图片数据集，有10000张狗的照片，10000张猫的照片，只有100张大象的照片）。

Adagrad主要的优势在于不需要人为的调节学习率，它可以自动调节。它的缺点在于，随着迭代次数的增多，学习率也会越来越低，最终会趋向于0。

RMSprop

RMSprop：

RMS（Root Mean Square）是均方根的缩写。

γ：动力，通常设置为0.9

η：取值一般为0.001

$E[g^2]_t$：表示前t次的梯度平方的平均值

$$g_t = \nabla_W J(W)$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g^2_t$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$

RMSprop借鉴了一些Adagrad的思想，不过这里RMSprop只用到了前t次梯度平方的平均值加上当前梯度的平方的和的开平方作为学习率的分母。这样RMSprop不会出现学习率越来越低的问题，而且也能自己调节学习率，并且可以有一个比较好的效果。

Adadelta

Adadelta：

$$g_t = \nabla_W J(W)$$

$$\Delta W_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$

$$\Delta W_t = -\frac{\eta}{RMS[g]_t} \odot g_t$$

$$W_{t+1} = W_t - \frac{RMS[\Delta W]_{t-1}}{RMS[g]_t}$$

使用Adadelta我们甚至不需要设置一个默认学习率，在Adadelta不需要使用学习率也可以达到一个非常好的效果。

Adam

Adam：

β_1：一般取值0.9

β_2：一般取值0.999

ε：避免分母为0，一般取值10^{-8}

$$m_t=\beta_1 m_{t-1}+(1-\beta_1)g_t$$

$$v_t=\beta_2 v_{t-1}+(1-\beta_2)g^2{}_t$$

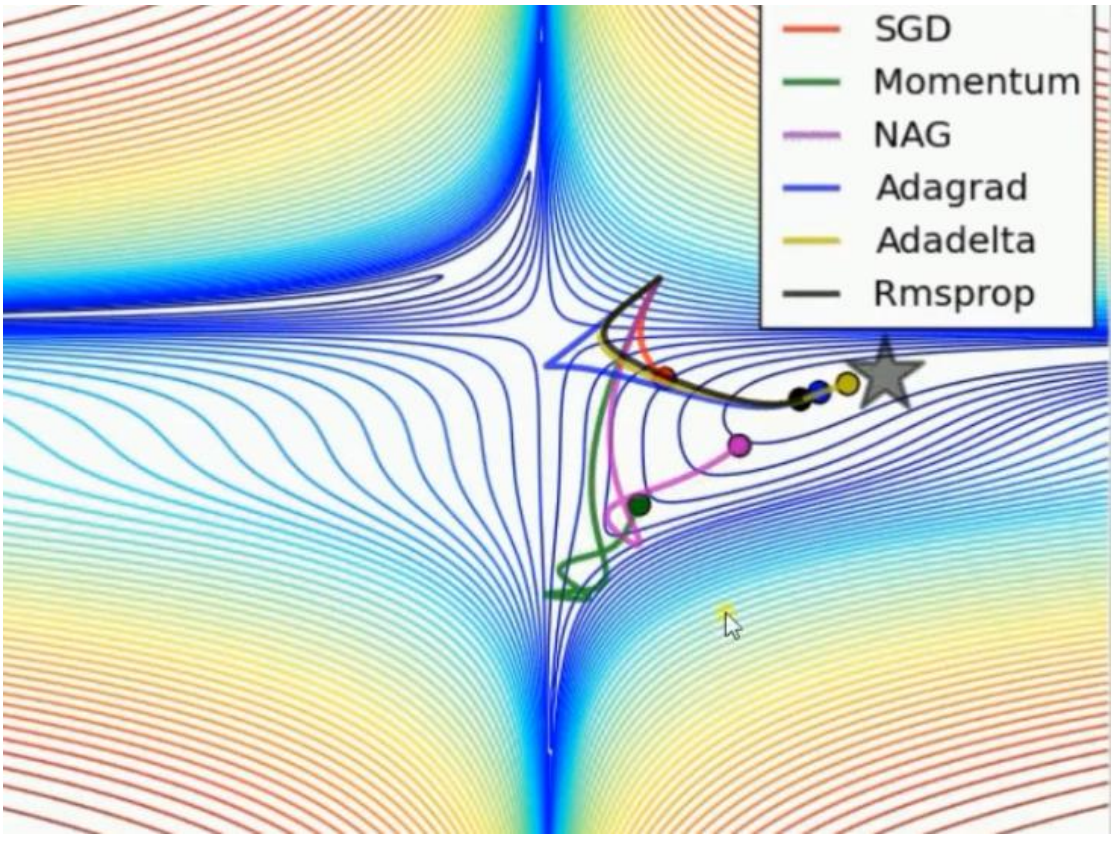
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{V}_t = \frac{v_t}{1-\beta_2^t}$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon}\hat{m}_t$$

就像Adadelta和RMSprop一样Adam会存储之前衰减的平方梯度，同时它也会保存之前衰减的梯度。经过一些处理之后再使用类似Adadelta和RMSprop的方式更新参数。

效果比较 1



效果图：

排名：

Adadelta

Adagrade

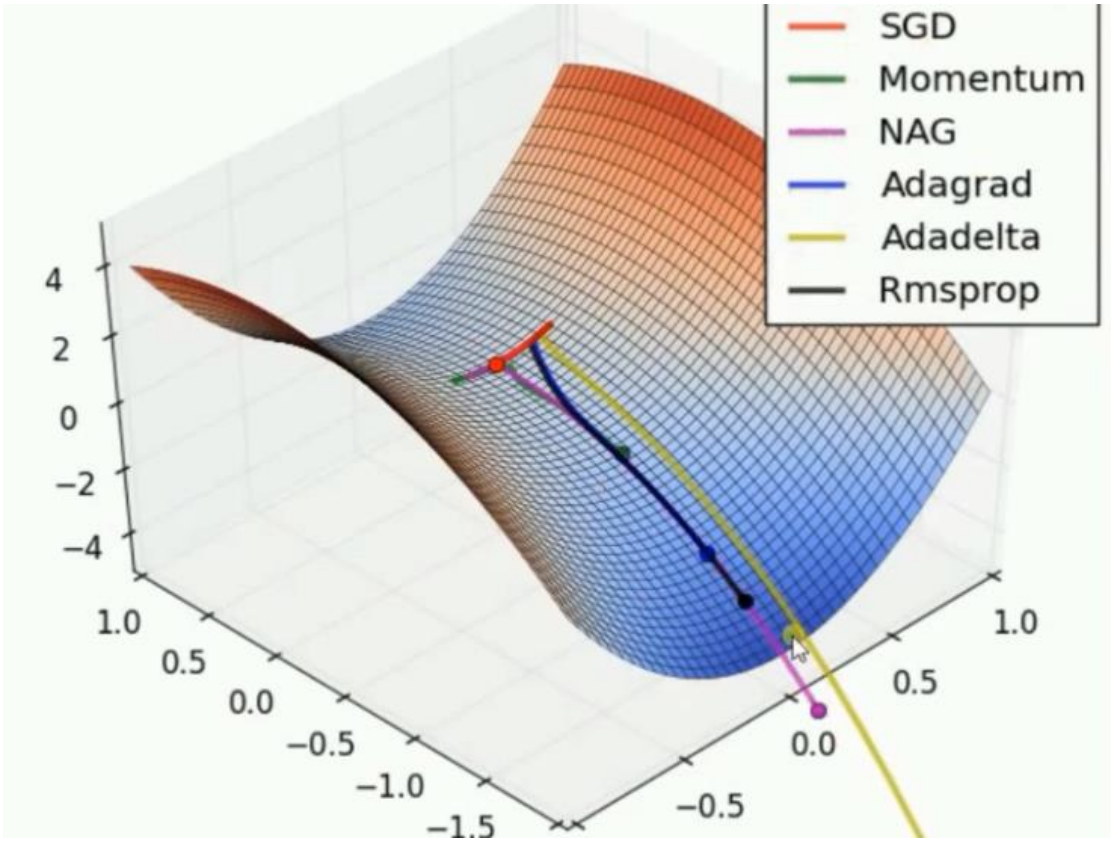
Rmsprop

NAG

Momentum

SGD

效果比较：鞍点问题



排名：

Adadelta 以很快的速度冲下来

NAG 在最初的迷茫之后快速走下鞍点，速度很快

Momentum 也在迷茫之后走下鞍点，但是没有 NAG

Rmsprop 没有迷茫，但是下降速度有点慢

Adagrad 也没有迷茫，但是速度更慢

SGD，直接在鞍点下不来了

效果比较：总结

优化器各有优缺点

别的优化器收敛速度会比较快，但是 SGD 最后的结果一般来说很好

以下网址有不错的总结，可以参考

https://blog.csdn.net/g11d111/article/details/76639460

第五周 使用 Tensorboard 进行结构可视化，以及网络运算过程可视化

1、打开 tensorboard

1 给网络输入值加上命名空间

```
# 命名空间
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None,784],name="X-input")
    y = tf.placeholder(tf.float32, [None, 10],name='y-input')
with tf.name_scope('layer'):
    with tf.name_scope('weights'):
        W = tf.Variable(tf.zeros([784,10]),name='W')
    with tf.name_scope('biases'):
        b = tf.Variable(tf.zeros([1, 10]),name='b')
    with tf.name_scope('xw_plus_b'):
        wx_plus_b = tf.matmul(x,W) + b
    with tf.name_scope('softmax'):
        prediction = tf.nn.softmax(wx_plus_b)
```

在运行网络时候加入日志

```
writer = tf.summary.FileWriter('logs/',sess.graph)
```

命令行：tensorboard –logdir = E:\pywork\DL\TensorFLow-Learning\logs

2、查看网络参数变化情况

主要用方法 tf.summary.scalar('name',value)来记录

在运行网络之前，使用以下方法来汇总要记录的变量

Merged = tf.summary.merge_all()

sess.run()里包含上面的变量并记录。看下面标准代码：

```
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)

# 批次的大小
batch_size = 128
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


# 命名空间
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None,784],name="X-input")
    y = tf.placeholder(tf.float32, [None, 10],name='y-input')

# 创建一个简单的神经网络
with tf.name_scope('layer'):
    with tf.name_scope('weights'):
        W = tf.Variable(tf.zeros([784,10]),name='W')
```

```
            variable_summaries(W)
        with tf.name_scope('biases'):
            b = tf.Variable(tf.zeros([1, 10]),name='b')
            variable_summaries(b)
        with tf.name_scope('xw_plus_b'):
            wx_plus_b = tf.matmul(x,W) + b
        with tf.name_scope('softmax'):
            prediction = tf.nn.softmax(wx_plus_b)

    # 代价函数
    with tf.name_scope('loss'):
        loss = tf.reduce_mean(tf.square(y-prediction))
        tf.summary.scalar('loss', loss)
    # 梯度下降法
    with tf.name_scope('train'):
        train_step = tf.train.GradientDescentOptimizer(0.2).minimize(loss)

    # 初始化变量
    init = tf.global_variables_initializer()

    # 得到一个布尔型列表，存放结果是否正确
    with tf.name_scope('accuracy'):
        with tf.name_scope('correct_prediction'):
            correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(prediction,1)) #argmax返回一维张量中最大值索引
        # 求准确率
        with tf.name_scope('accuracy'):
            accuracy = tf.reduce_mean(tf.cast(correct_prediction,tf.float32)) # 把布尔值转换为浮点型求平均数
            tf.summary.scalar('accuracy', accuracy)

    # 合并所有 summary
    merged = tf.summary.merge_all()

    with tf.Session() as sess:
```

```
sess.run(init)
writer = tf.summary.FileWriter('logs/',sess.graph)
for epoch in range(51):
    for batch in range(n_batch):
        # 获得批次数据
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # 运行网络并记录 log
        summary,_ = sess.run([merged, train_step], feed_dict={x:batch_xs, y:batch_ys})
    # 记录变量
    writer.add_summary(summary,epoch)
    acc = sess.run(accuracy, feed_dict={x:mnist.test.images,y:mnist.test.labels})
    print("Iter " + str(epoch) + " Testing Accuracy: " + str(acc))
```

3、补充资料

5-3 的代码里更新了两部分代码，一部分是查看网络数据的更优美的代码，另一部分是莫烦大神的关于可视化梯度下降的部分的代码。链接及代码都在 5-3 的 notebook 里。

4、发现一个重要问题，

计算 loss 的时候，如果使用 `tf.nn.softmax_cross_entropy_with_logit`

那么直接把函数值传进去就好，该函数会对 logits 先计算 softmax，再计算交叉熵。

笔者写的代码已经都更正过来。

关于函数验证，详见

https://blog.csdn.net/mao_xiao_feng/article/details/53382790

5、关于 tensorboard 可视化

能够利用 tensorboard 查看训练时参数变化情况，但是 tensorboard 可视化的部分代码运行不了会异常中止。

确定问题在于 metadata 上,猜测跟 tensorflow 版本相关，没有进行进一步验证

第六周 卷积神经网络 CNN

1、

经验之谈：样本数量最好是参数数量的 5-30 倍。数据量小而模型参数过的多容易出现

过拟合现象。

定义 weight、bias,

卷积、激活、池化、下一层

然后接 2 个全连接层，softmax，交叉熵、loss

第七周 SLTM

这部分先跳过，打算先看一下吴恩达的课程，学习一下理论。如果需要的话，再回来补这部分

第八周 保存和载入模型，使用 Google 的图像识别网络 inception-v3 进行图像识别

1、关键代码

Saver = tf.train.Saver()

保存模型： Saver.save(sess, “*.ckpt”)

载入模型: Saver.restore(sess, “*.ckpt”)

2、inception-v3

下面的代码包含了，下载、解压、使用 tensorflow 载入模型、保存 tensorborad 的 log 文件。

代码部分

```
inception_pretrain_model_url = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz'

inception_pretrain_model_dir = "inception_model"
if not os.path.exists(inception_pretrain_model_dir):
    os.makedirs(inception_pretrain_model_dir)

filename = inception_pretrain_model_url.split('/')[-1]
filepath = os.path.join(inception_pretrain_model_dir,filename)
```

```
    if not os.path.exists(filepath):
        print('download:',filename)
        r = requests.get(inception_pretrain_model_url, stream=True)
        with open(filepath, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024):
                if chunk:
                    f.write(chunk)
    print('finish: ',filename)

    #解压文件
    tarfile.open(filepath, 'r:gz').extractall(inception_pretrain_model_dir)

    log_dir = 'logs/inception_log'
    if not os.path.exists(log_dir):
        os.makedirs(log_dir)

    # classfy_image_graph_def.pb  为 google 训练好的模型
    inception_graph_def_fiel                                                    =
os.path.join(inception_pretrain_model_dir,'classify_image_graph_def.pb')
    with tf.Session() as sess:
        #创建一个图来存放 google 训练好的模型
        with tf.gfile.FastGFile(inception_graph_def_fiel, 'rb') as f:
            graph_def = tf.GraphDef()
#               graph_def.ParseFromSring(f.read())
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')

        # 保存图的结构
        writer = tf.summary.FileWriter(log_dir, sess.graph)
        writer.close()
```

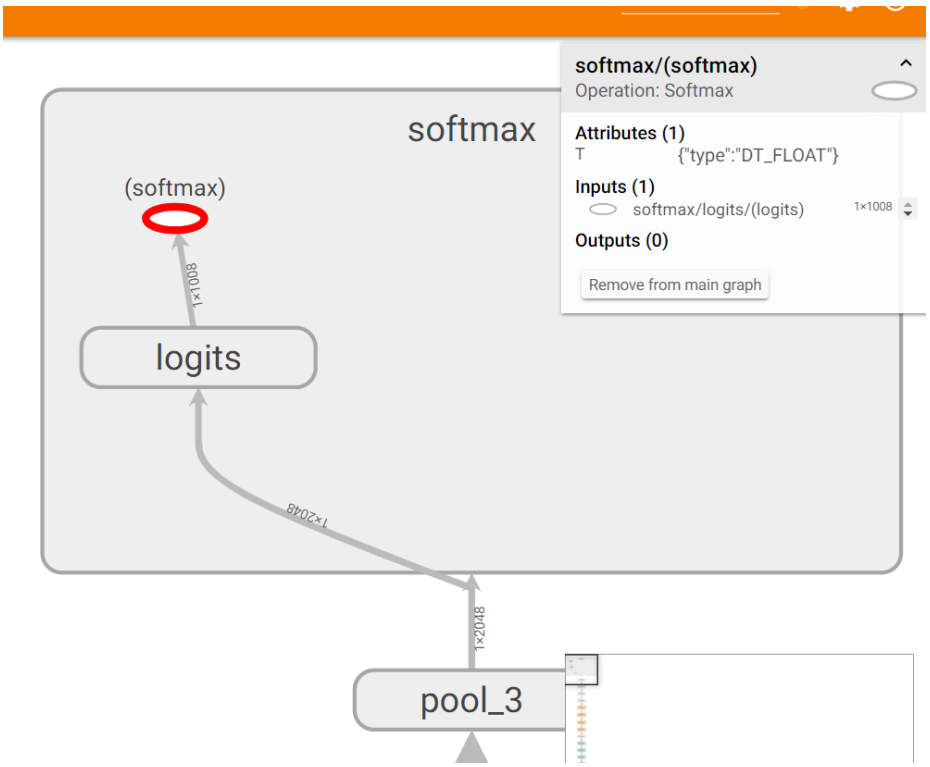
3、使用 inception-v3

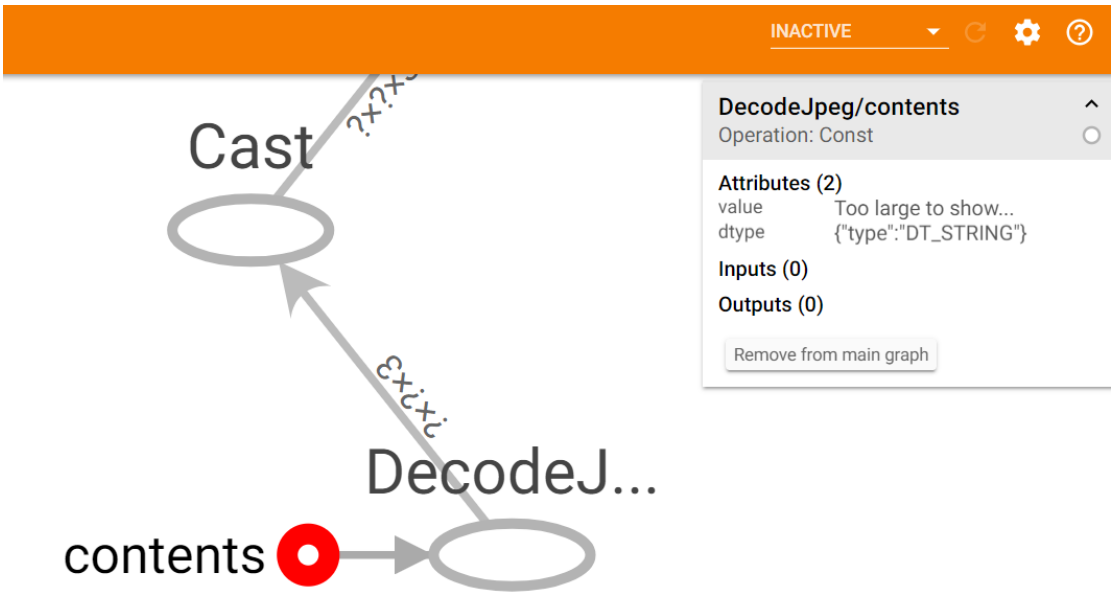
关键代码:

```
# 创建一个图来存放 google 训练好的模型
with tf.gfile.FastGFile('inception_model/classify_image_graph_def.pb','rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    tf.import_graph_def(graph_def, name='')

# 运行网络
with tf.Session() as sess:
    # 拿到 softmax 的 op
    # 'softmax:0'这个名字，可以在网络中找到这个节点，它的名字就'(softmax)',
    softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
    for root,dirs,files in os.walk('images/'):
        for file in files:
            image_data = tf.gfile.FastGFile(os.path.join(root,file),'rb').read()
            # 运行 softmax 节点，向其中 feed 值
            # 可以在网络中找到这个名字，DecodeJpeg/contents,
                    predictions
            =sess.run(softmax_tensor,{'DecodeJpeg/contents:0':image_data})
            predictions = np.squeeze(predictions)# 把结果转化为 1 维数据
```

关于取 op 及向网络 feed 值用到得名字，如下图：





据此可以发现，根据名字取网络中 op 时，如果其名字带括号，就用括号内的名字，如果不带括号，就用右上角介绍的名字。

而带个 0，是默认情况，如果网络中出现同名节点，这个编号会递增

第九周 tensorflow-gpu 安装，设计并训练自己的网络模型

1、安装

略

2、训练自己的网络

三个办法，

自己写自己从头构建网络，从头训练；

用一个现成的质量比较好的模型，固定前面参数，在后面添加几层，训练后面的参数

改造现成的质量比较好的模型，训练整个网络的模型（初始层的学习率比较低）；

9-2 这个部分的质量不是很实用，还是从网上找一些迁移学习的例子来学习比较实用，

略。

十、多任务学习及验证码识别

1、关于 tfrecord 文件

于 Tensorflow 读取数据，官网给出了三种方法：

供给数据(Feeding)： 在 TensorFlow 程序运行的每一步， 让 Python 代码来供给数据。

从文件读取数据： 在 TensorFlow 图的起始， 让一个输入管线从文件中读取数据。

预加载数据： 在 TensorFlow 图中定义常量或变量来保存所有数据(仅适用于数据量比较小的情况)。

对于数据量较小而言，可能一般选择直接将数据加载进内存，然后再分 batch 输入网络进行训练（tip:使用这种方法时，结合 yield 使用更为简洁，）。但是，如果数据量较大，这样的方法就不适用了，因为太耗内存，所以这时最好使用 tensorflow 提供的队列 queue，也就是第二种方法 从文件读取数据。使用 tensorflow 内定标准格式——TFRecords。

从宏观来讲，tfrecord 其实是一种数据存储形式。使用 tfrecord 时，实际上是先读取原生数据，然后转换成 tfrecord 格式，再存储在硬盘上。而使用时，再把数据从相应的 tfrecord 文件中解码读取出来。那么使用 tfrecord 和直接从硬盘读取原生数据相比到底有什么优势呢？其实，Tensorflow 有和 tfrecord 配套的一些函数，可以加快数据的处理。实际读取 tfrecord 数据时，先以相应的 tfrecord 文件为参数，创建一个输入队列，这个队列有一定的容量（视具体硬件限制，用户可以设置不同的值），在一部分数据出队列时，tfrecord 中的其他数据就可以通过预取进入队列，并且这个过程和网络的计算是独立进行的。也就是说，网络每一个 iteration 的训练不必等待数据队列准备好再开始，队列中的数据始终是充足的，而往队列中填充数据时，也可以使用多线程加速。

相关资料

https://blog.csdn.net/happyhorizion/article/details/77894055

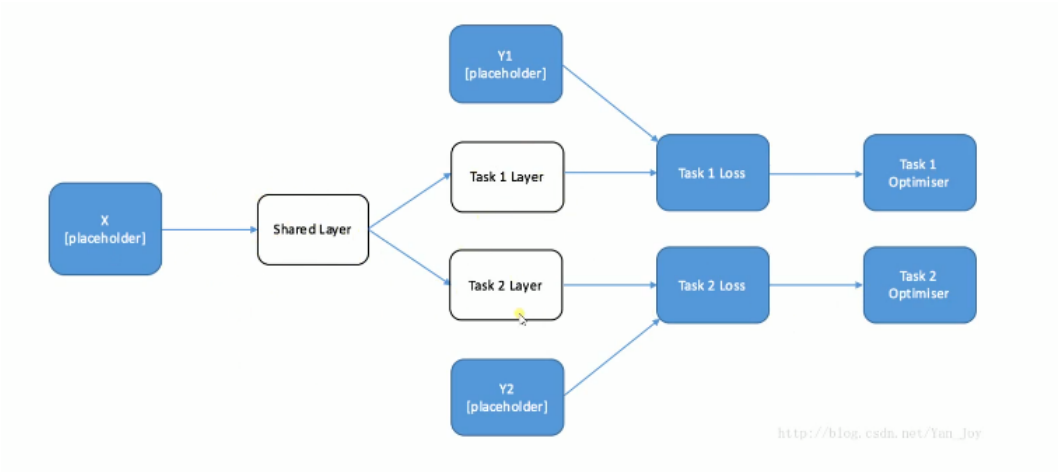
https://blog.csdn.net/u010358677/article/details/70544241

https://blog.csdn.net/Best_Coder/article/details/70146441

本目录下 tfrecord 文件夹下，意义 10.1 节都有相关实现代码

2、多任务学习

多个相似任务有多个数据集，交替训练



多个相似任务只有一个数据集，联合训练