内容目录

Tensorflow

一、Tensorflow 的简介

Tensorflow 的官方说明说是一个采用数据流图(data flow graph)，用于数值计算的开源软件库，图中由节点(node) 和边(edge)组成，节点是计算单元(op)，表示进行数学计算,一个 op 获得 0 或多个 Tensor，边则表示在节点间相互联系的多维数据数组，同时边也称之为张量(Tensor)。

Tensorflow 程序通常被分为两个阶段：构建阶段和执行阶段

构建阶段：通过创建 op 和所需的 Tensor 后就完成了 Tensorflow 图的构建，此时 op 的执行步骤被描述成一个图。

执行阶段：为了计算图中的节点，图必须在会话中启动执行，图中的 op 会分发至多个 GPU 或 CPU 上并行执行，每个 op 计算完成后会返回 Tensor，Python 中返回的 Tensor 是 numpy ndarray 对象。

Tensorflow 的主要特点包括

1、快速搭建神经网络或深度网络模型，

2、自动求微分

3、隐藏了在 CPU 和 GPU 上运行和加速细节，提高运行速度，协调并行问题

4、多语言支持，主要包括 python，C++，JAVA，Go 等语言

这段时间学习 tensorflow 的目的是用来构建自己设计的神经网络模型的，因此，官方说明中的节点其实指的就是神经网络中的神经元，神经元有抑制和激活的功能，对应于节点的数值计算操作(op)。而边指的就是神经元组成的层与层之间的权值矩阵和偏置矩阵(Tensor)。因此，在使用 tensorflow 时，首先应该完成神经网络模型设计，主要包括：

1. 神经元的层数
2. 每层神经元的个数
3. 各层神经元的激活函数
4. 确定层与层之间神经元的连接方式，也就确定了各层之间的权值矩阵
5. 损失函数的确定
6. 权值矩阵和偏置矩阵的调整策略（随机梯度下降等）

完成神经网络的设计之后就可以通过 tensorflow 提供的 API 搭建设计的模型，步骤如下：

1. 创建神经网络
2. 定义和设置各个神经元的激活函数
3. 定义损失函数 loss
4. 创建优化器(optimizer)(对应神经网络模型中的权值和偏置矩阵调整策略的选择)

二、Tensorflow API 简介

1. 创建变量

创建变量包括两个步骤，创建和初始化。创建变量的 API 是 `Variable()`；初始化的 API 是 initialize_all_variables()。Tensorflow 中变量主要是用来表示权值和偏置矩阵的，还有另外一个作用就是作为 tensorflow 在求偏导数或者梯度时的依据，若损失函数中没有变量，则无法计算梯度，也就无法使用梯度下降等优化策略。

2．在会话中执行初始化、优化策略

当完成变量、损失函数以及优化策略的设置之后，就可以启动一个会话，必须让 Tensorflow 执行变量初始化，因为在使用优化策略优化损失函数(训练模型)时需要执行 bp 算法，需要对某些变量(Tensor)进行链式求导（求导过程自动完成）。

初始化方法：

```
init = tf.initialize_all_variables()
…….
sess.run(init)
```

3．输出变量的值

在会话中执行了变量的初始化后，可以通过　会话.run(对象) 来输出对象的值，对象可以是通过 variable 函数创建的变量，也可以是包含变量的表达式。例如：

a = tf.Variable(tf.zeros[1]); b = a + 1; 查看 b 的值就可以通过 sess.run(b)来查看。

4．模型的保存和恢复

网络模型训练完毕后可以保存到文件中，以便之后的继续训练或者预测。

4.1 模型保存

```
saver = sess.train.Saver()
…….
#训练过程
…….
saver.save(sess,”保存路径文件名”)
```

4.2 模型恢复

必须保证构建的新模型和要恢复的模型一致。

```
saver = sess.train.Saver()
saver.restore(sess,”保存路径文件名”)
…….
#继续训练或者预测
…….
```

见示例 2.2 XOR 的实现

5．Tensorflow 图的可视化

tensorflow 可以将所构建的图通过 tensorboard 工具显示出来，为了可视化，在程序中需要通过 tf.scope_name(“标签”)创建一个视图窗口，窗口的名称由 scope_name 的参数指定，在窗口中显示存在的 Tensor 和子窗口的名称，示例：

```
#定义名为”layer_1”的窗口，其中包含两个 Tensor（权值矩阵和偏置矩阵）
with tf.scope_name(“layer_1”):
    with tf.scope_name(“weights”):
        w = tf.Variable(tf.random_normal([2,2]),name=”w_1”)
        b =  tf.Variable(tf.random_normal([2]),name=”b_1”)
    with tf.scope_name(“wx_plus_b”):
        wx_plus_b = tf.add(tf.matmul(w,x),b)
```

以上代码的结果会产生一个名为 layer_1 的顶层窗口，该窗口中有两个子窗口分别为 weights 和 wx_plus_b。其中 weights 窗口中包含两个 Tensor 分别为 w_1 和 b_1。

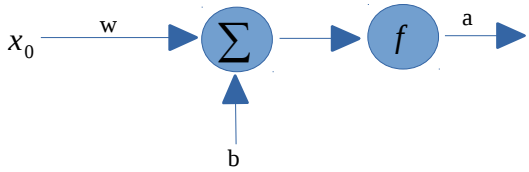
见示例 2.3 非线性函数的拟合

三、线性回归示例演示

3.1 神经网络模型

对于线性回归神经网络模型中只需要单层单神经元即可，下图就是一个单层单神经元网络模型，权值矩阵 $W=[w]$,网络的输出为 $a=f(wx+b)$ ，其中 f 是激活函数，因为是线性回归，因此选择的激活函数为线性函数，即： $a=f(wx+b)=wx+b$,损失函数设为均方误差，即 $loss=(a_i-y_i)^2$ ，权值矩阵和偏置矩阵采用随机梯度下降法更新。

神经网络模型如下：



3.2 使用 Tensorflow 搭建神经网络模型

(1). 批量梯度下降

此处列出一些关键的步骤。

```
import tensorflow as tf
import numpy as np
```

a. 构建阶段：创建权值矩阵和偏置矩阵作为变量

```
#设置神经网络模型的层数，总共一层
#（Tensor）创建一个 1*1 的矩阵，初始值为-1.0~1.0 间的随机数
weights = tf.Variable(tf.random_uniform([1],-1.0,1.0))
#创建偏置矩阵，1*1,初始值为 0
bias =  tf.Variable(tf.zeros([1]))
#定义激活函数
define purline(input,weights,bias):
        return weight*input + bias
#（op）神经网络模型输出,a 的 dimension:1*100
a = purline(x_data,weights,bias)

#创建训练数据集 x_data，dimension :1*100 和 y_data,dimension: 1*100
x_data = np.random.rand(100).astype("float32")
y_data = x_data *0.1 + 0.3

#（op）定义损失函数,reduce_mean 求平均值，此时输入的是所有训练数据
loss = tf.reduce_mean(tf.square(a – y_data))
#（op）定义优化策略梯度下降,0.5 为学习速度(下降速度),如果选择梯度上升则
maximum(loss)
train = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
```

b. 执行阶段：创建会话和执行优化策略

```
#初始化所有变量
init = tf.initialize_all_variables()
#创建会话
sess = tf.Session()
#执行初始化
sess.run(init)
```

```
#训练(执行优化策略),所有的数据参与模型的训练，循环执行 50 次
for step in xrange(0, 50):
    sess.run(train)
    if step % 2 == 0:
        #打印权值矩阵和偏置矩阵
            print step, sess.run(weights), sess.run(bias)
```

程序源码见附件 1：batch_gradient_descent.py

(2)随机梯度下降

与批量梯度下降的不同在于，随机梯度下降每次迭代只输入一个训练数据，每迭代一次就会更新一次权值矩阵和偏置矩阵。如何在每次迭代过程中修改输入变量的值为每个训练数据?

此处主要涉及到了两个操作:

1. tf.placeholder

该函数用来创建一个变量，这个变量一开始时的值是不确定的，是个占位符，运行之前需要通过 feed_dict 属性来填充。

2. feed_dict 是会话的 run 方法的参数，通过该参数就可以在运行时给指定的 placeholder 的变量填充数值。

示例:

```
x = tf.placeholder(“float32”)#参数指定了 x 的类型
sess = tf.Session()
init =  tf.initialize_all_variables()
sess.run(init)
#给变量 x 填充值 10
sess.run(x,feed_dict={x:10})
```

因此只需要修改程序使得每次输入一个数据去训练模型即可实现随机梯度下降。关键代码如下:

```
x_i = tf.placeholder(‘float32’)
y_data_i = tf.placeholder(‘float32’)
y_i = purline(x_i,weights,bias)
……..
#训练模型,顺序输入 100 个训练数据,执行 1000 次迭代
for step in range(1000):
    #填充 x_i 和 y_data_i 的值
    sess.run(train,feed_dict={x_i:x_data[step%100],y_data_i:y_data[step%100]})
    ……...
```

程序源代码见附件 2：stochastic_gradient_descent.py

四、 非线性分类和回归

4.1 实现异或的网络模型

训练数据为： $X=[[1,1],[1,0],[0,1],[0,0]]$ ，对应的标签为 $Y=[[1,0],[0,1],[0,1],[1,0]]$ ，其中 Y 中的每个标签是 one-hot 向量,第一位表示结果是否为 0，第二位表示结果是否为 1。

4.1.1 隐藏层

隐藏层需要两个神经元，来获取提取数据的两类的特征，因为每个神经元的一次输入包含两个数据，因此权值矩阵和偏置矩阵为：

$$W^1=\begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} \quad b^1=\begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$$

激活函数为 sigmoid 函数，则该层的输出为：

$$a_1=f(W^1X+b^1)=sigmoid(W^1X+b^1)$$

4.1.2 输出层

输出层根据隐含层提取的特征得出各个类别的概率，同时由于只有两个类别，因此只需要 2 个神经元，分别计算处各个类别的概率，因此激活函数 为 softmax 函数，而隐含层的输出作为输出层的输入，而隐含层的输出 2 个结果，因此权值矩阵和偏置矩阵为：

$$W^2=\begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{bmatrix} \quad b^2=\begin{bmatrix} b_1^2 \\ b_2^2 \end{bmatrix}$$

输出层的输出为：

$$a_2=f(W^2a_1+b^2)=softmax(W^2a_1+b^2)$$

4.1.3 损失函数和优化策略

损失函数选择交叉熵：

$$loss=-\sum y_i\log(a_{2i})$$

优化策略为梯度下降。

4.1.4 构建模型并且保存模型

关键代码说明：

def add_layer(input_data,in_size,out_size,activation_function=None)

函数功能：构建一层网络所需要的 Tensor（包含权值矩阵和偏置矩阵）和 op(激活函数)

函数参数：input_data:该层网络的输入数据；in_size：表示输入数据的个数；out_size: 表示该层网络的输出数据的个数；activation_function 指定激活函数。

返回说明： 类型为 Tensor（该层网络的输出数据）

```
#每次输入所有数据,其中 None 表示输入的数据个数可以通过 feed_dict 填充时的数据指定。
x_i = tf.placeholder(tf.float32,[None,2])
y_data_i = tf.placeholder(tf.float32,[None,2])
#构建图，包括所有的 op 和 Tensor
a_1 = add_layer(x_i,2,2,tf.nn.sigmoid)
a_2 = add_layer(x_i,2,2,tf.nn.softmax)
#设置损失函数,其中 reduce_sum(x)对 Tensor 变量 x 的所有元素求和，也可以通过参数 2 指定操作的轴，reduce_mean 同理。
loss = tf.reduce_mean(-1 * tf.reduce_sum(y_data_i * tf.log(a_2)))
…….
…….
#训练完成后，保存模型参数，包括各层的权值和偏置等信息。
saver = tf.train.Saver()
#开始训练
for step in range(3000):
    #每次迭代输入所有的数据，参加模型的训练
    sess.run(train,feed_dict={x_i:x_data,y_data_i:y_data})
#保存模型参数到文件名为 mode.ckpt 中
saver.save(sess,"mode.ckpt")
#检测训练后的网络
#a_2 是网络模型的输出，而 a_2 的计算仅依赖 x_i,因此需要使用 feed_dict 分别填充 4 组数据检
#验结果，其中 a_2 的输出两个概率，概率较大的元素所在的位值表示该输入数据所属的分类
print (x_data[0],y_data[0],sess.run(a_2,feed_dict={x_i:x_data[0]})
print (x_data[1],y_data[1],sess.run(a_2,feed_dict={x_i:x_data[1]})
print (x_data[2],y_data[2],sess.run(a_2,feed_dict={x_i:x_data[2]})
print (x_data[3],y_data[3],sess.run(a_2,feed_dict={x_i:x_data[3]})
```

4.1.5 模型的恢复

首先必须保证构造的图(op 和 Tensor)与要恢复的网络模型一致，才能恢复成功。

在该程序中所使用的 demo 恢复模型的步骤包括：

1、构造原图；2、恢复模型；3、进行预测(或继续训练)

关键代码如下：

```
#在训练之前恢复模型
x_i = tf.placeholder(tf.float32,[None,2])
y_data_i = tf.placeholder(tf.float32,[None,2])
#构建图，包括所有的 op 和 Tensor
a_1 = add_layer(x_i,2,2,tf.nn.sigmoid)
a_2 = add_layer(x_i,2,2,tf.nn.softmax)
#设置损失函数,其中 reduce_sum(x)对 Tensor 变量 x 的所有元素求和，也可以通过参数 2 指定操作的轴，reduce_mean 同理。
loss = tf.reduce_mean(-1 * tf.reduce_sum(y_data_i * tf.log(a_2)))
……..
……..
#训练完成后，保存模型参数，包括各层的权值和偏置等信息。
saver = tf.train.Saver()
#恢复模型(或者恢复上次训练的结果),第二参数指定了要恢复的模型所在文件
saver.restore(sess,"mode.ckpt")
```

```
#直接根据输入预测结果
print (x_data[0],y_data[0],sess.run(a_2,feed_dict={x_i:x_data[0]})
print (x_data[1],y_data[1],sess.run(a_2,feed_dict={x_i:x_data[1]})
print (x_data[2],y_data[2],sess.run(a_2,feed_dict={x_i:x_data[2]})
print (x_data[3],y_data[3],sess.run(a_2,feed_dict={x_i:x_data[3]})
#开始训练
for step in range(3000):
        #每次迭代输入所有的数据，参加模型的训练
        sess.run(train,feed_dict={x_i:x_data,y_data_i:y_data})
#保存模型参数到文件名为 mode.ckpt 中
saver.save(sess,”mode.ckpt”)
#检测训练后的网络
#a_2 是网络模型的输出，而 a_2 的计算仅依赖 x_i,因此需要使用 feed_dict 分别填充 4 组数据检
#验结果，其中 a_2 的输出两个概率，概率较大的元素所在的位值表示该输入数据所属的分类
print (x_data[0],y_data[0],sess.run(a_2,feed_dict={x_i:x_data[0]})
print (x_data[1],y_data[1],sess.run(a_2,feed_dict={x_i:x_data[1]})
print (x_data[2],y_data[2],sess.run(a_2,feed_dict={x_i:x_data[2]})
print (x_data[3],y_data[3],sess.run(a_2,feed_dict={x_i:x_data[3]})
```

详细代码见附件 2：xor.py

其他的功能如自定义 OP 和可视化部分的其他的数据的可视化正在学习中。

4.2 设计一个 BP 神经网络模型来拟合函数：

$$f(x)=1+\sin\left(\frac{\pi}{4}x\right),-2\leqslant x\leqslant 2$$

该神经网络模型需要两层网络，隐藏层和输出层,优化策略为反转传播法。

4.2.1 隐藏层

隐藏层需要两个神经元，与输入数据全连接，而输入数据只有一个，因此每个神经元需要一个权值，则隐藏层的权值矩阵和偏置矩阵为：

$$W^1=\begin{bmatrix} w_{11}^1 \\ w_{21}^2 \end{bmatrix}, b^1=\begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$$

其中 w_{mn}^k 表示网络模型的第 k 层的第 m 个神经元的第 n 条输入的权值。 b_m^k 表示网络模型的第 K 层的第 m 个神经元的偏置。

隐藏层的激活函数为对数-S 形函数： $a^1=f(W^1x+b^1)=logsig(W^1x+b^1)$,其中:

$$logsig(x)=\frac{1}{1+e^{-x}}$$

4.2.2 输出层

该层只有一个神经元，因此权值矩阵和偏置矩阵为：

$$W^2=\begin{bmatrix} w_{11}^2 \end{bmatrix}, b^2=\begin{bmatrix} b_1^2 \end{bmatrix}$$

该层的激活函数为 purline 函数：

$$a^2=f(W^2a^1+b^2)=purline(W^2a^1+b^2)=W^2a^1+b^2$$

4.2.3 网络模型的搭建与可视化设置

搭建过程跟之前的模型类似，本程序主要为可视化加入了标签了，以 add_layer 举例：

```
def add_layer(input_data,in_size,out_size,activation_funcation = None,label = “1”):
        lay_label = “layer”+label
        with tf.scope_name(laybel):
            with tf.scope_name(“weights”):
                w = tf.Variable(tf.random_normal([in_size,out_size]),name=”w”+label)
            with tf.scope_name(“biases”):
                b = tf.Variable(tf.random_normal([out_size]),name=”b”+label)
            with tf.scope_name(“wx_plus_b”):
                wx_plux_b = tf.matmul(w,input_data) + b
        …..
        return …..
sess.run(init)
……
#图构建完成后就可以保存图模型
writer = tf.summary.FileWriter(“图保存的路径/”,sess.graph)
#训练模型
…….
```

4.2.4 图可视化方法

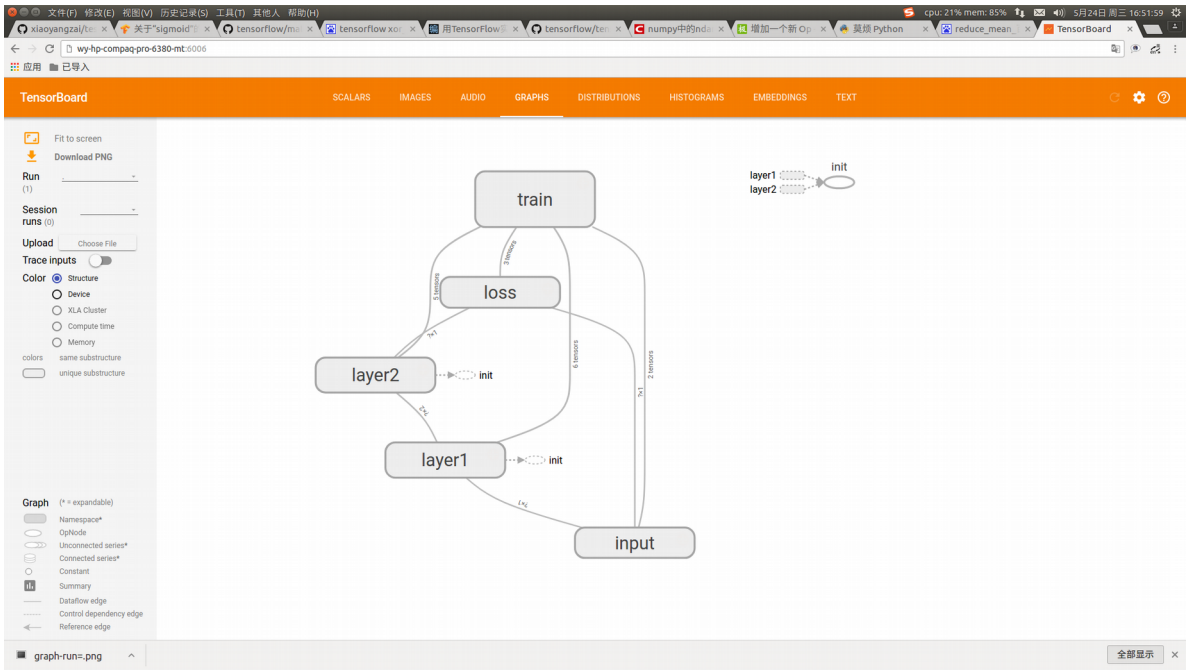
程序运行完成后，在终端中输入并执行：

tensorboard –logdir=图保存的路径/

随后将输出的网址:端口号粘贴至浏览器，网址示例如下：

http://wy-hp-compaq-pro-6380-mt:6006

浏览器显示如下，选择 GRAPHS 即可看到模型：



详细代码见附件 3：backpropagate.py