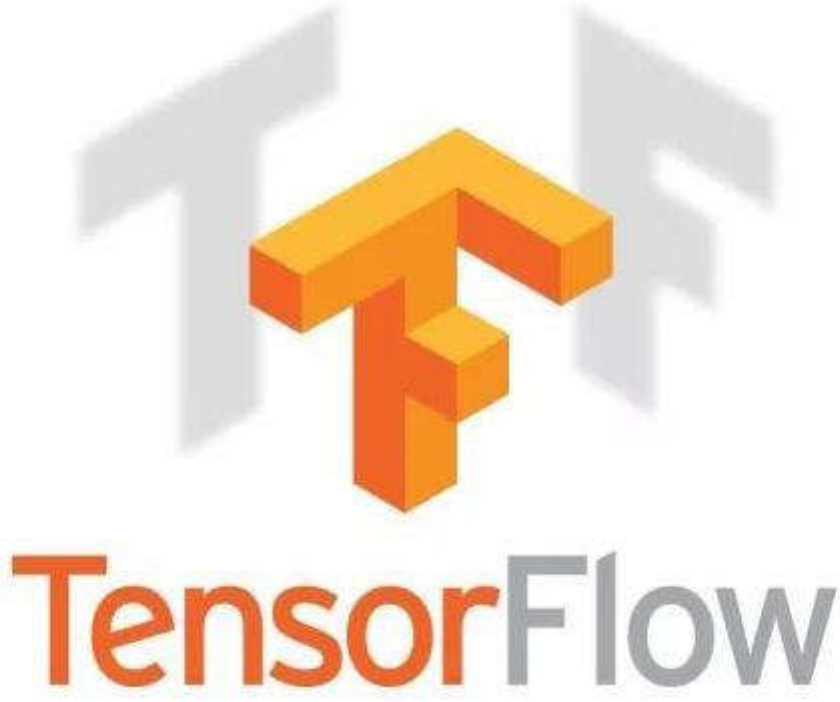

从线性回归入门Tensorflow

白裳
搞事情的直接拉黑不解释，离我远点。
关注他

为什么选择Tesorflow?

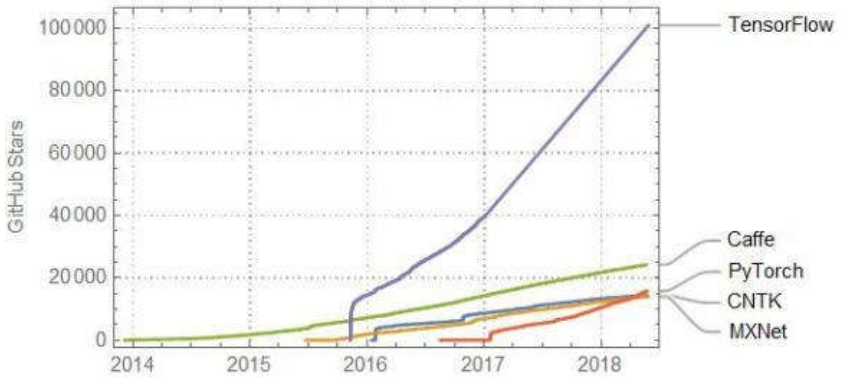

图1 GitHub深度学习框架Stars（2018.4）

TensorFlow是由Google开源的深度学习框架，也是目前最受欢迎的深度学习框架。截止2018.4，在GitHub上TensorFlow项目的Stars已经超过10K，基本是完爆竞争对手的节奏，非常恐怖。

该框架已经很好的集成了深度学习的各种算法，使用非常方便；再加上Google强大的生态与研发实力，选择TensorFlow进行深度学习是非常明智的。

但是，Tensorflow也并不是完美的。TensorFlow相当于在Python中重新创建了一套语言，而且引入非常多的概念，学习起来非常痛苦，对于新手有很强的反作用力....

所以本文希望从一个简单的二维线性回归例子带你入门Tensorflow框架。

来一起入门Tensorflow吧！

既然是程序员类入门文章，那么一起来看看“Hello world”吧：

```
# Example 1
import tensorflow as tf
tensor = tf.constant('Hellow Tensorflow!')
sess = tf.Session()
print sess.run(tensor)
sess.close()
```

例1其实就是创建了一个常量Tensor，然后开启Session，并在Session中打印出字符串。当然，也可以用Python with语句写的更Tensorflow一点：

```
# Example 1 - use with scope
import tensorflow as tf
tensor = tf.constant('Hellow Tensorflow!')
with tf.Session() as sess: # use python with scope
    print(sess.run(tensor))
```

再来看一个稍微复杂一点的例子：

```
# Example 2
import tensorflow as tf
w = tf.constant(0.6, name='w')
x = tf.constant(2.0, name='x')
b = tf.constant(-0.7, name='b')
y = tf.add(tf.multiply(w, x), b) # y = w * x + b

with tf.Session() as sess:
    print sess.run(y)
```

即创建了3个常量Tensor，并且通过Operation tf.multiply()和tf.add()连接Tensor获得y。

不过呢，上面使用的这种tf.constant()类型的常量Tensor，类似于C语言的const变量，一旦创建值就不可变了。无论线性回归还是神经网络，都需要通过训练改变Tensor值，所以这种常量Tensor肯定无法满足需求。所以下面改用可变的Tensor来修改例2：

```
# Example 3
import tensorflow as tf
x = tf.placeholder(dtype=tf.float, shape=(), name="x")

w = tf.get_variable(name='w', shape=(),
                    initializer=tf.constant_initializer(0.6))
b = tf.get_variable(name='b', shape=(),
                    initializer=tf.constant_initializer(-0.7))
y = tf.add(tf.multiply(w, x), b)

init_op = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init_op)
    print sess.run(y, feed_dict={x:2.0})
```

在例3中：

1. 首先创建了一个占位符x，即tf.placeholder创建的Tesnor
2. 然后创建了可变w和b，即tf.get_variable创建的Tensor
3. 最后通过tf.add和tf.multiply operation连接了一个新的y Tensor，有点类似于把Tensor连接成一个管道
4. 最后在Session中初始化w和b，并通过feed_dict向tf.placeholder类型的Tensor输入数据，这时

候Session会按照Tensor连接方式进行Dataflow

Tensorflow重要概念：Tensor，Operation，Graph，Session

看到这里，应该已经基本了解了Tensorflow的基本语法。但是上文中提到的Tensor等概念到底是什么意思?

=> Tensor

Tensor，张量，可以简单的理解为多维数组。

```
# Example 4
import tensorflow as tf
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], dtype=tf.float32, name='a')
print('shape =', a.get_shape().as_list()) # shape = [2, 3]
print('dtype =', a.dtype) # dtype = <dtype: 'float32'>
print('name =', a.name) # name = a:0
```

Tensor有3个重要属性：

- shape：Tensor的形状，与多维数组numpy.array shape类似
- dtype：Tensor的数据类型，与多维数组numpy.array dtype类似，常用的类型有：

> tf.uint8: 8-bit unsigned integer.
> tf.int32: 32-bit signed integer.
> tf.int64: 64-bit signed integer.
> tf.String: String.
> tf.float32: 32-bit single-precision floating-point.
> tf.float64: 64-bit double-precision floating-point.

- name：Tensor的名称。

对于Tensor的name属性需要注意的是：

1. 每一个Tensor都必须有name属性。在创建Tensor时，如果用户没有指定name，Tensorflow会自动设置。
2. 在同一张Graph中，不会有Tensor name重名。当用户设定的name有重复时，Tensorflow会自动加入后缀进行区分。

具体如例5：

```
# Example 5
import tensorflow as tf
a = tf.constant(1)
print(a.name) # Const:0

b = tf.constant(1, name='test')
c = tf.constant(1, name='test')
print(b.name, c.name) # test:0 test_1:0
```

最后需要注意：在Tensorflow中Tensor只是控制数据的“符号”。

怎么理解这句话呢?

1. 通过Tensor的确可以控制数据，所以Tensor直观上看起来“像是数据的容器”。
2. 但是Tensor只是一个控制数据的符号，本身不存储数据，并不是数据容器。只有在Dataflow的时候才能通过Tensor提取到的数据。

通过Tensor类成员函数eval()函数可以获取Tensor中的数据。举例说明：

```
# Example 6
import tensorflow as tf
tensor = tf.constant('Hellow Tensorflow!')
# print(tensor.eval()) # error!
with tf.Session() as sess: # use python with scope
    print(tensor.eval()) # correct, print "Hellow Tensorflow!"
```

在例6第4行中，直接通过eval()读取Tensor中的数据会报错；而在第6行，开启了Session进行Dataflow的时候，eval()则能正确的打印。

这就证明，只有开始Session进行Dataflow的时候，Tensor才会获取到数据。

所以：Tensor里面真的不保存数据！理解Tensor必须要掌握这一点细节。

另外，你可以用tf.device() API来控制每个Tensor的物理位置，如在那个CPU还是GPU。

```
# Example 6 - with tf device control
import tensorflow as tf
with tf.device('/cpu:0'):
    tensor = tf.constant('Hellow Tensorflow!') # tensor is located in CPU 0
with tf.Session() as sess:
    print(tensor.eval()) # print "Hellow Tensorflow!"
    print(tensor.device) # print "/device:CPU:0"
```

=> Operation

Operation，操作符，即将多个Tensor连接在一起，形成新的Tensor。

```
# Example 7
x = tf.placeholder(dtype=tf.float, shape=(), name="x")
y = tf.placeholder(dtype=tf.float, shape=(), name="y")
z = tf.add(x, y)
```

在上面的例子中，通过tf.add() Operation连接x和y，形成新的Tensor z。这其实就是建立数据流动的方式。

```
init_op = tf.global_variables_initializer()
```

当然，包括例3中的初始化init_op都是Operation。

=> Graph

TensorFlow的运算，都被表示为一个数据流图Graph。

一幅图中包含一些Operation，这是计算结点。而Tensor则是表示在不同的Operation间的数据结点。

```
# Example 9
import tensorflow as tf

graph = tf.Graph()
with graph .as_default():
    a = tf.constant(1)

assert(a.graph is graph)
```

```
with tf.Session(graph=graph) as sess:
    print(sess.run(a)) # 1
```

在例9中，首先显式创建了一个图graph，然后使用上下文处理器graph.as_default()，指定在graph中定义了Tensor a，然后将graph传入sess运行。可以看到如我们所料输出1。

那么，当没有显式创建图的时候呢？其实Tensorflow为我们准备了一个默认图Default Graph，所有的操作都定义在默认图中而已。

```
# Example 10
import tensorflow as tf

a = tf.constant(0)
assert(a.graph is tf.get_default_graph())

with tf.Session() as sess:
    assert(sess.graph is tf.get_default_graph())
    print(sess.run(a)) # 0
```

注意到，在正常使用时，同一个Graph中有很多的Tensor和Operation。保持清晰可读的Graph非常重要，便于用户查看Graph结构。所以建议使用tf.variable_scope()对Tensor进行域管理，使Graph条理更加清晰。

```
# Example 11
import tensorflow as tf
with tf.variable_scope("SSD"):
    with tf.variable_scope("ResNet"):
        tensor = tf.get_variable("conv", shape=[1, 3, 3, 1])
print('tesnor_name =', tensor.name) # tesnor_name = SSD/ResNet/conv:0
```

=> Session

Session包含了操作Operation执行的环境。

一般通过向Session传入Graph后，通过Session run对Graph内的数据进行处理：

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

如果不指定传入Graph，则默认使用Default Graph：

```
with tf.Session(graph=tf.get_default_graph()) as sess:
    ......
```

通过开启Session才能使Tensor中的数据真正流动flow起来。

了解概念之后，Tensorflow的工作流程如下：

1. 根据需求，创建计算图Graph
2. 开启会话Session，通过sess.run(op, feed_dict={...})输入数据运行Graph
3. 获取结果

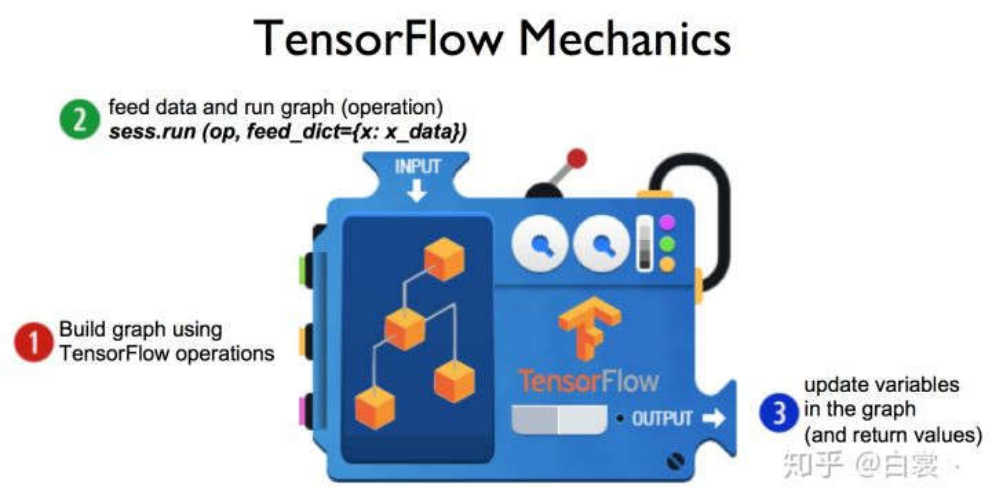

图2 Tensorflow mechanics

Tensroflow线性回归

了解Tensorflow的基本流程后，接下来一起看一个二维线性回归的例子。

已知在 R^2 空间有一组二维数据点：

$$X = \{x_0, x_1, \dots x_n\}$$

$$Y = \{y_1, y_2, \dots, y_n\}$$

如图3，有红色的数据点，我们希望拟合一条蓝色直线 $f(x) = Wx + b$ ，其中 W 和 b 是需要学习的参数。

使得损失函数 $\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$ 最小，即所有绿色线段长度之和最小。

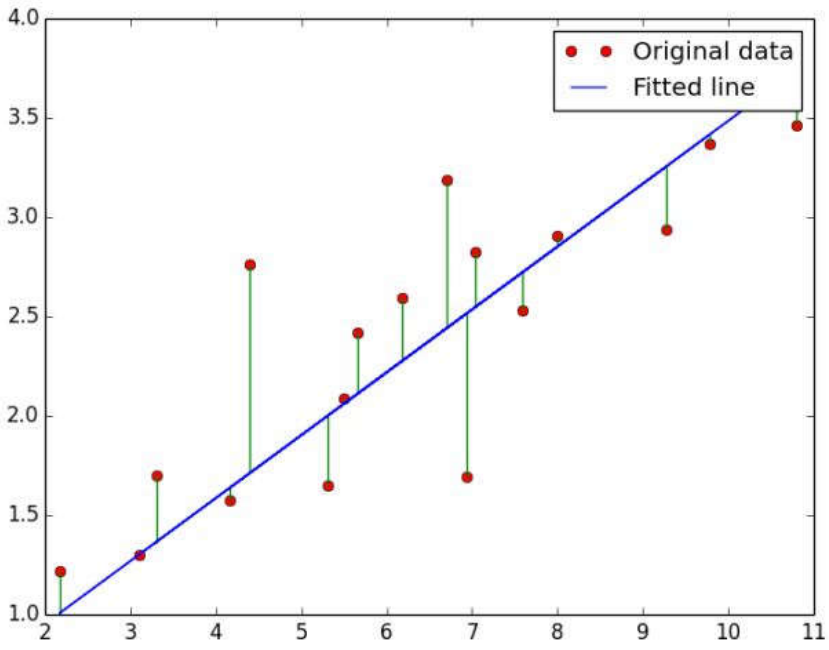

图3 二维线性回归

那么接下来，使用Tensorflow对上述问题建模，并使用梯度下降法求解 W 和 b 参数。

- Step1：首先导入必须的Python模块，并定义训练数据

```
# Example 12
from __future__ import print_function

import tensorflow as tf
import numpy
import matplotlib.pyplot as plt
rng = numpy.random

# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50
save_step = 500

# Training Data
train_X = numpy.asarray([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,
                         7.042,10.791,5.313,7.997,5.654,9.27,3.1])
train_Y = numpy.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.221,
                         2.827,3.465,1.65,2.904,2.42,2.94,1.3])
```

- Step2：然后定义Input tensor X与label tensor Y

```
# Define graph input
X = tf.placeholder("float", name="X")
Y = tf.placeholder("float", name="Y")
```

- Step3：定义线性回归结构 $y = Wx + b$

```
with tf.variable_scope("liner_regression"):
    # Set model weights
    W = tf.get_variable(initializer=rng.randn(), name="weight")
    b = tf.get_variable(initializer=rng.randn(), name="bias")

    # Construct a linear model
    mul = tf.multiply(X, W, name="mul")
    pred = tf.add(mul, b, name="pred")
```

- Step4：定义损失函数，即 $\mathbf{Loss} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$

```
# Mean squared error loss(L2 loss)
with tf.variable_scope("l2_loss"):
    loss = tf.reduce_mean(tf.pow(pred-Y, 2))
```

- Step5：创建梯度下降优化operation

```
# Gradient descent
#  Note, minimize() knows to modify W and b because Variable objects
#  are trainable=True by default
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- Step6：创建初始化operation

```
# Initialize the variables (i.e. assign their default value)
init_op = tf.global_variables_initializer()
```

- Step7：创建Saver，用于保存训练的model

```
# Checkpoints save path
ckpt_path = './ckpt/liner-regression-model.ckpt'
# Create a saver
saver = tf.train.Saver()
```

- Step8：创建summary monitor

```
# Summary save path
summary_path = './ckpt/'

# Create a summary to monitor tensors which you want to show in tensorboard
tf.summary.scalar('weight', W)
tf.summary.scalar('bias', b)
tf.summary.scalar('loss', loss)

# Merges all summaries collected in the default graph
merge_summary_op = tf.summary.merge_all()
```

- Step9：开启Session，通过feed_dict开始训练，同时保存summary events和checkpoints

```
# Start training
with tf.Session() as sess:

    summary_writer = tf.summary.FileWriter(summary_path, sess.graph)

    # Run the initializer
    sess.run(init_op)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            # Do feed dict
            _, summary = sess.run([train_op, merge_summary_op], feed_dict={X: x, Y: y

        # Save model per epoch step
        if (epoch+1) % save_step == 0:
            # Do save model
            save_path = saver.save(sess, ckpt_path, global_step=epoch)
            print("Model saved in file: %s" % save_path)

        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            # Display loss and value
            c = sess.run(loss, feed_dict={X: train_X, Y: train_Y})
            print("Epoch:", '%04d' % (epoch+1), "loss=", "{:.9f}".format(c),
                  "W=", W.eval(), "b=", b.eval())

            # Add variable state to summary file
            summary_writer.add_summary(summary, global_step=epoch)

    print("Optimization Finished!")

    # Save the final model
    save_path = saver.save(sess, ckpt_path, global_step=epoch)
    print("Final model saved in file: %s" % save_path)

    # Close summary file writer
    summary_writer.close()
```

```
# Calculate final loss after training
training_loss = sess.run(loss, feed_dict={X: train_X, Y: train_Y})
print("Training loss=", training_loss, "W=", sess.run(W), "b=", sess.run(b), '\n'

# Graphic display
plt.plot(train_X, train_Y, 'ro', label='Original data')
plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
plt.legend()
plt.show()
```

伴随着训练，通过梯度下降算法修正Tensor W和b参数，直线变化如下：

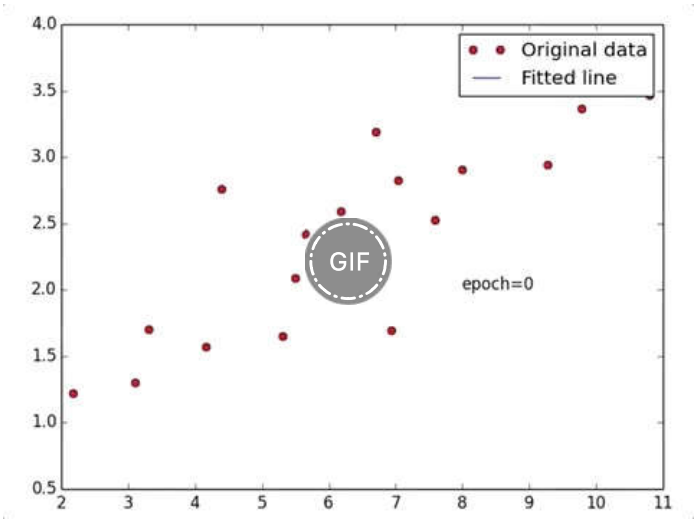

图4 梯度下降过程中直线的变化

当在Session中开启feed_dict时：

```
# Do feed dict
_, summary = sess.run([train_op, merge_summary_op], feed_dict={X: x, Y: y})
```

整个Default Graph的数据就开始流动起来，通过train_op计算梯度Gradients，然后使用梯度修正Tensor。具体流程类似于：

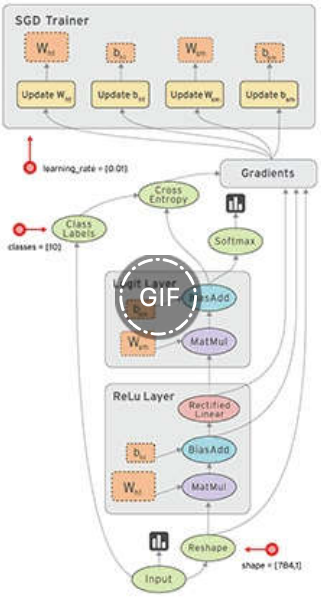

图5 Tensorflow训练过程中的数据流动

Tensorboard的使用

由于在训练代码Step8/Step9中我们有将summary写入到文件中：

```
summary_path = './ckpt'
# Start training
with tf.Session() as sess:
    summary_writer = tf.summary.FileWriter(summary_path, sess.graph)
    summary_writer.close()
```

运行后会在summary_path目录生产一个名为events.out.tfevents.xxxx的文件，通过Tensorflow自带可视化工具Tensorboard就可以查看：

```
tensorboard --logdir=./ckpt/
```

用Google chrome浏览器打开显示的网址，可以看到：

- Graph界面有sess.graph的可视化图。由于在Step3/Step4中使用了tf.varaible_scope()控制Tensor name域，整个图看起来很清晰。

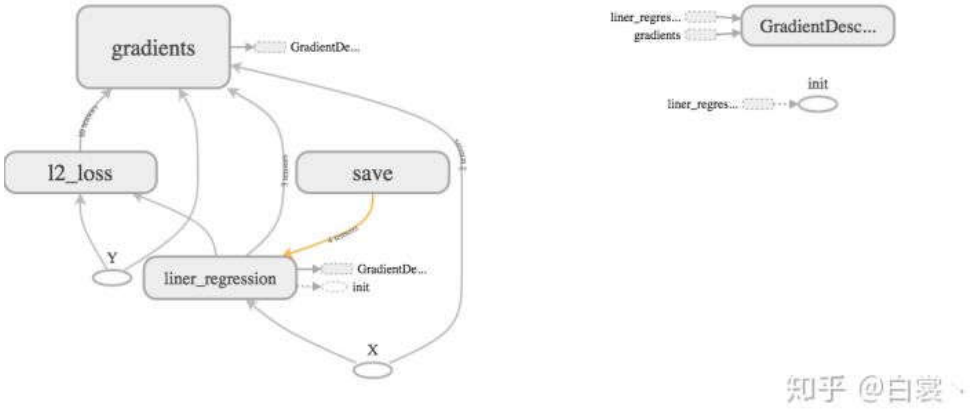

图6 Tensorboard绘制Example8 Graph

- 另外Step8添加了监视的weight/bias/loss monitor，所以Scalar界面有weight/bias/loss的值随global_step变化的曲线

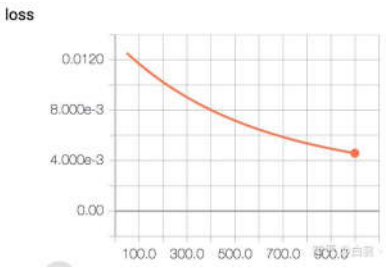

图7 Tensorboard绘制loss/global_step曲线

模型checkpoint有了，接下来如何使用呢？

在之前的线性回归Step9中，使用了tf.train.Saver()保存了将训练的模型保存了下来

```
# Do save model
save_path = saver.save(sess, ckpt_path, global_step=epoch)
```

那么在ckpt_path路径下，会生成如下的几个文件：

```
checkpoint
liner-regression-model.ckpt-499.data-00000-of-00001
liner-regression-model.ckpt-499.index
liner-regression-model.ckpt-499.meta
liner-regression-model.ckpt-999.data-00000-of-00001
liner-regression-model.ckpt-999.index
liner-regression-model.ckpt-999.meta
```

明显可以看到分别保存了step等于499和999时的checkpoint。这就是保存的训练好的model。

那么，使用checkpoint有两种办法：

- 定义一个和原来一模一样的Graph，然后restore checkpoint加载数据，即将checkpoint中的数值导入到当前Default Graph中的各个Tensor中

```
from __future__ import print_function

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy
rng = numpy.random

# Define graph input
X = tf.placeholder("float", name="X")
Y = tf.placeholder("float", name="Y")

with tf.variable_scope("liner_regression"):

    # Set model weights
    W = tf.Variable(rng.randn(), name="weight")
    b = tf.Variable(rng.randn(), name="bias")

    # Construct a linear model
    mul = tf.multiply(X, W, name="mul")
    pred = tf.add(mul, b, name="pred")

# Initialize the variables (i.e. assign their default value)
init_op = tf.global_variables_initializer()

# Create a saver
saver = tf.train.Saver()

# Save path
ckpt_path = './ckpt/liner-regression-model.ckpt-999'

# Start training
with tf.Session() as sess:

    # Restore model parameters to sess.graph
    saver.restore(sess, ckpt_path)

    print('Restored value W={}, b={}'.format(W.eval(), b.eval()))

    # Do whatever you want
    ......
```

- 由于在checkpoint meta文件中有保存之前的Graph，所以可以直接通过meta文件读取Graph，然后再restore checkpoint加载数据

```
from __future__ import print_function

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

config = tf.ConfigProto(allow_soft_placement=True)

ckpt = './ckpt/liner-regression-model.ckpt-999'

with tf.Session(config=config) as sess:

    # Import the saved meta graph into sess graph
    saver = tf.train.import_meta_graph(ckpt + '.meta')

    # Restore model parameter to sess.graph
    saver.restore(sess, ckpt)

    graph = sess.graph

    # Get input tensor for meta graph
    X = graph.get_tensor_by_name('X:0')

    # Get output tensor
    pred = graph.get_tensor_by_name('liner_regression/pred:0')

    # Do whatever you want
    ......
```

扩展：训练深度网络时怎么做？

其实训练包括cnn在内的深度网络，整个流程与上面的线性回归基本流程一致：

1. 定义输入数据，如cnn的输入数据是一个图片组成 NHWC 形状的numpy数组，替换Step1
2. 定义网络结构，如ResNet50结构，替换Step2/Step3
3. 定义网络Loss，如SoftmaxLoss，替换Step4
4. 设置合适的学习率LearningRate和优化器Optimizer（如GradientDescentOptimizer，AdamOptimizer等），替换Step5
5. 开启Session进行训练，并不断保存summary和checkpoint以备后使用。

这样就可以进行cnn等深度网络的训练啦。其实大同小异。

具体工作就留给各位有兴趣的读者作为Homework喽！

编辑于 2018-12-25

机器学习 深度学习（Deep Learning） TensorFlow