

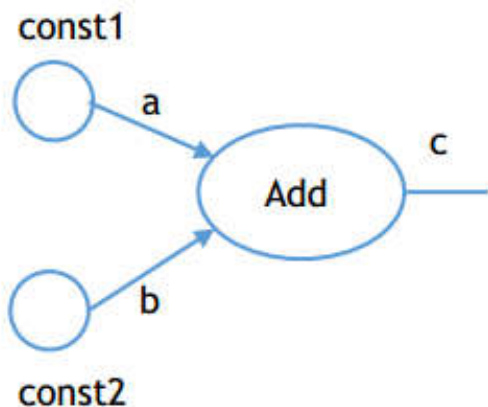
Tensorflow进阶一

计算机学院并行与分布处理国家重
点实验室

- 讲授内容
 - 变量与命名空间
 - 机器学习编程框架
 - 模型存储与恢复
- 要求
 - 掌握利用变量命名空间定义复杂模型方法，掌握模型复用方法，掌握TF中机器学习基本编程框架，掌握TF中模型存储与恢复的方法

上节回顾

- Graph定义+ Session执行



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

```
a = tf.constant(1., name='const1')
b = tf.constant(2., name='const2')
c = tf.add(a, b)
```

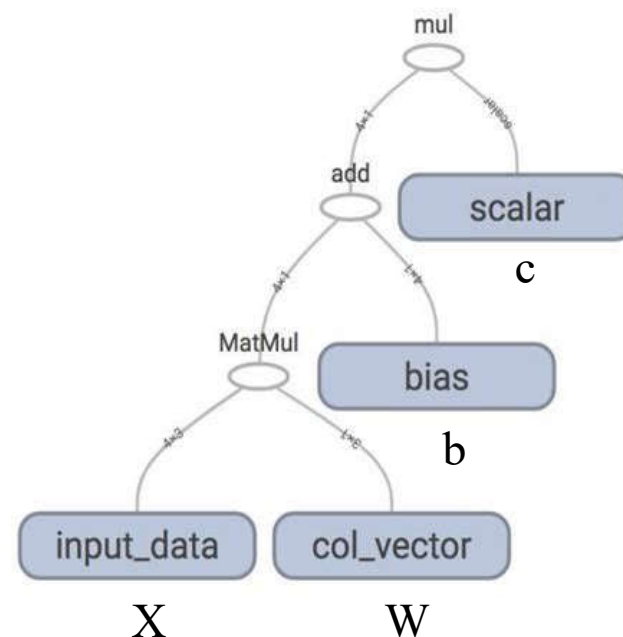
```
with tf.Session() as sess:
    print sess.run(c)
    print c.eval()
```

Graph的主要构成

- Graph 主要构成：（对应于图的元素和作用？）

- Tensor:
- 边，对结果的引用
- Operation:
- 内部节点，操作
- Variable:
- 边缘节点，操作附带的参数
- Placeholder:
- 边缘节点，外部数据输入

- 举例： $((W * X) + b) * c$



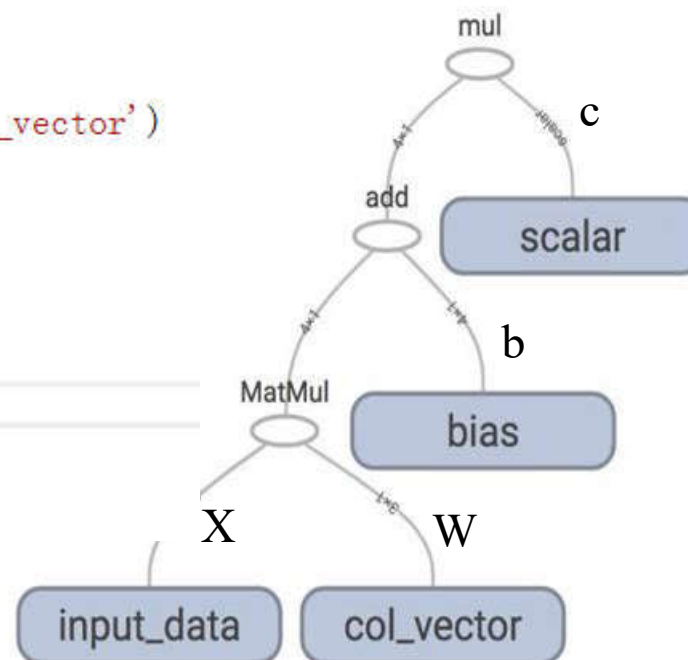
- 注意：TF的构图函数执行后只是画图，并没有直接计算（与Python函数区别！）
- Tensor的作用：引用结果

```
x=tf.placeholder(tf.int32, shape=(1, 2), name='input_data')
w=tf.Variable(np.random.randint(10, size=(2, 1)), name='col_vector')

b=tf.Variable(np.random.randint(10), name='bias')
c=tf.Variable(np.random.randint(10), name='scalar')

rMatMul=tf.matmul(x, w)
rAdd=tf.add(rMatMul, b)

rMul=tf.multiply(rAdd, c)
```



- 计算需要Session启动

Graph惰性计算

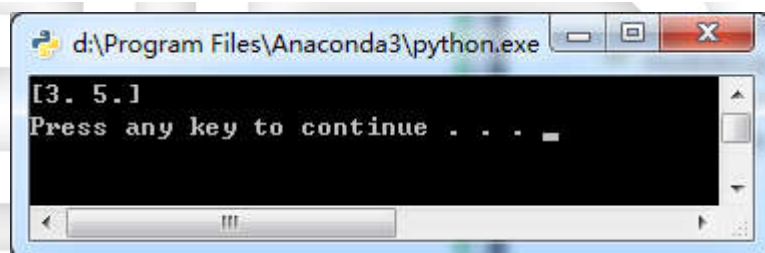
```
import numpy as np
```

```
a=np.array([1.0, 2.0])
```

```
b=np.array([2.0, 3.0])
```

```
results=a+b
```

```
print(results)
```



A terminal window titled 'd:\Program Files\Anaconda3\python.exe' showing the output of a NumPy addition. The output is '[3. 5.]' followed by 'Press any key to continue . . .'. The window has a scroll bar on the right.

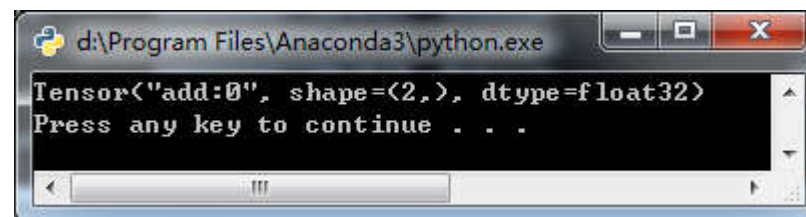
```
import tensorflow as tf
```

```
a=tf.constant([1.0, 2.0], name="a")
```

```
b=tf.constant([2.0, 3.0], name="b")
```

```
results=a+b
```

```
print(results)
```



A terminal window titled 'd:\Program Files\Anaconda3\python.exe' showing the output of a TensorFlow addition. The output is 'Tensor<"add:0", shape=(2,), dtype=float32>' followed by 'Press any key to continue . . .'. The window has a scroll bar on the right.

- 能不能用+, *简单代替tf.add, tf.matmul?

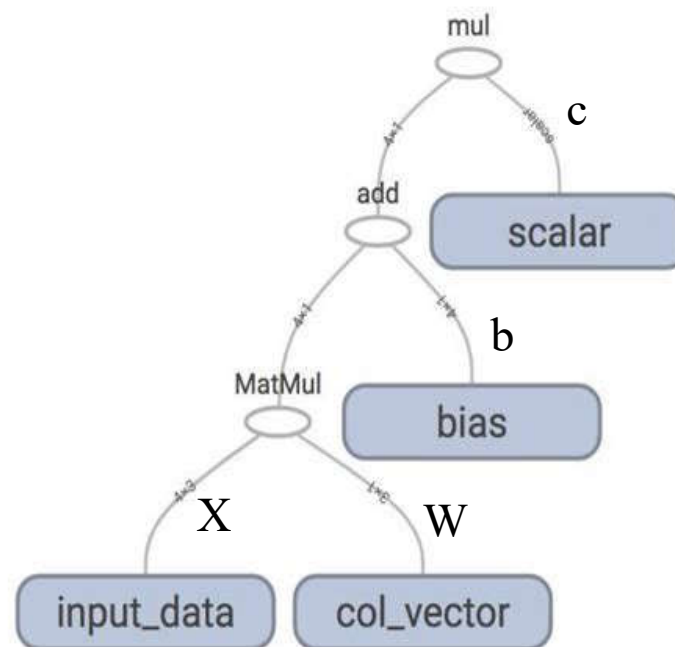
Numpy to TF Dictionary



Numpy	Tensorflow
<code>a = np.zeros([2, 2]); b = np.ones([2, 2])</code>	<code>a = tf.zeros([2, 2]); b = tf.ones([2, 2])</code>
<code>np.sum(b, axis=1)</code>	<code>tf.reduce_sum(a, axis=1)</code>
<code>a.shape</code>	<code>tf.shape(a)</code>
<code>np.reshape(a, [1, 4])</code>	<code>tf.reshape(a, [1, 4])</code>
<code>b*5+1</code>	<code>b*5+1</code>
<code>np.dot(a, b)</code>	<code>tf.matmul(a, b)</code>
<code>a[0, 0], a[:, 0], a[0, :]</code>	<code>a[0, 0], a[:, 0], a[0, :]</code>

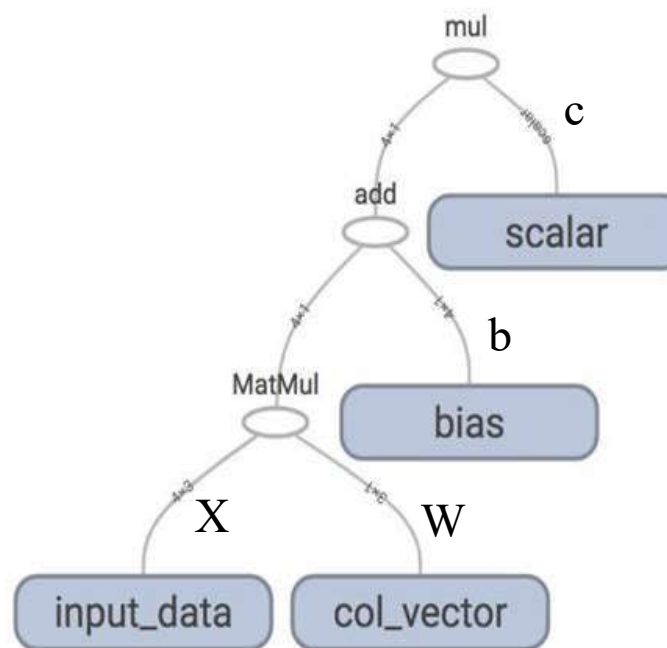


- 都在`sess.run()`中指定
- Feed 给输入结点送数据
- Fetch 指定取哪些结点的结果（用tensor引用）
 - `Sess.run([], feed_dict={x:xxx})`



总结示例

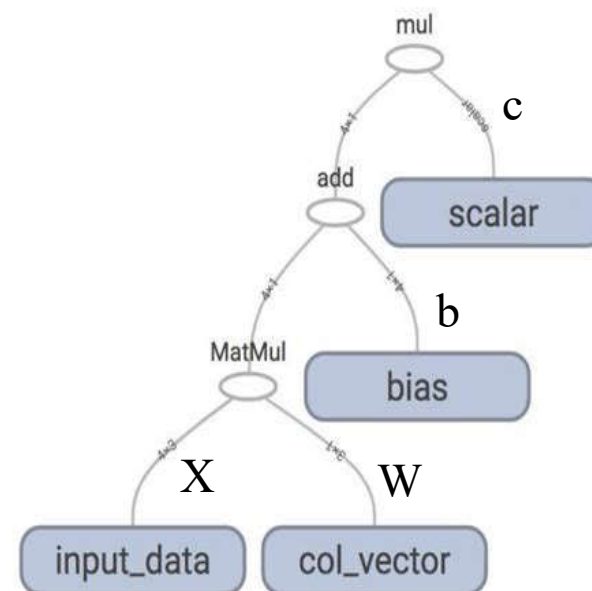
- $((W * X) + b) * c$
- 输入: $X(1,2)$
- 变量: W 、 b 、 c



变量与命名空间

变量用来做什么？

- TF：存储机器学习模型参数
- 例如：线性回归模型： $(W * X) + b$
- W权重，b偏置
- 参数在机器学习过程中不断被调整，所以用变量表示
- Graph：变量是依附其所属操作节点的终端节点



- 基本形式：
- 引用 `tensor=tf.Variable(初始化值, 形状, 数据类型, 是否可训练?, 名字, ...)`
- `w=tf.Variable(initial_value=np.random.randint(10,size=(2,1)),name='col_vector',trainable=True)`
- 变量初始化：参数初始化
- 形状,数据类型暗含在初始化方法里

- 变量的初始化主要分为两步：
- 1. 定义变量时给定初始化值函数：
 - `a=tf.Variable(initial_value=...)`
 - `b=tf.Variable(initial_value=...)`
 - ...
- 2. Session中执行初始化方法：
 -
 - `init= tf.global_variables_initializer()`
 - `sess.run(init)`

- 1. 定义变量时给定初始化值：
`tf.Variable(initial_value=...)`
- 常用的初始化值函数：
 - `tf.constant (const)`: 常量初始化
 - `tf.random_normal ()`: 正态分布初始化
 - `tf.truncated_normal (mean = 0.0, stddev = 1.0, seed = None, dtype = dtypes.float32)`: 截取的正态分布初始化
 - `tf.random_uniform()`: 均匀分布初始化

- 1. 定义变量时给定初始化值
- `tf.Variable(initial_value=...)`
- 还可以用python数据直接初始化，例如：
- `initial_value=np.random.randint(10,size=(2,1))`
- `initial_value=22`

- 注意，即是用常量、随机数直接在变量定义时给定初始化值，变量也此时也是没有值的，需要：
- 2. 在session中run初始化函数：
 - 全部初始化： `tf.global_variables_initializer`
 - 部分初始化： `tf.variables_initializer([a,b,...])`
 - ...
 - `init=tf.global_variables_initializer()`
 - `sess.run(init)`

案例：初始化部分变量

```
import tensorflow as tf
import numpy as np

np.random.seed(1)
b=tf.Variable(initial_value=np.random.randint(10),name='b')
a=tf.Variable(initial_value=np.random.randint(10),name='a')
print(b)
print('b without init',b)

with tf.Session() as sess:
    #sess.run(tf.global_variables_initializer())
    sess.run(tf.variables_initializer([a]))

    ra=sess.run([a])
    rb=sess.run([b])
    print('b with init',rb)
```

- 如何查找未初始化变量并进行初始化呢？

- 变量除了开始设置初始值，如何在计算过程中修改？
- assign operation:赋值
 - `assign()`、`assign_add()`、`assign_sub()`
- 注意：不是我们习惯的
`var=assign(assign_value)` 而是
- `assign_tensor=assign(var, assign_value)`
- 不但可以改值，变量定义参数
`validate_shape=False`，还可以改变量形状

变量更新案例1

```
import tensorflow as tf
import numpy as np

b=tf.Variable(initial_value=np.random.randint(10), name='b')
b=tf.add(b, 1)
out=b*2

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(3):
        print(sess.run([out, b]))
```

- 结果是什么？为什么？

变量更新案例2

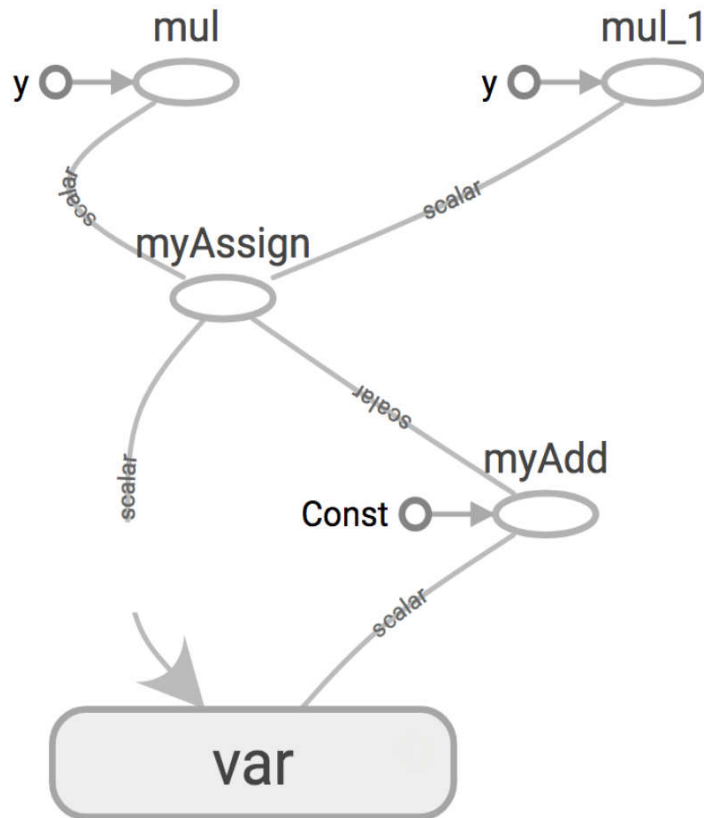
```
import tensorflow as tf
import numpy as np

b=tf.Variable(initial_value=np.random.randint(10), name='b')
#assign_op=tf.add(b,1)
assign_op=tf.assign(b, b+1)
out=assign_op*2

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(3):
        print(sess.run([out, b]))
```

- 如果不进行assign_op以后的计算（out），b 会不会变？

变量更新案例3



```
#Assign example
import tensorflow as tf
import numpy as np

var = tf.Variable(0., name='var')
const = tf.constant(1.)
add_op = tf.add(var, const, name='myAdd') # tmp = var + const

#return a tensor, and have a side effect, that is assign add_op to var
assign_op = tf.assign(var, add_op, name='myAssign')

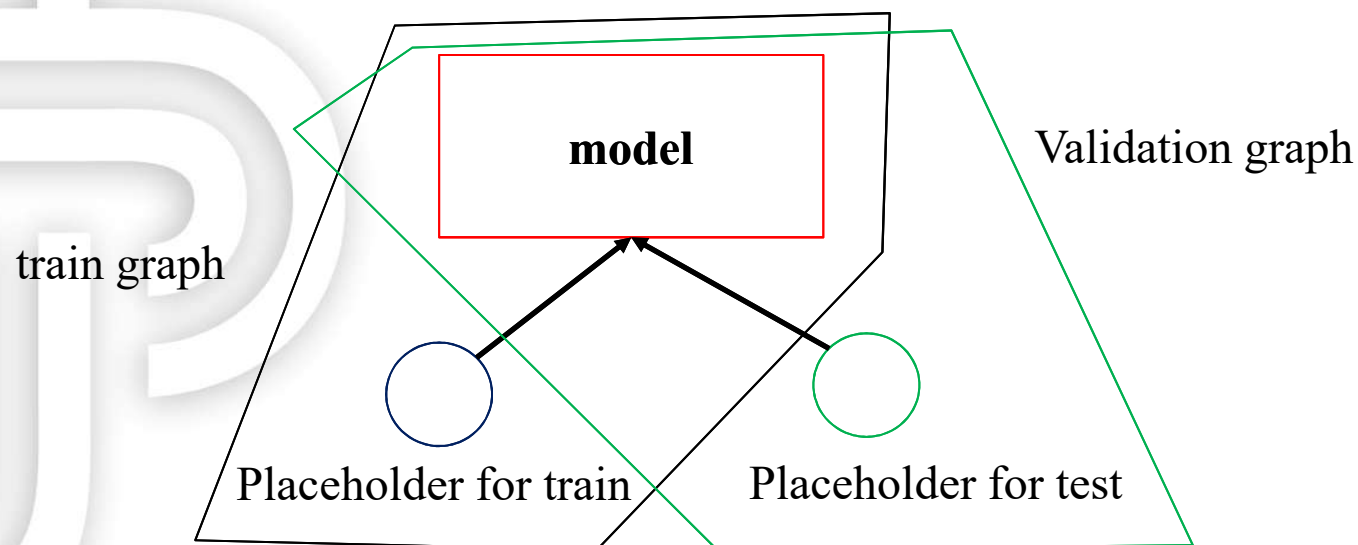
out1 = assign_op*1
out2 = assign_op*2

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(3):
        print "var:", sess.run(var), sess.run(out1), sess.run(out2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(3):
        print "var:", sess.run(var), sess.run([out1, out2])
```

```
var: 0.0 1.0 4.0    var: 2.0 3.0 8.0    var: 4.0 5.0 12.0
var: 0.0 [1.0, 2.0] var: 1.0 [2.0, 4.0]  var: 2.0 [3.0, 6.0]
```


- 为什么要引入命名空间？
- 案例1：某同学想在学习过程中测试模型（model），但训练和测试的输入是不同的（training data、validation data），他想这么做：



- 他大概是这么做的，为了显示出问题，用同一输入X检验效果
- 同样模型，同样输入，结果是否一样？为什么？
- w不是原来的w了

```
import tensorflow as tf
import numpy as np

X=tf.placeholder(tf.float32)

def model(X):
    w=tf.Variable(name="w", initial_value=tf.random_normal(shape=[1]))
    m=tf.multiply(X,w)
    return m

def train_graph(X):
    m=model(X)
    a=tf.add(m,X)

    return a

def test_graph(X):
    m=model(X)
    b=tf.add(m,X)

    return b

a=train_graph(X)
b=test_graph(X)

X_in=1.2

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    ar=sess.run(a, feed_dict={X:X_in})
    br=sess.run(b, feed_dict={X:X_in})
    print("ar=",ar)
    print("br=",br)
```

```
import tensorflow as tf
import numpy as np

def model():
    w=tf.Variable(name="w", initial_value=tf.random_normal(shape=[1]))
    print(w.name)

model()
model()
```

- w:0
- w_1:0
- 重新创建了一个变量w_1，但并不是对w的完全复制！变量如何复用？

- 复杂模型：变量繁多，如何有效组织，避免相同变量名引起的混淆？
 - 1.期望复用，却成为新建
 - 2.不同变量，繁琐写不同名字:w1, w2,...
- 利用Scope来对变量节点进行分组和访问
 - 每个scope可以对应到一个神经层或子模块
- `tf.variable_scope()`提供一个简单的命名空间机制
 - 在一个scope下面可以通过`tf.get_variable()`在当前scope下创建/复用一個variable

- 在tensorflow中所有的变量都是全局变量，Variable scope起到了划分命名空间的作用，给一个Variable名字加一个前缀

```
import tensorflow as tf

with tf.variable_scope('foo'):
    with tf.variable_scope('dummy') as sp:
        v = tf.get_variable('v', shape=[1, 2])
        v2 = tf.get_variable('v2', shape=[1, 1])

print v.name
print v2.name
```

```
foo/dummy/v:0
foo/v2:0
```

变量域与重用

- 可以实用tf.get_variable()创建一个Variable节点
- 当要引用一个variable的时候调用tf.get_variable()可以得到一个Variable节点的引用(把scope中的reuse置True)

```
1 import tensorflow as tf
2
3 with tf.variable_scope("foo"):
4     aaa = tf.get_variable("aaa", [1])
5     bbb = tf.get_variable("bbb", [1])
6
7 with tf.get_variable("foo", reuse=True):
8     ccc = tf.get_variable("aaa")
9
10 print aaa.name
11 print bbb.name
12 print ccc.name
```

foo/aaa:0
foo/bbb:0
foo/aaa:0

- 另外一种Variable引用方式

```
1 import tensorflow as tf
2
3 with tf.variable_scope("foo") as scope:
4     aaa = tf.get_variable("aaa", [1])
5     bbb = tf.get_variable("bbb", [1])
6     scope.reuse_variables()
7     ccc = tf.get_variable("aaa")
8
9 print aaa.name
10 print bbb.name
11 print ccc.name
```

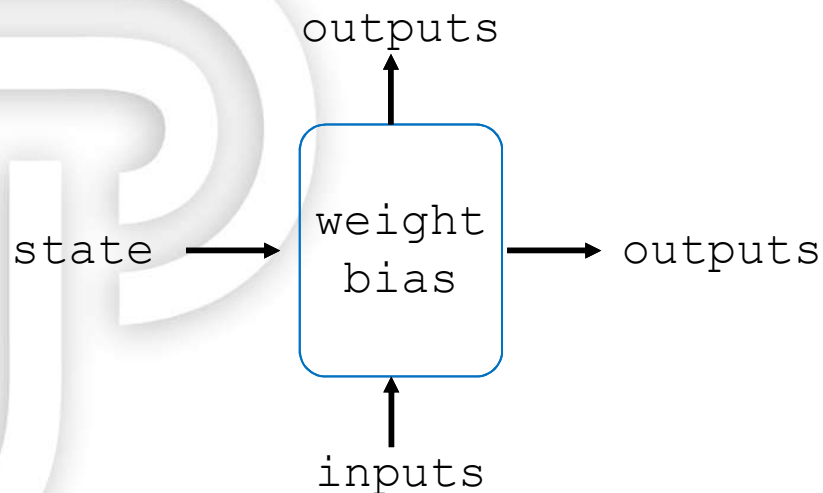
foo/aaa:0

foo/bbb:0

foo/aaa:0

变量域与重用

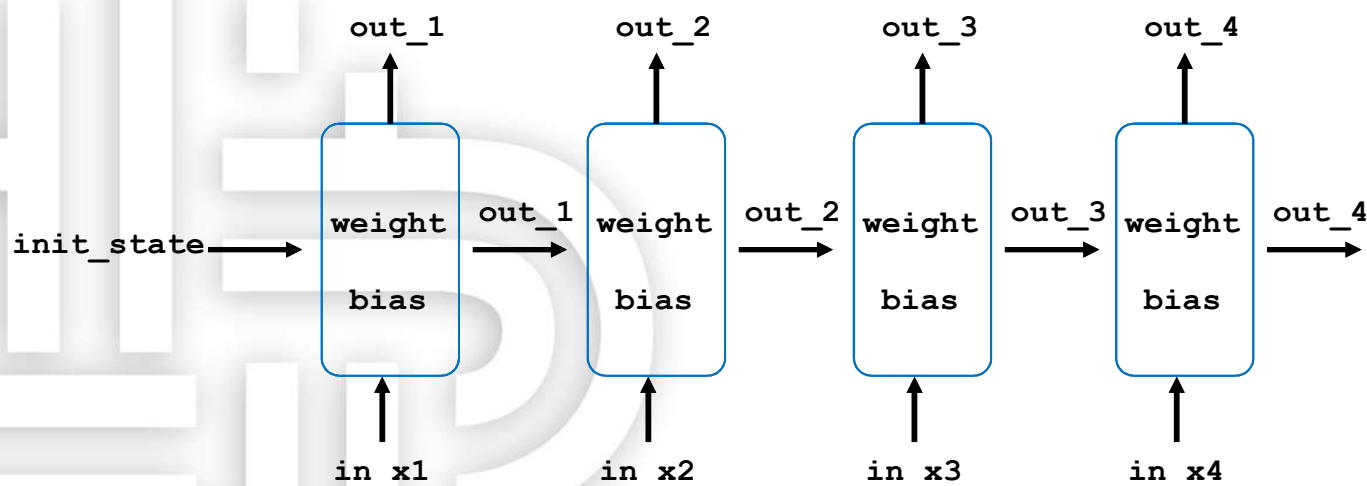
```
1 def rnn(inputs, state, hidden_size):
2     in_x = tf.concat([inputs, state], axis=1)
3     W_shape = [int(in_x.get_shape()[1]), hidden_size]
4     b_shape = [1, hidden_size]
5
6     W = tf.get_variable(shape=W_shape, name='weight')
7     b = tf.get_variable(shape=b_shape, name='bias')
8
9     out_linear = tf.nn.bias_add(tf.matmul(in_x, W), b)
10    output = tf.nn.tanh(out_linear)
11    return output
```



变量域与重用

```
out_1 = rnn(in_x1, init_state, 64)  
out_2 = rnn(in_x2, out_1, 64)  
out_3 = rnn(in_x3, out_2, 64)  
out_4 = rnn(in_x4, out_3, 64)
```

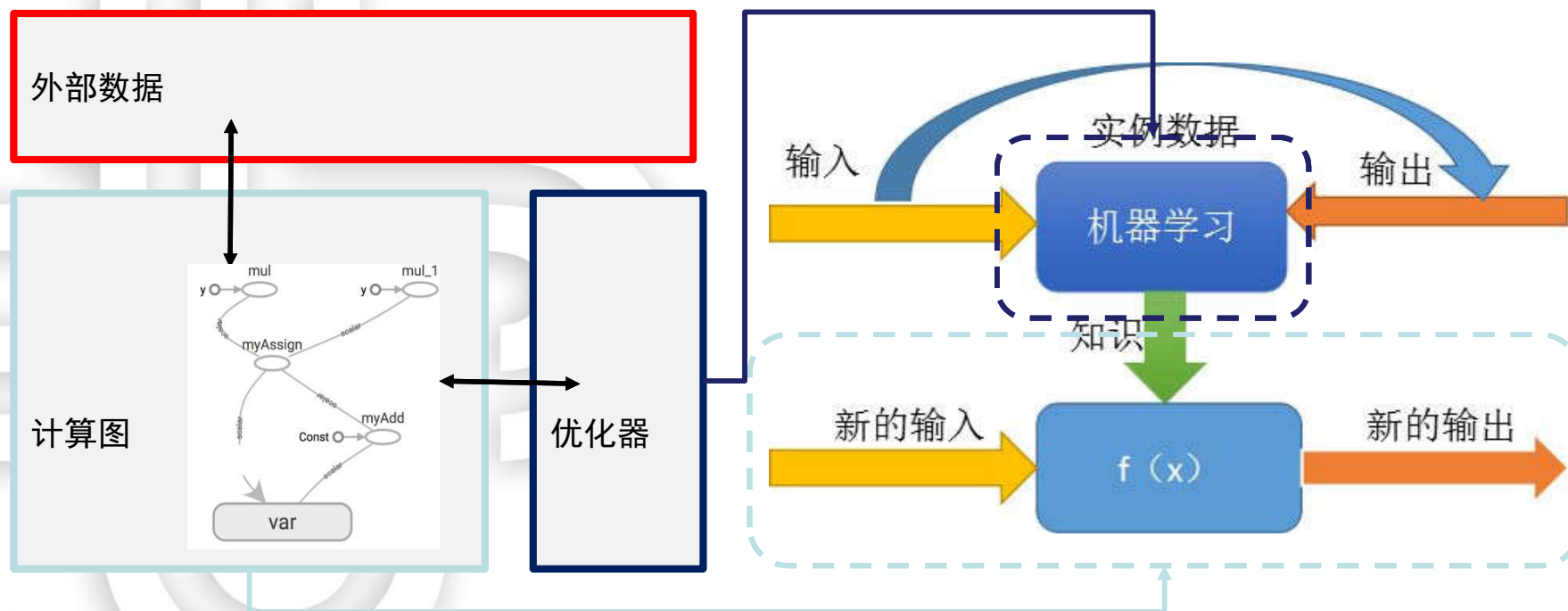
```
with tf.variable_scope('rnn_scope') as scope:  
    out_1 = rnn(in_x1, init_state, 64)  
    scope.reuse_variables()  
    out_2 = rnn(in_x2, out_1, 64)  
    out_3 = rnn(in_x3, out_2, 64)  
    out_4 = rnn(in_x4, out_3, 64)
```



机器学习编程框架

Tensorflow 编程框架和机器学习模型对应关系

- 输入、输出、模型计算过程用计算图Graph描述，并用优化器和训练数据对模型参数进行优化



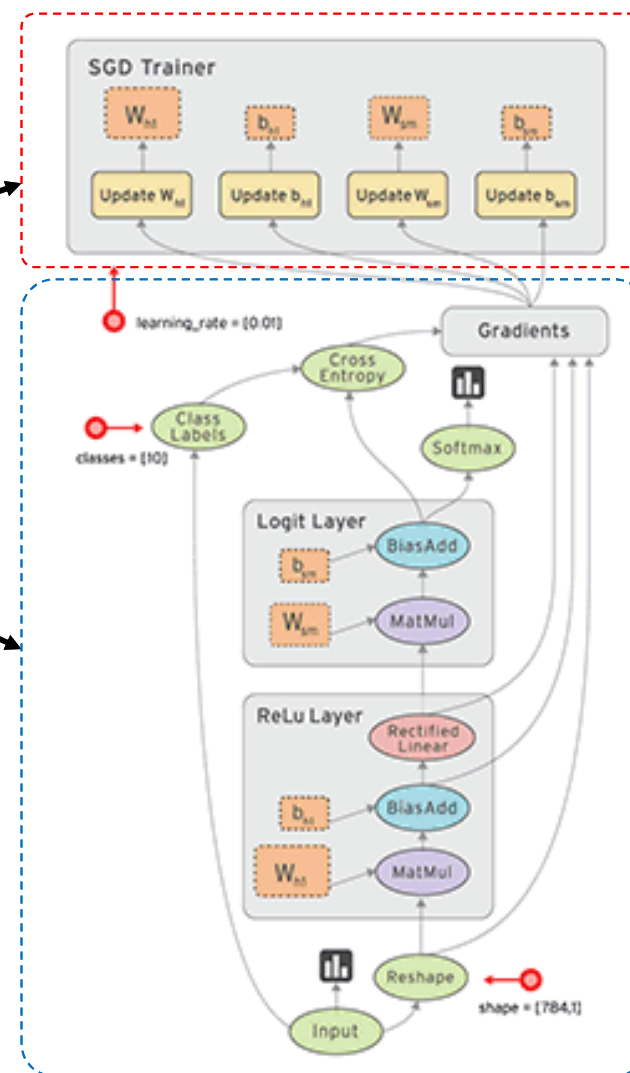
- 机器学习

- 模型

- Graph=计算路径+参数变量

- 优化

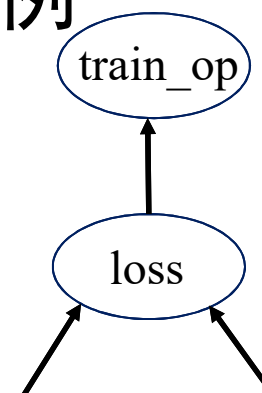
- 对象：参数变量
 - 目标：损失函数（最小）
 - 方法：梯度下降等



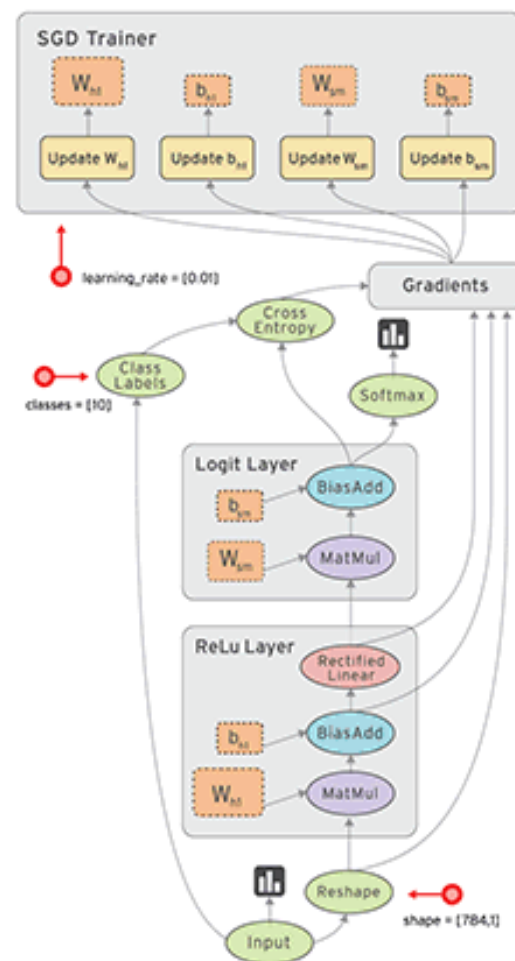
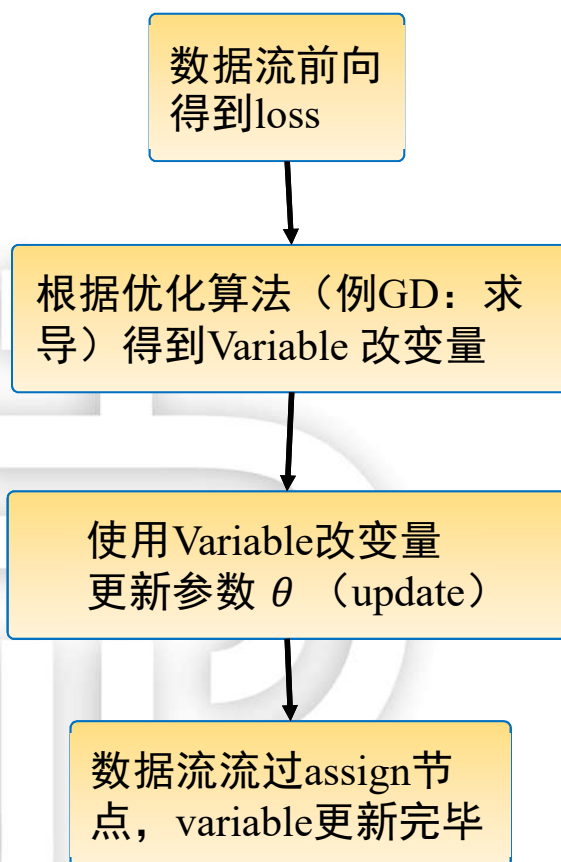
- 1. 定义目标函数（例：损失函数loss，模型预测与真值差距）
- 2. 定义一个优化器，并制定优化目标（例loss最小）

```
opt = tf.train.GradientDescentOptimizer(0.01)  
train_op = opt.minimize(loss)
```

- 3. feed训练数据的同时，在session.run
- 中对优化器tensor:train_op进行fetch操作：
 - sess.run([train_op...],feed_dict={input_x:xxx, label:xx})



TF优化机制



- Graph建图：
 - 1. 创建数据，定义输入结点
 - 2. 定义模型主要部分计算图（参数变量，计算路径）
 - 3. 定义损失函数
 - 4. 定义优化器及优化目标
- Session执行：
 - 5. 初始化参数
 - 6. 定义（迭代）训练脚本并执行（fetch: train_op, feed: input_data, input_label）

例子：逻辑斯蒂回归

- 数据

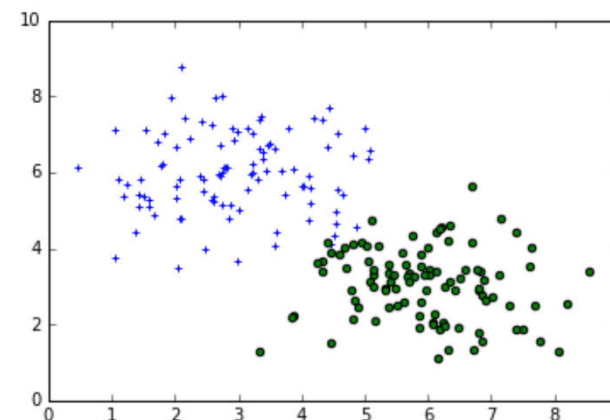
- 两组数据(分别在图中用 ‘+’ 以及 ‘o’ 表示)
分别采样于两个二维高斯分布

$$(x, y) \sim N(3, 6, 1, 1, 0)$$

$$(x, y) \sim N(6, 3, 1, 1, 0)$$

- 目标

- 将两组来自不同分布的点用一条直线分开



逻辑斯蒂回归：建图

```
x = tf.placeholder(tf.float32, shape=(None, 2))  
y = tf.placeholder(tf.float32, shape=(None, 1))
```

定义占位节点，数据入口

```
with tf.variable_scope("Logistic_regression"):
```

```
W = tf.Variable(np.random.rand(2, 1), 'weight', dtype=tf.float32)  
b = tf.Variable(np.random.rand(1, 1), 'bias', dtype=tf.float32)
```

定义参数节点

```
logits = tf.matmul(X, W) + b  
pred = tf.sigmoid(logits)
```

```
loss = tf.nn.sigmoid_cross_entropy_with_logits(logits, y)  
loss = tf.reduce_mean(loss)
```

定义目标函数loss

```
opt = tf.train.AdamOptimizer(0.01)  
train_op = opt.minimize(loss)
```

定义优化器及优化目标（使得loss变小）

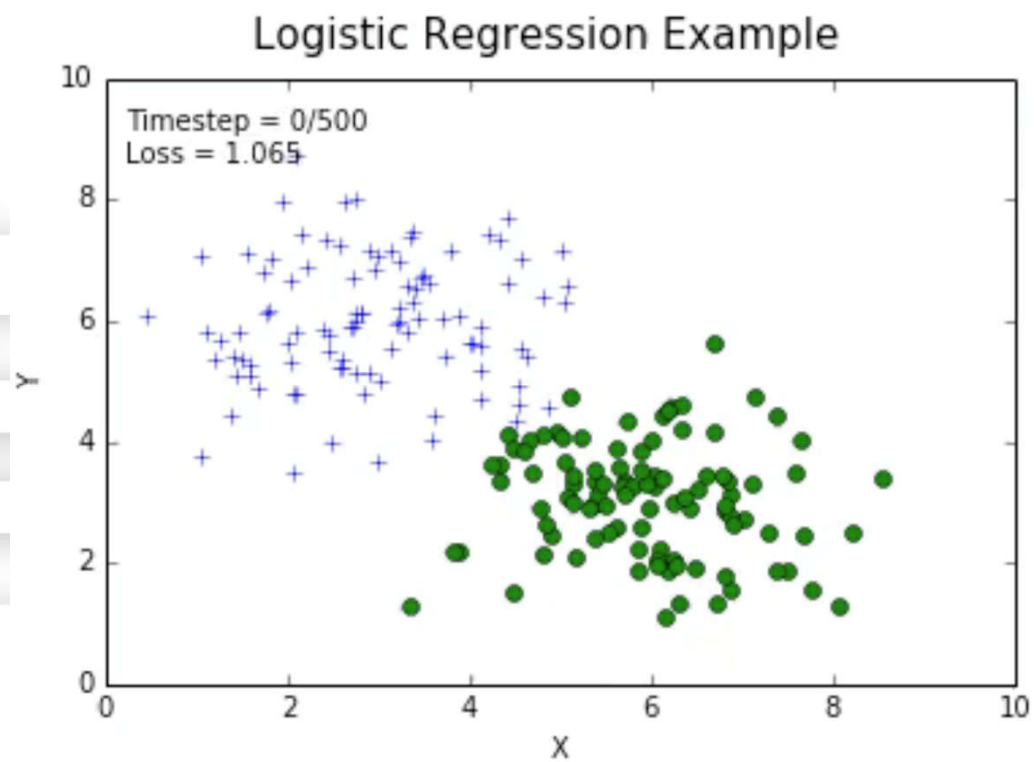
实例化图以及运行图

参数初始化

```
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    for _ in xrange(500):
        _, loss_val = sess.run([train_op, loss],
                                feed_dict={X: data_set[:, 2],
                                             y: data_set[:, 2].reshape((-1, 1))})
```

- 实例化并且feed&fetch实例图 定义并执行训练脚本
 - 实例化：将一副静态的图实例化成一个可运行实体（类似于创建一个进程）
 - feed：将外部数据喂给占位节点(placeholder)
 - fetch：例子中train_op以及loss都是fetch对象，fetch train_op目的是让loss得到优化。fetch loss是为了得到当前的loss值

可视化结果







模型的存储和恢复

- 存什么？
 1. Graph结构
 2. 变量值
- 怎么存？
 - 主要两种模式：
 1. ckpt模式
 2. pb模式

- 保存内容：
- a) Meta graph: .meta 文件
 - protocol buffer保存graph.例如variables, operations, collections等
- b) Checkpoint file: .ckpt 文件
 - 2个二进制文件：包含所有的weights, biases, gradients和其他variables的值。
 - mymodel.data-00000-of-00001 训练的变量值
 - mymodel.index

- c) ‘checkpoint’ 简单保存最近一次保存checkpoint文件的记录
- 例：文本打开查看：

 checkpoint	2018/5/1 16:19	文件	1 KB
 graph.chkp.data-00000-of-00001	2018/5/1 16:19	DATA-00000-OF...	1 KB
 graph.chkp.index	2018/5/1 16:19	INDEX 文件	1 KB
 graph.chkp.meta	2018/5/1 16:19	META 文件	5 KB

- 1. 模型存储:
- `Saver=tf.train.Saver(max_to_keep=4,keep_checkpoint_every_n_hours=2)`
- `Saver.save(sess, ckpt_file_path, global_step)`
- 2. 模型恢复:
- `saver.restore(sess,tf.train.latest_checkpoint('./ckpt'))`

- 基本存储:

```
import tensorflow as tf
w1 = tf.Variable(tf.random_normal(shape=[2]), name='w1')
w2 = tf.Variable(tf.random_normal(shape=[5]), name='w2')
saver = tf.train.Saver()
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver.save(sess, 'my_test_model')
```

- 每1000个iteration保存一次model, 使用save方法给传递一个步长:

```
saver.save(sess, 'my_test_model', global_step=1000)
```

```
my_test_model-1000.index
```

```
my_test_model-1000.meta
```

```
my_test_model-1000.data-00000-of-00001
```

```
checkpoint
```

- 每隔step个iteration保存一次model, 但不必每次都把.meta文件存一次 (graph没变) :

```
saver.save(sess, 'my-model', global_step=step, write_meta_graph=False)
```

- 每训练2小时保存一次, 且只想保存最后4个model:

```
#saves a model every 2 hours and maximum 4 latest models are saved.  
saver = tf.train.Saver(max_to_keep=4, keep_checkpoint_every_n_hours=2)
```

- `tf.train.Saver()`默认保存所有图和变量，如果只想保存部分变量：

```
import tensorflow as tf
w1 = tf.Variable(tf.random_normal(shape=[2]), name='w1')
w2 = tf.Variable(tf.random_normal(shape=[5]), name='w2')
saver = tf.train.Saver([w1,w2])
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver.save(sess, 'my_test_model', global_step=1000)
```


- 可否保存部分图？

```
checkpoint_dir = "mysaver"

# first creat a simple graph
graph = tf.Graph()

#define a simple graph
with graph.as_default():
    x = tf.placeholder(tf.float32,shape=[],name='input')
    y = tf.Variable(initial_value=0,dtype=tf.float32,name="y_variable")
    update_y = y.assign(x)
    saver = tf.train.Saver(max_to_keep=3)
    init_op = tf.global_variables_initializer()

# train the model and save the model every 4000 iterations.
sess = tf.Session(graph=graph)
sess.run(init_op)
for i in range(1,10000):
    y_result = sess.run(update_y,feed_dict={x:i})
    if i %4000 == 0:
        saver.save(sess,checkpoint_dir,global_step=i)
```

- 主要两种方式：
 1. 重复定义计算图为默认图，用 `tf.train.Saver()` 恢复默认图
 2. 指定 `.meta` 文件中的计算图为所需恢复图，用该图的 `Saver()` 恢复
- 获取图中张量：`get_tensor_by_name("name")`
- 需要记住图中张量名字

- 1. 恢复默认图，需有原模型计算图定义，
(适用场合：同一文件中包含训练、测试)

```
import tensorflow as tf

# 声明两个变量
v1 = tf.Variable(tf.random_normal([1, 2]), name="v1")
v2 = tf.Variable(tf.random_normal([2, 3]), name="v2")
init_op = tf.global_variables_initializer() # 初始化全部变量
saver = tf.train.Saver() # 声明tf.train.Saver类用于保存模型
with tf.Session() as sess:
    sess.run(init_op)
    print("v1:", sess.run(v1)) # 打印v1、v2的值一会读取之后对比
    print("v2:", sess.run(v2))
    saver_path = saver.save(sess, "save/model.ckpt") # 将模型保存到save/model.ckpt文件
    print("Model saved in file:", saver_path)
```

- 1. 恢复默认图，需有原模型计算图定义

```
import tensorflow as tf

# 使用和保存模型代码中一样的方式来声明变量
v1 = tf.Variable(tf.random_normal([1, 2]), name="v1")
v2 = tf.Variable(tf.random_normal([2, 3]), name="v2")
saver = tf.train.Saver() # 声明tf.train.Saver类用于保存模型
with tf.Session() as sess:
    saver.restore(sess, "save/model.ckpt") # 即将固化到硬盘中的Session从保存路径再读取出来
    print("v1:", sess.run(v1)) # 打印v1、v2的值和之前的进行对比
    print("v2:", sess.run(v2))
    print("Model Restored")
```

- 2. 直接从.meta文件加载持久化图并恢复
(不用重新定义原计算图, 适用于测试、训练分离)

```
import tensorflow as tf
# 在下面的代码中, 默认加载了TensorFlow计算图上定义的全部变量
# 直接加载持久化的图
saver = tf.train.import_meta_graph("save/model.ckpt.meta")
with tf.Session() as sess:
    saver.restore(sess, "save/model.ckpt")
    # 通过张量的名称来获取张量
    print(sess.run(tf.get_default_graph().get_tensor_by_name("v1:0")))
```

- 2. 直接从.meta文件加载持久化图并恢复
（如果在训练、测试在同一文件，需注意meta中计算图恢复后不要和原图混淆（擦除原图、另定义为新图））

```
tf.reset_default_graph()
restore_graph = tf.Graph()
with tf.Session(graph=restore_graph) as restore_sess:
    restore_saver = tf.train.import_meta_graph('mysaver-8000.meta')
    restore_saver.restore(restore_sess,tf.train.latest_checkpoint('./'))
    print(restore_sess.run("y_variable:0"))
```

- 是否可以修改恢复的计算图？

```
def restore_model_ckpt(ckpt_file_path):  
    sess = tf.Session()  
  
    # 《《《 加载模型结构 》》》  
    saver = tf.train.import_meta_graph('./ckpt/model.ckpt.meta')  
    # 只需要指定目录就可以恢复所有变量信息  
    saver.restore(sess, tf.train.latest_checkpoint('./ckpt'))  
  
    # 直接获取保存的变量  
    print(sess.run('b:0'))  
  
    # 获取placeholder变量  
    input_x = sess.graph.get_tensor_by_name('x:0')  
    input_y = sess.graph.get_tensor_by_name('y:0')  
    # 获取需要进行计算的operator  
    op = sess.graph.get_tensor_by_name('op_to_store:0')  
  
    # 加入新的操作  
    add_on_op = tf.multiply(op, 2)  
  
    ret = sess.run(add_on_op, {input_x: 5, input_y: 5})  
    print(ret)
```


- 优点:
- 灵活
- 缺点:
- 依赖TF框架，只能在TF中用
- 是否能在其他语言、框架中用TF训练好的模型？

- PB 文件: MetaGraph 的 protocol buffer格式的文件, 包括计算图, 数据流, 以及相关的变量等
- 可以把多个计算图保存到一个 PB 文件中
- 以计算图的功能和使用设备命名区分多个计算图, 比如 serving or training, CPU or GPU。

pb模式存储



```
import tensorflow as tf
import os
from tensorflow.python.framework import graph_util

pb_file_path = os.getcwd()

with tf.Session(graph=tf.Graph()) as sess:
    x = tf.placeholder(tf.int32, name='x')
    y = tf.placeholder(tf.int32, name='y')
    b = tf.Variable(1, name='b')
    xy = tf.multiply(x, y)
    # 这里的输出需要加上name属性
    op = tf.add(xy, b, name='op_to_store')

    sess.run(tf.global_variables_initializer())

    # convert_variables_to_constants 需要指定output_node_names, list(), 可以多个
    constant_graph = graph_util.convert_variables_to_constants(sess, sess.graph_def, ['op_to_store'])

    # 测试 OP
    feed_dict = {x: 10, y: 3}
    print(sess.run(op, feed_dict))

    # 写入序列化的 PB 文件
    with tf.gfile.GFile(pb_file_path+'model.pb', mode='wb') as f:
        f.write(constant_graph.SerializeToString())
```

- 保存为 save model 格式也可以生成模型的 PB 文件

```
# 官网有误, 写成了 saved_model_builder
builder = tf.saved_model.builder.SavedModelBuilder(pb_file_path+'savemodel')
# 构造模型保存的内容, 指定要保存的 session, 特定的 tag,
# 输入输出信息字典, 额外的信息
builder.add_meta_graph_and_variables(sess,
                                     ['cpu_server_1'])

# 添加第二个 MetaGraphDef
#with tf.Session(graph=tf.Graph()) as sess:
#    ...
#    builder.add_meta_graph([tag_constants.SERVING])
#...

builder.save() # 保存 PB 模型
```

pb模式恢复

```
from tensorflow.python.platform import gfile

sess = tf.Session()
with gfile.GFile(pb_file_path+'model.pb', 'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    sess.graph.as_default()
    tf.import_graph_def(graph_def, name='') # 导入计算图

# 需要有一个初始化的过程
sess.run(tf.global_variables_initializer())

# 需要先复原变量
print(sess.run('b:0'))
# 1

# 输入
input_x = sess.graph.get_tensor_by_name('x:0')
input_y = sess.graph.get_tensor_by_name('y:0')

op = sess.graph.get_tensor_by_name('op_to_store:0')

ret = sess.run(op, feed_dict={input_x: 5, input_y: 5})
print(ret)
# 输出 26
```

- Save_model对应的恢复方式

```
with tf.Session(graph=tf.Graph()) as sess:
    tf.saved_model.loader.load(sess, ['cpu_1'], pb_file_path+'savemodel')
    sess.run(tf.global_variables_initializer())

    input_x = sess.graph.get_tensor_by_name('x:0')
    input_y = sess.graph.get_tensor_by_name('y:0')

    op = sess.graph.get_tensor_by_name('op_to_store:0')

    ret = sess.run(op, feed_dict={input_x: 5, input_y: 5})
    print(ret)
```

只需要指定要恢复模型的 session, 模型的 tag, 模型的保存路径即可, 使用起来更加简单

pb模式也可以在其他语言中调用PDL

```
#include "tensorflow/core/public/session.h"
#include "tensorflow/core/platform/env.h"
// @brief: 从model_path 加载模型，在Session中创建图
// ReadBinaryProto() 函数将model_path的protobuf文件读入一个tensorflow::GraphDef的对象
// session->Create(graphdef) 函数在一个Session下创建了对应的图；

int ANNModelLoader::load(tensorflow::Session* session, const std::string model_path) {
    //Read the pb file into the graphdef member
    tensorflow::Status status_load = ReadBinaryProto(Env::Default(), model_path, &graphdef);
    if (!status_load.ok()) {
        std::cout << "ERROR: Loading model failed..." << model_path << std::endl;
        std::cout << status_load.ToString() << "\n";
        return -1;
    }

    // Add the graph to the session
    tensorflow::Status status_create = session->Create(graphdef);
    if (!status_create.ok()) {
        std::cout << "ERROR: Creating graph in session failed..." << status_create.ToString() << std::endl;
        return -1;
    }
    return 0;
}
```


- 变量命名空间可有效组织复杂模型变量，变量复用需用get_variable复用模式
- 一个基本的TF机器学习编程框架
- 模型存储的两种模式ckpt, pb. ckpt方便灵活，Pb模式适用于模型封装和移植