学习流程：

1. 了解tensorflow基本原理
2. 基本概念

1、定义计算与执行计算

Tensorflow的编程包含两个过程：1、构建计算图；2、运行计算图，这是一种将计算的定义和运行分开的设计结构。

举例说明：我们先定义了两个计算节点node1,和node2

node1 = tf.constant(3.0, tf.float32)

node2 = tf.constant(4.0) # also tf.float32 implicitly

print(node1, node2)

这时候打印出来的节点并没有实际的值，只是一个tensor张量

打印如下：

Tensor("Const:0", shape=(), dtype=float32) Tensor("Const\_1:0", shape=(), dtype=float32)

这时候我们建立一个新的节点node3，它是node1和node2的和，这里的效果更加清晰，node3在这里，并不是一个运算结果，任然是一个节点

node3 = tf.add(node1, node2)

print("node3: ", node3)#打印结果node3: Tensor("Add:0", shape=(), dtype=float32)

接下来使用计算过程，tensorflow的计算需要开启一个sesssion，session实际上是需要配置一个graph的，如果你没有指定graph，那么它会默认采用一个defaultgraph，之前你也没明确指明那些计算是在哪一张graph里面，他就会被默认定义在defaultgraph里面

sess=tf.Session()

print("sess.run(node3): ",sess.run(node3))#打印结果：sess.run(node3): 7.0

当我们调用sess.run(node3)时，tensorflow才会根据图去计算所有的节点

这种定义操作，再执行操作的模式被称之为“符号式编程” Symbolic Programming

1. graph

# 建立一个新的空的图  
graph1 = tf.Graph()  
**with** graph1.as\_default():  
 value1 = tf.constant([1,2])  
 value2 = tf.Variable([2,4])  
 mul = value2/value1  
  
**with** tf.Session(graph = graph1) **as** mysess:  
 tf. global\_variables\_initializer().run()  
 print(mysess.run(mul)) # [2. 2.]  
 print(mul.eval()) # [2. 2.]

这里我们将value1，value2，mul放入了一个自定义的名为graph1的图里面，我们想要建立一个关于该图的session，需要在session建立的时候指明参数graph的属性

1. placeholder占位符

占位符的使用可以帮助我们在运行时，传入相应的参数进行计算

**def user\_placeholder**():  
 graph1 = tf.Graph()  
 **with** graph1.as\_default():  
 value1 = tf.placeholder(dtype=tf.float64)  
 value2 = tf.constant([2,3],dtype=tf.float64)  
 mul = value1\*value2  
 **with** tf.Session(graph=graph1) **as** mysess:  
 tf.global\_variables\_initializer().run()  
 value = load\_from\_remote()  
 print(mysess.run(mul,feed\_dict={value1:value}))  
  
  
**def load\_from\_remote**():  
 **return** [2,3]  
  
**if** \_\_name\_\_ == "\_\_main\_\_":  
 user\_placeholder()

1. tensorflow自带的高阶api

预创建的estimator的使用：

在我们使用tensorflow中预创建的estimator进行学习时，（预创建也就是）我们需要定义几个运算：

1. 创建和准备训练数据集（特征和标签）； 2实例化estimator（需要传入特征集，超参数等）3、使用实例化后的estimator进行训练，4、使用测试数据集对训练后的estimator进行评估

我们用tensorflow教程里面的基本分类器进行说明：

**def main**(argv):  
 args = parser.parse\_args(argv[1:])  
  
 #从远程获取了训练数据集和测试数据集，是数据准备阶段，一般由另一个函数进行处理

#本例中处理远程数据的是另一个文件iris\_data  
 (train\_x, train\_y), (test\_x, test\_y) = iris\_data.load\_data()  
  
 # Feature columns describe how to use the input.

# 建立特征列，特征列是告诉estimator如何使用我们训练时输入的训练数据

# 特征列由tf.feature\_column里面的函数创建  
 my\_feature\_columns = []  
 **for** key **in** train\_x.keys():  
 my\_feature\_columns.append(tf.feature\_column.numeric\_column(key=key))  
 print("==============")  
 print(my\_feature\_columns)  
  
 # Build 2 hidden layer DNN with 10, 10 units respectively.

# 实例化预创建的estimator，并传入对应的超参数  
 classifier = tf.estimator.DNNClassifier(  
 feature\_columns=my\_feature\_columns,  
 # Two hidden layers of 10 nodes each.  
 hidden\_units=[10, 10],  
 # The model must choose between 3 classes.  
 n\_classes=3)  
  
 # Train the Model.

# 将训练数据作为输入，传入到实例化好的estimator，并调用train函数进行训练  
 classifier.train(  
 input\_fn=**lambda**:iris\_data.train\_input\_fn(train\_x, train\_y,  
 args.batch\_size),  
 steps=args.train\_steps)  
  
 # Evaluate the model.

# 我们将测试数据集传入训练好的分类器，调用分类器的evaluate方法进行评估   
 eval\_result = classifier.evaluate(  
 input\_fn=**lambda**:iris\_data.eval\_input\_fn(test\_x, test\_y,  
 args.batch\_size))  
  
 print('\nTest set accuracy: {accuracy:0.3f}\n'.format(\*\*eval\_result))  
  
 # Generate predictions from the model  
 expected = ['Setosa', 'Versicolor', 'Virginica']  
 predict\_x = {  
 'SepalLength': [5.1, 5.9, 6.9],  
 'SepalWidth': [3.3, 3.0, 3.1],  
 'PetalLength': [1.7, 4.2, 5.4],  
 'PetalWidth': [0.5, 1.5, 2.1],  
 }

# 将需要进行预测的特征数据传入训练好的分类器，输出分类结果  
 predictions = classifier.predict(  
 input\_fn=**lambda**:iris\_data.eval\_input\_fn(predict\_x,  
 labels=**None**,  
 batch\_size=args.batch\_size))  
  
 template = ('\nPrediction is "{}" ({:.1f}%), expected "{}"')  
  
 **for** pred\_dict, expec **in** zip(predictions, expected):  
 class\_id = pred\_dict['class\_ids'][0]  
 probability = pred\_dict['probabilities'][class\_id]  
  
 print(template.format(iris\_data.SPECIES[class\_id],  
 100 \* probability, expec))

创建自定义的estimator

预创建的 Estimator 已完全成形。不过有时，我们需要更好地控制 Estimator 的行为。这时，自定义 Estimator 就派上用场了，使用创建自定义 Estimator 来完成几乎任何操作。如果您需要以某种不寻常的方式连接隐藏层，则可以编写自定义 Estimator。如果您需要为模型计算独特的[指标](https://developers.google.com/machine-learning/glossary/#metric)，也可以编写自定义 Estimator。基本而言，如果我们需要一个针对具体问题进行了优化的 Estimator，就可以编写自定义 Estimator。

模型函数（即 model\_fn）会实现机器学习算法

* 如果采用预创建的 Estimator，则有人已为您编写了模型函数。
* 如果采用自定义 Estimator，则您必须自行编写模型函数

所有模型函数都必须接受一组标准输入参数并返回一组标准输出值

自定义一个estimator需要如下几个元素：

1. 编写输入函数
2. 创建特征列
3. 编写模型函数
4. 定义模型
5. 实现训练、评估和预测

编写输入函数

我们的自定义 Estimator 实现与我们的预创建的 Estimator 实现使用的是同一输入函数（来自 iris\_data.py）。即：

def train\_input\_fn(features, labels, batch\_size):

"""An input function for training"""

# Convert the inputs to a Dataset.

dataset = tf.data.Dataset.from\_tensor\_slices((dict(features), labels))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(1000).repeat().batch(batch\_size)

# Return the read end of the pipeline.

return dataset.make\_one\_shot\_iterator().get\_next()

此输入函数会构建可以生成批次 (features, labels) 对的输入管道，其中 features 是字典特征。

创建特征列

按照预创建的 Estimator 和特征列章节中详细介绍的内容，您必须定义模型的特征列来指定模型应该如何使用每个特征。无论是使用预创建的 Estimator 还是自定义 Estimator，您都要使用相同的方式定义特征列。

以下代码为每个输入特征创建一个简单的 numeric\_column，表示应该将输入特征的值直接用作模型的输入：

# Feature columns describe how to use the input.

my\_feature\_columns = []

for key in train\_x.keys():

my\_feature\_columns.append(tf.feature\_column.numeric\_column(key=key))

编写模型函数

我们要使用的模型函数具有以下调用签名：

def my\_model\_fn(

features, # This is batch\_features from input\_fn

labels, # This is batch\_labels from input\_fn

mode, # An instance of tf.estimator.ModeKeys

params): # Additional configuration

前两个参数是从输入函数中返回的特征和标签批次；也就是说，features 和 labels 是模型将使用的数据的句柄。mode 参数表示调用程序是请求训练、预测还是评估。

调用程序可以将 params 传递给 Estimator 的构造函数。传递给构造函数的所有 params 转而又传递给 model\_fn。在 custom\_estimator.py 中，以下行会创建 Estimator 并设置参数来配置模型。此配置步骤与我们在 TensorFlow 使用入门中配置 tf.estimator.DNNClassifier 的方法类似。

classifier = tf.estimator.Estimator(

model\_fn=my\_model,

params={

'feature\_columns': my\_feature\_columns,

# Two hidden layers of 10 nodes each.

'hidden\_units': [10, 10],

# The model must choose between 3 classes.

'n\_classes': 3,

})

要实现一般的模型函数，您必须执行下列操作：

定义模型。

分别为三种不同模式指定其他计算：

预测

评估

训练

定义输入层

model\_fn 的第一行调用 tf.feature\_column.input\_layer，以将特征字典和 feature\_columns 转换为模型的输入，如下所示：

# Use `input\_layer` to apply the feature columns.

net = tf.feature\_column.input\_layer(features, params['feature\_columns'])

四、使用tensorflow自带的低阶api

低阶api用于在 Estimator 框架之外使用 TensorFlow 进行调试和实验

4、1张量

TensorFlow 中的核心数据单位是张量。一个张量由一组形成阵列（任意维数）的原始值组成。张量的阶是它的维数，而它的形状是一个整数元组，指定了阵列每个维度的长度。以下是张量值的一些示例：

3. # 0阶张量，一个标量，shape是[]

[1., 2., 3.] # 1阶张量，一个向量，shape是[3]

[[1., 2., 3.], [4., 5., 6.]] # 2阶张量，一个矩阵，shape是[2,3]

[[[1., 2., 3.]], [[7., 8., 9.]]] # 3阶张量，shape是[2,1,3]

TensorFlow 使用 numpy 阵列来表示张量值。

图：

您可以将 TensorFlow Core 程序看作由两个互相独立的部分组成：

构建计算图 (tf.Graph)。

运行计算图 (tf.Session)。

计算图是排列成一个图的一系列 TensorFlow 指令。图由两种类型的对象组成。

指令（或“op"）：图的节点。 指令说明的是消耗和生成张量的计算。

张量：图的边。它们代表将流经图的值。大多数 TensorFlow 函数会返回 tf.Tensors。

重要提示：tf.Tensors 不具有值，它们只是计算图中元素的手柄。

我们使用一个例子来说明计算图的过程：

a = tf.constant(3.0, dtype=tf.float32)

b = tf.constant(4.0) # also tf.float32 implicitly

total = a + b

print(a)

print(b)

print(total)

这里构建了三个节点，a,b,total

打印结果是：

Tensor("Const:0", shape=(), dtype=float32)

Tensor("Const\_1:0", shape=(), dtype=float32)

Tensor("add:0", shape=(), dtype=float32)

显然，这里打印的结果并不是我们预期的3.0,4.0,7.0，而是代表将要运行的指令

图中的每个指令都拥有唯一的名称。这个名称不同于使用 Python 分配给相应对象的名称。张量是根据生成它们的指令命名的，后面跟着输出索引，如上文的 "add:0" 所示。

但是我们可以在定义变量的时候给它名称，比如

b = tf.constant(4.0,name=”b”)

这是由打印出来的b的名字就是”b:0”

我们可以将多个张量传递给 tf.Session.run。run 方法以透明方式处理元组或字典的任何组合，如下例所示：

print(sess.run({'ab':(a, b), 'total':total}))

它返回的结果拥有相同的布局结构：

{'total': 7.0, 'ab': (3.0, 4.0)}

在调用 tf.Session.run 期间，任何 tf.Tensor 都只有单个值，这里的意思是，哪怕某个变量是随机生成的，那么在一个session期间，这个随机值是不会改变的