tensorflow学习

赵海臣

聚美优品
JUMEI.COM

基本使用

- TensorFlow基本特点:
  - 使用图 (graph) 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(arbitrary operation) 赋值或者从其中获取数据.

聚美优品
JUMEI.COM

TensorFlow Graph

- TensorFlow 是一个编程系统, 使用图来表示计算任务.
  - 图中的节点被称之为 op (operation 的缩写).
  - 一个 op 获得 0 个或多个 Tensor, 执行计算, 产生 0 个或多个 Tensor.
  - 每个 Tensor 是一个类型化的多维数组.
- 一个 TensorFlow 图描述了计算的过程. 为了进行计算, 图必须在 会话 里被启动.
  - 会话 将图的 op 分发到诸如 CPU 或 GPU 之类的 设备 上, 同时提供执行 op 的方法.
  - 这些方法执行后, 将产生的 tensor 返回.
  - 在 Python 语言中, 返回的 tensor 是 numpy ndarray 对象

聚美优品
JUMEI.COM

Constuct Graph

- 构建图的第一步, 是创建源 op (source op).
  - 源 op 不需要任何输入, 例如 常量 (Constant).
  - 源 op 的输出被传递给其它 op 做运算.
  - op 构造器的返回值代表被构造出的 op 的输出, 这些返回值可以传递给其它 op 构造器作为输入.

```
import tensorflow as tf
# 构造器的返回值代表该常量 op 的返回值.
matrix1 = tf.constant([[3., 3.]])
# 创建另外一个常量 op, 产生一个 2x1 矩阵.
matrix2 = tf.constant([[2.],[2.]])
# 创建一个矩阵乘法 matmul op , 把 'matrix1' 和 'matrix2' 作为输入.
# 返回值 'product' 代表矩阵乘法的结果.
product = tf.matmul(matrix1, matrix2)
```

- 默认图现在有三个节点, 两个 constant() op, 和一个matmul() op. 为了真正进行矩阵相乘运算, 并得到矩阵乘法的 结果, 你必须在会话里启动这个图.

聚美优品
JUMEI.COM

在一个会话中启动图

- 构造阶段完成后, 才能启动图. 启动图的第一步是创建一个 Session 对象, 如果无任何创建参数, 会话构造器将启动默认图.
  - Session 对象在使用完后需要关闭以释放资源. 除了显式调用 close 外, 也可以使用 "with" 代码块 来自动完成关闭动作.
    - with tf.Session() as sess:
    - result = sess.run([product])
    - print result

聚美优品
JUMEI.COM

并行运算

在实现上, TensorFlow 将图形定义转换成分布式执行的操作, 以充分利用可用的计算资源(如 CPU 或 GPU).

- 一般你不需要显式指定使用 CPU 还是 GPU, TensorFlow 能自动检测.
- 如果检测到 GPU, TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作.
- 如果机器上有超过一个可用的 GPU, 除第一个外的其它 GPU 默认是不参与计算的. 为了让 TensorFlow 使用这些 GPU, 你必须将 op 明确指派给它们执行.

聚美优品
JUMEI.COM

交互式使用

为了便于使用诸如 IPython 之类的 Python 交互环境, 可以使用 InteractiveSession 代替 Session 类, 使用 Tensor.eval() 和 Operation.run() 方法代替 Session.run(). 这样可以避免使用一个变量来持有会话.

```
# 进入一个交互式 TensorFlow 会话.
import tensorflow as tf
sess = tf.InteractiveSession()
x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])
# 使用初始化器 initializer op 的 run() 方法初始化 'x'
x.initializer.run()
# 增加一个减法 sub op, 从 'x' 减去 'a'. 运行减法 op, 输出结果
sub = tf.sub(x, a)
print sub.eval()
# ==> [-2. -1.]
```

聚美优品
JUMEI.COM

Tensor

- TensorFlow 程序使用 tensor 数据结构来代表所有的数据
  - 计算图中, 操作间传递的数据都是 tensor.
  - 你可以把 TensorFlow tensor 看作是一个 n 维的数组或列表.
  - 一个 tensor 包含一个静态类型 rank, 和 一个 shape.

聚美优品
JUMEI.COM

变量

- 理解变量与run
  - # 变量op与常数op
  - state = tf.Variable(0, name="counter")
  - one = tf.constant(1)
  - # 加法op
  - new_value = tf.add(state, one)
  - update = tf.assign(state, new_value) # state+1
  - # 加法op
  - new_value2 = tf.add(update, one)
  - update2 = tf.assign(state, new_value2) # state+1
  - # 加法op
  - new_value3 = tf.add(update2, one)
  - update3 = tf.assign(state, new_value3) # state+1
  - init_op = tf.initialize_all_variables()
  - sess = tf.InteractiveSession()
  - sess.run(init_op)

聚美优品
JUMEI.COM

变量

- 运行结果
  - 每次run，都是graph执行到对应的op的结果，在对应op前的Variable迭代保存步骤都会生效并保存。
  - In [183]: sess.run(new_value)
  - Out[183]: 1
  - In [184]: sess.run(new_value)
  - Out[184]: 1
  - In [185]: sess.run(update)
  - Out[185]: 1
  - In [186]: sess.run(update)
  - Out[186]: 2
  - In [187]: sess.run(new_value2)
  - Out[187]: 4
  - In [188]: sess.run(new_value2)
  - Out[188]: 5
  - In [189]: sess.run(update2)
  - Out[189]: 6
  - In [190]: sess.run(update2)
  - Out[190]: 8
  - In [191]: sess.run(new_value3)
  - Out[191]: 11
  - In [192]: sess.run(new_value3)
  - Out[192]: 13
  - In [193]: sess.run(update3)
  - Out[193]: 15
  - In [194]: sess.run(update3)
  - Out[194]: 18

聚美优品
JUMEI.COM

变量运行分析

- tensorflow的执行顺序理解要基于graph来看：
  - 每一次运行sess.run(op)都是运行到指定op并停止的结果，若之前的op中有数据保存或者增加，则将执行，而指定op之后的步骤不会进行。
  - sess.run([op1,op2,op3])，将执行到[op1,op2,op3]中最深的那个，并停止，然后将op1,op2,op3的保存变量结果输出。

聚美优品
JUMEI.COM

变量feed

计算图中的 tensor, 以常量或变量的形式存储. TensorFlow 还提供了 feed 机制, 该机制可以临时替代图中的任意操作插入一个 tensor.

- feed 使用一个 tensor 值临时替换一个操作的输出结果. 你可以提供 feed 数据作为 run() 调用的参数.
- feed 只在调用它的方法内有效, 方法结束, feed 就会消失. 最常见的用例是将某些特殊的操作指定为 "feed" 操作, 标记的方法是使用 tf.placeholder() 为这些操作创建占位符.
  - input1 = tf.placeholder(tf.types.float32)
  - input2 = tf.placeholder(tf.types.float32)
  - output = tf.mul(input1, input2)
  - with tf.Session() as sess:
  - print sess.run([output], feed_dict={input1:[7.], input2:[2.]})
  - # 输出:
  - # [array([14.], dtype=float32)]

聚美优品
JUMEI.COM

变量:创建、初始化、保存和加载

- 变量是用来存储和更新参数。
  - 变量包含张量 (Tensor)存放于内存的缓存区。
  - 建模时它们需要被明确地初始化，模型训练后它们必须被存储到磁盘。
  - 这些变量的值可在之后模型训练和分析是被加载。
- 创建变量
  - 当创建一个变量时，你将一个张量作为初始值传入构造函数Variable()
  - 所有这些操作符都需要你指定张量的shape，那个形状自动成为变量的shape
  - 变量的shape通常是固定的
    - # Create two variables.
    - weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")
    - biases = tf.Variable(tf.zeros([200]), name="biases")

聚美优品
JUMEI.COM

变量:创建、初始化、保存和加载

- 变量初始化
  - 变量的初始化必须在模型的其它操作运行之前先明确地完成。
  - 在完全构建好模型并加载之后使用tf.initialize_all_variables()对变量做初始化。。
    - # Create two variables.
    - weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")
    - biases = tf.Variable(tf.zeros([200]), name="biases")
    - # Add an op to initialize the variables.
    - init_op = tf.initialize_all_variables()
    - # Later, when launching the model
    - with tf.Session() as sess:
    - # Run the init operation.
    - sess.run(init_op)
    - ...
    - # Use the model
    - ...

聚美优品
JUMEI.COM

变量:创建、初始化、保存和加载

- 变量保存和加载
  - 保存变量
    - 用tf.train.Saver()创建一个Saver来管理模型中的所有变量。
  - 检查点文件
    - 变量存储在二进制文件里，主要包含从变量名到tensor值的映射关系。
  - 恢复变量
    - 用同一个Saver对象来恢复变量。注意，当你从文件中恢复变量时，不需要事先对它们做初始化。

聚美优品
JUMEI.COM

数据读取

- TensorFlow程序读取数据一共有3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)。

聚美优品
JUMEI.COM

数据读取

虽然你可以使用常量和变量来替换任何一个张量， 但是最好的做法应该是使用placeholder op节点。

- 设计placeholder节点的唯一的意图就是为了提供数据供给(feeding)的方法。
- placeholder节点被声明的时候是未初始化的， 也不包含数据， 如果没有为它供给数据， 则TensorFlow运算的时候会产生错误

聚美优品
JUMEI.COM

从文件读取数据

- 一共典型的文件读取管线会包含下面这些步骤：
  - 文件名列表
  - 可配置的 文件名乱序(shuffling)
  - 可配置的 最大训练迭代数(epoch limit)
  - 文件名队列
  - 针对输入文件格式的阅读器
  - 纪录解析器
  - 可配置的预处理器
  - 样本队列
- 顺序
  - filename_queue = tf.train.string_input_producer(
  - filenames, num_epochs=num_epochs, shuffle=True)
  - reader = tf.SomeReader()
  - key, record_string = reader.read(filename_queue)
  - example, label = tf.some_decoder(record_string)
  - processed_example = some_processing(example)

聚美优品
JUMEI.COM

从文件读取数据

- 文件名, 乱序(shuffling), 和最大训练迭代数(epoch limits)
  - 可以使用字符串张量(比如["file0", "file1"], [("file%d" % i) for i in range(2)]， [("file%d" % i) for i in range(2)])来产生文件名列表。
  - 将文件名列表交给tf.train.string_input_producer 函数. string_input_producer来生成一个先入先出的队列， 文件阅读器会需要它来读取数据。
  - string_input_producer 提供的可配置参数来设置文件名乱序和最大的训练迭代数， QueueRunner会为每次迭代(epoch)将所有的文件名加入文件名队列中， 如果shuffle=True的话， 会对文件名进行乱序处理。
  - 这个QueueRunner的工作线程是独立于文件阅读器的线程，因此乱序和将文件名推入到文件名队列这些过程不会阻塞文件阅读器运行。
- 文件格式
  - 根据你的文件格式， 选择对应的文件阅读器， 然后将文件名队列提供给阅读器的read方法。
  - 阅读器的read方法会输出一个key来表征输入的文件和其中的纪录(对于调试非常有用)，同时得到一个字符串标量， 这个字符串标量可以被一个或多个解析器，或者转换操作将其解码为张量并且构造成为样本。

聚美优品
JUMEI.COM

读取CSV文件

每次read的执行都会从文件中读取一行内容， decode_csv 操作会解析这一行内容并将其转为张量列表。

```
filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])
reader = tf.TextLineReader()
# 每次read的执行都会从文件中读取一行内容
key, value = reader.read(filename_queue)
# Default values, in case of empty columns.
record_defaults = [[1], [1], [1], [1], [1]]
# decode_csv 操作会解析这一行内容并将其转为张量列表。
col1, col2, col3, col4, col5 = tf.decode_csv(
    value, record_defaults=record_defaults)
features = tf.concat(0, [col1, col2, col3, col4])
with tf.Session() as sess:
  # Start populating the filename queue.
  coord = tf.train.Coordinator()
  #在调用run执行read之前， 你必须调用tf.train.start_queue_runners来将文件名填充到队列。否则read操作会被阻塞到文件名队列中有值为止。
  threads = tf.train.start_queue_runners(coord=coord)
  for i in range(1200):
    # Retrieve a single instance:
    example, label = sess.run([features, col5])
  coord.request_stop()
  coord.join(threads)
```

聚美优品
JUMEI.COM

预处理

- 你可以对输入的样本进行任意的预处理， 例如数据归一化， 提取随机数据片，增加噪声或失真等等预处理。

聚美优品
JUMEI.COM

批处理

- 将文件流变成batch

```
def read_my_file_format(filename_queue):
  reader = tf.SomeReader()
  key, record_string = reader.read(filename_queue)
  example, label = tf.some_decoder(record_string)
  processed_example = some_processing(example)
  return processed_example, label
def input_pipeline(filenames, batch_size, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example, label = read_my_file_format(filename_queue)
  # min_after_dequeue defines how big a buffer we will randomly sample
  #   from -- bigger means better shuffling but slower start up and more
  #   memory used.
  # capacity must be larger than min_after_dequeue and the amount larger
  #   determines the maximum we will prefetch.  Recommendation:
  #   min_after_dequeue + (num_threads + a small safety margin) * batch_size
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch(
      [example, label], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch, label_batch
```

聚美优品
JUMEI.COM

批处理——并行读取文件

并行读取数据，虽然只使用了一个文件名队列， 但是TensorFlow依然能保证多个文件阅读器从同一次迭代(epoch)的不同文件中读取数据，知道这次迭代的所有文件都被开始读取为止。

```
def read_my_file_format(filename_queue):
  # Same as above
def input_pipeline(filenames, batch_size, read_threads,
num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example_list = [read_my_file_format(filename_queue)
                  for _ in range(read_threads)]
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch_join(
      example_list, batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch, label_batch
```

聚美优品
JUMEI.COM

批处理——并行读取文件

另一种替代方案是： 使用tf.train.shuffle_batch 函数,设置num_threads的值大于1。 这种方案可以保证同一时刻读取一个文件(多线程读，读取速度依然优于单线程)，而不是之前的同时读取多个文件。

聚美优品
JUMEI.COM

tf.train.batch和tf.train.shuffle_batch

capacity是队列的长度，min_after_dequeue是出队后，队列至少剩下min_after_dequeue个数据，假设现在有个test.tfrecord文件，里面按从小到大顺序存放整数0~100

- 1. tf.train.batch是按顺序读取数据，队列中的数据始终是一个有序的队列，
- 比如队列的capacity=20，开始队列内容为0,1，..,19=>读取10条记录后，队列剩下10,11，..,19，然后又补充10条变成=>10,11,...,29, 队头一直按顺序补充，队尾一直按顺序出队,到了第100条记录后，又重头开始补充0,1,2...
- 2. tf.train.shuffle_batch是将队列中数据打乱后，再读取出来，因此队列中剩下的数据也是乱序的，队头也是一直在补充（我猜也是按顺序补充），比如batch_size=5,capacity=10,min_after_dequeue=5, 初始是有序的0,1，..,9(10条记录)， 然后打乱8,2,6,4,3,7,9,2,0,1(10条记录), 队尾取出5条，剩下7,9,2,0,1(5条记录),然后又按顺序补充进来，变成7,9,2,0,1,10,11,12,13,14(10条记录)，再打乱13,10,2,7,0,12...1(10条记录)，再出队...
- capacity可以看成是局部数据的范围，读取的数据是基于这个范围的，在这个范围内，min_after_dequeue越大，数据越乱

聚美优品
JUMEI.COM

使用QueueRunner对象来预取

在你运行任何训练步骤之前，需要调用tf.train.start_queue_runners函数，否则数据流图将一直挂起。tf.train.start_queue_runners 这个函数将会启动输入管道的线程，填充样本到队列中，以便出队操作可以从队列中拿到样本。这种情况下最好配合使用一个tf.train.Coordinator，这样可以在发生错误的情况下正确地关闭这些线程。

```
# Create the graph, etc.
init_op = tf.initialize_all_variables()
# Create a session for running operations in the Graph.
sess = tf.Session()
# Initialize the variables (like the epoch counter).
sess.run(init_op)
# Start input enqueue threads.
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
try:
    while not coord.should_stop():
        # Run training steps or whatever
        sess.run(train_op)
except tf.errors.OutOfRangeError:
    print 'Done training -- epoch limit reached'
finally:
    # When done, ask the threads to stop.
    coord.request_stop()
# Wait for threads to finish.
coord.join(threads)
sess.close()
```

聚美优品
JUMEI.COM

数据流图过程

我们先创建数据流图，这个数据流图由一些流水线的阶段组成，阶段间用队列连接在一起。

- 第一阶段将生成文件名，我们读取这些文件名并且把他们排到文件名队列中
- 第二阶段从文件中读取数据（使用Reader），产生样本，而且把样本放在一个样本队列中。
- 在第二阶段的最后是一个排队操作，就是入队到队列中去，在下一阶段出队。迭代训练会使得样本队列中的样本不断地出队。
- 在tf.train中要创建这些队列和执行入队操作，就要添加tf.train.QueueRunner。每个QueueRunner负责一个阶段，处理那些需要在线程中运行的入队操作的列表。一旦数据流图构造成功，tf.train.start_queue_runners函数就会要求数据流图中每个QueueRunner去开始它的线程运行入队操作。
- Coordinator。这是负责在收到任何关闭信号的时候，让所有的线程都知道。最常用的是在发生异常时这种情况就会呈现出来，比如说其中一个线程在运行某些操作时出现错误

聚美优品
JUMEI.COM

Coordinator

Coordinator类用来帮助多个线程协同工作，多个线程同步终止。 其主要方法有：

- should_stop():如果线程应该停止则返回True。
- request_stop(<exception>): 请求该线程停止。
- join(<list of threads>):等待被指定的线程终止。

首先创建一个Coordinator对象，然后建立一些使用Coordinator对象的线程。这些线程通常一直循环运行，一直到should_stop()返回True时停止。 任何线程都可以决定计算什么时候应该停止。它只需要调用request_stop()，同时其他线程的should_stop()将会返回True，然后都停下来。

```
# 线程体：循环执行，直到`Coordinator`收到了停止请求。
# 如果某些条件为真，请求`Coordinator`去停止其他线程。
def MyLoop(coord):
  while not coord.should_stop():
    ...do something...
    if ...some condition...:
      coord.request_stop()
# Main code: create a coordinator.
coord = Coordinator()
# Create 10 threads that run 'MyLoop()'
threads = [threading.Thread(target=MyLoop, args=(coord)) for i in xrange(10)]
# Start the threads and wait for all of them to stop.
for t in threads: t.start()
coord.join(threads)
```

聚美优品
JUMEI.COM

THE END

THANK YOU!

聚美优品
JUMEI.COM