Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

# 1 数据加载

一般地，TensorFlow 输入样本数据到训练/推理子图中执行运算，存在三种读取样本数据的方法：

1. 数据注入：通过字典 feed_dict 将数据传递给 Session.run，以替代 Placeholder 的输出 Tensor 的值；
2. 数据管道：通过构造输入子图，并发地从文件中读取样本数据；
3. 数据预加载：对于小数据集，使用 Const 或 Variable 直接持有数据。

基于大型数据集的训练或推理任务，样本数据的输入常常使用数据的管道模式，确保高的吞吐率，提高训练/推理的执行效率。该过程使用队列实现输入子图与训练/推理子图之间的数据交互与异步控制。

本章将重点论述数据加载的 Pipeline 的工作机制，并深入了解 TensorFlow 并发执行的协调机制，及其队列在并发执行中扮演的角色。

## 1.1 数据注入

数据注入是最为常见的数据加载的方法，它通过字典 feed_dict 将样本数据传递给 Session.run，或者 Tensor.eval 方法；其中，字典的关键字为 Tensor 的名字，值为样本数据。

TensorFlow 将按照字典中 Tensor 的名字，将样本数据替换该 Tensor 的值。

```
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

with tf.Session():
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

一般地，feed_dict 可以替代任何 Tensor 的值。但是，常常使用 Placeholder 表示其输出 Tensor 的值未确定，待使用 feed_dict 替代。

## 1.2 数据预加载

可以使用 Const 或 Variable 直接持有数据，将数据预加载至内存中，提升执行效率。该方法仅适用于小数据集，当样本数据集比较大时，内存资源消耗非常可观。这里以 mnist 数据集为例，讲解数据预加载的使用方法。

```
from tensorflow.examples.tutorials.mnist import input_data

data_sets = input_data.read_data_sets('/tmp/mnist/data')
```

### 1.2.1 使用 Const

由于 Const OP 输出 Tensor 的值是直接内联在计算图中。如果该 Const OP 在图中被使用多次，可能造成重复的冗余数据，白白浪费了不必要的内存资源。

```
with tf.name_scope('input'):
  input_images = tf.constant(data_sets.train.images)
  input_labels = tf.constant(data_sets.train.labels)
```

### 1.2.2 使用 Variable

可以使用不可变、非训练的 Variable 替代 Const。一旦初始化了该类型的 Variable，便不能改变其值，从而具备 Const 的属性。

用于数据预加载的 Variable 与用于训练的 Variable 之间存在差异，它将置位 trainable=False，系统不会将其归类于 GraphKeys.TRAINABLE_VARIABLES 集合中。在训练过程中，系统不会对其实施更新操作。

另外，在构造该类型的 Variable 时，还将设置 collections=[]，系统不会将其归类于 GraphKeys.GLOBAL_VARIABLES 集合中。在训练过程中，系统不会对其实施 Checkpoint 操作。

为了创建不可变、非训练的 Variable，此处写了一个简单的工厂方法。

```
def immutable_variable(initial_value):
  initializer = tf.placeholder(
    dtype=initial_value.dtype,
    shape=initial_value.shape)
  return tf.Variable(initializer, trainable=False, collections=[])
```

immutable_variable 使用传递进来的 initial_value 构造 Placeholder 的类型与形状信息，并以此作为 Variable 的初始值。可以使用 immutable_variable 创建不可变的，用于数据预加载的 Variable。

```
with tf.name_scope('input'):
  input_images = immutable_variable(data_sets.train.images)
  input_labels = immutable_variable(data_sets.train.labels)
```

### 1.2.3 批次预加载

可以构建 Pipeline，结合数据预加载机制，实现样本的批式加载。首先，使用 tf.train.slice_input_producer 在每个 epoch 开始时将整个样本空间随机化，每次从样本集合中随机采样获取一个训练样本。

```
def one(input_xs, input_ys, num_epochs)
  return tf.train.slice_input_producer(
    [input_xs, input_ys], num_epochs=num_epochs)
```

然后，使用 tf.train.batch 每次得到一个批次的样本数据。

```
def batch(x, y, batch_size)
  return tf.train.batch(
    [x, y], batch_size=batch_size)
```

对于使用 Variable 预加载数据，可以如下方式获取一个批次的样本数据。

```
with tf.name_scope('input'):
  input_images = immutable_variable(data_sets.train.images)
  input_labels = immutable_variable(data_sets.train.labels)

  image, label = one(input_images, input_labels, epoch=1)
  batch_images, batch_labels = batch(image, label, batch_size=100)
```

事实上，tf.train.slice_input_producer 将构造样本队列，通过 QueueRunner 并发地通过执行 Enqueue 操作，将训练样本逐一加入到样本队列中去。在每次迭代训练启动时，通过调用 DequeueMany 一次性获取 batch_size 个的批次样本数据到训练子图中去。

## 1.3 数据管道

一个典型的数据加载的 Pipeline(Input Pipeline)，包括如下几个重要数据处理实体：

1. 文件名称队列：将文件名称的列表加入到该队列中；
2. 读取器：从文件名称队列中读取文件名 (出队)；并根据数据格式选择相应的文件读取器，解析文件的记录；
3. 解码器：解码文件记录，并转换为数据样本；
4. 预处理器：对数据样本进行预处理，包括正则化，白化等；
5. 样本队列：将处理后的样本数据加入到样本队列中。

以 mnist 数据集为例，假如数据格式为 TFRecord。首先，使用 tf.train.string_input_producer 构造了一个持有文件名列表的 FIFOQueue 队列 (通过执行 EnqueueMany OP)，并且在每个 epoch 周期内实现文件名列表的随机化。

### 1.3.1 构建文件名队列

```
def input_producer(num_epochs):
  return tf.train.string_input_producer(
    ['/tmp/mnist/train.tfrecords'], num_epochs=num_epochs)
```

构造好了文件名队列之后，使用 tf.TFRecordReader 从文件名队列中获取文件名 (出队，通过调用执行 Dequeue OP)，并从文件中读取样本记录 (Record)。然后，使用 tf.parse_single_example 解析出样本数据。

### 1.3.2 读取器

```
def parse_record(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      features={
          'image_raw': tf.FixedLenFeature([], tf.string),
          'label': tf.FixedLenFeature([], tf.int64),
      })
  return features
```



### 1.3.3 解码器

接着对样本数据进行解码，及其可选的预处理过程，最终得到训练样本。

```
def decode_image(features):
  image = tf.decode_raw(features['image_raw'], tf.uint8)
  image.set_shape([28*28])

  # Convert from [0, 255] -> [-0.5, 0.5] floats.
  image = tf.cast(image, tf.float32) * (1. / 255) - 0.5
  return image

def decode_label(features):
  label = tf.cast(features['label'], tf.int32)
  return label

def one_example(features):
  return decode_image(features), decode_label(features)
```

### 1.3.4 构建样本队列

可以使用 tf.train.shuffle_batch 构建一个 RandomShuffleQueue 队列，将解析后的训练样本追加在该队列中 (通过执行 Enqueue OP)；当迭代执行启动时，将批次获取 batch_size 个样本数据 (通过执行 DequeueMany OP)。

```
def shuffle_batch(image, label, batch_size):
    # Shuffle the examples and collect them into batch_size
    # batches.(Uses a RandomShuffleQueue)
    images, labels = tf.train.shuffle_batch(
      [image, label], batch_size=batch_size, num_threads=2,
      capacity=1000 + 3 * batch_size,
      # Ensures a minimum amount of shuffling of examples.
      min_after_dequeue=1000)
    return images, labels
```

### 1.3.5 输入子图

最后，将整个程序传接起来便构造了一个输入子图。

```
def inputs(num_epochs, batch_size):
  with tf.name_scope('input'):
    filename_queue = input_producer(num_epochs)
    features = parse_record(filename_queue)
    image, label = one_example(features)
    return shuffle_batch(image, label, batch_size)
```



## 1.4 数据协同

事实上，数据加载的 Pipeline 其本质是构造一个输入子图，实现并发 IO 操作，使得训练过程不会因操作 IO 而阻塞，从而实现 GPU 的利用率的提升。

对于输入子图，数据流的处理划分为若干阶段 (Stage)，每个阶段完成特定的数据处理功能；各阶段之间以队列为媒介，完成数据的协同和交互。

如下图所示，描述了一个典型的神经网络的训练模式。整个流水线由两个队列为媒介，将其划分为 3 个阶段。

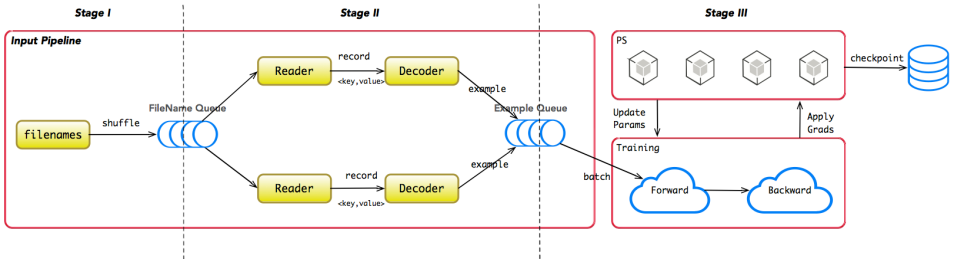


图 1-1 模型训练工作流

### 1.4.1 阶段 1

`string_input_producer` 构造了一个 `FIFOQueue` 的队列，它是一个有状态的 OP。根据 `shuffle` 选项，在每个 epoch 开始时，随机生成文件列表，并将其一同追加至队列之中。

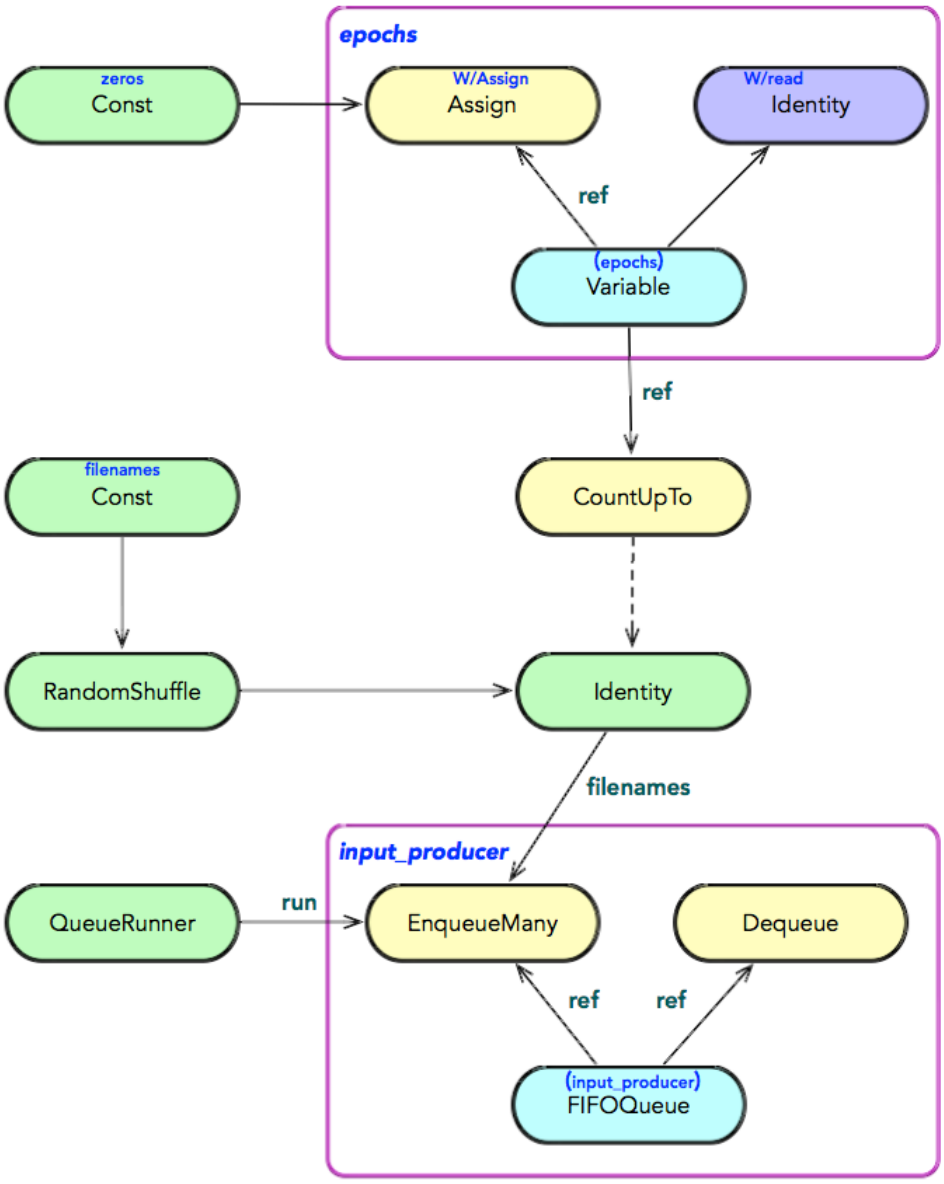


图 1-2 阶段 1：模型训练工作流

### 随机化

首先，执行名为 filenames 的 Const OP，再经过 RandomShuffle 将文件名称列表随机化。

### Epoch 控制

为了实现 epoch 的计数，实现巧妙地设计了一个名为 epochs 的本地变量。其中，本地变量仅对本进程的多轮 Step 之间共享数据，并且不会被训练子图实施更新。

在 Session.run 之前，系统会执行本地变量列表的初始化，将名为 epochs 的 Variable 实施零初始化。

epoch 的计数功能由 CountUpTo 完成，它的工作原理类似于 C++ 的 i++。它持有 Variable 的引用，及其上限参数 limit。每经过一轮 epoch，使其 Variable 自增 1，直至达到 num_epochs 数目。

其中，当 epoch 数到达 num_epochs 时，CountUpTo 将自动抛出 OutOfRangeError 异常。详细实现可以参考 CountUpToOp 的 Kernel 实现。

```
template <class T>
struct CountUpToOp : OpKernel {
  explicit CountUpToOp(OpKernelConstruction* ctxt)
    : OpKernel(ctxt) {
    OP_REQUIRES_OK(ctxt, ctxt->GetAttr("limit", &limit_));
  }

  void Compute(OpKernelContext* ctxt) override {
    T before_increment;
    {
      mutex_lock l(*ctxt->input_ref_mutex(0));

      // Fetch the old tensor
      Tensor tensor = ctxt->mutable_input(0, true);
      T* ptr = &tensor.scalar<T>()();
      before_increment = *ptr;

      // throw OutOfRangeError if exceed limit
      if (*ptr >= limit_) {
        ctxt->SetStatus(errors::OutOfRange(
            "Reached limit of ", limit_));
        return;
      }
      // otherwise increase 1
      ++*ptr;
    }
    // Output if no error.
    Tensor* out_tensor;
    OP_REQUIRES_OK(ctxt, ctxt->allocate_output(
        "output", TensorShape({}), &out_tensor));
    out_tensor->scalar<T>()() = before_increment;
  }

private:
  T limit_;
};
```

#### 入队操作

事实上，将文件名列表追加到队列中，执行的是 EnqueueMany，类似于 Assign 修改 Variable 的值，EnqueueMany 也是一个有状态的 OP，它持有队列的句柄，直接完成队列的状态更新。

在此处，EnqueueMany 将被 Session.run 执行，系统反向遍历，找到依赖的 Identity，发现控制依赖于 CountUpTo，此时会启动一次 epoch 计数，直至到达 num_epoch 数目抛出 OutOfRangeError 异常。同时，Identity 依赖于 RandomShuffle，以便得到随机化了的文件名列表。

### QueueRunner

另外，在调用 tf.train.string_input_producer 时，将往计算图中注册一个特殊的 OP：QueueRunner，并且将其添加到 GraphKeys.QUEUE_RUNNERS 集合中。并且，一个 QueueRunner 持有一个或多个 Enqueue，EnqueueMany 类型的 OP。

## 1.4.2 阶段 2

Reader 从文件名队列中按照 FIFO 的顺序获取文件名，并按照文件名读取文件记录，成功后对该记录进行解码和预处理，将其转换为数据样本，最后将其追加至样本队列中。



#### 读取器

事实上，实现构造了一个 ReaderRead 的 OP，它持有文件名队列的句柄，从队列中按照 FIFO 的顺序获取文件名。

因为文件的格式为 TFRecord，ReaderRead 将委托调用 TFRecordReader 的 OP，执行文件的读取。最终，经过 ReaderRead 的运算，将得到一个序列化了的样本。

#### 解码器

得到序列化了的样本后，将使用合适的解码器实施解码，从而得到一个期望的样本数据。可选地，可以对样本实施预处理，例如 reshape 等操作。

#### 入队操作

得到样本数据后，将启动 QueueEnqueue 的运算，将样本追加至样本队列中去。其中，QueueEnqueue 是一个有状态的 OP，它持有样本队列的句柄，直接完成队列的更新操作。

实施上，样本队列是一个 RandomShuffleQueue，使用出队操作实现随机采样。

#### 并发执行

为了提高 IO 的吞吐率，可以启动多路并发的 Reader 与 Decoder 的工作流，并发地将样本追加至样本队列中去。其中，RandomShuffleQueue 是线程安全的，支持并发的入队或出队操作。

### 1.4.3 阶段 3

当数据样本累计至一个 batch_size 时，训练/推理子图将取走该批次的样本数据，启动一次迭代计算 (常称为一次 Step)。

#### 出队操作

事实上，训练子图使用 DequeueMany 获取一个批次的样本数据。

#### 迭代执行

一般地，一次迭代运行，包括两个基本过程：前向计算与反向梯度传递。Worker 任务使用 PS 任务更新到本地的当前值，执行前向计算得到本次迭代的损失。

然后，根据本次迭代的损失，反向计算各个 Variable 的梯度，并更新到 PS 任务中；PS 任务更新各个 Variable 的值，并将当前值广播到各个 Worker 任务上去。

#### Checkpoint

PS 任务根据容错策略，周期性地实施 Checkpoint。将当前所有 Variable 的数据，及其图的元数据，包括静态的图结构信息，持久化到外部存储设备上，以便后续恢复计算图，及其所有 Variable 的数据。

### 1.4.4 Pipeline 节拍

例如，往 FIFOQueue 的队列中添加文件名称列表，此时调用 EnqueueMany 起始的子图计算，其中包括执行所依赖的 CountUpTo。当 CountUpTo 达到 limit 上限时，将自动抛出 OutOfRangeError 异常。

扮演主程序的 QueueRunner，捕获 coord.join 重新抛出的 OutOfRangeError 异常，随后立即关闭相应的队列，并且退出该线程的执行。队列被关闭之后，入队操作将变为非法；而出队操作则依然合法，除非队列元素为空。

同样的道理，下游 OP 从队列 (文件名队列) 中出队元素，一旦该队列元素为空，则自动抛出 OutOfRangeError 异常。该阶段对应的 QueueRunner 将感知该异常的发生，然后捕获异常并关闭下游的队列 (样本队列)，退出线程的执行。

在 Pipeline 的最后阶段，train_op 从样本队列中出队批次训练样本时，队列为空，并且队列被关闭了，则抛出 OutOfRangeError 异常，最终停止整个训练任务。