当我们已经训练好一个模型时，需要把模型保存下来，这样训练的模型才能够随时随地的加载与使用。模型的保存和加载分为两部分，一部分是模型定义的存取即**图的存取**，另一部分中图中的**变量的存取**（常量存储在图中）。Tensorflow可以分别完成两部分的内容的存取。

图与变量的存取均涉及到了文件IO操作。Tensorflow拥有独立的文件IO API。当然我们也可以使用Python中的文件IO方式，但Tensorflow的API的功能更加丰富。

1. Tensorflow的文件IO模块

在Tensorflow中，文件IO的API类似于Python中对file对象操作的API。但Tensorflow的文件IO会调用C++的接口，实现文件操作。同时Tensorflow的文件IO模块也支持更多的功能，包括操作本地文件、Hadoop分布式文件系统(HDFS)、谷歌云存储等。

Tensorflow的文件IO模块为 `tf.gfile` 。其中提供了丰富的文件操作的API，例如文件与文件夹的创建、修改、复制、删除、查找、统计等。这里我们简单介绍几种常用的文件操作方法。

1.1 打开文件

打开并操作文件首先需要创建一个文件对象。这里有两种方法进行操作：

- `tf.gfile.GFile` 。也可以使用 `tf.gfile.Open` ，两者是等价的。此方法创建一个文件对象，返回一个上下文管理器。

  用法如下：

  ```
  tf.gfile.GFile(name, mode='r')
  ```

  输入一个文件名进行操作。参数 `mode` 是操作文件的类型，
  有 `"r", "w", "a", "r+", "w+", "a+"` 这几种。分别代表只读、只写、增量写、读写、读写（包括创建）、可读可增量写。默认情况下读写操作都是操作的文本类型，如果需要写入或读取bytes类型的数据，就需要在类型后再加一个 `b` 。这里要注意的是，其与Python中文件读取的 `mode` 类似，但不存在 `"t","U"` 的类型（不加 `b` 就等价于Python中的 `t` 类型）。

- `tf.gfile.FastGFile` 。与 `tf.gfile.GFile` 用法、功能都一样(旧版本
  中 `tf.gfile.FastGFile` 支持无阻塞的读取， `tf.gfile.GFile` 不支持。目前的版本都支持无阻塞读取)。一般的，使用此方法即可。

例如：

```
# 可读、写、创建文件
with tf.gfile.GFile('test.txt', 'w+') as f:
    ...

# 可以给test.txt追加内容
with tf.gfile.Open('test.txt', 'a') as f:
    ...

# 只读test.txt
with tf.gfile.FastGFile('test.txt', 'r') as f:
    ...

# 操作二进制格式的文件
with tf.gfile.FastGFile('test.txt', 'wb+') as f:
    ...
```

1.2 文件读取

文件读取使用文件对象的 `read` 方法。（这里我们以 `FastGFile` 为例，与 `GFile` 一样）。文件读取时，会有一个指针指向读取的位置，当调用 `read` 方法时，就从这个指针指向的位置开始读取，调用之后，指针的位置修改到新的未读取的位置。 `read` 的用法如下：

```
# 返回str类型的内容
tf.gfile.FastGFile.read(n=-1)
```

当参数 `n=-1` 时，代表读取整个文件。 `n!=-1` 时，代表读取 `n` 个bytes长度。

例如：

```
with tf.gfile.FastGFile('test.txt', 'r') as f:
    f.read(3)  # 读取前3个bytes
    f.read()  # 读取剩下的所有内容
```

如果我们需要修改文件指针，只读部分内容或跳过部分内容，可以使用 `seek` 方法。用法如下：

```
tf.gfile.FastGFile.seek(
    offset=None,  # 偏移量 以字节为单位
    whence=0,  # 偏移其实位置 0表示从文件头开始(正向) 1表示从当前位置开始(正向) 2表示从文件末尾开始(反向)
    position=None  # 废弃参数 使用`offset`参数替代
)
```

例如：

```
with tf.gfile.FastGFile('test.txt', 'r') as f:
    f.seed(3)  # 跳过前3个bytes
    f.read()  # 读取剩下的所有内容
```

注意：读取文件时，默认的（不加 `b` 的模式）会对文件进行解码。会将bytes类型转换为UTF-8类型，如果读入的数据编码格式不是UTF-8类型，则在解码时会出错，这时需要使用二进制读取方法。

除此以外，还可以使用 `reafline` 方法对文件进行读取。其可以读取以 `\n` 为换行符的文件的一行内容。例如：

```
with tf.gfile.FastGFile('test.txt', 'r') as f:
    f.readline()  # 读取一行内容(包括行末的换行符)
    f.readlines()  # 读取所有行，返回一个list，list中的每一个元素都是一行
```

以行为单位读取内容时，还可以使用 `next` 方法或是使用生成器来读取。如下：

```
with tf.gfile.FastGFile('test.txt', 'r') as f:
    f.next()  # 读取下一行内容

with tf.gfile.FastGFile('test.txt', 'r') as f:
    # 二进制数据首先会将其中的代表`\n`的字符转换为\n，然后会以\n作为分隔符生成list
    lines = [line for line in f]
```

注意：如果没有使用with承载上下文管理器，文件读取完毕之后，需要显示的使用 `close` 方法关闭文件IO。

1.3 其它文件操作

- **文件复制**：`tf.gfile.Copy(oldpath, newpath, overwrite=False)`
- **删除文件**：`tf.gfile.Remove(filename)`
- **递归删除**：`tf.gfile.DeleteRecursively(dirname)`
- **判断路径是否存在**：`tf.gfile.Exists(filename)` # filename可指代路径
- **判断路径是否为目录**：`tf.gfile.IsDirectory(dirname)`
- **返回当前目录下的内容**：`tf.gfile.ListDirectory(dirname)` # 不递归 不显示'.'与'..'
- **创建目录**：`tf.gfile.MkDir(dirname)` # 其父目录必须存在
- **创建目录**：`tf.gfile.MakeDirs(dirname)` # 任何一级目录不存在都会进行创建
- **文件改名**：`tf.gfile.Rename(oldname, newname, overwrite=False)`
- **统计信息**：`tf.gfile.Stat(filename)`
- **文件夹遍历**：`tf.gfile.Walk(top, in_order=True)` # 默认广度优先
- **文件查找**：`tf.gfile.Glob(filename)` # 支持pattern查找

2. 图存取

图是算法过程的描述工具，当我们在某个文件中定义了一个图的时候，也就定义了一个算法，当我们需要运行这个算法时，可以直接找到定义此图的文件，就能操作它。所以，通常地我们并不需要保存图，但如果我们希望一个图能够脱离编程语言环境去使用，这时候就需要将其序列化成为某种固定的数据格式，这时候就可以实现跨语言的操作。

图是由一系列Op与Tensor构成的，我们可以通过某种方法对这些Op与Tensor进行描述，在Tensorflow中这就是'图定义' `GraphDef` 。图的存取本质上就是 `GraphDef` 的存取。

2.1 图的保存

图的保存方法很简单，只需要将图的定义保存即可。所以：

第一步，需要获取图定义。

可以使用 `tf.Graph.as_graph_def` 方法来获取序列化后的图定义 `GraphDef` 。

例如：

```
with tf.Graph().as_default() as graph:
    v = tf.constant([1, 2])
    print(graph.as_graph_def())
```

输出内容：

```
输入内容：
node {
  name: "Const"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_INT32
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_INT32
        tensor_shape {
          dim {
            size: 2
          }
        }
        tensor_content: "\001\000\000\000\002\000\000\000"
      }
    }
  }
}
versions {
  producer: 24
}
```

还可以使用绑定图的会话的 `graph_def` 属性来获取图的序列化后的定义，例如：

```
with tf.Graph().as_default() as graph:
    v = tf.constant([1, 2])
    print(graph.as_graph_def())

with tf.Session(graph=graph) as sess:
    sess.graph_def == graph.as_graph_def()  # True
```

注意：当会话中加入Op时， `sess.graph_def == graph.as_graph_def()` 不再成立。在会话中graph_def会随着Op的改变而改变。

获取了图的定义之后，便可以去保存图。

第二步：保存图的定义

保存图的定义有两种方法，第一种直接将图存文件。利用上面我们学习的文件IO操作即可完成。同时，Tensorflow也提供了专门的保存图的方法，更加便捷。

- 方法一：

  直接创建一个文件保存图定义。如下：

  ```
  with tf.Graph().as_default() as g:
    tf.Variable([1, 2], name='var')

    with tf.gfile.FastGFile('test_model.pb', 'wb') as f:
        f.write(g.as_graph_def().SerializeToString())
  ```

  `SerializeToString` 是将str类型的图定义转化为二进制的proto数据。

- 方法二：

  使用Tensorflow提供的 `tf.train.write_graph` 进行保存。使用此方法还有一个好处，就是可以直接将图传入即可。用法如下：

  ```
  tf.train.write_graph(
    graph_or_graph_def, # 图或者图定义
    logdir,  # 存储的文件路径
    name,   # 存储的文件名
    as_text=True)  # 是否作为文本存储
  ```

  例如，'方法一'中的图也可以这样保存：

  ```
  with tf.Graph().as_default() as g:
    tf.Variable([1, 2], name='var')

    tf.train.write_graph(g, '', 'test_model.pb', as_text=False)
  ```

  这些参数 `as_text` 的值为 `False` ，即保存为二进制的proto数据。此方法等价于'方法一'。

  当 `as_text` 值为 `True` 时，保存的是str类型的数据。通常推荐为 `False` 。

2.2 图的获取

图的获取，即将保存的图的节点加载到当前的图中。当我们保存一个图之后，这个图可以再次被获取到，这个操作是很有用的，例如我们在当前编程语言下构建了一个图，而这个图可能会应用到其它编程语言环境下，就需要将图保存并在需要的时候再次获取。

图的获取步骤如下：

1. 读取保存的图的数据
2. 创建 `GraphDef` 对象
3. 导入 `GraphDef` 对象到当前图中

具体如下：

```
with tf.Graph().as_default() as new_graph:
    with tf.gfile.FastGFile('test_model.pb', 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        tf.import_graph_def(graph_def)
```

这里 `ParseFromString` 是protocal message的方法，用于将二进制的proto数据读取成 `GraphDef` 数据。 `tf.import_graph_def` 用于将一个图定义导入到当前的默认图中。

这里有一个问题，那就是导入图中的Op与Tensor如何获取到呢？ `tf.import_graph_def` 都已经帮我们想到这些问题了。这里，我们可以了解下 `tf.import_graph_def` 的用法：

```
tf.import_graph_def(
    graph_def,  # 将要导入的图的图定义
    input_map=None,  # 替代导入的图中的Tensor
    return_elements=None,  # 返回指定的OP或Tensor(可以使用新的变量绑定)
    name=None,  # 被导入的图中Op与Tensor的name前缀 默认是'import'
    op_dict=None,
    producer_op_list=None):
```

注意：当 `input_map` 不为None时， `name` 必须不为空。

注意：当 `return_elements` 返回Op时，在会话中执行返回为 `None` 。

当然了，我们也可以使用 `tf.Graph.get_tensor_by_name`
与 `tf.Graph.get_operation_by_name` 来获取Tensor与Op，但要注意加上name前缀。

如果图在保存时，存为文本类型的proto数据，即 `tf.train.write_graph` 中的参数 `as_text`
为 `True` 时，获取图的操作稍有不同。即解码时不能使用 `graph_def.ParseFromString` 进行解码，而需要使用 `protobuf` 中的 `text_format` 进行操作，如下：

```
from google.protobuf import text_format

with tf.Graph().as_default() as new_graph:
    with tf.gfile.FastGFile('test_model.pb', 'r') as f:
        graph_def = tf.GraphDef()
        # graph_def.ParseFromString(f.read())
        text_format.Merge(f.read(), graph_def)
        tf.import_graph_def(graph_def)
```

3. 变量存取

变量存储是把模型中定义的变量存储起来，不包含图结构。另一个程序使用时，首先需要重新创建图，然后

将存储的变量导入进来，即模型加载。变量存储可以脱离图存储而存在。

变量的存储与读取，在Tensorflow中叫做检查点存取，变量保存的文件是检查点文件(checkpoint file)，扩展名一般为.ckpt。使用 `tf.train.Saver()` 类来操作检查点。

3.1 变量存储

变量是在图中定义的，但实际上是会话中存储了变量，即我们在运行图的时候，变量才会真正存在，且变量在图的运行过程中，值会发生变化，所以我们需要**在会话中保存变量**。保存变量的方法是 `tf.train.Saver.save()` 。

这里需要注意，通常，我们可以在图定义完成之后初始化 `tf.train.Saver()` 。 `tf.train.Saver()` 在图中的位置很重要，在其之后的变量不会被存储。

创建Saver对象之后，此时并不会保存变量，我们还需要指定会话运行到什么时候时再去保存变量，需要使用 `tf.train.Saver.save()` 进行保存。

例如：

```
with tf.Graph().as_default() as g:
    var1 = tf.Variable(1)
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    sess.run(var1.assign_add(1))
    saver.save(sess, './model.cpkt')
```

`tf.train.Saver.save()` 有两个必填参数，第一个是会话，第二个是存储路径。保存变量之和，会在指定目录下生成四个文件。分别是checkpoint文件、model.ckpt.data-00000-of-00001文件、model.ckpt.index文件、model.ckpt.meta文件。这四个文件的作用分别是：

- checkpoint：保存当前模型的位置与最近的5个模型的信息。这里我们只保存了一个模型，所有只有一个模型信息，也是当前模型的信息。
- model.ckpt.data-00000-of-00001：保存了当前模型变量的数据。
- model.ckpt.meta：保存了 `MetaGraphDef` ，可以用于恢复saver对象。
- model.ckpt.index：辅助model.ckpt.meta的数据文件。

循环迭代算法时存储变量

实际中，我们训练一个模型，通常需要迭代较多的次数，迭代的过程会用去很多的时间，为了避免出现意外情况（例如断电、死机），我们可以每迭代一定次数，就保存一次模型，如果出现了意外情况，就可以快速恢复模型，不至于重新训练。

如下，我们需要迭代1000次模型，每100次迭代保存一次：

```
with tf.Graph().as_default() as g:
    var1 = tf.Variable(1, name='var')
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(1, 1001):
        sess.run(var1.assign_add(1))
        if i % 100 == 0:
            saver.save(sess, './model2.cpkt')

    saver.save(sess, './model2.cpkt')
```

这时候每次存储都会覆盖上次的存储信息。但我们存储的模型并没有与训练的次数关联起来，我们并不知道当前存储的模型是第几次训练后保存的结果，如果中途出现了意外，我们并不知道当前保存的模型是什么时候保存下的。所以通常的，我们还需要将训练的迭代的步数进行标注。在Tensorflow中只需要给save方法加一个参数即可，如下：

```
with tf.Graph().as_default() as g:
    var1 = tf.Variable(1, name='var')
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(1, 1001):
        sess.run(var1.assign_add(1))
        if i % 100 == 0:
            saver.save(sess, './model2.cpkt', global_step=i)  #

    saver.save(sess, './model2.cpkt', 1000)
```

这里，我们增加了给 `saver.save` 增加了 `global_step` 的参数，这个参数可以是一个0阶Tensor或者一个整数。之后，我们生成的保存变量文件的文件名会加上训练次数，并同时保存最近5次的训练结果，即产生了16个文件。包括这五次结果中每次训练结果对应的data、meta、index三个文件，与一个checkpoint文件。

3.2 变量读取

变量读取，即加载模型数据，为了保证数据能够正确加载，必须首先将图定义好，而且必须与保存时的图定义一致。这里“一致”的意思是图相同，对于Python句柄等Python环境中的内容可以不同。

下面我们恢复上文中保存的图：

```
with tf.Graph().as_default() as g:
    var1 = tf.Variable(1, name='var')
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, './model.ckpt')
    print(sess.run(var))  # >> 1001
```

除了使用这种方法恢复变量以外，还可以借助meta数据恢复，

例如：

```
with tf.Graph().as_default() as g:
    var1 = tf.Variable(1, name='var')
    # saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.import_meta_graph('./log/model.ckpt.meta')
    saver.restore(sess, './model.ckpt')
    print(sess.run(var))  # >> 1001
```

不仅如此，使用meta数据进行恢复时，meta数据中也包含了图定义。所以在我们没有图定义的情况下，也可以使用meta数据进行恢复，例如：

```
with tf.Graph().as_default() as g:
    op = tf.no_op()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.import_meta_graph('./log/model.ckpt.meta')
    saver.restore(sess, './model.ckpt')
    print(sess.run(g.get_tensor_by_name('var:0')))  # >> 1001
```

所以，通过meta数据不仅能够恢复变量，也能够恢复图定义。

在恢复数据时，可能存在多个模型保存的文件，为了简便可以使用 `tf.train.get_checkpoint_state()` 来获取最新训练的模型。其返回一个 `CheckpointState` 对象，可以使用这个对象 `model_checkpoint_path` 属性来获得最新模型的路径。例如：

```
...
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    ckpt = tf.train.get_checkpoint_state('./')
    saver.restore(sess1, ckpt.model_checkpoint_path)
```

3.3 注意事项

注意事项1

当一个图对应多个会话时，在不同的会话中使用图的Saver对象保存变量时，并不会互相影响。

例如，对于两个会话 `sess1` 、 `sess2` 分别操作一个图 `g` 中的变量，并存储在不同文件中：

```
with tf.Graph().as_default() as g:
    var = tf.Variable(1, name='var')
    saver = tf.train.Saver()


with tf.Session(graph=g) as sess1:
    sess1.run(tf.global_variables_initializer())
    sess1.run(var.assign_add(1))

    saver.save(sess1, './model1/model.ckpt')


with tf.Session(graph=g) as sess2:
    sess2.run(tf.global_variables_initializer())
    sess2.run(var.assign_sub(1))

    saver.save(sess2, './model2/model.ckpt')
```

当我们分别加载变量时，可以发现，并没有互相影响。如下：

```
with tf.Session(graph=g) as sess3:
    sess3.run(tf.global_variables_initializer())
    saver.restore(sess3, './model1/model.ckpt')
    print(sess3.run(var))  # >> 2

with tf.Session(graph=g) as sess4:
    sess4.run(tf.global_variables_initializer())
    saver.restore(sess4, './model1/model.ckpt')
    print(sess4.run(var))  # >> 0
```

注意事项2

当我们在会话中恢复变量时，必须要求会话所绑定的图与所要恢复的变量所代表的图一致，这里我们需要知道什么样的图时一致。

例如：

```
with tf.Graph().as_default() as g1:
    a = tf.Variable([1, 2, 3])

with tf.Graph().as_default() as g2:
    b = tf.Variable([1, 2, 3])
```

这两个图是一致的，虽然其绑定的Python变量不同。

```
with tf.Graph().as_default() as g1:
    a = tf.Variable([1, 2, 3])

with tf.Graph().as_default() as g2:
    b = tf.Variable([1, 2, 3], name='var')
```

这两个图是不一样的，因为使用了不同的 `name` 。

```
with tf.Graph().as_default() as g1:
    a = tf.Variable([1, 2, 3], name='a')
    b = tf.Variable([4, 5], name='b')

with tf.Graph().as_default() as g2:
    c = tf.Variable([4, 5], name='b')
    d = tf.Variable([1, 2, 3], name='a')
```

这两个图是一致的。

```
with tf.Graph().as_default() as g1:
    a = tf.Variable([1, 2, 3])
    b = tf.Variable([4, 5])

with tf.Graph().as_default() as g2:
    c = tf.Variable([4, 5])
    d = tf.Variable([1, 2, 3])
```

这两个图是不一致。看起来，两个图一模一样，然而Tensorflow会给每个没有 `name` 的Op进行命名，两个图由于均使用了2个Variable，并且都没有主动命名，所以在 `g1` 中a的 `name` 与 `g2` 中c的 `name` 不同， `g1` 中b的 `name` 与 `g2` 中d的 `name` 不同。

4. 变量的值固化到图中

当我们训练好了一个模型时，就意味着模型中的变量的值确定了，不在需要改变了。这时候如果我们想要将训练好的模型运用于生产，按照上述所讲，我们需要分别将图与变量保存（或者保存变量时也保存了meta），这是一种方法。但这种方法是为训练与调整模型所设计，为了还原变量，我们不得不在会话中操作图。例如，现在已经训练好了模型A，现需要利用模型A作为输入，训练模型B，按照上述还原模型的方法，必须在会话中还原模型，那么构建模型的流程就发生了变化，而我们希望算法的设计全部都在图中完成，这就使得图的设计变得麻烦。

所以为了使用简便，可以将所有变量固化成常量，随图一起保存。这样使用起来也更加简便，我们只需要导入图即可完成模型的完整导入。利用固化后的图参与构建新的图也变得容易了。

实现上述功能，需要操作GraphDef中的节点，好在Tensorflow已经为我们提供了相关的API：`convert_variables_to_constants`。

`convert_variables_to_constants`在`tensorflow.python.framework.graph_util`中，用法如下：

```
# 返回一个新的图
convert_variables_to_constants(
    sess,  # 会话
    input_graph_def,  # 图定义
    output_node_names,  # 输出节点(不需要的节点在生成的新图将被排除)(注意是节点而不是Tensor)
    variable_names_whitelist=None,  # 将指定的variable转成constant
    variable_names_blacklist=None)  # 指定的variable不转成constant
```

例如：

将变量转化为常量，边存成图。

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

with tf.Graph().as_default() as g:
    my_input = tf.placeholder(dtype=tf.int32, shape=[], name='input')
    var = tf.get_variable(name='var', shape=[], dtype=tf.int32)
    output = tf.add(my_input, var, name='output')

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    sess.run(var.assign(10))

    # 将变量转化为常量，返回一个新的图
    new_graph = graph_util.convert_variables_to_constants(
        sess,
        sess.graph_def,
        output_node_names=['output'])
    # 将新的图保存
    tf.train.write_graph(new_graph, '', 'graph.pb', as_text=False)
```

导入刚刚序列化的图：

```
with tf.Graph().as_default() as new_graph:
    x = tf.placeholder(dtype=tf.int32, shape=[], name='x')
    with tf.gfile.FastGFile('graph.pb', 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        g_out = tf.import_graph_def(
            graph_def,
            input_map={'input:0': x},
            return_elements=['output:0'])

with tf.Session(graph=new_graph) as sess:
    print(sess.run(g_out[0], feed_dict={x: 5}))  # >> 15
```