

目录

第 1 章 重要的函数汇总.....	3
第 2 章 使用 Premade Estimators.....	5
2.1 创建输入函数 <code>input_fn</code>	5
2.2 定义 <code>feature columns</code>	6
2.3 初始化 <code>Estimator</code>	7
2.4 <code>Train, Evaluate, Predict</code>	7
第 3 章 Checkpoints.....	9
3.1 保存部分训练模型.....	9
3.2 恢复模型.....	9
第 4 章 Feature Columns(待续...).....	11
4.1 <code>Feature columns</code> 概述.....	11
4.2 <code>Numeric column</code>	11
4.3 <code>Bucketized column</code>	12
4.4 <code>Categorical identity column</code>	12
4.4 <code>Categorical vocabulary column</code>	13
4.5 <code>Hashed column</code>	15
4.6 <code>Crossed column</code>	15
4.7 <code>Indicator and embedding columns</code>	15
第 5 章 Datasets 和 Iterator.....	16
5.1 Datasets 概述.....	16
5.2 构建 Datasets.....	16
5.3 操纵 Datasets.....	19
5.4 查看 Datasets 元素.....	19
5.5 创建不同类型的 <code>iterator</code>	20
5.6 <code>iterator</code> 使用注意事项.....	23
5.7 读取输入数据.....	23
第 6 章 创建自定义 Estimators.....	24
6.1 创建自定义 <code>Estimator</code> 步骤.....	24
6.2 创建 <code>model_fn</code>	24
第 7 章 低级 APIs.....	27
7.1 <code>Tensor</code>	27
7.2 <code>Variable</code> 简单知识.....	27
7.3 共享 <code>Variable</code> 和作用域.....	29
7.4 命名 <code>Operations</code>	31
7.5 <code>Graphs</code> 和 <code>Sessions</code>	32
7.6 创建 <code>Layers</code> 和简单的训练代码.....	33
7.7 保存和恢复变量.....	35
7.7 保存和恢复模型.....	39
第 8 章 TensorBoard 可视化.....	40
8.1 相关的函数及其作用.....	40
第 9 章 输入流水线(待更...).....	41
9.1 <code>Tensorflow</code> 读取数据.....	41

9.2 创建 TFRecords.....	41
9.3 读取 TFRecords.....	42

第 1 章 重要的函数汇总

<code>tf.keras.utils.get_file</code>	从 url 下载 cache 中不存在的文件,详情见 help.
<code>tf.train.latest_checkpoint</code> (<code>checkpoint_dir</code>)	查找最新保存的 checkpoint 文件的文件名.一般查询 checkpoint 文件的第一行,如果 checkpoint 文件不存在,则返回 None. checkpoint 文件格式不合法会抛出 ValueError.
<code>Saver.export_meta_graph</code>	将'MetaGraphDef'保存为.meta 文件.
<code>tf.app.run</code>	使用可选的'main'和'argv'列表运行程序。 It's just a very quick wrapper that handles flag parsing and then dispatches to your own main.
<code>dataset.flap_map</code>	
<code>dataset.filter</code>	
<code>tf.data.Iterator.from_structure</code>	用给定的结构创建一个新的未初始化的`Iterator`。
<code>tf.data.Iterator.from_string_handle</code>	
<code>tf.train.MonitoredTrainingSession</code>	
<code>tf.add_n(inputs, name=None)</code>	inputs 中的元素相加,inputs 可以是 list 或 tuple 等
<code>tf.losses.sparse_softmax_cross_entropy</code>	怎么用?
<code>tf.image.convert_image_dtype</code>	转换图像数据类型
<code>tf.decode_raw</code>	将字符串的字节重新解释为数字的向量
<code>tf.subtract</code>	减法
<code>tf.multiply</code>	乘法
<code>tf.reshape</code>	图像 reshape
<code>tf.random_crop</code>	随机切取制定大小的图像
<code>tf.train.string_input_producer</code>	将字符串(e.g.文件名)输出到 queue for an input pipeline,比如读取 TFRecords.
<code>tf.train.Example.SerializeToString</code>	将消息序列化为一个字符串,仅用于已初始化的消息
<code>tf.train.batch</code>	生成 batch
<code>tf.train.shuffle_batch</code>	随机打乱张量来生成 batch
<code>tf.train.FloatList</code>	列表中每个元素为 float
<code>tf.train.Int64List</code>	列表中每个元素为 int64
<code>tf.train.BytesList</code>	列表中每个元素为 bytes
<code>tf.train.Example</code>	
<code>tf.train.Feature</code>	
<code>tf.train.Features</code>	
<code>tf.python_io.TFRecordWriter</code>	类:写入 TFRecords 文件的类
<code>tf.TFRecordReader</code>	类:从 TFRecords 文件输出 records 的类
<code>tf.TFRecordReader.read</code>	返回下一个 record(key,value)对
<code>tf.TFRecordReader.read_up_</code>	返回指定个数的 records

to	
tf.parse_single_example	Parses a single `Example` proto.
tf.FixedLenFeature	用于解析固定长度输入特征的配置

第2章 使用 Premade Estimators

2.1 创建输入函数 input_fn

一般需要创建 3 个输入函数: `train_input_fn`, `test_input_fn`, `predict_input_fn`, 由于前两者需要标签, 后者不需要标签. 通常可以将 `test_input_fn` 和 `predict_input_fn` 合并成一个含有有判断语句函数体的函数 `eval_input_fn`.

`input_fn` 需要返回 `tf.data.DataSet` 对象(类)或一个 `tuple(features, labels)`.

➤ 输入函数的返回值示例:

code1: 返回 Dataset:

```
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.
    # 返回的是 Dataset 的子类 BatchDataset
    return dataset.shuffle(1000).repeat().batch(batch_size)
```

code2: 返回 tuple(features, labels), numpy 格式的数据:

```
def input_evaluation_set():
    features = {'SepalLength': np.array([6.4, 5.0]),
               'SepalWidth': np.array([2.8, 2.3]),
               'PetalLength': np.array([5.6, 3.3]),
               'PetalWidth': np.array([2.2, 1.0])}
    labels = np.array([2, 1])
    return features, labels
```

code3: 返回 tuple(features, labels), tf.Tensor 格式的数据
(见下面的“train_input_fn 输入示例”)

➤ `train_input_fn` 输入示例:

```
def train_input_fn(features, labels, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))
    dataset = dataset.shuffle(1000).repeat().batch(batch_size)
    return dataset.make_one_shot_iterator().get_next()
```

`features` 参数是 `dict`, 每个 `key` 为特征的名称, 每个 `value` 为包含所有特征的值的 `array(np.array 或 pandas.Series)`.

`labels` 参数是包含 `labels` 值的 `array`.

➤ `eval_input_fn` 输入示例:

```
def eval_input_fn(features, labels, batch_size):
    """An input function for evaluation or prediction"""
    features=dict(features)
    if labels is None:
        # No labels, use only features.
        inputs = features
    else:
        inputs = (features, labels)

    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices(inputs)

    # Batch the examples
    assert batch_size is not None, "batch_size must not be None"
    dataset = dataset.batch(batch_size)

    # Return the dataset.
    return dataset
```

2.2 定义 feature columns

参考网址:

https://developers.google.cn/machine-learning/glossary/#feature_columns

特征列(**FeatureColumns**)是一组相关特征,**feature columns** 是描述模型如何使用特征字典中的原始输入数据的对象。例如用户可能居住的所有国家/地区的集合.样本的特征列可能包含一个或多个特征。

我认为,**feature columns** 是输入层的定义,表示了输入层要求的数据格式和输入层神经元的个数。

在 **Iris** 代码中 ,**feature columns** 构造如下 , 以供 **tf.estimator.DNNClassifier** 使用。

```
my_feature_columns = []
for key in train_x.keys():
    my_feature_columns.append(
        tf.feature_column.numeric_column(key=key))

# my_feature_columns 输出:
[NumericColumn(key='SepalLength', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='SepalWidth', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='PetalLength', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='PetalWidth', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None)]
```

2.3 初始化 Estimator

TensorFlow 提供了几个预先制作的分类器 Estimators:

- `tf.estimator.DNNClassifier` 执行多分类的深度模型.
- `tf.estimator.DNNLinearCombinedClassifier` wide&deep models.
- `tf.estimator.LinearClassifier` 基于线性模型的分类器

➤ `tf.estimator.DNNClassifier` 定义:

```
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)
```

2.4 Train, Evaluate, Predict

分 别 使 用 `classifier.train`, `classifier.evaluate`, `classifier.predict` 来执行训练,测试和预测.

Tips start:

在 `classifier.train` 的参数 `input_fn` 接收的是能返回数据的函数本身,比如示例代码中给出的 `train_input_fn` 返回的是 `batch_size` 大小的 `Tensor`, `input_fn` 接受的就是

```
input_fn=lambda:train_input_fn(train_x, train_y, 100)
```

使用的是 `lambda` 匿名函数,如果显式调用的话,可以使用如下方法:

```
def tif():  
    return train_input_fn(train_x, train_y, 100)  
input_fn=tif
```

显然匿名函数更简洁.

Tips end!

需要注意的是

`classifier.train` 返回的是 `op`

`classifier.evaluate` 返回的是 `dict`:

```
{'accuracy': 0.96666664,  
 'average_loss': 0.069688804,  
 'global_step': 5000,  
 'loss': 2.0906641}
```

返回的是生成器 `generator`:

```
[{'class_ids': array([0], dtype=int64),
```

```
'classes': array([b'0'], dtype=object),
'logits': array([ 13.200559 ,  5.3011374, -30.004778 ], dtype=float32),
'probabilities': array([9.9962914e-01, 3.7082061e-04, 1.7218655e-19], dtype=float32)},
{'class_ids': array([1], dtype=int64),
'classes': array([b'1'], dtype=object),
'logits': array([-11.945785 ,  3.5011022, -4.9434695], dtype=float32),
'probabilities': array([1.9561797e-07, 9.9978477e-01, 2.1501844e-04], dtype=float32)},
{'class_ids': array([2], dtype=int64),
'classes': array([b'2'], dtype=object),
'logits': array([-26.961567,  2.661598,  7.870895], dtype=float32),
'probabilities': array([7.414581e-16, 5.435804e-03, 9.945642e-01], dtype=float32)}]
```

'logits'表示输出的单元值

'probabilities'表示 `tf.nn.softmax(r['logits'])`, `r` 是生成器 `predictions` 中的元素.

'class_ids'表示所属类 id

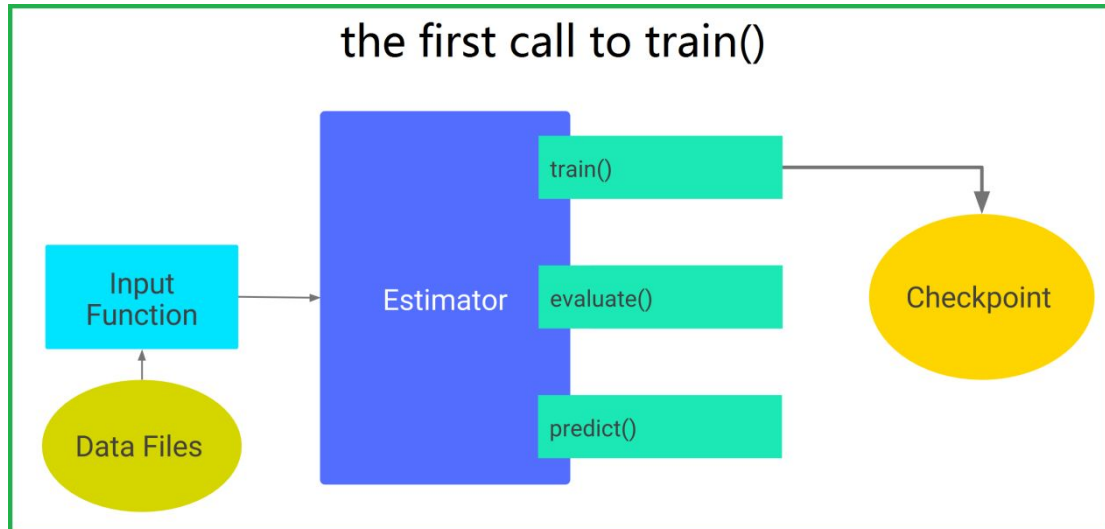
'classes'表示所属类

第3章 Checkpoints

3.1 保存部分训练模型

调用 `classifier.train(...)` 时,生成 Checkpoint

第一次调用 `train` 会向 `model_dir` 添加 checkpoints 和其他文件:



默认情况下, `Estimator` 会根据以下计划将检查点保存在 `model_dir` 中:

- 每 600s 写入一个 checkpoint
- 当 `train` 方法开始迭代和结束迭代的时候
- 只保留目录中最近的 5 个检查点

可以通过 `tf.estimator.RunConfig` 修改配置,如:

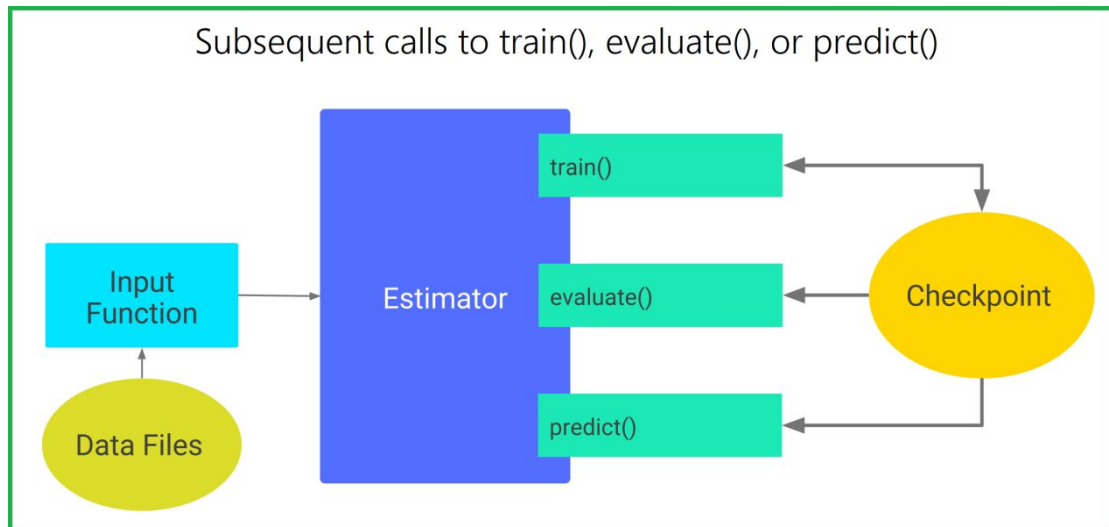
```
my_checkpointing_config = tf.estimator.RunConfig(  
    save_checkpoints_secs = 20*60, # Save checkpoints every 20 minutes.  
    keep_checkpoint_max = 10,      # Retain the 10 most recent checkpoints.)  
classifier = tf.estimator.DNNClassifier(  
    feature_columns=my_feature_columns,  
    hidden_units=[10, 10],  
    n_classes=3,  
    model_dir='models/iris',  
    config=my_checkpointing_config)
```

3.2 恢复模型

第一次调用 `Estimator` 的 `train` 方法, Tensorflow 将模型保存在 `model_dir`. 以后每次对 `Estimator` 的 `train`, `eval`, `predict` 的调用都会导致如下事情发生:

1. `Estimator` 通过运行 `model_fn()` 来建立 `model` 的 `graph`. (有关 `model_fn()` 的详细信息, 请参阅 `(Creating Custom Estimators.)`)
2. `Estimator` 根据最新的 `checkpoint` 中存储的数据初始化新模型的 `weight`.

也就是说,一旦 `checkmodel` 存在, `Tensorflow` 将会在你每次调用 `train()`, `evaluate()`, `predict()` 时重新构建模型.

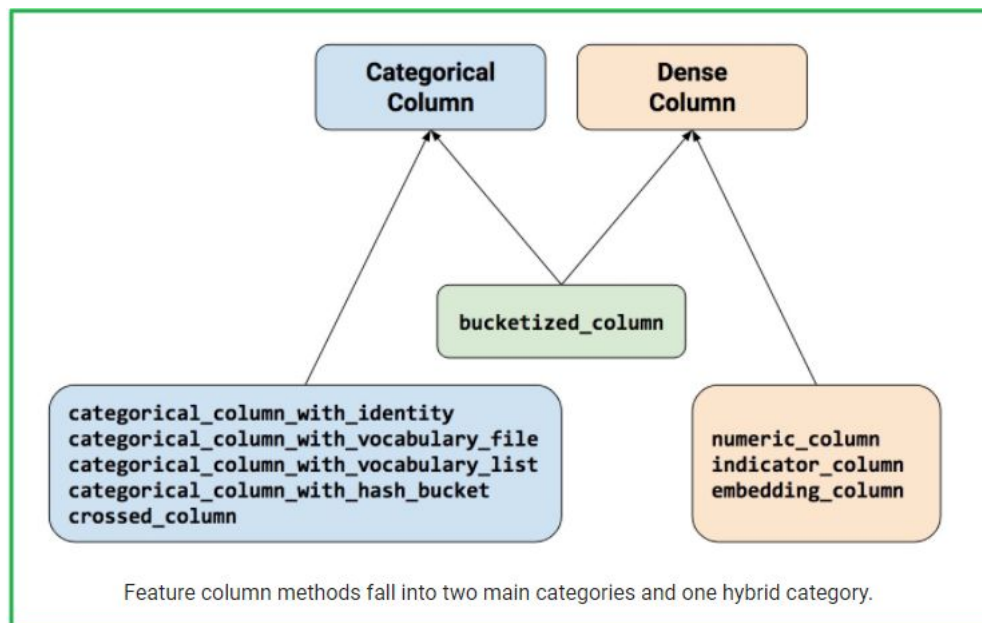
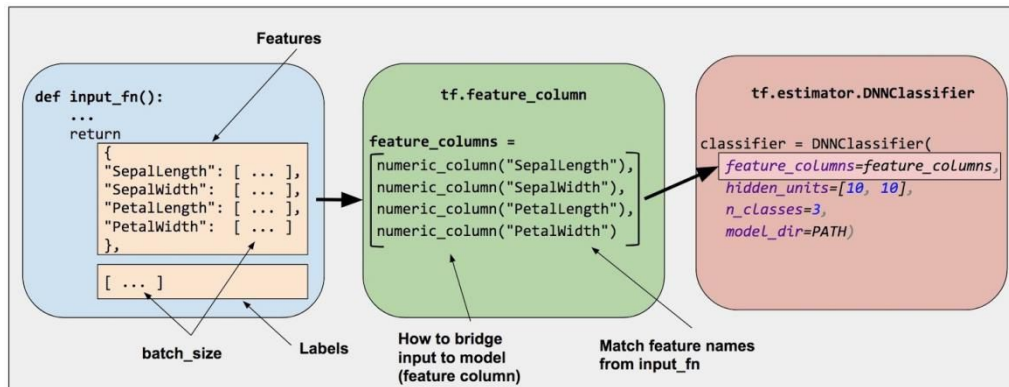


从检查点恢复模型的状态只适用于模型和检查点是兼容的.假设你第一次训练的网络是`[4 10 10 3]`,第二次将网络结构修改为`[4 8 8 3]`,会抛出 `InvalidArgumentError` 错误.这是为了避免错误的恢复.

第4章 Feature Columns(待续...)

4.1 Feature columns 概述

Feature columns bridge raw data with the data your model needs.



Feature column methods fall into two main categories and one hybrid category.

4.2 Numeric column

可以指定 feature columns 的 shape

```
# 定义一个 tf.float32 的标量(scalar).
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength")

# tf.float64 scalar.
numeric_feature_column =
tf.feature_column.numeric_column(key="SepalLength", dtype=tf.float64)
```

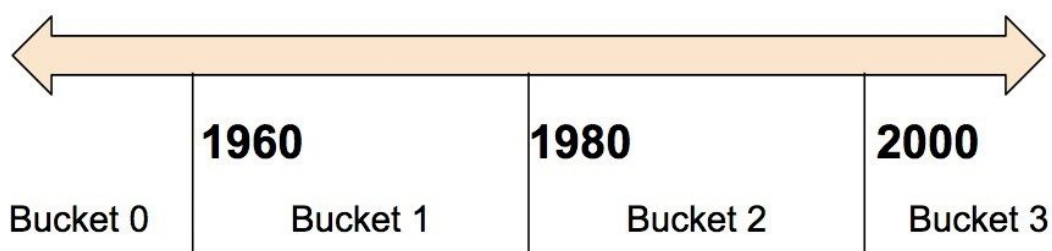
```
# Represent a 10-element vector in which each cell contains a tf.float32.
vector_feature_column = tf.feature_column.numeric_column(key="Bowling", shape=10)

# Represent a 10x5 matrix in which each cell contains a tf.float32.
matrix_feature_column = tf.feature_column.numeric_column(key="MyMatrix", shape=[10,5])
```

4.3 Bucketized column

Bucketized column 将一个 feature 特征分为多个段,如:

Dividing year data into four buckets.



模型如下:

Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
> 2000	[0, 0, 0, 1]

如何创建一个 bucketized column,

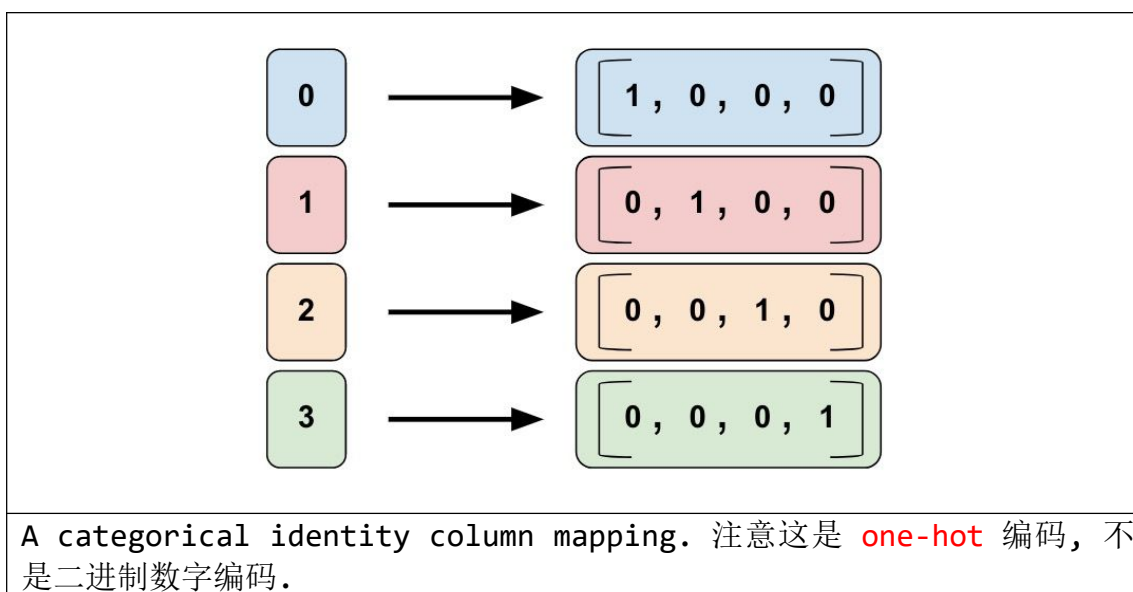
```
# 首先将原始输入转换为numeric column.
numeric_feature_column = tf.feature_column.numeric_column("Year")

# 然后, bucketize the numeric column on the years 1960, 1980, and 2000.
# 注意三个元素的boundaries 生成四个元素的bucketized vector.
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

4.4 Categorical identity column

Categorical identity columns 可以被视为特殊的 bucketized columns. 在传统的 bucketized columns 中,每个 bucket 代表一个值的范围(如,从 1960 到 1970).在 categorical identity columns 每个 bucket 代表单个的,独一无二的整数.

例如你要表示 $[0,4)$, 也就是说你想表示 0,1,2,3. 这种情况下,categorical identity mapping 如下:



调用 `tf.feature_column.categorical_column_with_identity` 实现 categorical identity column.例如:

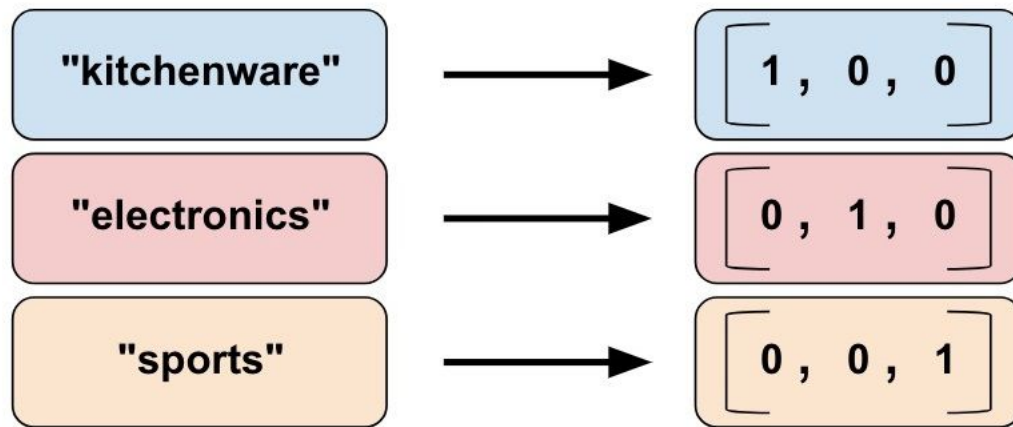
```
# Create categorical output for an integer feature named "my_feature_b",
# The values of my_feature_b must be >= 0 and < num_buckets
identity_feature_column = tf.feature_column.categorical_column_with_identity(
    key='my_feature_b',
    num_buckets=4) # Values [0, 4)

# In order for the preceding call to work, the input_fn() must return
# a dictionary containing 'my_feature_b' as a key. Furthermore, the values
# assigned to 'my_feature_b' must belong to the set [0, 4).
def input_fn():
    ...
    return ({ 'my_feature_a':[7, 9, 5, 2], 'my_feature_b':[3, 1, 2, 2] },
            [Label_values])
```

4.4 Categorical vocabulary column

我们不能直接将 strings 输入到 model 中,相反,我们必须将 strings 映射到 numerical or categorical values. Categorical vocabulary columns 提供了一个很好的方法去用 one-hot vector 表示 strings. 例如,

Mapping string values to vocabulary columns.



TF 提供两个不同的函数来创建 categorical vocabulary columns:

- [`tf.feature_column.categorical_column_with_vocabulary_list`](#)
 - [`tf.feature_column.categorical_column_with_vocabulary_file`](#)
- ...list 缺点: 类型很多时, vocabulary_list 将会很大, 此时使用 ...file 创建一个 `categorical_column_with_vocabulary_list`

```
# Given input "feature_name_from_input_fn" which is a string,
# create a categorical feature by mapping the input to one of
# the elements in the vocabulary list.
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_list(
        key="a feature returned by input_fn()",
        vocabulary_list=["kitchenware", "electronics", "sports"])
```

创建一个 `categorical_column_with_vocabulary_file`

```
# Given input "feature_name_from_input_fn" which is a string, # create a
categorical feature to our model by mapping the input to one of # the elements in
the vocabulary file
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_file(
        key="a feature returned by input_fn()",
        vocabulary_file="product_class.txt",
        vocabulary_size=3)

#product_class.txt
kitchenware
electronics
```

sports

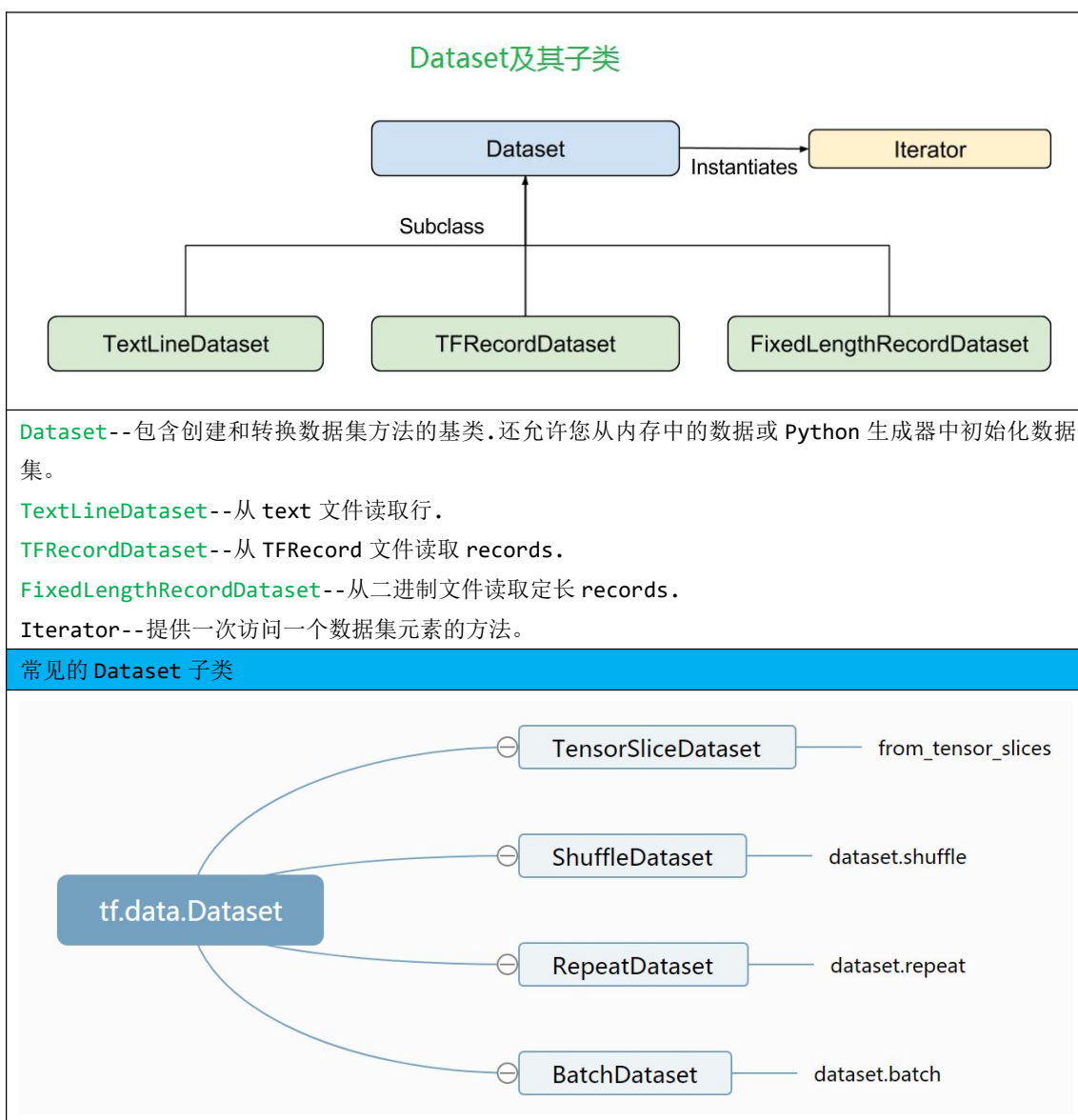
4.5 Hashed column

4.6 Crossed column

4.7 Indicator and embedding columns

第 5 章 Datasets 和 Iterator

5.1 Datasets 概述



5.2 构建 Datasets

- 通过 **tf.data.Dataset.from_tensor_slices** 函数构建
from_tensor_slices 返回 tf.data.Dataset 的子类 TensorSliceDataset。
from_tensor_slices 可接受 ndarray, DataFrame, Series 等类 array(__array__) 的数据结构,并在第 0th 维将数组分开,如:


```

In [13]: import numpy as np
eat = np.arange(24).reshape(-1,6)
eat

Out[13]: array([[ 0,  1,  2,  3,  4,  5],
               [ 6,  7,  8,  9, 10, 11],
               [12, 13, 14, 15, 16, 17],
               [18, 19, 20, 21, 22, 23]])

In [14]: dsl = tf.data.Dataset.from_tensor_slices(eat)

iterator = dsl.make_one_shot_iterator()
with tf.Session() as sess:
    while True:
        try:
            A = sess.run(iterator.get_next())
        except:
            break
    print(A)

[0 1 2 3 4 5]
[ 6  7  8  9 10 11]
[12 13 14 15 16 17]
[18 19 20 21 22 23]

```

给 `from_tensor_slices` 提供不同参数,返回值 `TensorSliceDataset` 中的元素也不一样.

- 当输入为 `np.ndarray` 时, `TensorSliceDataset` 中元素为 `np.ndarray`
- 当输入为 `(dict(train x))` 时, `TensorSliceDataset` 中元素为 `dict`
- 当输入为 `(dict(train x), train y)` 时, `TensorSliceDataset` 中元素为 `tuple`, 其中第一个元素为 `dict`, 与上述 `dict` 一致, 第二个元素为标签.

➤ 通过 csv 文件直接构建

`TextLineDataset` 按行读取文件

```

train_path = 'iris_train.csv'
# 读取 csv 文件并跳过第一行(第一行为标题)# ds 为 Dataset 子类 SkipDataset
ds = tf.data.TextLineDataset(train_path).skip(1)

```

ds.map 使用

在 `iris_data.py` 中, 生成 `SkipDataset` 实例 `ds` 后, 调用了

```
dataset = ds.map(_parse_line)
```

其中, `_parse_line` 为:

```

def _parse_line(line):
    # Decode the line into its fields
    fields = tf.decode_csv(line, record_defaults=CSV_TYPES)

    # Pack the result into a dictionary
    features = dict(zip(CSV_COLUMN_NAMES, fields))

    # Separate the label from the features
    label = features.pop('Species')

```

```
return features, label
```

以下两段代码输出的结果是一样的：

代码 1：

```
iterator = ds.make_one_shot_iterator()
line = iterator.get_next()
fields = tf.decode_csv(line, record_defaults=CSV_TYPES)
features = dict(zip(CSV_COLUMN_NAMES, fields))
label = features.pop('Species')

with tf.Session() as sess:
    while True:
        try:
            A = sess.run(iterator.get_next())
        except:
            break
    print(A)
```

代码 2：

```
dataset = ds.map(_parse_line)
with tf.Session() as sess:
    iterator = dataset.make_one_shot_iterator()
    count = 20
    while True:
        try:
            A = sess.run(iterator.get_next())
        except:
            break
    print(A)
```

tf.decode_csv

将 CSV records 转换为 tensors,

csv records	tensor
b'6.4,2.8,5.6,2.2,2'	[<tf.Tensor 'DecodeCSV_11:0' shape=() dtype=float32>, <tf.Tensor 'DecodeCSV_11:1' shape=() dtype=float32>, <tf.Tensor 'DecodeCSV_11:2' shape=() dtype=float32>, <tf.Tensor 'DecodeCSV_11:3' shape=() dtype=float32>, <tf.Tensor 'DecodeCSV_11:4' shape=() dtype=int32>]

转换为 tensors 后,还需要

```
features = dict(zip(SV_COLUMN_NAMES, fields))
```

转换为

```
{ 'PetalLength': <tf.Tensor 'DecodeCSV_11:2' shape=() dtype=float32>,  
  'PetalWidth': <tf.Tensor 'DecodeCSV_11:3' shape=() dtype=float32>,
```

```
'SepalLength': <tf.Tensor 'DecodeCSV_11:0' shape=() dtype=float32>,  
'SepalWidth': <tf.Tensor 'DecodeCSV_11:1' shape=() dtype=float32>}
```

5.3 操纵 Datasets

通过

```
dataset = tf.data.Dataset.from_tensor_slices((dict(train_x), train_y))
```

得到基本 dataset.

- `dataset.shuffle(1000)` 随机打乱数据, 返回 Dataset 子类 `ShuffleDataset`.
- `dataset.shuffle(1000).repeat(count)` 对 dataset 重复 count 次, 如果 count 为 `None` 或 -1, 则重复无限次. 返回 Dataset 子类 `RepeatDataset`.
- `dataset.shuffle(1000).repeat().batch(100)` 将连续 elements 组合成 batch. 返回 `BatchDataset`.
 - ◆ 调用 `batch(n)` 前, 每次迭代输出一个 tuple, 这个 tuple 包含一个 example.
 - ◆ 调用 `batch(n)` 后, 每次迭代亦输出一个 tuple, 只是这个 tuple 包含 n 个 examples.

关于 `dataset.batch`

若 dataset 在调用 `.repeat()` 前, 调用 `.batch(batch_size)`, 如果 `batch_size` 大于数据集的大小, 将会返回数据集大小的 batch.

5.4 查看 Datasets 元素

`dataset.make_one_shot_iterator` 与 `make_initializable_iterator` 的不同.

前者返回的 iterator 是自动 initialized, 而后者需要 initialize.

- (1) `Dataset.make_one_shot_iterator()` 不支持来自状态对象 (stateful objects) 的初始化, 如 `Variable` 或 `LookupTable`.
比如: `tf.random_uniform([4, 10])`
- (2) one-shot 不支持参数化 (parameterization)

前者只能访问 Dataset 一次, 后者可以用相同的程序重复访问 training 和 validation data 多次. (如何呈现?)

```
# make_one_shot_iterator  
iterator = dataset.make_one_shot_iterator()  
with tf.Session() as sess:
```

```

while True:
    try:
        A = sess.run(iterator.get_next())
    except:
        break
print(A)

```

-----分割线

```

# make_initializable_iterator
with tf.Session() as sess:
    iterator = dataset.make_initializable_iterator()
    sess.run(iterator.initializer)
    while True:
        try:
            A = sess.run(iterator.get_next())
        except:
            break
    print(A)

```

5.5 创建不同类型的 iterator

可以创建 4 钟类型的 iterator:

- one-shot
- initializable
- reinitializable
- feedable

one-shot

one-shot 只支持对 Dataset 一次迭代,且无需初始化.

API: `dataset.make_one_shot_iterator()`

参考代码:

```

dataset = tf.data.Dataset.range(100)
iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

for i in range(100):
    value = sess.run(next_element)
    assert i == value

```

initializable

参数化数据集的定义

API: `dataset.make_initializable_iterator()`

```
max_value = tf.placeholder(tf.int64, shape=[])
dataset = tf.data.Dataset.range(max_value)
iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()

# Initialize an iterator over a dataset with 10 elements.
sess.run(iterator.initializer, feed_dict={max_value: 10})
for i in range(10):
    value = sess.run(next_element)
```

reinitializable

可以初始化具有相同 structure 的 Dataset.

一般步骤:

(1) 创建迭代器

```
iterator = tf.data.Iterator.from_structure(...)
```

(2) 初始化迭代器

```
iterator.make_initializer(...)
```

```
# Define training and validation datasets with the same structure.
training_dataset = tf.data.Dataset.range(100).map(
    lambda x: x + tf.random_uniform([], -10, 10, tf.int64))
validation_dataset = tf.data.Dataset.range(50)

# 定义 reinitializable 迭代器
iterator = tf.data.Iterator.from_structure(training_dataset.output_types,
                                           training_dataset.output_shapes)

next_element = iterator.get_next()

training_init_op = iterator.make_initializer(training_dataset)
validation_init_op = iterator.make_initializer(validation_dataset)

for _ in range(20):
    # Initialize an iterator over the training dataset.
    sess.run(training_init_op)
    for _ in range(100):
        sess.run(next_element)

    # Initialize an iterator over the validation dataset.
    sess.run(validation_init_op)
    for _ in range(50):
        sess.run(next_element)
```

feedable

API: `iterator = tf.data.Iterator.from_string_handle(...)`

```
# Define training and validation datasets with the same structure.
training_dataset = tf.data.Dataset.range(100).map(
    lambda x: x + tf.random_uniform([], -10, 10, tf.int64)).repeat()
validation_dataset = tf.data.Dataset.range(50)

# A feedable iterator is defined by a handle placeholder and its structure.
We
# could use the `output_types` and `output_shapes` properties of either
# `training_dataset` or `validation_dataset` here, because they have
# identical structure.
handle = tf.placeholder(tf.string, shape=[])
iterator = tf.data.Iterator.from_string_handle(
    handle, training_dataset.output_types, training_dataset.output_shapes)
next_element = iterator.get_next()

# You can use feedable iterators with a variety of different kinds of iterator
# (such as one-shot and initializable iterators).
training_iterator = training_dataset.make_one_shot_iterator()
validation_iterator = validation_dataset.make_initializable_iterator()

# The `Iterator.string_handle()` method returns a tensor that can be evaluated
# and used to feed the `handle` placeholder.
training_handle = sess.run(training_iterator.string_handle())
validation_handle = sess.run(validation_iterator.string_handle())

# Loop forever, alternating between training and validation.
while True:
    # Run 200 steps using the training dataset. Note that the training dataset
    is
    # infinite, and we resume from where we left off in the previous `while`
    loop
    # iteration.
    for _ in range(200):
        sess.run(next_element, feed_dict={handle: training_handle})

    # Run one pass over the validation dataset.
    sess.run(validation_iterator.initializer)
    for _ in range(50):
        sess.run(next_element, feed_dict={handle: validation_handle})
```

5.6 iterator 使用注意事项

- 调用 `Iterator.get_next()` 不会立即更新 `iterator`:

```
dataset = tf.data.Dataset.range(5)
iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()

# Typically `result` will be the output of a model, or an optimizer's
# training operation.
result = tf.add(next_element, next_element)

sess.run(iterator.initializer)
print(sess.run(result)) # ==> "0"
print(sess.run(result)) # ==> "2"
print(sess.run(result)) # ==> "4"
print(sess.run(result)) # ==> "6"
print(sess.run(result)) # ==> "8"
try:
    sess.run(result)
except tf.errors.OutOfRangeError:
    print("End of dataset") # ==> "End of dataset"
```

- 如果数据集的每个元素都有嵌套结构, 则 `Iterator.get_next()` 的返回值将是同一个嵌套结构中的一个或多个 `tf.Tensor` 对象:

```
dataset1 = tf.data.Dataset.from_tensor_slices(tf.random_uniform([4, 10]))
dataset2 = tf.data.Dataset.from_tensor_slices((tf.random_uniform([4]),
tf.random_uniform([4, 100])))
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))

iterator = dataset3.make_initializable_iterator()

sess.run(iterator.initializer)
next1, (next2, next3) = iterator.get_next()
```

注意: `next1, next2, next3` 由相同的 `op/node` 产生, 因此, 评估任何这些张量将推进所有组件的迭代器。

5.7 读取输入数据

- 读取数据的方法:

1. 如果内存可以装下数据(**fit in memory**),最简单的方法是使用

```
tf.data.Dataset.from_tensor_slices()
```

2. 如果数据较大,可以使用 **TFRecordDataset**

```
dataset = tf.data.TFRecordDataset
```

3. 读取文本文件,可以使用 **TextLineDataset**,默认情况下,一次读取一行:

```
dataset = tf.data.TextLineDataset
```

● 对输入数据的处理:

包括:

```
Dataset.map()
```

```
tf.py_func()
```

```
Dataset.batch
```

```
Dataset.padded_batch
```

第 6 章 创建自定义 Estimators

6.1 创建自定义 Estimator 步骤

(1) 编写 input function

➤ 使用 `tf.data.Dataset.from_tensor_slices` 将输入转换为 Dataset;

➤ `shuffle, repeat, batch`;

➤ 返回 Dataset pipeline

(2) 创建 Feature columns

(3) 编写 classifier

```
classifier = tf.estimator.Estimator(  
    model_fn=my_model,  
    model_dir='./logs',  
    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,  
    })
```

(4) 编写 model function

(5) 实例化,训练

6.2 创建 model_fn

model_fn 编写步骤:

- (1)model_fn 签名
- (2)定义神经网络结构
- (3)实现 train,evaluate,predict

(1) model_fn 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
```

features,labels: 来自 train,evaluate 和 predict 函数参数调用的 input_fn.

params: 来自 Estimator 初始化的时候指定的参数.

mode: 不同的调用函数,mode 对应的值也不一样

调用函数	mode 值
classifier.train	tf.estimator.ModeKeys.TRAIN 即 'train'
classifier.evaluate	tf.estimator.ModeKeys.EVAL 即 'eval'
classifier.predict	tf.estimator.ModeKeys.PREDICT 即 'infer'

当 call train, evaluate, predict 方法时, Estimator 框架会 invoke model_fn.并设置 mode.

(2) 定义神经网络结构

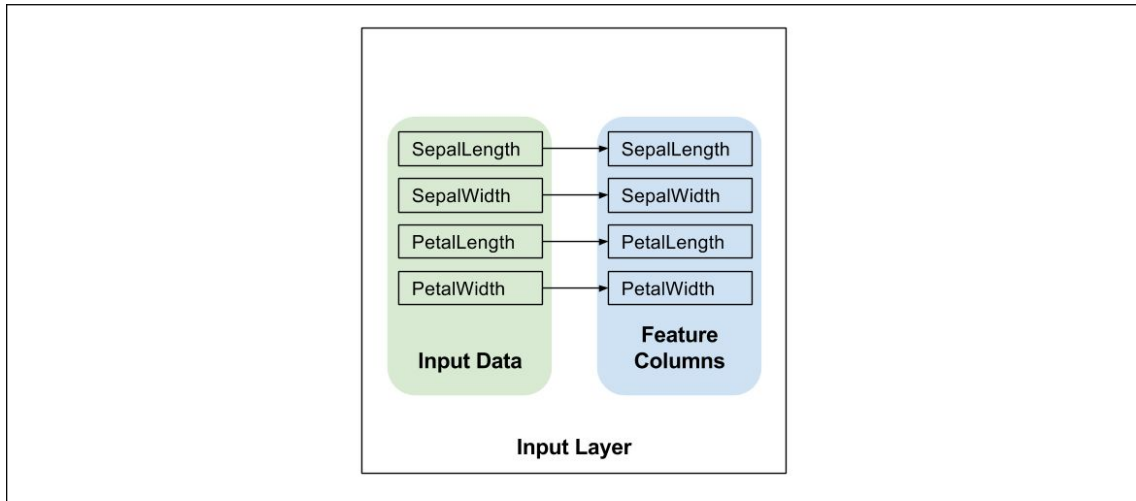
使用 tf.feature_column.input_layer 构建第一层输入.

使用 tf.layers.dense 构建隐层和输出层的.

示例代码:

```
# 输入层  
net = tf.feature_column.input_layer(features, params['feature_columns'])  
# 隐藏层  
for units in params['hidden_units']:  
    net = tf.layers.dense(net, units=units, activation=tf.nn.relu)  
# 输出层  
logits = tf.layers.dense(net, params['n_classes'], activation=None)
```

我的理解:input_layer 要求的 features 为从键到张量的映射(a mapping from key to tensor).而 model function 在解析 features 时,是通过 params['feature_columns'] 中的 keys 进行一一对应的.如下图



(3) 实现 train, evaluate, predict

前文说过,当 call `train`, `evaluate`, `predict` 方法时, `Estimator` 框架会 invoke model fn.并设置 `mode`. 因此,可以根据 `mode` 的值,分别编写 `train`,`evaluate`,`predict` 要处理的语句和返回的值.

对于 **predict**:

先求出最后一层最大值对应的索引,然后计算 `prediction`,并返回 `EstimatorSpec`.

```
predicted_classes = tf.argmax(logits, 1)

if mode == tf.estimator.ModeKeys.PREDICT:
    predictions = {
        'class_ids': predicted_classes[:, tf.newaxis],
        'probabilities': tf.nn.softmax(logits),
        'logits': logits,
    }
    return tf.estimator.EstimatorSpec(mode, predictions=predictions)
```

对于 **evaluate** 和 **train**:

先计算 `loss`

`train` 阶段需要构建优化器 `optimizer` 并最小化.

注:其余信息需要在以后更新!

第 7 章 低级 APIs

7.1 Tensor

- Tensor 的 rank 表示 Tensor 的维度。获取 rank: `tf.rank(...)`
- 主要的 Tensor:
 - `tf.Variable`
 - `tf.constant`
 - `tf.placeholder`
 - `tf.SparseTensor`

查看 tensor 的一些方法:

code1:

```
my_image = tf.zeros([10, 299, 299, 3])
r = tf.rank(my_image)
sess = tf.Session()
print(r.eval(session=sess))
sess.close()
```

code2:

```
my_image = tf.zeros([10, 299, 299, 3])
r = tf.rank(my_image)
sess = tf.Session()
with sess.as_default():
    print(r.eval())
sess.close()
```

code3:

```
my_image = tf.zeros([10, 299, 299, 3])
r = tf.rank(my_image)
with tf.Session() as sess:
    print(sess.run(r))
```

7.2 Variable 简单知识

1. 变量的创建和初始化

```
# 创建指定 shape 的 Variable, 随机初始化
my_variable = tf.get_variable("my_variable", [1, 2, 3])

# 指定 0 初始化
```

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3],
dtype=tf.int32,
    initializer=tf.zeros_initializer)
# 常数初始化
other_variable = tf.get_variable("other_variable", dtype=tf.int32,
    initializer=tf.constant([23, 42]))
```

2.Variable collection

默认情况下,tf.Variable 会被放在
tf.GraphKeys.GLOBAL_VARIABLES--可以在多个设备上共享的变量
tf.GraphKeys.TRAINABLE_VARIABLE--可以被计算梯度的变量
中.

如果不想让变量 trainable,使用

tf.GraphKeys.LOCAL_VARIABLES--不可以被训练

```
my_local = tf.get_variable("my_local", shape=(),
collections=[tf.GraphKeys.LOCAL_VARIABLES])
```

也可以将变量添加到指定 collection 中:

```
tf.add_to_collection("my_collection_name", my_local)
```

获得指定 collection 中的所有变量:

```
tf.get_collection("my_collection_name")
```

获得 global,trainable,local 的 collection 中的变量:

```
tf.trainable_variables()
tf.global_variables()
tf.local_variables()
```

3.初始化变量

初始化全局变量

tf.global_variables_initializer()

获取未被初始化的变量:

tf.report_uninitialized_variables()

4.assign 和 assign_add

`a = v.assign(1)` 是直接给 `a` 赋值 1

`a = v.assign_add(1)` 是把 `v` 的值加 1 后赋给 `a`

7.3 共享 Variable 和作用域

参考网址:

https://tensorflow.google.cn/programmers_guide/variables

<https://www.zhihu.com/question/54513728>

有关 `tf.Variable`, `tf.get_variable`, `tf.name_scope`, `tf.variable_scope` 的小结.

- `tf.Variable` 会自动检测命名冲突并自行处理.
- `tf.get_variable` 遇到重名的变量且变量名没有设置为共享变量时,则会报错.
- 在 `tf.name_scope` 下时, `tf.get_variable()` 创建的变量名不受 `name_scope` 的影响, 而且在未指定共享变量时, 如果重名会报错, `tf.Variable()` 会自动检测有没有变量重名, 如果有则会自行处理. 如:

```
In [11]: with tf.name_scope('ns1'):
          with tf.variable_scope('vs1'):
              var1 = tf.get_variable(name = 'v1', initializer=1.)
          var1
```

```
Out[11]: <tf.Variable 'vs1/v1:0' shape=() dtype=float32_ref>
```

```
In [13]: with tf.name_scope('ns2'):
          with tf.variable_scope('vs2'):
              var2 = tf.Variable(3., name = 'v2')
          var2
```

```
Out[13]: <tf.Variable 'ns2_1/vs2/v2:0' shape=() dtype=float32_ref>
```

- `tf.name_scope(<scope_name>)`: 主要用于管理一个图里面的各种 `op`, 返回的是一个以 `scope_name` 命名的 `context manager`。一个 `graph` 会维护一个 `name_space` 的堆, 每一个 `namespace` 下面可以定义各种 `op` 或者子 `namespace`, 实现一种层次化有条理的管理, 避免各个 `op` 之间命名冲突。
- `tf.variable_scope(<scope_name>)`: 一般与 `tf.name_scope()` 配合使用, 用于管理一个 `graph` 中变量的名字, 避免变量之间的命名冲突, `tf.variable_scope(<scope_name>)` 允许在一个 `variable_scope` 下面共享变量。

TF 官网上给出的代码重用例子:

https://tensorflow.google.cn/programmers_guide/variables

```
import numpy as np
import tensorflow as tf

def conv_relu(input, kernel_shape, bias_shape):
    # Create variable named "weights".
    weights = tf.get_variable("weights", kernel_shape,
                              initializer=tf.random_normal_initializer())
    # Create variable named "biases".
    biases = tf.get_variable("biases", bias_shape,
                              initializer=tf.constant_initializer(0.0))
    conv = tf.nn.conv2d(input, weights, strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)

def my_image_filter(input_images):
    with tf.variable_scope("conv1"):
        # Variables created here will be named "conv1/weights",
        # "conv1/biases".
        relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
    with tf.variable_scope("conv2"):
        # Variables created here will be named "conv2/weights",
        # "conv2/biases".
        return conv_relu(relu1, [5, 5, 32, 32], [32])

input1 = np.zeros(shape=[1,10,10,32],dtype='f4')
input2 = np.ones(shape=[1,10,10,32],dtype='f4')

# 变量重用方法 1
with tf.variable_scope("model"):
    output1 = my_image_filter(input2)
with tf.variable_scope("model", reuse=True):
    output2 = my_image_filter(input2)

# 变量重用方法 2
with tf.variable_scope("model") as scope:
    output1 = my_image_filter(input2)
    scope.reuse_variables()
    output2 = my_image_filter(input2)

# 变量重用方法 2
with tf.variable_scope("model") as scope:
    output1 = my_image_filter(input2)
```

```

with tf.variable_scope(scope, reuse=True):
    output2 = my_image_filter(input2)

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    A1 = sess.run(output1)
    A2 = sess.run(output2)

```

7.4 命名 Operations

`tf.Tensor` 对象是以生成张量作为输出的 `tf.Operation` 隐式命名的。`tf.Tensor` 名称的格式为 “<OP_NAME>:<i>”:

“<OP_NAME>” is the name of the operation that produces it.

“<i>” is an integer representing the index of that tensor among the operation's outputs.

代码例子:

```

# => operation named "c"
c_0 = tf.constant(0, name="c")

# => operation named "c_1"
c_1 = tf.constant(2, name="c")

# Name scopes add a prefix to all operations created in the same context.
with tf.name_scope("outer"):
    # => operation named "outer/c"
    c_2 = tf.constant(2, name="c")

    # Name scopes nest like paths in a hierarchical file system.
    with tf.name_scope("inner"):
        # => operation named "outer/inner/c"
        c_3 = tf.constant(3, name="c")

    # => operation named "outer/c_1"
    c_4 = tf.constant(4, name="c")

    # Already-used name scopes will be "uniquified".
    with tf.name_scope("inner"):
        # => operation named "outer/inner_1/c"
        c_5 = tf.constant(5, name="c")

```

7.5 Graphs 和 Sessions

1. 将 graph 保存为 events 文件的几个方法:

```
x = tf.constant([[37.0, -23.0], [1.0, 4.0]], name='input')
w = tf.Variable(tf.random_uniform([2, 2]), name='weight')
y_true = tf.constant([[10, 24], [5, 6]], name='labels')
y = tf.matmul(x, w, name='predict')
# ...
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    # `sess.graph` provides access to the graph used in a `tf.Session`.
    writer = tf.summary.FileWriter('./test_log')
    writer.add_graph(sess.graph)
    for i in range(1000):
        sess.run(train_op)
    writer.close()
```

调用

```
writer = tf.summary.FileWriter('./test_log')
writer.add_graph(sess.graph)
writer.close()
```

或

```
writer = tf.summary.FileWriter('./test_log', sess.graph)
writer.close()
```

都可以保存 graph.

保存默认 graph, 使用:

```
writer = tf.summary.FileWriter('./test_log', tf.get_default_graph())
writer.close()
```

2. 多个 graphs 的编程

```
import tensorflow as tf
import numpy as np

g_1 = tf.Graph()
```



```

with g_1.as_default():
    # Operations created in this scope will be added to `g_1`.
    a1 = tf.constant(3.,name='a1')
    b1 = tf.constant(4.,name='b1')
    c1 = tf.multiply(a1,b1)

    with tf.Session() as sess1:
        print(sess1.run(c1))

g_2 = tf.Graph()
with g_2.as_default():
    # Operations created in this scope will be added to `g_2`.
    a2 = tf.placeholder(tf.float32, shape=(2,2))
    b2 = tf.constant(np.arange(1,5).reshape(-1,2).astype('f4'))
    c2 = tf.matmul(a2,b2)

    with tf.Session() as sess2:
        sess2.run(tf.global_variables_initializer())
        print(sess2.run(c2,feed_dict={a2:np.array([[2,3],[1,4]]]}))

# 保存相应的graph .# 也可以使用 saver.add_graph 函数
saver = tf.summary.FileWriter('./log',graph=g_2)
saver.close()

```

3. 图和会话中的随机量

随机值变量(random value)在每次调用 run 时,值都不一样,但在单个 run 的过程中,值是相同的:

```

vec = tf.random_uniform(shape=(3,))
out1 = vec + 1
out2 = vec + 2
print(sess.run(vec))
print(sess.run(vec))
print(sess.run((out1, out2)))

```

7.6 创建 Layers 和简单的训练代码

1. 创建 Layer:

1).使用 tf.layers.Dense 类创建

```

x = tf.placeholder(tf.float32, shape=[None, 3])
linear_model = tf.layers.Dense(units=1, activation=None)

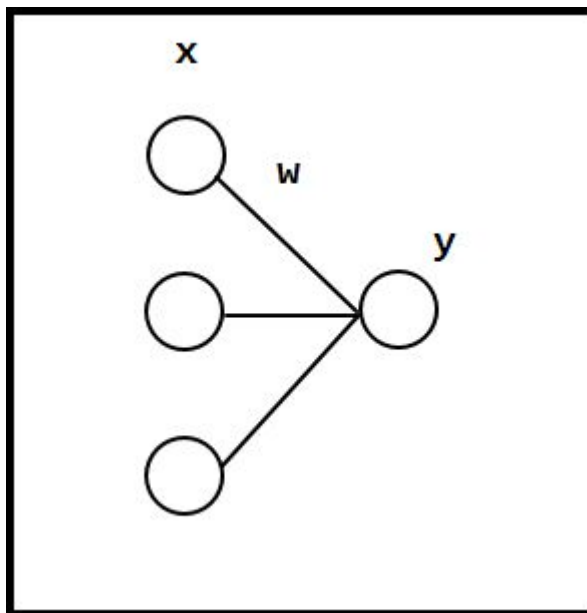
# linear_model 类可__call__
y = linear_model(x)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    A = sess.run({'y':y, 'wb':linear_model.weights},
feed_dict={x:[[1,2,3]]})
    print(A['y'])
    print(A['wb'])

```

这个模型结构图如下：



计算公式如下：

$\text{outputs} = \text{activation}(\text{inputs} * \text{kernel} + \text{bias})$

2). 使用 `tf.layers.dense` 函数创建

```

x = tf.placeholder(tf.float32, shape=[None, 3])
y = tf.layers.dense(x, units=1)

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

```

```
print(sess.run(y, {x: [[1, 2, 3], [4, 5, 6]]}))
```

如果获取 weights?

2. 简单训练过程的代码段:

```
# 输入和标签
x = tf.constant([[1], [2], [3], [4]], dtype=tf.float32)
y_true = tf.constant([[0], [-1], [-2], [-3]], dtype=tf.float32)

# 构造模型
linear_model = tf.layers.Dense(units=1)
y_pred = linear_model(x)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

# 查看训练前的预测值
print(sess.run(y_pred))

# Loss
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y_pred)
print(sess.run(loss))

optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

for i in range(1000):
    _, loss_value = sess.run((train, loss))

print(sess.run(y_pred))

sess.close()
```

7.7 保存和恢复变量

1. 概览

TF 实现 Saving 和 Restoring model 的类为 `tf.train.Saver`
`tf.train.Saver.save` 生成如下文件

- **checkpoint** 文件保存目录下所有模型文件列表。checkpoint 中内容的格式为 CheckpointState Protocol Buffer。
- **.meta** 文件保存了当前 graph 的结构, 可以理解为神经网络的网络结构通过元图 (MetaGraph) 来记录计算图中节点的信息以及运行计算图中节点所需要的元数据。TensorFlow 中元图是由 MetaGraphDef Protocol Buffer 定义的。MetaGraphDef 中的内容构成了 TensorFlow 持久化时的第一个文件。保存 MetaGraphDef 信息的文件默认以 .meta 为后缀名, 文件 model.ckpt.meta 中存储的就是元图数据。
- **.index** 文件保存了当前参数名
- **.data** 文件保存了当前参数值
- **event file** 包含了 TensorBoard 可视化需要的信息

2. 保存变量

inc_v1.op.run() 作用同 sess.run(inc_v1)





```
# Create some variables.
v1 = tf.get_variable("v1", shape=[3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", shape=[5], initializer = tf.zeros_initializer)

inc_v1 = v1.assign(v1+1)
dec_v2 = v2.assign(v2-1)

# Add an op to initialize the variables.
init_op = tf.global_variables_initializer()
# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, initialize the variables, do some work, and save the
# variables to disk.
with tf.Session() as sess:
    sess.run(init_op)
    # Do some work with the model.
    inc_v1.op.run()
    dec_v2.op.run()
    # Save the variables to disk.
    save_path = saver.save(sess, "/tmp/model.ckpt")
    print("Model saved in path: %s" % save_path)
```

执行以上代码会在指定目录中生成相应文件:

	checkpoint	2018/4/28 14:49	文件	1 KB
	model.ckpt.data-00000-of-00001	2018/4/28 14:49	DATA-00000-OF...	1 KB
	model.ckpt.index	2018/4/28 14:49	INDEX 文件	1 KB
	model.ckpt.meta	2018/4/28 14:49	META 文件	6 KB

3. 恢复变量

1). 查看模型中的全部变量

```
import tensorflow as tf
tf.reset_default_graph()
with tf.Session() as sess:
    # 加载 graph
    new_saver = tf.train.import_meta_graph('./logs/model.ckpt-1.meta')
    # 恢复先前保存的变量, 因为第 2 个参数返回的是 model.ckpt 的路径,
    # 所以可以直接替换为路径, 如 './logs\\model.ckpt-1000'
    new_saver.restore(sess, tf.train.latest_checkpoint('./logs'))
    # 返回使用 `trainable = True` 创建的所有变量的 List
    all_vars = tf.trainable_variables()
    B = sess.run(all_vars)
```

2). 这个方法需要预先知道有哪些变量

```
tf.reset_default_graph()
# op 名称需要与保存的模型变量相同
v1 = tf.get_variable("v1", shape=[3])
v2 = tf.get_variable("v2", shape=[5])

saver = tf.train.Saver()
with tf.Session() as sess:
    # Restore variables from disk.
    saver.restore(sess, "./tmp/model.ckpt")
    print("Model restored.")
    # Check the values of the variables
    print("v1 : %s" % v1.eval())
    print("v2 : %s" % v2.eval())
```

注:

保存的变量都以指定的名称作为保存名称, 如在 saving 中

```
v1 = tf.get_variable("v1", shape=[3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", shape=[5], initializer = tf.zeros_initializer)
```

v1, v2 指定的名称为 'v1' 和 'v2'. 则在 restoring 的过程中需要把变量名称指定为相同的名称, 如

```
v1 = tf.get_variable("v1", shape=[3])
v2 = tf.get_variable("v2", shape=[5])
```

和等式左边的值是没有关系的, 如

在 saving 中将要保存的变量名称指定为 'v1x', 'v2x'

```
v1 = tf.get_variable("v1x", shape=[3], initializer = tf.zeros_initializer)
```

```
v2 = tf.get_variable("v2x", shape=[5], initializer = tf.zeros_initializer)
```

则在 `restoring` 中,需要指定名称为'v1x','v2x',和等式左边的值没有关系,但在打印时需要使用等式左边的值.

```
v3 = tf.get_variable("v1x", shape=[3])  
v4 = tf.get_variable("v2x", shape=[5])
```

3). `restoring` 部分变量

- 在 `tf.train.Saver({key:value})` 的参数 `{key:value}` 中添加想要 `restoring` 的变量.
- 需要对 not `restoring` 的变量进行初始化.

代码如下:

```
tf.reset_default_graph()  
# Create some variables.  
v1 = tf.get_variable("v1", [3], initializer = tf.zeros_initializer)  
v2 = tf.get_variable("v2", [5], initializer = tf.zeros_initializer)  
  
# Add ops to save and restore only `v2` using the name "v2"  
saver = tf.train.Saver({"v2": v2})  
# Use the saver object normally after that.  
with tf.Session() as sess:  
    # Initialize v1 since the saver will not.  
    v1.initializer.run()  
    saver.restore(sess, "/tmp/model.ckpt")  
    print("v1 : %s" % v1.eval())  
    print("v2 : %s" % v2.eval())
```

4). `Inspect variables in checkpoint`

```
from tensorflow.python.tools import inspect_checkpoint as chkp  
# 打印model 所有的变量  
chkp.print_tensors_in_checkpoint_file(  
    file_name = "./tmp/model.ckpt",  
    tensor_name='',  
    all_tensors=True,  
    all_tensor_names=False)
```

查看 `help` 知:

`file_name`: 指 `checkpoint` 文件名称

`tensor_name`: 要打印的变量名,如果为 `None` 或'', 打印所有变量.

`all_tensors`: 表示是否打印所有的变量

`all_tensor_names`: 表示是否打印所有变量的名称

另外,该函数执行时调用了两个文件:`model.ckpt.data` 和 `model.ckpt.index`

7.7 保存和恢复模型

https://tensorflow.google.cn/programmers_guide/saved_model#build_and_load_a_savedmodel

没看懂!

第 8 章 TensorBoard 可视化

8.1 相关的函数及其作用

<code>tf.summary.scalar</code>	统计标量
<code>tf.summary.histogram</code>	统计直方图
<code>tf.summary.image</code>	显示图像
<code>tf.summary.merge_all</code>	在运行它们之前, TensorFlow 中的操作不会执行任何操作, 除非 op 依赖于它们的输出。 创建的 summary nodes 是图形的外围设备: 您当前运行的操作都不依赖于它们。 所以, 为了生成 summary , 我们需要运行所有这些汇总节点。 手动管理它们会很乏味, 所以使用 <code>tf.summary.merge_all</code> 将它们组合成一个单独的 op 来生成所有的 summary data 。
<code>tf.summary.FileWriter</code>	将 summary 写入 disk, 将 summary protobuf 传入 <code>tf.summary.FileWriter</code>

第 9 章 输入流水线(待更...)

9.1 Tensorflow 读取数据

参考网址:

tensorflow 学习笔记（四十二）：输入流水线：

<https://blog.csdn.net/u012436149/article/details/72353313>

tensorflow 载入数据的三种方式：

<https://blog.csdn.net/lujiandong1/article/details/53376802>

TensorFlow 读取数据主要有三种方式：

- (1) 预加载数据
- (2) Feeding
- (3) 从文件读取数

预加载数据:将数据全部放在内存中，适合少量数据。

Feeding: 运行时将数据喂到图中

从文件读取: 适合大规模数据，使用到了线程(`tf.Coordinator`)、队列(`tf.QueueRunner`)等知识。

9.2 创建 TFRecords

- (1) 创建写入 TFRecords 文件的类
- (2) 将特征和标签使用 `tf.train.Feature` 包装,并放入字典中
- (3) 使用 `tf.train.Features` 和 `tf.train.Example` 包装
- (4) 将(3)生成的数据序列化为 `string`,并写入 TFRecords 文件中
- (5) 关闭 TFRecords 文件

mnist 数据转化为 TFRecords 文件示例代码：

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

mnist = input_data.read_data_sets('./mnist')
train_images = mnist.train.images
train_labels = mnist.train.labels
test_images = mnist.test.images
test_labels = mnist.test.labels

''' 保存为 TFRecords '''

SAVE_PATH = './mnist_test.tfrecords'

# 准备一个 writer 用来写 TFRecords 文件
writer = tf.python_io.TFRecordWriter(SAVE_PATH)
```

```

for i in range(len(test_images)):
    image = test_images[i,:].reshape(-1,28).tobytes()
    label = test_labels[i]
    feature = {
        'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image])),
        'label': tf.train.Feature(int64_list=tf.train.Int64List(value=[label]))
    }
    example = tf.train.Example(features = tf.train.Features(feature = feature))
    writer.write(example.SerializeToString())

writer.close()

```

9.3 读取 TFRecords

- (1) 创建读取 TFRecords 文件的类, reader = `tf.TFRecordReader()`
- (2) 将 TFRecords 文件读取为 Queue
- (3) 调用 `reader.read` 或 `reader.read_up_to` 读取(2)中队列
- (4) 调用 `tf.parse_single_example` 解析(3)中的数据
- (5) 调用相关线程管理 APIs, 如 `tf.train.Coordinator`, `tf.train.start_queue_runners` 等读取数据。

读取 9.2 节生成的 TFRecords 文件代码:

```

# 所需库
import tensorflow as tf
import matplotlib.pyplot as plt

filename = 'mnist_test.tfrecords'
filename_queue = tf.train.string_input_producer([filename], num_epochs=None)

reader = tf.TFRecordReader()
_, serialized_example = reader.read_up_to(filename_queue, 32)
features = tf.parse_single_example(
    serialized_example[0],
    features={
        'image': tf.FixedLenFeature([], tf.string),
        'label': tf.FixedLenFeature([], tf.int64), # 读取 onehot 需加具体的维度
    })

# 要注意 decode 的数据类型, 类型不正确, image 的 shape 的大小也不一样
image = tf.decode_raw(features['image'], tf.float32)

```

```
label = features['label']

sess = tf.InteractiveSession()
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(coord=coord)

image_val, label_val = sess.run([image, label])

coord.request_stop()
coord.join(threads)
sess.close()
```