Google Developers

Local blog for Chinese language

TensorFlow 数据集和估算器介绍

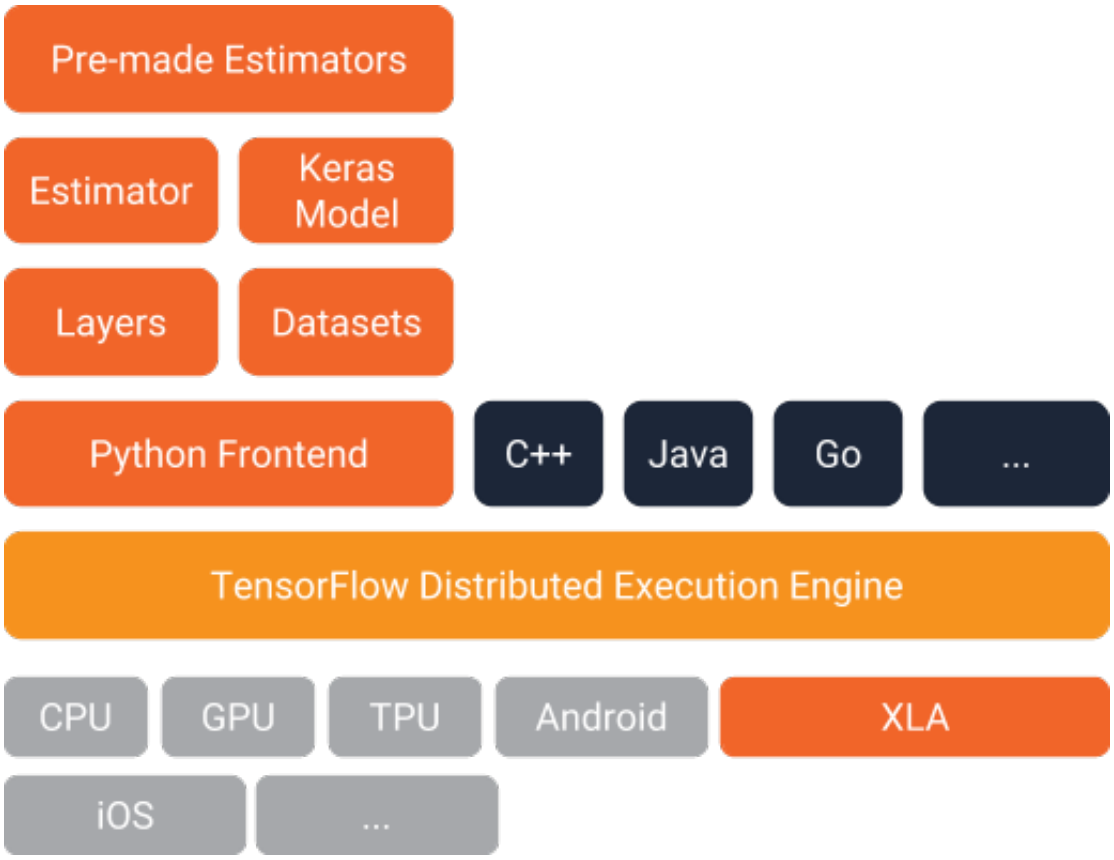
2017年9月25日星期一

发布人：TensorFlow 团队

TensorFlow 1.3 引入了两个重要功能，您应当尝试一下：

- 数据集：一种创建输入管道（即，将数据读入您的程序）的全新方式。
- 估算器：一种创建 TensorFlow 模型的高级方式。估算器包括适用于常见机器学习任务的预制模型，不过，您也可以使用它们创建自己的自定义模型。

下面是它们在 TensorFlow 架构内的装配方式。结合使用这些估算器，可以轻松地创建 TensorFlow 模型和向模型提供数据：



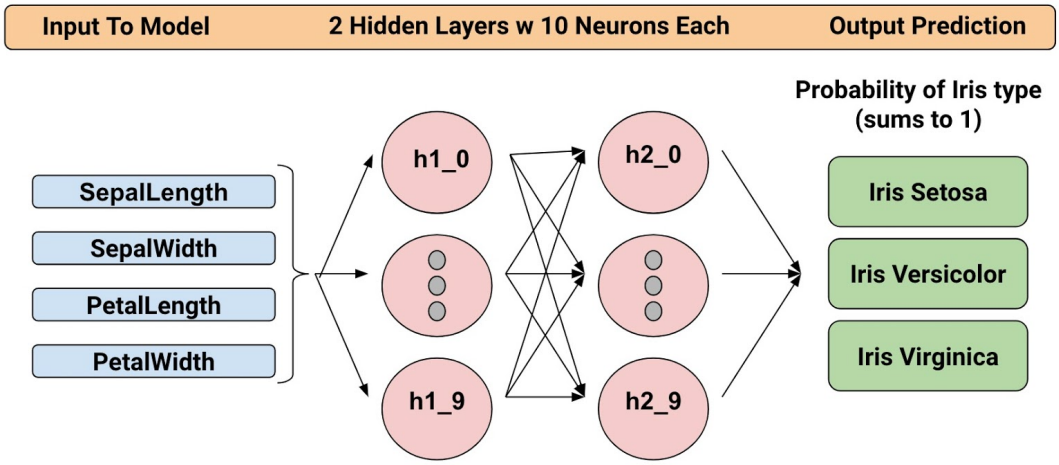
我们的示例模型

为了探索这些功能，我们将构建一个模型并向您显示相关的代码段。完整代码在这里，其中包括获取训练和测试文件的说明。请注意，编写的代码旨在演示数据集和估算器的工作方式，并没有为了实现最大性能而进行优化。

经过训练的模型可以根据四个植物学特征（萼片长度、萼片宽度、花瓣长度和花瓣宽度）对鸢尾花进行分类。因此，在推理期间，您可以为这四个特征提供值，模型将预测花朵属于以下三个美丽变种之中的哪一个：

从左到右依次为：山鸢尾（Radomil 摄影，CC BY-SA 3.0）、变色鸢尾（Dlanglois 摄影，CC BY-SA 3.0）和维吉尼亚鸢尾（Frank Mayfield 摄影，CC BY-SA 2.0）。

我们将使用下面的结构训练深度神经网络分类器。所有输入和输出值都是 `float32`，输出值的总和将等于 1（因为我们在预测属于每种鸢尾花的可能性）：



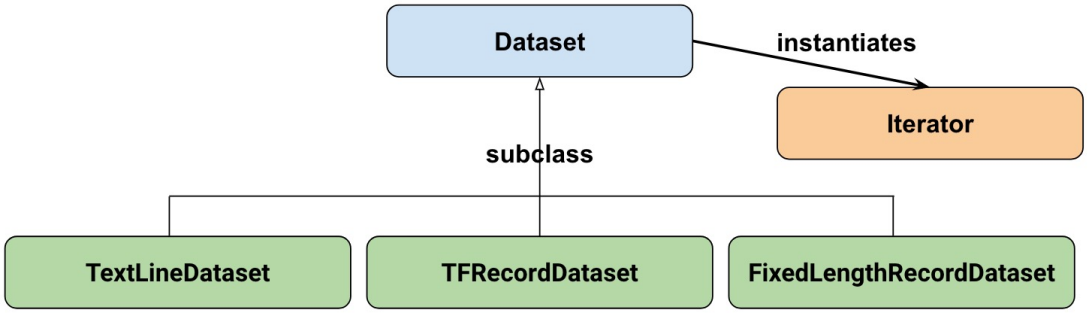
例如，输出结果对山鸢尾来说可能是 0.05，对变色鸢尾是 0.9，对维吉尼亚鸢尾是 0.05，表示这种花有 90% 的可能性是变色鸢尾。

好了！我们现在已经定义模型，接下来看一看如何使用数据集和估算器训练模型和进行预测。

数据集介绍

数据集是一种为 TensorFlow 模型创建输入管道的新方式。使用此 API 的性能要比使用 feed_dict 或队列式管道的性能高得多，而且此 API 更简洁，使用起来更容易。尽管数据集在 1.3 版本中仍位于 tf.contrib.data 中，但是我们预计会在 1.4 版本中将此 API 移动到核心中，所以，是时候尝试一下了。

从高层次而言，数据集由以下类组成：

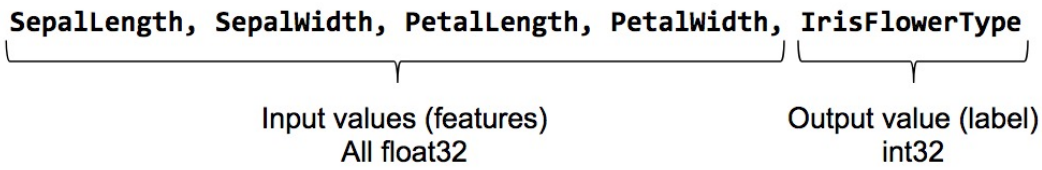


其中：

- 数据集：基类，包含用于创建和转换数据集的函数。允许您从内存中的数据或从 Python 生成器初始化数据集。
- TextLineDataset：从文本文件中读取各行内容。
- TFRecordDataset：从 TFRecord 文件中读取记录。
- FixedLengthRecordDataset：从二进制文件中读取固定大小的记录。
- 迭代器：提供了一种一次获取一个数据集元素的方法。

我们的数据集

首先，我们来看一下要用来为模型提供数据的数据集。我们将从一个 CSV 文件读取数据，这个文件的每一行都包含五个值 - 四个输入值，加上标签：



标签的值如下所述：

- 山鸢尾为 0
- 变色鸢尾为 1
- 维吉尼亚鸢尾为 2。

表示我们的数据集

为了说明我们的数据集，我们先来创建一个特征列表：

```
feature_names = [
    'SepalLength',
    'SepalWidth',
    'PetalLength',
    'PetalWidth']
```

在训练模型时，我们需要一个可以读取输入文件并返回特征和标签数据的函数。估算器要求您创建一个具有以下格式的函数：

```
def input_fn():
    ...<code>...
    return ({ 'SepalLength':[values], ..<etc>.., 'PetalWidth':[values
            [IrisFlowerType])
```

返回值必须是一个按照如下方式组织的两元素元组：

- 第一个元素必须是一个字典（其中的每个输入特征都是一个

键），然后是一个用于训练批次的值列表。

- 第二个元素是一个用于训练批次的标签列表。

由于我们要返回一批输入特征和训练标签，返回语句中的所有列表都将具有相同的长度。从技术角度而言，我们在这里说的“列表”实际上是指 1-d TensorFlow 张量。

为了方便重复使用 `input_fn`，我们将向其中添加一些参数。这样，我们就可以使用不同设置构建输入函数。参数非常直观：

- `file_path`：要读取的数据文件。
- `perform_shuffle`：是否应将记录顺序随机化。
- `repeat_count`：在数据集中迭代记录的次数。例如，如果我们指定 1，那么每个记录都将读取一次。如果我们不指定，迭代将永远持续下去。

下面是我们使用 Dataset API 实现此函数的方式。我们会将它包装到一个“输入函数”中，这个输入函数稍后将用于为我们的估算器模型提供数据：

```
def my_input_fn(file_path, perform_shuffle=False, repeat_count=1):
   def decode_csv(line):
       parsed_line = tf.decode_csv(line, [[0.], [0.], [0.], [0.], [0]
       label = parsed_line[-1:] # Last element is the label
       del parsed_line[-1] # Delete last element
       features = parsed_line # Everything (but last element) are the
       d = dict(zip(feature_names, features)), label
       return d

   dataset = (tf.contrib.data.TextLineDataset(file_path) # Read text
       .skip(1) # Skip header row
       .map(decode_csv)) # Transform each elem by applying decode_csv
```

```
    if perform_shuffle:
        # Randomizes input using a window of 256 elements (read into m
        dataset = dataset.shuffle(buffer_size=256)
    dataset = dataset.repeat(repeat_count) # Repeats dataset this # ti
    dataset = dataset.batch(32)  # Batch size to use
    iterator = dataset.make_one_shot_iterator()
    batch_features, batch_labels = iterator.get_next()
    return batch_features, batch_labels
```

注意以下内容：

- `TextLineDataset`：在您使用 Dataset API 的文件式数据集时，它将为您执行大量的内存管理工作。例如，您可以读入比内存大得多的数据集文件，或者以参数形式指定列表，读入多个文件。
- `shuffle`：读取 buffer_size 记录，然后打乱（随机化）它们的顺序。
- `map`：调用 `decode_csv` 函数，并将数据集中的每个元素作为一个参数（由于我们使用的是 TextLineDataset，每个元素都将是一行 CSV 文本）。然后，我们将向每一行应用 `decode_csv`。
- `decode_csv`：将每一行拆分成各个字段，根据需要提供默认值。然后，返回一个包含字段键和字段值的字典。map 函数将使用字典更新数据集中的每个元素（行）。

以上是数据集的简单介绍！为了娱乐一下，我们现在可以使用下面的函数打印第一个批次：

```
next_batch = my_input_fn(FILE, True) # Will return 32 random elements

# Now let's try it out, retrieving and printing one batch of data.
```

```
# Although this code looks strange, you don't need to understand
# the details.
with tf.Session() as sess:
    first_batch = sess.run(next_batch)
print(first_batch)

# Output
({'SepalLength': array([ 5.4000001, ...<repeat to 32 elems>], dtype=f
  'PetalWidth': array([ 0.40000001, ...<repeat to 32 elems>], dtype=f
  ...
 },
 [array([[2], ...<repeat to 32 elems>], dtype=int32) # Labels
)
```

这就是我们需要 Dataset API 在实现模型时所做的全部工作。不过，数据集还有很多功能；请参阅我们在这篇博文的末尾列出的更多资源。

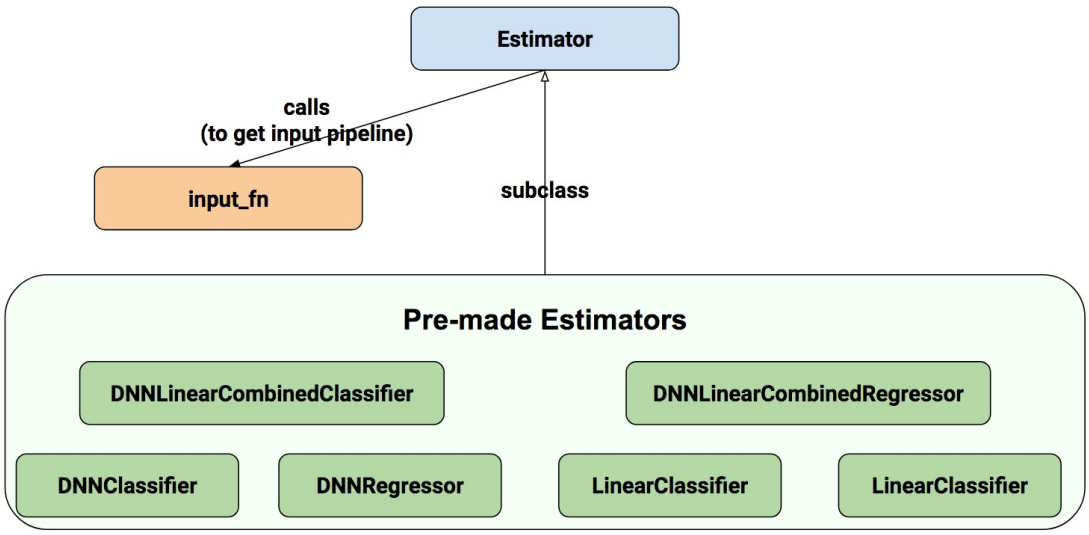
估算器介绍

估算器是一种高级 API，使用这种 API，您在训练 TensorFlow 模型时就不再像之前那样需要编写大量的样板文件代码。估算器也非常灵活，如果您对模型有具体的要求，它允许您替换默认行为。

使用估算器，您可以通过两种可能的方式构建模型：

- 预制估算器 - 这些是预先定义的估算器，旨在生成特定类型的模型。在这篇博文中，我们将使用 DNNClassifier 预制估算器。
- 估算器（基类）- 允许您使用 model_fn 函数完全掌控模型的创建方式。我们将在单独的博文中介绍如何操作。

下面是估算器的类图：



我们希望在未来版本中添加更多的预制估算器。

正如您所看到的，所有估算器都使用 `input_fn`，它为估算器提供输入数据。在我们的示例中，我们将重用 `my_input_fn`，这个函数是我们专门为演示定义的。

下面的代码可以将预测鸢尾花类型的估算器实例化：

```
# Create the feature_columns, which specifies the input to our model.
# All our input features are numeric, so use numeric_column for each 
feature_columns = [tf.feature_column.numeric_column(k) for k in featu

# Create a deep neural network regression classifier.
# Use the DNNClassifier pre-made estimator
classifier = tf.estimator.DNNClassifier(
   feature_columns=feature_columns, # The input features to our model
   hidden_units=[10, 10], # Two layers, each with 10 neurons
   n_classes=3,
   model_dir=PATH) # Path to where checkpoints etc are stored
```

我们现在有了一个可以开始训练的估算器。

训练模型

使用一行 TensorFlow 代码执行训练：

```
# Train our model, use the previously function my_input_fn
# Input to training is a file with training example
# Stop training after 8 iterations of train data (epochs)
classifier.train(
   input_fn=lambda: my_input_fn(FILE_TRAIN, True, 8))
```

不过，等一等... 这个“`lambda: my_input_fn(FILE_TRAIN, True, 8)`”是什么？这是我们将数据集与估算器连接的位置！估算器需要数据来执行训练、评估和预测，它使用 `input_fn` 提取数据。估算器需要一个没有参数的 `input_fn`，因此我们将使用 `lambda` 创建一个没有参数的函数，这个函数会使用所需的参数 `file_path, shuffle setting,` 和 `repeat_count` 调用 input_fn。在我们的示例中，我们使用 `my_input_fn,`，并向其传递：

- `FILE_TRAIN`，训练数据文件。
- `True`，告知估算器打乱数据。
- `8`，告知估算器将数据集重复 8 次。

评估我们经过训练的模型

好了，我们现在有了一个经过训练的模型。如何评估它的性能呢？幸运的是，每个估算器都包含一个 `evaluate` 函数：

```
# Evaluate our model using the examples contained in FILE_TEST
# Return value will contain evaluation_metrics such as: loss & averag
evaluate_result = estimator.evaluate(
   input_fn=lambda: my_input_fn(FILE_TEST, False, 4)
print("Evaluation results")
for key in evaluate_result:
   print("   {}, was: {}".format(key, evaluate_result[key]))
```

在我们的示例中，我们达到了 93% 左右的准确率。当然，可以通过多种方式提高准确率。一种方式是重复运行程序。由于模型的状态将持久保存（在上面的 `model_dir=PATH` 中），您对它训练的迭代越多，模型改进得越多，直至产生结果。另一种方式是调整隐藏层的数量或每个隐藏层中节点的数量。您可以随意调整；不过请注意，在进行更改时，您需要移除在 `model_dir=PATH` 中指定的目录，因为您更改的是 `DNNClassifier` 的结构。

使用我们经过训练的模型进行预测

大功告成！我们现在已经有一个经过训练的模型了，如果我们对评估结果感到满意，可以使用这个模型根据一些输入来预测鸢尾花。与训练和评估一样，我们使用一个函数调用进行预测：

```
# Predict the type of some Iris flowers.
# Let's predict the examples in FILE_TEST, repeat only once.
predict_results = classifier.predict(
    input_fn=lambda: my_input_fn(FILE_TEST, False, 1))
print("Predictions on test file")
for prediction in predict_results:
   # Will print the predicted class, i.e: 0, 1, or 2 if the predictio
   # is Iris Sentosa, Vericolor, Virginica, respectively.
   print prediction["class_ids"][0]
```

基于内存中的数据进行预测

之前展示的代码将 `FILE_TEST` 指定为基于文件中存储的数据进行预测，不过，如何根据其他来源（例如内存）中的数据进行预测呢？正如您可能猜到的一样，进行这种预测不需要对我们的 `predict` 调用进行更改。不过，我们需要将 Dataset API 配置为使用如下所示的内存结构：

```
# Let create a memory dataset for prediction.
# We've taken the first 3 examples in FILE_TEST.
prediction_input = [[5.9, 3.0, 4.2, 1.5],  # -> 1, Iris Versicolor
                    [6.9, 3.1, 5.4, 2.1],  # -> 2, Iris Virginica
```

```
                    [5.1, 3.3, 1.7, 0.5]]  # -> 0, Iris Sentosa
def new_input_fn():
   def decode(x):
       x = tf.split(x, 4) # Need to split into our 4 features
       # When predicting, we don't need (or have) any labels
       return dict(zip(feature_names, x)) # Then build a dict from th

   # The from_tensor_slices function will use a memory structure as i
   dataset = tf.contrib.data.Dataset.from_tensor_slices(prediction_in
   dataset = dataset.map(decode)
   iterator = dataset.make_one_shot_iterator()
   next_feature_batch = iterator.get_next()
   return next_feature_batch, None # In prediction, we have no labels

# Predict all our prediction_input
predict_results = classifier.predict(input_fn=new_input_fn)

# Print results
print("Predictions on memory data")
for idx, prediction in enumerate(predict_results):
   type = prediction["class_ids"][0] # Get the predicted class (index
   if type == 0:
       print("I think: {}, is Iris Sentosa".format(prediction_input[i
   elif type == 1:
       print("I think: {}, is Iris Versicolor".format(prediction_inpu
   else:
       print("I think: {}, is Iris Virginica".format(prediction_input
```

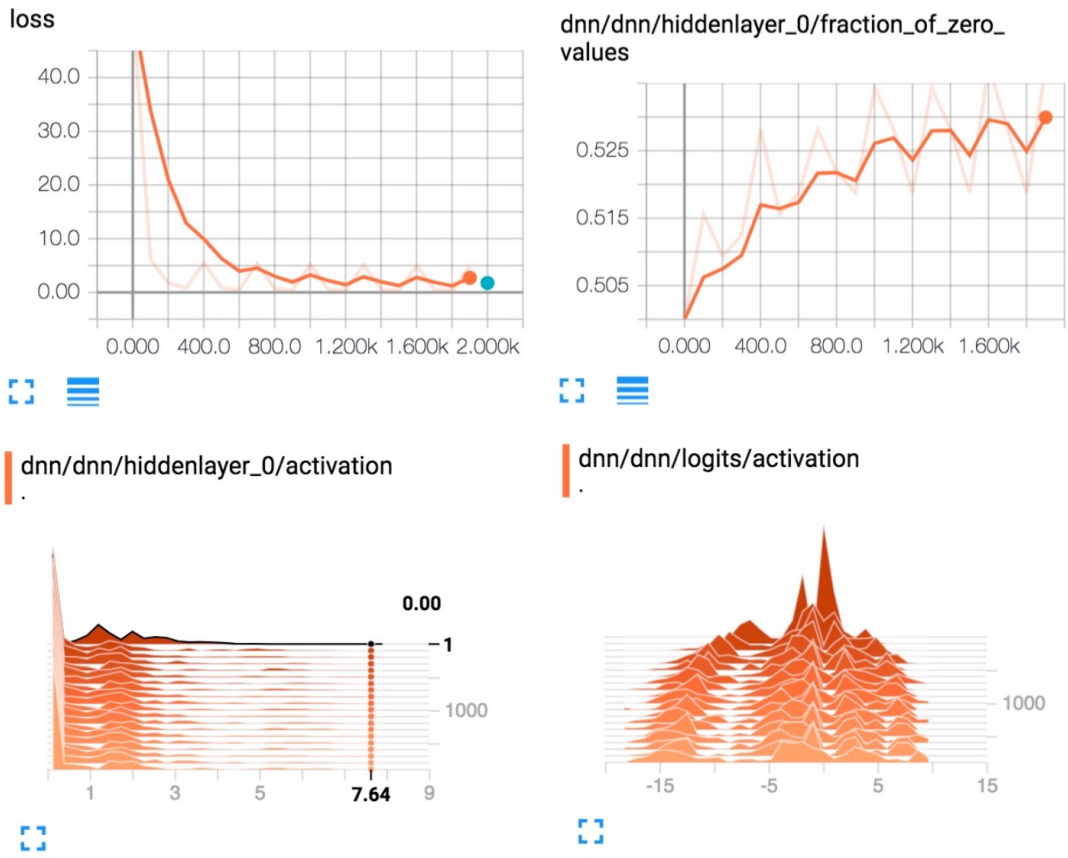
`Dataset.from_tensor_slides()` 面向可以装入内存的小数据集。按照与训练和评估时相同的方式使用 `TextLineDataset` 时，只要您的内存可以管理随机缓冲区和批次大小，您就可以处理任意大的文件。

拓展

使用像 DNNClassifier 一样的估算器可以提供很多值。除了易于使用外，预制估算器还提供内置的评估指标，并创建您可以在 TensorBoard 中看到的汇总。要查看此报告，请按照下面所示从您的命令行启动 TensorBoard：

```
# Replace PATH with the actual path passed as model_dir argument when
# DNNRegressor estimator was created.
tensorboard --logdir=PATH
```

下面几个图显示了 TensorBoard 将提供的一些数据：



总结

在这篇博文中，我们探讨了数据集和估算器。这些是用于定义输入数据流和创建模型的重要 API，因此花一些时间来学习它们非常值得！

如需了解更多详情，请参阅下面的资源

- 这篇博文使用的完整源代码在这里。
- Josh Gordon 有关这个问题非常不错的 Jupyter 笔记。使用

这个笔记，您可以学习如何运行具有不同类型特征（输入）的更丰富示例。正如您从我们的模型中发现的一样，我们仅仅使用了数值特征。

- 对于数据集，请参阅程序员指南和参考文档中的新章节。
- 对于估算器，请参阅程序员指南和参考文档中的新章节。

到这里还没有完。我们很快就会发布更多介绍这些 API 工作方式的博文，敬请关注！

在此之前，祝大家尽情享受 TensorFlow 编码！