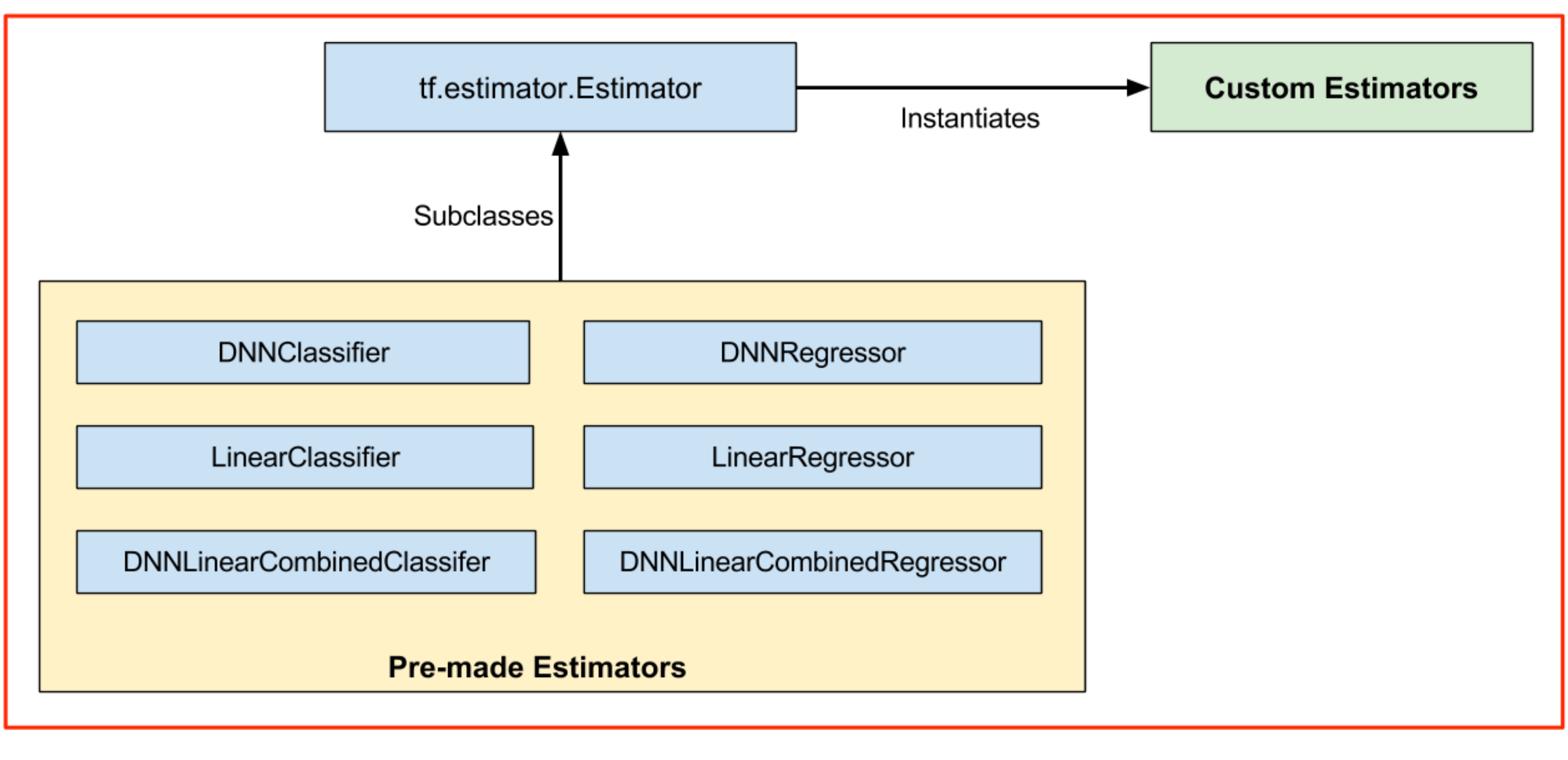
自定义Estimator

1、预创建的 Estimator 与自定义 Estimator

- 预创建的 Estimator 是 tf.estimator.Estimator 基类的子类
- 自定义 Estimator 是 tf.estimator.Estimator 的实例



- 自定义是为了更好地控制Estimator：
    - 例如，连接隐藏层
    - 例如，计算独特的指标
- 模型函数（即 model_fn）会实现机器学习算法
- 唯一区别：
    - 预创建 Estimator，别人已经写好的编写了模型函数
    - 自定义 Estimator，需要自行编写模型函数

2、编写输入函数

```
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.
    dataset = dataset.shuffle(1000).repeat().batch(batch_size)

    # Return the read end of the pipeline.
    return dataset.make_one_shot_iterator().get_next()
```

3、创建特征列

4、编写模型函数（my_model)

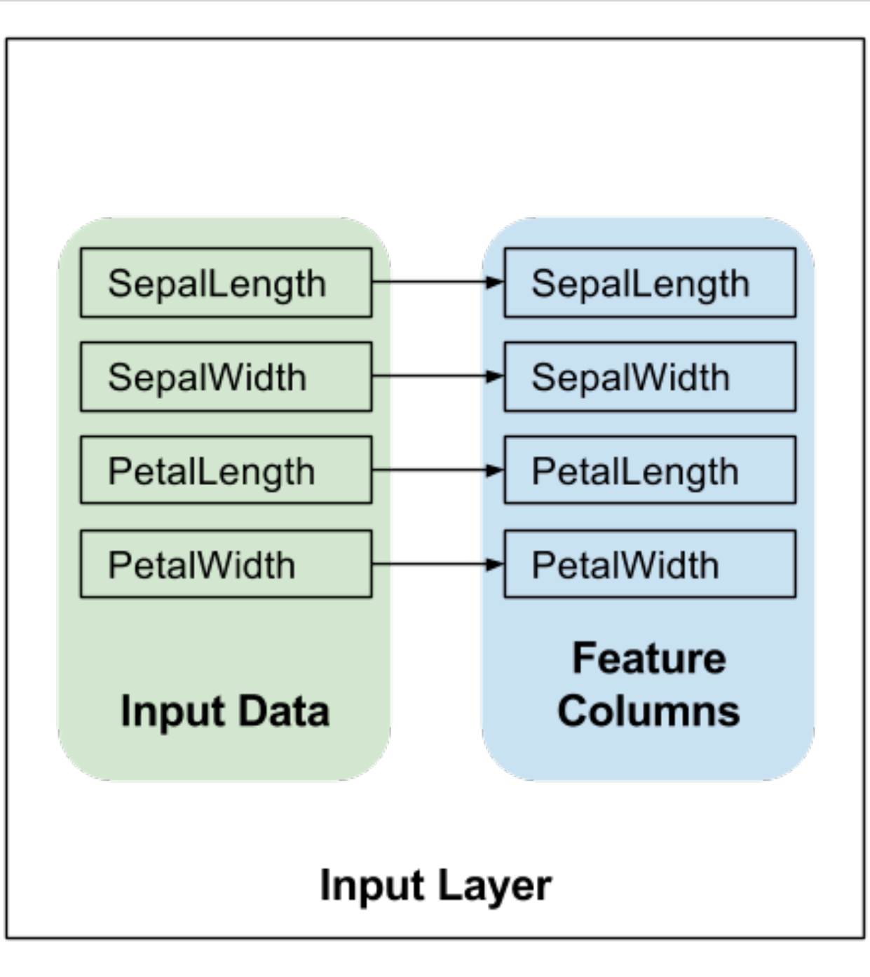
5、定义模型

基本的深度神经网络模型必须定义下列三个部分：

- 一个输入层
- 一个或多个隐藏层
- 一个输出层

（1）定义输入层

```
    # Use `input_layer` to apply the feature columns.
    net = tf.feature_column.input_layer(features, params['feature_columns'])
```



（2）隐藏层

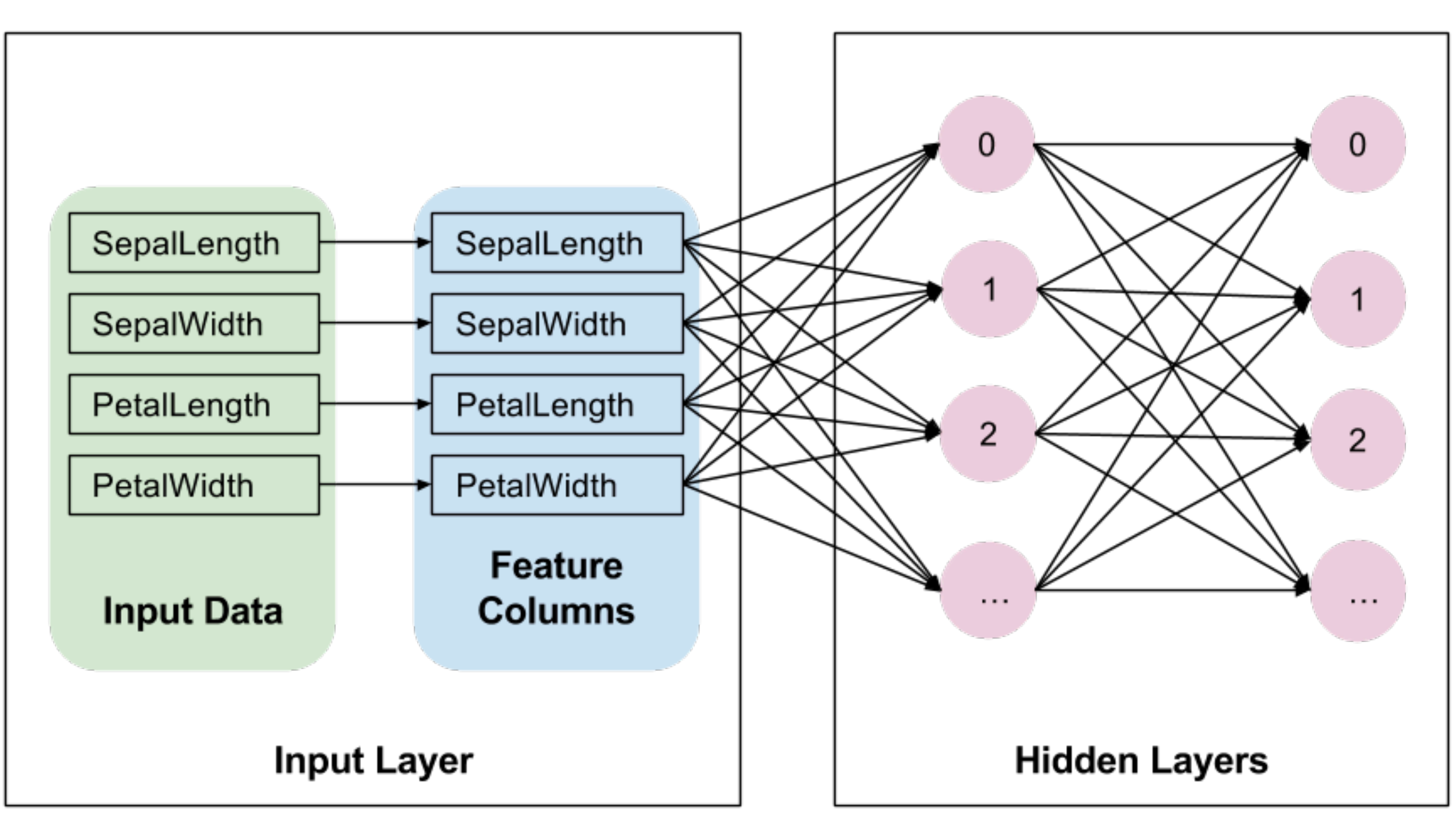
- Layers API 提供一组丰富的函数来定义所有类型的隐藏层
    - 卷积层：convolutional
    - 池化层：pooling
    - 丢弃层：dropout

- tf.layers.dense：全连接层

```
    # Build the hidden layers, sized according to the 'hidden_units' param.
    for units in params['hidden_units']:
        net = tf.layers.dense(net, units=units, activation=tf.nn.relu)
```

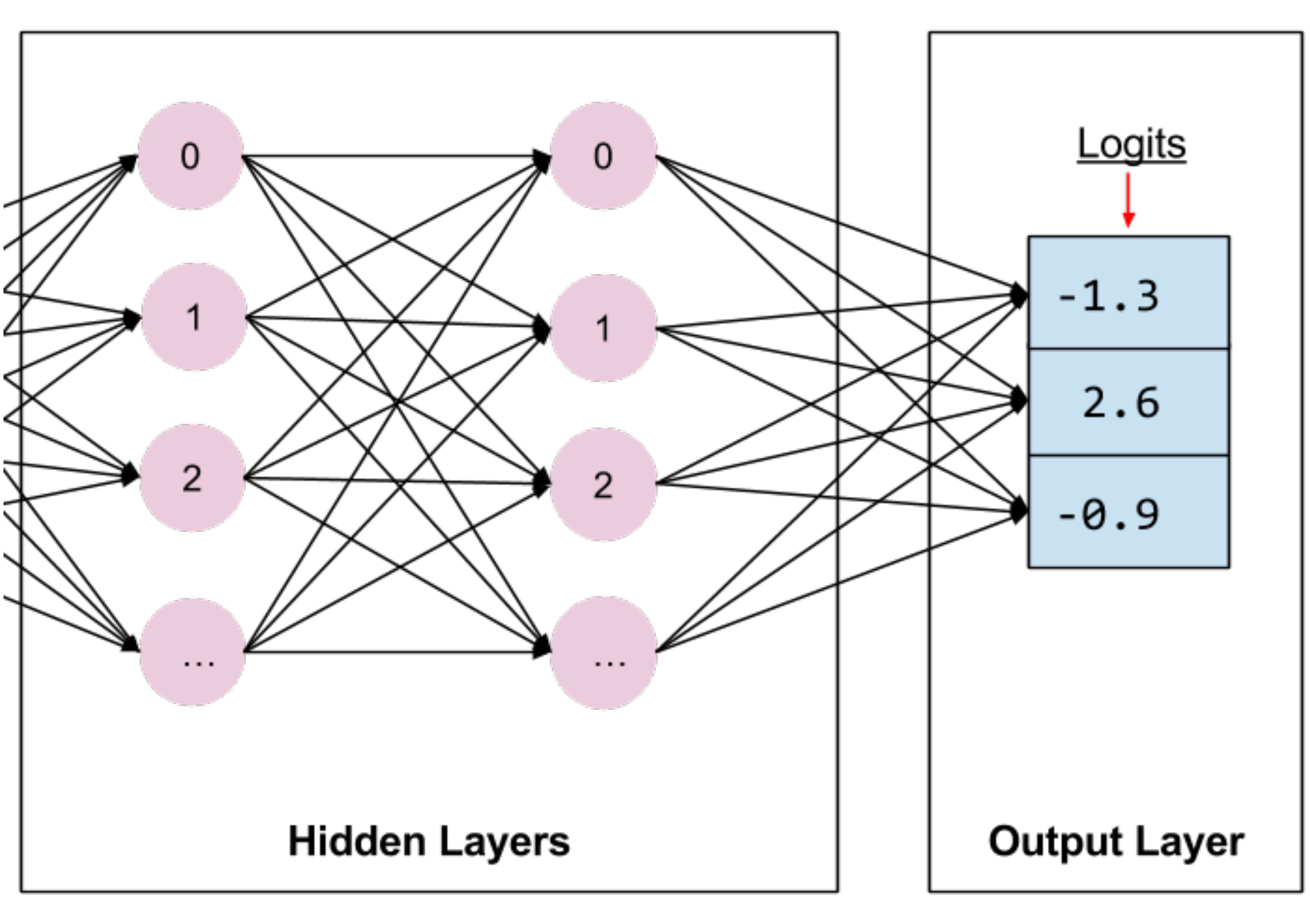
- units 参数：指定层中输出神经元的数量
- activation 参数：定义激活函数（如 Relu）

- 这里的变量 net 表示网络的当前顶层。在第一次迭代中，net 表示输入层。在每次循环迭代时，tf.layers.dense 使用变量 net 创建一个新层，该层将前一层的输出作为其输入。



（3）输出层

```
    # Compute logits (1 per class).
    logits = tf.layers.dense(net, params['n_classes'], activation=None)
```



- units 参数：指定输出的数量
- tf.nn.softmax 函数将Logits转换为概率

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

6、实现训练、评估和预测

- 创建模型函数的最后一步是编写实现预测、评估和训练的分支代码
- 当调用 train、evaluate 或 predict 时，Estimator 框架会调用模型函数并将 mode 参数设置为如下所示的值：

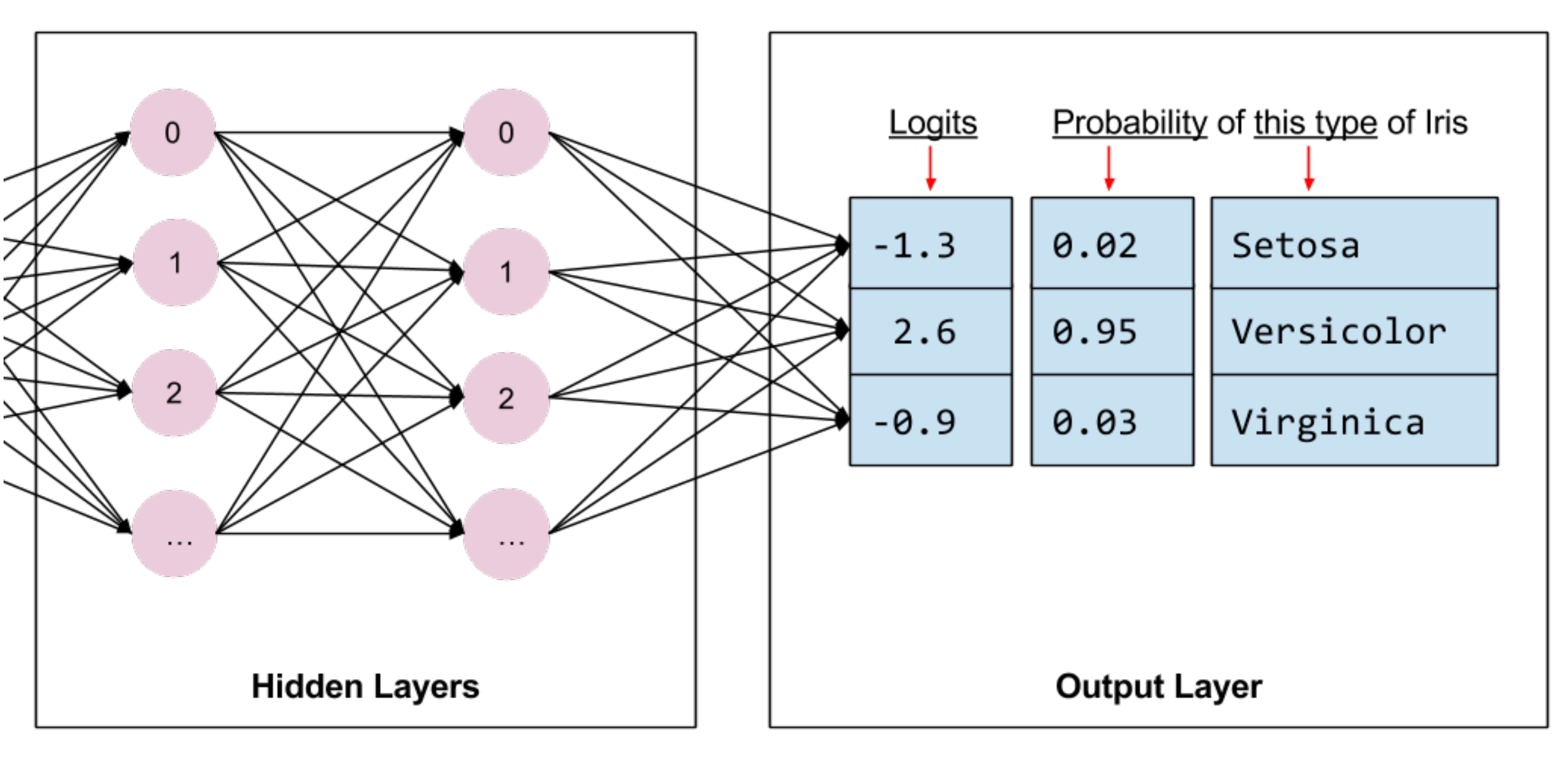
Estimator 方法	Estimator 模式
`train()`	`ModeKeys.TRAIN`
`evaluate()`	`ModeKeys.EVAL`
`predict()`	`ModeKeys.PREDICT`

- 模型函数必须提供代码来处理全部三个 mode 值
- 对于每个 mode 值，代码都必须返回 tf.estimator.EstimatorSpec 的一个实例，其中包含调用程序需要的信息

（1）预测

```
# Compute predictions.
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)
```

- 返回内容：



- predictions 存储的是下列三个键值对：
    - class_ids 存储的是类别 ID（0、1 或 2）
    - probabilities 存储的是三个概率（在本例中，分别是 0.02、0.95 和 0.03）
    - logit 存储的是原始对数值（在本例中，分别是 -1.3、2.6 和 -0.9）

（2）计算损失

- 训练和评估都需要计算损失，这是要进行优化的目标
- tf.losses.sparse_softmax_cross_entropy：针对整个批次返回平均值

```
# Compute loss.
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
```

（3）评估

- 模型函数必须返回一个tf.estimator.EstimatorSpec，包含：
    - 模型损失
    - 一个或多个指标（返回指标是可选的，但通常会返回至少一个）
- 指标模块：tf.metrics
    - tf.metrics.accuracy：准确率

```
# Compute evaluation metrics.
accuracy = tf.metrics.accuracy(labels=labels,
                               predictions=predicted_classes,
                               name='acc_op')
```

- 针对评估返回的 EstimatorSpec 通常包含以下信息：
    - loss：模型的损失
    - eval_metric_ops：可选的指标字典

- 创建一个包含指标的字典。如果我们计算了其他指标，则将这些指标作为附加键值对添加到同一字典中。然后，我们将在 tf.estimator.EstimatorSpec 的 eval_metric_ops 参数中传递该字典。具体代码如下：

```
metrics = {'accuracy': accuracy}
tf.summary.scalar('accuracy', accuracy[1])

if mode == tf.estimator.ModeKeys.EVAL:
    return tf.estimator.EstimatorSpec(
        mode, loss=loss, eval_metric_ops=metrics)
```

- tf.summary.scalar 会在 TRAIN 和 EVAL 模式下向 TensorBoard 提供准确率

（4）训练

- 模型函数必须返回一个 EstimatorSpec，包含：
    - 损失
    - 训练指令
- 构建训练指令需要优化器：tf.train.AdagradOptimizer

```
optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
```

- 被模仿的 DNNClassifier也默认使用 Adagrad
- tf.train 文件包提供很多其他优化器

- 使用优化器的 minimize 方法，根据我们之前计算的损失构建训练指令

```
train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
```

- minimize 方法还具有 global_step 参数。TensorFlow 使用此参数来计算已经处理过的训练步数（以了解何时结束训练）。此外，global_step 对于 **TensorBoard** 图能否正常运行至关重要。

- 针对训练返回的 EstimatorSpec 必须设置了下列字段：
    - loss：损失函数的值
    - train_op：执行训练步

```
return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

7、自定义Estimator

- 通过 Estimator 基类实例化自定义 Estimator，如下所示：

```
    # Build 2 hidden layer DNN with 10, 10 units respectively.
    classifier = tf.estimator.Estimator(
        model_fn=my_model,
        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,
        })
```

- 执行训练：

```
# Train the Model.
classifier.train(
    input_fn=lambda:iris_data.train_input_fn(train_x, train_y, args.batch_size),
    steps=args.train_steps)
```

8、TensorBoard

- 查看训练结果报告
- 启动TensorBoard：

```
# Replace PATH with the actual path passed as model_dir
tensorboard --logdir=PATH
```

- 打开 TensorBoard：http://localhost:6006
- 所有预创建的 Estimator 都会自动将大量信息记录到 TensorBoard 上
- 对于自定义 Estimator，TensorBoard 只提供一个默认日志（损失图）以及明确告知 TensorBoard 要记录的信息

- TensorBoard图：
    - global_step/sec：这是一个性能指标，显示训练时每秒处理的批次数（梯度更新）
    - loss：所报告的损失
    - accuracy：准确率由下列两行记录：
        - eval_metric_ops={'my_accuracy': accuracy}）（评估期间）
        - tf.summary.scalar('accuracy', accuracy[1])（训练期间）

- 注意 my_accuracy 和 loss 图中的以下内容：
    - 橙线表示训练
    - 蓝点表示评估：评估在每次调用 evaluate 时仅在图上生成一个点。此点包含整个评估调用的平均值。它在图上没有宽度，因为它完全根据特定训练步（一个检查点）的模型状态进行评估。

9、总结

- 预创建的 Estimator 可以快速高效地创建新模型
- 自定义 Estimator 更具灵活性
- 预创建的 Estimator 和自定义 Estimator 采用相同的编程模型
- 唯一的实际区别是必须为自定义 Estimator 编写模型函数
- 更多阅读：
    - 官方 TensorFlow MNIST 实现：使用了自定义 Estimator
    - TensorFlow 官方模型代码库：其中包含更多使用自定义 Estimator 的精选示例
    - TensorBoard 视频：介绍了 TensorBoard
    - 低阶 API 简介：展示了如何直接使用 TensorFlow 的低阶 API 更轻松地进行调试

更新时间：2018年04月25日16:33:40