

`layers.fully_connected` 完全连接的深度神经网络，每一层分别有 10、20、10 个隐藏节点，通过不同层的转换和训练，得到新的数据特征。TF.Learn 里面的 `models` 模块有很多经常使用的模型（比如逻辑回归），这里我们用 `models.logistic_regression_zero_init` 加一层，以 0 作为初始参数值的逻辑回归模型，这也是深度学习里比较常用的一种方法，从而得到最后的预测值和损失值。最后，使用 `contrib.layers.optimize_loss` 函数对损失值进行优化，可以根据需要选择不同的优化函数和学习率，`optimize_loss` 会得到一个训练算子（Training Operator），在每次训练迭代时会被用来优化模型的参数和决定模型发展的方向。这个自定义模型函数需要返回一些要求的值，比如预测值及预测概率、损失值和训练算子。读者可以比较灵活地使用 Python 的字典来返回预测值及预测概率，也可以只返回预测值和预测概率中的一个，这样做的主要目的是在之后能够更方便地使用 `estimator.predict` 函数。

接下来，我们把定义好的模型运用到比较常用的 iris 数据进行分类。

```
from sklearn import datasets, cross_validation

iris = datasets.load_iris()
x_train, x_test, y_train, y_test = cross_validation.train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=35)

classifier = learn.Estimator(model_fn=my_model)
classifier.fit(x_train, y_train, steps=700)

predictions = classifier.predict(x_test)
```

我们利用 Scikit-learn 的 `datasets` 引入数据，并用 `cross_validation` 把数据分为训练和评估。接下来把我们定义好的 `my_model` 直接放进 `learn.Estimator` 就可以使用 Scikit-learn 风格的 `fit` 和 `predict` 函数。通过快速和简单地定义自己的模型函数，能直接利用 Estimator 的各种功能，也能够直接进行分布式模型训练，完全不用担心许多实现的细节[比如不同的线程之间的交流和主监督（Master Supervisor）的建立]。

目前我们只介绍了其中一种自定义函数签名，其他的函数签名大同小异。简单来说，模式（Mode）可以被用来定义函数的使用阶段，例如 `training`、`evaluation`，以及 `prediction`。这些常用的模式可以在 `ModeKeys` 里面找到。一些比较复杂的深度学习模型可能会包含一些特殊的层，例如 `batch normalization` 层要求一些计算只发生在训练期间，而评估期间需

要跳过那些计算，所以可以在自定义函数里加一些条件语句来实现这样的复杂逻辑。params 是可以由自定义模型来调节的参数，读者使用 fit 函数时可以给更多的参数。具体细节请参考 TF.Learn 的官方文档。

### 10.1.2 建立自己的机器学习 Estimator

10.1.1 节我们简单介绍了怎样用自定义的模型使用 Estimator，接下来，我们来了解 Estimator 的一些基本架构及如何通过实现自己的 Estimator 子类建立自己的机器学习分布式 Estimator。

BaseEstimator 是最抽象也是最基本的实现 TensorFlow 模型的训练和评估的类。它提供了许多简单易用的功能，比如用 fit() 对模型进行训练，用 partial\_fit() 进行线上训练，用 evaluate() 评估模型，用 predict() 使用模型并对新的数据进行预测，等等。它利用了许多包含在 graph\_actions 里很复杂的逻辑进行模型的训练和预测。前面章节简单提到过它包含了许多类似 Supervisor、Coordinator、QueueRunner 的使用，从而使它能够进行分布式地训练和预测。它也使用了许多 learn.DataFeeder 或者 learn.DataFrame 的类来自动识别、处理和迭代不用类型的数据。再加上 estimators.tensor\_signature 的帮助对数据进行兼容性的判断[比如稀疏张量（Sparse Tensor）]，使数据的读入更加方便和稳定。与此同时，BaseEstimator 也对 learn.monitors 及模型的存储等进行了初始化设置。learn.monitors 是用来监测模型的训练的，在接下来的章节里我们也会对它进行简单的介绍。

虽然 BaseEstimator 已经提供了大多数建立和评估模型要求的逻辑，但它却把 \_get\_train\_ops()、\_get\_eval\_ops() 和 \_get\_predict\_ops() 等实现留给了它的子类，从而让它的子类能够更自由地实现自定义的一些逻辑处理。10.1.1 节中我们使用到的 Estimator 刚好提供了怎样实现 BaseEstimator 那些未实现方法的样本。

我们以 Estimator 为例，它的 \_get\_train\_ops() 接受 features 和 targets 为参数，使用自定义的模型函数返回一个 Operation 和损失 Tensor 的 Tuple，这个函数会被用在每个训练迭代时对模型的参数进行优化。如果想实现自己的 Estimator，你有绝对的自由来决定训练的逻辑。例如，如果想实现一个非监督学习模型的 Estimator，那么可以在这个函数里对 targets 进行忽略。

和 \_get\_train\_ops() 类似，\_get\_eval\_ops() 让 BaseEstimator 的子类来使用自定义的 metrics 评估每个模型训练的迭代。在 TensorFlow 高阶的模块里，比如 contrib.metrics，可以找到许多直接使用的 metrics，第 11 章会对这个模块进行简单的介绍。自定义的 metrics

函数需要返回一个 Tensor 对象的 Python 字典来代表评估 Operation，每次迭代时都会被用到。以下是 Estimator 的 `_get_train_ops()` 的实现：

```
predictions, loss, _ = self._call_model_fn(features, targets, ModeKeys.EVAL)
result = {'loss': contrib.metrics.streaming_mean(loss)}
```

先用到自定义的模型对新的数据进行预测和计算损失值，用 `ModeKeys` 中的 `EVAL` 表明这个函数只会在评估时被用到，然后用到了 `contrib.metrics` 模块里的 `streaming_mean` 对 `loss` 计算平均流，也就是在之前计算过的平均值基础上加上这次迭代的损失值再计算平均值。

`_get_predict_ops()` 是用来实现自定义的预测的，例如在这个函数里可以对预测的结果进行进一步的处理。再比如，把预测概率转换成简单的预测结果，把概率进行平滑加工（Smoothing），等等。这个函数需要返回一个 Tensor 对象的 Python 字典来代表预测 Operation。一旦这个函数被实现，就可以很轻松地使用 Estimator 的 `predict()` 函数，充分利用 Estimator 的分布式功能，完全不用担心一些复杂的内部实现逻辑。如果想建立非监督模型，也可以很快地在这个基础之上实现一个类似 Scikit-learn 里面的 `transform()` 函数。

在 TF.Learn 的模块里也可以找到许多自定义机器学习 Estimator 的例子，例如逻辑回归（`LogisticRegressor`）。由于 Estimator 已经提供了绝大部分需要的实现，`LogisticRegressor` 只需要提供自己的 metrics（例如 AUC、accuracy、precision，以及 recall，只用来处理二分类的问题），所以可以很快地在 `LogisticRegressor` 的基础上写一个子类来实现一个更个性化的二分类的 Estimator，完全不需要担心其他逻辑的实现。

TF.Learn 里的随机森林模型 `TensorForestEstimator` 把许多很细节的实现放到了 `contrib.tensor_forest` 里，只利用和暴露一些比较高阶的，需要用到的成分到 `TensorForestEstimator` 里，这样用户就能更轻松地使用这个高阶机器学习模块。下面的代码中，它所有的超参数都通过 `contrib.tensor_forest.ForestHParams` 被传到构造函数的 `params` 里，然后在构造函数里使用 `params.fill()` 建造随机森林的 TensorFlow 图，也就是 `tensor_forest.RandomForestGraphs`。

```
class TensorForestEstimator(estimator.BaseEstimator):
    """An estimator that can train and evaluate a random forest."""

    def __init__(self, params, device_assigner=None, model_dir=None,
```

```

graph_builder_class=tensor_forest.RandomForestGraphs,
master='', accuracy_metric=None,
tf_random_seed=None, config=None):
    self.params = params.fill()
    self.accuracy_metric = (accuracy_metric or
        ('r2' if self.params.regression else 'accuracy'))
    self.data_feeder = None
    self.device_assigner = (
        device_assigner or tensor_forest.RandomForestDeviceAssigner())
    self.graph_builder_class = graph_builder_class
    self.training_args = {}
    self.construction_args = {}

super(TensorForestEstimator, self). __init__(model_dir=model_dir,
                                             config=config)

```

由于很多实现太复杂而且通常需要非常有效率，它的很多细节都用 C++实现了单独的 Kernel。它的 \_get\_predict\_ops() 函数首先使用 tensor\_forest 内部 C++ 实现的 data\_ops.ParseDataTensorOrDict() 函数检测和转换读入的数据到可支持的数据类型，然后利用 RandomForestGraphs 的 inference\_graph 函数得到预测的 Operation。

```

def _get_predict_ops(self, features):
    graph_builder = self.graph_builder_class(
        self.params, device_assigner=self.device_assigner, training=False,
        **self.construction_args)
    features, spec = data_ops.ParseDataTensorOrDict(features)
    _assert_float32(features)
    return graph_builder.inference_graph(features, data_spec= spec)

```

类似地，它的 \_get\_train\_ops() 和 \_get\_eval\_ops() 函数分别调用了 RandomForestGraphs.training\_loss() 和 RandomForestGraphs.inference\_graph() 函数，它使用了 data\_ops.ParseDataTensorOrDict 和 data\_ops.ParseLabelTensorOrDict 分别检测和转换 features 和 targets 到可兼容的数据类型。

希望以上关于 Estimator 架构的介绍和几个例子能够帮助读者更好地了解 Estimator。

一旦读者建立好了自己的机器学习 Estimator 或者准备好使用 Estimator，可以轻松地在多台机器上、多个服务器上进行分布式的模型训练，10.1.3 节会介绍 RunConfig 来帮助读者更好地调节程序运行时参数。

### 10.1.3 调节 RunConfig 运行时参数

RunConfig 是 TF.Learn 里的一个类，用来帮助用户调节程序运行时参数，例如用 num\_cores 选择使用的核的数量，用 num\_ps\_replicas 调节参数服务器的数量，用 gpu\_memory\_fraction 控制使用的 GPU 存储的百分比，等等。

值得注意的是，RunConfig 里 master 这个参数是用来指定训练模型的主服务器地址的，task 是用来设置任务 ID 的，每个任务 ID 控制一个训练模型参数服务器的 replica。以下是一个例子，读者可以先初始化一个 RunConfig 对象，再把这个对象传进 Estimator 里。

```
config = tf.contrib.learn.RunConfig(task=0, master="",
                                     gpu_memory_fraction=0.8)
est = tf.contrib.learn.Estimator(model_fn=custom_model, config=config)
```

以上例子是使用 RunConfig 参数的默认值在本地运行一个简单的模型，只使用一个任务 ID 和 80% 的 GPU 存储作为参数传进 Estimator 里。当读者运行时，这些运行时参数会被自动运用上，不用担心 ConfigProto、GPUOptions 之类的使用细节。读者可以快速地改变这些参数来实现分布式模型的训练及参数服务器的使用，10.1.4 节会简单介绍。注意，这些 API 未来会有改动，所以最新的使用方法请参考 TF.Learn 官方文档。

### 10.1.4 Experiment 和 LearnRunner

Experiment 是一个简单易用的建立模型实验的类，它包含了建模所需要的所有信息，例如 Estimator、训练数据、评估数据、评估指标、监督器、评估频率，等等。可以选择在当地运行，也可以和 RunConfig 配合进行分布式地试验。LearnRunner 是用来方便做实验的一个模块。接下来我们举个简单的例子说明。

先用 `tf.app.flags` 定义一些可以从命令行传入的参数，例如数据、模型、输出文件的路径、训练和评估的步数等。这里有几个值得注意的参数。`schedule` 是指想做的试验类型，比如使用 `local_run()` 在当地做试验，可能的一些选项是 Experiment 里面的一些函数名字，例如 `run_std_server()` 可以在标准服务器上做试验。`master_grpc_url` 是主要的 GRPC TensorFlow 服务器。`num_parameter_servers` 是参数服务器的数量，等等。

```

flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/census-data",
                    "Directory for storing the census data data")
flags.DEFINE_string("model_dir", "/tmp/census_wide_and_deep_model",
                    "Directory for storing the model")
flags.DEFINE_string("output_dir", "", "Base output directory.")
flags.DEFINE_string("schedule", "local_run",
                    "Schedule to run for this experiment.")
flags.DEFINE_string("master_grpc_url", "",
                    "URL to master GRPC tensorflow server, e.g.,"
                    "grpc://127.0.0.1:2222")
flags.DEFINE_integer("num_parameter_servers", 0,
                     "Number of parameter servers")
flags.DEFINE_integer("worker_index", 0, "Worker index (>=0)")
flags.DEFINE_integer("train_steps", 1000, "Number of training steps")
flags.DEFINE_integer("eval_steps", 1, "Number of evaluation steps")

FLAGS = flags.FLAGS

```

接下来写一个建立 Experiment 对象的函数，在这个函数里首先使用之前设置好的一些 FLAGS 建立好 RunConfig 及想要建立的机器学习模型 Estimator，这里我们建立广度深度结合分类器（DNNLinearCombinedClassifier）。注意，我们省略了 input\_train\_fn 和 input\_test\_fn 的定义，这两个方程会定义数据的来源、提供训练，以及评估所用的数据。我们在接下来的机器学习 Estimator 里都会用到。不同的数据有不同的导入方法，在这里我们就不详细介绍了。

```

def create_experiment_fn(output_dir):
    config = run_config.RunConfig(master=FLAGS.master_grpc_url,
                                   num_ps_replicas=FLAGS.num_parameter_servers,
                                   task=FLAGS.worker_index)

    estimator = tf.contrib.learn.DNNLinearCombinedClassifier(
        model_dir=FLAGS.model_dir,
        linear_feature_columns=wide_columns,

```

```

        dnn_feature_columns=deep_columns,
        dnn_hidden_units=[5],
        config=config)
return tf.contrib.learn.Experiment(
    estimator=estimator,
    train_input_fn=data_source.input_train_fn,
    eval_input_fn=data_source.input_test_fn,
    train_steps=FLAGS.train_steps,
    eval_steps=FLAGS.eval_steps)

```

然后就可以把 `create_experiment_fn()` 函数传入 `LearnRunner` 里进行不同类型的试验，例如当地试验或者服务器试验，以及把试验的结果存储到不同的路径中，代码如下。

```

learn_runner.run(experiment_fn=create_experiment_fn,
                 output_dir=FLAGS.output_dir,
                 schedule=FLAGS.schedule)

```

## 10.2 深度学习 Estimator

TF.Learn 里包含了许多深度学习 Estimator 的实现，高阶的 API 让用户使用起来更方便。本节介绍一些基本的高阶深度学习 API 及它们和 TensorFlow 其他模块结合使用的例子。

### 10.2.1 深度神经网络

TF.Learn 里包含简单易用的深度神经网络 Estimator，例如分类问题可以使用 `DNNClassifier`，下面我们介绍一个最简单的例子。先在 `_input_fn()` 里建立数据，这里使用 `layers` 模块建立两个特征列——年龄和语言（后面我们将详细介绍它们的使用方法）。

```

def _input_fn(num_epochs=None):
    features = {'age': tf.train.limit_epochs(tf.constant([[.8],[.2],[.1]]),
                                             num_epochs=num_epochs),
                'language': tf.SparseTensor(values=['en', 'fr', 'zh'],
                                            indices=[[0, 0],[0, 1],[2, 0]],
                                            shape=[3, 2])}

```

```

    return features, tf.constant([[1], [0], [0]], dtype=tf.int32)

language_column = tf.contrib.layers.sparse_column_with_hash_bucket(
    'language', hash_bucket_size=20)
feature_columns = [
    tf.contrib.layers.embedding_column(language_column, dimension=1),
    tf.contrib.layers.real_valued_column('age')
]

```

接着就把特征列、每层的隐藏神经单元数、标识类别数等传入 DNNClassifier 里来迅速地建立我们的深度神经网络模型。

```

classifier = tf.contrib.learn.DNNClassifier(
    n_classes=2,
    feature_columns=feature_columns,
    hidden_units=[3, 3],
    config=tf.contrib.learn.RunConfig(tf_random_seed=1))

```

然后使用我们习惯的 fit()、evaluate() 等方法进行模型的训练和评估。

```

classifier.fit(input_fn=_input_fn, steps=100)
scores = classifier.evaluate(input_fn=_input_fn, steps=1)

```

在许多实际应用中，每行数据都有它们的权重，比如在图片分类运用中，每张图片的标识来自于不同的标识者，它们的可信度不一样，所以每张图片的标识权重也不同。在 DNNClassifier 中，我们可以指定一列为权重列，然后它会帮我们自动分配到训练过程中去。在以下的例子中，我们建立四行数据，每行有不同的权重，我们先把权重列和特征列放在 features 里面。

```

def _input_fn_train():
    target = tf.constant([[1], [0], [0], [0]])
    features = {
        'x': tf.ones(shape=[4, 1], dtype=tf.float32),
        'w': tf.constant([[100.], [3.], [2.], [2.]])
    }
    return features, target

```

然后就可以在 DNNClassifier 中表明权重列的列名，在这里也就是 w，然后表明特征列的列名 x（注意：我们需要将 x 转换为特征列）。

```
classifier = tf.contrib.learn.DNNClassifier(
    weight_column_name='w',
    feature_columns=[tf.contrib.layers.real_valued_column('x')],
    hidden_units=[3, 3],
    config=tf.contrib.learn.RunConfig(tf_random_seed=3))

classifier.fit(input_fn=_input_fn_train, steps=100)
```

我们也可以传入进我们自定义的 metrics 方程\_my\_metric\_op(), 需要做的就是操作 predictions 和 targets 进行我们心目中的 metrics 计算，此处只考虑二分类的问题，使用 tf.slice() 剪切 predictions 的第二列当作最终的预测值。

```
def _input_fn_train():
    target = tf.constant([[1], [0], [0], [0]])
    features = {'x': tf.ones(shape=[4, 1], dtype=tf.float32),}
    return features, target

def _my_metric_op(predictions, targets):
    predictions = tf.slice(predictions, [0, 1], [-1, 1])
    return tf.reduce_sum(tf.mul(predictions, targets))
```

这里我们举个例子来帮助理解 tf.slice()，假设我们有以下矩阵。

```
input = [[[1, 1, 1], [2, 2, 2]],
         [[3, 3, 3], [4, 4, 4]],
         [[5, 5, 5], [6, 6, 6]]]
```

tf.slice()需要传入输入矩阵 input，剪切的开始元素 begin，以及剪切的 Tensor 的形状 size，size[i]代表了第 i 个维度想剪切的矩阵的 shape，例如 tf.slice(input, [1, 0, 0], [1, 1, 3])可以得到[[[3, 3, 3]]]; tf.slice(input, [1, 0, 0], [1, 2, 3])可以得到[[[3, 3, 3], [4, 4, 4]]]。

我们根据需求任意地在 predictions 和 targets 上操作来实现想要的 metrics 计算，然后就可以在 evaluate()时传入自己定义好的 metrics 函数，TF.Learn 会根据你所指示的 metrics

评估模型。

```
classifier = tf.contrib.learn.DNNClassifier(
    feature_columns=[tf.contrib.layers.real_valued_column('x')],
    hidden_units=[3, 3],
    config=tf.contrib.learn.RunConfig(tf_random_seed=1))

classifier.fit(input_fn=_input_fn_train, steps=100)

scores = classifier.evaluate(
    input_fn=_input_fn_train,
    steps=100,
    metrics={

        'my_accuracy': tf.contrib.metrics.streaming_accuracy,
        ('my_precision', 'classes'): tf.contrib.metrics.streaming_precision,
        ('my_metric', 'probabilities'): _my_metric_op})
```

值得注意的是，我们可以在 `evaluate()` 时提供多个 `metrics`，其中一个 `_my_metric_op` 是我们之前自定义好的，其他两个是 `tf.contrib` 里自带的，之后的章节中也会简单提到一些内建的 `metrics` 的用法。

我们也可以在提供 `optimizer` 时提供自己定义的函数，例如，可以定义自己的优化函数来包含指数递减的学习率。

```
def optimizer_exp_decay():
    global_step = tf.contrib.framework.get_or_create_global_step()
    learning_rate = tf.train.exponential_decay(
        learning_rate=0.1, global_step=global_step,
        decay_steps=100, decay_rate=0.001)
    return tf.train.AdagradOptimizer(learning_rate=learning_rate)
```

这里用 `tf.contrib.framework.get_or_create_global_step()` 得到目前模型训练到达的全局步数，然后使用 `tf.train.exponential_decay()` 对学习率进行指数递减，这种方法在许多应用中特别常用，尤其是用来避免爆炸梯度之类的问题。

接着可以将这个自定义的优化函数放入 `DNNClassifier` 里继续使用我们熟悉的方法建

立深度神经网络分类器及它的训练，我们用 iris 数据举个例子。

```
iris = datasets.load_iris()
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.2, random_state=42)

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(
    x_train)
classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
                                             hidden_units=[10, 20, 10],
                                             n_classes=3,
                                             optimizer=optimizer_exp_decay)

classifier.fit(x_train, y_train, steps=800)
```

## 10.2.2 广度深度模型

广度深度模型的 DNNLinearCombinedClassifier 是谷歌最新研究的成果，研究团队将这个模型在 TF.Learn 里面实现，然后开源，这样更有利于其他研究者再次重复实验结果及学习。这个模型被谷歌广泛地利用在各种机器学习应用中，它是深度神经网络和逻辑回归的结合，因为在谷歌的研究中发现，将不同的特征通过两种不同的方式结合起来，更能体现应用的意义及更有效的推荐结果，这其实也和 Kaggle 竞赛中经常使用的 Ensemble 的方法比较类似。

使用的 DNNLinearCombinedClassifier 的方法和之前介绍的 DNNClassifier 及在接下来将介绍的 LinearClassifier 的使用方法类似，唯一的区别是你有更多的参数，并且可以将不同的特征列选择使用到 DNNClassifier 或者 LinearClassifier 中。

```
gender = tf.contrib.layers.sparse_column_with_keys(
    "gender", keys=["female", "male"])
education = tf.contrib.layers.sparse_column_with_hash_bucket(
    "education", hash_bucket_size=1000)
relationship = tf.contrib.layers.sparse_column_with_hash_bucket(
    "relationship", hash_bucket_size=100)
workclass = tf.contrib.layers.sparse_column_with_hash_bucket(
```

```
"workclass", hash_bucket_size=100)

wide_columns = [gender, education]
deep_columns = [relationship, workclass]

m = tf.contrib.learn.DNNLinearCombinedClassifier(
    model_dir=model_dir,
    linear_feature_columns=wide_columns,
    dnn_feature_columns=deep_columns,
    dnn_hidden_units=[100, 50])
```

我们将 gender、education、relationship、workclass 都转换为 FeatureColumn，这是特征工程中特别重要的一步。然后，将它们分为 wide\_columns 和 deep\_columns，其中 wide\_columns 将被用在 LinearClassifier 中，deep\_columns 会被用在 DNNClassifier 中，然后将它们分别传入 DNNLinearCombinedClassifier 建立广度深度模型，这样模型既具有线性特征，也具有深度神经网络特征。官方网站的 Tutorials (<https://www.tensorflow.org/tutorials/>) 上有非常有意思的例子，建议读者去学习并应用到自己的项目中。

## 10.3 机器学习 Estimator

TF.Learn 里不仅包括了许多流行的深度学习 Estimator，还包括了各种各样的机器学习算法，例如随机森林、支持向量机，等等。这让 TF.Learn 及 TensorFlow 与现有的其他软件包的界限和特色更明显。在谷歌内部的大力支持及外部开源社区的代码贡献下，相信 TF.Learn 会成为未来的分布式 Scikit-learn。接下来，我们将介绍 TF.Learn 里比较流行的机器学习高阶 API。

### 10.3.1 线性 / 逻辑回归

使用 TF.Learn 建立大家熟悉的线性或者逻辑回归非常简单，与之前提到的深度神经网络的使用方法类似。

举个简单的例子，假设我们在 input\_fn() 里建立简单的两个特征列的数据，分别是年龄和语言，以及它们的标识，这里我们用简单的常数代替阐述，使用在后面章节会提到的

特征列 API 建立稀疏的语言特征列和真值的特征列。

```
def input_fn():
    return {
        'age': tf.constant([1]),
        'language': tf.SparseTensor(values=['english'],
                                     indices=[[0, 0]],
                                     shape=[1, 1])
    }, tf.constant([[1]])

language = tf.contrib.layers.sparse_column_with_hash_bucket('language', 100)
age = tf.contrib.layers.real_valued_column('age')
```

然后就可以将这些特征列传入 LinearClassifier 里建立逻辑回归分类器，使用熟悉的 fit()、evaluate() 等函数。注意，我们可以使用 get\_variable\_names() 得到所有模型包含的变量的名称：

```
classifier = tf.contrib.learn.LinearClassifier(
    feature_columns=[age, language])
classifier.fit(input_fn=input_fn, steps=100)
classifier.evaluate(input_fn=input_fn, steps=1)['loss']
classifier.get_variable_names()
```

类似地，我们也可以像前文介绍的那样，使用自定义的优化函数，这里使用 tf.train.FtrlOptimizer() 进行优化，也可以对它进行任意改动然后传到 LinearClassifier 里：

```
classifier = tf.contrib.learn.LinearClassifier(
    n_classes=3,
    optimizer=tf.train.FtrlOptimizer(learning_rate=0.1),
    feature_columns=[feature_column])
```

### 10.3.2 随机森林

随机森林是在工业界得到广泛应用的一种机器学习算法，它是一个包含多个决策树的分类器及回归算法。在许多实际的运用中，它的效果非常好，尤其是处理不平衡的分类资料集时，它极大地平衡了误差。在许多 Kaggle 数据科学竞赛中，它的延伸版 XGBoost 更

是帮助了许多竞赛者取得了优异的成绩。

TF.Learn 里含有随机森林 Estimator，接下来我们用 iris 数据及随机森林 Estimator 进行分类。

```
hparams = tf.contrib.tensor_forest.python.tensor_forest.ForestHParams(
    num_trees=3, max_nodes=1000, num_classes=3, num_features=4)
classifier = tf.contrib.learn.TensorForestEstimator(hparams)
```

在之前的章节中讲解过 TensorForestEstimator 代码的内部架构，首先需要使用 tensor\_forest.ForestHParams 设置随机森林的参数，例如多少棵树、节点数目的上限、特征和类别的数目，等等。

```
iris = tf.contrib.learn.datasets.load_iris()
data = iris.data.astype(np.float32)
target = iris.target.astype(np.float32)
classifier.fit(x=data, y=target, steps=100)
```

然后直接传进 TensorForestEstimator 里初始化随机森林 Estimator，接下来，把数据特征列和类别列转换成 float32 的格式，这样能够保证 TensorForestEstimator 的训练更快地拟合。接下来，可以直接使用 Scikit-learn 风格的 fit() 等方法。

类似地，我们也可以把这个初始化好的 classifier 运用到 MNIST 图像数据上，这里我们从官方 tutorials 模块里导入 MNIST 数据。

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=False)
```

一般在实际应用中，随机森林容易过拟合，一种常用的防止过拟合的方法就是损失减少的速度变慢或者完全停止减少的情况下，提前停止模型的继续训练。在 TF.Learn 里，我们可以用 Monitor 模块达到这个目的。我们会在接下来的内容中仔细讲解它的各种用法，但是以下我们给出一个常用的 random\_forest 模块里自带的 LossMonitor 来迅速地达到我们的目的。我们设定每隔 100 步 Monitor 检查损失减少的速度，如果连续 100 次迭代仍然没有看见损失的减少，Monitor 会让整个模型训练停止，这样在实际应用中是非常有效的。

```
from tensorflow.contrib.learn.python.learn.estimators import random_forest

early_stopping_rounds = 100
```

```

check_every_n_steps = 100
monitor = random_forest.LossMonitor(early_stopping_rounds,
                                      check_every_n_steps)
classifier.fit(x=mnist.train.images, y=mnist.train.labels, batch_size=1000,
                monitors=[monitor])
results = estimator.evaluate(x=mnist.test.images, y=mnist.test.labels,
                             batch_size=1000)

```

### 10.3.3 K 均值聚类

K 均值聚类是非常常见的一种聚类方法，它的核心是把多维空间里的每个点划分到 K 个聚类中，使得每个点都属于离它最近的均值对应的聚类。TF.Learn 里也包含了 K 均值聚类的 Estimator，我们来看一个简单的例子。

```

import numpy as np

def make_random_centers(num_centers, num_dims):
    return np.round(np.random.rand(num_centers,
                                   num_dims).astype(np.float32) * 500)

def make_random_points(centers, num_points, max_offset=20):
    num_centers, num_dims = centers.shape
    assignments = np.random.choice(num_centers, num_points)
    offsets = np.round(np.random.randn(num_points,
                                       num_dims).astype(np.float32) * max_offset)
    return (centers[assignments] + offsets,
            assignments,
            np.add.reduce(offsets * offsets, 1))

```

以上两个函数是利用 NumPy 制造比较适合做聚类的一组数据，make\_random\_centers 函数来随机生成 num\_dims 个维度的数据集聚类的 num\_centers 个中心点，make\_random\_points 函数根据所生成的聚类中心点随便生成 num\_points 个点。我们生成二维的 10000 个点，以及 6 个随机的聚类中心点。

```
num_centers = 6
```

```

num_dims = 2
num_points = 10000
true_centers = make_random_centers(num_centers, num_dims)
points, _, scores = make_random_points(true_centers, num_points)

```

接着，可以调用 factorization 模块里 KMeans 里的一些初始化聚类的方法，例如随机初始化 RANDOM\_INIT，然后传入 RunConfig 及聚类中心数来初始化 KMeans 的 Estimator 对象，最后就可以像其他 Estimator 一样使用 Scikit-learn 风格的 fit() 和 predict()，读者可以通过 KMeans 的 clusters() 函数来看训练数据集每个点的聚类分布。

```

from tensorflow.contrib.factorization.python.ops import kmeans as kmeans_ops
from tensorflow.contrib.factorization.python.ops.kmeans import \
    KMeansClustering as KMeans
kmeans = KMeans(num_centers=num_centers,
                initial_clusters=kmeans_ops.RANDOM_INIT,
                use_mini_batch=False,
                config=RunConfig(tf_random_seed=14),
                random_seed=12)
kmeans.fit(x=points, steps=10, batch_size=8)
clusters = kmeans.clusters()

kmeans.predict(points, batch_size=128)
kmeans.score(points, batch_size=128)
kmeans.transform(points, batch_size=128)

```

值得注意的是，KMeans 的 Estimator 有多个经常用到的方法，使用 predict() 预测新的数据点的聚类，使用 score() 预测每个点和它最近的聚类的距离的总和，以及用 transform() 计算每个点和模型判断出来的聚类中心的距离。

### 10.3.4 支持向量机

支持向量机也是在机器学习应用中经常用到的一类算法，它包括使用各种不同的 kernel 或者不同的距离方程，针对不同特征的数据建立不同的线性及非线性的模型。它们有一个共同的特性就是能够同时最小化经验误差与最大化几何边缘区，所以也被称为最大边缘区分类器。在文本及图像分类等领域得到广泛的使用。TF.Learn 里面的 SVM Estimator

提供了非常简单易用的 API 来建立支持向量机模型。

我们先定义 `input_fn()` 建立一个有着两个数据特征列、一个 ID 列和一个标识列的模拟数据，然后使用 `contrib.layers` 里面的 `FeatureColumn` API 将 `feature1` 和 `feature2` 转换为方便和 `Estimator` 一起使用的 `FeatureColumn`（我们将在第 11 章中详细介绍这个功能）。

```
def input_fn():
    return {
        'example_id': tf.constant(['1', '2', '3']),
        'feature1': tf.constant([[0.0], [1.0], [3.0]]),
        'feature2': tf.constant([[1.0], [-1.2], [1.0]]),
    }, tf.constant([[1], [0], [1]])

feature1 = tf.contrib.layers.real_valued_column('feature1')
feature2 = tf.contrib.layers.real_valued_column('feature2')
```

然后就可以将这些特征列及 ID 列传入 SVM 来初始化这个支持向量机，许多参数是调节的，例如在 `l1_regularization` 和 `l2_regularization` 中加入一些正规化来防止过度拟合之类的问题，和我们之前在随机森林那一节简单提到过的问题相似，许多机器学习算法在特征列过多而例子不多的情况下很容易发生这样的情况。

```
svm_classifier = tf.contrib.learn.SVM(feature_columns=[feature1, feature2],
                                       example_id_column='example_id',
                                       l1_regularization=0.0,
                                       l2_regularization=0.0)
```

接下来就可以使用熟悉的 `fit()`、`evaluate()`、`predict()` 之类和其他 `Estimator` 共用的方法了。

```
svm_classifier.fit(input_fn=input_fn, steps=30)
metrics = svm_classifier.evaluate(input_fn=input_fn, steps=1)
loss = metrics['loss']
accuracy = metrics['accuracy']
```

## 10.4 DataFrame

TF.Learn 还包括了一个单独的 DataFrame 模块，类似于 Pandas、Spark 或者 R 编程语言里面的 DataFrame，它提供了 TF.Learn 所需的读入数据的迭代，包括读入各种数据类型，例如 pandas.DataFrame、tensorflow.Example、NumPy，等等。它包括了 FeedingQueueRunner 等功能来对数据进行分批读入，然后存在一个 Queue 里，以便 Estimator 很容易地取过去用于模型的训练。简单来说，FeedingQueueRunner 在 Estimator 训练时同时进行了更多数据的分批读入，这种多线程的方式使 Estimator 的训练更有效，也使 TF.Learn 的扩展性更强。

以 NumPy 为例，假设我们用 NumPy 的 eye() 建了一个简单的对角矩阵，然后就可以直接使用 TensorFlowDataFrame.from\_numpy() 将这个 NumPy 矩阵转换为 TensorFlow 的 DataFrame。

```
import tensorflow.contrib.learn.python.learn.dataframe.tensorflow_dataframe
as df
x = np.eye(20)
tensorflow_df = df.TensorFlowDataFrame.from_numpy(x, batch_size=10)
```

类似地，我们也可以直接像 Pandas 一样读入各种文件类型，这里我们以 csv 文件为例。

```
pandas_df = pd.read_csv(data_path)
tensorflow_df = df.TensorFlowDataFrame.from_csv([data_path], enqueue_size=20,
                                                batch_size=10, shuffle=False,
                                                default_values=default_values)
```

当使用 TensorFlowDataFrame 读入使用的文件或者数据类型之后，就可以使用 run() 制造一个数据批量 (batch) 的生成器，也就是在 Python 里经常用 yield 生成的 generator，这个生成器维持着数据列名和数据值的字典 mapping。可以调节 number\_batches 来选择生成的 batch 的数量，也可以选择性地使用自己的 graph 和 session，这样数据的 batch 会被存在对应 session 的 coordinator 里，以便之后更方便地获取。

```
tensorflow_df.run(num_batches=10, graph=graph, session=sess)
```

我们也可以使用 batch() 重新改变每个 batch 的大小，也可以选择将数据洗一遍来打乱

顺序，很多应用都通过这种方式增加数据的随机性。

```
tensorflow_df.batch(batch_size=10, shuffle=True, num_threads=3)
```

还有许多实用的函数，例如用 `split()` 将 DataFrame 分成多个 DataFrame，用 `select_rows()` 选择具体某行数据，等等。这里我们就不多介绍了。DataFrame 将会被主要用在 Estimator 里，这样用户就可以把精力放在数据的供给，而不用担心数据的数据类型和文件类型。这一模块以后的变化将会很大，请读者参考最新的官方文档和代码。

## 10.5 监督器 Monitors

训练模型时，没有程序日志的话整个过程就像是个黑匣子，我们很难知道模型的进展及发展方向，例如模型在进行各种优化，使用 SGD 做优化时，我们无法看到模型是否在拟合及拟合的速度。

当然，用户可以把训练的过程分为几个部分，然后在 `fit()` 迭代时时不时地打印出一些有用的信息，但是这样的程序往往会很慢。这时，TF.Learn 里自带的 Monitor 就派上用场了，它提供各种 logging 及监督控制训练的过程，这样用户就能更清楚地知道模型是否在进行有效的训练。在之前的章节中我们简单提到过，接下来将给出详细的例子来分析 Monitors 的使用方法。

TensorFlow 有 5 个等级的 log，以严重性最小到最大排列，它们是 DEBUG、INFO、WARN、ERROR，以及 FATAL。当用户选择好 log 的等级之后，只有那个等级和更严重等级的 log 会被打印出来。举例来说，如果等级设置为 ERROR，那么你会看到 ERROR 和 FATAL 等级的 log；如果等级设置为 DEBUG，那么所有等级的 log 都会打印出来。TensorFlow 的默认 log 等级是 WARN，所以如果想在模型训练时看到 log，需要用下面这行代码把等级改到 INFO。

```
tf.logging.set_verbosity(tf.logging.INFO)
```

改了等级之后，你会看到类似以下的 log。注意这些是由一个默认的 Monitor 提供的，每 100 步会打印出一些损失值信息。

```
INFO:tensorflow:Training steps [0, 200)
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:Step 1: loss_1:0 = 2.34635
```

```
INFO:tensorflow:training step 100, loss = 0.18227 (0.001 sec/batch).
INFO:tensorflow:Step 101: loss_1:0 = 0.191003
INFO:tensorflow:Step 200: loss_1:0 = 0.0835024
INFO:tensorflow:training step 200, loss = 0.080932 (0.002 sec/batch).
```

TF.Learn 提供几个方便使用的高阶 Monitor 类，例如用 CaptureVariable 将一个指定的变量的值存储到一个 Collection 里，用 PrintTensor 打印 Tensor 的值，用 SummarySaver 存储 Summary 所需要的协议缓冲（Protocol Buffer），ValidationMonitor 在训练时打印多个评估 Metrics，以及监督模型的训练以便提前停止训练防止模型的过度拟合。这些不同的 Monitor 都会在每隔 N 步时执行。

接下来，我们将详细地讲解怎样使用 Monitor，主要以 ValidationMonitor 作为例子。首先，假设手头有 CSV 格式的 iris 数据，我们可以使用 TF.Learn 自带的 learn.datasets.base.load\_csv() 读入这些 CSV 数据文件。

```
import numpy as np
import tensorflow as tf

iris_train = tf.contrib.learn.datasets.base.load_csv(
    filename="iris_training.csv", target_dtype=np.int)
iris_test = tf.contrib.learn.datasets.base.load_csv(
    filename="iris_test.csv", target_dtype=np.int)
```

接着，定义一个评估模型的 metrics 字典，这里使用 contrib.metrics 模块里面的 streaming\_accuracy、streaming\_precision，以及 streaming\_recall，对模型的准确度、精确度，以及召回率进行评估。

```
validation_metrics = {"accuracy": tf.contrib.metrics.streaming_accuracy,
                      "precision": tf.contrib.metrics.streaming_precision,
                      "recall": tf.contrib.metrics.streaming_recall}
```

然后用定义好的 validation\_metrics 建立一个 validation\_monitor，这里需要提供用来评估的数据及目标，提供 every\_n\_steps 来指示每 50 步以实行一次这个 ValidationMonitor，把之前定义好的 validation\_metrics 传入 metrics，用 early\_stopping\_metric 选择用来提前停止所需要监测的 metric，early\_stopping\_metric\_minimize=True 表明我们需要最小化之前提供的 early\_stopping\_metric。最后，用 early\_stopping\_rounds 表明如果超过 200 步训练损失

仍然不减少，ValidationMonitor 会停止 Estimator 的训练。

```
validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
    iris_test.data,
    iris_test.target,
    every_n_steps=50,
    metrics=validation_metrics,
    early_stopping_metric="loss",
    early_stopping_metric_minimize=True,
    early_stopping_rounds=200)
```

紧接着，我们建立一个深度神经网络分类器 DNNClassifier，它有三层神经网络，每一层分别有 10、15 和 10 个隐藏单元。我们在分类器进行 fit() 时来指定我们定义好的监督器 validation\_monitor，注意，也可以指定多个监督器来实现不同功能的监督，例如 [validation\_monitor, debug\_monitor, print\_monitor]。

```
classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
                                             hidden_units=[10, 15, 10],
                                             n_classes=3,
                                             model_dir="/iris_model_dir",
                                             config=tf.contrib.learn.RunConfig(save_checkpoints_secs=2))

classifier.fit(x=iris_train.data, y=iris_train.target, steps=1000,
                monitors=[validation_monitor])
```

接下来，可以使用我们熟悉的 evaluate() 或者 predict() 用新的数据评估模型的准确度。

```
accuracy_score = classifier.evaluate(x=iris_test.data,
                                      y=iris_test.target)["accuracy"]

new_samples = np.array([[5.2, 3.1, 6.5, 2.2], [2.8, 3.2, 5.5, 3.3]], dtype=float)
y = classifier.predict(new_samples)
```

我们将会得到类似以下的 log，可以观察到模型在 750 步时被终止了，因为损失值没有继续减少。

```
INFO:tensorflow:Validation (step 950): recall = 1.0, accuracy = 0.966667, gl
```

```
global_step = 932, precision = 1.0, loss = 0.0608345
INFO:tensorflow:Stopping. Best step: 750 with loss = 0.0581324.
```

虽然 ValidationMonitor 提供了很多信息和功能，但是当训练步数很大时，我们很难观察模型的准确率到底是怎么变化的。值得庆幸的是，TF.Learn 生成的 log 及 checkpoint 的文件是能够直接读入 TensorBoard 里进行可视化的。如果在命令行里执行以下几行，就会在给出的地址里看到 TensorBoard 对整个模型训练可视化，如图 10-1 所示。

```
$ tensorboard --logdir=/iris_model_dir/
Starting TensorBoard 22 on port 6006
(You can navigate to http://0.0.0.0:6006)
```

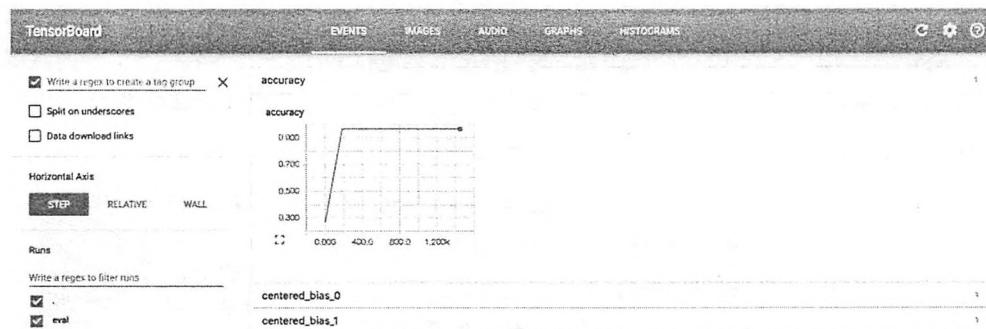


图 10-1 TensorBoard 对模型训练的可视化

图片来源于 tensorflow.org



# 11

## TF.Contrib 的其他组件

TF.Contrib 是 TensorFlow 里很重要的一个部分，很大一部分开源社区的贡献都被集中在这里，特别是一些比较新的功能，由于都是一些刚贡献的功能，谷歌会将这些代码暂时放在这里，由谷歌内部及外部的用户一起测试，根据反馈意见改进性能和改善 API 的友好度，等它们的 API 都比较稳定时，会被移到 TensorFlow 的核心模块。

这个模块里提供了机器学习需要的大部分功能，包括统计分布、机器学习层、优化函数、指标，等等。本章将简单介绍其中的一些功能让大家了解 TensorFlow 的涵盖范围和感受到社区积极地参与和贡献度。注意这部分功能在未来会不断变动和改进，如果是写生产代码的话，请以最新的官方教程和 API 指南作为最权威的参考。

### 11.1 统计分布

在 TF.contrib.distributions 模块里有许多的统计分布，例如 Bernoulli、Beta、Binomial、Gamma、Exponential、Normal、Poisson、Uniform，等等。这些统计分布大多数都是统计研究和应用中经常用到的，也是各种统计及机器学习模型的基石，许多的概率模型和图形模型（例如 Bayesian 模型）都非常依赖这些统计分布。

每个不同的统计分布有着不同的特征和函数，但是它们都是从同样的子类 Distribution

扩展开来的，Distribution 是建立和组织随机变量和统计分布的一个最基础的类，它有着许多有用的属性及类函数，例如用 `is_continuous` 表明这个随机变量分布是不是连续的，用 `allow_nan_stats` 表示这个分布是否接受 nan 的数据，用 `sample()` 从这个分布里取样，用 `prob()` 计算随机变量密度函数，用 `cdf()` 求累积分布函数，以及用 `entropy()`、`mean()`、`std()`、`variance()` 得到统计分布的平均值和方差之类的特征。如果想贡献自己的统计分布类，需要实现一些对应以上的方程，例如 `_mean()`、`_std()` 和 `_variance()`，也需要实现 `_is_continuous` 之类表明这个变量分布的属性。

我们接下来以实现好的 Gamma 分布为例来说明这个模块的大概使用方法。首先，从 `contrib.distributions` 里导入 Gamma 分布，然后初始化 alpha 和 beta 的 `tf.constant`，这些 `constant` 被用于建立 Gamma 分布，我们可以通过 `batch_shape().eval()` 得到每个样本的形状，这里例子的样本形状 `shape1` 是 `(5, )`，我们也可以使用 `get_batch_shape()` 得到样本形状，但是是以 `tf.TensorShape` 的类出现的，这个例子里 `shape2` 是 `tf.TensorShape(5)`，两种方法各有所长，需要依据具体应用的需求来使用。

```
from tensorflow.contrib.distributions import Gamma
import tensorflow as tf
alpha = tf.constant([3.0] * 5)
beta = tf.constant(11.0)
gamma = Gamma(alpha=alpha, beta=beta)
shape1 = gamma.batch_shape().eval()
shape2 = gamma.get_batch_shape()
```

然后，可以用 `log_pdf()` 函数取对应的一些值的 log 转换后的概率密度函数，我们把 6 个值放在 `numpy.array` 里，然后得到相应的 log 概率密度函数值。

```
x = np.array([2.5, 2.5, 4.0, 0.1, 1.0, 2.0], dtype=np.float32)
log_pdf = gamma.log_pdf(x)
```

也可以建立多维的 Gamma 分布，和一维的类似，只需要传入多维的 alpha 和 beta 参数就可以建立多维的 Gamma 分布。同样，我们可以对多维的 x 取得相应的 log 概率密度函数值。

```
batch_size = 6
alpha = tf.constant([[2.0, 4.0]] * batch_size)
beta = tf.constant([[3.0, 4.0]] * batch_size)
```

```
x = np.array([[2.5, 2.5, 4.0, 0.1, 1.0, 2.0]], dtype=np.float32).T
gamma = Gamma(alpha=alpha, beta=beta)
log_pdf = gamma.log_pdf(x)
```

## 11.2 Layer 模块

Contrib.layer 包含了机器学习算法所需的各种各样的成份和部件，例如卷积层、批标准化层、机器学习指标、优化函数、初始器、特征列，等等。有了这些基础的建设部件，我们可以高效地建立复杂的机器学习及机器学习系统。本章将介绍这个模块里一些主要的部件，来帮助理解 TensorFlow 的各种可能性及灵活性。

### 11.2.1 机器学习层

contrib.layers 里含有许多常用的深度学习及机器学习的层，例如卷积层、pooling 层、批标准化等，这些都是各种模型必不可少的部分，也是机器学习研究领域最活跃的一部分。

深度学习和计算机视觉里经常用到的二维的平均池是 avg\_pool2d。我们用 np.random.uniform 建立宽和高都是 3 的几张假图片，读者可以通过 contrib.layers.avg\_pool2d() 对图片快速地建立  $3 \times 3$  的二维平均池，这里 output 的形状是 [5, 1, 1, 3]，因为我们对每个  $3 \times 3$  的区域取计算平均值。

```
height, width = 3, 3
images = np.random.uniform(size=(5, height, width, 3))
output = tf.contrib.layers.avg_pool2d(images, [3, 3])
```

用类似的方法建立卷积层，这里使用同样的图片矩阵，然后用 contrib.layers.convolution2d() 建立一个有 32 个  $3 \times 3$  过滤器的卷积层，也可以改动 stride、padding、activation\_fn 等参数建立不同架构的卷积层，使用不同的卷积层激活函数。

```
output = tf.contrib.layers.convolution2d(images, num_outputs=32,
                                         kernel_size=[3, 3])
```

值得注意的是，contrib.layers 会自动建立 op 的名字，例如 output.op.name 的值是 'Conv/Relu'，因为我们使用了 Conv 层及使用了 ReLU 的激活函数，这些 layer 有自己对应的 op 名字，然后会在每个 op 空间存储对应的变量，可以通过

contrib.framework.get\_variables\_by\_name()得到对应的 op 空间变量的值。例如，可以用 get\_variables\_by\_name 得到我们建立的卷积层的权重，这里权重的形状，也就是 weights\_shape 的值，是[3, 3, 4, 32]。

```
weights = tf.contrib.framework.get_variables_by_name('weights')[0]
weights_shape = weights.get_shape().as_list()
```

接下来我们看看怎么将卷积层 layers.convolution2d()和批标准化层 layers.batch\_norm()结合使用，我们先建立一些图片的矩阵。

```
images = tf.random_uniform((5, height, width, 32), seed=1)
```

接着，使用 contrib.framework 里面的 arg\_scope 减少代码的重复使用，我们将 layers.convolution2d 及一些即将传入的参数放入 arg\_scope 中，这些参数通常是接下来会被重复使用的，把它们放在 arg\_scope 里就可以避免重复在多个地方传入，这里需要用到的参数是 normalizer\_fn 和 normalizer\_params，也就是需要用到的标准化方程及它所需要的参数，一但在 arg\_scope 里设置了这些，接下来用到 convolution2d()时就不用重复传入 normalizer\_fn 和 normalizer\_params 这两个参数了。

```
with tf.contrib.framework.arg_scope(
    [tf.contrib.layers.convolution2d],
    normalizer_fn=tf.contrib.layers.batch_norm,
    normalizer_params={'decay': 0.9}):
    net = tf.contrib.layers.convolution2d(images, 32, [3, 3])
    net = tf.contrib.layers.convolution2d(net, 32, [3, 3])
```

可以看到，TensorFlow 自动帮我们建立好了默认的一些层的名字。以上的例子里，我们可以通过 len(tf.contrib.framework.get\_variables('Conv/BatchNorm')) 得到第一个 Conv/BatchNorm 层的长度。

再来看一个完全连接的神经网络层 fully\_connected()的例子。首先，建立一些输入的矩阵，用 fully\_connected()建立一个输出 7 个神经单元的神经网络层。

```
height, width = 3, 3
inputs = tf.random_uniform((5, height * width * 3), seed=1)
with tf.name_scope('fe'):
    fc = tf.contrib.layers.fully_connected(inputs, 7,
```

```

        outputs_collections='outputs',
        scope='fc')

output_collected = tf.get_collection('outputs')[0]
self.assertEquals(output_collected.alias, 'fe/fc')

```

值得注意的一些小细节是，我们利用 `tf.name_scope` 将截下来的运算放进一个 `name_scope` 里，这样以后就可以更简单地找到我们想要的某个层的值，我们在 `fully_connected()` 里传入一个 `scope`，然后就可以通过“`fe/fc`”，也就是这个层的别号得到这个层的一些信息。我们通过传入的 `outputs_collections`，可以直接得到这个层的输出。

在 `contrib.layers` 里有许多特别方便使用的方法，例如，可以通过 `repeat()` 重复使用同样的参数重复建立某个层，例如 `y = repeat(x, 3, conv2d, 64, [3, 3], scope='conv1')` 是和以下代码等同的。

```

x = conv2d(x, 64, [3, 3], scope='conv1/conv1_1')
x = conv2d(x, 64, [3, 3], scope='conv1/conv1_2')
y = conv2d(x, 64, [3, 3], scope='conv1/conv1_3')

```

可以使用 `stack()` 来使用不同的参数建立多个 `fully_connected()` 层，我们可以建立一个三层的完全连接的神经网络，每层的单元数分别为 32、64 和 128。

```
y = stack(x, fully_connected, [32, 64, 128], scope='fc')
```

以上代码等同于：

```

x = fully_connected(x, 32, scope='fc/fc_1')
x = fully_connected(x, 64, scope='fc/fc_2')
y = fully_connected(x, 128, scope='fc/fc_3')

```

注意，`stack` 会帮你建立一个新的 `scope`，通过在 `scope` 里附加一个增量，例如在“`fc`”的基础上加上“`fc_1`”、“`fc_2`”等。之前提到的 `repeat()` 也会使用类似的机制建立新的 `scope`。

我们只简单介绍一些在深度学习中经常使用的层，如果想了解更多、更复杂的层，例如 `conv2d_transpose`、`conv2d_in_plane`、`separable_conv2d` 等，可以参考官方文档。

## 11.2.2 损失函数

`Tf.contrib.losses` 模块里包含了各种常用的损失函数，适用于二类分类、多类分类，以

及回归模型等各式各样的机器学习算法。接下来，我们将举例说明它们的使用方法。

我们先以绝对差值举例说明，首先用 `tf.constant` 建立一些 `predictions` 和 `targets` 的数列。注意，它们必须是同样的 `shape`，然后可以选择性地建立权重，因为在许多实际应用中，每个预测值的权重也是特别关键的。

```
predictions = tf.constant([4, 8, 12, 8, 1, 3], shape=(2, 3))
targets = tf.constant([1, 9, 2, -5, -2, 6], shape=(2, 3))
weight = tf.constant([1.2, 0.0], shape=[2,])
```

接着，可以使用 `losses.absolute_difference()` 计算这组预测的损失值，从而在之后的建模中起到引导性的作用。

```
loss = tf.contrib.losses.absolute_difference(predictions, targets, weight)
```

接下来，来看一个计算 softmax 交叉熵的例子，这种方法多适用于多类分类的机器学习模型。同样地，我们先建立 `predictions` 和 `labels`，与之前不一样的是，它们是多维的，也就是 softmax 交叉熵最擅长处理的。然后，使用 `losses.softmax_cross_entropy()` 计算这组预测中 softmax 交叉熵的值。注意，需要像其他 TensorFlow 的应用一样使用 `loss.eval()` 运行得到它的值。可以从 `loss.op.name` 得到 TensorFlow 自动赋值的 op 的名字，这个情况下它是'softmax\_cross\_entropy\_loss/value'。其他的损失函数也是使用这样的命名习俗。

```
predictions = tf.constant([[10.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 10.0]])
labels = tf.constant([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

loss = tf.contrib.losses.softmax_cross_entropy(predictions, labels)
loss.eval()
loss.op.name
```

其他的损失函数的使用方法大同小异，值得注意的是，许多损失函数里有许多的参数可以使用。例如，可以使用 `softmax_cross_entropy()` 里面的 `label_smoothing` 将所有的标识进行平滑，从而使在某些应用中计算出来的 softmax 交叉熵更具有实际应用的代表性，使用方法如下。

```
logits = tf.constant([[100.0, -100.0, -100.0]])
labels = tf.constant([[1, 0, 0]])
label_smoothing = 0.1
```

```
loss = tf.contrib.losses.softmax_cross_entropy(logits, labels,
                                              label_smoothing=label_smoothing)
```

许多应用大部分标识的分布都比较稀疏，可以使用 `sparse_softmax_cross_entropy()`，这样计算起来会更有效率。

```
logits = tf.constant([[10.0, 0.0, 0.0], 0.0, 10.0, 0.0], [0.0, 0.0, 10.0]])
labels = tf.constant([[0], [1], [2]], dtype=tf.int64)
loss = tf.contrib.losses.sparse_softmax_cross_entropy(logits, labels)
```

### 11.2.3 特征列 Feature Column

在很多数据科学和机器学习的应用中，大家都习惯以表格的形式存储和处理数据，然后将数据输入机器学习模型中。处理数据的方式多种多样，例如 Python 里有大家熟悉的 Pandas 包。数据从各种数据源得来，经过各种方式的清理、筛选、合并，以及特征工程，然后进行模型的建立。在 TensorFlow 里怎样更好地进行我们的特征工程和建模的工作呢？

`TF.contrib.layers` 里有许多高阶的特征列（Feature Column）API，可以让大家的特征工程更有效率，然后紧密地和 `TF.Learn` 的 API 结合使用，建立最适合自己的数据的模型。接下来，我们将介绍如何使用这些高阶的特征列 API 及如何和 `TF.Learn` 结合使用。

数据里一般包含连续特征（Continuous Feature）及类别特征（Categorical Feature）。像花瓣的长度和宽度这样连续的数值特征称为连续特征，我们可以直接把它们用在模型里。如果特征代表了类别，例如性别、种族这样的不连续的类别特征，那么往往需要对它们进行处理，例如将它们数值化，也就是将它们转换为一系列的数值来代表每个不同的类别。Feature Column API 可以很方便地将各种类型的特征转换为想要的格式。

假设读者已经用类似以下的 `learn.datasets` 的 API 来读入数据，例如：

```
training_set = learn.datasets.base.load_csv(filename=iris_training,
                                             target_dtype=np.int)
test_set = learn.datasets.base.load_csv(filename=iris_testing,
                                         target_dtype=np.int)
```

接下来就可以用 `layers.FeatureColumn` 的 API 定义一些特征列，例如，使用 `real_valued_column()` 定义连续的特征（如年龄、收入、开销，以及工作市场）。

```
from tf.contrib import layers
age = layers.real_valued_column("age")
income = layers.real_valued_column("income")
spending = layers.real_valued_column("spending")
hours_of_work = layers.real_valued_column("hours_of_work")
```

紧接着，用 `sparse_column_with_keys()` 处理像性别这样的类别特征。

```
gender = layers.sparse_column_with_keys(column_name="gender",
                                         keys=["female", "male"])
```

注意，使用 `sparse_column_with_keys()` 前，必须要知道这个特征所有可能的值，本例中，性别分为男性和女性。如果事先不知道所有可能的值，可以使用 `sparse_column_with_hash_bucket()` 将特征转换为特征列。以教育程度这样的特征为例，由于对数据不是特别熟悉，无法事先知道所有可能的教育程度，我们可以用哈希表建立这样的特征。

```
education = layers.sparse_column_with_hash_bucket("education",
                                                 hash_bucket_size=1000)
```

`sparse_column_with_keys()` 及 `sparse_column_with_hash_bucket()` 都能将数据转换为 `SparseColumn`，然后可以直接在 `TF.Learn` 里使用，传入 `Estimator` 里。

有时，在数据科学的应用中，一些连续的特征可能需要被离散化，从而形成新的类别特征，这样能更好地代表特征和目标分类类别之间的关系。例如年龄是连续特征，分类的类别是职业的类别（如经理、猎头等），往往这些职业的类别和年龄阶段有关，而不是简单的数值年龄。因为 18 岁、19 岁、20 岁往往没有明显的区别，所以有时会将这样的连续特征区间化和离散化，例如将 18 岁~20 岁分为一类。在 `FeatureColumn API` 里，我们可以很快地进行这样的转换。

```
age_range = layers.bucketized_column(age, boundaries=[18, 25, 30, 35, 40,
                                                       45, 50, 55, 60, 65])
```

在以上的例子里，使用 `bucketized_column()` 将之前的年龄 `SparseColumn` 进行进一步的区间化，将年龄段分为 18 岁~25 岁、26 岁~30 岁、31 岁~35 岁，等等。

在许多应用里，一个好的模型不仅需要一些单独的特征列，有时两个或多个特征之间

的综合和交互与目标分类类别之间的关系更紧密。有时多个特征之间是相关的，使用特征的交互往往能建立更有效的模型。例如，对年龄、职业和种族这三个特征，我们可以使用 crossed\_column() 建立交叉特征列：

```
combined = layers.crossed_column([age_range, race, occupation],
                                 hash_bucket_size=int(1e7))
```

建立好各式各样的特征列之后，我们可以直接将它们传入不同的 TF.Learn Estimator。以下面这个简单的逻辑回归分类模型为例，我们可以使用之前介绍过的 fit()、predict() 等方法训练和评估模型。

```
classifier = tf.contrib.learn.LinearClassifier(feature_columns=[gender, education, occupation, combined, age_range, race, income, spending], model_dir=model_dir)
```

这里我们只是简单地介绍了一些比较常用的函数，在实际应用中有各种各样的需求，例如有时想取一部分特征的加权求和作为一个新的特征列，可以使用 weighted\_sum\_from\_feature\_columns() 来很快地实现。读者可以在官方文档里找到更多需要的函数。

#### 11.2.4 Embeddings

在许多深度模型应用中，包含许多稀疏的、高维的类别特征向量，我们通常先把它们转换成低维的、稠密的实数值的向量，也通常将它们和连续特征向量联合起来，一起输入进神经网络模型中进行训练和优化损失函数，这些被统一称为嵌入向量（Embedding Vectors）。大部分文本识别都是先将文本转换成嵌入向量，然后对它们进行分析并用在模型训练中。

contrib.layers 模块里的 embedding\_column() 能迅速将高维稀疏的类别特征向量转换为读者想要的维数的嵌入向量，以下是一个例子。

```
embedding_columns = [
    tf.contrib.layers.embedding_column(title, dimension=8),
    tf.contrib.layers.embedding_column(education, dimension=8),
    tf.contrib.layers.embedding_column(gender, dimension=8),
    tf.contrib.layers.embedding_column(race, dimension=8),
    tf.contrib.layers.embedding_column(country, dimension=8)]
```

这里的 title、education、gender、race，以及 country 都是比较稀疏的类别特征向量，我们通过使用 embedding\_column()，把它们转换为低维数的稠密向量，从而更好地归纳数据中的特性，特别是当一组特征中的交互矩阵比较稀疏，级别比较高时，这种方法会使模型更具有概括性且更有效。

接下来，可以直接将它们传入 TF.Learn 的 Estimator 里进行模型的建立、训练，以及评估。例如，可以将 embedding\_columns 传入 DNNLinearCombinedClassifier 里的深度神经网络特征列里。

```
est = tf.contrib.learn.DNNLinearCombinedClassifier(
    model_dir=model_dir,
    linear_feature_columns=wide_columns,
    dnn_feature_columns=embedding_columns,
    dnn_hidden_units=[100, 50])
```

embedding\_columns()是 contrib.layers 模块里最简单易用的一个，当涉及实际数据时，许多稀疏高维的数据里通常有空的特征及无效的 ID，这时可以使用 safe\_embedding\_lookup\_sparse()安全地建立嵌入向量。这里先用 tf.SparseTensor 建立好稀疏的 ID 及稀疏的权重。

```
indices = [[0, 0], [0, 1], [0, 2], [1, 0], [3, 0], [4, 0], [4, 1]]
ids = [0, 1, -1, -1, 2, 0, 1]
weights = [1.0, 2.0, 1.0, 1.0, 3.0, 0.0, -0.5]
shape = [5, 4]

sparse_ids = tf.SparseTensor(
    tf.constant(indices, tf.int64), tf.constant(ids, tf.int64),
    tf.constant(shape, tf.int64))

sparse_weights = tf.SparseTensor(
    tf.constant(indices, tf.int64), tf.constant(weights, tf.float32),
    tf.constant(shape, tf.int64))
```

接下来，建立嵌入向量的权重 embedding\_weights，这取决于词汇量大小、嵌入向量维数，以及 shard 数量。然后，使用 initializer.run()和 eval()初始化嵌入向量的权重，具体

细节和参数的说明请参考最新的官方文档。

```
vocab_size=4
embed_dim=4
num_shards=1
embedding_weights = tf.create_partitioned_variables(
    shape=[vocab_size, embed_dim],
    slicing=[num_shards, 1],
    initializer=tf.truncated_normal_initializer(mean=0.0,
                                                stddev=1.0 / math.sqrt(vocab_size), dtype=tf.float32))
for w in embedding_weights:
    w.initializer.run()
embedding_weights = [w.eval() for w in embedding_weights]
```

最后，可以使用 `safe_embedding_lookup_sparse()` 将原来的特征向量安全地转换为低维和稠密的特征向量，这里使用 `eval()`，然后将它们收集到一个 tuple 里。

```
embedding_lookup_result = (tf.contrib.layers.safe_embedding_lookup_sparse(
    embedding_weights, sparse_ids, sparse_weights).eval())
```

### 11.3 性能分析器 tfprof

TensorFlow 也在 Contrib 模块里提供了自己的性能分析器 `tfprof`，可以通过它帮助分析模型的架构及衡量系统的性能。它涵盖了许多实用的功能，例如衡量模型的参数、浮点运算、op 执行时间、要求的存储大小、探索模型的结构等。本节将简单地介绍一些功能。

首先，通过以下命令安装 `tfprof` 命令行的工具。

```
bazel build -c opt tensorflow/contrib/tfprof/...
```

可以通过以下命令查询帮助文件。

```
bazel-bin/tensorflow/contrib/tfprof/tools/tfprof/tfprof help
```

可以执行互动模式，然后指定 `graph_path` 来分析模型的 `shape` 和参数。

```
bazel-bin/tensorflow/contrib/tfprof/tools/tfprof/tfprof \
--graph_path=/graph.pbtxt
```

类似地，我们用 graph\_path 和 checkpoint\_path 查看 checkpoint 里 Tensor 的数据和相对应的值。

```
bazel-bin/tensorflow/contrib/tfprof/tools/tfprof/tfprof \
--graph_path=graph.pbtxt \
--checkpoint_path=model.ckpt
```

与此同时，我们可以多提供一个 run\_meta\_path 来查看不同 op 请求的存储和计时。

```
bazel-bin/tensorflow/contrib/tfprof/tools/tfprof/tfprof \
--graph_path=graph.pbtxt \
--run_meta_path=run_meta \
--checkpoint_path=model.ckpt
```

值得注意的是，上面用到了 run\_meta\_path、graph\_path 和 checkpoint\_path 几个路径，我们是怎么得到这几种类型的文件的呢？

graph\_path 的文件是 GraphDef 文本文件，用来在内存里建立模型的代表，例如用 tf.Supervisor 写出来的 graph.pbtxt 就是一个 GraphDef 文本文件的例子。如果不使用 tf.Supervisor，那么可以使用 tf.Graph.as\_graph\_def() 或者其他类似的 API 存储模型的定义到一个 GraphDef 文件里。

run\_meta\_path 所需的文件是 tensorflow::RunMetadata 的结果，这个方程是用来得到模型中每个 op 所需的存储和时间消耗的，以下简单的几行代码可以写出一个 RunMetadata 文件。

```
run_options = config_pb2.RunOptions(
    trace_level=config_pb2.RunOptions.FULL_TRACE)
run_metadata = config_pb2.RunMetadata()
_ = self._sess.run(..., options=run_options, run_metadata=run_metadata)
with gfile.Open(os.path.join(output_dir, "run_meta"), "w") as f:
    f.write(run_metadata.SerializeToString())
```

checkpoint\_path 是模型的 checkpoint，它包含了所有 checkpoint 的变量的 op 类型、shape 和它们的值。

读者也可以提供其他的路径，例如 op\_log\_path 路径，它是 tensorflow::tfprof::OpLog

的结果，包含了额外的 op 的信息，由于它包含了 op 的组的类别名字，用户可以很简单地综合 op 的一些数据，而不会不小心错过其中一部分 op。以下用一个暴露出来的 API 来很快地写出了一个 OpLog 文件。

```
tf.contrib.tfprof.tfprof_logger.write_op_log(graph, log_dir, op_log=None)
```

由于 tfprof 是一个 CLI 命令行的工具，当输入之前的 tfprof 命令按下回车键时，会进入互动模式，再按一下回车键会看到一些类似以下的命令行参数的默认值。

```
tfprof>
-max_depth          4
-min_bytes          0
-min_micros         0
-min_params         0
-min_float_ops      0
-device_regexes     .* 
-order_by           name
-account_type_regexes Variable
-start_name_regexes .* 
-trim_name_regexes
-show_name_regexes  .* 
-hide_name_regexes VariableInitialized_[0-9]+,save\/*,^zeros[0-9_]* 
-account_displayed_op_only false
-select              params
-viz                false
-dump_to_file
```

然后就可以调节里面的参数，例如用 show\_name\_regexes 查找 scope 名字正则式为 unit\_1\_0.\*gamma，max\_depth 为 5 的变量，查看这些 tensor 的值：

```
tfprof> scope -show_name_regexes unit_1_0.*gamma -select tensor_value \
-max_depth 5
```

读者会得到类似以下符合条件的 tensor 的值：

```
unit_1_0/shared_activation/init_bn/gamma ()
[1.80 2.10 2.06 1.91 2.26 1.86 1.81 1.37 1.78 1.85 1.96 1.54 2.04 2.34 2.22
```

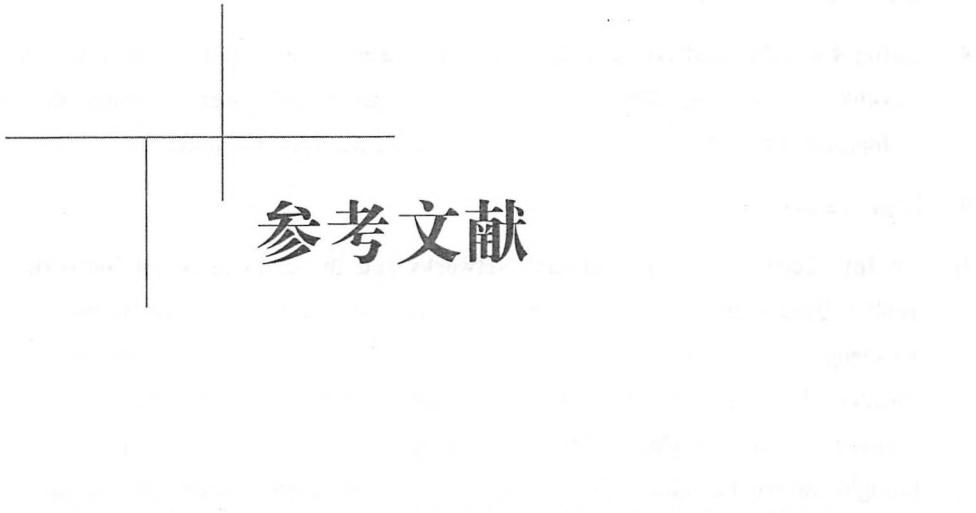
```
1.99 ],
unit_1_0/sub2/bn2/gamma ()
[1.57 1.83 1.30 1.25 1.59 1.14 1.26 0.82 1.19 1.10 1.48 1.01 0.82 1.23 1.21
1.14 ]
```

tfprof 提供了两种类型的分析：scope 和 graph。当读者想查看一些变量和 scope 的值的时候，你可以使用 scope，也就是我们上面阐述过的例子。当读者想查看 op 在 graph 里所花的内存和时间时，可以使用 graph 查看。

类似地，可以改动一些命令行参数，例如使用 start\_name\_regexes 选择想要查看的 op 的名字，这里我们假设需要查看命名为 cost 的损失 op 的内存和时间花费的情况：

```
tfprof> graph -start_name_regexes cost.* -max_depth 100 -min_micros 10000 \
-select micros -account_type_regexes .*
init/init_conv/Conv2D (11.75ms/3.10sec)
random_shuffle_queue_DequeueMany (3.09sec/3.09sec)
unit_1_0/sub2/conv2/Conv2D (74.14ms/3.19sec)
unit_1_3/sub2/conv2/Conv2D (60.75ms/3.34sec)
unit_2_4/sub2/conv2/Conv2D (73.58ms/3.54sec)
unit_3_3/sub2/conv2/Conv2D (10.26ms/3.60sec)
```

我们就先简单地介绍以上这些模块，这一部分变化很大，更多的功能还需要读者自己摸索并查看官方文档。



## 参考文献

1. **Large Scale Distributed Deep Networks**, *NIPS 2012*, Authors: Jeffrey Dean and Greg S. Corrado and Rajat Monga and Kai Chen and Matthieu Devin and Quoc V. Le and Mark Z. Mao and Marc'Aurelio Ranzato and Andrew Senior and Paul Tucker and Ke Yang and Andrew Y. Ng
2. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**, <http://tensorflow.org/>, Authors: Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Leve nberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike S chuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng
3. <http://eigen.tuxfamily.org/>
4. <http://www.netlib.org/blas/>
5. <https://developer.nvidia.com/cublas>

6. <https://github.com/akrizhevsky/cuda-convnet2>
7. <https://developer.nvidia.com/cudnn>
8. **Caffe: Convolutional Architecture for Fast Feature Embedding**, <https://arxiv.org/abs/1408.5093>, Authors: Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, Trevor Darrell
9. <https://keras.io/>
10. **An Introduction to Computational Networks and the Computational Network Toolkit**, *Microsoft Research*, Authors: Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, Xuedong Huang
11. <https://github.com/torch/torch7>
12. **MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems**, *NIPS 2015*, Authors: Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang
13. <https://github.com/autumnai/leaf>
14. <http://deeplearning.net/software/theano/>
15. <https://deeplearning4j.org/>
16. <https://github.com/Lasagne/Lasagne>
17. <https://github.com/NervanaSystems/neon>
18. <https://developer.nvidia.com/cuda-zone>
19. <https://tensorflow.github.io/serving/>
20. <https://www.continuum.io/downloads>

21. <http://yann.lecun.com/exdb/mnist/>
22. [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist\\_softmax.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist_softmax.py)
23. Reducing the dimensionality of data with neural networks, *Science*, Authors: G. E. Hinton and R. R. Salakhutdinov
24. Deep belief networks, *Scholarpedia*, Geoffrey E. Hinton
25. <https://github.com/tensorflow/models/blob/master/autoencoder/AdditiveGaussianNoiseAutoencoderRunner.py>
26. Understanding the difficulty of training deep feedforward neural networks, *AISTATS 2010*, Authors: Xavier Glorot and Yoshua Bengio
27. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research*, Authors: Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov
28. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, *Journal of Machine Learning Research*, Authors: John Duchi, Elad Hazan, Yoram Singer
29. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit, *Nature*, Authors: Hahnloser, R. Sarpeshkar, M A Mahowald, R. J. Douglas, H.S. Seung
30. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION, *ICLR 2015*, Authors: Diederik P. Kingma and Jimmy Lei Ba
31. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD, <https://arxiv.org/abs/1212.5701>, Authors: Matthew D. Zeiler
32. On the momentum term in gradient descent learning algorithms, *Neural Networks : The Official Journal of the International Neural Network*, Authors: Ning Qian
33. A method for unconstrained convex minimization problem with the rate of

- convergence  $O(1/k^2)$ , *Doklady ANSSSR*, Authors: Nesterov, Y.
- 34. **Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)**, *ICLR 2016*, Authors: Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter
  - 35. **Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification**, <https://arxiv.org/abs/1502.01852>, Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
  - 36. **Empirical Evaluation of Rectified Activations in Convolutional Network**, <https://arxiv.org/abs/1505.00853>, Authors: Bing Xu, Naiyan Wang, Tianqi Chen, Mu Li
  - 37. **Observations on the scratch-reflex in the spinal dog**, *Journal of Physiology*, Authors: Sherrington, C. S.
  - 38. **Neocognitron: a neural network model for a mechanism of visual pattern recognition**, *IEEE Transactions on Systems, Man, and Cybernetics*, Authors: Fukushima, K.; Miyake, S.; Ito, T.
  - 39. **Gradient-based learning applied to document recognition**, *Proceedings of the IEEE*, Authors: Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner
  - 40. **ImageNet Classification with Deep Convolutional Neural Networks**, *NIPS 2012*, Authors: Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
  - 41. <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/tutorials/mnist/pros/index.md>
  - 42. **Learning Multiple Layers of Features from Tiny Images**, <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>, Authors: Alex Krizhevsky
  - 43. <http://groups.csail.mit.edu/vision/TinyImages/>
  - 44. <https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10>
  - 45. **ImageNet Classification with Deep Convolutional Neural Networks**, *NIPS 2012*, Authors: Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

46. **VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION**, *ICLR 2015*, Authors: Karen Simonyan and Andrew Zisserman
47. **Going Deeper with Convolutions**, *CVPR 2015*, Authors: Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich
48. **Deep Residual Learning for Image Recognition**, <https://arxiv.org/abs/1512.03385>, Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
49. **ImageNet Large Scale Visual Recognition Challenge**, *IJCV 2015*, Authors: Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei
50. **ImageNet: A Large-Scale Hierarchical Image Database**, *CVPR 2009*, Authors: J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei
51. [https://github.com/tensorflow/models/blob/master/tutorials/image/alexnet/alexnet\\_benchmark.py](https://github.com/tensorflow/models/blob/master/tutorials/image/alexnet/alexnet_benchmark.py)
52. <https://github.com/machrisaa/tensorflow-vgg>
53. [https://github.com/tensorflow/models/blob/master/slim/nets/inception\\_v3.py](https://github.com/tensorflow/models/blob/master/slim/nets/inception_v3.py)
54. [https://github.com/tensorflow/models/blob/master/slim/nets/resnet\\_v2.py](https://github.com/tensorflow/models/blob/master/slim/nets/resnet_v2.py)
55. **Efficient Estimation of Word Representations in Vector Space**, <https://arxiv.org/abs/1301.3781>, Authors: Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean
56. [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec\\_basic.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py)
57. **Long short-term memory**, <https://dx.doi.org/10.1162%2Fneco.1997.9.8.1735>, Authors: Sepp Hochreiter, Jürgen Schmidhuber
58. [https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb\\_word\\_lm.py](https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py)
59. **Bidirectional recurrent neural networks**, *IEEE Transactions on Signal Processing*,

Authors: Mike Schuster and Kuldip K. Paliwal

60. [https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3\\_Neural\\_Networks/bidirectional\\_rnn.py](https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_Neural_Networks/bidirectional_rnn.py)
61. **Human-level control through Deep Reinforcement Learning**, *Nature*, Authors: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis
62. **Mastering the game of Go with deep neural networks and tree search**, *Nature*, Authors: David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis
63. **OpenAI Gym**, <http://arxiv.org/abs/1606.01540>, Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba
64. <https://github.com/awjuliani/DeepRL-Agents/blob/master/Policy-Network.ipynb>
65. **Learning from Delayed Rewards**, *Ph.D. thesis, Cambridge University*, Authors: Watkins, C.J.C.H.
66. <https://github.com/awjuliani/DeepRL-Agents/blob/master/Double-Dueling-DQN.ipynb>
67. [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist\\_with\\_summaries.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist_with_summaries.py)
68. [https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar10\\_multi\\_gpu\\_train.py](https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar10_multi_gpu_train.py)
69. [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/dist\\_test/python/mnist\\_replica.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/dist_test/python/mnist_replica.py)

## 作者简介



· 黄文坚 ·

PPmoney大数据算法总监，负责集团的风控、理财、互联网证券等业务的数据挖掘工作。Google TensorFlow Contributor。前明略数据技术合伙人，领导了对诸多大型银行、保险公司、基金的数据挖掘项目，包括建立金融风控模型、新闻舆情分析、保险复购预测等。曾就职于阿里巴巴搜索引擎算法团队，负责天猫个性化搜索系统。曾参加阿里巴巴大数据推荐算法大赛，于7000多支队伍中获得前10名。本科、研究生就读于香港科技大学，曾在顶级会议和期刊SIGMOBILE MobiCom、IEEE Transactions on Image Processing发表论文，研究成果获美国计算机协会移动计算大会（MobiCom）最佳移动应用技术冠军，并获得两项美国专利和一项中国专利。



· 唐 源 ·

目前在芝加哥的Uptake公司带领团队建立用于多个物联网领域的数据科学引擎进行条件和健康监控，也建立了公司的预测模型引擎，现在被用于航空、能源等大型机械领域。一直活跃在开源软件社区，是TensorFlow和DMLC的成员，是TensorFlow、XGBoost、MXNet等软件的committer，TF.Learn、ggfortify等软件的作者，以及caret、pandas等软件的贡献者。曾获得谷歌Open Source Peer Bonus，以及多项高校和企业编程竞赛的奖项。在美国宾州州立大学获得荣誉数学学位，曾在本科学习期间成为创业公司DataNovo的核心创始成员，研究专利数据挖掘、无关键字现有技术搜索、策略推荐等。



博文视点Broadview



@博文视点Broadview

上架建议：人工智能 > 深度学习

ISBN 978-7-121-30912-0



9 787121 309120 >

定价：79.00元



策划编辑：郑柳洁

欢迎投稿

责任编辑：郑柳洁

邮箱：zhenglj@phei.com.cn

封面设计：侯士卿

微信号：Alinamercy