利用sklearn api gridsearch 进行keras参数调优

2018年02月12日 17:14:43 mishidemudong 阅读数 1446 更多

本文主要想为大家介绍如何使用scikit-learn网格搜索功能，并给出一套代码实例。你可以将代码复制粘贴到自己的项目中，作为项目起始。

下文所涉及的议题列表：

1. 如何在scikit-learn模型中使用Keras。
2. 如何在scikit-learn模型中使用网格搜索。
3. 如何调优批尺寸和训练epochs。
4. 如何调优优化算法。
5. 如何调优学习率和动量因子。
6. 如何确定网络权值初始值。
7. 如何选择神经元激活函数。
8. 如何调优Dropout正则化。
9. 如何确定隐藏层中的神经元的数量。

如何在scikit-learn模型中使用Keras

通过用 **KerasClassifier** 或 **KerasRegressor** 类包装Keras模型，可将其用于scikit-learn。

要使用这些包装，必须定义一个函数，以便按顺序模式创建并返回Keras，然后当构建 **KerasClassifier** 类时，把该函数传递给 **build_fn** 参数。

例如：

```
def create_model():
    ...
    return model

model = KerasClassifier(build_fn=create_model)
```

KerasClassifier类 的构建器为可以采取默认参数，并将其被传递给 **model.fit()** 的调用函数，比如 epochs数目和批尺寸（batch size)。

例如：

```
def create_model():
    ...
    return model

model = KerasClassifier(build_fn=create_model, nb_epoch=10)
```

KerasClassifier类的构造也可以使用新的参数，使之能够传递给自定义的create_model()函数。这些新的参数，也必须由使用默认参数的create_model() 函数的签名定义。

例如：

```
def create_model(dropout_rate=0.0):
    ...
    return model

model = KerasClassifier(build_fn=create_model, dropout_rate=0.2)
```

您可以在 ***Keras API文档*** 中，了解到更多关于scikit-learn包装器的知识。

如何在scikit-learn模型中使用网格搜索

网格搜索（grid search）是一项模型超参数优化技术。

在scikit-learn中，该技术由 **GridSearchCV** 类提供。

当构造该类时，你必须提供超参数字典，以便用来评价 **param_grid** 参数。这是模型参数名称和大量列值的示意图。

默认情况下，精确度是优化的核心，但其他核心可指定用于GridSearchCV构造函数的score参数。

默认情况下，网格搜索只使用一个线程。在GridSearchCV构造函数中，通过将 n_jobs参数设置为-1，则进程将使用计算机上的所有内核。这取决于你的Keras后端，并可能干扰主神经网络的训练过程。

当构造并评估一个模型中各个参数的组合时，GridSearchCV会起作用。使用交叉验证评估每个单个模型，且默认使用3层交叉验证，尽管通过将cv参数指定给 GridSearchCV构造函数时，有可能将其覆盖。

下面是定义一个简单的网格搜索示例：

```
param_grid = dict(nb_epochs=[10,20,30])
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
```

一旦完成，你可以访问网格搜索的输出，该输出来自结果对象，由grid.fit()返回。best_score_成员提供优化过程期间观察到的最好的评分，best_params_描述了已取得最佳结果的参数的组合。

您可以在 ***scikit-learn API文档*** 中了解更多关于GridSearchCV类的知识。

问题描述

现在我们知道了如何使用scikit-learn 的Keras模型，如何使用scikit-learn 的网格搜索。现在一起看看下面的例子。

所有的例子都将在一个小型的标准机器学习数据集上来演示，该数据集被称为 ***Pima Indians onset of diabetes 分类数据集*** 。该小型数据集包括了所有容易工作的数值属性。

下载数据集 ，并把它放置在你目前工作目录下，命名为： **pima-indians-diabetes.csv** 。

当我们按照本文中的例子进行，能够获得最佳参数。因为参数可相互影响，所以这不是网格搜索的最佳方法，但出于演示目的，它是很好的方法。

注意并行化网格搜索

所有示例的配置为了实现并行化（n_jobs=-1）。

如果显示像下面这样的错误：

```
INFO (theano.gof.compilelock): Waiting for existing lock by process '55614' (I am process '55613')
INFO (theano.gof.compilelock): To manually release the lock, delete ...
```

结束进程，并修改代码，以便不并行地执行网格搜索，设置n_jobs=1。

如何调优批尺寸和训练epochs

在第一个简单的例子中，当调整网络时，我们着眼于调整批尺寸和训练epochs。

迭代梯度下降的批尺寸大小是权重更新之前显示给网络的模式数量。它也是在网络训练的优选法，定义一次读取的模式数并保持在内存中。

训练epochs是训练期间整个训练数据集显示给网络的次数。有些网络对批尺寸大小敏感，如LSTM复发性神经网络和卷积神经网络。

在这里，我们将以20的步长，从10到100逐步评估不同的微型批尺寸。

完整代码如下：

```
# Use scikit-learn to grid search the batch size and epochs
import numpy
```

```
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# define the grid search parameters
batch_size = [10, 20, 40, 60, 80, 100]
epochs = [10, 50, 100]
param_grid = dict(batch_size=batch_size, nb_epoch=epochs)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.686198 using {'nb_epoch': 100, 'batch_size': 20}
0.348958 (0.024774) with: {'nb_epoch': 10, 'batch_size': 10}
0.348958 (0.024774) with: {'nb_epoch': 50, 'batch_size': 10}
0.466146 (0.149269) with: {'nb_epoch': 100, 'batch_size': 10}
0.647135 (0.021236) with: {'nb_epoch': 10, 'batch_size': 20}
0.660156 (0.014616) with: {'nb_epoch': 50, 'batch_size': 20}
0.686198 (0.024774) with: {'nb_epoch': 100, 'batch_size': 20}
0.489583 (0.075566) with: {'nb_epoch': 10, 'batch_size': 40}
0.652344 (0.019918) with: {'nb_epoch': 50, 'batch_size': 40}
0.654948 (0.027866) with: {'nb_epoch': 100, 'batch_size': 40}
0.518229 (0.032264) with: {'nb_epoch': 10, 'batch_size': 60}
0.605469 (0.052213) with: {'nb_epoch': 50, 'batch_size': 60}
0.665365 (0.004872) with: {'nb_epoch': 100, 'batch_size': 60}
0.537760 (0.143537) with: {'nb_epoch': 10, 'batch_size': 80}
0.591146 (0.094954) with: {'nb_epoch': 50, 'batch_size': 80}
0.658854 (0.054904) with: {'nb_epoch': 100, 'batch_size': 80}
0.402344 (0.107735) with: {'nb_epoch': 10, 'batch_size': 100}
0.652344 (0.033299) with: {'nb_epoch': 50, 'batch_size': 100}
0.542969 (0.157934) with: {'nb_epoch': 100, 'batch_size': 100}
```

我们可以看到，批尺寸为20、100 epochs能够获得最好的结果，精确度约68%。

如何调优训练优化算法

Keras提供了一套最先进的不同的优化算法。

在这个例子中，我们调整用来训练网络的优化算法，每个都用默认参数。

这个例子有点奇怪，因为往往你会先选择一种方法，而不是将重点放在调整问题参数上（参见下一个示例）。

在这里，我们将评估 ***Keras API支持的整套优化算法*** 。

完整代码如下：

```
# Use scikit-learn to grid search the batch size and epochs
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(optimizer='adam'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
param_grid = dict(optimizer=optimizer)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.704427 using {'optimizer': 'Adam'}
0.348958 (0.024774) with: {'optimizer': 'SGD'}
0.348958 (0.024774) with: {'optimizer': 'RMSprop'}
0.471354 (0.156586) with: {'optimizer': 'Adagrad'}
0.669271 (0.029635) with: {'optimizer': 'Adadelta'}
0.704427 (0.031466) with: {'optimizer': 'Adam'}
0.682292 (0.016367) with: {'optimizer': 'Adamax'}
0.703125 (0.003189) with: {'optimizer': 'Nadam'}
```

结果表明，ATOM优化算法结果最好，精确度约为70%。

如何优化学习速率和动量因子?

预先选择一个优化算法来训练你的网络和参数调整是十分常见的。目前，最常用的优化算法是普通的随机梯度下降法（Stochastic Gradient Descent，SGD），因为它十分易于理解。在本例中，我们将着眼于优化SGD的学习速率和动量因子（momentum）。

学习速率控制每批（batch）结束时更新的权重，动量因子控制上次权重的更新对本次权重更新的影响程度。

我们选取了一组较小的学习速率和动量因子的取值范围：从0.2到0.8，步长为0.2，以及0.9（实际中常用参数值）。

一般来说，在优化算法中包含epoch的数目是一个好主意，因为每批（batch）学习量（学习速率）、每个 epoch更新的数目（批尺寸）和epoch的数量之间都具有相关性。

完整代码如下：

```
# Use scikit-learn to grid search the learning rate and momentum
```

```
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.optimizers import SGD
# Function to create model, required for KerasClassifier
def create_model(learn_rate=0.01, momentum=0):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]
param_grid = dict(learn_rate=learn_rate, momentum=momentum)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.680990 using {'learn_rate': 0.01, 'momentum': 0.0}
0.348958 (0.024774) with: {'learn_rate': 0.001, 'momentum': 0.0}
0.348958 (0.024774) with: {'learn_rate': 0.001, 'momentum': 0.2}
0.467448 (0.151098) with: {'learn_rate': 0.001, 'momentum': 0.4}
0.662760 (0.012075) with: {'learn_rate': 0.001, 'momentum': 0.6}
0.669271 (0.030647) with: {'learn_rate': 0.001, 'momentum': 0.8}
0.666667 (0.035564) with: {'learn_rate': 0.001, 'momentum': 0.9}
0.680990 (0.024360) with: {'learn_rate': 0.01, 'momentum': 0.0}
0.677083 (0.026557) with: {'learn_rate': 0.01, 'momentum': 0.2}
0.427083 (0.134575) with: {'learn_rate': 0.01, 'momentum': 0.4}
0.427083 (0.134575) with: {'learn_rate': 0.01, 'momentum': 0.6}
0.544271 (0.146518) with: {'learn_rate': 0.01, 'momentum': 0.8}
0.651042 (0.024774) with: {'learn_rate': 0.01, 'momentum': 0.9}
0.651042 (0.024774) with: {'learn_rate': 0.1, 'momentum': 0.0}
0.651042 (0.024774) with: {'learn_rate': 0.1, 'momentum': 0.2}
0.572917 (0.134575) with: {'learn_rate': 0.1, 'momentum': 0.4}
0.572917 (0.134575) with: {'learn_rate': 0.1, 'momentum': 0.6}
0.651042 (0.024774) with: {'learn_rate': 0.1, 'momentum': 0.8}
0.651042 (0.024774) with: {'learn_rate': 0.1, 'momentum': 0.9}
0.533854 (0.149269) with: {'learn_rate': 0.2, 'momentum': 0.0}
0.427083 (0.134575) with: {'learn_rate': 0.2, 'momentum': 0.2}
0.427083 (0.134575) with: {'learn_rate': 0.2, 'momentum': 0.4}
0.651042 (0.024774) with: {'learn_rate': 0.2, 'momentum': 0.6}
0.651042 (0.024774) with: {'learn_rate': 0.2, 'momentum': 0.8}
0.651042 (0.024774) with: {'learn_rate': 0.2, 'momentum': 0.9}
0.455729 (0.146518) with: {'learn_rate': 0.3, 'momentum': 0.0}
0.455729 (0.146518) with: {'learn_rate': 0.3, 'momentum': 0.2}
0.455729 (0.146518) with: {'learn_rate': 0.3, 'momentum': 0.4}
```

```
0.348958 (0.024774) with: {'learn_rate': 0.3, 'momentum': 0.6}
0.348958 (0.024774) with: {'learn_rate': 0.3, 'momentum': 0.8}
0.348958 (0.024774) with: {'learn_rate': 0.3, 'momentum': 0.9}
```

可以看到，SGD在该问题上相对表现不是很好，但当学习速率为0.01、动量因子为0.0时可取得最好的结果，正确率约为68%。

如何调优网络权值初始化

神经网络权值初始化一度十分简单：采用小的随机数即可。

现在，有许多不同的技术可供选择。 ***点击此处查看Keras 提供的清单*** 。

在本例中，我们将着眼于通过评估所有可用的技术，来调优网络权值初始化的选择。

我们将在每一层采用相同的权值初始化方法。理想情况下，根据每层使用的激活函数选用不同的权值初始化方法效果可能更好。在下面的例子中，我们在隐藏层使用了整流器（rectifier）。因为预测是二进制，因此在输出层使用了sigmoid函数。

完整代码如下：

```
# Use scikit-learn to grid search the weight initialization
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(init_mode='uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init=init_mode, activation='relu'))
    model.add(Dense(1, init=init_mode, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform', 'he_normal', 'he_un
param_grid = dict(init_mode=init_mode)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.720052 using {'init_mode': 'uniform'}
0.720052 (0.024360) with: {'init_mode': 'uniform'}
0.348958 (0.024774) with: {'init_mode': 'lecun_uniform'}
0.712240 (0.012075) with: {'init_mode': 'normal'}
0.651042 (0.024774) with: {'init_mode': 'zero'}
0.700521 (0.010253) with: {'init_mode': 'glorot_normal'}
0.674479 (0.011201) with: {'init_mode': 'glorot_uniform'}
0.661458 (0.028940) with: {'init_mode': 'he_normal'}
0.678385 (0.004872) with: {'init_mode': 'he_uniform'}
```

我们可以看到，当采用均匀权值初始化方案（uniform weight initialization ）时取得最好的结果，可以实现约72%的性能。

如何选择神经元激活函数

激活函数控制着单个神经元的非线性以及何时激活。

通常来说，整流器（rectifier）的激活功能是最受欢迎的，但应对不同的问题， sigmoid函数和tanh 函数可能是更好的选择。

在本例中，我们将探讨、评估、比较 ***Keras提供的不同类型的激活函数*** 。我们仅在隐层中使用这些函数。考虑到二元分类问题，需要在输出层使用sigmoid激活函数。

通常而言，为不同范围的传递函数准备数据是一个好主意，但在本例中我们不会这么做。

完整代码如下：

```
# Use scikit-learn to grid search the activation function
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(activation='relu'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init='uniform', activation=activation))
    model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']
param_grid = dict(activation=activation)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.722656 using {'activation': 'linear'}
0.649740 (0.009744) with: {'activation': 'softmax'}
0.720052 (0.032106) with: {'activation': 'softplus'}
0.688802 (0.019225) with: {'activation': 'softsign'}
0.720052 (0.018136) with: {'activation': 'relu'}
0.691406 (0.019401) with: {'activation': 'tanh'}
0.680990 (0.009207) with: {'activation': 'sigmoid'}
0.691406 (0.014616) with: {'activation': 'hard_sigmoid'}
0.722656 (0.003189) with: {'activation': 'linear'}
```

令人惊讶的是（至少对我来说是），“线性（linear）”激活函数取得了最好的效果，准确率约为72%。

如何调优Dropout正则化

在本例中，我们将着眼于调整正则化中的dropout速率，以期限制过拟合（overfitting）和提高模型的泛化能力。为了得到较好的结果，dropout最好结合一个如最大范数约束之类的权值约束。

了解更多dropout在深度学习框架Keras的使用请查看下面这篇文章：

- ***基于Keras/Python的深度学习模型Dropout正则项***

它涉及到拟合dropout率和权值约束。我们选定dropout percentages取值范围是：0.0-0.9（1.0无意义）；最大范数权值约束（maxnorm weight constraint）的取值范围是0-5。

完整代码如下：

```
# Use scikit-learn to grid search the dropout rate
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
# Function to create model, required for KerasClassifier
def create_model(dropout_rate=0.0, weight_constraint=0):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init='uniform', activation='linear', W_constraint=maxnorm(weight_constraint)
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
weight_constraint = [1, 2, 3, 4, 5]
dropout_rate = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
param_grid = dict(dropout_rate=dropout_rate, weight_constraint=weight_constraint)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.723958 using {'dropout_rate': 0.2, 'weight_constraint': 4}
0.696615 (0.031948) with: {'dropout_rate': 0.0, 'weight_constraint': 1}
0.696615 (0.031948) with: {'dropout_rate': 0.0, 'weight_constraint': 2}
0.691406 (0.026107) with: {'dropout_rate': 0.0, 'weight_constraint': 3}
0.708333 (0.009744) with: {'dropout_rate': 0.0, 'weight_constraint': 4}
0.708333 (0.009744) with: {'dropout_rate': 0.0, 'weight_constraint': 5}
0.710937 (0.008438) with: {'dropout_rate': 0.1, 'weight_constraint': 1}
0.709635 (0.007366) with: {'dropout_rate': 0.1, 'weight_constraint': 2}
0.709635 (0.007366) with: {'dropout_rate': 0.1, 'weight_constraint': 3}
0.695312 (0.012758) with: {'dropout_rate': 0.1, 'weight_constraint': 4}
0.695312 (0.012758) with: {'dropout_rate': 0.1, 'weight_constraint': 5}
0.701823 (0.017566) with: {'dropout_rate': 0.2, 'weight_constraint': 1}
```

```
0.710938 (0.009568) with: {'dropout_rate': 0.2, 'weight_constraint': 2}
0.710938 (0.009568) with: {'dropout_rate': 0.2, 'weight_constraint': 3}
0.723958 (0.027126) with: {'dropout_rate': 0.2, 'weight_constraint': 4}
0.718750 (0.030425) with: {'dropout_rate': 0.2, 'weight_constraint': 5}
0.721354 (0.032734) with: {'dropout_rate': 0.3, 'weight_constraint': 1}
0.707031 (0.036782) with: {'dropout_rate': 0.3, 'weight_constraint': 2}
0.707031 (0.036782) with: {'dropout_rate': 0.3, 'weight_constraint': 3}
0.694010 (0.019225) with: {'dropout_rate': 0.3, 'weight_constraint': 4}
0.709635 (0.006639) with: {'dropout_rate': 0.3, 'weight_constraint': 5}
0.704427 (0.008027) with: {'dropout_rate': 0.4, 'weight_constraint': 1}
0.717448 (0.031304) with: {'dropout_rate': 0.4, 'weight_constraint': 2}
0.718750 (0.030425) with: {'dropout_rate': 0.4, 'weight_constraint': 3}
0.718750 (0.030425) with: {'dropout_rate': 0.4, 'weight_constraint': 4}
0.722656 (0.029232) with: {'dropout_rate': 0.4, 'weight_constraint': 5}
0.720052 (0.028940) with: {'dropout_rate': 0.5, 'weight_constraint': 1}
0.703125 (0.009568) with: {'dropout_rate': 0.5, 'weight_constraint': 2}
0.716146 (0.029635) with: {'dropout_rate': 0.5, 'weight_constraint': 3}
0.709635 (0.008027) with: {'dropout_rate': 0.5, 'weight_constraint': 4}
0.703125 (0.011500) with: {'dropout_rate': 0.5, 'weight_constraint': 5}
0.707031 (0.017758) with: {'dropout_rate': 0.6, 'weight_constraint': 1}
0.701823 (0.018688) with: {'dropout_rate': 0.6, 'weight_constraint': 2}
0.701823 (0.018688) with: {'dropout_rate': 0.6, 'weight_constraint': 3}
0.690104 (0.027498) with: {'dropout_rate': 0.6, 'weight_constraint': 4}
0.695313 (0.022326) with: {'dropout_rate': 0.6, 'weight_constraint': 5}
0.697917 (0.014382) with: {'dropout_rate': 0.7, 'weight_constraint': 1}
0.697917 (0.014382) with: {'dropout_rate': 0.7, 'weight_constraint': 2}
0.687500 (0.008438) with: {'dropout_rate': 0.7, 'weight_constraint': 3}
0.704427 (0.011201) with: {'dropout_rate': 0.7, 'weight_constraint': 4}
0.696615 (0.016367) with: {'dropout_rate': 0.7, 'weight_constraint': 5}
0.680990 (0.025780) with: {'dropout_rate': 0.8, 'weight_constraint': 1}
0.699219 (0.019401) with: {'dropout_rate': 0.8, 'weight_constraint': 2}
0.701823 (0.015733) with: {'dropout_rate': 0.8, 'weight_constraint': 3}
0.684896 (0.023510) with: {'dropout_rate': 0.8, 'weight_constraint': 4}
0.696615 (0.017566) with: {'dropout_rate': 0.8, 'weight_constraint': 5}
0.653646 (0.034104) with: {'dropout_rate': 0.9, 'weight_constraint': 1}
0.677083 (0.012075) with: {'dropout_rate': 0.9, 'weight_constraint': 2}
0.679688 (0.013902) with: {'dropout_rate': 0.9, 'weight_constraint': 3}
0.669271 (0.017566) with: {'dropout_rate': 0.9, 'weight_constraint': 4}
0.669271 (0.012075) with: {'dropout_rate': 0.9, 'weight_constraint': 5}
```

我们可以看到，当 dropout率为0.2%、最大范数权值约束（ maxnorm weight constraint）取值为4时，可以取得准确率约为72%的最好结果。

如何确定隐藏层中的神经元的数量

每一层中的神经元数目是一个非常重要的参数。通常情况下，一层之中的神经元数目控制着网络的代表性容量，至少是拓扑结构某一节点的容量。

此外，一般来说，一个足够大的单层网络是接近于任何神经网络的，至少在理论上成立。

在本例中，我们将着眼于调整单个隐藏层神经元的数量。取值范围是：1—30，步长为5。

一个大型网络要求更多的训练，此外，至少批尺寸（batch size）和 epoch的数量应该与神经元的数量优化。

完整代码如下：

```
# Use scikit-learn to grid search the number of neurons
import numpy
from sklearn.grid_search import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
# Function to create model, required for KerasClassifier
def create_model(neurons=1):
    # create model
```

```
    model = Sequential()
    model.add(Dense(neurons, input_dim=8, init='uniform', activation='linear', W_constraint=maxnorm(4)))
    model.add(Dropout(0.2))
    model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=100, batch_size=10, verbose=0)
# define the grid search parameters
neurons = [1, 5, 10, 15, 20, 25, 30]
param_grid = dict(neurons=neurons)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

运行之后输出如下：

```
Best: 0.714844 using {'neurons': 5}
0.700521 (0.011201) with: {'neurons': 1}
0.714844 (0.011049) with: {'neurons': 5}
0.712240 (0.017566) with: {'neurons': 10}
0.705729 (0.003683) with: {'neurons': 15}
0.696615 (0.020752) with: {'neurons': 20}
0.713542 (0.025976) with: {'neurons': 25}
0.705729 (0.008027) with: {'neurons': 30}
```

我们可以看到，当网络中隐藏层内神经元的个数为5时，可以达到最佳结果，准确性约为71%。

超参数优化的小技巧

本节罗列了一些神经网络超参数调整时常用的小技巧。

- **K层交叉检验（k-fold Cross Validation），** 你可以看到，本文中的不同示例的结果存在一些差异。使用了默认的3层交叉验证，但也许K=5或者K=10时会更加稳定。认真选择您的交叉验证配置，以确保您的结果是稳定的。
- **审查整个网络。** 不要只注意最好的结果，审查整个网络的结果，并寻找支持配置决策的趋势。
- **并行（Parallelize），** 如果可以，使用全部的CPU，神经网络训练十分缓慢，并且我们经常想尝试不同的参数。参考AWS实例。
- **使用数据集的样本。** 由于神经网路的训练十分缓慢，尝试训练在您训练数据集中较小样本，得到总方向的一般参数即可，并非追求最佳的配置。
- **从粗网格入手。** 从粗粒度网格入手，并且一旦缩小范围，就细化为细粒度网格。
- **不要传递结果。** 结果通常是特定问题。尽量避免在每一个新问题上都采用您最喜欢的配置。你不可能将一个问题的最佳结果转移到另一个问题之上。相反地，你应该归纳更广泛的趋势，例如层的数目或者是参数之间的关系。
- **再现性（Reproducibility）是一个问题。** 在NumPy中，尽管我们为随机数发生器设置了种子，但结果并非百分百重现。网格搜索wrapped Keras模型将比本文中所示Keras模型展现更多可重复性（reproducibility）。

总结

在这篇文章中，你可以了解到如何使用Keras和scikit-learn/Python调优神经网络中的超参数。

尤其是可以学到：

- 如何包装Keras模型以便在scikit-learn使用以及如何使用网格搜索。

- 如何网格搜索Keras 模型中不同标准的神经网络参数。
- 如何设计自己的超参数优化实验。