第 5 章

CHAPTER 5

模型精度的提高

本章介绍了神经网络正则化的概念。正则化旨在防止模型在训练过程中的过拟合，并为在新的未知的数据上进行测试提供更准确的结果。这一章将学习使用不同的正则化技术（L1 和 L2 正则化及丢弃正则化）来提高模型性能。正则化是一个重要的组成部分，因为它可以防止神经网络过拟合训练数据，并帮助我们构建健壮、准确的模型，这些模型在新的、未知的数据上表现良好。学完本章，读者就能在 scikit-learn 中实现网格搜索和随机搜索，并找到最佳的超参数。

5.1 简介

第 4 章学习了使用交叉验证这一方法无偏差地测试各种超参数，以便精确地构建神经网络。使用留一交叉验证法，在训练过程中留下一条记录用于验证，并对数据集中的每条记录重复此操作。然后，使用 k-fold 交叉验证法，将训练数据集拆分为 k 个子数据集，用 $k-1$ 个子数据集训练模型，并使用最后一个子数据集进行验证。这些交叉验证方法能够使用不同的超参数训练模型并用无偏差的数据测试模型。

深度学习不仅仅是建立神经网络，还要使用可用的数据集训练它们，并报告模型的正确率。它涉及理解模型和数据集，以及多维度地对模型进行提升。本章将介绍两项非常重要的技术用于整体地改进机器学习模型，特别是深度学习模型。这两项技术就是正则化和超参数调整。

本章将详细介绍正则化方法。首先阐述为什么需要这个方法，以及它如何帮助我们。然后，将介绍两种最重要和最常用的正则化技术，即参数正则化和它的两个变体——L1 和 L2 范数正则化。接着，介绍一种专门为神经网络设计的正则化技术，即丢弃正则化。通过完成涉及现实生活的数据集的实践，练习在 Keras 模型上实现这些技术中的每一步操作。最后，还会简要介绍一些其他的正则化技术。

本章还会讨论超参数微调的重要性，特别是对深度学习模型而言。首先探讨微调超参数的值为何极大地影响模型的正确率，以及微调超参数会面临的挑战。然后学习 scikit-learn 中两个非常有用的方法，它们可以在 Keras 模型上进行超参数微调。接着学习每种方法的优点和缺点，以及如何将它们结合在一起，使之成为一种新的方法。最后，练习使用 scikit-learn 的

优化器来实现 Keras 模型的超参数微调。

5.2 正则化

由于深度神经网络是高度灵活的模型，过拟合是训练时经常出现的问题，因此检测和解决过拟合是一名深度学习者必备的技能。如果模型在训练数据上表现出色，但在新的未知的数据上表现很差，那么很明显这个模型过拟合了。

例如，建立一个用于区分猫和狗的图像分类模型，该模型在训练过程中表现出很高的正确率，但在新的例子中却表现不佳，这就表明该模型对训练数据进行了过度训练。正则化技术是专门用于减少机器学习模型中的过拟合现象的一组重要方法。

彻底了解正则化技术并将它们应用于深层神经网络是构建深层神经网络以解决现实问题的关键一步。本节将介绍正则化的基本概念，这是学习以下各节的基础。

5.2.1 正则化的需求

机器学习的主要目的是建立一个模型，这个模型不仅要在训练例子上表现良好，在新例子上也要表现良好。一个好的机器学习模型可以找到真正底层过程/功能的形式和参数生成训练示例，但不会捕获与单个训练示例相关的噪声。这样的机器学习模型可以很好地泛化到稍后相同过程产生的新数据。

之前讨论的方法——例如将数据集拆分为训练集和测试集，以及交叉验证——都是为了估计训练模型的泛化能力而设计的。事实上，用来指代测试集误差和交叉验证误差的术语是“泛化误差”，表示在训练中未使用示例的错误率。而机器学习的主要目的是建立具有低泛化错误率的模型。

在第 3 章的学习中，讨论了两个非常重要的关于机器学习模型的问题：过拟合和欠拟合。欠拟合是指预测模型不够灵活或复杂，以至于捕捉不到所有与真实过程相关的关系和模式。欠拟合会导致模型高偏差，当训练误差很高时会被检测出来。过拟合是指预测模型过于灵活或复杂，导致模型高方差，当训练误差和泛化误差之间存在较大差距时，会被诊断出来。图 5.1 中展示了以上情况在一个二进制分类问题上的表现。

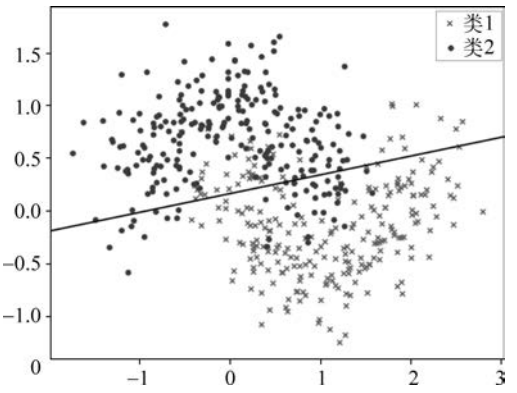


图 5.1 欠拟合

正如图 5.1 所示，欠拟合问题比过拟合问题更好解决。通过使模型更灵活/更复杂，可以很容易地解决欠拟合问题。在深度神经网络中，这意味着改变网络的架构，通过向网络添加更多层或增加层中的单元数量来使网络变得更复杂。

再看一下过拟合的图示，如图 5.2 所示。

有一些简单的解决方案可以解决过拟合问题，例如降低模型的灵活性/复杂性（同样，通过改变网络的架构）或为网络提供更多的训练示例。然而，降低网络的复杂性有时会以大幅增加偏差或训练错误率为代价。其原因是，大多数时候导

致过拟合的原因不是模型的灵活性，而是训练示例太少。另一方面，提供更多数据示例以减少过拟合并不总是可行的。因此，如何在保持模型复杂度和训练样本数量不变的情况下减少泛化误差是一项重要而又具有挑战性的工作。

再看一下模型正拟合的图示，如图 5.3 所示。

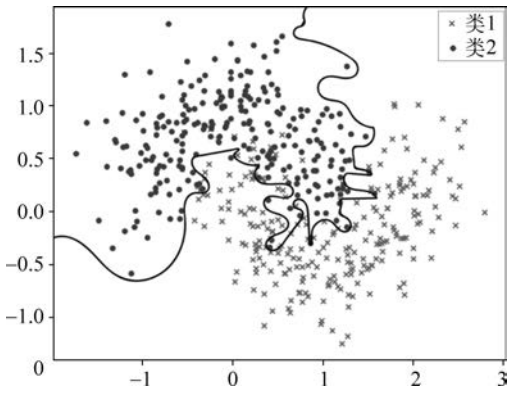


图 5.2　过拟合

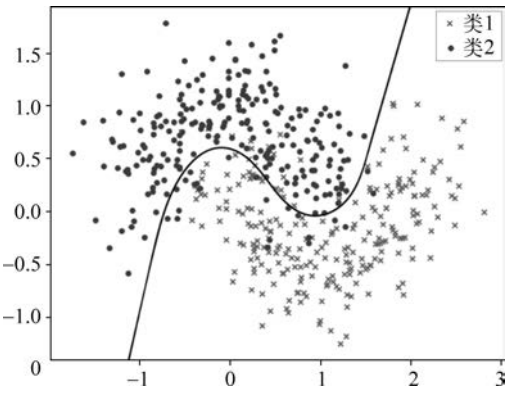


图 5.3　正拟合

图 5.3 显示了在构建高度灵活的机器学习模型(例如深度神经网络)时需要正则化技术来抑制模型的灵活性，使其不能对单个例子进行过拟合。下一节将学习正则化方法如何减少模型在训练数据上的过拟合，以降低模型的方差。

5.2.2　用正则化减少过拟合

正则化方法以减少模型方差的方式修改算法。通过减少方差，正则化技术可以在不增加训练误差(或者至少不大幅增加训练误差)的情况下减少泛化误差。

正则化方法提供了某种限制，有助于提升模型稳定性，有很多种方法可以实现这一点。在深度神经网络上执行正则化最常见的方法之一是在权重上添加某种类型的惩罚项以保持权重较小。

一方面，保持较小的权重可以降低网络对单个数据示例中噪声的敏感度。事实上，神经网络中的权重是决定每个处理单元对网络最终输出的影响大小的系数。如果这些单位的权重较大，这意味着它们中的每一个都将对权重产生重大影响。每个处理单元造成的所有重大影响相结合，最终的输出将产生很大波动。

另一方面，保持较小的权重会减少每个单元对最终输出的影响。实际上，通过将权重保持在零附近，一些单元对输出几乎没有影响。如果训练一个大型神经网络，其中每个单元对输出的影响很小或没有影响，这相当于训练一个更简单的网络，因此减少了方差和过拟合。图 5.4 显示了正则化如何将大型网络中某些单元的影响归零。

图 5.4 是正则化过程的示意图。顶部是一个没有正则化的网络，而底部是一个具有正则化的网络，其中白色单元表示对输出几乎没有影响的单元，因为它们受到正则化的惩罚。

目前为止，已经了解了正则化的概念。下一节将介绍深度学习模型最常用的正则化方法——L1、L2 和丢弃正则化——以及如何在 Keras 中实现它们。

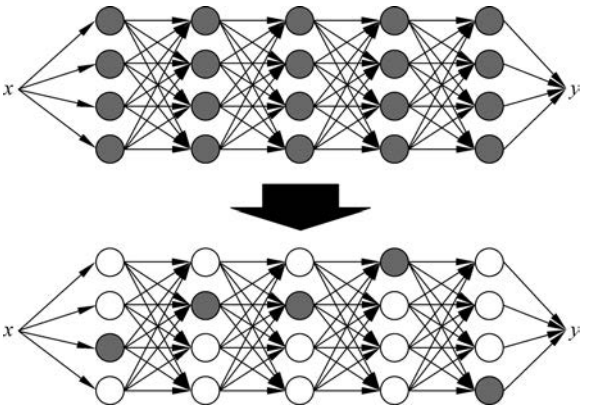


图 5.4　正则化如何将大型网络中某些单元的影响归零

5.3　L1 和 L2 正则化

深度学习模型最常见的正则化类型是保持网络的权重较小。这种类型的正则化称为权重正则化，有两种不同的种类：L1 正则化和 L2 正则化。本节将详细介绍这些正则化方法，以及如何在 Keras 中实现它们。接下来还会将其应用到现实问题中，并观察它们是如何提高模型性能的。

5.3.1　L1 和 L2 正则化公式

在权重正则化中，损失函数中被添加了一个惩罚项，称为权重的 L1 范数（绝对值之和）或权重的 L2 范数（平方和）。如果使用 L1 范数，则称为 L1 正则化。如果使用 L2 范数，则称为 L2 正则化。无论是哪种情况，最后得到的总和都会乘以 λ 正则化超参数。

因此 L1 正则化公式为

$$\text{损失函数} = \text{旧损失函数} + \lambda \times \text{权重绝对值之和}$$

L2 正则化公式为

$$\text{损失函数} = \text{旧损失函数} + \lambda \times \text{权重的平方和}$$

λ 可以取 0 到 1 之间的任何值，其中 $\lambda = 0$ 表示完全没有惩罚（相当于没有正则化的网络），$\lambda = 1$ 表示完全惩罚。

与其他所有超参数一样，正确的 λ 值可以通过尝试不同的值并观察哪个值提供较低的泛化误差来选择。事实上，先从不具有正则化的网络观察结果是一种很好的做法。然后，不断地增加 λ 值执行正则化，例如 0.001，0.01，0.1，0.5，…，并观察每种情况下的结果，以确定哪个正则化参数的值对一个特定问题的权重值的惩罚合适。

在正则化优化算法的每次迭代中，权重（w）变得越来越小，所以权重正则化通常被称为权重衰减。

目前为止，只在深度神经网络中对权重进行正则化。但是同样的过程也可以应用于偏差。更准确地说，通过添加一个偏差惩罚项来再次更新损失函数，以保持在神经网络的训练过程中偏差值始终很小。

说明：如果通过向损失函数添加两项惩罚来执行正则化(一项用于惩罚权重，另一项用于惩罚偏差)，那么会将其称为参数正则化而不是权重正则化。

然而，正则化偏差值在深度学习中并不常见，因为权重对于神经网络是更重要的参数。事实上，添加权重正则化之后，再添加偏差正则化不会导致明显的结果改变。

L2 正则化是机器学习中最常用的正则化技术。一方面，L1 正则化和 L2 正则化之间的区别在于 L1 会产生一个更稀疏的权重矩阵，这意味着 L1 有更多的权重等于 0，因此更多的节点会完全从网络中移除。另一方面，L2 正则化极大地降低了权重，但同时等于 0 的权重较少。当然也可以同时执行 L1 和 L2 正则化。

现在已经了解了 L1 和 L2 正则化的工作原理，接下来就需要在 Keras 的深度神经网络上来实现 L1 和 L2 正则化了。

5.3.2 Keras 的 L1 和 L2 正则化实现

Keras 提供了一个正则化 API，用于将惩罚项添加到损失函数中以便对深度神经网络每一层的权重或偏差进行正则化。定义惩罚项或正则化，需要在 keras.regularizers 中定义所需的正则化方法。

例如，要定义一个 $\lambda=0.01$ 的 L1 正则器，可以这么写。

```
from keras.regularizers import l1
keras.regularizers.l1(0.01)
```

同样，要定义一个 $\lambda=0.01$ 的 L2 正则器，可以这么写。

```
from keras.regularizers import l2
keras.regularizers.l2(0.01)
```

最后，要使用 $\lambda=0.01$ 同时定义 L1 和 L2 正则器，可以这么写。

```
from keras.regularizers import l1_l2
keras.regularizers.l1_l2(l1=0.01, l2=0.01)
```

每一个 regularizer 都可以应用于层中的权重和/或偏差。例如，如果想对具有 8 个节点的密集层的权重和偏差应用 L2 正则化($\lambda=0.01$)，可以这么写。

```
from keras.layers import Dense
from keras.regularizers import l2
model.add(Dense(8, kernel_regularizer=l2(0.01), \
          bias_regularizer=l2(0.01)))
```

接下来在实践 5.01 中进一步练习 L1 和 L2 正则化的实现。该实践将在 Avila 数据集的深度学习模型上应用正则化，观察结果与之前相比有何变化。

说明：本章中的所有实践都将在 Jupyter Notebook 中开发。在 https://packt.live/2OOBjqq 下载本书的 GitHub 存储库和所有准备好的模板。

实践 5.01 Avila模式分类器上的权重正则化

Avila 数据集是从 Avila Bible 的 800 张图像中提取的，该数据集包括文本图像的各种特征，例如列间距和文本的边距。该数据集还包含一个类标签，代表图像的格式是否属于常出现的类别。在本实践中，构建一个 Keras 模型，并根据给定的网络架构和超参数值对该数据集进行分类。目的是在模型上应用不同类型的权重正则化，并观察每种类型如何改变结果。

出于以下两个原因，使用训练集/测试集的方法在本次实践中进行评估。首先，由于需要尝试几种不同的正则化器，因此执行交叉验证需要很长时间。其次，还需要绘制训练误差和测试误差的趋势，以便直观地了解正则化是如何防止模型过拟合数据示例的。

按照下述步骤完成本实践。

(1) 使用命令 X = pd.read_csv('../data/avila-tr_feats.csv')和 y = pd.read_csv ('../data/ avila-tr_target.csv')从 GitHub 的 Chapter05 中加载 data 子文件夹。使用 sklearn.model _selection.train_test_split 方法对训练集和测试集进行拆分，保留 20%的数据为测试数据。

(2) 定义一个具有三个隐藏层的 Keras 模型，第一个大小为 10，第二个大小为 6，第三个大小为 4 来执行分类。将这些值用于以下超参数：activation='relu', loss='binary_ crossentropy', optimizer='sgd',metrics=['accuracy'],batch_size=20,epochs=100,shuffle=False。

(3) 在训练集上训练模型并使用测试集对其进行评估，存储每次迭代的训练损失和测试损失。训练完成后，绘制训练误差和测试误差的趋势图(将纵轴的界限改为(0,1)，以便更好地观察损失的变化)。测试集上的最小错误率是多少？

(4) 将 λ=0.01 的 L2 正则化器添加到模型的隐藏层并重复训练。训练完成后，绘制训练误差和测试误差的趋势。测试集上的最小错误率是多少？

(5) 使用 λ=0.1 和 λ=0.005 重复步骤(4)，针对每个 λ 值训练模型，并报告结果。在这个深度学习模型和这个数据集上执行 L2 正则化时，哪个 λ 值是更好的选择？

(6) 重复步骤(5)，这次使用 λ=0.01 和 λ=0.005 的 L1 正则化器，为每个 λ 值训练模型并报告结果。在这个深度学习模型和此数据集上执行 L1 正则化时，哪个 λ 值是更好的选择？

(7) 将 L1 λ=0.005 和 L2 λ=0.005 的 L1_L2 正则化器添加到模型的隐藏层并重复训练。训练完成后，绘制训练误差和测试误差的趋势。测试集上的最小错误率是多少？

完成上述步骤后，预期输出结果如图 5.5 所示。

彩色图片

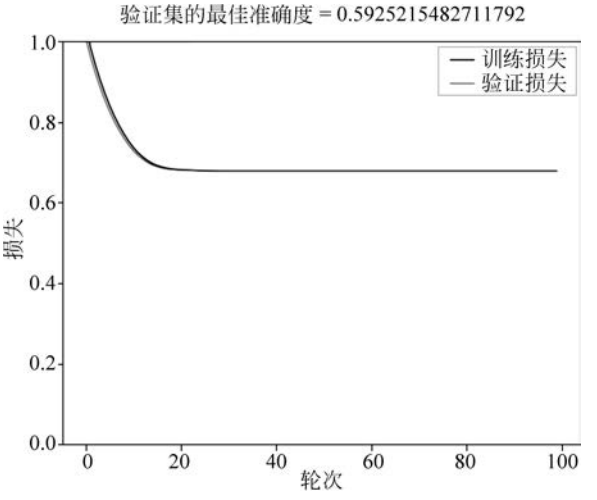


图 5.5　模型在训练期间的训练误差和验证误差图

说明：该实践的答案见附录 A 的实践 5.01 Avila 模式分类器上的权重正则化。

在本实践中，针对实际问题实施 L1 和 L2 权重正则化，并将正则化模型的结果与未进行任何正则化的模型结果进行了比较。下一节将探讨丢弃正则化。

5.4　丢弃正则化

本节将了解丢弃正则化的工作原理、其是如何实现减少过拟合的以及如何使用 Keras 实现它。最后，通过完成一项涉及真实数据集的实践来练习所学到的关于丢弃正则化的知识。

5.4.1　丢弃正则化原理

丢弃正则化在训练期间通过从神经网络中随机删除节点来工作。更准确地说，丢弃正则化在每个节点上设置了一个概率。这个概率是指节点在学习算法的每次迭代中被包含在训练中的机会。假如有一个大型神经网络，其中为每个节点分配了 0.5 的丢弃正则化机会。在这种情况下，每次迭代中，学习算法都会为每个节点掷硬币，以决定该节点是否会从网络中移除。图 5.6 中可以看到此类过程的说明。

这个过程在每次迭代中重复，这意味着每次迭代随机选择的节点都会从网络中移除，也意味着参数更新过程是在不同的较小网络上完成的。例如，图 5.6 所示的底部显示的网络将仅被用于一次训练迭代。对于下一次迭代，一些其他随机选择的节点将从顶部网络中删除，因此删除这些节点所产生的网络将与图中的底部网络不同。

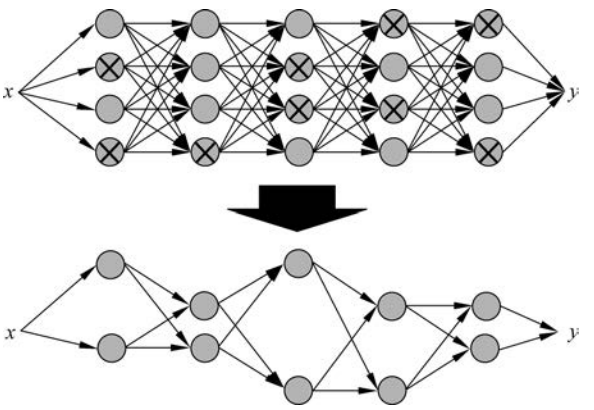


图 5.6　从深度神经网络中删除节点的丢弃正则化

当学习算法在迭代中选择删除/忽略某些节点时，这意味着这些节点在该迭代中根本不会参与参数的更新过程。更准确地说，预测输出的前向传播、损失计算和计算导数的反向传播都是在较小的网络上完成的，但去掉了一些节点。因此，参数更新将仅在该迭代中出现在网络中的节点上进行；被移除节点的权重和偏差不会更新。

需要注意的是，丢弃正则化需要在删除了随机选择的节点所建立的网络上进行训练，评估必须使用原始网络进行。如果对删除了随机节点的网络进行评估，结果中会引入噪声，这是不可取的。下一节将了解为什么丢弃正则化有助于防止过拟合。

5.4.2　使用丢弃正则化减少过拟合

本节将讨论丢弃正则化方法的概念。正如之前所讨论的，正则化技术的目标是防止模型产生过拟合。因此，我们将探讨从神经网络中随机删除部分节点是如何有助于减少方差和过拟合的。

对这一点最简单的解释是，通过从网络中删除节点，可以在更小的网络上进行训练，较小的神经网络灵活性较低，因此网络对数据过拟合的可能性较低。

丢弃正则化之所以能更好地减少过拟合还有另一个原因。通过在深度神经网络的每一层

随机去除输入，整个网络对单个输入的敏感性降低。在训练神经网络时，权重会以最终模型适合训练示例的方式更新。通过从训练过程中随机移除一些权重，丢弃正则化强制其他权重参与到与训练示例相关的迭代学习中，因此最终的权重将分布得更广泛。

换句话说，这不是因为某些权重为了适应输入值而更新了太多次，而是所有权重都参与学习了这些输入值，因此过拟合减少了。这就是为什么与较小的简单网络相比，丢弃正则化会在更大的网络上对于新的未见过的数据表现更好。

现在我们了解了丢弃正则化的基本过程及其有效性的原因，接下来继续在 Keras 中实现丢弃正则化。

训练 5.01 使用Keras实现丢弃正则化

丢弃正则化是 Keras 的核心层，因此可以像在网络中添加其他层一样，向模型添加丢弃正则化层。在 Keras 中定义丢弃正则化层时，需要提供 rate 超参数。rate 可以取 0 到 1 之间的任何值，代表确定要删除或忽略的输入单元的部分。在本训练中将逐步学习使用丢弃正则化层实现 Keras 深度学习模型。

模拟数据集是树木的各种测量值，例如高度、树枝数量和树干的周长。要根据给定的测量值将记录分类为落叶树(类别值为 1)或针叶树(类别值为 0)。该数据集由 10000 条记录组成，这些记录由两个类组成，代表两种树类型，每个数据示例有 10 个特征值。按照以下步骤完成此训练。

(1) 首先，执行以下代码块加载数据集并将其拆分为训练集和测试集。

```
# Load the data
import pandas as pd
X = pd.read_csv('../data/tree_class_feats.csv')
y = pd.read_csv('../data/tree_class_target.csv')

"""
Split the dataset into training set and test set with a 80-20 ratio
"""
from sklearn.model_selection import train_test_split
seed = 1
X_train, X_test, \
y_train, y_test = train_test_split(X, y, \
                                   test_size=0.2, \
                                   random_state=seed)
```

(2) 导入所有的依赖项。构建一个没有丢弃正则化的四层 Keras 序列模型。构建一个有 16 个单元的第一隐藏层、12 个单元的第二隐藏层、8 个单元的第三隐藏层、4 个单元的第四隐藏层的网络，所有单元都加入 ReLU 激活函数。添加一个带有 sigmoid 激活函数的输出层。

```
#Define your model
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
from tensorflow import random

np.random.seed(seed)
random.set_seed(seed)
```

```
model_1 = Sequential()
model_1.add(Dense(16, activation='relu', input_dim=10))
model_1.add(Dense(12, activation='relu'))
model_1.add(Dense(8, activation='relu'))
model_1.add(Dense(4, activation='relu'))
model_1.add(Dense(1, activation='sigmoid'))
```

(3) 使用二元交叉熵作为损失函数和 sgd 作为优化器编译模型,并在训练集上使用 batch_size=50 训练模型 300 个轮次。然后,在测试集上评估训练好的模型。

```
model_1.compile(optimizer='sgd', loss='binary_crossentropy')
# train the model
model_1.fit(X_train, y_train, epochs=300, batch_size=50, \
            verbose=0, shuffle=False)
# evaluate on test set
print("Test Loss =", model_1.evaluate(X_test, y_test))
```

预期输出如下。

```
2000/2000 [==============================] - 0s 23us/step
Test Loss = 0.1697693831920624
```

因此,在训练模型 300 个轮次后,预测树种的测试错误率为 16.98%。

(4) 重新定义一个与先前一样的模型,但是添加一个 rate=0.1 的丢弃正则化到模型的第一个隐藏层,并在测试数据上重复模型的编译、训练和评估步骤。

```
"""
define the keras model with dropout in the first hidden layer
"""
from keras.layers import Dropout
np.random.seed(seed)
random.set_seed(seed)
model_2 = Sequential()
model_2.add(Dense(16, activation='relu', input_dim=10))
model_2.add(Dropout(0.1))
model_2.add(Dense(12, activation='relu'))
model_2.add(Dense(8, activation='relu'))
model_2.add(Dense(4, activation='relu'))
model_2.add(Dense(1, activation='sigmoid'))

model_2.compile(optimizer='sgd', loss='binary_crossentropy')
# train the model
model_2.fit(X_train, y_train, \
            epochs=300, batch_size=50, \
            verbose=0, shuffle=False)
# evaluate on test set
print("Test Loss =", model_2.evaluate(X_test, y_test))
```

预期输出如下。

```
2000/2000 [==============================] - 0s 29us/step
Test Loss = 0.16891103076934816
```

在网络第一层加入 rate=0.1 的丢弃正则化后,测试错误率从 16.98%降低到 16.89%。

(5) 重新定义一个与先前一样的模型,但是添加一个 rate=0.2 的丢弃正则化到模型的第一个隐藏层,再对每个剩余层都添加一个 rate=0.1 的丢弃正则化,并在测试数据上重复模型的编译、训练和评估步骤。

```
# define the keras model with dropout in all hidden layers
np.random.seed(seed)
random.set_seed(seed)

model_3 = Sequential()
model_3.add(Dense(16, activation='relu', input_dim=10))
model_3.add(Dropout(0.2))
model_3.add(Dense(12, activation='relu'))
model_3.add(Dropout(0.1))
model_3.add(Dense(8, activation='relu'))
model_3.add(Dropout(0.1))
model_3.add(Dense(4, activation='relu'))
model_3.add(Dropout(0.1))
model_3.add(Dense(1, activation='sigmoid'))

model_3.compile(optimizer='sgd', loss='binary_crossentropy')
# train the model
model_3.fit(X_train, y_train, epochs=300, \
            batch_size=50, verbose=0, shuffle=False)
# evaluate on test set
print("Test Loss =", model_3.evaluate(X_test, y_test))
```

预期输出如下。

```
2000/2000 [==============================] - 0s 40us/step
Test Loss = 0.19390961921215058
```

在第一层保持 rate=0.2 的丢弃正则化，同时在后续层添加 rate=0.1 的丢弃正则化，测试错误率从 16.89%增加到 19.39%。与 L1 和 L2 正则化一样，添加过多的丢弃正则化会阻止模型学习与训练数据相关的底层函数，导致比没有丢弃正则化时偏差更高。

正如在本训练中看到的那样，还可以根据在这些层中可能发生的过拟合程度，对不同的层应用不同 rate 的丢弃正则化。通常不希望在输入层和输出层上执行丢弃正则化。对于隐藏层，需要调整速率值并观察结果，以确定最适合特定问题的值。

说明：源代码网址为 https://packt.live/3iugM7K。在线运行代码网址为 https://packt.live/31HlSYo。

在下一个实践中，将对 Traffic Volume 数据集实现 Keras 深度学习模型及丢弃正则化。

实践 5.02 Traffic Volume数据集的丢弃正则化

在实践 4.03 中使用 Keras 包装器并通过交叉验证评估了模型。当给定与交通数据相关的各种标准化特征(如当天和前一天的交通量等)时，使用交通量数据集构建了一个模型来预测穿过城市桥梁的交通量。这个数据集包含 10000 条记录，每条记录包含 10 个属性/特征。

接下来基于实践 4.03 中的模型来开始本实践。使用训练集/测试集方法来训练和评估模型，绘制训练误差和泛化误差的趋势图，并观察模型是如何过拟合数据示例的。然后，尝试通过使用丢弃正则化解决过拟合问题来提高模型性能。最后，尝试找出应该向哪些层添加丢弃正则化，以及哪个 rate 值将会最大限度地改进模型。按照以下步骤完成此实践。

(1) 使用 Pandas 的 read_csv 函数加载数据集。数据集也存储在 GitHub 上的 Chapter05 的 data 子文件夹中。按照 80：20 的比例将数据集拆分为一个训练集和一个测试集。

(2) 定义具有两个大小为 10 的隐藏层的 Keras 模型来预测交通量。接下来，将这些值用

于下面这些超参数：activation＝'relu'，loss＝'mean_squared_error'，optimizer＝'rmsprop'，batch_size＝50，epochs＝200，shuffle＝False。

(3) 在训练集上训练模型并在测试集上进行评估，存储每次迭代的训练损失和测试损失。

(4) 训练完成后，绘制训练误差和测试误差的趋势图。最后，计算训练集和测试集的最低错误率是多少。

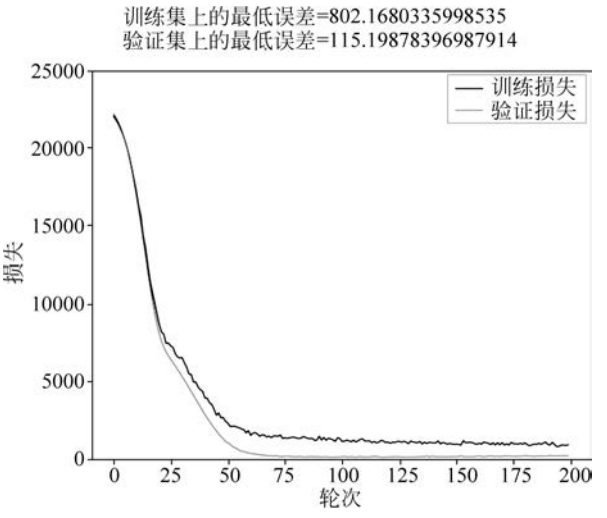
(5) 将 rate＝0.1 的丢弃正则化添加到模型的第一个隐藏层中，并重复此训练过程(因为使用丢弃正则化进行训练需要很长的时间，所以训练 200 个轮次)。训练完成后，绘制训练误差和测试误差的趋势图。最后，计算训练集和测试集的最低错误率是多少。

(6) 重复步骤(5)，将 rate＝0.1 的丢弃正则化添加到模型的两个隐藏层中，训练模型并报告结果。

(7) 重复步骤(6)，这次在第一层设置 rate＝0.2，在第二层设置 rate＝0.1，训练模型并报告结果。

(8) 最终，在这个深度学习模型和这个数据集上，选出性能最好的丢弃正则化方法。

预期输出如图 5.7 所示。



彩色图片

图 5.7　使用丢弃正则化训练模型时的训练误差和验证误差图

说明：*该实践答案见附录 A 的实践 5.02 Traffic Volume 数据集的丢弃正则化。*

本实践中我们一起学习了如何在 Keras 中实现丢弃正则化，并在交通流量数据集上训练如何使用它。丢弃正则化是专门为减少神经网络中的过拟合而设计的，其工作原理是在训练过程中随机地从神经网络中移除节点。这个过程会使神经网络的权重值均匀分布，从而减少单个数据示例的过拟合。下一节将讨论其他防止模型过拟合的正则化方法。

5.5　其他正则化方法

本节将简要介绍一些其他我们常用的并且已被证明在深度学习中有效的正则化技术。因

为正则化是机器学习中一个广泛而活跃的研究领域，所以不可能在一章中涵盖所有可用的正则化方法。因此，本节将简要介绍另外三种正则化方法：早停、数据增强和添加噪声。了解它们的基本思想，并获得一些如何使用它们的提示和建议。

5.5.1 在 Keras 中实现早停

在本章前面讨论了机器学习的一个主要假设是有一个真实的函数或过程可以产生训练样本。然而，这个过程是未知的，也没有明确的方法可以找到它。不仅没有办法找到确切的底层流程，而且选择一个具有适当级别的灵活性或复杂性的模型来评估流程也是具有挑战性的。因此，一种好的做法是选择高度灵活的模型，例如深度神经网络，来对过程进行建模并仔细监控训练过程。

通过监控训练过程可以训练模型，使其能够捕获训练过程，并且可以在它开始过度适应单个数据示例之前停止训练。这就是早停的基本思想。在第 3 章的模型评估部分就简要讨论了早停的想法。通过在训练过程中监测、观察训练误差和测试误差的变化，可以确定训练多少合适。

图 5.8 显示了在数据集上训练高度灵活的模型时训练误差和测试误差的变化，训练需要在标记为“合适的拟合”的区域中停止，以避免过拟合。

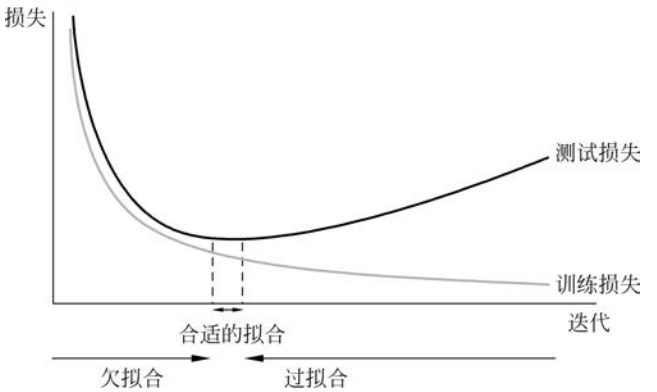


图 5.8 训练模型时的训练误差和测试误差图

在第 3 章中练习了通过存储和绘制训练误差和测试误差的变化来识别过拟合。在训练 Keras 模型时提供验证集或测试集，并使用以下代码在每个训练时期存储它们的指标值。

```
history=model.fit(X_train, y_train, validation_data=(X_test, y_test), \
                  epochs=epochs)
```

本节将学习如何在 Keras 中实现早停，即当所需指标（如测试错误率）不再提高时，迫使 Keras 模型停止训练。为此需要定义一个 EarlyStopping()回调，并将其作为参数提供给 model.fit()。

定义 EarlyStopping()回调时，需要为其提供正确的参数。第一个参数是监视器，它确定在训练期间将监控什么指标，以执行早停。通常 monitor='val_loss'是一个很好的选择，这意味着要监控测试错误率。

此外，根据为监视器选择的参数，需要将模式参数设置为“min”或“max”。如果度量标准

是错误/损失，就将其最小化。例如，以下代码块定义了一个 EarlyStopping()回调，用它在训练期间监视测试错误并检测错误是否不会再减少。

```
from keras.callbacks import EarlyStopping
es_callback = EarlyStopping(monitor='val_loss', mode='min')
```

如果错误率有很大的波动或噪声，那么当损失开始增加时停止训练可能不是一个好办法。出于这个原因，将 patience 参数设置为若干轮次，以便在停止训练过程之前用更长的时间来监控所需的指标。

```
es_callback = EarlyStopping(monitor='val_loss', \
                            mode='min', patience=20)
```

如果监控器指标在监控中没有改进，或者监控器指标已达到基线水平，还可以修改 EarlyStopping()回调，则停止训练过程。

```
es_callback = EarlyStopping(monitor='val_loss', \
                            mode='min', min_delta=1)
es_callback = EarlyStopping(monitor='val_loss', \
                            mode='min', baseline=0.2)
```

定义 EarlyStopping()回调后，可以将其作为回调参数提供给 model.fit()并训练模型。训练将根据 EarlyStopping()回调自动停止。

```
history=model.fit(X_train, y_train, validation_data=(X_test, y_test), \
                  epochs=epochs, callbacks=[es_callback])
```

训练 5.02 用Keras实现早停

在本训练中将学习如何在 Keras 深度学习模型上实现早停。使用一个模拟数据集，它包含树木的各种测量值，例如高度、树枝的数量和树干的周长。根据给定的测量值将记录分类为落叶树或针叶树。

首先，执行以下代码块加载包含 10000 条记录的模拟数据集，该数据集由代表两种树种的两个类组成，落叶树种的类值为 1，针叶树种的类值为 0。每条记录中有 10 个特征值。

建立一个模型，在给定树的测量值上预测树的种类。

(1) 使用 Pandas 库的 read_csv 函数加载数据集，并使用 train_test_split 函数将数据集拆分为 80：20 的比例。

```
# Load the data
import pandas as pd
X = pd.read_csv('../data/tree_class_feats.csv')
y = pd.read_csv('../data/tree_class_target.csv')

"""
Split the dataset into training set and test set with an 80-20 ratio
"""
from sklearn.model_selection import train_test_split
seed=1
X_train, X_test, \
y_train, y_test = train_test_split(X, y, test_size=0.2, \
                                   random_state=seed)
```

(2) 导入所有必要的依赖项。在不早停的情况下构建三层 Keras 序列模型。第一层有 16

个单元，第二层有 8 个单元，第三层有 4 个单元，都带有 ReLU 激活函数。添加带有 sigmoid 激活函数的输出层。

```
# Define your model
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
from tensorflow import random

np.random.seed(seed)
random.set_seed(seed)

model_1 = Sequential()
model_1.add(Dense(16, activation='relu', \
                  input_dim=X_train.shape[1]))
model_1.add(Dense(8, activation='relu'))
model_1.add(Dense(4, activation='relu'))
model_1.add(Dense(1, activation='sigmoid'))
```

(3) 设置损失函数为二元交叉熵，优化器为 sgd，编译模型。在 batch_size＝50 的情况下对模型训练 300 次，同时在每次迭代中存储训练误差和验证误差。

```
model_1.compile(optimizer='sgd', loss='binary_crossentropy')
# train the model
history = model_1.fit(X_train, y_train, \
                      validation_data=(X_test, y_test), \
                      epochs=300, batch_size=50, \
                      verbose=0, shuffle=False)
```

(4) 导入绘图相关的库。

```
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
```

(5) 绘制在拟合过程中创建并存储的训练误差和验证误差图。

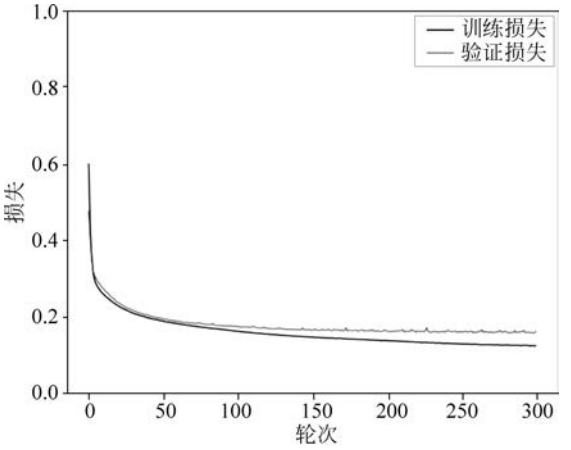
```
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylim(0,1)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], \
           loc='upper right')
```

预期输出如图 5.9 所示。

从图 5.9 中可以看出，将模型训练 300 个轮次的结果是训练误差和验证误差之间的差距越来越大，这表明开始发生过拟合。

(6) 重新定义模型，用相同的层数和每层内相同的单元数来创建模型，这样做的目的是确保模型以同样的方式被初始化。在训练过程中添加一个回调 es_callback＝EarlyStopping (monitor ＝'val_loss'，mode＝'min')。重复步骤(5)绘制训练误差和验证误差图。

```
#Define your model with early stopping on test error
from keras.callbacks import EarlyStopping
```



彩色图片

图 5.9　在不早停的情况下的训练误差和验证误差图

```
np.random.seed(seed)
random.set_seed(seed)

model_2 = Sequential()
model_2.add(Dense(16, activation='relu', \
                  input_dim=X_train.shape[1]))
model_2.add(Dense(8, activation='relu'))
model_2.add(Dense(4, activation='relu'))
model_2.add(Dense(1, activation='sigmoid'))
"""
Choose the loss function to be binary cross entropy and the optimizer
to be SGD for training the model
"""
model_2.compile(optimizer='sgd', loss='binary_crossentropy')
# define the early stopping callback
es_callback = EarlyStopping(monitor='val_loss', \
                            mode='min')
# train the model
history=model_2.fit(X_train, y_train, \
                    validation_data=(X_test, y_test), \
                    epochs=300, batch_size=50, \
                    callbacks=[es_callback], verbose=0, \
                    shuffle=False)
```

(7) 绘制误差图。

```
# plot training error and test error
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylim(0,1)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], \
           loc='upper right')
```

输出如图 5.10 所示。

通过在模型中添加 patience=0 早停回调，训练过程在大约 39 个轮次后自动停止。

彩色图片

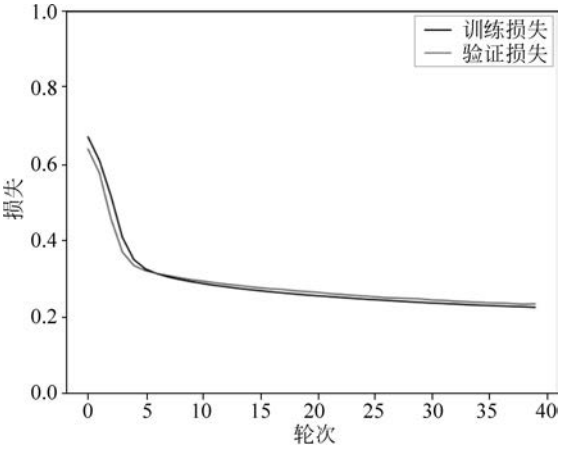


图 5.10　早停的训练误差和验证误差图(patience＝0)

(8) 重复步骤(3),同时将 patience＝10 添加到早停回调中。重复步骤(5)绘制训练误差和验证误差图。

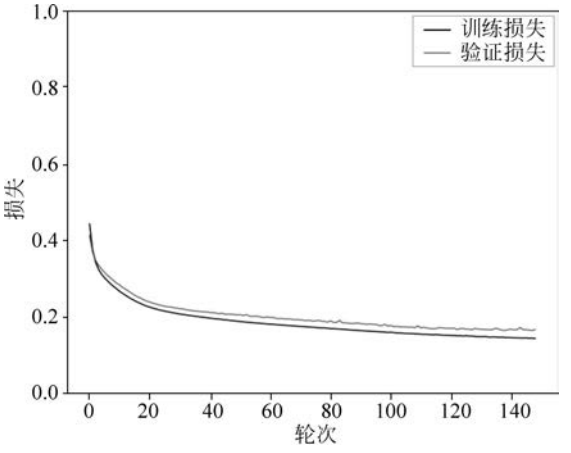
```
"""
Define your model with early stopping on test error with patience=10
"""
from keras.callbacks import EarlyStopping
np.random.seed(seed)
random.set_seed(seed)

model_3 = Sequential()
model_3.add(Dense(16, activation='relu', \
                  input_dim=X_train.shape[1]))
model_3.add(Dense(8, activation='relu'))
model_3.add(Dense(4, activation='relu'))
model_3.add(Dense(1, activation='sigmoid'))
"""
Choose the loss function to be binary cross entropy and the optimizer
to be SGD for training the model
"""
model_3.compile(optimizer='sgd', loss='binary_crossentropy')
# define the early stopping callback
es_callback = EarlyStopping(monitor='val_loss', \
                            mode='min', patience=10)
# train the model
history=model_3.fit(X_train, y_train, \
                    validation_data=(X_test, y_test), \
                    epochs=300, batch_size=50, \
                    callbacks=[es_callback], verbose=0, \
                    shuffle=False)
```

(9) 再次画出误差图。

```
# plot training error and test error
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```

```
plt.ylim(0,1)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], \
           loc='upper right')
```

预期输出如图 5.11 所示。



彩色图片

图 5.11　早停的训练误差和验证误差图(patience=10)

通过在模型中添加 patience=10 的早停回调,训练过程在大约 150 个轮次后自动停止。

在本训练中学习了如何应用早停来防止 Keras 模型过拟合训练数据。为此,使用 EarlyStopping 回调并用它来训练模型。使用此回调在验证损失出现增加时停止模型训练,并添加了一个 patience 参数,该参数规定了在多少个轮次之后早停模型。

说明:源代码网址为 https://packt.live/3iuM4eL,在线运行代码网址为 https://packt.live/38AbweB。

5.5.2　数据增强

数据增强是一种正则化技术,它试图通过廉价的方式在更多训练示例上训练模型来解决过拟合问题。在应用数据增强时,可用数据以不同的方式转换成为新的训练数据到模型中。这种类型的正则化已被证明是有效的,特别是对于某些特定的应用,如计算机视觉和语音处理中的目标检测/识别。

例如,在计算机视觉的应用中,可以通过将每个图像的镜像版本和旋转版本添加到数据集来简单地将训练数据集的大小增加一倍或三倍。这些转换生成的新训练样例显然不如原来的训练样例好,但它们可以改进模型过拟合。

执行数据增强的一个具有挑战性的方面是选择要对数据执行的正确变换。可根据数据集和应用程序的类型选择转换方式。

5.5.3　添加噪声

对数据添加噪声来正则化的基本思想与数据增强相同。在小数据集上训练深度神经网络增加了网络记忆单个数据示例,而不是获取输入和输出之间的关系。这会导致其在之后的新

数据上表现不佳，表明模型过拟合了训练数据。相反，在大型数据集上训练模型会增加模型捕获真实底层过程的机会，而不是记住单个数据点，减少过拟合。

扩充训练数据和减少过拟合的一种方法是向可用数据中注入噪声来生成新的数据示例。这种类型的正则化可以减少过拟合，与权重正则化技术的效果相当。

通过将单个示例的不同版本添加到训练数据中（每个版本都是通过在原始示例中添加少量噪声而创建的），可以确保模型不会拟合数据中的噪声。此外，通过扩充这些修改后的示例来增加训练数据集的大小，使模型可以更好地表示底层数据的生成过程，增加了模型学习真实过程的机会。

在深度学习的应用中，可以通过向隐藏层的权重、激活函数、网络的梯度甚至输出层添加噪声，以及通过向训练样本（输入层）添加噪声来提高模型性能。另一个需要通过尝试不同网络并观察结果来解决的挑战：确定在深度神经网络中的何处添加噪声。

在 Keras 中可以轻松地将噪声定义为一个层，并将其添加到模型中。例如，要将标准差为 0.1（均值等于 0）的高斯噪声添加到模型中，可以这样写。

```
from keras.layers import GaussianNoise
model.add(GaussianNoise(0.1))
```

以下代码将为模型的第一个隐藏层的输出/激活结果中添加高斯噪声。

```
model = Sequential()
model.add(Dense(4, input_dim=30, activation='relu'))
model.add(GaussianNoise(0.01))
model.add(Dense(4, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

在本节中学习了 3 种正则化方法：早停、数据增强和添加噪声。除了学习基本概念和过程之外，还学习了它们是如何减少过拟合的，并获得了一些关于如何使用它们的提示和建议。下一节将学习使用 scikit-learn 提供的函数来调整超参数。这样做的目的是将 Keras 模型合并到 scikit-learn 工作流中。

5.6 scikit-learn 超参数调优

超参数调优是提高深度学习模型性能的一项非常重要的技术。在第 4 章中了解了如何将 Keras 包装器与 scikit-learn 组合使用，它允许在 scikit-learn 工作流中使用 Keras 模型。因此，scikit-learn 中有很多不同的机器学习和数据分析工具及方法都已应用于 Keras 深度学习模型了，这其中就包括 scikit-learn 超参数优化器。

第 4 章学习了通过编写用户自定义函数，以循环每个超参数的可能值来微调超参数。本节将学习使用 scikit-learn 中可用的各种超参数优化方法以更简单的方式执行它。通过完成涉及真实数据集的实践来训练应用这些方法。

5.6.1 使用 scikit-learn 进行网格搜索

目前为止，构建深度神经网络会涉及多个超参数的确定。超参数包括隐藏层的数量、每个隐藏层包含的单元数量、每层的激活函数、网络的损失函数、优化器的类型及其参数、正则器的

类型及其参数、批量大小、训练轮数等。不同的超参数值会显著影响模型的性能。

因此，如何确定最优的超参数值已成为深度学习专家极重要和极具挑战性的部分之一。由于选择适用于每个数据集和每个问题的超参数没有绝对的规则，因此需要通过对每个特定问题的反复试验来确定超参数的值。使用不同超参数训练和评估模型并根据模型性能决定最终超参数的这个过程称为超参数微调或超参数调优。

为每个超参数设置一个范围或一组可能的值是非常重要的。可以创建一个网格，如图 5.12 所示。因此，超参数调优可以看作一个网格搜索问题；尝试网格中的每个单元格（超参数的每种可能组合）并找到一个可以为模型带来最佳性能的单元格。

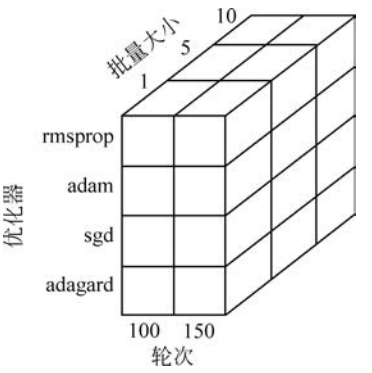


图 5.12　由优化器、批量大小、轮次的一些值创建的超参数网格

scikit-learn 提供了一个名为 GridSearchCV()的参数优化器来执行这种详尽的网格搜索。GridSearchCV()接收模型作为估计器参数，接收包含超参数所有可能值的字典作为 param_grid 参数。然后它遍历网格中的每个点，使用该点的超参数值对模型执行交叉验证，并返回最佳交叉验证值，以及导致该分数的超参数值。

在第 4 章中了解到在 scikit-learn 中使用 Keras 模型，需要定义一个返回 Keras 模型的函数。例如，下面的代码块定义了一个 Keras 模型，稍后在该模型上进行超参数调优。

```
from keras.models import Sequential
from keras.layers import Dense
def build_model():
    model = Sequential(optimizer)
    model.add(Dense(10, input_dim=X_train.shape[1], \
                    activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', \
                  optimizer= optimizer)
    return model
```

下一步是定义参数网格。假设调整 optimizer=['rmsprop','adam','sgd','adagrad']，epochs=[100,150]，batch_size=[1,5,10]。

```
optimizer = ['rmsprop', 'adam', 'sgd', 'adagrad']
epochs = [100, 150]
batch_size = [1, 5, 10]

param_grid = dict(optimizer=optimizer, epochs=epochs, \
                  batch_size= batch_size)
```

现在已经创建了超参数网格，可以创建包装器了，以便为 Keras 模型构建接口并将其用作估计器来执行网格搜索。

```
from keras.wrappers.scikit_learn import KerasRegressor
model = KerasRegressor(build_fn=build_model, \
```

```
                    verbose=0, shuffle=False)

from sklearn.model_selection import GridSearchCV
grid_search = GridSearchCV(estimator=model, \
                           param_grid=param_grid, cv=10)
results = grid_search.fit(X, y)
```

前面的代码遍历网格中的每个单元格，并使用每个单元格中的超参数值执行 10-fold 交叉验证(这里，它执行 10-fold 交叉验证 4×2×3=24 次)。然后，返回这 24 个单元格中每个单元格的交叉验证值，以及获得最佳分数的那个值。

说明：对许多可能的超参数组合执行 k-fold 交叉验证肯定需要很长时间。由于这个原因，可以通过将 n_jobs=-1 参数传递给 GridSearchCV()来并行化该过程，这样会用每个可用的处理器来执行网格搜索。此参数的默认值为 n_jobs=1，这意味着没有并行化。

创建超参数网格只是迭代超参数并寻找最佳选择的一种方法。另外一种方法是简单地随机选择超参数，将在下一节中学习。

5.6.2 使用 scikit-learn 进行随机搜索

因为详尽的网格搜索效率不高，所以它并不是调整深度学习模型超参数的最佳选择。在深度学习中有许多的超参数，详尽的网格搜索可能会需要很长的时间才能完成。执行超参数调优的另一种方法就是在网格上进行随机抽样，并对一些随机选择的单元格执行 k-fold 交叉验证。scikit-learn 提供了一个称为 RandomizedSearchCV()的优化器来执行随机搜索以实现超参数优化。

例如，更改 5.6.1 节的网格搜索为随机搜索，如下所示。

```
from keras.wrappers.scikit_learn import KerasRegressor
model = KerasRegressor(build_fn=build_model, verbose=0)

from sklearn.model_selection import RandomizedSearchCV
grid_search = RandomizedSearchCV(estimator=model, \
                                 param_distributions=param_grid, \
                                 cv=10, n_iter=12)
results = grid_search.fit(X, y)
```

注意，RandomizedSearchCV()需要提供额外的 n_iter 参数，该参数的作用是确定函数选择多少个随机单元格，它决定了需要执行多少次 k-fold 交叉验证。因此，如果选择较小的值，将考虑较少的超参数组合，并且该方法将花费较少的时间。另外，param_grid 参数在此处更改为 param_distributions。param_distributions 参数可以作为一个字典，其中参数名称作为键，参数列表或分布作为每个键的值。

RandomizedSearchCV()不如 GridSearchCV()好，因为它没有考虑超参数的所有可能值和组合。因此，对深度学习模型执行超参数调整的一种聪明方法是，要么对许多超参数执行 RandomizedSearchCV()，要么对较少且它们之间有较大差距的超参数使用 GridSearchCV()。

通过对超参数进行随机搜索，可确定哪些超参数对模型性能影响最大。它还可以帮助我们缩小重要超参数的范围。然后通过对较少数量的超参数和每个超参数的较小范围执行 GridSearchCV()来完成超参数调整。这被称为 coarse-to-fine 超参数调优。

这一节中练习使用 scikit-learn 优化器实现超参数调优。接下来将在下一个实践中尝试通过调整超参数来优化 Avila，数据集的模型。

实践 5.03 对Avila模式分类器进行超参数调优

在本实践中将构建一个与实践 5.01 类似的 Keras 模型，但这次，需要将向模型添加正则

化。然后，再使用 scikit-learn 优化器对模型执行超参数调整，包括对正则化器的超参数进行调整，具体步骤如下。

(1) 使用 X=pd.read_csv('../data/avila-tr_feats.csv')和 y = pd.read_csv('../data/avila-tr_target.csv')从 GitHub 的 Chapter05 文件夹的 data 子文件夹中加载数据集。

(2) 该函数返回一个具有三个隐藏层的 Keras 模型，第一个隐藏层的大小为 10，第二个隐藏层的大小为 6，第三个隐藏层的大小为 4，均采用 L2 权重正则化。使用这些值作为模型的超参数：activation='relu',loss='binary_crossentropy',optimizer='sgd'和 metrics = ['accuracy']。此外，还应将 L2 lambda 超参数作为参数传递给函数，以便稍后对其进行调优。

(3) 为 Keras 模型创建包装器并使用 cv=5 对其执行 GridSearchCV()。然后，在参数网格中添加以下值：lambda_parameter=[0.01,0.5,1],epochs=[50,100]和 batch_size=[20]。参数搜索完成后，输出最佳交叉验证分数的准确率和超参数。还可以输出所有其他交叉验证分数，以及产生该分数的超参数。

(4) 重复步骤(3)，这次在更小的范围内使用 GridSearchCV()，lambda_parameter=[0.001, 0.01,0.05,0.1],epochs=[400],batch_size=[10]。

(5) 重复步骤(4)，但从 Keras 模型中移除 L2 正则化器，在每个隐藏层添加带有 rate 参数的丢弃正则化。使用参数网格中的以下值对模型执行 GridSearchCV()并输出结果。rate=[0,0.2,0.4],epochs=[350,400]和 batch_size=[10]。

(6) 重复步骤(5)，使用 rate=[0.0,0.05,0.1]和 epochs=[400]。

执行这些步骤后，预期输出如下。

```
Best cross-validation score= 0.7862895488739013
Parameters for Best cross-validation score= {'batch_size': 20, 'epochs':
100, 'rate': 0.0}
Accuracy 0.786290 (std 0.013557) for params {'batch_size': 20, 'epochs':
100, 'rate': 0.0}
Accuracy 0.786098 (std 0.005184) for params {'batch_size': 20, 'epochs':
100, 'rate': 0.05}
Accuracy 0.772004 (std 0.013733) for params {'batch_size': 20, 'epochs':
100, 'rate': 0.1}
```

说明：本实践答案见附录 A 的实践 5.03 对 Avila 模式分类器进行超参数调优。

在本实践中，一起学习了如何使用正则化器在 Keras 模型上实现超参数调优，以使用真实数据集执行分类。还学习了如何使用 scikit-learn 优化器对模型超参数进行调优，包括正则化器的超参数。在本节通过创建超参数网格并迭代它们来实现超参数调优，使用 scikit-learn 工作流找到最优的超参数集。

5.7　总结

本章介绍了用于提高深度学习模型正确率的两个非常重要的技术：正则化和超参数调优。了解了正则化是如何通过几种不同的方法来解决模型过拟合问题的，包括 L1 和 L2 正则化及丢弃正则化，还有其他常用的一些正则化技术。另外，我们发现了超参数调优对机器学习模型的重要性，尤其是对深度学习模型进行超参数调整极具挑战性。

下一章将探讨评估模型性能时正确率指标及其他指标(如准确度、灵敏度、特异性和 AUC-ROC 评分)的局限性，包括如何使用它们来更好地评估模型。