**翻译** [**Deep Learning and the Game of Go**](https://www.manning.com/books/deep-learning-and-the-game-of-go)

**第5~8章**

**Chapter 5. Getting started with neural networks**

**第5章 神经网络入门**

本章涵盖

* 介绍人工神经网络的基础知识
* 使神经网络学会识别手写数字
* 通过堆叠层来创建神经网络
* 了解神经网络如何从数据中学习
* 从头开始实现简单的神经网络

本章介绍了人工神经网络（ANN）的核心概念，这是现代深度学习的核心算法。人工神经网络的历史令人惊讶，可以追溯到1940年代初。耗费了几十年时间，它的应用才在许多领域取得巨大成功，但基本思想仍然有效。

人工神经网络（ANN）的核心思想是从神经科学中汲取灵感，并对一类算法进行建模，该算法的工作方式与我们假设大脑部分功能的方式类似。特别是，我们将神经元的概念作为人工神经网络的基本单位。神经元形成的组称为层 layer，这些层以特定的方式彼此连接以跨越网络。在给定输入数据的情况下，神经元可以通过连接逐层传递信息，并且我们说，如果信号足够强，它们就会激活。」通过这种方式，数据将通过网络传播，直到到达最后一步，即输出层，我们才能从中进行预测。 然后，可以将这些预测与预期输出进行比较，以计算预测误差，网络将使用该误差来学习和改进未来的预测。

尽管灵感来源于大脑的架构比喻有时很有用，但我们不想在这里过分强调。 我们确实对大脑的视觉皮层了解很多，但是这种类比有时可能会误导甚至有害。 我们认为最好将ANN视为试图发现生物学习的指导原则，就像飞机是利用空气动力学原理，而不是复制鸟类一样。

为了使本章更加具体，我们从头开始提供了神经网络的基本实现。 您将使用此网络来解决光学字符识别（OCR）的问题； 即，如何使计算机预测在一张手写数字的图像上显示的是哪个数字。

我们的OCR数据集中的每个图像都是由布置在网格上的像素组成的，您必须分析像素之间的空间关系以找出其代表的数字。 像许多其他棋盘游戏一样，围棋是在网格上进行的，您必须考虑棋盘上的空间关系才能做出正确的选择。您可能希望OCR的机器学习技术也可以应用于Go等游戏。事实证明，他们确实能做到。[第6章](" \l "%E7%AC%AC6%E7%AB%A0%20%E4%B8%BA%E5%9B%B)至[第8章](#%E7%AC%AC8%E7%AB%A0%20%E5%9C%A8%E9%87%8)介绍了如何将这些方法应用于围棋游戏。

在本章中，我们相关的数学知识保持在较低水平。 如果您不熟悉线性代数，微积分和概率论的基础知识，或者需要简要而实用的提示，建议您先阅读附录A。 另外，神经网络学习过程中比较困难的部分可以在附录B中找到。如果您了解神经网络，但是从未实现过，建议您立即跳至第5.5节。 如果您也熟悉网络的实现，请直接跳至第6章，在第6章中将应用神经网络来预测第4章中生成的游戏的落子点。

**5.1. 一个简单的使用案例：对手写数字进行分类**

在详细介绍神经网络之前，让我们从一个具体的用例开始。 在本章中，您将构建一个应用程序，该应用程序可以很好地预测手写图像数据中的数字，并且准确率约为95％。值得注意的是，您只需将图像的像素值公开到神经网络来完成所有这些工作； 该算法将学习自行提取有关数字结构的相关信息。

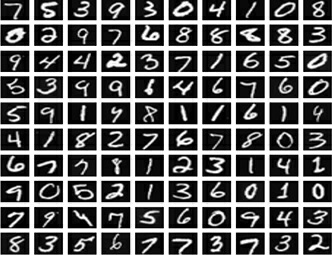
您将使用美国国家标准与技术研究院（MNIST）的手写数字数据集，是经过机器学习的从业人员经过深入研究的数据集以及深度学习的成果。

在本章中，您将使用NumPy库处理低级数学运算。NumPy是Python中机器学习和数学计算的行业标准，在本书的其余部分中都将使用它。 在尝试本章中的任何代码示例之前，应使用首选的软件包管理器安装NumPy。 如果使用pip，请在Shell中运行pip install numpy进行安装。 如果使用Conda，请运行conda install numpy。

**5.1.1. MNIST手写数字数据集**

MNIST数据集包含60,000张28×28像素的图像。此数据的一些示例如图5.1所示。 对人类来说，识别大多数示例图像是一件微不足道的任务，您可以轻松地将第一行中的示例读为7、5、3、9、3、0，依此类推。 但是在某些情况下，即使人类也很难理解图片所代表的意思。 例如，图5.1中第五行的第四张图片很容易是4或9。

图5.1 来自MNIST数据集的一些手写数字样本，这是光学字符识别领域中经过充分研究的实体



MNIST中的每个图像都带有一个标签，从0到9的数字代表图像上描述的真实值。

在查看数据之前，您需要首先加载它。 在本书的GitHub仓库中，您可以找到一个名为mnist.pkl.gz的文件，该文件位于文件夹<http://mng.bz/P8mn>中。

在此文件夹中，您还将找到在本章中编写的所有代码。 和以前一样，我们建议您按照本章的流程进行操作并构建代码库，但是您也可以按照GitHub存储库中的代码运行代码。

**5.1.2. MNIST数据预处理**

因为此数据集中的标签是从0到9的整数，所以您将使用一种称为“独热编码” one-hot encoding （注：又称为一位有效编码，主要是采用N位状态寄存器来对N个状态进行编码，每个状态都有它独立的寄存器位，并且在任意时候只有一位有效）的技术将数字1转换为全为0的、长度为10的向量，然后将1放在下标为1的位置。这种表示形式在机器学习中非常有用，并且广泛用于机器学习中。 在向量中为标签1保留第一个槽slot，可使神经网络等算法更轻松地区分标签。 例如，使用one-hot编码，数字2具有以下表示：[0, 0, 1, 0, 0, 0, 0, 0, 0, 0] .

Listing 5.1. One-hot encoding of MNIST labels

import six.moves.cPickle as pickle

import gzip

import numpy as np

def encode\_label(j): #1

e = np.zeros((10, 1))

e[j] = 1.0

return e

* 1 You one-hot encode indices to vectors of length 10.
* 1 您one-hot编码长度为10的向量的索引。

one-hot编码的好处是使得每个数字都有其自己的“槽”，并且您可以使用神经网络来输出一个输入图像的概率，这将在后面有用。

检查mnist.pkl.gz文件的内容，您可以访问三个数据池：训练，验证和测试数据。 回顾第一章，您使用训练数据来训练或拟合机器学习算法，并使用测试数据来评估算法的有效程度。验证数据可用于调整和验证算法的配置，但在本章中可以安全地忽略。

MNIST数据集中是的二维图像，高度和宽度均为28个像素。 将图像数据加载到大小为28×28=784的特征向量中；这里完全放弃了图像结构，只查看了表示为矢量的像素。此向量的每个值表示介于0和1之间的灰度值，其中0是白色，而1是黑色。

Listing 5.2. Reshaping MNIST data and loading training and test data

def shape\_data(data):

features = [np.reshape(x, (784, 1)) for x in data[0]] #1

labels = [encode\_label(y) for y in data[1]] #2

return zip(features, labels) #3

def load\_data():

with gzip.open('mnist.pkl.gz', 'rb') as f:

train\_data, validation\_data, test\_data = pickle.load(f) #4

return shape\_data(train\_data), shape\_data(test\_data) #5

* 1 将输入图像展开为784(28\*28)的特征向量
* 2 所有的label都是one-hot编码的
* 3 将特征和标签组合，创建特征和标签对
* 4 解压缩并加载MNIST数据会产生三个数据集
* 5 在此处丢弃验证数据，并重塑其他两个数据集

现在，您可以简单地表示MNIST数据集；特征和标签都被编码为向量。 您的任务是设计一种机制，该机制学习如何准确地将特征映射到对应的标签上。具体来说，您希望设计一种算法，该算法通过特征和标签的训练来学习，以便可以预测测试特征的标签。

神经网络可以很好地完成这项工作，正如您将在下一节中看到的那样，但是让我们首先讨论一个简单的方法，该方法将向您展示此应用程序必须解决的一般问题。对人类来说，识别数字是一项相对简单的任务，但是很难确切解释如何做到这一点，以及我们如何知道自己所知道的。 这种无所不能的知识现象被称为波兰尼悖论Polanyi’s paradox。 这使得特别难于向机器明确描述如何解决此问题。

至关重要的一个方面就是模式识别——每一个手写数字都具有某些特征，这些特征源于其原型数字版本。 例如，0大致是一个椭圆形，在许多国家中1只是一条垂直线。 有了这种启发式方法，您就可以通过将手写数字相互比较来对其进行简单分类：给定一个8的图像，该图像应该比任何其他数字更接近8的平均图像。 下面的average\_digit函数可以为您完成此操作。

Listing 5.3. Computing the average value for images representing the same digit

import numpy as np

from dlgo.nn.load\_mnist import load\_data

from dlgo.nn.layers import sigmoid\_double 195

def average\_digit(data, digit): #1

filtered\_data = [x[0] for x in data if np.argmax(x[1]) == digit]

filtered\_array = np.asarray(filtered\_data)

return np.average(filtered\_array, axis=0)

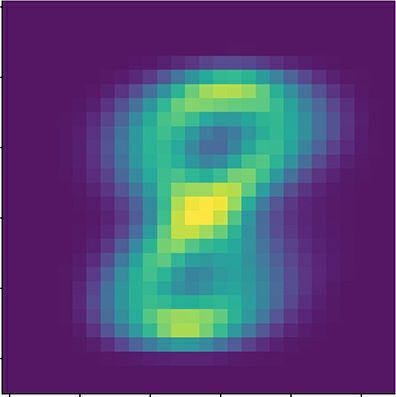
train, test = load\_data()

avg\_eight = average\_digit(train, 8) #2

* 1 计算代表给定数字的数据集中所有样本的平均值
* 2 Use the average 8 as parameters for a simple model to detect 8s.
* 2 使用平均8作为简单模型的参数来检测所有的8的图像

在你的训练集中，这个平均8是什么样子的？ 图5.2给出了答案。

图5.2. 这就是MNIST训练集中的平均手写8的样子。通常，对数百张图像进行平均将导致无法识别的斑点，但是这个平均值8仍然看起来很像8。



由于笔迹之间的差异可能会很大，因此，平均8有点模糊，但是形状仍然明显像8。也许您可以使用此表示法来识别数据集中的其他8。 您使用以下代码来计算和显示图5.2。

Listing 5.4. Computing and displaying the average 8 in your training set 计算并显示您的训练集中的平均8

from matplotlib import pyplot as plt

img = (np.reshape(avg\_eight, (28, 28)))

plt.imshow(img)

plt.show()

在MNIST的训练集中，平均8的平均值 avg\_eight 应当包含许多信息，说明图像上带有8的含义。 您将使用 avg\_eight 作为一个简单模型的参数来决定给定的输入向量x（代表一个数字）是否为8。在神经网络的背景下，当我们引用参数时，我们经常说权重weight，而 avg\_eight 将作为您的权重。

为方便起见，您将使用使用矩阵转置方法并定义W = np.transpose(avg\_eight)。 然后，您可以计算W和x的点积，对W和x的值进行逐点相乘，并相加得到784个结果值。 如果您的启发式算法/试探法正确，如果x是一个8，则各个像素在W大致相同的位置应具有较暗的色调。相反，如果x不是8，则应该少一点重叠。让我们通过几个例子来检验这个假设。

Listing 5.5. Computing how close a digit is to your weights by using the dot product 使用点积计算数字与体重的接近程度

x\_3 = train[2][0] #1

x\_18 = train[17][0] #2

W = np.transpose(avg\_eight)

np.dot(W, x\_3) #3

np.dot(W, x\_18) #4

* 1 索引2处的训练样本为4
* 2 索引17处的训练样本为8
* 3 评估的结果大概是约为20.1
* 4 这一项要大得多，结果约为54.2。

你计算两个MNIST样本与W的点积来计算权重，一个代表4，一个代表8。可以看到，后者8的54.2结果远高于4的20.1结果。现在，您如何确定结果值是否足够高以至于能将其预测为8？ 原则上，两个向量的点积可以得到任何实数。 解决该问题的方法是将点积的输出转换为[0，1]范围的数。例如，您可以尝试将临界值定义为0.5，并将高于该值的所有值都声明为8。

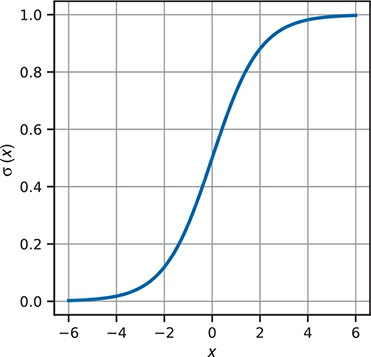
You compute the dot product of your weights W with two MNIST samples, one representing a 4, and another one representing an 8. You can see that the latter result of 54.2 for the 8 is much higher than the 20.1 result for the 4.

看来您要完成198项操作。 现在，您如何确定结果值何时足够高以将其预测为8？ 原则上，两个向量的点积可以吐出任何实数。 解决该问题的方法是将点积的输出转换为[0，1]范围。 这样做时，例如，您可以尝试将临界值定义为0.5，并将高于该值的所有值都声明为8。

一种方法是使用Sigmoid函数。 Sigmoid函数函数通常用s表示，希腊字母sigma σ。 对于实数x，Sigmoid函数定义为：​

图5.3 显示了如何获得直觉。

图5.3 Sigmoid函数图。Sigmoid将实际值映射到[0,1]的范围。在0左右，曲线的斜率相当陡峭，无论大小，曲线都趋于平坦。



接下来，让我们先在Python中对Sigmoid函数进行编码，然后再将其应用于点积的输出。

Listing 5.6. Simple implementation of sigmoid function for double values and vectors 双值和向量的Sigmoid函数的简单实现

def sigmoid\_double(x):

return 1.0 / (1.0 + np.exp(-x))

def sigmoid(z): return

np.vectorize(sigmoid\_double)(z)

请注意，您提供了对double值进行运算的sigmoid\_double函数，以及为本章中将广泛使用的计算向量的sigmoid函数。 在将sigmoid应用于之前的计算之前，请注意sigmoid(2)已经接近1，因此对于您先前计算的两个样本，sigmoid(54.2)和sigmoid(20.1)实际上是无法区分的。 您可以通过将点积的输出向0靠近来解决这个问题，这称为应用一个偏移量，通常用b来表示。在这个样本中，你可以设置偏移量为b=-45。使用权重和偏移量，你现在可以用你的模型进行计算如下：

Listing 5.7. Computing predictions from weights and bias with dot product and sigmoid 通过点积和S形计算权重和偏差的预测

def predict(x, W, b): #1

return sigmoid\_double(np.dot(W, x) + b)

b = -45 #2

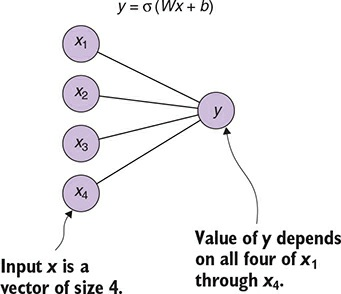
print(predict(x\_3, W, b)) #3

print(predict(x\_18, W, b)) #4

* 1 通过将sigmoid应用于np.dot(W, x) + b的输出来定义简单的预测。
* 2 根据到目前为止计算的示例，将偏差项设置为–45。
* 3 对于具有4的示例的预测接近于0。
* 4 这里的8预测为0.96，启发式似乎有些道理。

在两个示例x\_3和x\_18上，您得到令人满意的结果。后者的预测接近于1，而对于前者的预测则几乎为0。将输入向量x映射到s(Wx+b)的过程W（与x大小相同的向量）称为逻辑回归logistic regression。 图5.4 示意性地描述了长度为4的向量的算法。

Figure 5.4. An example of logistic regression, mapping an input vector x of length 4 to an output value y between 0 and 1. The schematic indicates how the output y depends on all four values in the input vector x. 逻辑回归的一个示例，将长度为4的输入向量x映射到介于0和1之间的输出值y。该示意图指示输出y如何依赖于输入向量x中的所有四个值。



为了更好地了解这个过程是如何工作的，让我们计算所有训练和测试样本的预测结果。如前所述，您定义了一个决策阈值来决定是否可以预测为8。作为这里的评估指标，您选择精确性；您计算所有预测中正确预测的比率。

Listing 5.8. Evaluating predictions of your model with a decision threshold

def evaluate(data, digit, threshold, W, b):

# 1

total\_samples = 1.0 \* len(data)

correct\_predictions = 0

for x in data:

if predict(x[0], W, b) > threshold and np.argmax(x[1]) == digit: # 2

correct\_predictions += 1

if predict(x[0], W, b) <= threshold and np.argmax(x[1]) != digit: # 3

correct\_predictions += 1

return correct\_predictions / total\_samples

* 1 对于所有的预测指标，你选择准确性，即所有预测中正确的比率
* 2 将一个8的实例预测为8是正确的预测
* 3 如果预测低于你的阈值且样本不是8，你的预测依旧正确

让我们使用这个评估函数来评估三个数据集的预测质量：训练集、测试集和测试集中所有8的集合。你这样做的阈值、权重和偏移量都与以前一样。

evaluate(data=train, digit=8, threshold=0.5, W=W, b=b) # 1

evaluate(data=test, digit=8, threshold=0.5, W=W, b=b) # 2

eight\_test = [x for x in test if np.argmax(x[1]) == 8]

evaluate(data=eight\_test, digit=8, threshold=0.5, W=W, b=b) # 3

* 1 简单模型在训练数据上的准确性是78%(0.7814)
* 2 测试数据的准确性略低，为77%(0.7749)
* 3 只对测试集中的8进行评估，结果精度只有67%(0.6663)

你可以看到在训练集上的准确率最高约为78%。这不应该令人惊讶，因为你在训练集上校准了你的模型。值得注意的是，它并不能在训练集上评估，因为它无法告诉算法泛化的程度。测试数据的表现接近于训练时的表现，大约77%。在测试集中所有是8的集合中，您只达到了66%，因此使用你的简单模型，您只在两个不都是8的例子中比较精确。这个结果作为开始阶段是可以接受的，但远远不是你能做的最好的。那到底出了什么问题，你能做得更好吗？

* 您的模型能够区分一个特定数字（这里是一个8）与其他的数字。因为在训练和测试中，每个数字的图像数量都是平均的，只有10%左右是8。因此，一个一直预测0的模型将产生大约90%的准确性。在分析像这样的分类问题时，经常会遇到这样的问题。有鉴于此，你在训练数据上的77%精确性看起来不再那么强大了。您需要定义一个模型，它可以准确地预测所有数字。
* 您的模型的参数相当小。对于成千上万种的手写图像集合，你所拥有的只是一组与其中一幅图像大小相同的权重集合。相信你能通过使用这样一个小的模型去捕捉到这些手写图像上的笔迹变化是不现实的。您必须找到一类算法，这些算法有效地使用更多的参数来捕获数据的可变性。
* 对于给定的预测，你只是简单地选择一个阈值去宣称该数字是8还是不是8。您没有使用实际预测值来评估模型的质量。例如，一个正确的预测在0.95，那当然表明这比精确度0.51的更加精确。你必须把预测与实际结果有多接近的概念进行形式化。
* 你在直觉的指导下手工制作了模型的参数。即使作为开始阶段这样做还是可以的，但机器学习的前景是您不必将自己的观点强加于数据上，而是让算法从数据中学习。 每当您的模型做出正确的预测时，您都需要加强这种行为，而当输出错误时，则需要相应地调整模型。 换句话说，您需要设计一种机制，根据对训练数据的预测程度来更新模型参数。

尽管对这个小用例和您所建立的幼稚模型的讨论似乎不多，但您已经看到许多组成神经网络的部分。 在下一节中，您将使用围绕此用例构建的直觉来解决神经网络的第一步，即解决这四个问题。

**5.2 神经网络的基础**

如何改善OCR模型？正如我们在引言中所暗示的那样，神经网络可以在这种任务上做得更好，比我们手工制作的模型要好得多。 但是，手工制作的模型确实说明了用于构建神经网络的关键概念。 本节以神经网络的术语去描述上一部分的模型。

**5.2.1 逻辑回归作为简单人工神经网络**

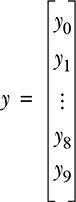
在5.1节中，您看到了用于二进制分类的逻辑回归。重述一下，您取了一个表示数据样本的特征向量x，将其输入到算法中，首先将其乘以权重矩阵W，然后添加一个偏移量b。为了输出的预测y在0到1之间，您将应用了Sigmoid函数：y=s(Wx+b)

您应该注意到这里的一些事情。首先，特征向量x可以解释为神经元的集合，有时称为单元，通过W和b连接到y，您已经在图5.4中看到了这一点。接下来，注意sigmoid函数它被看作是一个激活函数，因为它接受Wx+b的结果并将其映射到范围[0，1]。如果你把一个接近1的值解释为神经元y被激活，而反过来，接近0是没有被激活，这个设置可以看作是人工神经网络的一个小例子。

**5.2.2.多个输出维度的网络**

在5.1节中的用例中，您将识别手写数字的问题简化为二进制分类问题；即将8与所有其他数字区分开来。但你感兴趣的是要预测10个类，每个数字一个类。至少在形式上，你可以很容易地通过改变你所说的y、W和b去实现这一点；也就是你改变了模型的输出、权重和偏差。

首先，你使y成为一个长度为10的向量；y中的每个值，表示10的数字中每一数字的可能性：

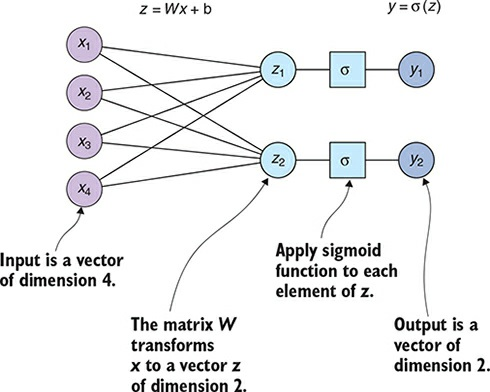


其次，让我们相应地调整权重和偏差。回想一下，目前W是长度为784的向量。取而代之的是，您要让W变成（10，784）的矩阵。这样，可以让W矩阵输入向量x相乘，即Wx，其结果将是长度为10的向量。紧接着，如果使偏移量变成长度为10的向量，那么久可以将其添加到Wx中。最后，请注意，你可以通过计算z向量的sigmoid值将应用于每个组件：



下图描述了四个输入和两个输出神经元的稍微改变的设置。

图5.5 在这个简单的网络中，四个输入神经元首先与一个2\*4的矩阵相乘，加上一个二维偏离量，然后应用sigmoid函数,输出一个2维向量



现在，你得到了什么？现在可以将输入向量x映射到输出向量y，而以前y只是一个值。这样做的好处是可以实现向量的多次转换，从而构建我们所说的前馈网络。

**5.3 前馈网络**

让我们快速回顾一下你在上一节中所做的事情。在一个较高的层次上，您执行了以下步骤：

1. 您从输入神经元x的向量开始，并应用了一个简单的转变，即z=Wx+b。在线性代数的概念中，这些变换被称为仿射线性。在这里，您使用了z作为中介变量
2. 你应用了一个激活函数sigmoid：y=s(z)，得到输出神经元y。应用程序的结果告诉你被激活了多少。

前馈网络的核心是你可以应用这个概念迭代处理，从而可以多次应用于这两个指定的步骤去简单构建块。这些块就构成了我们所说的一层。用这个概念，你可以说堆叠多层会形成多层神经网络。让我们通过再介绍一个层来修改我们的上一个示例。你现在必须运行以下步骤：

1. 从输入x开始计算z=W¹x¹+ b¹
2. 对于中间结果z，再计算y=W²z¹+b²得到输出y。

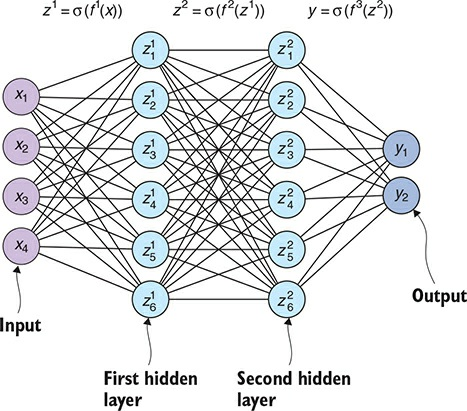
请注意，这里用上标来表示你在哪一层，下标表示带有一个向量或矩阵的位置。使用两层而不是一层的方式在下图中可以看到。

图5.6 具有两层的人工神经网络。 输入神经元x连接到单元z的中间集合，单元本身连接到输出神经元y。



在这一点上，应该很清楚的是你不能绑定到任何要堆叠特定数量的层，你还需要使用更多。此外，你不一定一直使用sigmoid作为激活函数，您有大量的激活函数可供选择，我们将在下一章中介绍其中的一些函数。对于一个或多个数据点顺序应用网络中所有层的功能，通常被称为前向传递。它被称为前向的原因，是数据总是向前流动，从输入到输出，然后就不往回走。 有了这个符号，描述一个具有三层的常规前馈网络如下图5.7。

图5.7 具有三层的神经网络。 定义神经网络时，层数和每层神经元数都不受限制。



为了回顾你到目前为止所学到的东西，让我们在下面一个简洁的列表中把提到的所有概念放出来：

* 顺序神经网络是一种映射特征的机制，或者输入神经元，x去预测，或输出神经元，y。通过顺序地逐层堆叠简单函数来实现这一点。
* 一个层是将给定的输入映射到输出的东西。计算一批数据的一层输出，我们称为前向传递。同样，计算顺序网络的前向传递就是顺序地计算从输入层开始的每层的前向传递。
* Sigmoid函数是一个激活函数，它接受实值神经元的向量作为参数并激活它们，以至于它们可以映射到范围[0，1]。您可以将接近1的值解释为激活。
* 给定一个权重矩阵W和一个偏置项b，应用仿射非线性变换Wx+b形成一层。这种层通常被称为dense(密集)层或完全连接层,往下走，我们坚持称它们为dense(连接)层。
* 根据实现的不同，dense(连接)层可能会或可能不会与激活函数一起；您可能会看到该层有s（Wx+b），而不仅仅是一个仿射线性变换信息。另一方面，一层只考虑激活函数是很常见的，您将在实现中也这样做。
* 一个前馈神经网络是由具有激活的dense(连接)层组成的顺序网络,这种架构通常也被称为多层感知器，简称MLP。
* 所有的既不是输入也不是输出的神经元叫做隐藏单元。相应的，输入和输出神经元有时被称为可见单元。直觉是隐藏单元是内部的网络，而可见的单位是可以直接观察到。这在某种程度上是一个延伸，因为正常情况下你可以访问系统的任何部分。因此，输入和输出两层之间称为隐藏层：每个至少有两个层的顺序网络至少有一个隐藏层。
* 如果没有另外说明，x将代表网络的输入，y表示输出，有时用下标来表示你正在使用哪个样本。将许多层堆叠起来，建立一个包含许多隐藏层的大网络被称为深层神经网络，因此这个名字叫做深度学习。

**非顺序神经网络**

到现在，你只学习到顺序神经网络，其中的层形成一个序列。在顺序网络中，从输入开始，每个跟随的隐藏层都正好有一个前驱和一个后继，直到输出层结束。这足以使你将深度学习应用到围棋游戏中。

总的来说，理论上神经网络也允许任意的非顺序结构。例如，在某些应用程序中，连接或添加两个层的输出（合并两个或多个的先前层）是有意义的。在这种情况下，合并多个输入然后产生一个输出。

在其他应用程序中，将一个输入分成几个输出是有用的。一般来说，一个层可以有多个输入和输出。我们在第11章和第12章分别介绍了多输入和多输出网络。

具有n层的多层感知器被描述为由权重集W=W¹，...，Wⁿ和偏差集b=b¹，...，bⁿ,以及为每层选择的激活函数集组成。但是一个重要的从数据中学习并更新参数的因素仍然缺少：损失函数和如何优化它们。

**5.4 我们的预测有多好？**

损失函数和优化器 第5.3节定义了如何建立前馈神经网络并通过它传递输入数据，但您仍然不知道如何评估预测的质量。要做到这一点，你需要一个措施定义预测和实际的结果有多接近。

**5.4.1 什么是损失函数？**

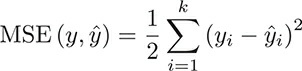
为了用你的预测来量化你错过了多少目标，我们引入了损失函数的概念，通常称为目标函数。假设你有一个带有权重W、偏移量b和sigmoid激活函数的前馈网络，对于给定的一组输入特征X1，...，Xk和相应的标签Y1，...，Yk，使用网络你可以计算预测Y1，...，YK。在这种情况下，损失函数的定义如下：

image-20210105235245914

这里，Loss()是一个可微函数differentiable function。损失函数是一个平滑函数，可以将非负值表示为多个（预测、标签）对。一堆特征和标签的损失是样本损失的总和。你的训练目标是通过找到好的策略去调整参数，从而能够尽量减少损失。

**5.4.2 均方误差**

一个广泛使用的损失函数是均方误差（MSE）。虽然使用MSE对我们的用例并不理想，但它是最直观的损失函数之一。你要测量预测值和实际结果有多接近，可以与所有观察到的例子计算均方和平均。表示标签，y=y1，...，yk表示预测，均方误差定义如下：



在你已经看到了函数被应用之后，我们将介绍各种损失函数的优点和缺点。现在，让我们在Python中实现均方误差。

Listing 5.10. Mean squared error loss function and its derivative

import random

import numpy as np

class MSE: # 1

def \_\_init\_\_(self):

pass

@staticmethod

def loss\_function(predictions, labels):

diff = predictions - labels

return 0.5 \* sum(diff \* diff)[0] # 2

@staticmethod

def loss\_derivative(predictions, labels):

return predictions - labels #3

* 1 使用平均平方误差作为损失函数
* 2 将MSE定义为0.5乘以predictions和label之间的平方差。
* 3 损失导数只是简单的predictions减去label

请注意，您不仅实现了损失函数本身，而且还实现了导数：loss\_derivate。这个导数是一个向量，是通过将预测向量减去标签向量。接下来，您将看到像MSE导数这样的在训练神经网络中起着至关重要的作用。

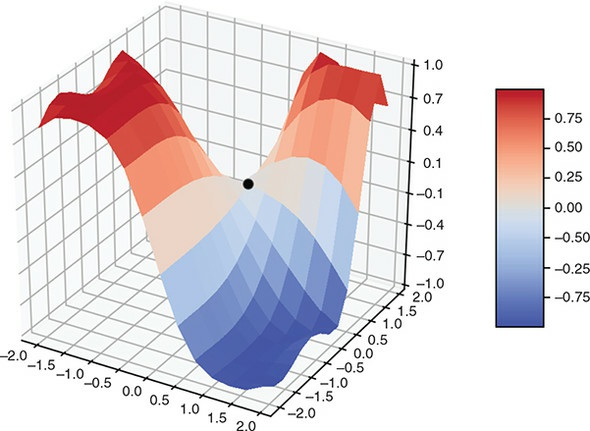
**5.4.3 损失函数找到最小值**

一组预测和标签的损失函数为您提供了关于模型参数调整情况的信息。损失越小，你的预测就越好，反之亦然。这个损失函数本身是您网络参数的函数。在您的均方误差实现中，您不直接提供权重，但是通过预测隐含地给出，因为你可以使用权重去计算他们。

理论上，你从微积分中知道，要使损失最小化，你需要它的导数置为0。此时，我们将参数集称为一种解决方案。计算一个函数的导数，并在一个特定的点上评估它，被称为计算梯度。现在您已经完成了在均方误差实现中计算导数的第一步，但是有更重要的事。您的目标是显式地计算网络中所有权重和偏置项的梯度。

如果你需要复习微积分的基础知识，一定要看看附录A。图5.8显示了三维空间中的一个曲面。这个曲面可以解释为一个二维输入的损失函数。平面的两个轴代表你的权重，竖轴表示为损失值。

图5.8 二维输入（损耗面）的损耗函数的示例。 该表面在右下角的暗区附近具有最小值，可以通过求解损失函数的导数来计算。



**5.4.4 梯度下降寻找最小值**

直观地说，当您计算给定点的函数梯度时，该梯度会指向最陡峭的上升方向。从一个损失函数、损失和一组参数开始，为了找到这个函数的最小值，梯度下降算法是这样的：

1. 计算当前参数W的损失梯度Δ（重复计算每个权重下的损失函数的导数）。
2. 通过减去梯度来更新W。我们把这个步骤称为沿着梯度。因为梯度指向最陡峭的上升方向，减去它会引导你朝这个方向前进最多的下降。
3. 重复这些过程，直到梯度为0

因为损失函数是非负的，而且它有一个最小值。它可能有很多，甚至无限多的最小值。例如，如果你考虑一个平面，它上的每一点都是最小的。

**局部和全局的最小值**

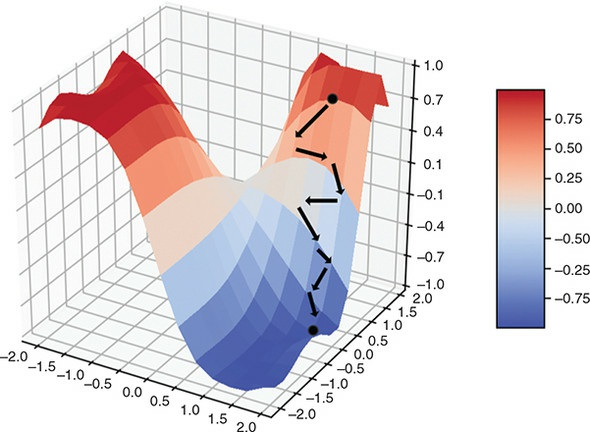
梯度下降达到零梯度的点被定义为最小值。对于许多变量的可微函数，最小值的精确数学会使用有关函数曲率的信息。

随着梯度下降，你最终会找到一个最小值；你可以跟随函数的梯度，直到你找到一个零点梯度。有一个问题：你不知道这个最小值是局部的还是全局的最小值。你可能会被困在一个地方上，这个地方是函数所能接受的最小点，但其他的点可能具有较小的绝对值。图5.8中的标记点是局部最小值，但显然在这个表面存在更小的值。

我们为解决这个问题所做的一切可能会使你感到震惊：我们将忽略它。在实践中，梯度下降往往导致令人满意的结果，因此在神经网络损失函数的背景下，我们往往会忽略一个最小值是否是局部的还是全球的。我们通常在收敛之前甚至不会运行算法，而是在预定义的数数之后停止。

图5.9显示了从上图开始的梯度下降对损失面的作用，以及右上角标记点所指示的参数的选择。

图5.9 迭代地遵循损失函数的梯度将最终使您降至最低。



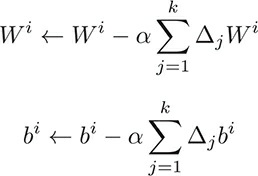
在均方误差实现中，您已经看到均方误差损失函数的导数很容易，就是标签和预测之间的区别。但要评估这样一个导数，你必须首先要计算预测值。要查看所有参数的梯度，您必须评估和聚合训练集中每个样本的导数。假如你通常需要处理数百万的数据样本，这样做实际上是不可行的。取而代之的，你将使用一种叫做随机梯度下降的技术来近似计算梯度。

**5.4.5.损失函数的随机梯度下降**

要计算梯度，并将梯度下降应用于神经网络，您必须要评估损失函数和在训练集上每个点上网络参数的导数，在大多数情况下太费时间了。取而代之的，我们将使用一种称为随机梯度下降（SGD）的技术。要使用SGD，首先要从你的训练集中选择一些样本，您称之为小训练集。每份选择的小训练集都有一个固定的长度，我们称之为小训练集个数。对于一个分类问题，像你正在处理的手写数字问题，这是一个的做法，因此它的训练集与标签数量具有相同的数量级，从而确保每个标签都能在一个小批量中表示。

对于一个给定的具有l层和输入数据为x₁,...xⁿ的小训练集的前馈神经网络，您可以计算您的神经网络的前向传递，并计算该小训练集的损失。对于本训练集中的每个样本xj，则可以选择都去计算网络中的每个参数来评估损失函数的梯度，在第i层的权重和偏移量梯度我们分别用∆jWi,∆jbi。

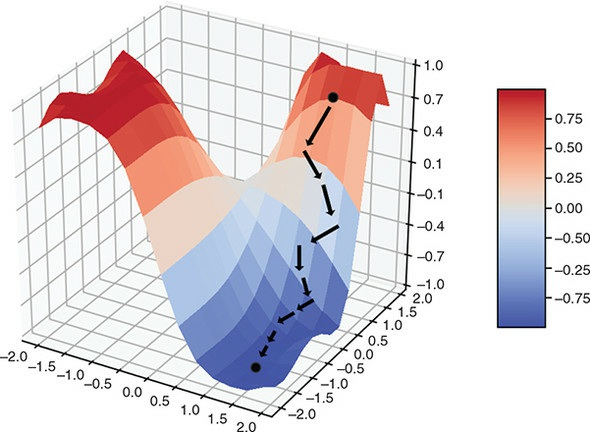
对于训练集中的每一层和每一个样本，您计算各自的梯度，并对使用参数以下更新规则：



您通过减去该批处理收到的累积错误来更新参数。这里a>0表示学习率，这是在训练网络之前指定的一个实体。

如果你要一次总结所有的训练样本，你会得到更多关于梯度的精确信息。在梯度精度方面，使用小训练集是一个折衷方案，但计算效率要高得多。我们称这种方法为随机梯度下降，因为小训练集样本是随机选择的。虽然在梯度下降中，你有一个接近局部最小值的理论保证，而在随机梯度下降中。事实并非如此。图5.10显示了SGD（随机梯度下降）的典型行为。你的一些近似随机梯度可能并不指向下降的方向，但是如果有足够的迭代，你会通常接近接近最小值。

图5.10 随机梯度的精确度较低，因此在损失面上跟踪它们时，您可能会先走一些弯路，然后再接近局部最小值。



**优化器**

计算（随机）梯度是由微积分的基本原理来定义的,而使用梯度更新参数的方式却不是。像SGD（随机梯度下降）的更新规则这样的技术称为优化器。

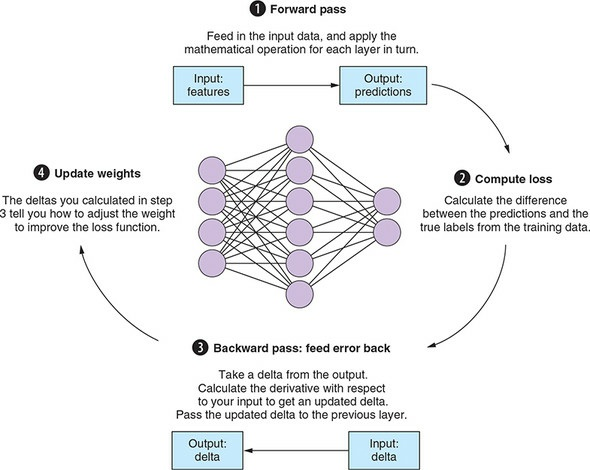
现在还有许多其他的优化器，以及更复杂的随机梯度下降版本。我们在第7章中涵盖了SGD（随机梯度下降）的一些扩展。大多数扩展都是围绕着适应随着时间的推移的学习速率，或有更多的粒度去更新的个人权重 。

**5.4.6.通过你的网络反向传播梯度**

我们已经讨论了如何使用随机梯度下降来更新神经网络的参数，但我们没有解释如何到达梯度。计算这些梯度的算法是称为反向传播算法，我们已并在附录B中详细介绍。本节给出反向传播背后的原理和实现前馈网络的必要构建块。

回想一下，在前馈网络中，您通过计算一个又一个简单的构建块来实现数据的正向传递，由最后一层进行输出网络的预测，然后你可以根据对应的标签去计算损失，而损失函数本身就是较简单函数的组成，若要计算损失函数的导数，可以使用微积分的基本属性：链式法则。这一规律粗略地说，组成函数的导数就是这些函数的导数的组成。因此，当您将输入数据向前传递一层又一层时，你可以一层又一层地往回传递导数。因为是通过网络往回传播导数，所以其名叫反向传播。在图5.11中，您可以看到一个拥有两个dense层和sigmoid激活函数的前馈网络反向传播的行为：

图5.11 具有sigmoid激活函数和MSE损失功能的两层前馈神经网络中的向前和向后传递



为了指导你完成上图的步骤，让我们一步一步地做：

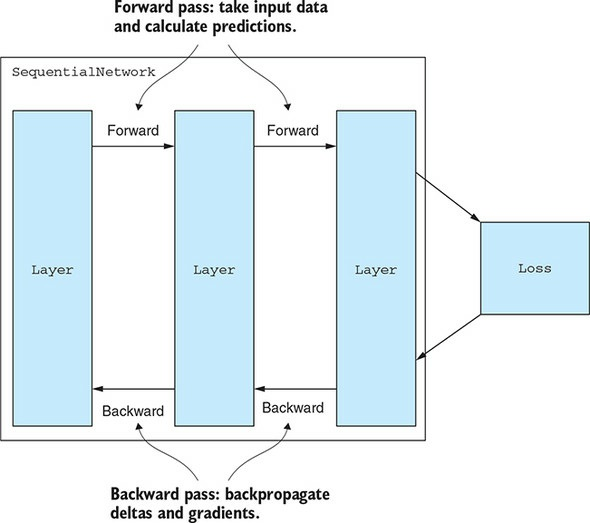
1. **向前传递训练数据。**在此步骤中，您获得了一个输入数据样本x并将其沿着网络传递，从而获得预测，详情如下：
   1. 你计算仿线性部分：Wx+b。
   2. 将Sigmoid函数σ（X）应用于结果。
   3. 重复这两个步骤，直到到达输出层。我们在这个例子中选择了两个层，但是层数并不重要。
2. **损失函数评估。**在此步骤中，您将样本x的标签取出来，并通过与你的预测值进行比较得出损失值。在本例中，您选择了均方误差作为损失函数。
3. **反向传递误差。**在这一步中，您将获取损失值并通过网络往回传递。你这样做是按层计算导数，因为符合链规则，所以是可行的。在一个方向上沿着网络向前传递输入数据，向后传递反馈的错误。
   1. 您可以按正向传递的相反顺序的传播误差项（即反向传递错误项），用Δ表示。
   2. 首先，您计算损失函数的导数，它将是您的初始Δ。然后不断往回传递
   3. 计算Sigmoid导数，这只是简单的σ(1-σ)。若要将Δ传递到下一层，则可以进行计算乘法：σ(1-σ)·Δ。
   4. 你的仿线性变换Wx+b相对简单的。为了传递Δ，你要计算​
   5. 这两个步骤会重复，直到你到达网络的第一层。
4. 用梯度信息更新权重。在最后一步中，您使用一路上计算的Δ来更新您的网络参数（权重和偏移量）。
   1. sigmoid函数没有任何参数，所以你不需要做什么。
   2. 每层中的偏移项b的更新量Δb只是简单Δ
   3. 一层中权重W的更新量ΔW应为​（在与Δ相乘前x需要转置）。
   4. 请注意，我们首先说x是一个单一的样本。我们讨论过的每件事都离不开小样本。如果x表示一小批样本（x是输入向量中任何一列的矩阵），那么向前和向后传递的计算看起来完全相同。

既然你拥有构建和运行一个前馈网络的所有数学知识，让我们应用您在理论层面上学到的东西，从零开始构建一个神经网络。

**5.5 使用Python一步接一步地训练神经网络**

前一节涵盖了许多理论基础，但在概念层面上，你只学到了几个基本概念。对于我们的实现，您只需关注三件事：一个Layer类，一个SequentialNetwork类(它是通过一个接一个地添加Layer对象来构建的)，以及一个需要反向传播的Loss类。在您将加载和检测手写数字数据，并将您的网络实现应用于它之后，这三个类接下来会被使用。图5.12显示了这些Python类是如何组合在一起实现前一节描述的前向和后向传递。

图5.12 用Python实现的前馈网络类图。一个SequentialNetwork包含几个Layer实例。 每个层实现一个数学函数及其派生函数。 正向和反向方法分别实现正向和反向传递。 损失实例计算您的损失函数，即您的预测和训练数据之间的误差。



**5.5.1.用Python实现神经网络层**

要从一个一般的Layer类开始，请注意，正如我们前面讨论过的那样，Layer不仅有一个处理输入数据（前向传递）的方法，而且还有一个反向传递错误的方法。为了在反向传递过程上不重新计算激活值，保持两次进出数据的状态是必要的。既然这么说了，Layer类的初始化应该是直截了当的。现在，您将开始创建一个Layer模块；在本章后面，您将使用此模块中的组件来构建神经网络。

Listing 5.11. Base layer implementation

import numpy as np

class Layer: #1

def \_\_init\_\_(self):

self.params = []

self.previous = None #2

self.next = None #3

self.input\_data = None #4

self.output\_data = None

self.input\_delta = None #5

self.output\_delta = None

1 Layers are stacked to build a sequential neural network.

2 A layer knows its predecessor (previous)...

3 ...and its successor (next).

4 Each layer can persist data flowing into and out of it in

the forward pass.

5 Analogously, a layer holds input and output data for the

backward pass.

* 1 层被堆叠起来以建立一个顺序的神经网络。
* 2 每个层都定义前驱层
* 3 每个层都定义后继层
* 4 每层可以像之前的层中持久化流入和流出的数据
* 5 类似的，每层都要保持像后面的层输入输出数据

每层有一个参数列表，并存储其当前的输入和输出数据，以及向后传递的相应的输入和输出增量。

另外，因为你关心的是在顺序神经网络中，每一层都有一个前驱和后继。因此继续定义，添加以下内容。

Listing 5.12. Connecting layers through successors and predecessors

def connect(self, layer): #1

self.previous = layer

layer.next = self

* 1 在顺序网络中，这种方法将每一层与他的邻居直接相连

接下来，你将让抽象Layer类中的前向和后向传递方法占位，而让子类去实现这些方法。

Listing 5.13. 在顺序神经网络的一层中Forward and backward passes

def forward(self): #1

raise NotImplementedError

def get\_forward\_input(self): #2

if self.previous is not None:

return self.previous.output\_data

else:

return self.input\_data

def backward(self): #3

raise NotImplementedError

def get\_backward\_input(self): #4

if self.next is not None:

return self.next.output\_delta

else:

return self.input\_delta

def clear\_deltas(self): #5

pass

def update\_params(self, learning\_rate): #6

pass

def describe(self): #7

raise NotImplementedError

* 1 每个层的实现都必须提供一个功能来向前输入数据
* 2 input\_data是为第一层保留的；所有其他层都从前一层的输出中获得他们的输入。

1. 3 层必须实现错误项的反向传播——一种通过网络反馈传递错误的方法
2. 4 输入增量保留在最后一层；所有其他层都从它们的后继那里获得错误项
3. 5 每个小样本集计算和累积增量，然后需要重置这些增量
4. 6 根据当前增量更新图层参数，使用指定的学习率
5. 7 层实现可以打印其属性

作为辅助函数，您提供了get\_forward\_input和get\_backward\_input，它们只是为各自的传递检索输入，但要特别注意输入和输出神经元。最重要的是，你实现了一个clear\_deltas方法，在将增量积累到小批次之后，定期重置增量，还有update\_params，该方法负责在网络使用这层之后更新该层的参数。

请注意，作为功能的最后一部分，您为一个层添加了一个方法来打印自己的描述，为了方便起见，添加了该方法可以更容易的获知你网络的样子。

**5.5.2.神经网络中的激活层**

接下来，您将提供您的第一层，激活层。你将使用你已经实现的Sigmoid函数。为了做反向传播，你还需要它的导数，这也是很容易实现。

Listing 5.14. Sigmoid函数的派生实现

def sigmoid\_prime\_double(x):

return sigmoid\_double(x) \* (1 - sigmoid\_double(x))

def sigmoid\_prime(z):

return np.vectorize(sigmoid\_prime\_double)(z)

请注意，对于Sigmoid本身，您提供了导数的标量版本和向量版本。现在，要定义一个使用sigmoid函数作为内置激活的ActivationLayer类，而Sigmoid函数没有任何参数，所以您不需要更新任何参数。

Listing 5.15. Sigmoid激活层

class ActivationLayer(Layer): #1

def \_\_init\_\_(self, input\_dim):

super(ActivationLayer, self).\_\_init\_\_()

self.input\_dim = input\_dim

self.output\_dim = input\_dim

def forward(self):

data = self.get\_forward\_input()

self.output\_data = sigmoid(data) #2

def backward(self):

delta = self.get\_backward\_input()

data = self.get\_forward\_input()

self.output\_delta = delta \* sigmoid\_prime(data) #3

def describe(self):

print("|-- " + self.\_\_class\_\_.\_\_name\_\_)

print(" |-- dimensions: ({},{})"

.format(self.input\_dim, self.output\_dim))

* 1 这个激活层利用sigmoid函数来激活神经元
* 2 向前传递只是将Sigmoid应用于输入数据
* 3 往回传递时的输出是用Sigmoid函数的导数与增量相乘

对于这一层，向后传递的只是将该层当前的元素增量与sigmoid导数乘积：​。

**5.5.3.Python中的Dense层作为前馈网络的构建块**

Dense层，是更复杂的层，也是您将在本章中实现的最后一个层。初始化这个层需要还有几个变量，权重矩阵、偏置项，以及它们各自的梯度。

Listing 5.16. Dense layer weight initialization

class DenseLayer(Layer):

def \_\_init\_\_(self, input\_dim, output\_dim): #1

super(DenseLayer, self).\_\_init\_\_()

self.input\_dim = input\_dim

self.output\_dim = output\_dim

self.weight = np.random.randn(output\_dim, input\_dim) #2

self.bias = np.random.randn(output\_dim, 1)

self.params = [self.weight, self.bias] #3

self.delta\_w = np.zeros(self.weight.shape) #4

self.delta\_b = np.zeros(self.bias.shape)

* 1 Dense层具有输入和输出尺寸
* 2 随机初始化权重矩阵和偏置向量
* 3 层参数由权重和偏置项组成
* 4 权重和偏差的∆设置为0

注意，您随机初始化了W和b。有许多方法可以初始化神经网络的权重。随机初始化是一个可以接受的方式，但其实还有许多更复杂的方式去初始化参数，可以更加准确地反映输入数据的结构。

**参数初始化是优化的一个起点**

初始化参数是一个有趣的主题，我们将在第6章中讨论一些其他的初始化技术。

现在，只要记住，初始化将影响你的学习行为。如果你想到图5.10的损失表面中，参数的初始化将意味着选择一个优化的起点；您可以很容易地想象到，图5.10损失面上的SGD的不同起点可能会导致不同的结果，这使得初始化成为神经网络研究的一个重要课题。现在，一个Dense层的向前传递是直接向前的。

现在，对于一个密集的层来说，向前传很简单。

Listing 5.17. Dense layer forward pass

def forward(self):

data = self.get\_forward\_input()

self.output\_data = np.dot(self.weight, data) + self.bias #1

* 1 Dense层的前传是仿射线性变换根据权重和偏差定义的输入数据

至于向后传递，请记住，要计算此层的增量，只需将W转置并乘以传入的增量： 。W和b的梯度也很容易计算d：ΔW=Δyt,Δb=Δ，其中y表示该层的输入（用您当前使用的数据计算）。

Listing 5.18. Dense layer backward pass

def backward(self):

data = self.get\_forward\_input()

delta = self.get\_backward\_input() #1

self.delta\_b += delta #2

self.delta\_w += np.dot(delta, data.transpose()) #3

self.output\_delta = np.dot(self.weight.transpose(), delta) #4

* 1 对于向后传递，您首先获得输入数据和delta。
* 2 当前的delta被添加到偏置增量中。
* 3 然后把这一项加上。
* 4 通过将输出delta传递到前一层。

根据您为网络指定的学习速率，通过累积增量来给出此层的更新规则。

Listing 5.19. Dense layer weight update mechanism

def update\_params(self, rate): #1

self.weight -= rate \* self.delta\_w

self.bias -= rate \* self.delta\_b

def clear\_deltas(self): #2

self.delta\_w = np.zeros(self.weight.shape)

self.delta\_b = np.zeros(self.bias.shape)

def describe(self): #3

print("|--- " + self.\_\_class\_\_.\_\_name\_\_)

print(" |-- dimensions: ({},{})"

.format(self.input\_dim, self.output\_dim))

* 1 使用权值和偏差delta，您可以用梯度下降法更新模型参数。
* 2 更新参数后，应该重置所有增量。
* 3 Dense层可以用它的输入和输出维度来描述。

**5.5.4.用Python做顺序神经网络**

把层作为一个网络的构建块，让我们转向网络本身。通过添加一系列空的层列表来初始化一个顺序神经网络，然后让它使用MSE作为损失函数。



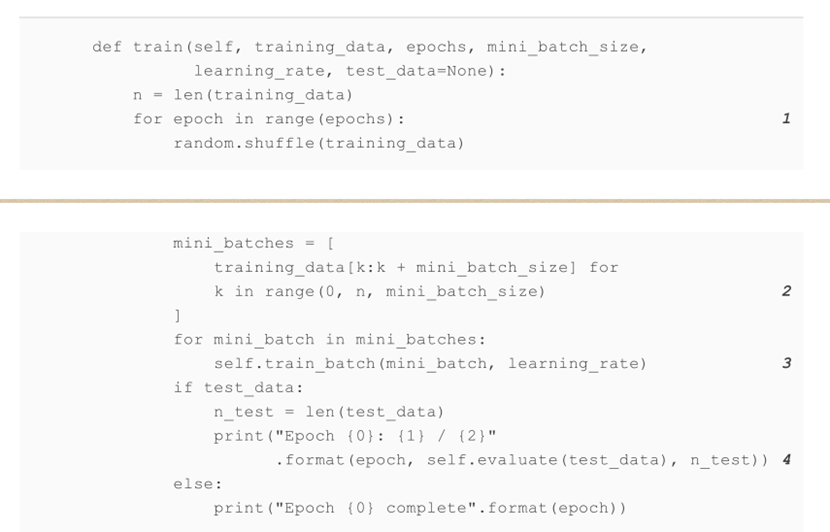
1. 在顺序神经网络中，你按顺序堆叠层。
2. 如果没有提供损耗函数，则使用MSE。

接下来，添加逐个添加层的功能。



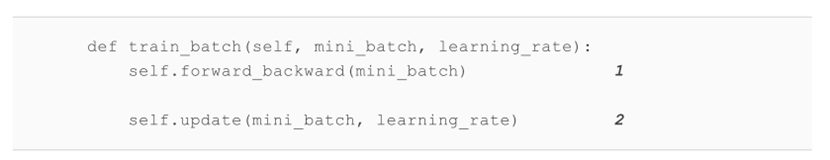
1. 无论何时添加一个新层，都要保证你把它和它的前任连接起来，让并可以描述自己。

网络实现的核心是训练方法。您使用小训练集作为输入：您对训练数据进行洗牌，并将其拆分为大小为mini\_batch\_size的训练集。为了训练你的网络，你需要一个接一个地喂它。为了提高学习能力，您将多次分批向网络提供您的训练数据。我们说我们训练它要经历多少轮次。如果提供了test\_data，则在每一轮之后评估网络性能。



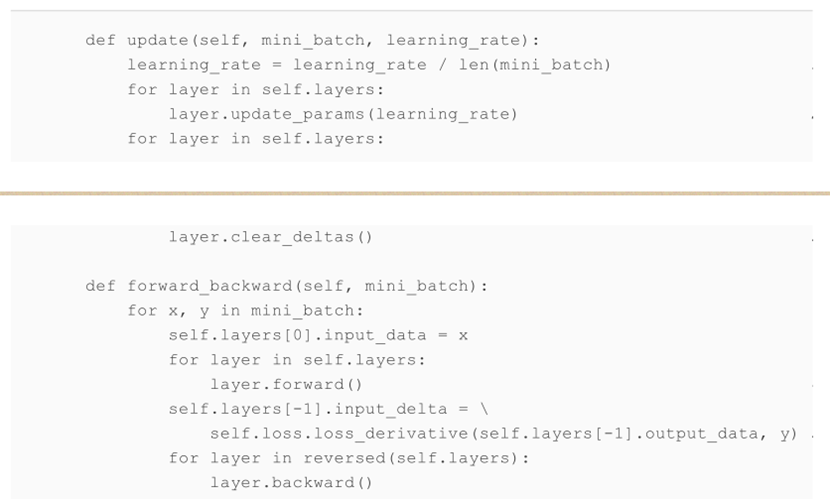
1. 为了训练你的网络，你传递数据的次数要和轮数一样多
2. 打乱训练数据并创建小训练集
3. 对于每一个小训练集，你都要训练你的网络
4. 如果您提供了测试数据，则在每轮之后对您的网络进行评估。

现在，您的train\_batch计算此小训练集的前向和后向传递，并在之后更新参数。



1. 要在一个小训练上训练网络，您需要计算前向和后向传递
2. 更新模型参数

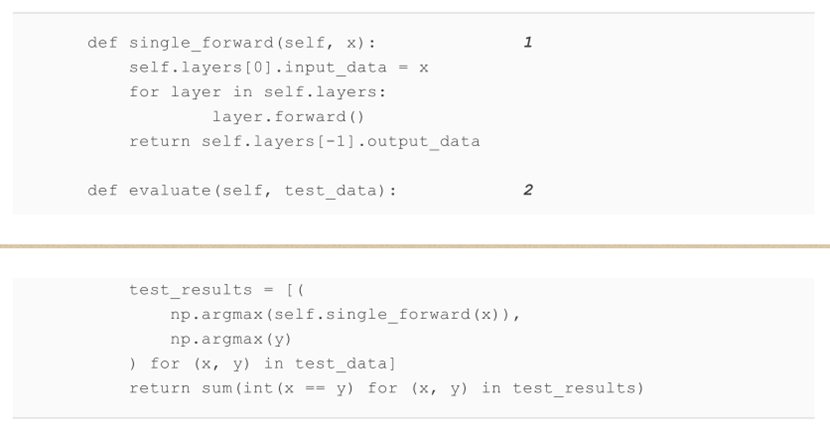
这两个步骤，update和forward\_backward，计算如下：



1. 一种常见的技术是通过小训练集大小来规范学习率
2. 更新该层所有参数
3. 清除每层所有增量
4. 对于小训练集中的每个样本，要逐层向前提供特征
5. 计算输出数据的损失导数
6. 误差项的逐层反向传播

实施是直截了当的，但有几个值得注意的点需要观察。首先，您将学习速率标准化为您的小训练集大小，以保持较小的更新。第二，在通过反向遍历层计算完整的向后传递之前，要计算网络输出的损失导数，它是作为向后传递的第一个输入增量。

SequentialNet work剩下未实现的一部分涉及模型性能和评估。要在测试数据上评估您的网络，您需要通过您的网络将这些数据向前传递，这是正是single\_forward所做的。评估发生在evaluate方法中，它会返回正确预测结果的数量以评估准确性。

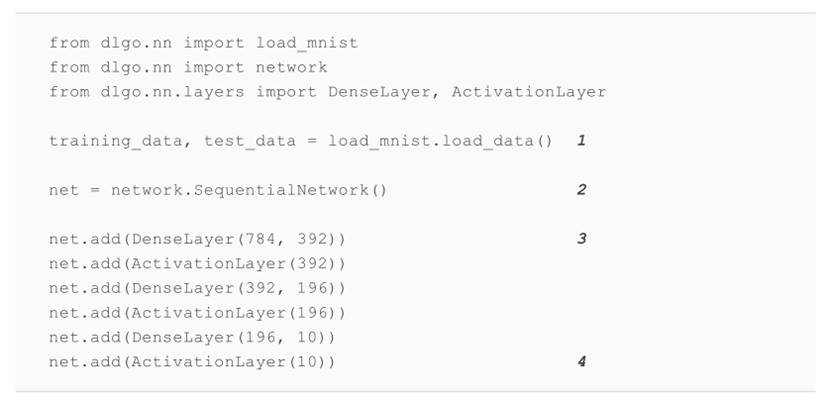


1. 向前传递单个样本并返回结果
2. 评估测试时的准确性

**5.5.5.应用你的网络手写数字分类**

在实现了前馈网络之后，让我们回到我们预测MNIST数据集手写数字的初始用例。在导入刚刚构建的必要类之后，我们加载MNIST数据，初始化网络，向其添加层，然后使用数据训练和评估网络。

要建立一个网络，请记住你的输入维度是784，而你的输出维度是10。您选择三个dense层，分别具有输出维度392、196和10，并在每个层之后添加Sigmoid激活。对于每一个新的dense层，你都有效地将层容量分成两半。层的大小和层数是该网络的参数，您已经选择了这些值来设置网络体系结构。



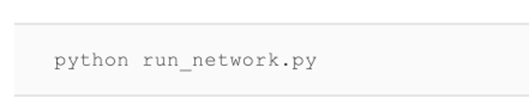
1. 加载训练数据
2. 初始化顺序神经网络
3. 逐层添加dense层和激活层
4. 最后一层大小是10，要产生预测

您通过调用train方法来训练网络的数据。您运行10轮的训练，并将学习率设置为3.0。作为小训练集，您选择10个类。如果您将训练数据完全打算，那么在大多数批次中，每个类都将被表示，从而得到良好的随机梯度。

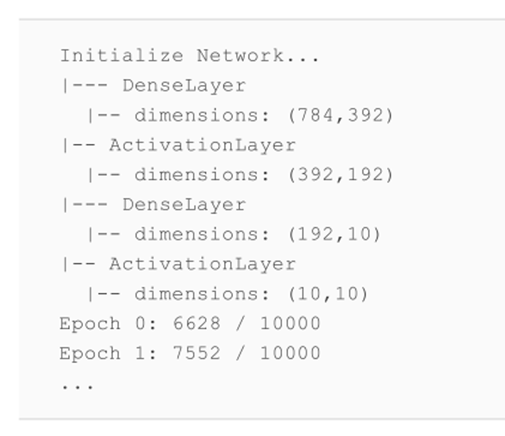


1. 您现在可以通过指定训练和测试数据轻松地训练模型，epoch的数量、迷你批处理大小和学习率。

现在可以运行了



这会产生以下提示:



你得到的每轮的数字在这里都不重要，因为事实上结果高度依赖于权重的初始化。但是值得注意的是，你经常在不到10轮时就具有95%以上的精度。这已经是一个相当大的成就，特别是考虑到你完全从零开始这样做。特别要提的是，这种模型的表现要你从本章开始的天真模型要好的多。不过，你可以做得更好。

请注意，对于您所研究的用例，您完全忽略了输入图像的空间结构而把它们当作向量。但应该清楚的是，给定像素的邻域是应该使用的重要信息。最终，你想回到围棋的游戏，如果你看到了第二章和第三章，那么就会明白（一串）棋子的邻域是多么重要。

在下一章中，您将看到如何构建一种更适合于检测空间数据的神经网络，如图像或围棋盘。这将使您更接近于在第7章中开发一个围棋AI。

**5.6.总结**

* 顺序神经网络是一种简单的人工神经网络，是由一层线性堆叠构建的。你可以将神经网络应用于各种各样的机器学习问题，包括图像识别。
* 前馈网络是一个连续的网络由具有激活函数的dense层组成的。
* 损失函数评估我们预测的质量。均方误差是实际应用中最常见的损失函数之一。一个损失函数给你一个严格的方法来量化你的模型的准确性。
* 梯度下降是一种损失最小化的算法。梯度下降包括沿着最陡峭的斜坡进行。在机器学习中，你使用梯度下降来找到最小损失的模型权重。
* 随机梯度下降是梯度下降算法的一种变化。随机梯度在下降时，您将计算您的训练集的一个小子集上的梯度，称为小训练集，然后根据每个小训练集去更新网络权重。随机梯度下降通常比大型训练集上的常规梯度下降快得多。
* 使用顺序神经网络，您可以使用反向传播算法有效地计算梯度。反向传播和小训练集的结合使训练足够快，足以在庞大的数据集上实用。

**Chapter 6. Designing a neural network for Go data**

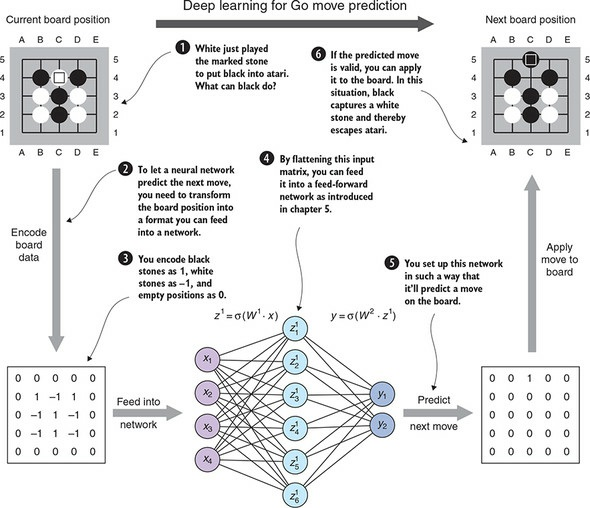
**第6章 为围棋数据设计神经网络**

本章涵盖

* 构建一个深度学习应用来预测下一步的围棋落子点
* 引入Keras深度学习框架
* 理解卷积神经网络
* 构建分析空间围棋数据的神经网络

在上一章中，您了解了神经网络在行棋中的基本原理，并从零开始实现了前馈网络。在本章中，你将把注意力转回到围棋游戏，并可以解决如何使用深度学习技术来预测在给定任何围棋盘面的情况下的下一步落子的问题。特别是，您将使用在第4章中运用的树搜索技术来生成围棋数据，并可以使用这个数据来训练神经网络。图6.1给出了您将在本章中构建的应用程序的概述。

图6.1 如何通过使用深度学习来预测围棋游戏中的下一步行动



如图6.1所示，要利用上一章中的神经网络工作知识，您必须首先解决一些关键步骤：

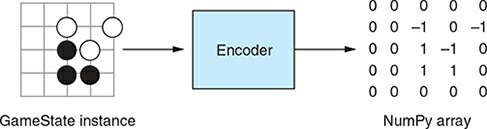
1. 在第3章中，你把重点放在通过在棋盘行棋告诉机器围棋的规则（实现围棋棋盘上的数据结构）。第4章使用这些结构进行树搜索。但在第5章中，你看到了神经网络需要数值的输入；对于你实现的前馈网络，向量vector是必须的。
2. 要将围棋棋盘局面转换为向量输入到神经网络中，您必须创建一个Encoder类来完成这项工作。图中6.1，我们勾画了一个简单的编码器，这会在6.1节中实现；围棋棋盘会被编码为一个棋盘大小的矩阵，白色的棋子表示为-1，黑色的棋子表示为1，空点表示为0。其矩阵可以被展开化为向量，就像你在前一章中对MNIST数据集所做的那样。虽然这种表示有点太简单了，不能为落子预测提供出色的结果，但这是朝着正确方向迈出的第一步。在第7章中，您将看到更复杂和更有用的方法来编码棋盘局面。
3. 要训练一个神经网络来预测下一步落子点，你必须先得到要输入到神经网络的数据。在6.2节中，您将使用第4章的技术来生成游戏记录。您将编码每个棋盘的局面，如刚才讨论的，这将作为您的特征，并将每个局面的下一步存储为标签。
4. 虽然你在第5章中实现的一个神经网络是有用的，但同样重要的是我们需要获得更多的速度和可靠性，因此我们引入一个更成熟的深度学习库。为此，在第6.3节介绍了Keras，一个用Python编写的流行深度学习库。您将使用Keras来构建预测落子的网络。
5. 在这一点上，你可能会想，为什么你完全放弃了围棋棋盘的空间结构，把编码的棋盘展平到一个向量。在第6.4节中，你将了解到一个新的层类型称为卷积层，它更适合您的用例。您将使用这些层来构建一个名为卷积神经网络的新体系结构。
6. 本章即将结束时，您将了解更多的现代深度学习的关键概念，这些概念将进一步提高落子预测的准确性，例如在第6.5节或第6.5节中使用Softmax有效地预测概率。在6.6节中建立更深层次的神经网络，并具有一个有趣的激活函数，称为线性整流函数（Rectified Linear Units, ReLU），又称修正线性单元。

**6.1 给棋盘局面进行编码**

在第三章中，您构建了一个Python类库，该库表示围棋游戏中的所有实体：Player、Board、GameState等。现在您想将机器学习应用于解决围棋中的一些问题。但是神经网络无法在像GameState类这样的高级对象上工作；它们只能处理数学对象，比如向量和矩阵。因此在本节中，我们将创建一个Encoder类，该类将你的本地游戏对象转换为数学形式。在本章的其余部分，您可以将该数学形式提供给你的机器学习工具。

建立围棋落子点预测的深度学习模型的第一步是加载可以输入神经网络的数据。通过为围棋棋盘定义一个简单的编码器来做到这一点，这在上图中已经介绍了。编码器是以合适的方式转换您在第3章中实现的围棋棋盘的一种方法。你所学到的神经网络，多层感知器，会作为输入的向量，但在6.4节中，您将看到另一个基于高维数据的网络体系结构。图6.2给出了如何定义这样一个编码器的想法。

图6.2 编码器类的说明。 它采用您的GameState类并将其转换为数学形式— NumPy数组。



其核心思想就是，编码器必须知道如何编码一个完整的围棋游戏状态。特别是，它应该定义如何在围棋板上对单点进行编码。有时反过来看也很有趣：如果你用网络预测下一步落子，该落子将被编码，您需要将其转换回围棋实际的落子。这个操作叫做解码，对于应用预测落子很重要。

考虑到这些，您现在可以定义Encoder类，这是您将在本章和下一章中将要创建的编码器的接口。您将在dlgo中定义一个名为Encoder的新模块。您将使用一个空的\_\_init\_\_.py进行初始化，并将文件base.py放入其中。然后，您将在下面的内容放入到文件中。

# 编码器接口

class Encoder:

# 允许您支持日志记录或保存你的模型中正在使用的编码器的名称

def name(self):

raise NotImplementedError()

# 把棋盘棋盘转化成数字数据

def encode(self, game\_state):

raise NotImplementedError()

# 将围棋棋盘点转换为整数索引

def encode\_point(self, point):

raise NotImplementedError()

# 将整数索引转换为围棋棋盘落子点

def decode\_point\_index(self, index):

raise NotImplementedError()

#围棋棋盘上的交叉点的数目 - 棋盘宽度乘以棋盘高度

def num\_points(self):

raise NotImplementedError()

# 已经编码的棋盘结构外形

def shape(self):

raise NotImplementedError()

编码器的定义很简单，但是我们希望在base.py中增加一个更方便的特性：一个函数，用它的名字创建一个编码器，一个字符串，而不是创建一个对象。你可以使用下面的get\_encoder\_by\_name函数进行此操作。

# 用来导入模块的库

import importlib

# 通过名字返回相应的编码器

def get\_encoder\_by\_name(name, board\_size):

if isinstance(board\_size, int):

board\_size = (board\_size, board\_size)

# 获取名字对应的模块

module = importlib.import\_module('dlgo.Encoder.' + name)

# 得到该模块用来初始化的create方法

constructor = getattr(module, 'create')

return constructor(board\_size)

现在您知道了编码器是什么，以及如何构建一个编码器，让我们实现第一个编码器：一种颜色表示为1，另一种颜色表示为-1，空点表示为0。为了得到准确的预测，模型也需要知道现在该轮到谁下了。因此，不要用1表示黑棋和-1表示白棋，你将使用1表示下一回合落子方，并用-1表示对手。你将把这种编码称为OnePlaneEncoder，因为您将要把围棋棋盘编码成与棋盘相同大小的单个矩阵或平面。在第7章中，您将看到具有更多特征平面的编码器；例如，您将实现这样一个编码器，一个平面放每个黑棋和白棋，一个平面用来识别劫。现在，您将在oneplane.py中实现的最简单的one-plane编码思想。下面的列表显示了第一部分。

class OnePlaneEncoder(Encoder):

def \_\_init\_\_(self, board\_size):

self.board\_width = board\_size

self.board\_height = board\_size

self.num\_planes = 1 # 平面数量

# 允许您支持日志记录或保存你的模型中正在使用的编码器的名称

def name(self):

return"oneplane"

# 把棋盘棋盘转化成数字数据,编码成三维矩阵

def encode(self, game\_state):

# 棋盘对应的三维矩阵

board\_matrix = np.zeros(self.shape())

current\_player = game\_state.current\_player

for r in range(self.board\_height):

for c in range(self.board\_width):

point = Point(row=r, col=c)

point\_color = game\_state.board.get(point)

if point\_color is None:

continue

# 是当前落子方，编码为1

elif point\_color == current\_player:

board\_matrix[0, r, c] = 1

# 不是当前落子方，编码为-1

else:

board\_matrix[0, r, c] = -1

return board\_matrix

在定义的第二部分，您将负责编码和解码棋盘的交叉点。编码是通过将棋盘上的一个交叉点映射到具有棋盘宽度乘以棋盘高度这样长度的一维向量来完成的；解码将一维转为二维来恢复点坐标。

# 将围棋棋盘点转换为整数索引

def encode\_point(self, point):

return self.board\_width\*(point.row-1)+point.col-1

# 将整数索引转换为围棋棋盘落子点

def decode\_point\_index(self, index):

r = index // self.board\_width+1

c = index % self.board\_width+1

return Point(row=r, col=c)

# 围棋棋盘上的交叉点的数目 - 棋盘宽度乘以棋盘高度

def num\_points(self):

return self.board\_width \* self.board\_height

# 已经编码的棋盘结构外形

def shape(self):

return self.num\_planes,self.board\_width,self.board\_height

这样就结束了我们关于围棋棋盘编码器的部分。您现在可以继续创建你可以编码的数据并将其输入到神经网络中。

**6.2.生成网络训练数据的树搜索游戏**

在您可以将机器学习应用于围棋游戏之前，您需要一组训练数据。幸运的是，棋力高的玩家一直在公共围棋服务器上玩。第7章中涉及如何查找和处理这些游戏记录来创建训练数据。目前，您可以生成自己的游戏记录。本节演示如何使用您在第4章中创建的树搜索机器人来生成游戏记录。在本章的其余部分，你可以使用这些机器人游戏记录作为训练数据来进行深度学习的实验。

如果传统算法不是很慢！用机器学习来模仿经典算法看起来是不是很愚蠢？在这里，您希望使用机器学习来对一个缓慢的树搜索的进行快速的近似。这个概念是AlphaGoZero的关键部分，这是AlphaGo的最强版本。第14章讲述了AlphaGoZero是如何工作的。

继续在dlgo模块之外创建一个名为generate\_mcts\_games.py的文件。正如文件名所建议的，您将编写用MCTS生成游戏的代码。每一个在这些游戏中的落子将被编码成第6.1节实现的OnePlaneEncoder，并存储在numpy数组，以供未来使用。首先，将下列导入语句放在generate\_mcts\_games.py.

import argparse # python自带的命令行参数解析包

import numpy as np

from dlgo.Encoder.Base import get\_encoder\_by\_name

from dlgo.agent import MCTSAgent

from dlgo.agent.FastRandomAgent import goboard\_fast as goboard

from dlgo.utils import print\_board,print\_move

从这些导入中，您已经可以看到哪些工具将会被使用：mcts模块、第三章中的goboard以及您刚刚定义的编码器模块。让我们把注意力转向为你生成游戏数据的函数上来。在generate\_game中，您让第4章的MCTSAgent的一个实例与自己对弈（回想第4章中的tempature一个MCTS来调节你的树搜索的波动性）。对于每一个落子，您在落子之前要对棋盘状态进行编码，并将棋盘编码为一个one-hot向量，然后将落子应用到棋盘上。

# 自我对弈生成游戏数据

def generate\_game(board\_size, round, temperature, max\_moves):

# 在围棋棋盘中存储编码的围棋棋盘状态；落子是编码的落子

boards, moves = [], []

# 获取oneplane编码器

encoder = get\_encoder\_by\_name("OnePlaneEncoder", board\_size)

# 开始游戏

game = goboard.GameState.new\_game(board\_size)

# 创建MCTS机器人

bot = MCTSAgent.MCTSAgent(round, temperature)

num\_moves = 0

while not game.is\_over():

print\_board(game.board)

# 让MCTS机器人选择落子点

move = bot.select\_move(game)

if move.is\_play:

# 把落子前的游戏进行编码，并加入到boards里

boards.append(encoder.encode(game))

# 将one-hot编码后的下一步落子添加到moves里

move\_one\_hot = np.zeros(encoder.num\_points())

move\_one\_hot[encoder.encode\_point(move.point)] = 1

moves.append(move\_one\_hot)

print\_move(game.current\_player, move)

game = game.apply\_move(move)

num\_moves += 1

if num\_moves > max\_moves:

break

return np.array(boards), np.array(moves)

现在您已经有了使用蒙特卡洛树搜索创建和编码游戏数据的方法，您可以定义一个主方法来运行几个游戏并在后面持久化，您也可以将它们放入到generate\_mcts\_games.py.

def main():

parser = argparse.ArgumentParser()

parser.add\_argument('--board-size', '-b', type=int, default=9)

parser.add\_argument('--rounds', '-r', type=int, default=1000)

parser.add\_argument('--temperature', '-t', type=float, default=0.8)

parser.add\_argument('--max-moves', '-m', type=int, default=60,

help='Max moves per game.')

parser.add\_argument('--num-games', '-n', type=int, default=10)

parser.add\_argument('--board-out')

parser.add\_argument('--move-out')

args = parser.parse\_args() # 此应用程序允许通过命令行参数自定义

xs = []

ys = []

for i in range(args.num\_games):

print('Generating game %d/%d...' % (i + 1, args.num\_games))

x, y = generate\_game(args.board\_size, args.rounds,

args.max\_moves, args.temperature) # 特定的游戏数量生成的游戏数据

xs.append(x)

ys.append(y)

x = np.concatenate(xs) # 在生成所有游戏之后，您将分别连接特征和标签

y = np.concatenate(ys)

np.save(args.board\_out, x) # 您将功能和标签数据存储到单独的文件中，如命令行选项所指定的那样。

np.save(args.move\_out, y)

if \_\_name\_\_ == '\_\_main\_\_':

main()

使用此工具，您现在可以轻松地生成游戏数据。假设您想为20个9×9Go游戏创建数据，并将功能存储在feature.npy中，并在label.npy中存储标签

请注意，这样生成游戏可能相当缓慢，因此生成大量的游戏将需要一段时间。你可以减少MCTS的轮数，但这也减少了机器人的水平。因此，我们已经为您生成了游戏数据，您可以下面地址中找到这些数据。

<https://github.com/maxpumperla/deep_learning_and_the_game_of_go/tree/chapter_6/code/generated_games>

此时，您已经完成了所有您需要的预处理，以便将神经网络应用于生成的数据。从第5章开始，您可以直接使用网络来实现一些东西--这是一个很好的锻炼方式-但向前看，你需要一个更强大的工具与日益复杂的深层神经网络来满足你的需要。为此，我们接下来介绍了Keras。

**6.3 使用Keras 深度学习库**

计算梯度和神经网络的反向传递越来越成为一种过时的艺术形式，因为许多强大的深度学习库的出现隐藏了较低层次的抽象概念。在前一章中从零开始实现神经网络是很好的，但现在是时候转向更成熟和功能丰富的软件了。

这个Keras深度学习库是一种特别优雅和流行的深度学习工具，用Python编写。开源项目创建于2015年，迅速积累了强大的用户基础，其代码托管在<https://github.com/keras-team/keras>，并且有很好的文档可以在<https://keras.io>上找到

**6.3.1.了解Keras的设计原则**

Keras的一个强项是它是一个直观且易于提取的API，其允许快速原型和快速实验周期。这使得Keras在许多数据科学挑战中成为一个受欢迎的选择，比如<https://kaggle.com>。Keras是模块化构建的，最初灵感来自于其他深度学习工具，如Torch。另一大优点是它的可扩展性，添加新的自定义层或增强现有功能相对简单。

另一个让Keras很容易的方面是它带有电池。例如，许多流行的数据集，如MNIST，可以直接加载Keras，您可以在GitHub存储库中找到许多很好的示例。除此之外，还有一个完整的keras扩展和独立项目社区生态系统<https://github.com/fchollet/keras-resources>

Keras的一个显著特点是后端的概念：它运行强大的引擎使得其可以交换需求。一种看待Keras的方法是将其作为一个深度学习前端，一个提供一组方便的高级抽象和功能来运行模型的库。在写这本书的时候，三个官方后端可供Keras使用：TensorFlow、Theano和the Microsoft Cognitive Toolkit。在这本书中，您将只与Google的TensorFlow库一起工作，该库也是Keras使用的默认后端。但是，如果你喜欢另一个后端，你不需要太多的努力去切换；Keras为你处理大部分的差异。

在本节中，您将首先安装Keras。然后您将通过运行第5章中的手写数字分类示例来了解它的API，然后转到围棋去预测落子。

**6.3.2.安装Keras深度学习库**

要开始使用Keras之前，您需要首先安装后端。您可以从TensorFlow开始，它通过PIP安装最简单，运行以下操作：

pip install tensorflow

如果您的机器中安装了NVIDIA GPU和CUDA驱动程序，您可以尝试安装GPU加速版本的TensorFlow。

pip install tensorflow-gpu

如果tensorflow-gpu与您的硬件和驱动程序兼容，这将给您带来巨大的速度提升。

一些有助于模型序列化和可视化的可选依赖项可以为使用Keras下载安装，但现在你将跳过它们，直接继续安装库本身：

pip install Keras

「注」国内下载推荐用douban镜像源加速

pip install --index-url https://pypi.douban.com/simple tensorflow

&&

pip install --index-url https://pypi.douban.com/simple Keras

**6.3.3.运行keras的第一个例子**

在本节中，您将看到定义和运行Keras模型要遵循四步工作流：

1. 数据预处理-加载和准备一个数据集，以输入到神经网络。
2. 模型定义-模型实例化模型并根据需要向其添加层。
3. 模型编译-用优化器、损失函数和可选的评估指标列表编译您以前定义的模型。
4. 模型的训练和评估-对您的深度学习模型进行训练和评估。

要开始使用Keras，我们将给您介绍上一章中遇到的一个示例用例：用MNIST数据集预测手写数字。如你所见，我们第5章的简单模型已经非常接近Keras语法，因此使用Keras就变得更容易。

使用Keras，您可以定义两种类型的模型：顺序模型和更一般的非顺序模型。您将在这里只使用顺序模型。这两种模型类型都可以在keras.models中找到。要定义一个顺序模型，您必须向它添加层，就像您在第5章中在自己的示例中所做的那样。可以通过keras.layers模块获得keras层。用Keras加载MNIST很数据集很简单，这个数据集可以在keras.dataset模块中找到。让我们先引入这个应用程序中所需要的一切。

import keras

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

接下来，加载和预处理MNIST数据，这只用几行就可以实现的。加载后，您将6万个训练样本和1万个测试样本展开（即原来的28*28展开成784*1），并将它们转换为float类型，通过除以255来规范输入数据。这样做是因为数据集的像素值从0到255不等，并且将这些值规范化为[0，1]的范围，因为这可以让你的网络有更好的训练。此外，标签必须是一个one-hot编码，就像你在第5章中所做的那样。下面的代码展示了我们用Keras去做刚才描述的事情。

Listing 6.9 使用Keras加载和预处理MNIST数据

# 加载数据

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# 将60000个训练样本和10000个测试样本都进行展平

x\_train = x\_train.reshape(60000, 784)

x\_test = x\_test.reshape(10000, 784)

# 转变数据类型

x\_train = x\_train.astype('float32')

x\_test = x\_test.astype('float32')

x\_train /= 255

x\_test /= 255

# 符合one-hot编码，将原来向量的每个值都转为矩阵中的一个行向量，比如原来是2，那对应的行向量是第3行是1

'''

# 类别向量定义

b = [0, 1, 2, 3, 4, 5, 6, 7, 8]

# 调用to\_categorical将b按照9个类别来进行转换

b = to\_categorical(b, 9)

print(b)

执行结果如下：

[[1. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 1. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 1. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 1. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 1. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 1. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 1.]]

'''

y\_train = keras.utils.to\_categorical(y\_train, 10)

y\_test = keras.utils.to\_categorical(y\_test, 10)

随着数据准备就绪，您现在可以继续定义要运行的神经网络。在Keras中，先初始化一个顺序模型，然后逐层添加。在第一层中，你必须提供i通过input\_shape提供的输入数据的形状。在我们的例子中，输入数据是长度为784的向量，因此必须提供input\_shape=（784，）（注：1可以省略）作为形状信息。Keras的Dense层可以被为该层提供激活函数的激活关键字创造出来。你将选择Sigmoid作为激活函数，因为它是到目前为止您所知道的唯一激活函数。Keras有更多的激活函数，其中一些我们之后会更详细地讨论。

「补充」激活函数是为了增加神经网络模型的非线性，负责将输入映射为输出，主要有sigmoid、Tanh、Relu

Listing 6.10. 使用Keras构建简单的顺序模型

# 新建一个顺序模型，给顺序模型添加Dense层

model = Sequential()

# 指定Dense层有392个神经元，激活函数为sigmoid，输入的向量形状是784\*1（第一层必须要输入)

model.add(Dense(392, activation="sigmoid", input\_shape=(784,)))

model.add(Dense(196, activation="sigmoid"))

model.add(Dense(10, activation="sigmoid"))

# 输出各层的参数情况

model.summary()

创建Keras模型的下一步是使用损失函数和优化器去编译模型。您可以通过指定字符串来做到这一点，您将选择sgd（随机梯度下降）作为优化器和使用mean\_square\_error(均方误差)作为损失函数。同样，Keras有更多的损失函数和优化器，但因为是刚开始，您将使用您已经在第5章遇到的东西。另一位需要争辩的是编译中你可以输入评估列表给keras模型。对于您的第一个应用程序，您将使用精确性作为唯一的度量。精度度量表示如何使在模型的最高得分预测可以匹配真正的标签。

# 编译模型，给定优化器为sgd，损失函数为mean\_squared\_error,度量根据精确性

model.compile(optimizer='sgd',loss='mean\_squared\_error',metrics=['accuracy'])

此应用程序的最后一步是执行网络的训练步骤，然后在测试数据上对其进行评估。这是通过调用model上的fit来完成的，它不仅提供训练数据，同时也提供每个梯度更新的样本数和训练模型的时期数

# 执行训练，再拿测试数据进行评估

model.fit(x\_train, y\_train,batch\_size=128,epochs=20)

# 评分包括损失值和你的精确值

score = model.evaluate(x\_test, y\_test)

print('Test loss:', score[0])

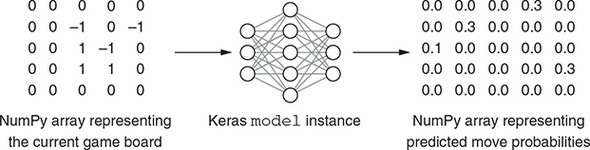
print('Test accuracy:', score[1])

总结，建立和运行Keras模型分为四个步骤：数据预处理、模型定义、模型编译和模型训练加评估。Keras的核心优势之一是这四步循环可以快速完成，从而导致一个快速的实验周期。这是非常重要的，因为通常您的初始模型定义可以通过调整参数得到很大的提高。

**6.3.4 利用Keras中的前馈神经网络进行落子预测**

现在，您已经知道了用于顺序神经网络的Keras的API是什么样子的，让我们回到我们的围棋落子预测用例。图6.3说明了该过程的这一步骤。你要先使用从6.2节生成的围棋数据，如下面代码所示。请注意，与前面的MNIST一样，您需要将围棋棋盘数据展平化为向量。

图6.13 神经网络可以预测游戏动作。 已经将游戏状态编码为矩阵，您可以将该矩阵提供给移动预测模型。 模型输出代表每个可能移动概率的向量。



神经网络可以预测游戏的落子。我们已经将游戏状态编码为矩阵，你可以将该矩阵提供给落子预测模型。该模型输出一个向量，表示每个可能落子的概率

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

# 通过设置随机种子，您可以确保此脚本完全可复制

np.random.seed(123)

# 加载特征(就是局面)

X = np.load("../generate\_game/features-200.npy")

# 加载标签(当前局面的落子）

Y = np.load("../generate\_game/features-200.npy")

# 样本数量

num\_samples = X.shape[0]

board\_size = 9

# 把特征和标签展平

X = X.reshape(num\_samples,board\_size\*board\_size)

Y = Y.reshape(num\_samples,board\_size\*board\_size)

# 拿出90%用来训练,10%用于测试

num\_train\_samples = int(0.9\*num\_samples)

X\_train,X\_test = X[:num\_train\_samples],X[num\_train\_samples:]

Y\_train,Y\_test = Y[:num\_train\_samples],Y[num\_train\_samples:]

接下来，让我们定义并运行一个模型来预测围棋落子。对于一个9×9的棋盘，有81个可能的落子，所以你需要用网络预测81类。伯克。假设你闭上眼睛，随意地指着棋盘上的一个点。你有1/81的机会找到下一个落子，所以你想要你的模型精度显著超过1/81。

定义了一个简单的Keras机器语言程序，它有三个Dense层，每个层都有Sigmoid激活函数，用均方误差作为损失函数和随机梯度下降作为优化器。然后，你让这个网络训练15轮，并评估它的测试数据。

# 2.模型定义

# 新建一个顺序神经网络，并添加3个Dense层

model = Sequential()

model.add(Dense(300,activation="sigmoid",input\_shape=(board\_size\*board\_size,)))

model.add(Dense(200,activation="sigmoid"))

model.add(Dense(board\_size\*board\_size,activation="sigmoid"))

model.summary()

# 模型定义结束

# 3.模型编译

model.compile(optimizer="sgd",loss="mean\_squared\_error",metrics=["accuracy"])

# 模型编译结束

# 4.模型训练与评估

# 模型训练，其中verbase为1表示显示进度条

# validation\_data用来在每轮之后，或者每几轮后都去验证一次验证集，用来及早发现问题，防止过拟合，或者超参数设置有问题。

model.fit(X\_train,Y\_train,batch\_size=64,epochs=15,verbose=1,validation\_data=(X\_test,Y\_test))

# 模型训练结束

# 模型评估

# verbose为0表示不显示进度条

score = model.evaluate(X\_test,Y\_test,verbose=0)

# 模型评估结束

# 得到错误数和精确数

print("loss:",score[0])

print("accuracy:",score[1])

运行此代码，您应该看到评估结果：

Layer (type) Output Shape Param # ================================================================= dense\_1 (Dense) (None, 1000) 82000 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dense\_2 (Dense) (None, 500) 500500 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dense\_3 (Dense) (None, 81) 40581 ================================================================= Total params: 623,081

Trainable params: 623,081

Non-trainable params: 0 ...

Test loss: 0.0129547887068

Test accuracy: 0.0236486486486

注意Trainable params中的623,081这一行,这意味着培训过程正在更新超过600,000个权重的值。 这是模型计算强度的粗略指标.它还使您对模型的能力有一个大概的了解：它学习复杂关系的能力。 当您比较不同的网络体系结构时，参数总数提供了一种近似比较模型总大小的方法。

正如你所看到的，你的实验的预测准确率只有2.3%左右，这一点乍一看并不令人满意。但是你随机猜测的正确率只有是1.2%左右。这告诉你，虽然性能不是很好，但模型正在学习，并且可以比随机预测更好地预测落子点。图6.4显示了一个棋盘局面，无论是哪一方下在A还是B，都可以在棋盘吃掉对方。这个局面不会出现在我们的训练集中。

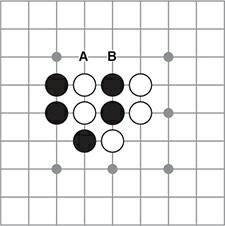


图6.4 一个测试我们模型的示例游戏局面。在这个局面下，黑棋可以通过在A处落子来吃掉两颗白棋，或者白色可以下在B处吃掉两颗黑棋。这样在游戏中就会有巨大的优势。

现在你可以把当前的局面输入到已经训练好的模型，并打印出它的预测。

test\_board = np.array([[

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 1, -1, 1, -1, 0, 0, 0, 0,

0, 1, -1, 1, -1, 0, 0, 0, 0,

0, 0, 1, -1, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

]])

move\_probs = model.predict(test\_board)[0]

i = 0

for row in range(9):

row\_formatted = []

for col in range(9):

row\_formatted.append('{:.3f}'.format(move\_probs[i]))

i += 1

print(' '.join(row\_formatted))

输出看起来像这样：

0.037 0.037 0.038 0.037 0.040 0.038 0.039 0.038 0.036

0.036 0.040 0.040 0.043 0.043 0.041 0.042 0.039 0.037

0.039 0.042 0.034 0.046 0.042 0.044 0.039 0.041 0.038

0.039 0.041 0.044 0.046 0.046 0.044 0.042 0.041 0.038

0.042 0.044 0.047 0.041 0.045 0.042 0.045 0.042 0.040

0.038 0.042 0.045 0.045 0.045 0.042 0.045 0.041 0.039

0.036 0.040 0.037 0.045 0.042 0.045 0.037 0.040 0.037

0.039 0.040 0.041 0.041 0.043 0.043 0.041 0.038 0.037

0.036 0.037 0.038 0.037 0.040 0.039 0.037 0.039 0.037

这个矩阵原来的9×9棋盘的映射：每个数字表示模型下在这一点上的可能性。但这个结果并不令人满意；它甚至没有学会避开已有落子的地方。但请注意，围棋棋盘边缘的分数总是低于接近中心的分数。围棋中的传统智慧是你应该避免在棋盘的边缘玩，除非在游戏结束和其他特殊情况下。因此，该模型已经学会了一个关于游戏的合理概念：不是通过理解策略或者效率，而只是复制我们的MCTS机器人所做的事情。这个模型不太可能预测许多好的落子，但它已经学会了避免一些糟糕的落子。

这是真正的进步，但是您可以做得更好。本章的其余部分解决了第一个实验的缺点，并在此过程中提高了围棋落子预测的准确性。 您将注意以下几点：

* 你是通过树搜索来预测落子的，它具有很强的随机性。有时MCTS引擎会产生奇怪的落子点，特别是当它们在游戏中不落后或远远落后。在第七章中，您将从人类游戏数据中创建一个深度学习模型。当然，人类也是不可预测的，但他们不太可能去走一些无意义的落子。
* 你的神经网络体系结构可以得到极大的改进。多层感知器不太适合捕获围棋棋盘数据。你必须把二维棋盘数据放平到一个平面向量上，丢弃关于围棋棋盘的所有空间信息。在6.4节中，您将了解一种新类型的网络，它在捕获围棋空间结构方面要好得多。
* 在所有的网络中，你只使用sigmoid激活函数。在第6.5节和第6.6节中，您将了解两个新的激活函数，这些函数通常会带来更好的结果。
* 到目前为止，你只使用MSE作为损失函数，它很直观，但不太适合你的用例。在6.5节中，你将使用另一个损失函数，该函数是为这样的分类任务量身定做的。

在本章结尾，我们就可以解决了其中的大部分问题，您将能够构建一个神经网络，该神经网络的预测比你之前的预测要好的多。在第七章中你将会学习构建一个更强大的机器人所需的关键技术

请记住，你不是为了能尽可能准确地预测落子，而是创建一个可以尽可能好地发挥水平的机器人。即使你的深层神经网络非常善于从历史数据中预测下一步的落子，但深度神经网络的核心仍然是获得棋盘结构并选择合理落子

**6.4 使用卷积网络对空间进行分析**

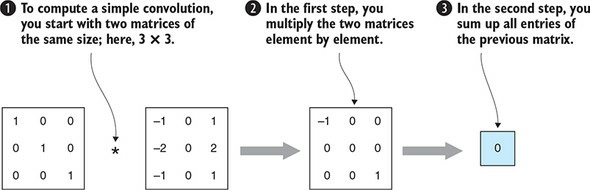
在围棋中，你经常会看到一些特定的局部棋形。人类玩家已经学会识别几十种棋形，并经常给它们起一些令人回味的名字（比如老虎口，双关，或我个人最喜欢的，花六）。要像人类一样做出决定，我们的围棋A I还必须认识到许多的局部棋形。现在有一种特殊类型的神经网络，称为卷积网络，它是专门为检测像这样的空间形状而设计的。卷积神经网络（CNNs）有很多游戏以外的应用：你会发现它们被应用到图像、音频，甚至文字。本节演示如何构建CNN并将其应用于围棋游戏数据。首先，我们要介绍卷积的概念。接下来，我们将展示如何在Keras中构建CNNs，最后，我们展示处理卷积层输出的有用方法。

**6.4.1 什么卷积是直观的？**

卷积层和我们构建的网络是从计算机视觉的传统操作中得到他们的名字：卷积。卷积是一种直观的变换图像或应用过滤器的方法。对于两个相同大小的矩阵，通过以下方法计算简单卷积：

1. 将这两个矩阵里的元素对应相乘
2. 计算先前矩阵所有值的和

这种简单卷积的输出是标量值。 图6.5显示了这种运算的示例，将两个3×3矩阵进行卷积以计算标量。



这些简单的卷积本身并不能马上对你有帮助，但它们可以用来计算更复杂的卷积，而更复杂的卷积对你的用例是有用的。现在我们一开始不从两个相同大小的矩阵开始，让我们固定第二个矩阵的大小，并任意增加第一个矩阵的大小。在这个场景中，您将第一个矩阵称为输入图像，第二个矩阵称为卷积内核，或简单的内核（有时您也会看到使用的过滤器）。由于内核比输入图像小，您可以在输入图像的许多块上计算简单的卷积。在图6.6中，你看到这样一个卷积操作，一个10×10的输入图像与一个3×3内核在相互作用。

图6.6 通过将卷积核传递到输入图像的斑块上，可以计算图像与内核的卷积。 在此示例中选择的内核是垂直边缘检测器。

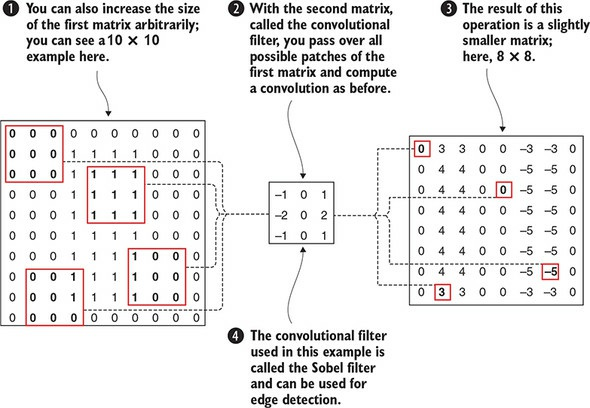


图6.6中的示例可能会给您第一个提示，来说明为什么卷积对我们来说是有趣的。输入图像是由中心4×8块的1被0包围的10×10矩阵。被选择的内核，矩阵的第一列（-1，-2，-1）为第三列（1，2，1）的相反数，中间列均为0。因此，以下几点是正确的：

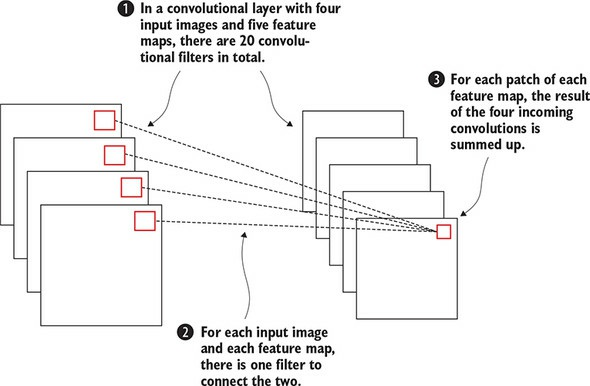
* 每当将此内核应用于所有像素值都相同的输入图像的3×3色块时，卷积的输出将为0。
* 当您将此卷积内核应用于图像块时，左列的值比右列高，卷积将是负的。
* 当您将此卷积内核应用于一个图像块时，右列的值比左边高，卷积将是正的。

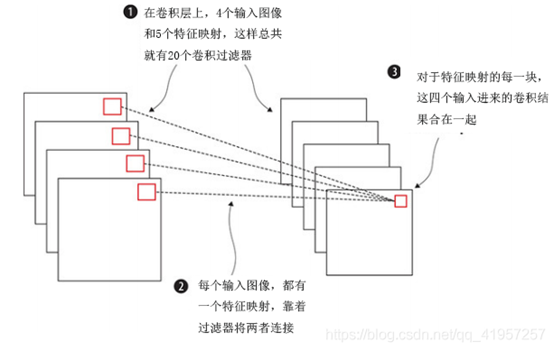
卷积内核被选择去检测输入图像中的垂直边缘。一个物体左边边缘将有正值，而右边边缘是负值。这正是您可以在图6.6中的卷积结果中看到的。

图6.6中的内核是许多应用程序中使用的经典内核，称为Sobel内核。如果你把这个内核翻转90度，你最终会得到一个水平边缘检测器。同样，您可以定义使图像模糊或锐化、检测角和任何其他事情的卷积内核，其中许多内核可以在标准图像处理库中找到。

有趣的是看到卷积可以用来从图像数据中提取有价值的信息，这正是您打算从围棋数据中预测下一步落子所要做的事情。虽然在前面的例子中，我们选择了一个特定的卷积核，但是卷积方式是在神经网络中使用这些核通过反向传播从数据中学习得到的。

到目前为止，我们已经讨论了如何将一个卷积核应用于一个输入图像。一般来说，将许多内核应用于许多图像以产生许多输出图像是有用的。为什么可以这样做？假设您有四个输入图像并定义了四个内核，然后你可以计算每个输入和输出图像的卷积之和。在下面的内容中，您将调用这样的卷积特征映射的输出图像。现在，如果您想要五个而不是一个结果的特征映射，你需要五个内核。利用n×m个卷积内核将n个输入图像映射到m个特征映射，称为卷积层。图6.7就说明了这种情况。





这样看来，卷积层是一种将多个输入图像转换为输出图像的方法，从而提取输入的相关空间信息。特别地，你可能有预想到，卷积层可以被链化，从而形成有卷积层的神经网络。通常，仅由卷积层和Dense层组成的网络被称为卷积神经网络，或者简单地说是卷积网络。

**深度学习中的张量**

我们需要指出的是，卷积层的输出是一堆图像。虽然这样子是有帮助的，但还有更多的事情要做。正如向量（1D）由个别条目组成，它们不仅仅是一堆数字。同样，矩阵（2D）由列向量组成，但其具有固定的二维结构，可用于矩阵乘法和其他操作（如卷积）。一个卷积层的输出具有三维结构。卷积层中的滤波器具有更多的一维，并且具有4D结构（每个输入和输出图像组合的二维滤波器），而且它并没有停止——先进的深度学习技术可以处理更高维度的数据结构。

在线性代数中，向量和矩阵的高维等价物是张量，在附录A有更多的细节。想得到张量的更多具体知识，但我们不能在这里讨论张量的定义，这本书的剩下部分中，你不需要知道任何正式的张量定义。但是，张量给了我们方便的术语，我们在后面的章节中将会使用。例如，从卷积层输出的图像集合可被称为3-Tensor，卷积层中的4D滤波器形成4-Tensor，因此你可以说卷积是一种运算一个4-Tensor（卷积滤波器）在一个3-Tensor（输入图像）上工作，并将其转换为另一个3-Tensor。

更一般地，你可以说顺序神经网络是逐步变换不同维数张量的一种机制。这种利用张量在网络中“流动”来输入数据的思想正是由此产生了TensorFlow这个名字，它是谷歌最受欢迎的机器学习库，你将可以用来运行你的Keras模型。

请注意，在所有这些讨论中，我们只讨论了如何通过卷积层提供数据，而没有讨论反向传播将如何做。我们故意把这一部分排除在外，因为它在数学上超出了这本书的范围，更重要的是，Keras已经为我们实现反向传递做了一些工作。

一般来说，卷积层与dense层相比有更少的参数。如果在28×28的输入图像上定义一个内核大小（3，3）的卷积层，将会导致26×26的输出，则卷积层具有3×3=9个参数。在卷积层中，您通常也会有一个偏置项，它被添加到每个卷积的输出中，因此总共产生了10个参数。作为比较，dense层连接28×28的输入向量和26×26的输出向量，则这样层将具有28×28×26×26=529984个参数，并且不包括偏移量。同时，卷积运算在计算上比dense层中使用的矩阵乘法要耗费更多的时候。

**6.4.2 使用Keras构建卷积神经网络**

要使用Keras构建和运行卷积神经网络，您需要使用一种名为Conv2D的新图层类型在二维数据上执行卷积，如围棋棋盘数据。你还可以了解另一个称为Flatten的层，它将卷积层的输出展平为向量，然后将其输入到dense层。

首先，您的输入数据的预处理步骤看起来与以前有点不同，并不是扁平化围棋棋盘，而是保持其二维结构完整。

import numpy as np

import keras

from keras.models import Sequential

from keras.layers import Dense

# 通过设置随机种子，您可以确保此脚本完全可复制

np.random.seed(123)

# 1.获取输入数据

# 加载特征(就是局面)

X = np.load("../generate\_game/features-200.npy")

# 加载标签(当前局面的落子）

Y = np.load("../generate\_game/labels-200.npy")

# 样本数量

num\_samples = X.shape[0]

board\_size = 9

# 输入数据形状是三维的；您使用一个平面表示9×9的棋盘。

input\_shape = (board\_size,board\_size,1)

# 将样本形状改掉

X = X.reshape(num\_samples,board\_size,board\_size,1)

# 拿出90%用来训练,10%用于测试

train\_num\_samples = int(0.9\*num\_samples)

X\_train,X\_test = X[:train\_num\_samples],X[train\_num\_samples:]

Y\_train,Y\_test = Y[:train\_num\_samples],Y[train\_num\_samples:]

现在您可以使用Keras的Conv2D对象来构建网络。需要使用两个卷积层，然后将第二层的输出展平，然后使用两个dense层，和以前一样。

#模型定义

model = Sequential()

# 网络中的第一层是具有32个输出滤波器,内核尺寸3\*3的Conv2D层

# 通常，卷积的输出尺寸小于输入。通过添加padding="same"，您可以要求Keras在边缘周围用0来填充矩阵，使得输出具有与输入相同的尺寸

model.add(Conv2D(filters=32,kernel\_size=(3,3),activation="sigmoid",padding='same',input\_shape=input\_shape))

# 与上一层几乎一样，只是因为不是第一层，不需要传入输入数据形状

model.add(Conv2D(filters=64,kernel\_size=(3,3),activation="sigmoid",padding='same'))

# 然后，将上一卷积层的3D输出压平

model.add(Flatten())

# 增加两个Dense层

model.add(Dense(300,activation="sigmoid"))

model.add(Dense(board\_size\*board\_size,activation="sigmoid"))

model.summary()

该模型的编译、运行和评估可以与前面的示例完全相同。

# 模型编译

model.compile(optimizer="sgd",loss="mean\_squared\_error",metrics=["accuracy"])

#模型训练和评估

model.fit(X\_train,Y\_train,batch\_size=64,epochs=15,verbose=1,validation\_data=(X\_test,Y\_test))

score = model.evaluate(X\_test,Y\_test,verbose=0)

print("loss:",score[0])

print("accuracy:",score[1])

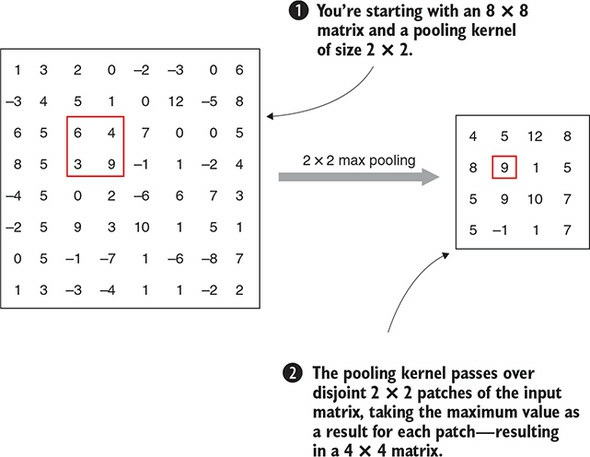
您唯一更改的是输入数据形状和模型本身的规范。

如果您运行起来，您将看到测试精度几乎没有改变：它应该再次降落在1.3%左右的某个地方。这完全没问题。在本章的其余部分，我们将介绍更先进的深度学习技术，以提高您的落子预测精度。

6.4.3 用池化层缩小空间

池化层pooling layer。在大多数具有卷积层的深度学习应用中，你会发现一种常见的技术是池化技术。你用池化去缩小图像，减少之前层神经元的数量。

池化的概念很容易解释：你可以通过将图像分组或池化图的块成一个简单值来降低图像样本。图6.8中的示例演示了如何切割图像，即保持图中2\*2的块最大值。

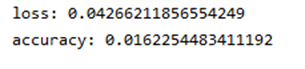


这种技术称为最大池，用于池的不相交块的大小称为池大小。您还可以定义其他类型的池；例如，计算一块中的平均值，这个版本称为平均池。您可以定义一个神经网络层，通常在卷积层之前或之后，如下所示。

# 增加2\*2的最大池

model.add(MaxPooling2D(pool\_size=(2, 2)))

加了一层池化层后结果如下，精确性提高了一些。



您也可以尝试使用Listing 6.4中的AveragePooling2D替换MaxPooling2D。结果如下，差不多

image-20210106164127633

在诸如图像识别的情况下，实际上在减小卷积层的输出大小时，池在实践中往往是必不可少的。尽管该操作会因对图像进行下采样而丢失一些信息，但通常会保留足够的信息以进行准确的预测，但与此同时会大大减少所需的计算量。

在您看到在池化层之前，让我们讨论一些其他工具，这些工具将使您的围棋落子预测更加准确。

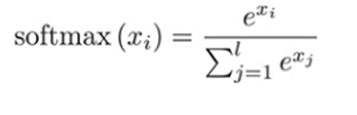
**6.5 预测围棋落子准确性**

自从我们在第五章中第一次引入神经网络以来，您只使用了一个激活函数：Sigmoid函数。另外，您一直使用均方误差作为损失函数。这两种选择在开始使用时挺好，可以在你的深度学习工具箱中占有一席之地，但并不特别适合我们的用例。

最后，当预测围棋落子时，一个真正的问题：对于棋盘上的每一个可能的落子点，这个落子点是下一步要下的落子点可能性有多大？在每个时间点上，许多围棋落子都是可以信赖的。深度学习可以充分了解游戏的结构，以预测落子的可能性。你想预测所有可能落子的概率分布，而sigmoid函数并不能保证实现这个功能。为了取代这个函数，您引入了Softmax激活函数，该函数用于预测最后一层的概率。

**6.5.1.使用最后一层的Softmax激活函数**

Softmax激活函数是Sigmoid的直接推广。要计算向量x=（x1，...，XL）的Softmax函数，首先应用指数函数到每个组件；然后计算。接着用所有值之和对每个值进行规范化：



根据定义，Softmax函数的分量是非负的，加起来是1，这意味着Softmax得出了概率。让我们用一个例子来看看它是如何工作的。

import numpy as np

def softmax(x):

e\_x = np.exp(x)

e\_x\_sum = np.sum(e\_x)

return e\_x / e\_x\_sum

x = np.array([100, 100])

print(softmax(x))

在Python中定义Softmax后，您将其计算长度为2的向量，即x=（100，100）。如果计算sigmoid（x），结果将接近（1，1），但是计算softmax(x)则是（0.5，0.5）。这就是你应该期待的：因为softmax函数的值会合并为1，而且两个条目是相同的，所以softmax分配给两个组件的概率相同的。 大多数情况下，你会看到softmax激活函数作为神经网络最后一层的激活函数，这样你就可以保证得到预测输出概率

model.add(Dense(9\*9, activation='softmax'))

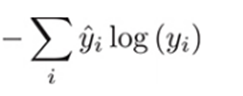
**6.5.2.分类问题的交叉熵损失**

在前一章中，你以均方误差作为损失函数，我们注意到它不是你用例的最佳选择。为了跟进这件事，让我们仔细看看什么地方可能出错，并提出一个可行的替代方案。

回想一下，您将您的落子预测用例描述为一个分类问题，其中您有9×9个可能的类，而只有其中一个类是对的。正确的类被标记为1，其他的都被标记为0。您对每个类的预测将始终是0到1之间的值。从你的预测数据看起来这是一个强有力的假设，而你正在使用的损失函数应该反映这一点。如果你使用均方误差，用预测和标签之间的差异的平方，事实上没有被限制在0到1的范围内。事实上，均方误差对于回归问题最有效，其中输出是一个连续的范围。想想预测一个人的身高。在这样的情景下，均方误差将限制巨大的差异，预测和实际结果之间的绝对最大差异趋于1。

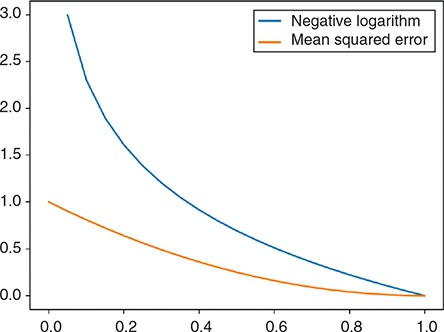
均方误差的另一个问题是它用同样地方式限制了所有81个预测值。最后，你只关心一个正确的类，标记为1。假设你有一个预测正确的落子，其值为0.6，而其他被标记为0的点分配概率加起来是0.4。在这种情况下，平均平方误差为(1-0.6)²+(0-0.4)²=2×0.4²，约为0.32。你的预测是正确的，但对于两个非零预测值你给出了相同的损失值：大约0.16。如果情况是正确落子点0.6，另外两个落子点是0.2，那么其均方误差是(1-0.6)²+2\*0.2²，大约0.24，这比前面的场景要低得多。但如果0.4的数值更精确一点，这个点也可能是下一落子的候选点，而你真的应该用你的损失函数去排斥这一点吗？

为了解决这些问题，我们介绍了分类交叉熵损失函数，简称交叉熵损失。对于模型的标签和预测值y，此损失函数定义如下：



请注意，尽管这看起来涉及到很多计算，但对于我们的用例，这个公式可以归结为一个简单的东西：其中一个是1。对于索引i，当是1的时候，交叉熵误差是简单的-log（）。很简单，你能从中得到什么？

* 由于交叉熵损失只限制标签为1的点，而其他所有值的分布并不会直接受它影响。特别地，在您以0.6的概率预测正确的下一步落子的场景中，另一个0.4落子点或另两个0.2落子点之间没有区别。在这两种情况下，交叉熵损失都是-log（0.6）=0.51。
* 交叉熵损失是根据[0，1]的范围量身定做的。如果你的模型预测实际落子点的概率为0，这肯定是错的。您知道log(1)=0，x在0到1之间的-log(x)在x接近0时接近无穷大，这意味着-log(X)可以变得任意大（而不像MSE那样二次变化）。
* 此外，当x接近1时，MSE下降得更快，这意味着对于不那么好的预测，损失要小得多。图6.9给出均方误差和交叉熵损失的直观对比。



区分交叉熵损失和均方误差的另一个关键点是它在随机梯度下降（SGD）学习过程中的行为。事实上，均方误差在获得更好预测值的时候，梯度会越来越小，导致其学习通常会减慢。与此相比，使用交叉熵损失时，SGD并没有放缓，参数更新与预测值和真实值之间的差异成正比。我们不能在这里讨论细节，但这对我们的落子预测用例来说是一个巨大的帮助。

使用分类交叉熵损失函数来编译keras模型而不用均方误差，其实现也是十分简单的。

model.compile(loss="categorical\_crossentropy"...)

随着交叉熵损失和Softmax激活函数应用到你的用例中，您现在可以更好地处理分类标签和使用神经网络预测概率。为了结束这一章节，让我们添加两种技术，允许您构建更深层次的网络

替换编译时用的损失函数，结果显示精确概率1.8%左右，比之前提高了一些

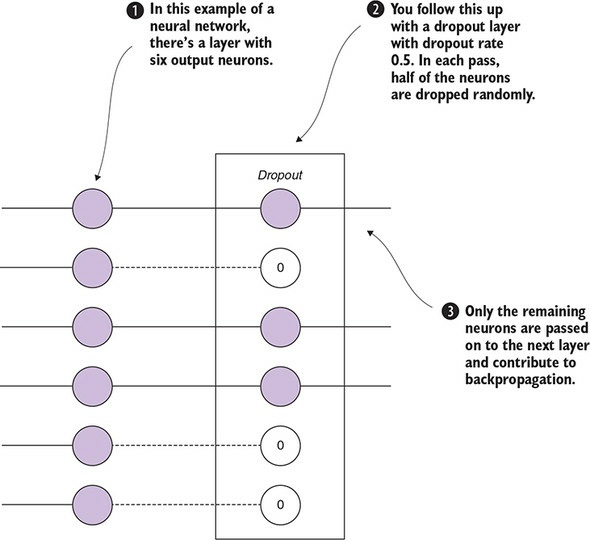
image-20210106164826924

**6.6 用ReLu和DropOut去构建更深层的网络**

到目前为止，你还没有建立一个超过2到4层的神经网络,你可能希望通过增加相同的内容去提高准确率，如果真能这样就太好了，但在实践中，你有几个方面需要考虑。尽管不断地建立更深层次的神经网络可以增加模型的参数数目，从而提高模型适应你输入数据的能力，但这样做你也可能会遇到麻烦。导致失败的主要原因之一是过度拟合：你的模型在预测训练数据方面变得越来越好，但是在测试数据上表现不行。举一个极端情况，对于一个几乎完全可以预测，甚至能够记住它以前看到过的东西的模型来说，当其遇到稍微有些不同的数据时就不知道该怎么办了，因此你需要会概括。对于像围棋这样复杂的游戏，要去预测下一步落子，这是特别重要的。不管你花多少时间去收集训练数据，在游戏中总是会出现你的模型以前没有遇到过的情况。

**6.6.1.将神经元丢弃使之规范化**

防止过度拟合是机器学习中普遍面临的挑战。您可以找到许多关于规范化技术的文献，这些技术旨在解决过度拟合问题。对于深度神经网络，你可以应用一种令人惊讶的简单却有效的技术，称为dropout。当dropout应用于网络中的一层时，每个训练步骤都会随机选择一些神经元设置为0；然后在训练过程中完全删除这些神经元。在每个训练步骤中，您随机选择要丢弃的新神经元。这通常是通过一个特定的丢弃概率来完成的。图6.10显示了一个dropout层的示例，在该层中，每个小训练集（前向和后向）的神经元都有一半的几率被丢弃。



这一过程背后的原理是，通过随机丢弃神经元，您可以避免单个层，从而防止整个网络对给定数据的过度拟合化。层必须要足够灵活，不能过分依赖单个神经元。通过这样做，你可以防止你的神经网络过度拟合。在Keras中，您可以定义具有降低活性率的dropout层，如下所示。

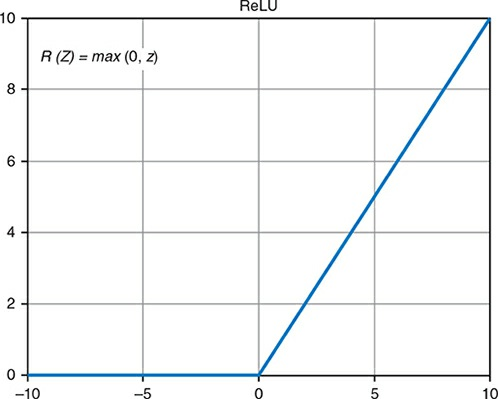
from keras.layers import Dropout

model.add(Dropout(rate=0.25))

您可以在其他的每层之前或之后的顺序网络中添加类似的dropout层。特别是在更深层次的体系结构中，添加dropout层往往是必不可少的。

**6.6.2 ReLU函数**

作为本节的最后一个构建块，您将了解校正线性单元（ReLU）激活函数，它通常比Sigmoid和其他激活函数对深度网络产生更好的激活功能。图6.11显示了ReLU的形状：



通过设置为0来忽略负输入，返回正输入不变。正信号越强，ReLU激活越强。考虑到这种解释，线性单元激活函数非常接近大脑中神经元的一个简单模型，其中较弱的信号被忽略，而较强的信号会导致神经元的放电。在这个基本的类比之外，我们不关心ReLU的任何理论好处，但请注意，使用它们往往会得到令人满意的结果。若要在Keras中使用Relu，请将任何层的激活参数Sigmoid替换为Relu，测试如下，精确概率提高了不少

image-20210106164958288

**6.7 把上面的改进全部放在一起加强网络预测准确率**

前面的章节不仅引入了具有最大池化层的卷积网络，而且还引入了交叉熵损失、最后一层的Softmax激活函数、丢弃规范，和ReLU激活函数以提高您的网络的性能。为了结束这一章，让我们一起把你学到的所有新的成分都输入一个神经网络，来看看你的神经网络在预测围棋落子方面的正确率怎么样。

首先，让我们回顾一下如何加载围棋数据，用简单的one-plane编码，并为了卷积网络重塑它的形状。

# 通过设置随机种子，您可以确保此脚本完全可复制

np.random.seed(123)

# 1.获取输入数据

# 加载特征(就是局面)

X = np.load("../generate\_game/features-200.npy")

# 加载标签(当前局面的落子）

Y = np.load("../generate\_game/labels-200.npy")

# 样本数量

num\_samples = X.shape[0]

board\_size = 9

# 输入数据形状是三维的；您使用一个平面表示9×9的棋盘。

input\_shape = (board\_size,board\_size,1)

接下来，让我们增强您以前的卷积网络，如下所示：

* 保持基本架构完整，从两个卷积层开始，然后一个最大池层和两个密集层。
* 为了规范化我们添加三个dropout层：在每个卷积层和第一个密集层之后。使用50%的丢弃率。
* 将输出层的激活函数更改为Softmax，内部层更改为ReLU。
* 改变损失函数为交叉熵损失，而不是均方误差。

#模型定义

model = Sequential()

# 网络中的第一层是具有32个输出滤波器,内核尺寸3\*3的Conv2D层

# 通常，卷积的输出尺寸小于输入。通过添加padding="same"，您可以要求Keras在边缘周围用0来填充矩阵，使得输出具有与输入相同的尺寸

model.add(Conv2D(filters=32,kernel\_size=(3,3),activation="relu",padding='same',input\_shape=input\_shape))

#添加dropout层

model.add(Dropout(rate=0.6))

# 与上一层几乎一样，只是因为不是第一层，不需要传入输入数据形状

model.add(Conv2D(filters=64,kernel\_size=(3,3),activation="relu",padding='same'))

# 添加一个最大池化层

model.add(MaxPool2D(pool\_size=(2,2)))

# model.add(AveragePooling2D(pool\_size=(2,2)))

#添加dropout层

model.add(Dropout(rate=0.6))

# 然后，将上一卷积层的3D输出压平

model.add(Flatten())

# 增加两个Dense层

model.add(Dense(300,activation="relu"))

#添加dropout层

model.add(Dropout(rate=0.6))

model.add(Dense(board\_size\*board\_size,activation="softmax"))

model.summary()

# 模型编译

model.compile(optimizer="sgd",loss="categorical\_crossentropy",metrics=["accuracy"])

最后，评估这个模型

#模型训练和评估

model.fit(X\_train,Y\_train,batch\_size=64,epochs=5,verbose=1,validation\_data=(X\_test,Y\_test))

score = model.evaluate(X\_test,Y\_test,verbose=0)

print("loss:",score[0])

print("accuracy:",score[1])

# 尝试预测

# 表示棋盘的矩阵

test\_board = np.array([[

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 1, -1, 1, -1, 0, 0, 0, 0,

0, 1, -1, 1, -1, 0, 0, 0, 0,

0, 0, 1, -1, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0,

]])

test\_board = test\_board.reshape(1,board\_size,board\_size,1)

# 输出一个棋盘局面下的预测值

move\_prob = model.predict(test\_board)[0]

i = 0

for row in range(9):

row\_formatted = []

for cow in range(9):

row\_formatted.append('{:.3f}'.format(move\_prob[i]));

i += 1

print(' '.join(row\_formatted))

**Chapter 7. Learning from data: a deep-learning bot**

**第7章 从数据中学习：深度学习机器人**

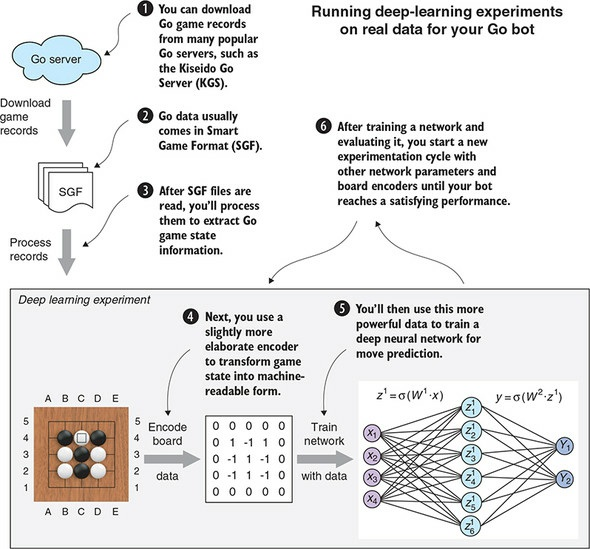
本章涵盖：

* 下载和处理实际的围棋游戏记录
* 了解存储围棋游戏的标准格式
* 训练一个使用这样的数据进行落子预测的深度学习模型
* 运行自己的实验并评估它们

在前一章中，您看到了构建深度学习应用程序的许多基本要素，并构建了一些神经网络来测试您所学到的工具。而关键的是你仍然缺少好的数据来学习。一个监督式的深度神经网络需要你提高好的数据——但目前为止，你只拥有自己生成的数据。

在这一章中，您将了解围棋数据最常见的数据格式-----SGF。您可以从几乎每个流行的围棋服务器中获得SGF游戏记录。为了加强深度神经网络的落子预测能力，在本章中，您将从围棋服务器中下载许多SGF文件，用智能的方式对它们进行编码，并使用这些数据训练神经网络。由此产生的经过训练的神经网络，将比以前的任何模型都要强得多。

图7.1说明了到本章结尾时可以构建的内容。



在本章的末尾，您可以使用复杂的神经网络运行自己的试验，完全独立地构建一个强大的AI。要开始，您需要访问真实的围棋数据。

**7.1. 引入围棋数据**

到目前为止，您使用的所有围棋数据都是由你自己生成的。在上一章中，你训练了一个深度神经网络来预测生成的数据的落子。你所希望的是你的网络可以完美地预测这些落子，在这种情况下，网络将像生成数据的树搜索算法一样发挥作用。在某种程度上，你输入的数据将会提供了一个深度学习机器人训练的上限。机器人不能超过输入的数据。「注：预测网络还是局限于人类的策略」如果利用强大的人类棋手游戏记录作为深层神经网络的输入，就可以大大提高您的机器人的水平。现在您将使用KGS围棋服务器（以前称为Kiseido GoServer）的游戏数据，这是世界上最流行的围棋游戏平台。在介绍如何从KGS下载和处理数据之前，我们将首先向你介绍围棋数据的数据格式。

**7.1.1 SGF文件格式**

SGF，80年代后期就开始开发。它目前的第四个主要版本（表示FF[4]）是在90年代后期发布的。SGF是基于文本的一种格式，可以用来表达围棋游戏及围棋游戏的变体（例如，围棋高手的游戏评论）以及其他棋盘游戏。章节的剩下部分，你将假设你正在处理的SGF文件是由围棋游戏组成，没有其他任何别的东西。在本节中，我们会教你一些关于这个丰富游戏格式的基本知识，但如果你想学习更多关于它的知识的话，请去<https://senseis.xmp.net/?SmartGameFormat>。

SGF主要包括游戏情况和落子数据，是通过两个指定的大写字母包在两个大括号里面。例如，在SGF中，一个大小为9×9的围棋盘将被编码为SZ[9]。围棋落子将会如下编码，在第三行和第三列上的一个交叉点上落白棋将是W[cc]，而在第七行和第三列上的一个交叉点上落黑棋将被表示成B[gc]；字母B和W代表棋子的颜色，行和列的坐标按字母顺序索引。若要表示pass，请使用空步骤B[ ]和W[ ]。

下面的SGF文件示例取自第二章9\*9棋盘上的完整对局。它显示了一个围棋游戏（GM[1]代表是围棋），HA[0]表示让子数为0，KM[6.5]表示贴目6.5，R U[Japanese]表示规则是日本规则，RE[W9.5]表示白赢了9.5目

(;FF[4] GM[1] SZ[9] HA[0] KM[6.5] RU[Japanese] RE[W+9.5]

;B[gc];W[cc];B[cg];W[gg];B[hf];W[gf];B[hg];W[hh];B[ge];W[df];B[dg]

;W[eh];B[cf];W[be];B[eg];W[fh];B[de];W[ec];B[fb];W[eb];B[ea];W[da]

;B[fa];W[cb];B[bf];W[fc];B[gb];W[fe];B[gd];W[ig];B[bd];W[he];B[ff]

;W[fg];B[ef];W[hd];B[fd];W[bi];B[bh];W[bc];B[cd];W[dc];B[ac];W[ab]

;B[ad];W[hc];B[ci];W[ed];B[ee];W[dh];B[ch];W[di];B[hb];W[ib];B[ha]

;W[ic];B[dd];W[ia];B[];

TW[aa][ba][bb][ca][db][ei][fi][gh][gi][hf][hg][hi][id][ie][if]

[ih][ii]

TB[ae][af][ag][ah][ai][be][bg][bi][ce][df][fe][ga]

W[])

一个SGF文件被组织成一个节点列表，节点由分号分隔。第一个节点包含有关游戏的信息：棋盘大小、使用的规则、游戏结果和其他背景信息。后面的每个节点表示游戏中的一个落子。最后，你也可以看到属于白棋地盘的点，列在TW之下，以及属于黑棋地盘的点，列在TB之下。

**7.1.2.从KGS下载和回放Go游戏记录**

如果你进入到 <https://u-go.net/gamerecords/> 你会看到一张表格，上面有可供下载的各种格式游戏记录。这个游戏数据是从KGS 围棋服务器收集的，所有这些游戏都是在19×19的棋盘上进行的，而在第六章中，我们为了减少计算而只使用了个9×9的棋盘。

这是一个令人难以置信的强大数据集，可以用于围棋落子预测，您将在本章中使用该数据集来为强大的深度学习机器人提供动力。您需要可以自动通过获取单个文件的链接下载，然后解压文件，最后处理其中包含的SGF游戏记录。

作为使用这个数据作为深度学习模型的输入第一步，你可以在主dlgo模块中创建一个名为data的新子模块，并像往常一样提供一个空的*init*.py。这个子模块将包含所有这本书所需的数据处理。

接下来，要下载游戏数据，您可以在数据子模块中添加新文件index\_processor.py中，并创建一个名为KGSIndex的类，然后实现其中的download\_files方法。这个方法将在本地镜像页面<https://ugo.net/gamerecords/>，找到所有相关的下载链接，然后在一个单独的名为data的文件夹中下载相应的tar.gz文件。方法如下：

Listing 7.1 创建包含来自KGS的Go数据的压缩包的索引

from dlgo.data.index\_processor import KGSIndex

index = KGSIndex()

index.download\_files()

运行该命令会得到如下的命令行输出：

>>> Downloading index page

KGS-2017\_12-19-1488-.tar.gz 1488

KGS-2017\_11-19-945-.tar.gz 945

...

>>> Downloading data/KGS-2017\_12-19-1488-.tar.gz

>>> Downloading data/KGS-2017\_11-19-945-.tar.gz

...

现在已经将数据存储在本地，接下来让我们处理它，以便在神经网络中使用。

**7.2 为深度学习准备数据**

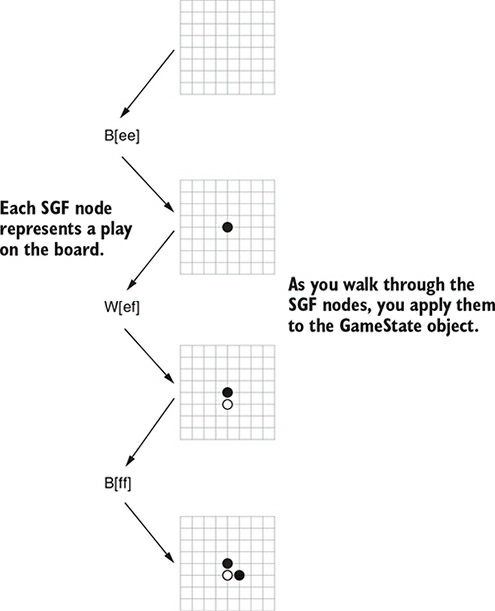
在第6章中，您看到了一个简单的围棋数据编码器，该编码器已经表示了在第3章中介绍的Board和GameState类。当使用SGF文件时，您首先需要对内容进行回放，产生对应的一个对局，得到必要的游戏信息。

**7.2.1.根据SGF记录重放围棋对局**

读取SGF文件的游戏信息意味着要理解和实现格式规范。虽然这并不是特别难做到（只是强加一个规则在一串文本上），这也不是构建围棋AI最令人兴奋的方面，需要大量的努力和时间才能做到完美无缺。出于这些原因，我们将引入另一个子模块gosgf到dlgo中，它负责处理SGF文件所需的所有逻辑。gosgf模块是从Gomill Python库改编而来的，地址是<https://mjw.woodcraft.me.uk/gomill/>

您将需要一个来自gosgf的实体，它足以处理您需要的所有内容：sgf\_game。让我们看看如何使用SGF\_Game加载一个SGF游戏，逐步读出游戏信息，并将落子应用于Game State对象。图7.2显示了围棋游戏的开始，用SGF命令表示。

图7.2 从SGF文件中重放游戏记录。原来的SGF文件编码游戏移动与字符串，如B[ee]。Sgf\_game类解码这些字符串并将它们作为Python元组返回。你就可以将这些落子应用到GameState对象以重建游戏



Listing 7.2. Replaying moves from an SGF file with your Go framework

from dlgo.gosgf import Sgf\_game #1

from dlgo.goboard\_fast import GameState, Move

from dlgo.gotypes import Point

from dlgo.utils import print\_board

sgf\_content = "(;GM[1]FF[4]SZ[9];B[ee];W[ef];B[ff]" + \ #2

";W[df];B[fe];W[fc];B[ec];W[gd];B[fb])"

sgf\_game = Sgf\_game.from\_string(sgf\_content) #3

game\_state = GameState.new\_game(19)

for item in sgf\_game.main\_sequence\_iter(): #4

color, move\_tuple = item.get\_move() #5

if color is not None and move\_tuple is not None:

row, col = move\_tuple

point = Point(row + 1, col + 1)

move = Move.play(point)

game\_state = game\_state.apply\_move(move) #6

print\_board(game\_state.board)

* 1 首先从新的gosgf模块导入Sgf\_game类。
* 2 定义一个示例SGF字符串。这些内容稍后将来自下载的数据。
* 3 使用from\_string方法，您可以创建Sgf\_game。
* 4 遍历这个游戏的主要序列;忽略变化和注释。
* 5 主序列中的项目以元组(颜色，移动)的形式出现，其中“move”是一对棋盘坐标
* 6 读出move然后可以应用到你的当前游戏状态。

从本质上讲，在您有了一个有效的SGF字符串之后，您就可以根据它得到主要序列，而这些序列你可以通过迭代得到。上面代码是本章的核心，它给出了一个粗略的大纲，告诉你将如何继续处理深度学习所需的数据：

1. 下载并解压缩围棋游戏文件。
2. 遍历这些文件中包含的每个SGF文件，读取文件中的内容变成字符串，然后从这些字符串中创建一个Sgf\_game。
3. 读出每个SGF字符串的围棋游戏的主要顺序，确保处理重要的细节，如放置棋子，并将产生的落子数据输入到GameState对象中。
4. 对于每一个落子，棋盘局面采用编码器进行编码成特征，并将落子本身存储为标签，然后将其放置在棋盘上。这样，您将创建落子预测数据，以便在后面训练中进行深入学习。
5. 将生成的特征和标签以合适的格式存储起来，这样您就可以稍后将其添加到神经网络中。

在接下来的几节中，您将非常详细地处理这五个任务。处理完这些数据后，您可以回到您的落子预测应用程序，看看如何让数据影响落子预测精度。

**7.2.2.构建围棋数据处理器**

在本节中，您将构建一个围棋数据处理器，该处理器可以将原始SGF数据转换为机器学习算法的特征和标签。这将是一个相对较长的实现，因此我把它分成几个部分。当你完成的时候，你就可以准备好在真实数据上运行一个深度学习模型。 开始，先在data模块下新建一个名为processor.py的新文件，让我们导入几个核心Python库，除了用于数据出来的NumPy之外，您还需要相当多的包来处理文件。

import os.path

import tarfile

import gzip

import glob

import shutil

import numpy as np

from keras.utils import to\_categorical

至于dlgo本身所需要的功能，您需要导入到目前为止构建的许多核心类。

Listing 7.4 从dlgo模块导入以进行数据处理

from dlgo.gosgf import Sgf\_game

from dlgo.goboard\_fast import Board, GameState, Move

from dlgo.gotypes import Player, Point

from dlgo.encoders.base import get\_encoder\_by\_name

from dlgo.data.index\_processor import KGSIndex

from dlgo.data.sampling import Sampler #1 从文件中采样训练和测试数据

我们还没有讨论清单中的最后两个导入(Sampler和DataGenerator)，但是将在构建Go数据处理器时引入它们。继续processor.py。通过提供一个字符串形式的编码器和一个存储SGF数据的data\_directory来初始化GoDataProcessor。

Listing 7.5 使用编码器和本地数据目录初始化Go数据处理器

class GoDataProcessor:

def \_\_init\_\_(self, encoder='oneplane', data\_directory='data'):

self.encoder = get\_encoder\_by\_name(encoder, 19)

self.data\_dir = data\_directory

接下来，您将实现主要的数据处理方法，称为load\_go\_data。在此方法中，您可以指定要处理的游戏数量以及要加载的数据类型，即训练或测试数据。load\_go\_data将从KGS中下载在线游戏记录，对指定数量的游戏进行采样，通过创建功能和标签进行处理，然后将结果持久化到本地作为NumPy数组。并行处理parallel.py

def load\_go\_data(self, data\_type='train', num\_samples=1000):

# 1

# 2

index = KGSIndex(data\_directory=self.data\_dir)

index.download\_files()

# 3

sampler = Sampler(data\_dir=self.data\_dir)

data = sampler.draw\_data(data\_type, num\_samples)

# 4

zip\_names = set()

indices\_by\_zip\_name = {}

for filename, index in data:

zip\_names.add(filename)

if filename not in indices\_by\_zip\_name:

indices\_by\_zip\_name[filename] = []

indices\_by\_zip\_name[filename].append(index)

# 6

for zip\_name in zip\_names:

base\_name = zip\_name.replace('.tar.gz', '')

data\_file\_name = base\_name + data\_type

if not os.path.isfile(self.data\_dir + '/' + data\_file\_name):

self.process\_zip(zip\_name, data\_file\_name,

indices\_by\_zip\_name[zip\_name])

# 7

features\_and\_labels = self.consolidate\_games(data\_type, data)

# 8

return features\_and\_labels

1. 对于data\_type，您可以选择训练或测试.
2. num\_samples表示要从其中加载数据的游戏数量。
3. 从KGS下载所有游戏到您的本地数据目录。如果数据可用，则不会再次下载。
4. 采样器为一个数据类型选择指定的游戏数量。
5. 在列表中收集数据中包含的所有zip文件名。
6. 按zip文件名称分组所有SGF文件索引。
7. 然后分别处理zip文件。
8. 然后聚合并返回每个压缩包中的特性和标签。

请注意，在下载数据之后，您可以使用一个Sampler实例对其进行分割，sampler所做的是确保它随机选择指定数量的游戏，但更重要的是，训练和测试数据不会以任何方式重叠。Sampler通过将训练和测试数据分割到一个文件级别来实现这一点，即简单地将2014年以前玩的游戏声明为测试数据，将更新的游戏声明为训练数据。这样做，您就可以绝对确保在训练数据中(部分)没有包含测试数据中可用的游戏信息，这可能会导致模型的过拟合（过拟合，指的是模型在训练集上表现的很好，但是在交叉验证集合测试集上表现一般，也就是说模型对未知样本的预测表现一般，泛化（generalization）能力较差。）

**分割训练和测试数据**

将数据分割成训练数据和测试数据的原因是为了获得可靠的性能指标。您可以根据训练数据训练一个模型，并根据测试数据对它进行评估，以查看该模型对以前未见的情况的适应能力，以及它从训练阶段学到的知识对现实世界的推断能力。正确的数据收集和分割对于信任您从模型中得到的结果至关重要。

加载你所有的数据，打乱它，然后随机地把它分成训练和测试数据。根据手头的问题，这种幼稚的方法可能是好主意，也可能不是。如果你考虑下围棋的比赛记录，那么一场比赛的走法是相互依赖的。用一组动作训练一个模型，这些动作也包含在测试集中，这会导致一种已经找到一个强大模型的错觉。但在实际操作中，你的机器人可能没有那么强大。一定要花时间分析你的数据，找到一个合理的分割。

**7.3.用人类数据进行深度学习训练**

现在您可以访问HighDan Go数据并对其进行处理以适应移动预测模型，让我们连接这些点并为这些数据构建一个深度神经网络。在我们的GitHub存储库中，在我们的DLGO包中有一个名为网络的模块，您将使用它来提供神经网络的示例体系结构，您可以使用它作为基线来构建强移动预测模型。因斯坦在网络模块中，您会发现三个不同复杂度的卷积神经网络，分别称为small.py、media.py和size.py。每个文件都包含一个返回的层函数可以添加到顺序Keras模型中的层的列表。您将构建一个由四个卷积层组成的卷积神经网络，然后是最后一个密集层，所有这些都是ReLUactiv。iations.除此之外，您将在每个卷积层之前使用一个新的实用程序层-Zero Patding2D层。零填充是一种操作，其中输入特性被填充为0。让我们一起是的，你使用你的一个平面编码器从第6章编码板作为一个19×19矩阵。如果您指定了2的填充，这意味着您添加了左右两列0，以及两行从0到该矩阵的顶部和底部，导致一个扩大的23×23矩阵。在这种情况下，使用零填充来人为地增加卷积层的输入，从而使co卷积操作不会使图像缩小太多。在我们给你看代码之前，我们必须讨论一个小的技术问题。回想一下，卷积层的输入和输出都是四个子国际：我们提供了一个小批量的过滤器，每个都是二维的（即它们有宽度和高度）。这四个维度的顺序（小批量大小，过滤器的数量，宽度和高度）是一个惯例问题，你在实践中主要发现两个这样的顺序。请注意，过滤器通常也被称为通道（C）和小批量大小也称为例子的数目（N）。此外，您可以使用速记宽度（W）和高度（H）。有了这个符号，两个主要的顺序是NWHC和NCWH。在凯拉斯，这个命令就是由于一些明显的原因，LLED数据*Format和NWHC被称为通道*last和NCWH通道\_first。现在，你建造第一个Go板编码器的方式，一个平面编码器，是在通道冷杉圣约定（编码板具有形状1,19，19，这意味着单个编码的平面是第一位的）。这意味着您必须首先提供data\_format=Channels\_first作为所有卷积层的参数。让我们看看这个模型是什么样子的。

Listing 7.18. Specifying layers for a small convolutional network for Go move prediction

from keras.layers.core import Dense, Activation, Flatten

from keras.layers.convolutional import Conv2D, ZeroPadding2D

def layers(input\_shape):

# 应对9\*9棋盘所需的层

# 使用ZeroPadding2D层放大层，避免卷积过后矩阵太小

return [

ZeroPadding2D(padding=3, input\_shape=input\_shape,

data\_format='channels\_first'), #1

# 通过使用channels\_first，您可以指定您的特征输入在平面维度优先。

Conv2D(48, (7, 7), data\_format='channels\_first'),

Activation('relu'),

ZeroPadding2D(padding=2, data\_format='channels\_first'), #2

Conv2D(32, (5, 5), data\_format='channels\_first'),

Activation('relu'),

ZeroPadding2D(padding=2, data\_format='channels\_first'),

Conv2D(32, (5, 5), data\_format='channels\_first'),

Activation('relu'),

ZeroPadding2D(padding=2, data\_format='channels\_first'),

Conv2D(32, (5, 5), data\_format='channels\_first'),

Activation('relu'),

Flatten(),

Dense(512),

Activation('relu'),

]

该层函数返回一个Keras层列表，您可以将其逐个添加到顺序模型中。使用这些层，您现在可以构建一个应用程序，从t开始执行前五个步骤他在图7.1中概述了一个应用程序，它下载、提取和编码Go数据，并使用它来训练神经网络。对于训练部分，您将使用您构建的数据生成器。但首先，让我们导入您正在成长的Go机器学习库的一些基本组件。您需要Go数据处理器、编码器和神经网络体系结构来构建此应用程序。

Listing 7.19. Core imports for building a neural network for Go data

from dlgo.data.parallel\_processor import GoDataProcessor

from dlgo.encoders.oneplane import OnePlaneEncoder

from dlgo.networks import small

from keras.models import Sequential

from keras.layers.core import Dense

from keras.callbacks import ModelCheckpoint # 1 存储进度

最后导入了名为ModelCheckpoint的Keras工具。因为你访问大量的数据进行训练去建立一个模型可能需要几个小时甚至几天。如果这样的实验因为某种原因而失败，你最好有一个备份。而这正是ModelChecpoint对你的作用：它们每轮训练完后都会保存一个模型。即使有些事情失败了，你也可以从最后一个检查点恢复训练。

接下来，让我们定义训练和测试数据。为此，首先初始化OnePlaneEncoder用来创建GoDataProcessor。使用此处理器，您可以实例化一个训练和一个测试数据生成器，该生成器将与Keras模型一起使用。

Listing 7.20. Creating training and test generators

if \_\_name\_\_ == '\_\_main\_\_':

go\_board\_rows, go\_board\_cols = 19, 19

num\_classes = go\_board\_rows \* go\_board\_cols # 初始化围棋数据进程

num\_games = 100

encoder = OnePlaneEncoder((go\_board\_rows, go\_board\_cols)) # 1 创建OnePlane编码器

processor = GoDataProcessor(encoder=encoder.name()) #2 初始化围棋数据进程

generator = processor.load\_go\_data('train', num\_games, use\_generator=True) # 3 创建训练数据生成器

test\_generator = processor.load\_go\_data('test', num\_games,

use\_generator=True) # 创建测试数据生成器

下一步，您可以使用dlgo.networks.small.py中的Layers函数来定义带有Keras的神经网络。你把这个小网络的层逐一添加到一个新的顺序网络中，然后最后添加一个最终的Dense层与Softmax激活。然后用分类交叉熵损失编译这个模型，并用SGD进行训练。

Listing 7.21. Defining a Keras model from your small layer architecture

input\_shape = (encoder.num\_planes, go\_board\_rows, go\_board\_cols)

network\_layers = small.layers(input\_shape)

model = Sequential()

for layer in network\_layers:

model.add(layer)

model.add(Dense(num\_classes, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='sgd', metrics=['accuracy'])

使用生成器训练Keras模型的工作方式与使用数据集的训练方式稍有不同。您现在需要调用fit\_generator，而不是在模型上调用fit，还需要替换evaluate为evaluate\_generator。此外，这些方法的特征与你之前看到的略有不同。使用fit\_generator通过指定一个generator，指定训练轮数，以及您提供的step\_per\_epoch。这三个参数提供了训练模型的最小值。您还希望用测试数据验证培训过程。为此，您可以使用测试数据生成器提供validation\_data，并将每轮的验证步骤数指定为validation\_steps。最后，在模型中添加一个回调,以便在每轮之后存储Keras模型。作为示例，你训练一个五轮模型，每批大小为128。

Listing 7.22. Fitting and evaluating Keras models with generators

epochs = 5

batch\_size = 128

model.fit\_generator(

generator=generator.generate(batch\_size, num\_classes), #1

epochs=epochs,

steps\_per\_epoch=generator.get\_num\_samples() / batch\_size, #2

validation\_data=test\_generator.generate(

batch\_size, num\_classes), #3

validation\_steps=test\_generator.get\_num\_samples() / batch\_size, #4

callbacks=[

ModelCheckpoint('../checkpoints/small\_model\_epoch\_{epoch}.h5')

]) #5

model.evaluate\_generator(

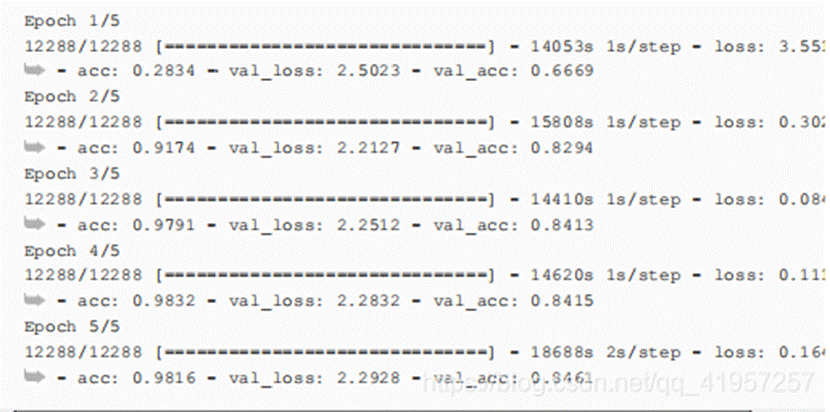
generator=test\_generator.generate(batch\_size, num\_classes),

steps=test\_generator.get\_num\_samples() / batch\_size) #6

请注意，如果您自己运行此代码，您应该知道完成此实验所需的时间。如果你在CPU上运行这个，训练一轮可能需要几个小时。恰好，机器学习中使用的数学与计算机图形学中使用的数学有很多共同之处。因此，在某些情况下，您可以将您的神经网络计算移动到您的GPU上,这样你可以获得一个大的加速。

如果你想使用GPU进行机器学习，那么带有Windows或Linux操作系统的NVIDIA芯片是最好的支持组合。

如果你不想自己尝试这个，或者只是现在不想这样做，我们已经为你预先计算了这个模型。看看我们的GitHub存储库，以下是训练运行的输出（计算在笔记本电脑上的旧CPU上，以鼓励您立即获得快速GPU）



正如你所看到的，经过三轮，就已经达到了98%的训练准确率和84%的测试数据。这是一个巨大的进步，为了建立一个真正强大的对手，您需要下一步使用更好的围棋数据编码器。在第7.4节中，您将了解两个更复杂的编码器，这将提高您的培训性能。

**7.4.建造更好的围棋数据编码**

第2章和第3章涵盖了围棋中的打劫规则。回想一下，这个规则的存在是为了防止游戏无限循环。如果我们给你一个随机的棋盘局面，你必须要判断是否有有一个劫。如果没有看到导致这个局面的序列，就没有办法知道。特别是，你使用一个平面编码器，它将黑色的棋子编码为1，白色的棋子编码为-1，空的位置编码为0，这样根本不可能了解任何关于劫的信息。你在第6章中构建的OnePlaneEncoder是有点过于简单，无法捕捉到构建强大围棋AI所需的所有内容。

在本节中，我们将为您提供两个更精细的编码,可以导致相对较强的落子预测性能。第一个我们称为SevenPlaneEncoder，它由七个特征平面组成。每架平面都是19×19矩阵，描述了一组不同的特征：

* 第一个平面用1表示只有1口气的白棋，否则都是0。
* 第二和第三特征平面用1分别表示有两口或至少三口气的白棋。
* 第四到第六个平面对于黑石头也是如此；它们编码1，2或至少3口气的黑棋。
* 最后一个平面再标记不能落的点，因为劫被标上了1。

除了显式编码ko的概念外，使用这组功能，您还可以模拟气，并区分黑棋和白棋。只有一口气的棋子有额外的意义，因为它们有可能在下一个回合被吃掉。因为该模型可以直接“看到”这个属性，所以它更容易了解这是如何影响游戏的。通过为诸如劫和气等概念创建平面，你可以给出对模型的暗示，这些概念是重要的，而不必解释它们是如何或为什么重要的。

让我们看看如何通过从编码器扩展来实现这一点。将下面的代码保存在sevenplane.py中。

Listing 7.23. Initializing a simple seven-plane encoder

import numpy as np

from dlgo.encoders.base import Encoder

from dlgo.goboard import Move, Point

class SevenPlaneEncoder(Encoder):

def \_\_init\_\_(self, board\_size):

self.board\_width, self.board\_height = board\_size

self.num\_planes = 7

def name(self):

return 'sevenplane'

下面实现编码:

Listing 7.24. Encoding game state with a SevenPlaneEncoder

def encode(self, game\_state):

board\_tensor = np.zeros(self.shape())

# 白棋从0下标平面开始，黑棋从3下标平面开始

base\_plane = {game\_state.next\_player: 0,

game\_state.next\_player.other: 3}

for row in range(self.board\_height):

for col in range(self.board\_width):

p = Point(row=row + 1, col=col + 1)

go\_string = game\_state.board.get\_go\_string(p)

if go\_string is None:

# 最后一层设置劫：把不能提回的劫设为1

if game\_state.does\_move\_violate\_ko(game\_state.next\_player,

Move.play(p)):

board\_tensor[6][row][col] = 1

# 第1-3层留给白棋,分别存1,2，至少是3口气的白棋

# 第4-6层留给黑棋，分别存1,2，只杀3口气的白棋

#1

else:

liberty\_plane = min(3, go\_string.num\_liberties) - 1

liberty\_plane += base\_plane[go\_string.color]

board\_tensor[liberty\_plane][row][col] = 1

#2

return board\_tensor

* 前四个特征平面分别描述具有1、2、3、4口气的黑棋。
* 后四个特征平面分别描述具有1、2、3、4口气的白棋。
* 如果轮到黑棋下，第九个特征平面设为1，如果轮到白下，就把第十个特征平面设为1。
* 最后一个特征平面同样表示劫。

import numpy as np

from dlgo.Encoder.Base import Encoder

from dlgo.gotypes import Point,Player

from dlgo.agent.FastRandomAgent.goboard\_fast import Move

class ElevenPlaneEncoder(Encoder):

def \_\_init\_\_(self, board\_width, board\_height):

self.board\_width = board\_width

self.board\_height = board\_height

self.num\_planes = 11

def name(self):

return "ElevenPlanEncoder"

def encode(self, game\_state):

board\_matrix = np.zeros(self.shape())

# 白棋从0下标平面开始，黑棋从3下标平面开始

base\_plane = {

game\_state.current\_player: 0,

game\_state.current\_player.other: 4

}

# 轮到黑下

if game\_state.current\_player == Player.black:

board\_matrix[8] = 1

# 轮到白下

else:

board\_matrix[9] = 1

for row in self.board\_width:

for col in self.board\_height:

point = Point(row+1, col+1)

point\_string = game\_state.board.get\_go\_string(point)

if point\_string is None:

# 最后一层设置劫：把不能提回的劫设为1

if game\_state.does\_move\_violate\_ko(game\_state.current\_player, Move.play(point)):

board\_matrix[10][row][col] = 1

else:

# 第1-4层留给白棋,分别存1,2，3,4口气的白棋

# 第5-8层留给黑棋，分别存1,2，3,4口气的白棋

liberty\_plane = min(4, point\_string.num\_liberties)-1

liberty\_plane += base\_plane[point\_string.color]

board\_matrix[liberty\_plane][row][col] = 1

def encode\_point(self, point):

return self.board\_width\*(point.row-1)+(point.col-1)

def decode\_point\_index(self, index):

row = index // self.board\_width + 1

col = index % self.board\_width + 1

return Point(row, col)

def num\_points(self):

return self.board\_height\*self.board\_width

def shape(self):

return self.num\_planes, self.board\_width, self.board\_height

def create(board\_width, board\_height):

return ElevenPlaneEncoder(board\_width, board\_height)

这个有11个平面的编码器更加具体地说明了一串棋子的气。两个都是很好的编码器，将会导致模型性能的显著改进。

在整个第5章和第6章中，你了解到了许多深度学习的技术，但其中一个重要的实验要素：您使用随机梯度下降（SGD）作为优化器。虽然SGD提供了一个很好的基线，但在下一节中，我们将教您Adagrad和Adadelta这两个优化器，使你的训练过程将大大受益。

**7.5 具有适应性的培训**

为了进一步提高围棋落子预测模型的性能，我们将在本章中介绍最后一组工具-随机梯度下降(SGD)以外的优化器。回顾第5章，SGD有一个相当简单的更新规则。如果对于参数W，您接收到ΔW的反向传播误差，还有特定的α学习速率，则用SGD更新此参数仅仅是计算。在许多情况下，这种更新规则可以导致良好的结果，但也存在一些缺点。为了解决这些问题，您可以使用其他许多优秀的优化器

**7.5.1.SGD的衰退和动量**

例如，一个广泛使用的想法是让学习率随着时间的推移而衰减；随着您采取的每一个更新步骤，学习率就会变小。这种技术通常很有效，因为在开始阶段，你的网络还没有学到任何东西，因此大的更新步骤可能会导致最小的损失，但当训练过程达到一定程度后，你应该使您的更新变小，并且只对不破坏进度的学习过程进行适当的改进。通常，你指定了一个衰减率来表示学习率衰减，这个百分比下降使得你会减少下一步的。

另一种流行的技术是动量技术，其中最后一个更新步骤的一小部分被添加到当前的更新步骤中。例如，如果W是你想要更新的参数向量，而]W是W的当前梯度，如果您使用的最后一次更新是U，那么下一个更新步骤将如下：

img

从上次更新中保留的这个分数g称为动量项。如果两个梯度项指向大致相同的方向，则下一个更新步骤将得到加强（接收动量）。如果梯度指向相反的方向，它们相互抵消，梯度受到抑制。这种技术被称为动量，因为物理概念同名相似，你可以将你的损失函数看成表面，而里面的参数则像一个滚下表面的球，而参数的更新就好像球在移动。如果你正在做梯度下降，你就可以想象成球在一个接一个地往下滚。如果最后几步（梯度）都指向同一个方向，球就会加快速度到达它的目的地。动量技术就利用了这种类比。

如果您想在SGD中使用衰变、动量或两者兼而有之，那就提供各自的比率。假如SGD的学习率为0.1，衰减率为1%，动量为90%，你会做以下工作

from keras.optimizers import SGD

sgd = SGD(lr=0.1, momentum=0.9, decay=0.01)

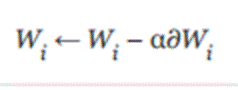
**7.5.2 利用Adagrad优化神经网络**

学习率衰减和动量都在改进普通SGD方面做得很好，但仍然存在一些弱点。例如，如果你想到围棋棋盘，专业棋手几乎前几步只会下在棋盘的第三到五行，从来不会下在第一或第二行，但在对局结束时，形势有些逆转，因为最后的许多棋子会落在棋盘的边界。在你迄今为止使用的所有深度学习模型中，最后一层是Dense层（这里是19×19）。这一层的每个神经元对应一个棋盘上的落子点。如果你使用SGD，无论是否有动量或衰减，这些神经元的学习速率是相同的，这样就可能出现问题。也许你在训练中的糟糕数据，而且其学习率已经下降了很多，以至于在第一行和第二行上的落子不再得到任何重要的更新，这样这意味着没有学习。一般来说，你想确保很少观察到的模式仍然得到足够大的更新，而频繁使用的模式收到越来越小的更新。

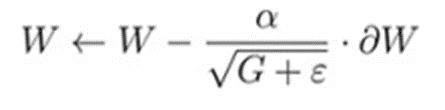
要解决设置由全局学习率引起的问题，您可以使用自适应梯度的技术。我们将向你们展示两种方法：Adagrad 和Adadelta

在Adagrad中，没有全局学习率，您可以调整每个参数的学习率。当你有很多数据时，Adagrad可以工作得很好，而且数据中的模式很少能找到。这些标准都非常适用于我们的情况：你虽然有很多数据，但专业的围棋对局是非常复杂，以至于某些落子组合很少出现在你的数据集中。

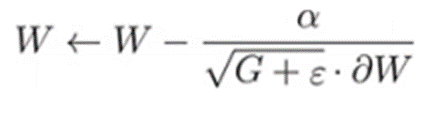
假设你有一个长度为l的权重向量W（在这里更容易想到向量，但这种技术也更普遍地适用于张量），其中单独的分量设为。对于这些参数给定梯度]W，在学习速率为a的普通SGD中，每个Wi的更新规则如下：



在Adagrad中，您用一个东西替换α，它可以通过查看你过去更新了多少Wi来动态适应每个索引i。事实上，在Adagrad中，个人的学习率是与先前的更新成正比的。更准确地说，在Adagrad，您更新参数如下：



在这个式子中，ε是一个很小的正值，以确保分母不为0，而GI是到这一点平方梯度Wi的总和。我们把这个写成Gi，是因为你可以看到它作为长度为l的平方矩阵G的一部分，其中所有对角线项Gj都有我们刚才描述的形式，所有非对角线项都是0，因此这种形式的矩阵叫做对角矩阵。每次参数更新后，通过向对角线上元素添加最新的梯度来更新G矩阵，但如果您想将此更新规则写成简洁的形式使其独立于索引i，式子如下



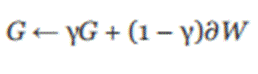
请注意，由于G是一个矩阵，您需要在每个分量Gi中添加ε，并将α除以每个分量。此外，用G.]W表示G与W的矩阵乘法。使用此优化器创建模型的工作如下。

from keras.optimizers import Adagrad adagrad = Adagrad()

与其他SGD技术相比，Adagrad的一个关键好处是你不必手动设置学习速率。事实上，您可以通过使用Adagrad（lr=0.02）来改变Keras的初始学习率，但不建议这样做

**7.5.3.利用Adadelta精炼自适应梯度**

一个类似于Adagrad的优化器是Adadelta。在这个优化器中，G矩阵中不是累积所有过去的（平方）梯度，而是使用我们的动量技术，只保留上次更新的一小部分，并将当前梯度添加到它上面：



虽然这个想法大致是在Adadelta发生的事情，但使这个优化器工作的细节在这里讲有点太复杂了。我们建议你查看原始文件以了解更多细节（<https://arxiv.org/abs/1212.5701>）

在keras中，你要这样使用Adadelta优化器:

from keras.optimizers import Adadelta

adadelta = Adadelta()

与随机梯度下降(SGD)相比，Adagrad和Adadelta都对围棋数据上的深层神经网络训练非常有益。在后面的章节中，您将经常使用其中一个作为模型的优化器。

**7.6运行你自己的示例并评估表现**

在第5章、第6章和这一章中，我们向您展示了许多深度学习技术。我们给了你一些作为基线的提示和示例架构，但是现在是时候训练你的自己的模型了。在机器学习实验中，至关重要的是尝试各种超参数组合，如层数、选择哪一层、训练的轮数等等。特别是，有了深度神经网络，你面临的选择数量可能是很大的，并不总是那么清楚如何调整一个特定的参数去影响模型的性能。深度学习研究员可以依靠几十年实验结果和进一步的理论论点拥有一些直觉，但我们不能给你提供这么深层次的知识，不过我们可以帮助你开始建立自己的直觉。

像我们这样的实验装置要取得很好的结果的一个关键因素是尽可能快速地训练一个神经网络去预测围棋的落子。建立模型架构、开始模型训练、观察和评估性能指标所需的时间，然后回去调整模型和重新开始的过程时间必须都要短。当你看到数据科学的挑战，比如kaggle.com上的那些挑战时，往往是那些尝试最多的团队赢得了胜利。你真幸运，keras可以快速建立示例那样。这也是我们选择它作为本书的深度学习框架的主要原因之一。

**7.6.1.测试体系结构和超参数的指南**

让我们看看构建落子预测网络时的一些实际考虑：

卷积神经网络是围棋落子预测网络的一个很好的选择。如果只使用Dense层将导致较低的预测质量。建立一个由几个卷积层和一个或两个Dense层组成的网络通常是必须的。在后面的章节中，您将看到更复杂的体系结构，但是目前，就使用卷积网络。

在你的卷积层中，改变卷积核大小，看看这种变化是如何影响的模型性能。一般来说，2到7之间的卷积核大小是合适的，你不应该比这个大得多。

如果您使用池化层，请确保同时使用max和average池，但更重要的是，不要选择太大的池尺寸。在你的情况下，一个实际的上限可能是（3,3）。您可能还想尝试在没有池化层的情况下构建网络，其计算很耗时间，但可以达到很好的效果。

使用DRopout层进行正则化。在第六章中，您看到了如何使用Dropout来防止模型过度拟合。如果你不使用太多的Drought层，也不把Dropout rate设置得太高，那么你的网络通常会从中受益。

在最后一层使用softmax激活产生概率分布是有好处的，如果再将其与分类交叉熵损失相结合使用，这将非常适合您的用例。

用不同的激活函数进行实验。我们已经给你介绍了ReLU，这应该作为您现在的默认选择，还有Sigmoid激活。您可以在Keras中使用大量其他激活函数，如elu、selu、PReLU和Leaky ReLU。我们可以这里不讨论这些relu变体，但它们的用法在<https://keras.io/activations/>中找到很好的描述

变化的小训练集大小会对模型性能有影响。预测问题，如第五章的预测MNIST手写数字，通常建议选择与类数相同数量级的小训练集大小。对于MNIST，您经常会看到从10到50的训练集大小。如果数据是完全随机的，那么每个梯度都会从各个类接收信息，这使得SGD通常表现得更好。在我们的用例中，一些围棋落子比其他落子更频繁。例如，围棋的四个尖角很少会去下，特别是与星位相比。我们称之为数据中的类不平衡。在这种情况下，你不能指望一个小训练集，所有的类，应该使用从16到256不等的训练集。优化器的选择也会对你的网络有很大的影响，比如有或没有学习率衰减的SGD，以及Adagrad和Adadelta。在h<https://keras.io/optimizers/>下你会发现其他的优化器。

用来训练模型的轮数必须适当地选择。如果您使用模型检查点并跟踪每轮的各种性能指标，这样当训练停止时，你可以有效地测量。在本章的下一节也是最后一节中，我们将简要讨论如何评估性能指标。一般来说，拥有足够的计算能力，设置轮数太高而不是太低。如果模型训练停止改进，甚至会出现过度拟合而变得更糟

**权重初始化**

调整深层神经网络的另一个关键方面是如何在训练开始前初始化权值。因为优化网络意味着在损失表面找到最小损失值所需的权重，因此你开始的权重是很重要的。在第5章的网络实现中，我们随机分配初始权重，这通常是个不好的做法。

权重初始化是一个有趣的研究课题，值得书写一章。keras有很多权重初始化方案，每个有权重的层都可以进行相应的初始化，不过Keras默认选择通常是非常好的，因此不值得费心更改它们。

**7.6.2.评估训练和测试数据的性能指标**

在7.3节中，我们向您展示了在一个小数据集上执行训练运行的结果。我们使用的网络是一个相对较小的卷积网络，然后我们对这个网络进行了五轮的训练，接着我们跟踪训练数据的损失和准确性，并使用测试数据进行验证。最后，我们计算了测试数据的准确性。这就是你应该遵循的一般工作流程，但是你能判断什么时候应该停止训练或检测什么时候就应该关闭训练？以下是一些指导方针：

你的训练准确性和损失通常每过一轮都会提高。后面阶段，这些指标会逐渐减少，有时会有一些波动。如果你好几轮都看不到改善，你可能想停下来。

同时，您应该看看您的测试损失和准确性是什么样子的。在早期，验证损失会持续下降，但后来，经常会开始增加，这样就表示模型已经对训练数据过拟合了。

如果使用模型检查点，请选择高精度低损失的模型。

如果训练和测试损失都很高，请尝试选择更深层次的网络架构或其他超参数。

当你的模型训练误差较低，但验证误差较高，那说明已经出现了过拟合。当您有一个真正大的训练数据集时，通常不会发生此场景。去学习拥有17万围棋对局和数百万个落子的话，问题不是很大。

要选择一个符合硬件要求的训练数据大小。如果一轮的训练需要超过几个小时，那就不是很有趣了。相反，试着在许多中等规模的数据集上找到一个表现良好的模型，然后在尽可能大的数据集上再次训练这个模型。

如果你没有一个好的GPU，你可能想选择在云中训练你的模型。在附录D中，我们将向您展示如何使用AmazonWeb服务（AWS）用GPU训练模型。

当运行比较时，不要停止看起来初始阶段学习比较差的模型，因为有些模型适应的比较慢，后面可能会慢慢赶上，甚至超越。

您可能会问自己，您可以使用本章中介绍的方法构建多强的AI。理论上的上限是这样的：网络永远不会比你提供的数据要强。特别是，使用监督式学习之后，AI不会超过人类。在实践中，如果有足够的计算能力和时间，绝对有可能达到大约2段水平。

为了达到超越人类的游戏表现，你需要使用强化学习技术，在第9章中介绍了这一技术。然后，您可以结合第四章的树搜索、强化学习和监督深度学习，在第13章和第14章中构建更强的机器人。在下一章中，我们将向您展示如何部署一个机器人，并让它通过与人类对手或其他机器人打交道来与其环境进行交互。

**7.7.总结**

无处不在的智能游戏格式（SGF）可用于围棋和其他游戏记录。

可以并行处理围棋数据以获得更快速度和更有效的生成器。

有了强大的接近职业的游戏记录，你就可以建立起预测围棋落子的深度学习模型。

如果你知道你的训练数据的重要属性，你可以显式地将它们编码在特征平面上。然后，模型可以快速学习特征平面与你预测的结果之间的联系。对于围棋机器人，您可以添加一串棋子气的特征平面。

通过使用自适应梯度技术，你可以更有效地进行训练，如Adagrad或Adadelta。随着训练的进展，这些算法对学习速率进行了动态调整。

最终的模型训练可以在一个相对较小的脚本中实现，您可以使用它作为模板来训练你自己的AI。

**Chapter 8. Deploying bots in the wild**

**第8章 部署机器人**

本章涵盖

* 构建一个端到端的应用程序来训练和运行一个围棋机器人
* 在前端运行来对抗你的机器人
* 让你的机器人在本地与其他机器人对抗
* 部署到在线围棋服务器

到目前为止，你已经知道如何为围棋落子预测去构建和训练一个强大的深度学习模型，但是你如何将它集成到一个与对手玩游戏的应用程序中？训练神经网络工作只是构建端到端应用程序的一部分，不管你是自我对弈，还是让你的机器人与其他机器人竞争，这个模型必须集成成一个可以使用的引擎。

在本章中，您将构建一个简单的围棋模型服务器和两个前端。首先，我们为您提供一个HTTP前端，您可以用来对抗您的机器人。然后，我们介绍围棋文本协议（GTP），这是一种广泛使用的协议，围棋AI可以互相进行对抗，比如你可以挑战GNUGo或Pachi，这是两个免费提供的基于GTP的围棋程序。最后，我们将向您展示如何在AmazonWebServices（AWS）上部署围棋机器人并将其与在线围棋服务器（OGS）连接。这样做可以让你的机器人在真正的游戏中与世界各地的其他机器人和人类玩家竞争，获得相应的排名。为了做到这一切，你需要完成以下任务：

* 构建一个围棋落子预测-你在第6章和第7章中训练的神经网络需要集成到一个框架中，允许你在游戏中使用它们。在第8.1节中，我们将按照第3章中得到的随机落子AI的概念创建深度学习机器人
* 提供一个图形界面---作为人类，我们需要这个界面可以方便地对抗围棋机器人。在8.2节中，我们将为您配备一个有趣的届满让你可以与AI进行对抗。
* 把AI部署到云上----如果你的计算机中没有强大的GPU，你就不会得到训练强大的围棋机器人。幸运的是，大多数的云都提供GPU实例，但即使你有足够强大的GPU进行训练，你仍然可能希望把你以前训练过的模型托管在服务器上。在第8.3节中，我们将向您展示如何托管，要了解更多的细节可以去看附录D。
* 与其他AI对弈----人类喜欢使用图形和其他界面，但对于机器人来说，习惯上通过标准化的协议进行通信。在第8.4节中，我们将向您介绍通用围棋文本协议（GTP）。下面两点是重要的组成部分：

1. 与其他机器人对弈-----后你将为你的机器人建立一个GTP前端，让它与8.5节中的其他程序进行对抗。我们将教你如何让你的机器人在本地与另外两个围棋程序进行比赛，去看看你的AI有多好。
2. 在联机围棋服务器上部署机器人-----在第8.6节中，我们将向您展示如何在联机围棋平台上部署机器人，以便让其他机器人可以和你的机器人竞争。这样，您的机器人甚至可以得到排名，所有这些我们将在最后一节展示给您。因为大部分材料都是技术性的，你可以附录E中找到大量细节。

**8.1 创建一个深度学习的落子预测AI**

现在，您已经有了所有的构建块来为围棋数据构建一个强大的神经网络，让我们将这些网络集成到一个为它们服务的代理中。回顾第三章的概念，我们将其定义实现select\_move方法为当前游戏状态选择下一个落子点的类。让我们使用Keras模型和围棋盘Encoder去编写DeepLearningAgent（将此代码放入dlgo中的agent模块中的predict.py中）

from dlgo.agent.base import Agent  
from dlgo.agent.helpers import is\_point\_true\_eye  
from dlgo import goboard  
from dlgo import Encoder  
  
  
class DeepLearningAgent(Agent):  
  
   def \_\_init\_\_(self, model, encoder):  
       super().\_\_init\_\_()  
       self.model = model  
       self.encoder = encoder

接着，您将使用编码器将棋盘撞他转换为特征，然后使用该模型去预测下一步落子。实际上，您将使用该模型去计算所有可能的概率分布。

# 返回整个棋盘预测概率分布  
   def predict(self, game\_state):  
       encoded\_state = self.encoder.encode(game\_state)  
       input\_tensor = np.array([encoded\_state])  
       return self.model.predict(input\_tensor)[0]  
  
   def select\_move(self,game\_state):  
       num\_moves = self.encoder.board\_width\*self.encoder\*board\_height  
       # 获得预测概率分布  
       move\_probs = predict(game\_state)

接下来，您可以稍微改变存储在move\_probs中的概率分布。首先，计算所有值的三次方，以大幅增加可能和不可能落子点之间的距离。你希望最好的落子点能尽可能挑选频繁，然后使用一种叫做裁剪的技巧，它可以防止落子概率太接近0或1。这是通过定义一个极小的正值，ε=0.000001，设置值小于ε到ε，值大于1-ε到1-ε。然后，对得到的值进行规范化，再次得到概率分布

# 大幅拉开可能点与非可能点之间的距离  
       move\_probs = move\_probs \*\* 3  
       small\_data = 1e-6  
       # 防止落子概率卡在0或1  
       move\_probs = np.clip(move\_probs,small\_data,1-small\_data)  
       # 重新得到另一个概率分布  
       move\_probs = move\_probs/sum(small\_data)

您进行此转换是因为您希望根据它们的概率从这个分布采样落子点。（另一个可行的策略不是抽样调查，而是始终采取分布的最大值）。你不采取第二种做法的好处是有时会选择其他可能有用的落子

|  |
| --- |
| # 把概率转成一个排序列表  candidates = np.arange(num\_moves)  # 按照落子概率进行采样，不允许取相同的落子  ranked\_moves = np.random.choice(  candidates, num\_moves, replace=False, p=move\_probs)  # 从最高层开始，找到一个有效的落子，不会减少视野空间  for point\_index in ranked\_moves:  point = self.encoder.decode\_point\_index(point\_index)  if game\_state.is\_valid(goboard.Move.play(point)) and \  not is\_point\_an\_eye(game\_state.board, point, game\_state.current\_player):  return goboard.Move.play(point)  return goboard.Move.pass\_turn() |

你训练了一个深度学习模型，并创建了一个代理，然后你坚持下去。稍后，这个代理将被反序列化成了服务，这样人类玩家或其他机器人就可以对抗它。要执行序列化步骤，就需要序列化Keras的格式。当您持久化Keras模型时，它将以HDF5进行存储，这是一种有效的序列化格式。HDF5文件包含灵活的组，用于存储元信息和数据。对于任何一个Keras模型，您可以调用model.save（“model\_path.h5”）将完整模型（即神经网络体系结构和所有权重）持久化为本地文件model\_path.h5。你唯一需要的就是在保持像这样的Keras模型之前，先安装Python库h5py

要存储完整的代理，您可以添加一个额外的组以获取有关你的围棋棋盘编码器。

|  |
| --- |
| # 序列化模型  def serialize(self, h5file):  # 先创建一个编码器组  h5file.create\_group('encoder')  h5file['encoder'].attrs['name'] = self.encoder.name()  h5file['encoder'].attrs['board\_width'] = self.encoder.board\_width  h5file['encoder'].attrs['board\_height'] = self.encoder.board\_height  # 再创建一个模型组  h5file.create\_group('model')  kerasutil.save\_model\_to\_hdf5\_group(self.model, h5file['model']) |

最后，在序列化模型之后，您还需要知道如何从HDF5文件去加载它。

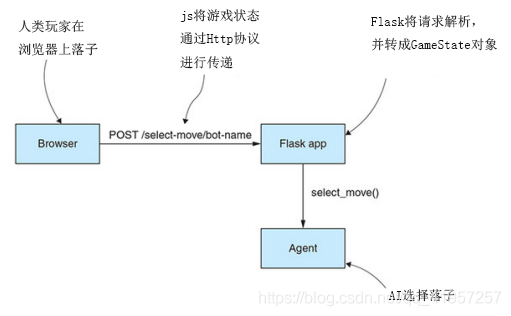
|  |
| --- |
| # 从序列化的文件中加载模型  def load\_prediction\_agent(h5file):  # 加载模型  model = kerasutil.load\_model\_from\_hdf5\_group(h5file['model'])  # 加载编码器  encoder\_name = h5file['encoder'].attrs['name']  if not isinstance(encoder\_name, str):  encoder\_name = encoder\_name.decode('ascii')  board\_width = h5file['encoder'].attrs['board\_width']  board\_height = h5file['encoder'].attrs['board\_height']  encoder = encoders.get\_encoder\_by\_name(  encoder\_name, (board\_width, board\_height))  return DeepLearningAgent(model, encoder) |

这就完成了我们深度学习代理的定义。作为下一步，您必须确保此代理与环境能与人进行交互。通过将DeepLearningAgent嵌入到人类玩家可以在浏览器中使用的Web应用程序中

## 8.2.将你的机器人放到一个网络前端

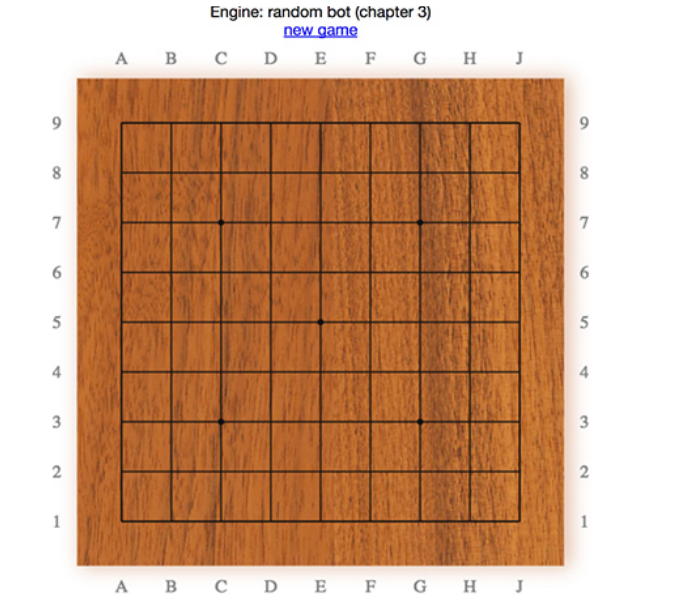
在第6章和第7章中，你设计并训练了一个神经网络，它预测一个人在围棋游戏中会下哪里。在第8.1节中，你将落子预测模型转换为DeepLearningAgent，下一步是与你的机器人对弈！回到第三章，你建立了一个光秃秃的界面，让你可以在控制台输入落子坐标，而你的RandomBot则将其回应打印到控制台。现在你已经建立了一个更复杂的机器人，它应该有一个更好的前端来与人类玩家交流对弈。

在本节中，您将把DeepLearningAgent加入到PythonWeb应用程序，这样您就可以在Web浏览器中与它对局了。您将使用Flask通过HTTP为这样的代理服务。在浏览器方面，您将使用JavaScript中一个名叫jgoboard的库来渲染人类可以使用的围棋棋盘。.8.1提供了将在本章中构建的应用程序的概述。



如果您查看httpfrontend的结构，您会发现一个名为server.py的文件。它有一个单独的、众所周知的方法get\_web\_app，您可以使用该方法返回要运行的web应用程序。下面是如何使用get\_web\_app加载随机机器人并为其提供服务的示例。

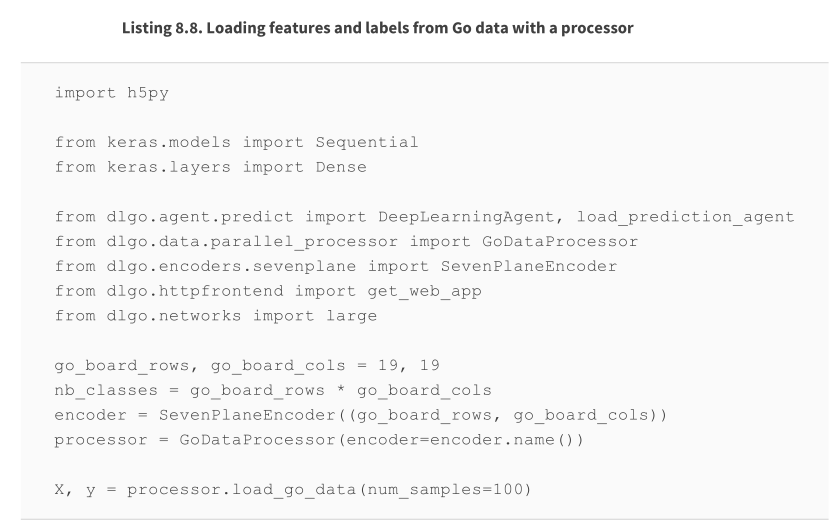
当我们运行这个示例时，一个web应用程序将，监听端口5000，这是Flask应用程序中使用的默认端口。刚刚注册的random的随机机器人所对应于httpfrontend: play\_random\_99.html静态文件夹中有一个HTML文件。在这个文件中，将呈现一个围棋棋盘，它也是定义人类和机器人的游戏规则的地方。人类的对手执黑先行;机器人执白。每当人走了一步棋，/select­move/random 就会被触发，接收机器人的下一步棋。接收到机器人的移动后，它被应用到棋盘上，之后又是人类的移动。要对抗机器人，请在浏览器导航到http://127.0.0.1:5000/static/play\_random\_99.html。应该会看到一个可播放的演示，如下图所示。



你将在接下来的章节中添加越来越多的机器人，但是现在请注意，在play\_predict\_19.html下还有另一个前端，这个web前端与一个叫做predict的机器人交互，可以用来玩19 \* 19的游戏。因此，如果您使用Go数据训练Keras神经网络模型并使用Go board编码器，您可以首先创建一个实例agent = DeepLearningAgent(model, encoder)，然后注册它一个web应用程序web\_app = get\_web\_app({'predict': agent})，您可以用web\_app.run()开始。

### 8.2.1一个端到端的Go机器人示例

下图显示了一个涵盖整个流程的端到端示例(与我们在第7章开头介绍的流程相同)。从需要的导入开始，使用编码器和Go数据处理器将Go数据加载到特性X和标签Y中，加载数据得到过程如下面代码所示





## 只要有了特性和标签，就可以构建一个深度卷积神经网络，并对该数据进行训练，这次从dlgo.networks中选择large network，使用Adadelta作为优化器。

## 8.3.在云上的训练和部署你的AI

直到这一点，所有的开发都是在本地机器上进行的。如果你的电脑上有一个现代化的GPU，那就训练我们开发的深层神经网络第如果你没有强大的GPU，或者不能在它上留出任何计算时间，那么去租用云使用GPU计算通常是一个很好的选择。在第8.2节中，通过Web应用程序运行机器人由本地主机托管。如果你想和朋友分享你的机器人，或者公开它，这并不完全理想。你既不想确保你的电脑日夜运转，也不想让公众知道去你的机器。通过在云中托管您的机器人，让你可以将开发与部署分开，并且可以简单地与任何对与你的AI对弈感兴趣的人共享URL。

因为这个话题很重要我们把它完全放在了附录D。在附录D中，您将了解如何开始使用一个特定的云提供商AmazonWeb服务（AWS）。您将在附录中学习以下技能：

* 使用AWS创建帐户
* 灵活设置、运行和终止虚拟服务器实例创建一个AWS实例
* 在费用合理的情况下创建适合在云GPU上进行深度学习的模型训练
* 在一个（几乎）免费的服务器上放置你的AI

通过HTTP除了学习这些有用的技能外，附录D也展示了部署一个完整的Go机器人的先决条件，该机器人可以连接到联机Go服务器

## 8.4 与其他AI对弈：使用GTP

在8.2节中，您看到了如何将AI框架集成到Web前端。为此，您使用HTTP（运行Web的核心协议之一）处理了机器人和人类棋手之间的通信。为了避免分心，我们有目的地忽略了所有细节，但有一个标准化的协议是必要的，以实现这一点。人类和机器人不能使用一种通用语言来交换围棋落子，但是协议可以充当桥梁。

GTP(围棋文本协议)是世界各地围棋服务器用来连接人类棋手和他们平台上的机器人。许多离线Go程序也是基于GTP的。本节通过示例向您介绍GTP；您将在Python中实现协议的一部分并使用此实现来让你的机器人与其他AI进行对弈。

在附录C中，我们解释了如何下载GNUGO和Pachi，这两个通用的围棋程序几乎适用于所有的操作系统。我们建议两者都安装，所以请确保在您的系统上有两个程序。你不需要任何前端，只需要简单的命令行工具。如果安装了GNU Go，您可以GTP模式启动它：

gnugo ­­--mode gtp

使用此模式，您现在可以探索GTP是如何工作的。GTP是一种基于文本的协议，因此您可以在终端中键入命令并按回车键。例如，要设置9×9的棋盘，可以键入boardsize 9。这将触发GNU Go返回响应并确认命令已正确执行。每一个成功的GTP命令都会触发一个以=开头的响应，而失败的命令会导致一个？要检查当前的棋盘状态，您可以发出showboard命令，它将按预期打印出一个空的9×9棋盘。

在实际的游戏中，两个命令是最重要的：genmove和play..第一个命令genmove用于请求GTP bot去生成下一个落子。GTP机器人通常也会应用这个落子到它内部的游戏状态。所有这些命令都需要参数是玩家的棋子颜色，无论是黑色还是白色。例如，要生成一个白色落子并将其放置在GNU Go的棋盘上，请键入genmove white。这将导致响应，如=C4，这意味着GNUGo接受此命令（=），并将白石放置在C4。如您所见，GTP接受第2章和第2章所介绍的标准坐标

另一个与游戏相关的动作是play。此命令用于让GTP机器人知道它必须在棋盘上落子。例如，你通过键入play black D4来让它将黑棋下在D4上，它将返回一个=来确认此命令。当两个机器人互相对抗时，他们会轮流要求对方落出下一步，然后将选择的落子放在棋盘上。这一切都很直截了当----但我们遗漏了许多细节。一个完整的GTP客户端有更多的命令要处理，从处理让子棋到处理时间设置和计数规则。如果您对GTP的细节感兴趣，请参阅http://mng.bz/MWNQ.。尽管如此，在一个基本的层次上，genmove和play将足以让我们的深度学习机器人对抗GNUGo和Pachi。

要处理GTP并包装您的Agent概念，以便它可以通过使用此协议交换围棋落子，您可以创建一个名为GTP的新dlgo模块,我们建议直接跟踪我们在http://mng.bz/a4Wj的GitHub上的实现。

首先，让我们形式化GTP命令。要做到这一点，我们必须注意，在许多围棋服务器上，命令需要获得序列号，以确保我们能够匹配命令和响应。这些序号数字是可选的，可以是None的。对我们来说，GTP命令由序列号、命令和该命令的潜在多个参数组成，这个命令在gtp模块下的command.py里找到。 ,

|  |
| --- |
| class Command:  # GTP命令由序列号、命令和该命令的潜在多个参数组成  def \_\_init\_\_(self, sequence, name, args):  self.sequence = sequence  self.name = name  self.args = tuple(args)    # 判断命令是否相同  def \_\_eq\_\_(self, other):  return self.sequence == other.sequence and \  self.name == other.name and \  self.args == other.args    def \_\_repr\_\_(self):  return 'Command(%r, %r, %r)' % (self.sequence, self.name, self.args)    def \_\_str\_\_(self):  return repr(self) |

接下来，要将从命令行输入的文本解析为Command。例如，解析“999playwhiteD4”应该导致命令（999，‘play’，（‘white’，‘D4’）。parse就是实现了这种功能。

|  |
| --- |
| # 根据命令字符串解析命令  def parse(command\_string):  pieces = command\_string.split()  try:  sequence = int(pieces[0]) # 序号  pieces = pieces[1:]  except ValueError: # 如果第一个不是数字，就没有序列号  sequence = None  name, args = pieces[0], pieces[1:]  return Command(sequence, name, args) |

|  |
| --- |
| from dlgo.gotypes import Point  from dlgo.agent.FastRandomAgent.goboard\_fast import Move    COLS = 'ABCDEFGHJKLMNOPQRST'    # 将落子转成gtp棋盘位置  def coords\_to\_gtp\_position(move):  point = move.point  return COLS[point.col - 1] + str(point.row)    # 将gtp棋局位置转成坐标  def gtp\_position\_to\_coords(gtp\_position):  col\_str, row\_str = gtp\_position[0], gtp\_position[1:]  point = Point(int(row\_str), COLS.find(col\_str.upper()) + 1)  return Move(point) |

我们刚才认为GTP坐标是标准表示法，所以将GTP坐标解析成棋盘位置，反之也是简单的。定义两个辅助函数使用gtp在坐标和位置之间转变

## 8.5 和本地其他机器人进行对弈

现在您了解了GTP的基本知识，让我们深入到一个应用程序中，并构建一个程序来加载您的一个机器人，并让它与GNUGo或Pachi竞争。在我们开始程序之前，我们只有一个技术问题要解决---什么时候让我们的机器人投降或pass

### 8.5.1 当一个机器人应该投降或pass

到目前为止，你的深度学习机器人没有办法知道什么时候停止下棋。你至今设计它们的方式是，你的机器人总是会选择最好的落子来下。这个直到对局结束才停止是有害的，当情况看起来有点太糟糕的时候，选择pass甚至投降可能会更好。出于这个原因，您将实施终止策略：您将告诉机器人什么时候停下来。在第13章和第14章中，您将学习强大的技术，这将使这完全无用（您的机器人将学会判断当前的棋局，从而判断是否投降）。但目前，这个概念是有用的，并将帮助您试验这种方式部署一个机器人对抗其他对手。

调用dlgo下的agent下的termination.py的TerminationStrategy 。它所做的只是决定你什么时候应该pass投降----默认情况下，你永远不会pass或投降。

|  |
| --- |
| from dlgo import goboard  from dlgo.agent.base import Agent  from dlgo.ComputeWinner import compute\_game\_result      class TerminationStrategy:  def \_\_init\_\_(self):  pass    def should\_pass(self,game\_state):  return False    def should\_resign(self,game\_state):  return False |

停止游戏的一个简单的启发是当你的对手pass时pass。你必须依靠这样一个事实，即你的对手知道什么时候要pass，但这是一个开始，而且它在对抗GNU Go和Pachi。

|  |
| --- |
| class PassWhenOpponentPasses(TerminationStrategy):    def should\_pass(self, game\_state):  if game\_state.last\_move is not None:  return True if game\_state.last\_move.is\_pass else False    def get(termination):  if termination == 'opponent\_passes':  return PassWhenOpponentPasses()  else:  raise ValueError("Unsupported termination strategy: {}"  .format(termination)) |

在termination.py中，你还发现了另一种叫做ResignLargeMargin的策略，它每当游戏的估计分数对手过于有利时就会投降。你还能做许多别的策略，但要记住，最终你可以通过机器学习摆脱这种东西。

|  |
| --- |
| # 过于有利于对手就resign  class ResignLargeMargin(TerminationStrategy):    def \_\_init\_\_(self, own\_color, cut\_off\_move, margin):  TerminationStrategy.\_\_init\_\_(self)  self.own\_color = own\_color  self.cut\_off\_move = cut\_off\_move  self.margin = margin    self.moves\_played = 0    def should\_pass(self, game\_state):  return False    def should\_resign(self, game\_state):  self.moves\_played += 1  if self.moves\_played:  game\_result = scoring.compute\_game\_result(self)  if game\_result.winner != self.own\_color and game\_result.winning\_margin >= self.margin:  return True  return False |

最后一件事，为了让机器人相互对抗，你最不需要的就是给代理配备一个终止策略，以便在适当的时候pass或投降。

|  |
| --- |
| class TerminationAgent(Agent):    def \_\_init\_\_(self, agent, strategy=None):  Agent.\_\_init\_\_(self)  self.agent = agent  self.strategy = strategy if strategy is not None \  else TerminationStrategy()    def select\_move(self, game\_state):  if self.strategy.should\_pass(game\_state):  return goboard.Move.pass\_turn()  elif self.strategy.should\_resign(game\_state):  return goboard.Move.resign()  else:  return self.agent.select\_move(game\_state) |

### 8.5.2 让你的围棋AI与其他围棋程序对抗

在讨论了终止策略之后，您现在可以让围棋机器人与其他程序对局。在gtp模块中的play\_local.py下，找到一个在你的某个机器人与无论是GNUGo还是Pachi之间设置游戏的脚本。从必要的导入开始，一步一步地完成这个脚本

|  |
| --- |
| from \_\_future\_\_ import print\_function  import subprocess  import re  import h5py    from dlgo.agent.DeepLearningAgent.DeepLearningAgent import load\_prediction\_agent  from dlgo.agent.DeepLearningAgent.termination import PassWhenOpponentPasses, TerminationAgent  from dlgo.agent.FastRandomAgent.goboard\_fast import GameState, Move  from dlgo.gotypes import Player  from dlgo.gtp.board import gtp\_position\_to\_coords, coords\_to\_gtp\_position  from dlgo.utils import print\_board  from dlgo.ComputeWinner import compute\_game\_result |

要初始化LocalGtpBot，您需要提供深度学习代理和可选的终止策略。此外，您还可以指定应该让几个子，以及要对弈的对手。对于后者，你可以在gnugo和pachi之间进行选择..LocalGtpBot将这些程序中的任何一个初始化为子进程，然后您的AI和它的对手将通过GTP进行通信。

|  |
| --- |
| # tag::play\_local\_init[]  class LocalGtpBot:    def \_\_init\_\_(self, go\_bot, termination=None, handicap=0,  opponent='gnugo',our\_color='b'):  self.bot = TerminationAgent(go\_bot, termination) # 用代理和终止策略初始化机器人  self.handicap = handicap  self.\_stopped = False  self.game\_state = GameState.new\_game(19)    self.our\_color = Player.black if our\_color == 'b' else Player.white  self.their\_color = self.our\_color.other    cmd = self.opponent\_cmd(opponent) #你的对手将是GNU Go或Pachi  # 开辟子进程  pipe = subprocess.PIPE  self.gtp\_stream = subprocess.Popen(  cmd, stdin=pipe, stdout=pipe #从命令行读写GTP命令  )    @staticmethod  def opponent\_cmd(opponent):  if opponent == 'gnugo':  return ["gnugo", "--mode", "gtp"]  elif opponent == 'pachi':  return ["pachi"]  else:  raise ValueError("Unknown bot name {}".format(opponent)) |

我们在这里演示的工具是command\_and\_response，它可以发出GTP命令并读取该命令的响应

|  |
| --- |
| # 发送命令  def send\_command(self, cmd):  self.gtp\_stream.stdin.write(cmd.encode('utf-8'))    # 得到回应  def get\_response(self):  succeeded = False  result = ''  while not succeeded:  line = self.gtp\_stream.stdout.readline()  if line[0] == '=':  succeeded = True  line = line.strip()  result = re.sub('^= ?', '', line)  return result    def command\_and\_response(self, cmd):  self.send\_command(cmd)  return self.get\_response() |

下棋的效果如下：

1. 用GTP棋盘大小命令去设置棋盘。你在这里只允许19×19围棋棋盘，因为你的深度学习机器人是为此量身定做的。
2. 在set\_handicap方法中正确设置让子。
3. 对弈本身，你将在play方法中涵盖它。

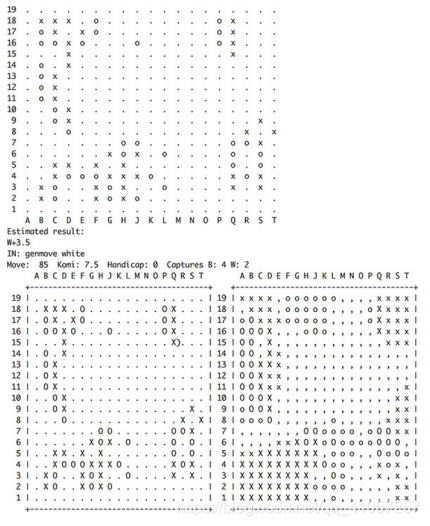
|  |
| --- |
| # 运行  def run(self):  self.command\_and\_response("boardsize 19\n")  self.set\_handicap()  self.play()    # 设置让子  def set\_handicap(self):  if self.handicap == 0:  self.command\_and\_response("komi 7.5\n")  else:  stones = self.command\_and\_response("fixed\_handicap {}\n".format(self.handicap))  for pos in stones.split(" "):  move = gtp\_position\_to\_coords(pos)  self.game\_state = self.game\_state.apply\_move(move) |

你的机器人对弈的游戏逻辑很简单：没有一个对手停下来，轮流并持续落子。机器人在名为play\_our\_move和play\_their\_move的方法中做到了这一点。您还清除屏幕，并打印出当前的棋局和对结果的粗略估计。

|  |
| --- |
| def play(self):  while not self.\_stopped:  if self.game\_state.current\_player == self.our\_color:  self.play\_our\_move()  else:  self.play\_their\_move()  print(chr(27) + "[2J")  print\_board(self.game\_state.board)  print("Estimated result: ")  print(compute\_game\_result(self.game\_state))  # 我方落子  def play\_our\_move(self):  move = self.bot.select\_move(self.game\_state)  self.game\_state = self.game\_state.apply\_move(move)    our\_name = self.our\_color.name    if move.is\_pass:  self.command\_and\_response("play {} pass\n".format(our\_name))  elif move.is\_resign:  self.command\_and\_response("play {} resign\n".format(our\_name))  else:  pos = coords\_to\_gtp\_position(move)  self.command\_and\_response("play {} {}\n".format(our\_name, pos))      # 对方落子  def play\_their\_move(self):  their\_name = self.their\_color.name  pos = self.command\_and\_response("genmove {}\n".format(their\_name))  if pos.lower() == 'resign':  self.game\_state = self.game\_state.apply\_move(Move.resign())  self.\_stopped = True  elif pos.lower() == 'pass':  self.game\_state = self.game\_state.apply\_move(Move.pass\_turn())    if self.game\_state.last\_move.is\_pass:  self.\_stopped = True  else:  move = gtp\_position\_to\_coords(pos)  self.game\_state = self.game\_state.apply\_move(move) |

然后就可以使用你之前训练的AI进行对弈了

|  |
| --- |
| from dlgo.gtp.play\_gtp\_local import LocalGtpBot  from dlgo.agent.DeepLearningAgent.DeepLearningAgent import load\_prediction\_agent  from dlgo.agent.DeepLearningAgent.termination import PassWhenOpponentPasses    my\_bot = load\_prediction\_agent("deep\_go.h5")  gnu\_go = LocalGtpBot(go\_bot=my\_bot, termination=PassWhenOpponentPasses(),  handicap=0, opponent='pachi', )  gnu\_go.run() |



在图的顶部，你看到你打印的棋盘，然后是你目前的估计。在下半变，你可以在左边看到Pacho的比赛状态（与你的状态相同），在右边Pachi给你一个关于游戏的评估，它用于判断棋盘上的点属于哪一方。

这是一个希望令人信服和激动人心的演示.，但这还不是故事的结尾。

## 8.6.部署一个GO机器人到一个在线GO服务器

我们发现play\_local.py实际上是一个小型的围棋服务器，供两个机器人对手相互竞争。它接收并发送GTP命令，并知道何时开始和结束游戏。这就产生了开销，因为程序扮演了裁判的角色，控制着对手之间的互动。

如果你想把一个机器人连接到一个真正的Go服务器上，这个服务器会处理所有的游戏逻辑，而你可以完全专注于发送和接收GTP命令。另一方面，连接到合适的Go服务器意味着您必须确保该服务器支持所有GTP命令，否则您的机器人可能会崩溃。

为了确保不会发生这种情况，让我们对GTP命令的处理进行更多的规范化。首先，为成功和失败的命令实现一个适当的GTP响应类。

|  |
| --- |
| class Response:      def \_\_init\_\_(self, status, body):          self.success = status          self.body = body  #利用响应体响应成功的GTP命令  def success(body=''):  return Response(status=True, body=body)  #响应失败的GTP命令  def error(body=''):  return Response(status=False, body=body)  #将Python布尔值转换为GTP  def bool\_response(boolean):  return success('true') if boolean is True else success('false')  # 将GTP响应序列化为字符串  def serialize(gtp\_command, gtp\_response):      return '{}{} {}\n\n'.format(          '=' if gtp\_response.success else '?',          '' if gtp\_command.sequence is None else str(gtp\_command.sequence),          gtp\_response.body      ) |

这就剩下实现本节的主类GTPFrontend了。您将这个类放入gtp模块的frontend.py中，需要导入以下内容，包括来自gtp模块的命令和响应。

|  |
| --- |
| import sys  from dlgo.gtp import command, response  from dlgo.gtp.board import gtp\_position\_to\_coords, coords\_to\_gtp\_position  from dlgo.goboard\_fast import GameState, Move  from dlgo.agent.termination import TerminationAgent  from dlgo.utils import print\_board |

要初始化GTP前端，您需要指定一个代理实例和一个可选的终止策略。然后，GTPFrontend将实例化您处理的GTP事件字典。这些事件(包括play等常见命令)都必须由您实现。

|  |
| --- |
| HANDICAP\_STONES = {      2: ['D4', 'Q16'],      3: ['D4', 'Q16', 'D16'],      4: ['D4', 'Q16', 'D16', 'Q4'],      5: ['D4', 'Q16', 'D16', 'Q4', 'K10'],      6: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10'],      7: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K10'],      8: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K4', 'K16'],      9: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K4', 'K16', 'K10'],  }  class GTPFrontend:      def \_\_init\_\_(self, termination\_agent, termination=None):          self.agent = termination\_agent          self.game\_state = GameState.new\_game(19)          self.\_input = sys.stdin          self.\_output = sys.stdout          self.\_stopped = False          self.handlers = {              'boardsize': self.handle\_boardsize,              'clear\_board': self.handle\_clear\_board,              'fixed\_handicap': self.handle\_fixed\_handicap,              'genmove': self.handle\_genmove,              'known\_command': self.handle\_known\_command,              'komi': self.ignore,              'showboard': self.handle\_showboard,              'time\_settings': self.ignore,              'time\_left': self.ignore,              'play': self.handle\_play,              'protocol\_version': self.handle\_protocol\_version,              'quit': self.handle\_quit,          } |

在使用以下run方法启动游戏之后，将持续读取转发到相应事件处理程序的GTP命令，这是由process方法完成的。

|  |
| --- |
| def run(self):          while not self.\_stopped:              input\_line = self.\_input.readline().strip()              cmd = command.parse(input\_line)              resp = self.process(cmd)              self.\_output.write(response.serialize(cmd, resp))              self.\_output.flush()  def process(self, cmd):          handler = self.handlers.get(cmd.name, self.handle\_unknown)          return handler(\*cmd.args) |

完成这个GTPFrontend剩下的工作是实现各个GTP命令。下面的清单显示的是三种最重要的;其余内容请参考GitHub存储库。

|  |
| --- |
| def handle\_play(self, color, move):          if move.lower() == 'pass':              self.game\_state = self.game\_state.apply\_move(Move.pass\_turn())          elif move.lower() == 'resign':              self.game\_state = self.game\_state.apply\_move(Move.resign())          else:              self.game\_state =       self.game\_state.apply\_move(gtp\_position\_to\_coords(move))          return response.success()      def handle\_genmove(self, color):          move = self.agent.select\_move(self.game\_state)          self.game\_state = self.game\_state.apply\_move(move)          if move.is\_pass:              return response.success('pass')          if move.is\_resign:              return response.success('resign')          return response.success(coords\_to\_gtp\_position(move))      def handle\_fixed\_handicap(self, nstones):          nstones = int(nstones)          for stone in HANDICAP\_STONES[nstones]:              self.game\_state = self.game\_state.apply\_move(                  gtp\_position\_to\_coords(stone))          return response.success() |

您现在可以在一个小脚本中使用这个GTP前端从命令行启动它。

|  |
| --- |
| from dlgo.gtp import GTPFrontend  from dlgo.agent.predict import load\_prediction\_agent  from dlgo.agent import termination  import h5py  model\_file = h5py.File("agents/betago.hdf5", "r")  agent = load\_prediction\_agent(model\_file)  strategy = termination.get("opponent\_passes")  termination\_agent = termination.TerminationAgent(agent, strategy)  frontend = GTPFrontend(termination\_agent)  frontend.run() |

在这个程序运行之后，您可以使用它，就像您在8.4节中测试GNU Go一样:您可以向它抛出GTP命令，它将正确地处理它们。继续并通过使用genmove生成一个移动或使用showboard打印出板状态来测试它。GTPFrontend事件处理程序中涉及的任何命令都是可行的。

### 8.6.1.在线上Go服务器上注册一个机器人

现在你的GTP前端已经完成，并且以与GNU Go和Pachi相同的方式在本地工作，你可以在一个使用GTP进行通信的在线平台上注册你的机器人。您会发现大多数流行的Go服务器都基于GTP，而附录C明确地涵盖了其中的三个。在欧洲和北美最流行的服务器之一是在线Go服务器(OGS)。我们选择了OGS作为平台来展示如何运行机器人，但是您也可以在大多数其他平台上做同样的事情。

因为你的机器人在OGS的注册过程有点复杂，而且连接你的机器人到OGS的软件是用JavaScript编写的，所以我们把这一部分放在附录E中。您可以现在阅读这个附录，然后回到这里，或者跳过它，如果您对运行自己的在线机器人不感兴趣。当你完成附录E，你将学习以下技能:

* 在OGS上创建两个帐户，一个用于您的bot，另一个用于管理您的bot帐户。
* 为了测试目的，将您的机器人从本地计算机连接到OGS
* 在AWS实例上部署机器人以连接到OGS，时间不限

这将允许你进入一个(排名)游戏与你自己的创造围棋AI同时在线。同时，每个拥有OGS帐号的玩家都可以和你的机器人对弈了，这是一件让人兴奋的事情。除此之外，你的机器人还可以参加OGS主办的比赛!

# 8.7.概要

* 通过在代理框架中构建深度网络，您可以使您的模型能够与它们的环境交互。
* 在web应用程序中注册代理，通过构建HTTP前端，您可以通过图形界面与自己的机器人进行较量。
* 使用像AWS这样的云提供商，你可以在GPU上租借计算能力来高效地运行你的深度学习实验。
* 在AWS上部署您的web应用程序，您可以轻松地共享您的机器人并让它与其他人一起运行。
* 通过让您的机器人发出和接收Go文本协议(GTP)命令，它可以以标准化的方式在本地与其他Go程序进行对抗。
* 为你的机器人建立一个GTP前端是在一个在线Go平台上注册它的最重要的一步。
* 在云中部署一个机器人，您可以让它参加在线围棋服务器(OGS)上的常规游戏和锦标赛，并在任何时候与它进行较量。

# GNU Go

GNU Go是在1989年开发的，是至今仍在使用的最古老的围棋引擎之一。最近一次发布是在2009年。尽管最近没有什么进展，但GNU Go仍然是一个受欢迎的AI 对手，在许多围棋o服务器上为初学者服务。此外，它是基于人工制作规则的最强的围棋引擎之一；这与MCTS和深度学习机器人形成了很好的对比。你可以从www.gnu.org/software/gnugo/download.html下载安装GNUGo。此页面包括安装GNUGo作为命令行接口（CLI）工具的说明和各种图形界面。为了要安装CLI工具，您需要从http://ftp.gnu.org/gnu/gnugo/下载最近版本的AI二进制文件，解压缩各自的tarball，并遵循下载中包含的INSTALL和README文件中的平台说明。对于图形界面，我们建议从www.rene­grothmann.de/jago/下载Windows和Linux版本的JagoClient和从http://sente.ch/software/goban/freegoban.html.下载MacOS版本的FreeGoban

# Pachi

你可以在http://pachi.or.cz/.上找到一个比GNU Go整体要强得多的程序Pachi,此外，Pachi的源代码和详细的安装说明可以在https://github.com/pasky/pachi中找到.要测试Pachi，请在命令行上运行Pachi，并键入genmove Black，让它在9×9棋盘上为您生成一个黑色落子。