

# Deep Learning with PyTorch

Eli Stevens  
Luca Antiga

MANNING



Deep Learning with PyTorch

by Eli Stevens and Luca Antiga

ISBN 9781617295263

400 pages (estimated)

\$49.99

Publication in Winter, 2019 (estimated)

# 目录

1 介绍深度学习和 PyTorch 库 1

    1.1 什么是 PyTorch? 2

    1.2 这本书是什么? 2

    1.3 为什么使用 PyTorch ? 3

    1.4 PyTorch 拥有内置的电池 10

2 从张量开始 15

    2.1 张量基础 18

    2.2 张量和存储 22

    2.3 尺寸、存储偏移量和步长 24

    2.4 数值类型 30

    2.5 索引张量 31

    2.6 NumPy 的互操作性 31

    2.7 序列化张量 32

**2.8 将张量移动到 GPU** 34

**2.9 张量的 API** 35

**3 使用张量对真实世界数据的表示** 39

**3.1 表格式数据** 40

**3.2 时间序列** 49

**3.3 文本** 54

**3.4 图像** 60

**3.5 体积数据** 63

**4 学习的机理** 67

**4.1 学习就是参数估计** 70

**4.2 PyTorch 的 autograd: 将一切都后向传播** 83

**5 使用神经网络来拟合你的数据** 101

**5.1 人工神经元** 102

**5.2 PyTorch 的 nn 模块** 110

**5.3 继承 nn.Module 类** 120

**索引** 127



# 关于作者

Eli Stevens 过去作为一名软件工程师在硅谷工作了 15 年，并在过去的 7 年中在一家生产医疗设备软件的初创公司中担任首席技术官。

Luca Antiga 是位于意大利贝加莫的一家人工智能工程公司的联合创始人兼首席执行官，同时也是 PyTorch 的定期撰稿人。

# 1 介绍深度学习和 PyTorch 库

本章覆盖内容有：

- 这本书将会教你什么
- PyTorch 库用于构建深度学习项目的作用
- PyTorch 的优点和缺点

我们正生活在一个激动人心的时代。我们面临的景象是计算机能做的事情是每周都有变化。几年前还认为需要更高的认知能力来解决的任务，现在正在被机器以接近超人类的性能水平解决。许多任务，诸如用一句英语俗语来描述一幅照片图像、玩复杂的策略游戏、从放射性扫描中诊断肿瘤等等任务现在都可以通过电脑来实现。更令人瞩目的是，计算机通过实例获得解决这类任务的能力，而不需要通过人来手工编写代码来制定规则。

如果声称机器正在以人类意义来学习“思考”单词，那是不真实的。相反，一类通用的算法本质是能够极其有效的复杂的非线性过程。在某种程度上，我们正在学习我们主观感受到的智能，是一个经常被混为一谈的概念——自我意识，而自我意识绝对不是解决或进行这些问题的必要条件。最后，计算机智能化的问题可能甚至都不重要。正如计算机科学家先驱 Edsger W.Dijkstra 在《计算科学的威胁》中说的那样：

Alan M. Turing 在思索...机器是否会思考，这个问题...和潜艇是否会游泳的问题一样。

我们所说的这一类通用算法属于深度学习的范畴，它处理的是对名为深度神经网络的数学实体在实例的基础上进行训练。深度学习利用大量的数据来拟合复杂函数，这类函数的输入和输出通常相差很大。比如 输入是图像，输出是一行描述输入的文字；一个书面脚本（输入）和一个自然音的朗诵脚本（输出）；或者，更简单的说，将金毛犬的图像打上带有表示金毛犬存在的标记。此功能允许开发人员创建程序，使程序具有直到现在仍然是人类专属领域的功能。

## 1.1 什么是 PyTorch?

PyTorch 是一个 Python 程序库，以方便构建深度学习项目。它强调灵活性，允许深度学习模型用符合 Python 的习惯来使用。这种已接近性和易于使用的特点，让早期采用者广泛用于研究界，而在程序库发行后的几年里，它已经发展成为一个最杰出的深度学习工具，并且应用范围广泛。

PyTorch 提供了一个核心的数据结构——Tensor，这是一个多维数组，它与 NumPy 数组有许多相似之处。在此基础上，一系列的功能的建立可以方便一个项目的启动和运行，或者使得对一种新的神经网络架构的研究设计和训练变得容易。Tensors 可以加快数学操作（假设硬件和软件的适当组合存在），PyTorch 有用于分布式训练的包、用于高效数据加载的工人进程的，以及大量的常用深度学习函数库。

正如 Python 用于编程一样，PyTorch 既是深度学习的出色入门介绍，也是在专业环境中用于现实世界中高水平工作的有用工具。

我们相信 PyTorch 应该是您学习的第一个深度学习库，至于是否是最后一个深度学习库，那就留给你自己来做决定。

## 1.2 这本书是什么？

本书旨在作为软件工程师、数据科学家和精通 Python 并愿意使用 PyTorch 来构建深度学习项目的积极进取的学生的起点。为此，我们采取了动手的方法。我们鼓励您随时准备使用计算机，以便您可以使用示例并进一步进行操作。

尽管我们强调实际应用，但我们也相信，提供对基础深度学习工具（如 PyTorch）的容易理解的介绍，也是促进获得新技术技能的一种方式。这也是朝着为各学科的新一代科学家、工程师和从业人员配备工具的实用知识迈出的一步，这些工具将成为未来几十年许多软件项目的基础。

要充分利用本书，您需要做两件事：

- 有一些使用 Python 进行编程的经验——我们将不会对此有过多要求：您只需要掌握 Python 数据类型、类、浮点数等。
- 愿意投入进去，亲自动手——如果您愿意跟随我们，那么学习起来会容易得多。

深度学习是一个巨大的空间。在本书中，我们将覆盖该空间的一小部分，特别是将 PyTorch 用于较小范围的项目。大多数激励示例使用 2D 和 3D 数据集的图像处理。我们专注于实用的 PyTorch，旨在覆盖足够的基础，使您可以通过深度学习解决现实问题，或者探索研究文献中涌现的新模型。有关深度学习研究的最新出版物的一个重要资源是 ArXiV 公共预印本存储库，托管在 <https://arxiv.org> 上。

## 1.3 为什么使用 PyTorch ?

就像我们已经说过的那样，通过将示例展示给模型，深度学习使您可以执行各种复杂的任务，例如执行机器翻译、玩策略游戏以及识别混乱场景中的对象。为此，在实践中，您需要灵活的工具，以便可以针对您的特定问题进行调整，并且效率很高，以便在合理的时间内对大量数据进行训练。您还需要训练过的网络在输入中存在不确定性的情况下正确执行操作。在本节中，我们介绍了我们决定使用 PyTorch 的一些原因。

PyTorch 简单易行，值得推荐。许多研究人员和从业人员发现，它易于学习、使用、扩展和调试。它是 Python 风格的，尽管（像其他任何复杂的领域一样）它有一些警告和最佳实践，但是对于以前使用过 Python 的开发人员来说，使用该库通常会感到很熟悉。

对于熟悉 NumPy 数组的用户，很快就能熟悉 PyTorch 的 Tensor 类。PyTorch 感觉像 NumPy，但是具有 GPU 加速和自动计算梯度的功能，这使其适合从前向表达式开始自动计算后向传递的数据。

Tensor API 使得与深度学习相关的类的附加功能不再困难；用户大多可以假装这些功能不存在，直到需要它们为止。

PyTorch 的设计驱动力是表现力，允许开发人员实现复杂的模型，而不会因库而造成不必要的复杂性。（该库不是框架！）在深度学习领域，PyTorch 可以说是将想法最无缝地翻译成 Python 代码之一。因此，PyTorch 在研究中得到了广泛采用，国际会议上的高引用率就证明了这一点。

PyTorch 也有一个引人入胜的故事，说明了从研发到生产的过渡。尽管 PyTorch 最初专注于研究工作流程，但已配备了高性能 C ++ 运行时，用户可以在不依赖 Python 的情况下利用它来部署用于推理的模型，从而保留了 PyTorch 的大部分灵活性，而无需付出 Python 运行时的开销。

当然，单单做出易用性和高性能的声称是不值一提的。我们希望，当您精读本书时，您将同意我们在这些声称是有充分根据的。

### 1.3.1 深度学习革命

在本节中，我们将退后一步，并提供一些背景说明 PyTorch 适合深度学习工具的当前和历史前景。

直到 2000 年代后期，属于“机器学习”类别的更广泛的系统类别严重依赖于特征工程。特征是对输入数据的转换，产生的数字特征有助于下游算法（例如分类器）对新数据产生正确的结果。特征工程的目的是获取原始数据，并提出可以提供给算法以解决问题的相同数据

的表示形式。例如，要在手写数字图像中将数字 1 从数字 0 中分辨出来，您需要使用一组过滤器来估计图像上边缘的方向，然后训练分类器以预测正确的数字（给定边缘方向的分布）。另一个有用的特征可能是 0 或 8 的环状的两个封闭孔的数量。

另一方面，深度学习负责从原始数据中自动查找此类表示形式，以成功执行任务。在 1 对 0 的示例中，可以在训练期间通过迭代查看示例和目标标签对来优化过滤器。这并不是说特征工程在深度学习中没有地位；开发人员经常需要向学习系统中注入某种形式的知识。然而，神经网络能够根据示例消化数据并提取有用表示的能力使深度学习变得如此强大。深度学习从业人员的重点不是手工制作这些表示，而是在数学实体上操作，以便它可以从训练数据中自动发现表示。通常，这些自动创建的特征比手工制作的特征要好！与许多破坏性技术一样，这一事实导致了观点的改变。

在图 1.1 的左侧，从业者正忙于定义工程特征并将其提供给学习算法。任务的结果将与他设计的特征一样好。在图的右侧，通过深度学习，原始数据被馈送到基于优化任务性能的算法上，自动提取分层特征。结果将与实践者将算法推向目标的能力一样好。

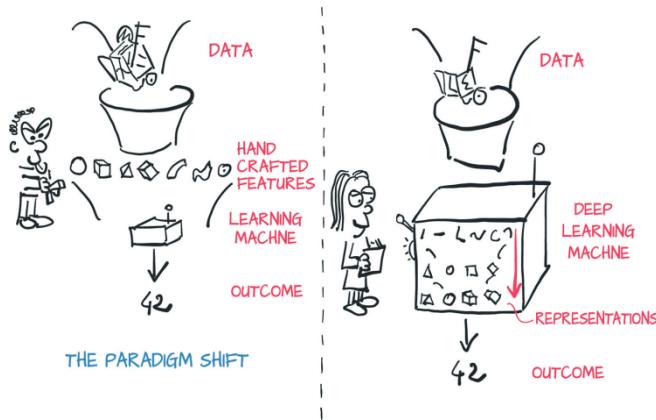


图 1.1 深度学习带来的观点上的改变

### 1.3.2 直接和推迟执行

深度学习库的主要分类标准之一是立即执行与延迟执行。PyTorch 的易用性很大程度上归功于它实现即时执行的方式，因此我们在此简要介绍该实现。

考虑实现毕达哥拉斯定理的表达式  $(a ** 2 + b ** 2) ** 0.5$ 。如果要执行此表达式，则需要方便地使用  $a$  和  $b$ ，如下所示：

```
>>> a = 3
>>> b = 4
>>> c = (a**2 + b**2) ** 0.5
>>> c
5.0
```

这样的立即执行获取输入并产生输出值（此处为  $c$ ）。PyTorch 与一般的 Python 一样，

默认情况下为立即执行（在 PyTorch 文档中称为急切模式 eager mode）。立即执行很有用，因为如果在执行表达式时出现问题，则 Python 解释器、调试器和类似工具可以直接访问所涉及的 Python 对象。可以在问题发生时直接引发异常。

另外，您甚至可以在还不知道输入是什么的时候定义毕达哥拉斯表达式，并在输入可用时使用该定义生成输出。您定义的可调用函数稍后可以在不同的输入中重复使用：

```
>>> p = lambda a, b: (a**2 + b**2) ** 0.5
>>> p(1, 2)
2.23606797749979
>>> p(3, 4)
5.0
```

在第二种情况下，您定义了要执行的一系列操作，从而产生了输出函数（在这种情况下为 p）。直到稍后，当您传递输入时，您才执行任何操作，这是延迟执行的一个示例。延迟执行意味着大多数异常是在调用函数时（而不是在定义函数时）引发的。对于普通的 Python（如您在此处看到的），这很好，因为在发生错误时，解释器和调试器可以完全访问 Python 状态。

当使用具有大量运算符重载的专用类时，事情变得棘手，这允许立即执行的操作推迟到后台进行。这些类如下所示：

```
>>> a = InputParameterPlaceholder()
>>> b = InputParameterPlaceholder()
>>> c = (a**2 + b**2) ** 0.5
>>> callable(c)
True
>>> c(3, 4)
5.0
```

通常在使用这种形式的函数定义的库中，对 a 和 b 进行平方、加法和取平方根的操作不会记录为高级 Python 字节码。相反，重点通常是将表达式编译成静态计算图（基本操作图），该图比纯 Python 有一些优势（例如出于性能原因将数学直接编译为机器码）。

计算图建在一个地方并在另一个地方使用的事实使调试变得更加困难，因为异常通常对出错的地方缺乏明确性，而 Python 调试工具对数据的中间状态没有任何可见性。此外，静态图通常无法与标准 Python 流量控制完美融合：它们是事实上的基于特定领域的语言，是在宿主语言（在这种情况下为 Python）上实现的。

接下来，我们将更具体地研究立即执行和延迟执行之间的区别，特别是与神经网络相关的问题。在这里，我们将不深入讲授这些概念，而是为您简要介绍这些术语以及这些概念之

间的关系。了解这些概念和关系为了解如何使用 PyTorch 之类的立即执行库与延迟执行框架之间的差异奠定了基础，即使两种类型的基础数学都相同。

神经网络的基本组成部分是神经元。神经元被大量串在一起形成网络。您可以在图 1.2 的第一行中看到单个神经元的典型数学表达式： $o = \tanh(w * x + b)$ 。在下图中解释执行模式时，请牢记以下事实：

- $x$  是单神经元计算的输入。
- $w$  和  $b$  是神经元的参数或权重，可以根据需要进行更改。
- 为了更新参数（以产生更符合我们期望的输出），我们通过反向传播将误差分配给每个权重，然后相应地调整权重。
- 反向传播需要计算输出相对于权重的梯度（除其他外）。
- 我们使用自动微分来自动计算梯度，从而免除了手工编写计算的麻烦。

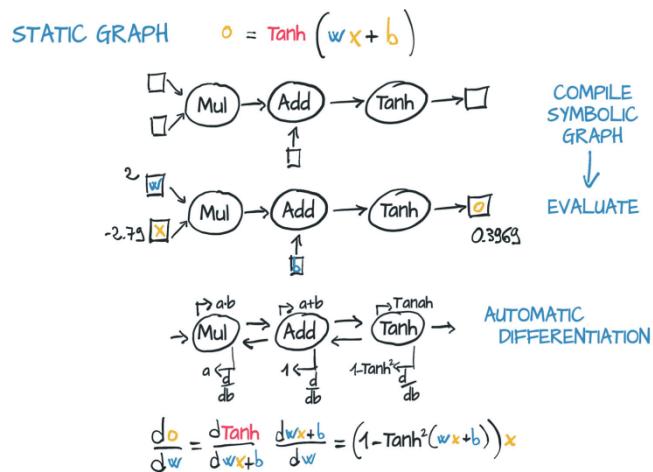


图 1.2 单个神经元上简单计算的静态图

在图 1.2 中，神经元被编译成一个符号图，其中每个节点代表单独的操作（第二行），图 1.2 静态图，用于使用占位符作为输入和输出的单个神经元的简单计算。然后，在将具体数字插入占位符时（在此情况下，数字是存储在  $w$ 、 $x$  和  $b$  中的值），将对图形进行数值评估（第三行）。通过自动微分来符号化地构建输出相对于权重的梯度，该自动微分向后遍历图并在各个节点（第四行）处乘以梯度。第五行显示了相应的数学表达式。

TensorFlow 是主要的有竞争力的深度学习框架之一，它具有使用相似类型的延迟执行的图形模式。图形模式是 TensorFlow 1.0 中的默认操作模式。相比之下，PyTorch 使用按运行定义的动态图引擎，其中逐个节点地构建计算图，急切地评估代码。

图 1.3 的上半部分显示了在动态图引擎下运行的相同计算。该计算被分解成单个表达式，遇到这些表达式时会对其进行贪婪地求值。该程序没有计算之间的互连的高级概念。图的下半部分显示了针对相同表达式的动态计算图的幕后构造。该表达式仍分为单独的操作，但是在此急切地评估这些操作，并逐步构建图形。类似于静态计算图，可以通过向后遍历结果图来实现自动微分。请注意，这并不意味着动态图库在本质上比静态图库要强大，只是动态图

库通常更容易完成循环或条件行为。

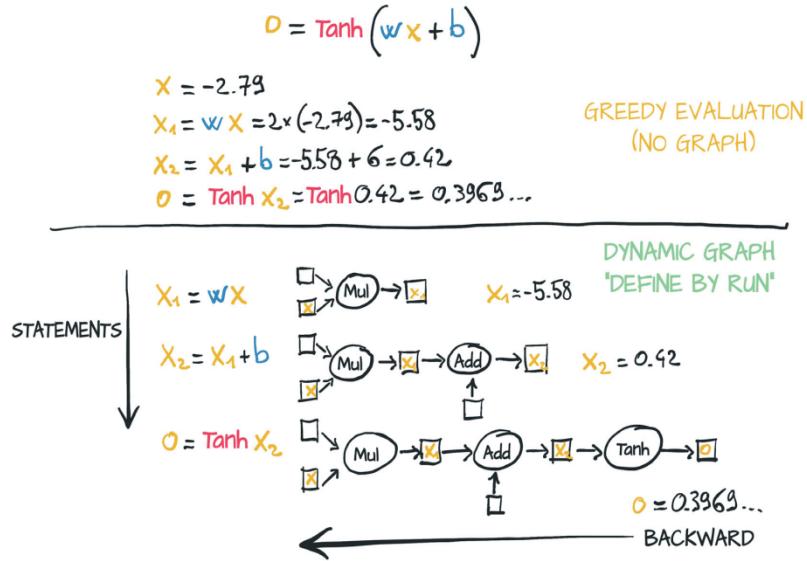


图 1.3 单个神经元的简单计算的动态图

动态图可以在连续的前向传递的过程中改变。例如，可以根据先前节点的输出上的条件来调用不同的节点，而无需在图形本身中表示这样的条件——与静态图形方法相比这样具有明显的优势。

主要框架正在趋向于支持两种操作模式。PyTorch 1.0 能够在静态计算图中记录模型的执行情况，或者通过预编译的脚本语言对其进行定义，从而提高了性能并易于模型投入生产。TensorFlow 还获得了“急切模式”，这是一种新的按运行定义的 API，从而增加了库的灵活性，正如我们已经所讨论过的。

### 1.3.3 深度学习竞争格局

尽管所有类比都有缺陷，但似乎于 2017 年 1 月发布的 PyTorch 0.1 标志着从寒武纪爆炸式增长（如深度学习库，包装器和数据交换格式的扩散）过渡到整合和统一的时代。

**注意** 深度学习的领域近来发展得如此之快，以至于您阅读本书时，某些方面可能已过时。如果您不熟悉此处提到的某些库，那很好。

在 PyTorch 的第一个 Beta 版本发布之时

- Theano 和 TensorFlow 是首要的低级延迟执行库。
- Lasagne 和 Keras 是 Theano 的高级包装，Keras 也包装 TensorFlow 和 CNTK。
- Caffe、Chainer、Dynet、Torch（PyTorch 基于 Lua 的前身）、mxnet、CNTK、DL4J 等填充了该生态系统中的各个领域。

在随后的大约两年中，情况发生了巨大变化。社区在 PyTorch 或 TensorFlow 的支持下已基本整合，其他库的使用逐渐减少或填补了领域：

- Theano 是最早的深度学习框架之一，已停止活跃发展。
- TensorFlow
  - 使用了 Keras，将其提升为第一类 API
  - 提供了立即执行急切模式
  - 宣布 TF 2.0 将默认启用急切模式
- PyTorch
  - 消耗了 Caffe2 作为其后端
  - 替换了基于 Lua 的 Torch 项目中重复使用的大多数低级代码
  - 添加了对 ONNX 的支持，这是一种与供应商无关的模型描述和交换格式
  - 添加了名为 TorchScript 的延迟执行图模式运行时
  - 发布版本 1.0

TensorFlow 拥有强大的生产流水线、广泛的行业社区和广泛的思维份额。由于 PyTorch 的易用性，它已在研究和教学领域取得了巨大进展，并且随着研究人员和毕业生训练学生并走向行业的发展，PyTorch 取得了增长。有趣的是，随着 TorchScript 和急切模式的问世，两个库都看到了它们的功能集开始融合。

## 1.4 PyTorch 拥有内置的电池

我们已经暗示了 PyTorch 的一些组件。现在，我们将花一些时间来正式确定构成 PyTorch 的主要组件的高级图谱。

首先，PyTorch 具有 Python 的 Py，但是其中包含许多非 Python 代码。出于性能原因，大多数 PyTorch 都是用 C++ 和 CUDA (<https://www.geforce.com/hardware/technology/cuda>) 编写的，这是 NVIDIA 的一种类似 C++ 的语言，可以将其编译为在 NVIDIA GPU 上以大规模并行性运行。有多种方法可以直接从 C 运行 PyTorch。此功能的主要动机之一是提供在生产中部署模型的可靠策略。但是，大多数情况下，您将通过 Python 与 PyTorch 进行交互，构建模型，对其进行训练并使用经过训练的模型来解决问题。根据给定用例对性能和规模的要求，纯 Python 解决方案足以将模型投入生产。例如，使用 Flask Web 服务器使用 Python API 包装 PyTorch 模型是完全可行的。

实际上，PyTorch 在可用性和与更广泛的 Python 生态系统的集成方面是 Python API 的亮点。接下来，我们来看看 PyTorch 的心理模型。

PyTorch 的核心是提供多维数组的库，在 PyTorch 术语中称为张量，而 torch 模块则提供了对其进行扩展的操作库。张量和相关操作都可以在 CPU 或 GPU 上运行。与 CPU 相比，在 GPU 上运行可以显着提高速度（特别是如果您愿意为高端 GPU 付费），而使用 PyTorch 则不需要多于一个或两个额外的函数调用。PyTorch 提供的第二个核心功能是允许张量跟踪对其执行的操作，并通过反向传播来分析输出相对于其任何输入的导数。此功能由张量本机提供，并在 torch.autograd 库中进一步完善。

我们可以说，通过拥有张量和启用了 autograd 的张量标准库，PyTorch 不仅可以用于神经网络，而且我们指出：PyTorch 可以用于物理、渲染、优化、模拟、建模等等。我们很可能看到 PyTorch 在各种科学应用中都以创造性的方式被使用。

但是 PyTorch 首先是深度学习库，因此，它提供了构建和训练神经网络所需的所有构建块。图 1.4 显示了一个标准设置，该标准设置可加载数据、训练模型，然后将该模型部署到生产中。

用于构建神经网络的 PyTorch 核心模块位于 torch.nn 中，该模块提供了常见的神经网络层和其他体系结构组件。此处可以找到完全连接的层、卷积层、激活函数和损失函数。这些组件可用于构建和初始化图 1.4 中所示的未经训练的模型。

要训练此模型，您需要做一些事情（除了循环本身，它可以是标准的 Python for 循环）：训练数据源、用于使模型适应训练数据的优化器以及获取模型的方法并将数据发送到硬件，该硬件将执行训练模型所需的计算。

可以在 torch.util.data 中找到用于数据加载和处理的实用程序。您将使用的两个主要类是 Dataset 类，它充当您的自定义数据（可能采用任何格式）之间的桥梁，以及标准化 PyTorch Tensor 类。您将看到的另一个类是 DataLoader，它可以产生子进程以在后台从 Dataset 加载数据，以便其准备就绪等待训练循环，并在循环中能立即使用。

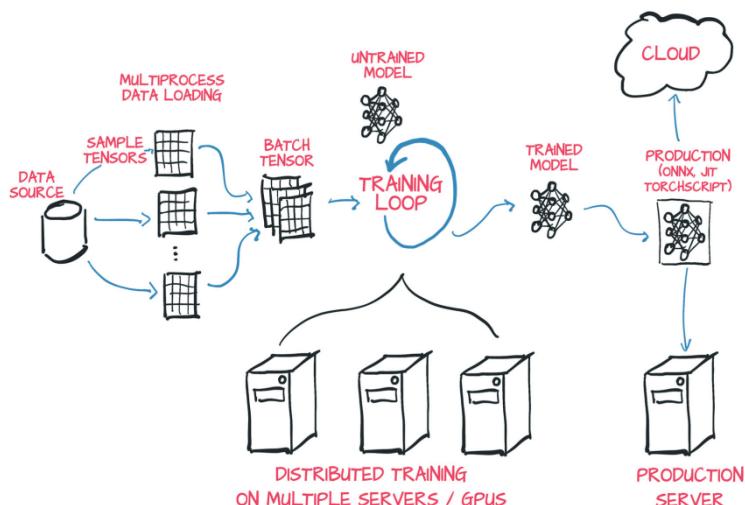


图 1.4 PyTorch 项目的基本高层结构，包括数据加载、训练和生产部署

在最简单的情况下，模型将在本地 CPU 或单个 GPU 上运行所需的计算，因此，当训练循环具有数据时，可以立即开始计算。但是，更常见的是要使用专用硬件（例如多个 GPU）

或让多台机器贡献其资源来训练模型。在这些情况下，可以使用 `torch.nn.DataParallel` 和 `torch.distributed` 来利用可用的其他硬件。

当从训练数据上运行模型获得结果时，`torch.optim` 提供了更新模型的标准方法，以便输出的答案更接近于训练数据中指定的结果。

如前所述，PyTorch 默认为立即执行模型（急切模式）。每当 Python 解释器执行涉及 PyTorch 的指令时，相应的操作就会由底层 C++ 或 CUDA 实现立即执行。随着更多的指令对张量进行操作，后端实现将执行更多的操作。此过程的速度与通常在 C++ 端的速度一样快，但是会增加通过 Python 调用该实现的开销。这是很小的费用，但会累加起来。

为了绕开 Python 解释器的成本并提供独立于 Python 运行时运行模型的机会，PyTorch 还提供了一个名为 TorchScript 的延迟执行模型。使用 TorchScript，PyTorch 可以序列化一组可以独立于 Python 调用的指令。您可以将此模型视为具有特定于张量操作的有限指令集的虚拟机。除了不产生调用 Python 的开销外，这种执行模式还为 PyTorch 提供了将已知操作序列及时转换为更有效的融合操作的机会。这些特征是 PyTorch 生产部署能力的基础。

### 1.4.1 深度学习的硬件

在任何新的便携式计算机或个人计算机的能力范围内，都可以在新数据上运行经过预训练的网络。即使重新训练一小部分预先训练的网络以使其专用于新数据集，也不一定需要专门的硬件。您可以在标准的个人计算机或笔记本电脑上跟随本书来学习。但是，我们预计，要完成对更高级示例的完整训练，将需要具有 CUDA 功能的图形处理单元 (GPU)，例如具有 8GB RAM 的 GPU（我们建议使用 NVIDIA GTX 1070 或更高版本）。但是，如果您的硬件具有较少的可用 RAM，则可以调整这些参数。

需要明确的是：如果您愿意等待，这种硬件不是必需的，但是在 GPU 上运行可以将训练时间至少缩短一个数量级（通常快 40 到 50 倍）。单独来看，在现代硬件（例如典型的笔记本电脑 CPU）上，计算参数更新所需的操作很快（从几分之一秒到几秒钟）。问题在于，训练涉及一次又一次地运行这些操作，并逐步更新网络参数以最大程度地减少训练错误。

规模适中的大型网络可能需要数小时至数天才能从头开始在配备了良好 GPU 的工作站上对大型、真实世界的数据集进行训练。可以通过在同一台计算机上使用多个 GPU 来减少时间，甚至可以通过使用多个配备有 GPU 的计算机集群来减少时间。由于云计算提供商提供服务，这些设置的价格比听起来要低很多。**DAWN-Bench**

(<https://dawn.cs.stanford.edu/benchmark/index.html>) 是斯坦福大学的一项有趣计划，旨在提供与公开数据集上的常见深度学习任务相关的训练时间和云计算成本的基准。

如果周围有 GPU，那就太好了。否则，我们建议您检查各种云平台的产品，其中许多云平台都提供预装有 PyTorch 且支持 GPU 的 Jupyter notebooks，通常会提供一个免费配额。

最后考虑：操作系统 (OS)。PyTorch 从其首个版本开始就支持 Linux 和 macOS，并在 2018 年获得 Windows 支持。由于当前的 Apple 笔记本电脑不包括支持 CUDA 的 GPU，因此 PyTorch 的预编译 macOS 软件包仅支持 CPU。我们尽量避免假设您运行的是特定操作

系统。脚本的命令行应转换为 Windows 兼容形式。为了方便起见，只要有可能，我们都将代码列出并运行在 Jupyter Notebook 上。

有关安装信息，请参阅官方网站上的《入门指南》(<https://pytorch.org/get-started/locally>)。我们建议 Windows 用户使用 Anaconda 或 Miniconda 进行安装。其他操作系统（例如 Linux）通常具有多种可行的选择，其中 Pip 是最常见的安装程序之一。当然，有经验的用户可以自由地以与其偏好开发环境最兼容的方式来安装软件包。

## 1.4.2 使用 Jupyter Notebooks

我们将假设您已安装 PyTorch 和其他依赖项，并已验证一切正常。我们将大量使用 Jupyter Notebooks 来运行示例代码。Jupyter Notebook 在浏览器中显示为页面，您可以通过该页面交互地运行代码。该代码由内核进行评估，内核是在服务器上运行的进程，准备接收代码以执行并发送回结果，这些结果将在页面上内嵌显示。笔记本在内存中维护内核状态，例如在代码评估期间定义的变量，直到内核终止或重新启动为止。与笔记本交互的基本单位是一个单元格，即页面上的一个框，您可以在其中键入代码并让内核对其进行评估（通过选择菜单项或按 Shift-Enter）。您可以将多个单元格添加到笔记本中，新单元格将显示您在较早的单元格中创建的变量。执行后，单元格最后一行返回的值将打印在该单元格下方，而绘图也是如此。通过混合源代码、评估结果和 Markdown 格式的文本单元，您可以生成漂亮的交互式文档。您可以在项目网站上 (<https://jupyter.org>) 阅读有关 Jupyter Notebooks 的所有内容。

此时，您需要从 GitHub 代码检查出的根目录中启动笔记本服务器。启动服务器时的外观取决于操作系统的详细信息以及 Jupyter 的安装方式和安装位置。如果您有任何疑问，请随时在我们的论坛 (<https://forums.manning.com/forums/deep-learning-with-pytorch>) 上提问。notebook 服务器启动时，将弹出默认浏览器，并显示本地 notebook 文件列表。

Jupyter Notebooks 是用于通过代码来表达和调查想法的强大工具。尽管我们认为它们非常适合我们的用例，但它们并不适合所有人。我们认为，重要的是要集中精力消除摩擦并最大程度地减少认知负担，这对每个人来说都是不同的。在 PyTorch 进行实验期间，可以使用您喜欢的工具。

您可以在 GitHub 上的存储库 (<https://github.com/deep-learning-with-pytorch/dlwpt-code>) 中找到本书清单中的完整工作代码。

## 练习

· 启动 Python 以获取交互式命令提示符。

- 您使用的是哪个 Python 版本：2.x 或 3.x？

- 你能 import torch 吗？您得到的 PyTorch 是什么版本？

- torch.cuda.is\_available()的结果是什么？根据您使用的硬件，它是否符合您的期望？

- 启动 Jupyter Notebook 服务器。
  - Jupyter 使用的是什么版本的 Python?
  - Jupyter 使用的 torch 库的位置与您从交互式命令提示符中导入的 torch 库的位置相同吗?

## 总结

- 深度学习模型会自动学习将示例中的输入与所需输出相关联。
- 像 PyTorch 这样的库可让您有效地构建和训练神经网络模型。
- PyTorch 可以最大程度地减少认知开销，同时注重灵活性和速度。它默认的模式是立即执行操作。
- TorchScript 是可以从 C ++ 调用的预编译的延迟执行模式。
- 自 2017 年初发布 PyTorch 以来，深度学习工具生态系统已得到大幅整合。
- PyTorch 提供了一些实用程序库来促进深度学习项目。

## 2 从张量开始

本章覆盖内容有：

- 张量，PyTorch 中的基本数据结构
- 索引并在 PyTorch 张量上进行操作以探索和处理数据

深度学习支持许多应用程序，这些应用程序总是包括以某种形式获取数据，例如图像或文本，并以另一种形式生成数据，例如标签，数字或更多文本。从这个角度来看，深度学习包括构建一个可以将数据从一种表示转换为另一种表示的系统。通过从呈现所需映射的一系列示例中提取共性来驱动此转换。例如，系统可能会记录狗的一般形状和金毛猎犬的典型颜色。通过组合这两个图像属性，系统可以将具有给定形状和颜色的图像正确地映射到金毛猎犬标签，而不是黑色实验室（就此而言，是黄褐色的雄猫）。最终的系统可以利用大量类似输入，并为这些输入产生有意义的输出。

这个过程的第一步是将输入转换为浮点数，如图 2.1 的第一步所示（以及许多其他类型的数据）。由于网络使用浮点数来处理信息，因此您需要一种方法，将要处理的实际数据编码为网络可利用的内容，为此，然后将输出解码回可以理解和使用的形式。

从一种数据形式到另一种数据形式的转换通常是由一个深度神经网络分阶段学习的，这意味着您可以将阶段之间的部分转换的数据视为一系列中间表示。对于图像识别，早期的表示形式可以是事物（例如边缘检测）或纹理（例如毛发）。较深的表示可以捕获更复杂的结构（如耳朵、鼻子或眼睛）。

通常，这种中间表示形式是浮点数的集合，这些浮点数表征输入并捕获数据中的结构，从而有助于描述输入如何映射到神经网络的输出。这种表征特定于手头的任务，可以从相关示例中学习。这些浮点数的集合及其操作是现代 AI 的核心。请务必记住，这些中间表示（例如图 2.1 第二步所示）是将输入与前一层神经元的权重相结合的结果。每个中间表示对于之前的输入都是唯一的。

在开始将数据转换为浮点输入的过程之前，您必须对 PyTorch 如何处理和存储数据有深刻的理解：作为输入，作为中间表示和作为输出。本章致力于提供这部分的准确理解。

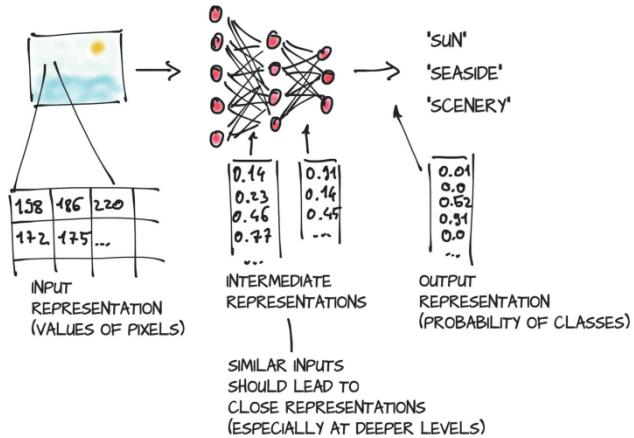


图 2.1 深度神经网络学习如何将输入表示转换为输出表示。（注意：神经元和输出的数量不成比例。）

为此，PyTorch 引入了一个基本的数据结构：张量。对于那些来自数学、物理学或工程学的人，张量一词与空间、参考系统以及它们之间转换的概念联系在一起。对于其他人，张量指的是将向量和矩阵推广到任意数量的维度，如图 2.2 所示。相同概念的另一个名称是多维数组。张量的维数与用于引用张量内标量值的索引数一致。

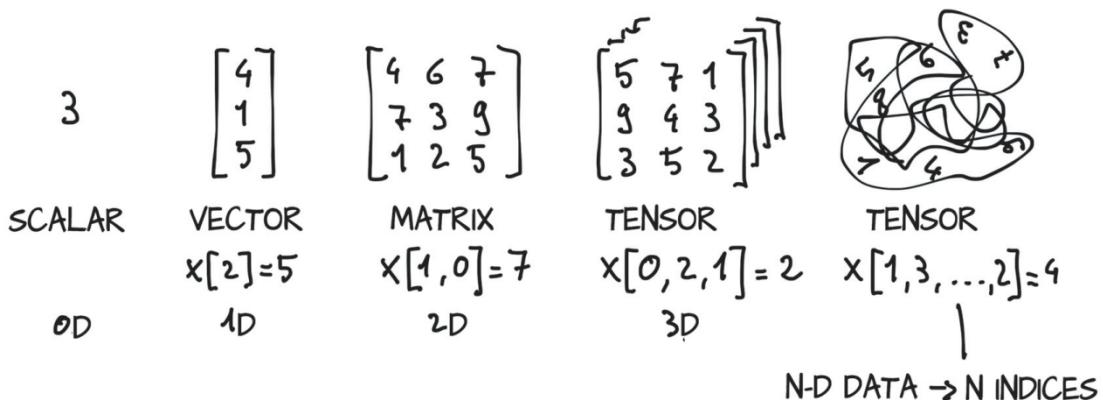


图 2.2 张量是在 PyTorch 中表示数据的基础

PyTorch 并不是唯一处理多维数组的库。NumPy 是迄今为止最受欢迎的多维数组库，以至于它可以说已经成为数据科学的通用语言。实际上，PyTorch 具有与 NumPy 的无缝互操作性，与 NumPy 具有优良的集成，可与 Python 中的其他科学库（如 SciPy (<https://www.scipy.org>)，Scikit-learn (<https://scikit-learn.org/stable>) 和 Pandas (<https://pandas.pydata.org>) ) 进行优良的集成。

与 NumPy 数组相比，PyTorch 张量具有一些超级功能，例如能够在图形处理单元 (GPU) 上执行快速操作，在多个设备或机器上分配操作以及跟踪创建它们的计算图的能力。所有这些功能对于实现现代深度学习库都很重要。

我们通过介绍 PyTorch 张量开始本章，涵盖了使事物运转的基础知识。我们将向您展示如何使用 PyTorch 张量库来操纵张量，涵盖诸如数据如何存储在内存中以及如何以恒定的时间对任意大张量执行某些操作之类的内容。然后我们继续前面提到的 NumPy 互操作性和

GPU 加速。

如果要将张量用作编程工具箱中的工具，那么了解张量的功能和 API 很重要。

## 2.1 张量基础

您已经了解到，张量是 PyTorch 中的基本数据结构。张量是一个数组，即一种存储数字集合的数据结构，这些数字可以通过索引单独访问，并且可以使用多个索引进行索引。

查看实际使用的列表索引，以便将其与张量索引进行比较。以下清单显示了 Python 中三个数字的列表。

**Listing 2.1 code/p1ch3/1\_tensors.ipynb**

```
# In[1]:  
a = [1.0, 2.0, 1.0]
```

您可以使用对应的从 0 开始的索引来访问列表的第一个元素：

```
# In[2]:  
a[0]  
  
# Out[2]:  
1.0  
  
# In[3]:  
a[2] = 3.0  
a  
  
# Out[3]:  
[1.0, 2.0, 3.0]
```

处理数字矢量（例如 2D 线的坐标）的简单 Python 程序使用 Python 列表存储矢量并不罕见。但是，由于以下几个原因，这种做法可能不是最佳的：

- Python 中的数字是成熟的对象。浮点数可能只需要 32 位就可以在计算机上表示，而 Python 将它们封装在功能齐全的 Python 对象中，并带有引用计数等。如果您需要存储少量数字，这种情况就不成问题了，但是分配数百万个这样的数字效率很低。
- Python 中的列表用于对象的顺序集合。没有定义用于有效地获取两个向量的求和或者点积操作。另外，Python 列表无法优化其内容在内存中的布局，因为它们是指向 Python 对象（任何类型，而不仅仅是数字）的可索引指针集合。最后，Python 列表是一维的，尽管您可以创建列表来嵌套列表，但这种做法仍然效率低下。
- 与经过优化的编译后的代码相比，Python 解释器的运行速度较慢。使用已编译的低级语言（如 C）编写的优化代码，必然可以更快地对大量数字数据执行数学运算。

由于这些原因，数据科学库依赖 NumPy 或引入专用数据结构（例如 PyTorch 张量），这些结构提供了方便的高级 API 封装的数字数据结构及其相关操作的有效低层实现。

张量可以表示许多类型的数据，表示范围包括图像、时间序列、音频甚至句子。通过定义张量上的操作（本章中将探讨其中的一些操作），您可以同时高效且富有表现力地对数据进行切片和操作，尽管你也可以使用高级（但不是特别快）的语言（例如 Python）进行切片和操作。

现在，您可以构建第一个 PyTorch 张量，以查看其外观。这个张量现在不会特别有意义，因为它只有一列三个元素 1：

```
# In[4]:  
import torch  
a = torch.ones(3)  
a  
  
# Out[4]:  
tensor([1., 1., 1.])  
  
# In[5]:  
a[1]  
  
# Out[5]:  
tensor(1.)  
  
# In[6]:  
float(a[1])  
  
# Out[6]:  
1.0  
  
# In[7]:  
a[2] = 2.0  
a  
  
# Out[7]:  
tensor([1., 1., 2.])
```

现在看看您在这里做了什么。导入 `torch` 模块后，您调用了一个函数，该函数创建大小为 3 的（一维）张量，且填充值为 1.0。您可以使用基于 0 的索引来访问元素，也可以为其分配新的值。

尽管从表面上看，此示例与一系列数字对象并没有太大区别，但实际上，情况完全不同。Python 列表或数字元组是在内存中单独分配的 Python 对象的集合，如图 2.3 左侧所示。另一方面，PyTorch 张量或 NumPy 数组是（通常）包含未装箱的 C 数值类型而不是 Python 对象的连续内存块上的视图。在这种情况下，是 32 位（4 字节）浮点数，如图 2.3 右侧所示。因此，包含一百万个浮点数的一维张量需要 400 万个连续字节来存储，再加上少量的元数据开销（尺寸、数字类型等）。

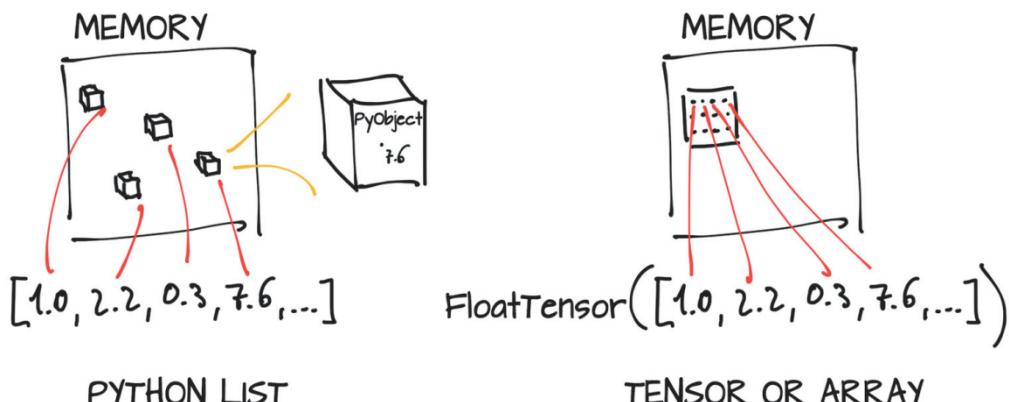


图 2.3 Python 对象（装箱）数值与张量（未装箱数组）数值

假设您要管理一个二维坐标列表，以表示一个几何对象，例如三角形。该示例与深度学习并不特别相关，但是很容易理解。您可以通过将 x 坐标存储在偶数索引中并将 y 坐标存储在奇数索引中来使用一维张量，而不是在 Python 列表中将坐标作为数字使用，如下所示：

```
# In[8]:  
points = torch.zeros(6)  
points[0] = 1.0  
points[1] = 4.0  
points[2] = 2.0  
points[3] = 1.0  
points[4] = 3.0  
points[5] = 5.0
```

The use of .zeros here is a way to get an appropriately sized array.  
Overwrite those zeros with the values you want.

您还可以将 Python 列表传递给构造器以达到相同的效果

```
# In[9]:  
points = torch.tensor([1.0, 4.0, 2.0, 1.0, 3.0, 5.0])  
points  
  
# Out[9]:  
tensor([1., 4., 2., 1., 3., 5.])
```

获取第一点的坐标：

```
# In[10]:  
float(points[0]), float(points[1])  
  
# Out[10]:  
(1.0, 4.0)
```

这项技术还可以，尽管让第一个索引引用单个 2D 点而不是点坐标是可行的。为此，可以使用 2D 张量：

```
# In[11]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
points  
  
# Out[11]:  
tensor([[1., 4.],  
       [2., 1.],  
       [3., 5.]])
```

在这里，您将嵌套列表的列表传递给了构造器。您可以询问张量其形状，

```
# In[12]:  
points.shape  
  
# Out[12]:  
torch.Size([3, 2])
```

它会告知您张量沿每个维度的大小。 您也可以使用 `zeros` 函数或 `ones` 函数来初始化张量，并且以元组形式指定张量的大小：

```
# In[13]:  
points = torch.zeros(3, 2)  
points  
  
# Out[13]:  
tensor([[0., 0.],  
       [0., 0.],  
       [0., 0.]])
```

现在，您可以使用两个索引访问张量中的单个元素：

```
# In[14]:  
points = torch.FloatTensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
points  
  
# Out[14]:  
tensor([[1., 4.],  
       [2., 1.],  
       [3., 5.]])  
  
# In[15]:  
points[0, 1]  
  
# Out[15]:  
tensor(4.)
```

此代码返回数据集中第 0 个点的 y 坐标。您还可以像以前一样访问张量中的第一个元素，以获取第一个点的 2D 坐标：

```
# In[16]:  
points[0]  
  
# Out[16]:  
tensor([1., 4.])
```

请注意，输出结果是另一个张量，但大小为 2 的一维张量且包含 `points` 张量的第一行中的值。此输出是否意味着分配了新的内存块，将值复制到其中，并返回了将新的张量对象包装在内新的内存？不是的，因为该过程效率不高，尤其是如果您拥有数百万个点时。相反，您得到的是相同底层数据的仅限于第一行的不同视图。

## 2.2 张量和存储

在本节中，您将开始获得有关内部实现的提示。值分配在由 `torch.Storage` 实例管理的连续内存块中。一个 `storage` 对象是数值数据的一维数组，例如包含给定类型（可能是 `float` 或 `int32`）数字的连续内存块。PyTorch `Tensor` 类型是这种 `Storage` 类型的视图，能够通过使用偏移量和维度步长将其索引到该存储中。

多个张量可以索引同一存储，即使它们以不同的方式索引数据。您可以在图 2.4 中看到一个示例。实际上，当您在最后一个片段中请求 `points[0]` 时，您得到的是另一个张量，该张量索引与张量 `points` 相同的存储，但不是全部的存储并且具有不同的维数（1D 与 2D）。但是，底层内存仅分配一次，因此，无论 `Storage` 实例管理的数据大小如何，都可以快速完成在数据上创建替代张量视图的操作。

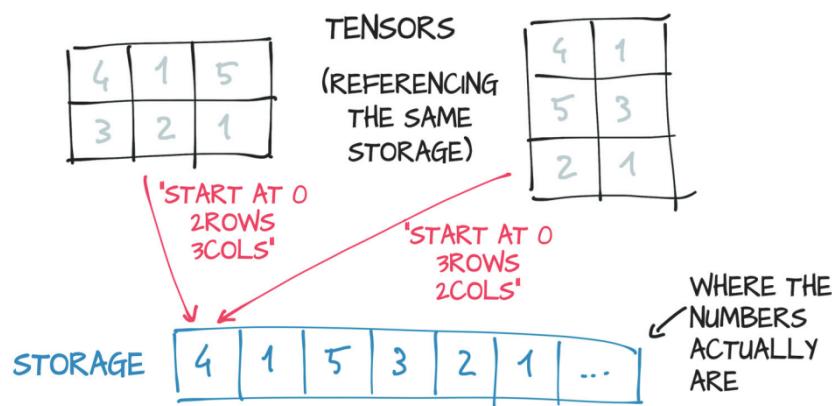


图 2.4 张量 Tensor 是 Storage 实例的视图

接下来，您将看到在实际中使用 2D 点索引到存储的方式。您可以使用 `.storage` 属性访问给定张量的存储：

```
# In[17] :
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points.storage()

# Out[17] :
1.0
4.0
2.0
1.0
3.0
5.0
[torch.FloatTensor of size 6]
```

尽管张量显示其本身具有三行两列，但底层存储之下的却是大小为 6 的连续数组。从这个意义上讲，张量知道如何将一对索引转换为存储中的某个位置。

您还可以手动索引到存储中：

```
# In[18]:  
points_storage = points.storage()  
points_storage[0]  
  
# Out[18]:  
1.0  
  
# In[19]:  
points.storage()[1]  
  
# Out[19]:  
4.0
```

您无法使用两个索引来索引 2D 张量的存储。存储器的布局始终是一维的，而与可能引用它的任何张量的维数无关。

在这一点上，更改存储的值会更改其引用张量的内容不足为奇：

```
# In[20]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
points_storage = points.storage()  
points_storage[0] = 2.0  
points  
  
# Out[20]:  
tensor([[2., 4.],  
       [2., 1.],  
       [3., 5.]])
```

您几乎很少会直接使用 `storage` 实例，但是了解张量和底层 `storage` 存储之间的关系对于以后了解某些操作的代价（或其缺乏）很有用。当您要编写有效的 PyTorch 代码时，请牢记这一思维模型。

## 2.3 尺寸、存储偏移量和步长

为了索引存储，张量依赖于几条信息以及它们的存储，并且明确地定义它们 大小、存储偏移和步长（图 2.5）。大小（或形状，按照 NumPy 的说法）是一个元组，指示张量表示的每个维度上有多少个元素。存储偏移量是存储器与张量中第一个元素相对应的索引。步长是存储中需要跳过的元素数量，以便沿每个维度获取下一个元素。

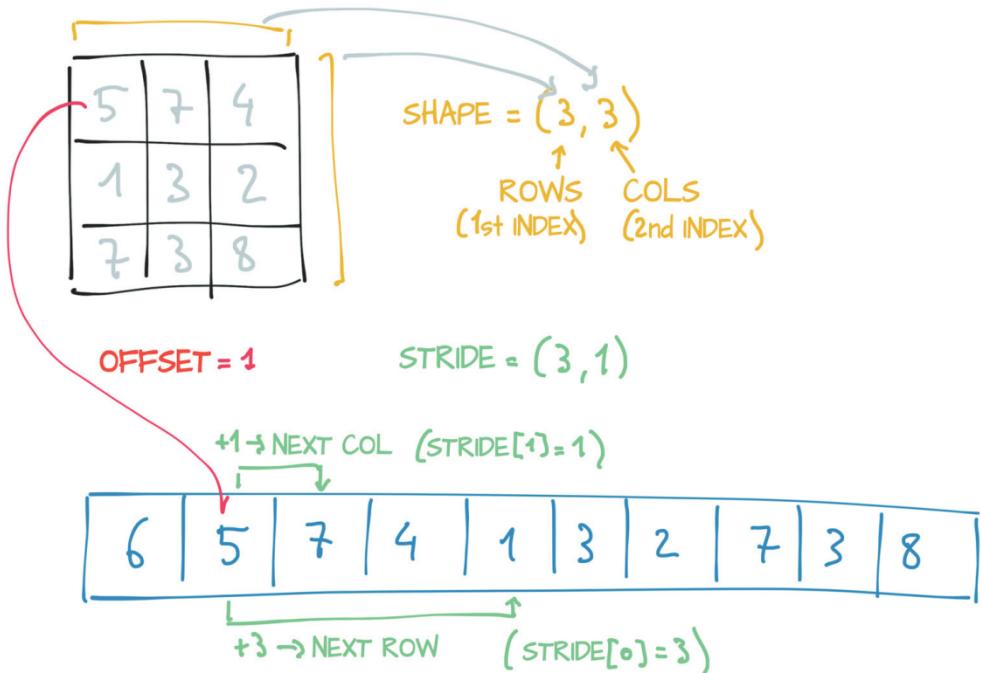


图 2.5 张量的偏移量、大小和步幅之间的关系

您可以通过提供相应的索引来获得张量中的第二个点：

```
# In[21]:
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
second_point = points[1]
second_point.storage_offset()

# Out[21]:
2

# In[22]:
second_point.size()

# Out[22]:
torch.Size([2])
```

结果张量在存储中的偏移量为 2 (因为我们需要跳过第一个点, 该点有两个项目), 并且 size 是包含一个元素的 Size 类的实例, 因为张量是一维的。重要说明 此信息与张量对象的 shape 属性中包含的信息相同:

```
# In[23]:
second_point.shape

# Out[23]:
torch.Size([2])
```

最后, 步长是一个元组, 指示当索引在每个维度上增加 1 时必须跳过的存储中元素的数量。例如, 您的 points 张量使用一个步长:

```
# In[24]:  
points.stride()  
  
# Out[24]:  
(2, 1)
```

在 2D 张量中访问元素  $i$ 、 $j$  导致访问存储器中的位置  $\text{storage\_offset} + \text{stride}[0] * i + \text{stride}[1] * j$  元素。偏移量通常为零；如果此张量是在存储更大的张量基础上而创建的存储的视图，则偏移量可能为正值。

张量和存储之间的这种间接性导致某些操作（例如转置张量或提取子张量）的代价很小，因为它们不会导致内存重新分配；相反，它们包括分配一个新的张量对象，该对象的大小、存储偏移量或步长具有不同的值。

您看到了在为特定点建立索引时如何提取子张量，并且看到存储偏移量增加了。现在看看大小和步幅发生了什么：

```
# In[25]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
second_point = points[1]  
second_point.size()  
  
# Out[25]:  
torch.Size([2])  
  
# In[26]:  
second_point.storage_offset()  
  
# Out[26]:  
2  
  
# In[27]:  
second_point.stride()  
  
# Out[27]:  
(1,)
```

最重要的是，子张量减少了一个维度（如您所愿），同时仍然索引到与原始点张量相同的存储。更改子张量也会对原始张量产生影响：

```
# In[28]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
second_point = points[1]  
second_point[0] = 10.0  
points  
  
# Out[28]:  
tensor([[ 1.,  4.],  
       [10.,  1.],  
       [ 3.,  5.]])
```

这种效果可能并不总是我们想要的，因此您同样也可以将子张量克隆到新的张量中：

```
# In[29]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
second_point = points[1].clone()  
second_point[0] = 10.0  
points  
  
# Out[29]:  
tensor([[1., 4.],  
       [2., 1.],  
       [3., 5.]])
```

现在尝试转置。取 points 张量，在行中具有单个点，在列中具有 x 和 y 坐标，然后将其翻转以使单个点沿着列排列：

```
# In[30]:  
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])  
points  
  
# Out[30]:  
tensor([[1., 4.],  
       [2., 1.],  
       [3., 5.]])  
  
# In[31]:  
points_t = points.t()  
points_t  
  
# Out[31]:  
tensor([[1., 2., 3.],  
       [4., 1., 5.]])
```

您可以很容易地验证两个张量共享存储

```
# In[32]:
id(points.storage()) == id(points_t.storage())

# Out[32]:
True
```

并且它们仅在形状和步长上有所不同：

```
# In[33]:
points.stride()

# Out[33]:
(2, 1)
# In[34]:
points_t.stride()

# Out[34]:
(1, 2)
```

该结果告诉您，将在 points 中的第一个索引增加 1（即，从 points [0,0] 到 points [1,0]）沿存储跳过两个元素，并且将第二个索引从 points [0,0] 到点 [0,1] 沿存储跳过一。换句话说，存储器将张量中的元素逐行顺序地保存。

您可以将 points 转置为 points\_t，如图 2.6 所示。您可以用步长来更改元素的顺序。之后，增加行（张量的第一个索引）会沿着存储跳过 1，就像沿着 points 的列移动一样。这是转置的定义。没有分配新的内存：仅通过创建一个新的 Tensor 实例来获得转置，而该 Tensor 实例的步长顺序与原始顺序不同。

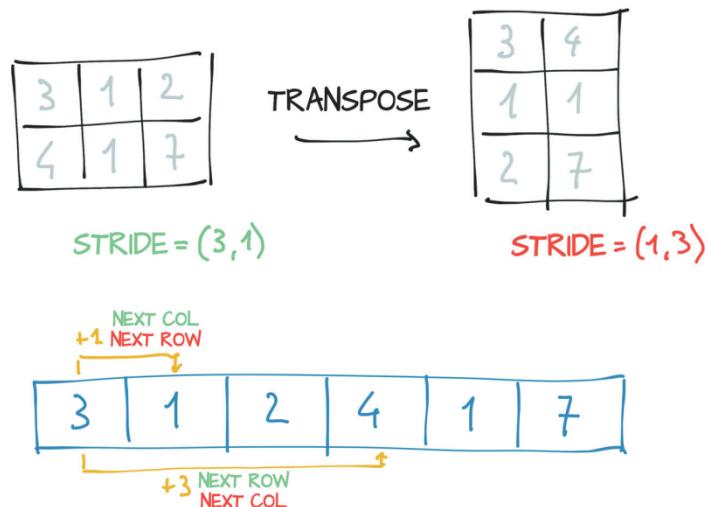


图 2.6 应用于张量的转置操作

在 PyTorch 中进行转置不仅限于矩阵。您可以通过指定应沿其发生转置（例如翻转形状和步长）的两个维度来转置多维数组：

```
# In[35]:  
some_tensor = torch.ones(3, 4, 5)  
some_tensor_t = some_tensor.transpose(0, 2)  
some_tensor.shape  
  
# Out[35]:  
torch.Size([3, 4, 5])  
  
# In[36]:  
some_tensor_t.shape  
  
# Out[36]:  
torch.Size([5, 4, 3])  
  
# In[37]:  
some_tensor.stride()  
  
# Out[37]:  
(20, 5, 1)  
  
# In[38]:  
some_tensor_t.stride()  
  
# Out[38]:  
(1, 5, 20)
```

将其值从最右边的维度开始（例如，针对 2D 张量沿行移动），布置在存储中的张量定义为连续的。连续张量很方便，因为您可以高效且有序地访问它们，而无需在存储中四处跳转。（由于内存访问在现代 CPU 中的工作方式，因此改善数据局部性可提高性能。）

在这种情况下，`points` 是连续的，但其转置不是：

```
# In[39]:  
points.is_contiguous()  
  
# Out[39]:  
True  
  
# In[40]:  
points_t.is_contiguous()  
  
# Out[40]:  
False
```

您可以使用 `contiguous` 方法从非连续张量中获得新的连续张量。张量的内容保持不变，但步长发生变化，存储也是如此：

```
# In[41]:
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points_t = points.t()
points_t

# Out[41]:
tensor([[1., 2., 3.],
       [4., 1., 5.]])

# In[42]:
points_t.storage()

# Out[42]:
1.0
4.0
2.0
1.0
3.0
5.0
[torch.FloatTensor of size 6]

# In[43]:
points_t.stride()

# Out[43]:
(1, 2)

# In[44]:
points_t_cont = points_t.contiguous()
points_t_cont

# Out[44]:
tensor([[1., 2., 3.],
       [4., 1., 5.]])

# In[45]:
points_t_cont.stride()

# Out[45]:
(3, 1)

# In[46]:
points_t_cont.storage()

# Out[46]:
1.0
2.0
3.0
4.0
1.0
5.0
[torch.FloatTensor of size 6]
```

请注意，已对存储进行了重新安排，以便在新存储中逐行布置元素。步长已更改以反映新的布局。

## 2.4 数值类型

很好，您现在已经知道张量如何工作的基础知识。但是我们没有涉及可以存储在张量中的数值类型。张量构造器的 `dtype` 参数（即 `tensor`、`zeros` 和 `ones` 的函数）指定将包含在张量中的数值数据类型。数据类型指定张量可以容纳的可能值（整数与浮点数）以及每个值的字节数。`dtype` 参数故意类似于同名的标准 NumPy 参数。以下是 `dtype` 参数的可能值的列表：

- `torch.float32` 或 `torch.float-32` 位浮点数
- `torch.float64` 或 `torch.double-64` 位双精度浮点数
- `torch.float16` 或 `torch.half-16` 位半精度浮点数
- `torch.int8`-带符号的 8 位整数
- `torch.uint8`-无符号 8 位整数
- `torch.int16` 或 `torch.short`-有符号 16 位整数
- `torch.int32` 或 `torch.int`-有符号 32 位整数
- `torch.int64` 或 `torch.long`-带符号的 64 位整数

`torch.float`、`torch.double` 等每个都有对应的具体类 `torch.FloatTensor`、`torch.DoubleTensor` 等。`torch.int8` 的类为 `torch.CharTensor`，而 `torch.uint8` 的类为 `torch.ByteTensor`。`torch.Tensor` 是 `torch.FloatTensor` 的别名。默认数据类型为 32 位浮点数。

要分配正确的数值类型的张量，可以为构造器的参数指定适当的 `dtype`，如下所示：

```
# In[47]:  
double_points = torch.ones(10, 2, dtype=torch.double)  
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
```

您可以通过访问相应的属性来查找张量的 `dtype`：

```
# In[48]:  
short_points.dtype  
  
# Out[48]:  
torch.int16
```

您还可以使用相应的转换方法将张量创建函数的输出转换为正确的类型，例如

```
# In[49]:  
double_points = torch.zeros(10, 2).double()  
short_points = torch.ones(10, 2).short()
```

或者更方便的方法：

```
# In [50]:  
double_points = torch.zeros(10, 2).to(torch.double)  
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

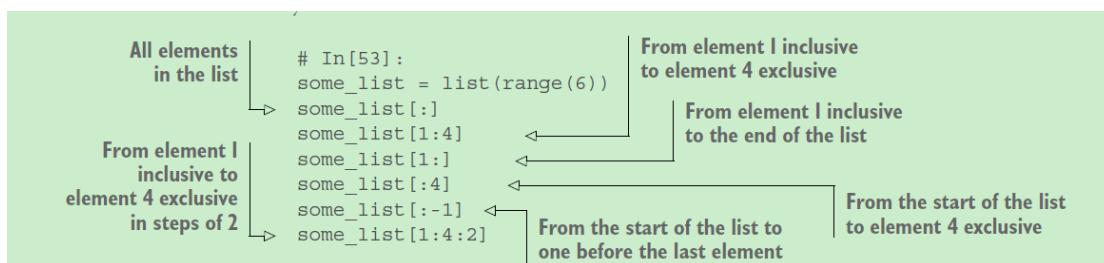
在底层，`type` 和 `to` 函数执行相同的类型的操作，即检查若有需要并转换操作，但是 `to` 方法可以接收其他参数。

您始终可以使用 `type` 方法将一种类型的张量转换为另一种类型的张量：

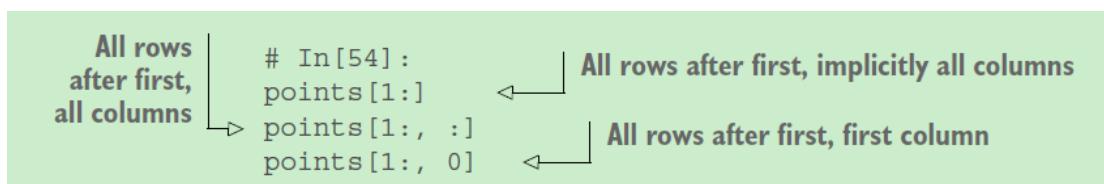
```
# In [51]:  
points = torch.randn(10, 2)      ← randn initializes the tensor elements to  
short_points = points.type(torch.short)    random numbers between 0 and 1.
```

## 2.5 索引张量

您已经看到 `points[0]` 返回一个张量，该张量包含在原张量第一行的 2D 点。如果您需要获取一个包含除第一个点之外的所有点的张量怎么办？当您使用范围索引表示法（适用于标准 Python 列表的类型）时，该任务很容易：



为了实现您的目标，您可以对 PyTorch 张量使用相同的符号，并具有与 NumPy 和其他 Python 科学库一样的额外好处，我们可以对张量的每个维度使用范围索引：



除了使用范围外，PyTorch 还具有强大的索引形式，称为高级索引。

## 2.6 NumPy 的互操作性

尽管我们不认为 NumPy 的经验是阅读本书的先决条件，但由于 NumPy 在 Python 数据科学

生态系统中无处不在，因此我们强烈建议您熟悉 NumPy。PyTorch 张量可以高效转换为 NumPy 数组，反之亦然。这样，您可以利用围绕 NumPy 数组类型构建的更广泛的 Python 生态系统中的大量功能。NumPy 数组的这种零拷贝互操作性是由于存储系统可与 Python 缓冲区协议（<https://docs.python.org/3/c-api/buffer.html>）一起工作。

要从 points 张量中获取 NumPy 数组，请调用

```
# In[55]:  
points = torch.ones(3, 4)  
points_np = points.numpy()  
points_np  
  
# Out[55]:  
array([[1., 1., 1., 1.],  
       [1., 1., 1., 1.],  
       [1., 1., 1., 1.]], dtype=float32)
```

它返回大小、形状和数值类型正确的 NumPy 多维数组。有趣的是，返回的数组与张量的存储共享一个底层缓冲区。这将导致，只要数据位于 CPU 的 RAM 中，`numpy` 方法就可以基本不花费任何代价地有效执行，并且修改 NumPy 数组会导致原始张量发生变化。

如果张量分配在 GPU 上，PyTorch 会将张量的内容复制到在 CPU 上分配的 NumPy 数组中。

相反，您可以通过以下这种方式从 NumPy 数组获得 PyTorch 张量

```
# In[56]:  
points = torch.from_numpy(points_np)  
which uses the same buffer-sharing strategy.
```

这个过程也使用相同的缓冲区共享策略。

## 2.7 序列化张量

动态创建张量很好，但是，如果其中的数据对您来说具有任何价值，那么您希望将其保存到文件中并在某个时候加载回去。毕竟，您不想要每次开始运行程序都从头开始重新训练模型！PyTorch 在内部中使用的 pickle 来序列化 tensor 张量对象，以及用于存储的专用序列化代码。您可以通过以下方法将 points 张量保存到 ourpoints.t 文件中：

```
# In[57]:  
torch.save(points, '../data/p1ch3/ourpoints.t')
```

或者，您可以传递文件描述符来代替文件名：

```
# In[58]:  
with open('../data/p1ch3/ourpoints.t', 'wb') as f:  
    torch.save(points, f)
```

同样地，将您的 points 用一行代码就能载入：

```
# In[59]:  
points = torch.load('../data/p1ch3/ourpoints.t')
```

等效为

```
# In[60]:  
with open('../data/p1ch3/ourpoints.t', 'rb') as f:  
    points = torch.load(f)
```

如果只想通过 PyTorch 来加载张量，则该技术可让您快速保存张量，但是该文件格式本身不具有互操作性。除 PyTorch 外，您无法使用其他软件读取张量。根据使用情况，这种情况可能不是限制条件，但您应该学习如何在那些时候互操作性地保存张量。尽管每个用例都是唯一的，但我们相信当您将 PyTorch 引入已经依赖于不同库的现有系统中时，这种情况会更常见。新项目可能不需要经常互操作性地保存张量。

但是，对于有需要的情况，可以使用 HDF5 (<https://www.hdfgroup.org/solutions/hdf5>) 格式和该库。HDF5 是一种可移植的、广泛支持的格式，用于表示以嵌套键值字典形式组织的序列化多维数组。Python 通过 h5py 库 (<http://www.h5py.org>) 支持 HDF5，该库以 NumPy 数组的形式接收和返回数据。

您可以这样使用安装 h5py：

```
$ conda install h5py
```

此时，您可以通过将 points 张量转换为 NumPy 数组（如前所述，无需额外开销）并将其传递给 create\_dataset 函数来保存 points 张量：

```
# In[61]:  
import h5py  
  
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'w')  
dset = f.create_dataset('coords', data=points.numpy())  
f.close()
```

在这里，'coords'是 HDF5 文件的关键字。您可以有其他关键字，甚至是嵌套键。HDF5 的一件有趣的事是，您可以在磁盘上对数据集编制索引，并仅访问您感兴趣的元素。假设您只

想加载数据集中的最后两点：

```
# In[62]:  
f = h5py.File('../data/plch3/ourpoints.hdf5', 'r')  
dset = f['coords']  
last_points = dset[1:]
```

在这里，打开文件或需要数据集时并未加载数据。事实上，数据一直保留在磁盘上，直到您请求数据集中的第二行和最后一行。这时，h5py 访问了这两列，并返回了一个类似 NumPy 数组的对象，该对象封装了该数据集中该区域，该区域的行为类似于 NumPy 数组并且具有相同的 API。

由于这个事实，您可以将返回的对象传递给 `torch.from_numpy` 函数以直接获取张量。请注意，在这种情况下，数据将复制到张量的存储中：

```
# In[63]:  
last_points = torch.from_numpy(dset[1:])  
f.close()  
>>> last_points = torch.from_numpy(dset[1:])  
When you finish loading data, close the file.
```

完成数据加载后，关闭文件。

## 2.8 将张量移动到 GPU

关于 PyTorch 张量的最后一点与 GPU 上的计算有关。每个 Torch 张量都可以转移到 GPU，以执行快速、大规模的并行计算。在张量上执行的所有操作均由 PyTorch 上的 GPU 特定例程执行。

**注意** 从 2019 年初开始，主要的 PyTorch 版本仅在支持 CUDA 的 GPU 上具有加速功能。已经存在在 AMD 的 ROCm (<https://rocm.github.io>) 平台上运行的 PyTorch 的概念验证版本已经问世，但是从 1.0 版开始，尚未完全支持 PyTorch。支持 Google TPU 的工作正在进行中（<https://github.com/pytorch/xla>），目前的概念验证版本已在 Google Colab 中向公众提供（<https://colab.research.google.com>）。在我们撰写本文时，尚未存在计划在其他 GPU 技术（例如 OpenCL）上实现数据结构和内核。

除 `dtype` 外，PyTorch 张量还具有设备的概念，这是在计算机上用于放置张量数据的位置。通过为构造器指定相应的参数，以下是在 GPU 上创建张量的方法：

```
# In[64]:  
points_gpu = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 4.0]],  
    device='cuda')
```

您可以改为使用 `to` 方法将在 CPU 上创建的张量复制到 GPU 中去：

```
# In[65]:  
points_gpu = points.to(device='cuda')
```

此代码返回一个具有相同数值数据的新张量，但将其存储在 GPU 的 RAM 中，而不是常规的系统 RAM 中。

既然数据已经存储在本地 GPU 上，那么在张量上执行数学运算时，您就会开始看到加速。同样，此新的 GPU 支持的张量的类也更改为 `torch.cuda.FloatTensor`。（考虑到之前的类型是 `torch.FloatTensor`；存在其他相应的集合 `torch.cuda.DoubleTensor`，依此类推。）几乎在所有情况下，基于 CPU 和 GPU 的张量都具有相同的面向用户的 API，因此更容易实现编写与运行位置无关的完成繁琐数字运算过程的代码。

如果您的机器具有多个 GPU，则可以通过传递标识机器上 GPU 的从零开始的整数来决定将张量分配给哪个 GPU：

```
# In[66]:  
points_gpu = points.to(device='cuda:0')
```

这时，对张量执行的任何操作（例如将所有元素乘以一个常数）都在 GPU 上执行：

```
# In[67]:  
points = 2 * points    ↪ Multiplication performed on the CPU  
points_gpu = 2 * points.to(device='cuda')    ↪ Multiplication performed on the GPU
```

请注意，计算结果后，`points_gpu` 张量不会被带回 CPU。以下是整个发生的过程：

- 1 `points` 张量复制到 GPU。
- 2 在 GPU 上分配了一个新的张量，用于存储乘法结果。
- 3 返回该 GPU 张量的句柄。

因此，如果您还想向结果中加上一个常数，

```
# In[68]:  
points_gpu = points_gpu + 4
```

加法仍然在 GPU 上执行，并且没有信息流到 CPU（除非您打印或访问得到的张量）。要将张量移回 CPU，需要为 `to` 方法提供一个 `cpu` 参数：

```
# In [69]:  
points_cpu = points_gpu.to(device='cpu')
```

您可以使用简写的 `cpu` 方法和 `cuda` 方法代替 `to` 方法来实现相同的目标：

```
# In [70]:  
points_gpu = points.cuda() ← Defaults to GPU index 0  
points_gpu = points.cuda(0)  
points_cpu = points_gpu.cpu()
```

值得一提的是，使用 `to` 方法时，可以通过提供 `device` 和 `dtype` 作为参数来同时更改位置和数据类型。

## 2.9 张量的 API

至此，您知道了什么是 PyTorch 张量以及它们如何在后台运行。在结束本章之前，我们将看一下 PyTorch 提供的张量操作。在这里没有必要用列举出所有这些操作。相反，我们将为您提供有关该 API 的一般信息，并在 (<http://pytorch.org/docs>) 的在线文档中向您展示在何处查找内容。

首先，在张量上和张量之间的绝大多数操作都可以在 `torch` 模块下获得，也可以通过 `tensor` 对象来调用这些方法。例如，您可以通过 `torch` 模块使用先前遇到的转置函数：

```
# In [71]:  
a = torch.ones(3, 2)  
a_t = torch.transpose(a, 0, 1)
```

或作为张量 `tensor` 的方法来使用：

```
# In [72]:  
a = torch.ones(3, 2)  
a_t = a.transpose(0, 1)
```

这两种形式之间没有区别，可以互换使用。需要注意的是：少量的操作仅作为张量 `tensor` 对象的方法而存在。可以通过名称中的下划线来识别它们，例如 `zero_`，这表示该方法通过原地修改输入并返回它而不是创建新的输出张量。例如，`zero_` 方法会将输入的所有元素清零。任何不带下划线的方法都将使源张量保持不变并返回新的张量：

```
# In[73]:  
a = torch.ones(3, 2)  
  
# In[74]:  
a.zero_()  
a  
  
# Out[74]:  
tensor([[0., 0.],  
       [0., 0.],  
       [0., 0.]])
```

之前，我们提到了在线 docs (<http://pytorch.org/docs>)，它是详尽的，并且将张量操作分为几组：

- 创建操作-构造张量的函数，例如，`ones` 函数和 `from_numpy` 函数
  - 索引、切片、连接和变异操作-更改张量的形状、步幅或内容的函数，例如转置 `transpose` 函数
  - 数学操作-通过计算来操作张量内容的函数：
  - 点向运算-通过将函数独立地应用于每个元素来获得新张量的函数，例如 `abs` 和 `cos` 函数
  - 化简运算-通过迭代张量来计算合计值的函数，例如 `mean`、`std`、`norm` 函数
  - 比较运算-用于张量上的数值谓词求值的函数，例如 `equal` 和 `max` 函数
  - 频谱运算-用于在频域中转换并在频域中操作的函数，例如 `stft` 和 `hamming_window` 函数
  - 其他运算-在向量上操作的特殊函数，例如 `cross` 函数或在矩阵上的 `trace` 函数
  - BLAS 和 LAPACK 操作-遵循 BLAS (基本线性代数子程序) 规范的函数，用于标量、向量-矢量、矩阵-矢量和矩阵-矩阵运算
  - 随机采样操作-通过从概率分布中随机抽取值来生成值的函数 (例如 `randn` 和 `normal`)
  - 序列化操作-用于保存和加载张量的函数 (例如 `load` 和 `save` 函数)
  - 并行操作-用于控制并行 CPU 执行的线程数的函数，例如 `set_num_threads` 函数
- 使用常规张量 API 很有用。本章应提供进行这种交互式探索的所有先决条件。

## 练习

- 从 `list(range(9))` 创建一个张量 `a`。预测然后检查一下尺寸、偏移量和步长。
- 创建一个张量 `b = a.view(3,3)`。`b[1,1]` 的值是多少？
- 创建一个张量 `c = b[1:,1:]`。预测然后检查一下尺寸、偏移量和步长。
- 选择一个数学运算，例如求余弦或求平方根。您可以在 `torch` 库中找到相应的函数吗？
- 你的函数是否有可以原地运行的版本？

## 总结

- 神经网络将浮点表示形式转换为其他浮点表示形式，起始和结束的表示形式通常是人类可以解释的。中间的表示则不具备可解释性。
- 这些浮点表示存储在张量中。
- 张量是多维数组，也是 PyTorch 中的基本数据结构。
- PyTorch 具有用于张量创建和操作以及数学运算的综合标准库。
- 张量可以序列化到磁盘并重新加载回来。
- PyTorch 中的所有张量操作都可以在 CPU 和 GPU 上执行，而无需更改代码。
- PyTorch 使用结尾的下划线表示函数在张量上原地运行（例如 `Tensor.sqrt_` 函数）。

## 3 使用张量对真实世界数据的表示

本章覆盖内容有：

- 将不同类型的真实世界数据表示为 PyTorch 张量
- 处理各种数据类型，包括电子表格、时间序列、文本、图像和医学成像
- 从文件加载数据
- 将数据转换为张量

张量是 PyTorch 中数据的构建块。神经网络将张量作为输入并产生张量作为输出。实际上，神经网络内以及优化期间的所有操作都是张量之间的操作，而神经网络中的所有参数（例如权重和偏置）都是张量。掌握如何在张量上执行操作并有效索引它们是成功使用 PyTorch 之类的工具的关键。现在您已经了解了张量的基本知识，您对它们的熟练性将会增强。

我们现在可以解决一个问题：您如何获取一条数据、一段视频或一段文本，并用张量表示它，并以适合于训练深度学习模型的方式做到这一点？

您将在本章中学到问题的答案。我们涵盖了不同类型的数据，并向您展示了如何使它们表示为张量。然后，我们向您展示如何从最常见的磁盘格式加载数据，并了解这些数据类型的结构，以便您可以了解如何为训练神经网络做准备。通常，对于要解决的问题，原始数据的格式可能并不完美，因此您将有机会在一些更有趣的张量操作中练习张量操纵技巧。您将使用大量的图像和体积数据，因为这些数据类型很常见并且可以以书本格式很好地复制。我们还将介绍表格数据、时间序列和文本，这也是许多读者感兴趣的。

本章的每个部分都描述一种数据类型，并且每个部分都有其自己的数据集。尽管我们已经对本章进行了结构设计，以使每种数据类型都建立在上一章的基础上，但如果您愿意的话，也可以略过一点。

正如您在电子表格中所发现的那样，我们首先从关于葡萄酒数据的表格数据开始。接下来，我们转到有序表格数据，其中包含来自自行车共享程序的时间序列数据集。之后，我们向您展示如何使用 Jane Austen 的文本数据。文本数据保留了有序的方面，但是引入了将单词表示为数字数组的问题。因为一张图片价值一千个单词，所以我们演示了如何处理图像数据。最后，我们使用 3D 数组深入研究医学数据，该 3D 阵列表示包含患者解剖结构的体积。

在每个部分中，我们都将在深度学习研究人员开始的地方停止：在将数据提供给模型之前。我们鼓励您保留这些数据集。当您开始学习如何训练神经网络模型时，它们将是极好的材料。

## 3.1 表格式数据

您在机器学习工作中遇到的最简单的数据形式是电子表格、CSV（逗号分隔值）文件或在一个数据库中。无论使用哪种介质，此数据都是一个表，每个样本（或记录）包含一行，其中各列包含有关样本的一条信息。

首先，假设样本在表格中的显示顺序没有意义。在时间序列中，样本与时间维度相关，与时间序列不同，此类表是独立样本的集合。

列可能包含数值（例如特定位置的温度）或标签（例如表示样品属性的字符串（例如“蓝色”））。因此，表格数据通常不是同质的；不同的列没有相同的类型。您可能有一列显示苹果的重量，另一列在标签中编码其颜色。

另一方面，PyTorch 张量是同质的。其他数据科学程序包（例如 Pandas）具有数据框的概念，数据框是一个对象，它表示具有可命名化的异构列的数据集。相比之下，PyTorch 中的信息被编码为数字，通常为浮点数（尽管也支持整数类型）。数值编码是有意为之的，因为神经网络是将实数作为输入并通过连续应用矩阵乘法和非线性函数产生实数作为输出的数学实体。

因此，作为深度学习从业人员，您的第一项工作是在浮点数张量中编码异类、真实世界的数据，以供神经网络使用。

互联网上免费提供了大量表格数据集。

例如，请参见 <https://github.com/caesar0301/awesome-public-data>。

我们从有趣的事情开始：葡萄酒。葡萄酒质量数据集是可免费获得的表格，其中包含 vinho verde（葡萄牙北部的葡萄酒）样品的化学表征以及感官质量评分。您可以从 <https://archive.ics.uci.edu/ml/machine-learning-databases/winequality/> winequality-white.csv 下载白葡萄酒的数据集。为了方便起见，我们在使用 PyTorch Git 的深度学习仓库中的 data/p1ch4 /tabular-wine 下创建了数据集的副本。

该文件包含用逗号分隔的值的集合，这些值按 12 列组织，其前面带有包含列名称的标题行。前 11 列包含化学变量的值；最后一列包含从 0（最差）到 10（优秀）的感官质量得分。以下是按照在数据集中显示的顺序的列名：

```
fixed acidity  
volatile acidity  
citric acid  
residual sugar  
chlorides  
free sulfur dioxide  
total sulfur dioxide  
density  
pH  
sulphates  
alcohol  
quality
```

此数据集上可能的机器学习任务是仅通过化学表征来预测质量得分。不过，请放心——机器学习不会在短期内扼杀品酒活动。我们必须从某个地方获取训练数据！

如图 3.1 所示，您希望找到数据中化学列之一与质量列之间的关系。在这里，您期望看到随着硫磺减少，质量得到提高。

但是，在进行该观察之前，您需要一种比在文本编辑器中打开文件更有用的方式检查数据。我们将向您展示如何使用 Python 加载数据，然后将其转换为 PyTorch 张量。

Python 提供了几种用于快速加载 CSV 文件的选项。三种流行的选择是：

- Python 随附的 csv 模块
- NumPy
- Pandas

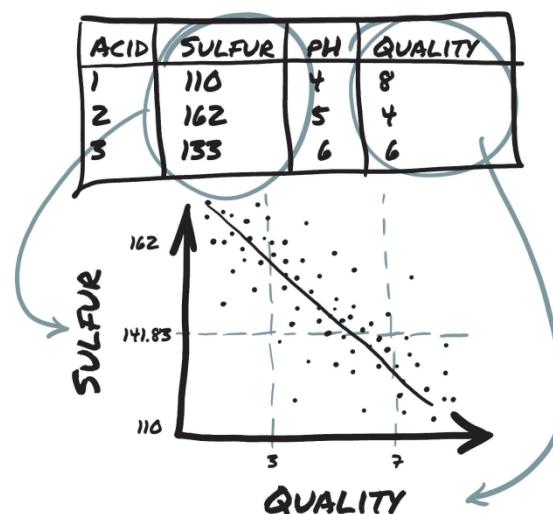


图 3.1 葡萄酒中硫与质量之间的关系

第三个选项是最省时和最节省内存的，但是我们将避免仅在加载文件时就在学习过程中引入额外的库。因为我们已经介绍了 NumPy，并且 PyTorch 具有出色的 NumPy 互操作性，所以

您将继续使用它。加载文件并将生成的 NumPy 数组转换为 PyTorch 张量，如下面的清单所示。

**Listing 3.1 code/p1ch4/1\_tabular\_wine.ipynb**

```
# In[2]:  
import csv  
wine_path = "../data/p1ch4/tabular-wine/winequality-white.csv"  
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32, delimiter=";",  
    skiprows=1)  
wineq_numpy  
  
# Out[2]:  
array([[ 7. ,  0.27,  0.36, ...,  0.45,  8.8 ,  6. ],  
       [ 6.3 ,  0.3 ,  0.34, ...,  0.49,  9.5 ,  6. ],  
       [ 8.1 ,  0.28,  0.4 , ...,  0.44, 10.1 ,  6. ],  
       ...,  
       [ 6.5 ,  0.24,  0.19, ...,  0.46,  9.4 ,  6. ],  
       [ 5.5 ,  0.29,  0.3 , ...,  0.38, 12.8 ,  7. ],  
       [ 6. ,  0.21,  0.38, ...,  0.32, 11.8 ,  6. ]], dtype=float32)
```

在这里，您规定了 2D 数组的类型（32 位浮点数）和用于分隔每行中值的定界符，并指出不应读取第一行，因为它包含列名。接下来，检查是否已读取所有数据，

```
# In[3]:  
col_list = next(csv.reader(open(wine_path), delimiter=';'))  
wineq_numpy.shape, col_list  
  
# Out[3]:  
(4898, 12),  
(['fixed acidity',  
 'volatile acidity',  
 'citric acid',  
 'residual sugar',  
 'chlorides',  
 'free sulfur dioxide',  
 'total sulfur dioxide',  
 'density',  
 'pH',  
 'sulphates',  
 'alcohol',  
 'quality'])
```

并继续将 NumPy 数组转换为 PyTorch 张量：

```
# In[4]:  
wineq = torch.from_numpy(wineq_numpy)  
  
wineq.shape, wineq.type()  
  
# Out[4]:  
(torch.Size([4898, 12]), 'torch.FloatTensor')
```

此时，您将拥有一个 `torch.FloatTensor`，其中包含所有列、包括最后一列，这是质量得分。

## 间隔、序数和分类值

在尝试理解数据时，应注意三种数值。

第一种是连续值。当用数字表示时，这些值是最直观的。它们是严格排序的，各个值之间的差异具有严格的含义。说明包 A 比包 B 重 2 公斤，或者包 B 比包 A 远 100 英里，这是有固

定含义的，无论包 A 重 3 公斤还是 10 公斤，或者包 B 是从 200 英里还是 2,000 英里。如果您要以单位进行计数或测量，则该值可能是连续值。

接下来是序数值。连续值的严格排序仍然保留，但是值之间的固定关系不再适用。一个很好的例子是订购小、中或大饮料，其中小映射到值 1，中到 2，大到 3。大酒杯比中酒杯大，就像 3 比 2 大一样，但是它并没有告诉您大多。如果将 1、2 和 3 转换为实际体积（例如 8、12 和 24 流体盎司），则这些值将切换为间隔值。重要的是要记住，除了对数值进行排序外，您无法对它们进行数学运算；尝试对大杯饮料= 3 和小杯饮料= 1 进行平均，并不会产生中杯饮料！最后，分类值既没有顺序也没有数值含义。这些值通常是对概率的枚举，并任意分配的数字。将水分配给 1，将咖啡分配给 2，将苏打分配给 3，将牛奶分配给 4 是一个很好的例子。将水排在前面，最后放牛奶没有逻辑意义。您只需要不同的值就可以区分它们。您可以将咖啡分配给 10，将牛奶分配给 -3，而没有明显变化（尽管分配值的范围是 0..N-1 在稍后讨论独热编码时会很有优势）。

您可以将分数视为连续变量，将其保留为实数，然后执行回归任务，也可以将其视为标签，然后尝试从分类任务中的化学分析中猜出该标签。在这两种方法中，通常都将分数从输入数据的张量中删除，并将其保存在单独的张量中，以便您可以将分数用作真值，而无需将其输入到模型中：

```
# In[5]:
data = wineq[:, :-1]    # Select all rows and all columns except the last.
data, data.shape

# Out[5]:
(tensor([[ 7.0000,  0.2700,  ...,  0.4500,  8.8000],
       [ 6.3000,  0.3000,  ...,  0.4900,  9.5000],
       ...,
       [ 5.5000,  0.2900,  ...,  0.3800, 12.8000],
       [ 6.0000,  0.2100,  ...,  0.3200, 11.8000]]), torch.Size([4898, 11]))
```

```
# In[6]:
target = wineq[:, -1]    # Select all rows and the last column.
target, target.shape
```

```
# Out[6]:
(tensor([6., 6., ..., 7., 6.]), torch.Size([4898]))
```

如果要在标签的张量中转换目标张量 target，则有两个选择，具体取决于策略或要使用分类数据的方式。一种选择是将标签视为分数的整数向量：

```
# In[7]:
target = wineq[:, -1].long()
target

# Out[7]:
tensor([6, 6, ..., 7, 6])
```

如果目标是字符串标签（例如葡萄酒的颜色），则为每个字符串分配一个整数将允许您采用相同的方法。

另一种方法是构建分数的独热编码——即，在十个元素的向量中对十个分数中的每一个进行编码，所有元素都设置为 0，但对于每个分数的不同索引处编码为 1。这样，得分 1 可以映射到向量 (1,0,0,0,0,0,0,0,0) ，得分 5 映射到 (0,0,0,0,1,0,0,0,0) 等。分数与非零元素的索引相对应的事实纯粹是偶然的；您可以重新打乱分配任务，从分类的角度来看，什

么都不会改变。

两种方法有明显的区别。将葡萄酒质量分数保持在分数的整数向量中会导致分数排序，这在这种情况下可能是适当的，因为分数 1 低于分数 4。这还导致分数之间存在一定距离。（例如，1 和 3 之间的距离与 2 和 4 之间的距离相同。）如果这符合您的数量，那就太好了。另一方面，如果分数是纯定性的（例如颜色），则独热编码更适合，因为不涉及隐含的顺序或距离。当整数分数之间的分数值（例如 2.4）对应用没有意义时（当得分非此即彼，没有中间值），独热编码也适用于定量得分。

您可以通过使用 `scatter_` 方法来实现独热编码，该方法将源张量中的值沿着作为参数提供的索引来填充张量。

```
# In[8]:
target_onehot = torch.zeros(target.shape[0], 10)

target_onehot.scatter_(1, target.unsqueeze(1), 1.0)

# Out[8]:
tensor([[0., 0., ..., 0., 0.],
        [0., 0., ..., 0., 0.],
        ...,
        [0., 0., ..., 0., 0.],
        [0., 0., ..., 0., 0.]])
```

现在看一下 `scatter_` 的作用。首先，请注意其名称以下划线结尾。PyTorch 中的此约定表示该方法不会返回新的张量，而是会在原地修改该张量。`scatter_` 的参数如下：

- 指定以下两个参数的维度
- 列张量，指示要散布的元素的索引
- 一个张量，包含要散布的元素或单个散布的标量（在这种情况下为 1）

换句话说，前面的调用可以这样理解：“对于每一行，获取目标标签的索引（在这种情况下，该索引与得分分数一致），并将其用作列索引以设置值 1.0。得到的结果是编码分类信息的张量。”

`scatter_` 函数的第二个参数，即索引张量，必须具有与分散后的张量相同的维数数量。由于 `target_onehot` 具有两个维度（4898x10），因此您需要使用 `unsqueeze` 函数为目标 `target` 添加一个额外的空维度：

```
# In[9]:
target_unsqueezed = target.unsqueeze(1)
target_unsqueezed

# Out[9]:
```

```
tensor([[6],  
       [6],  
       ...,  
       [7],  
       [6]])
```

`unsqueeze` 函数的调用增加了一个单独维度，从 4898 个元素的 1D 张量到大小为 (4898x1) 的 2D 张量，而不更改其内容。没有添加元素；您决定使用额外的索引来访问元素。也就是说，您对于 `target` 的第一个元素使用 `target[0]` 来访问，并使用 `target_unsqueezed[0,0]` 来索引 `unsqueezed` 处理后的对应对象的第一个元素。

尽管 PyTorch 允许您在训练神经网络时直接将类索引用作目标。但是，如果您想将分数用作网络的分类输入，则必须将其转换为独热编码的张量。

现在回到数据 `data` 张量，其中包含与化学分析相关的 11 个变量。您可以使用 PyTorch Tensor API 中的函数以张量形式处理数据。首先，获取每列的均值和标准差：

```
# In[10]:  
data_mean = torch.mean(data, dim=0)  
data_mean  
  
# Out[10]:  
tensor([6.8548e+00, 2.7824e-01, 3.3419e-01, 6.3914e+00, 4.5772e-02,  
       3.5308e+01,  
       1.3836e+02, 9.9403e-01, 3.1883e+00, 4.8985e-01, 1.0514e+01])  
# In[11]:  
data_var = torch.var(data, dim=0)  
data_var  
  
# Out[11]:  
tensor([7.1211e-01, 1.0160e-02, 1.4646e-02, 2.5726e+01, 4.7733e-04,  
       2.8924e+02,  
       1.8061e+03, 8.9455e-06, 2.2801e-02, 1.3025e-02, 1.5144e+00])
```

在这种情况下，`dim = 0` 表示沿维数 0 进行缩减。在这一点上，您可以通过减去平均值并除以标准偏差来对数据进行归一化，这有助于学习过程。

```
# In[12]:  
data_normalized = (data - data_mean) / torch.sqrt(data_var)  
data_normalized  
  
# Out[12]:  
tensor([[ 1.7209e-01, -8.1764e-02, ..., -3.4914e-01, -1.3930e+00],  
       [-6.5743e-01,  2.1587e-01, ...,  1.3467e-03, -8.2418e-01],  
       ...,  
       [-1.6054e+00,  1.1666e-01, ..., -9.6250e-01,  1.8574e+00],  
       [-1.0129e+00, -6.7703e-01, ..., -1.4882e+00,  1.0448e+00]])
```

接下来，着眼于数据以寻找一种简单的方法来一眼分辨好酒和坏酒。首先，使用 `torch.le` 函数确定目标 `target` 中哪些行对应的分数小于或等于 3：

```
# In[13]:
bad_indexes = torch.le(target, 3)
bad_indexes.shape, bad_indexes.dtype, bad_indexes.sum()

# Out[13]:
(torch.Size([4898]), torch.uint8, tensor(20))
```

请注意，`bad_indexes` 条目中只有 20 个被设置为 1！通过利用 PyTorch 中称为 advanced indexing 高级索引的功能，可以使用二进制张量来索引数据张量 `data`。此张量实际上将数据筛选为仅与索引张量中的 1 对应的项目（或行）。`bad_indexes` 张量具有与目标 `target` 相同的形状，其值是 0 或 1，具体取决于阈值与原始目标张量中每个元素之间比较的结果：

```
# In[14]:
bad_data = data[bad_indexes]
bad_data.shape

# Out[14]:
torch.Size([20, 11])
```

请注意，新的 `bad_data` 张量具有 20 行，与 `bad_indexes` 张量中带有 1 的行数相同。它保留了所有的 11 列。

现在，您可以开始获取有关葡萄酒的信息，这些葡萄酒分为好、中和坏三类。对每一列取`.mean()`函数：

```
# In[15]:
bad_data = data[torch.le(target, 3)]
mid_data = data[torch.gt(target, 3) & torch.lt(target, 7)] ←
good_data = data[torch.ge(target, 7)]

bad_mean = torch.mean(bad_data, dim=0)
mid_mean = torch.mean(mid_data, dim=0)
good_mean = torch.mean(good_data, dim=0)

for i, args in enumerate(zip(col_list, bad_mean, mid_mean, good_mean)):
    print('{:2} {:20} {:6.2f} {:6.2f} {:6.2f}'.format(i, *args))

# Out[15]:
0 fixed acidity      7.60   6.89   6.73
1 volatile acidity   0.33   0.28   0.27
2 citric acid        0.34   0.34   0.33
3 residual sugar     6.39   6.71   5.26
4 chlorides          0.05   0.05   0.04
5 free sulfur dioxide 53.33  35.42  34.55
6 total sulfur dioxide 170.60 141.83 125.25
7 density            0.99   0.99   0.99
8 pH                 3.19   3.18   3.22
9 sulphates          0.47   0.49   0.50
10 alcohol           10.34  10.26  11.42
```

For numpy arrays and  
PyTorch tensors, the &  
operator does a logical  
and operation.

看来您正在这里做某事。乍看之下，相比其他差异，劣质葡萄酒似乎具有较高的总二氧化硫含量。您可以使用二氧化硫总量的阈值来作为区分好酒和差酒的粗略评判标准。现在获取总二氧化硫列上的值低于您之前计算的中点的索引，如下所示：

```
# In[16]:  
total_sulfur_threshold = 141.83  
total_sulfur_data = data[:,6]  
predicted_indexes = torch.lt(total_sulfur_data, total_sulfur_threshold)  
  
predicted_indexes.shape, predicted_indexes.dtype, predicted_indexes.sum()  
  
# Out[16]:  
(torch.Size([4898]), torch.uint8, tensor(2727))
```

您的阈值意味着，一半以上的葡萄酒将是高品质的。

接下来，您需要获取优质葡萄酒的索引：

```
# In[17]:  
actual_indexes = torch.gt(target, 5)  
  
actual_indexes.shape, actual_indexes.dtype, actual_indexes.sum()  
  
# Out[17]:  
(torch.Size([4898]), torch.uint8, tensor(3258))
```

由于您拥有的优质葡萄酒比您阈值预计的多约 500 种，因此您已经有确凿的证据证明该阈值并不完美。

现在，您需要查看您的预测与实际排名的匹配程度。在预测索引和良好索引之间执行逻辑 `and` 运算（请记住，每个索引是 0 和 1 组成的数组），并使用葡萄酒的交叉一致程度来确定您的表现如何：

```
# In[18]:  
n_matches = torch.sum(actual_indexes & predicted_indexes).item()  
n_predicted = torch.sum(predicted_indexes).item()  
n_actual = torch.sum(actual_indexes).item()  
  
n_matches, n_matches / n_predicted, n_matches / n_actual  
  
# Out[18]:  
(2018, 0.74000733406674, 0.6193984039287906)
```

您有大约 2,000 种葡萄酒！因为您预测有 2700 种葡萄酒，所以如果您预测葡萄酒是高质量的，那么有 74% 的可能性预测正确。不幸的是，您有 3200 种优质葡萄酒，仅能识别出 61%。好吧，我们猜您已经签了名。该结果仅比随机结果好。

当然，这个例子很幼稚。您可以肯定地知道，多个变量会影响葡萄酒的质量，这些变量的值与结果之间的关系（可能是实际分数，而不是二值化版本）可能比简单的阈值更为复杂，而不仅仅是单个值。

实际上，简单的神经网络将克服所有这些限制，就像许多其他基本的机器学习方法一样。在完成第 5 章和第 6 章后，您将拥有解决此问题的工具，在这当中，您将从头开始构建第一个神经网络。

## 3.2 时间序列

在上一节中，我们介绍了如何表示以平面表组织的数据。如前所述，表中的每一行都独立于其他行；他们的顺序没有任何影响。同样的，编码信息的列对于哪些行在前和哪些行在后也没有影响。

回到葡萄酒数据集，您可能有一个年份 Year 栏，可以查看葡萄酒质量如何逐年变化。（很遗憾，我们手头没有此类数据，但我们正在努力手动逐瓶收集数据样本。）

同时，我们将切换到另一个有趣的数据集：华盛顿特区的自行车共享系统中的数据，报告了首都自行车共享系统中 2011 年至 2012 年之间每小时的租用自行车计数以及相应的天气和季节性信息。（<https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset>）

目标是获取平坦的 2D 数据集并将其转换为 3D 数据集，如图 3.2 所示。

在源数据中，每行是一个单独的小时数据（图 3.2 显示了该数据的转置版本以更好地适合打印页面）。我们要更改为行每小时的组织，以便使一个轴以每个索引增量一天的速度增加，而另一个轴则表示一天中的小时（与日期无关）。第三个轴是不同的数据列（天气、温度等）。

加载数据，如下面的清单所示。

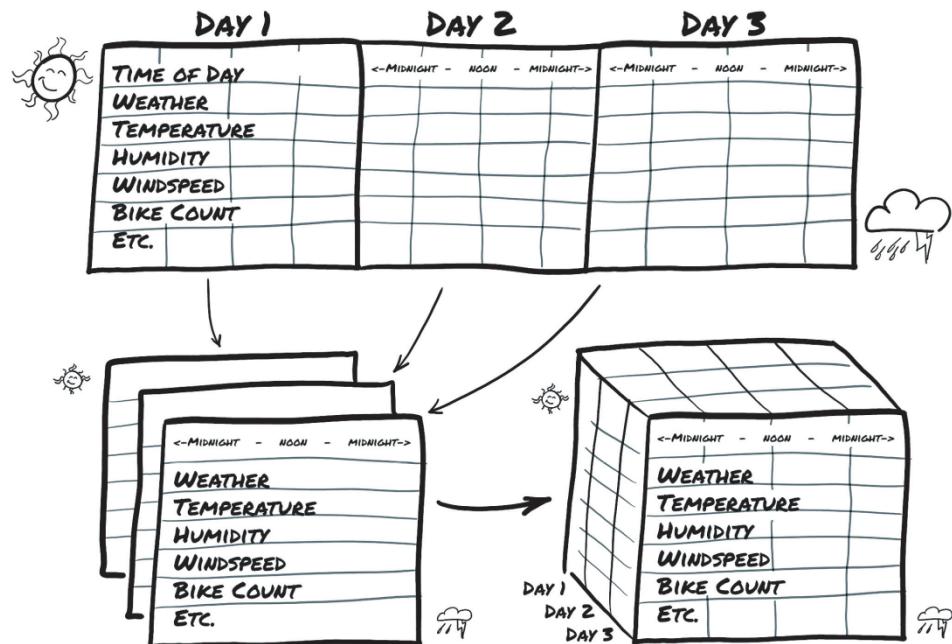


图 3.2 通过将每个样本的日期和小时分为不同的轴，将一维多通道数据集转换为二维多通道数据集

### Listing 3.2 code/p1ch4/2\_time\_series\_bikes.ipynb

```
# In[2]:  
bikes_numpy = np.loadtxt("../data/p1ch4/bike-sharing-data set/hour-  
fixed.csv",  
                        dtype=np.float32,  
                        delimiter=",",  
                        skiprows=1,  
                        converters={1: lambda x: float(x[8:10])}) ←  
bikes = torch.from_numpy(bikes_numpy)           Convert date strings to numbers  
bikes                                         corresponding to the day  
                                                of the month in column 1.  
  
# Out [2]:  
tensor([[1.0000e+00, 1.0000e+00, ..., 1.3000e+01, 1.6000e+01],  
       [2.0000e+00, 1.0000e+00, ..., 3.2000e+01, 4.0000e+01],  
       ...,  
       [1.7378e+04, 3.1000e+01, ..., 4.8000e+01, 6.1000e+01],  
       [1.7379e+04, 3.1000e+01, ..., 3.7000e+01, 4.9000e+01]])
```

对于每小时，数据集报告以下变量：

```
instant      # index of record  
day          # day of month  
season       # season (1: spring, 2: summer, 3: fall, 4: winter)  
yr           # year (0: 2011, 1: 2012)  
mnth         # month (1 to 12)  
hr           # hour (0 to 23)  
holiday      # holiday status  
weekday      # day of the week  
workingday   # working day status  
weathersit   # weather situation  
temp          # temperature in C  
atemp         # perceived temperature in C  
hum           # humidity  
windspeed    # windspeed  
casual        # number of causal users  
registered   # number of registered users  
cnt           # count of rental bikes
```

在这样的时间序列数据集中，行表示连续的时间点：按其排序的维度。当然，您可以将每一行视为独立的，并尝试根据例如一天中的特定时间来预测自行车的循环数量，而不考虑先前发生的情况。

但是，这种排序的存在使您有机会利用跨时间的因果关系。例如，您可以根据下雨的时间来预测自行车骑行次数。暂时，您将专注于学习如何将自行车共享数据集转换为神经网络可以提取的固定大小的数据块。

该神经网络模型需要查看每个数量的值序列，例如乘车次数、一天中的时间、温度和天气情况，因此是 N 个并行的大小为 C 的序列。C 代表通道 channel，在神经网络术语中，它是与此处所示的一维数据列相同。N 维度代表时间轴，这里是每小时一个条目。

您可能希望在更宽的观察期内（例如天）分解 2 年数据集。这样，您将拥有 N 个（样本数）长度为 L 的 C 序列的集合。换句话说，您的时间序列数据集是有 3 个维度的张量，形状为  $N \times C \times L$ 。C 仍然是 17 个通道，而 L 则是 24 个通道对应一天中个小时。没有必要特别说明为什么我们必须使用 24 小时的时间段，尽管一般的日常节奏可能会给我们提供可用于预测的模式。如果需要，我们可以改为使用  $7 * 24 = 168$  小时的块按周划分。

现在回到您的自行车共享数据集。第一列是索引（数据的全局顺序）；第二个是日期；

第六列是一天中的时间。您拥有创建行驶计数和其他外来变量的每日序列数据集所需的一切。您的数据集已经排序，但是如果没有排序，则可以在其上使用 `torch.sort` 进行适当排序。

注意：您在此处使用的文件版本 `hour-fixed.csv` 已进行了一些处理，以包含原始数据集中缺少的行。我们假设丢失的时间上有零个自行车处于活动状态（通常是清晨的时间）。

要获取每日工作时间数据集，您要做的就是每 24 小时查看同一张量。看一下您的自行车张量 `bikes` 的形状和步长：

```
# In[3]:  
bikes.shape, bikes.stride()  
  
# Out[3]:  
(torch.Size([17520, 17]), (17, 1))
```

这是 17,520 小时，共 17 列。现在，将数据重构为具有三个轴（天、小时，然后是 17 列）：

```
# In[4]:  
daily_bikes = bikes.view(-1, 24, bikes.shape[1])  
daily_bikes.shape, daily_bikes.stride()  
  
# Out[4]:  
(torch.Size([730, 24, 17]), (408, 17, 1))
```

这里发生了什么？首先，`bikes.shape[1]` 是 17，它是 `bikes` 张量中的列数。但是代码的真正关键之处在于调用视图 `view`，这很重要：它改变了张量查看在相同存储中的数据的方式。

在张量上调用 `view` 将返回一个新的张量，该张量将更改维数和步长信息，而不会更改存储 `storage`。从而，您可以以零成本重新布置张量，因为根本没有数据被复制。您的 `view` 调用要求您为返回的张量提供新的形状。将 -1 用作占位符，表示“给定其他维度和元素的原始数量，剩下多少。”

请记住，`Storage` 是数字的连续线性容器，在这种情况下为浮点数。您的 `bikes` 张量具有在相应存储区中一个接一个地存储的行，这已由之前对 `bikes.stride()` 调用的输出得到确认。

对于 `daily_bikes`，步长告诉您，沿小时维度（第二个维度）前进 1 个位置需要您在存储中的位置前进 17 个位置（或一组列），而沿白天维度（第一个）前进需要您进行以下操作：在存储中前进的元素数等于行的长度的乘以 24（此处为 408，即  $17 * 24$ ）。

最右边的维度是原始数据集中的列数。在中间维度中，您将时间分为 24 个连续小时的块。换句话说，您现在每天有 C 个通道的 N 个 L 小时的序列。为了获得所需的  $N \times C \times L$  顺序，您需要转置张量：

```
# In[5]:  
daily_bikes = daily_bikes.transpose(1, 2)  
daily_bikes.shape, daily_bikes.stride()  
  
# Out[5]:  
(torch.Size([730, 17, 24]), (408, 1, 17))
```

前面我们提到天气条件变量是序数。实际上，它有 4 个级别：1 表示最佳天气，4 表示最坏天气。您可以将此变量视为分类变量，将级别解释为标签或连续变量。如果选择分类变量，则将变量转换为独热编码的向量，并将列与数据集连接起来。为了使数据渲染更容易，现在暂时限制为第一天。首先，初始化一个零填充矩阵，其行数等于一天中的小时数，列数等于天气水平的数：

```
# In[6]:  
first_day = bikes[:24].long()  
weather_onehot = torch.zeros(first_day.shape[0], 4)  
first_day[:, 9]  
  
# Out[6]:  
tensor([1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 2, 2, 2])
```

然后根据每一行的对应级别将它们分散到我们的矩阵中。请记住，之前曾使用 `unsqueeze` 来添加单个维度：

```
# In[7]:  
weather_onehot.scatter_()  
    dim=1,  
    index=first_day[:, 9].unsqueeze(1) - 1, ←  
    value=1.0)  
  
# Out[7]:
```

You're decreasing the values by 1 because  
the weather situation ranges from 1 to 4,  
whereas indices are 0-based.

```
tensor([[1., 0., 0., 0.],  
       [1., 0., 0., 0.],  
       ...,  
       [0., 1., 0., 0.],  
       [0., 1., 0., 0.]])
```

一天从天气 1 开始，到天气 2 结束，所以这似乎是正确的。

最后，使用 `cat` 函数将矩阵连接到原始数据集。查看您的第一个结果：

```
# In[8]:  
torch.cat((bikes[:24], weather_onehot), 1) [:1]  
  
# Out[8]:  
tensor([[ 1.0000,   1.0000,   1.0000,   0.0000,   1.0000,   0.0000,   0.0000,  
         6.0000,  
        0.0000,   1.0000,   0.2400,   0.2879,   0.8100,   0.0000,   3.0000, 13.0000,  
       16.0000,   1.0000,   0.0000,   0.0000,   0.0000]])
```

在这里，您规定了原始 `bikes` 数据集和独热编码的天气状况矩阵将沿着列维（例如 1）连接在一起。换句话说，将两个数据集的列堆叠在一起，或者将新的独热编码列添加到原始数据

集。为了使 `cat` 函数成功调用，张量在其他尺寸（在这种情况下为行尺寸）上必须具有相同的大小。

请注意，您最后的新四列分别是 1、0、0、0，这正是您期望的天气值 1。

您可以使用经过重塑的 `daily_bikes` 张量完成相同的操作。请记住，它的形状是  $(B, C, L)$ ，其中  $L=24$ 。首先，创建零张量，其  $B$  和  $L$  相同，但附加列数  $C$ ：

```
# In[9]:  
daily_weather_onehot = torch.zeros(daily_bikes.shape[0], 4,  
daily_bikes.shape[2])  
daily_weather_onehot.shape  
  
# Out[9]:  
torch.Size([730, 4, 24])
```

然后将独热编码散布到张量中的 C 维中。由于操作是在原地执行的，因此仅张量的内容会更改：

```
# In[10]:  
daily_weather_onehot.scatter_(1, daily_bikes[:, 9, :].long().unsqueeze(1) - 1,  
    1.0)  
daily_weather_onehot.shape  
  
# Out[10]:  
torch.Size([730, 4, 24])
```

沿 C 维连接；

```
# In[11]:  
daily_bikes = torch.cat((daily_bikes, daily_weather_onehot), dim=1)
```

前面我们提到，这种方法不是处理天气情况变量的唯一方法。实际上，其标签具有序数关系，因此您可以假装它们是连续变量的特殊值。您可以转换变量，使其从 0.0 到 1.0 运行。

```
# In[12]:  
daily_bikes[:, 9, :] = (daily_bikes[:, 9, :] - 1.0) / 3.0
```

正如我们在第 4.1 节中提到的，将变量重新缩放为  $[0.0, 1.0]$  间隔或  $[-1.0, 1.0]$  间隔是您要对所有定量变量（例如温度（数据集中第 10 列））所做的。稍后您会看到原因；现在，我们说这对训练过程很有帮助。

你有多种重新缩放变量的可能性。你可以将其范围映射到[0.0, 1.0]。

```
# In[13]:  
temp = daily_bikes[:, 10, :]  
temp_min = torch.min(temp)  
temp_max = torch.max(temp)  
daily_bikes[:, 10, :] = (daily_bikes[:, 10, :] - temp_min) / (temp_max -  
    temp_min)
```

或减去平均值并除以标准偏差：

```
# In[14]:  
temp = daily_bikes[:, 10, :]  
daily_bikes[:, 10, :] = (daily_bikes[:, 10, :] - torch.mean(temp)) /  
    torch.std(temp)
```

在后一种情况下，变量具有的均值为零和标准偏差为 1。如果从高斯分布中提取变量，则 68 % 的样本将位于 [-1.0, 1.0] 区间内。

非常棒——您已经建立了另一个不错的数据集，以后将使用。目前，仅了解一个时间序列的布局以及如何将数据整理成网络可消化的形式这一点很重要。

其他类型的数据看起来像时间序列，因为存在严格的排序。该类别中最重要的两类是文本和音频。

### 3.3 文本

深度学习已席卷自然语言处理（NLP）领域，特别是通过使用反复消耗新输入和先前模型输出的组合的模型。这些模型称为递归神经网络，并且已成功应用于文本分类、文本生成和自动翻译系统。以前的 NLP 工作量的特点是复杂的多级管道，其中包括编码语言语法的规则。

（Nadkarni et al., “Natural language processing: an introduction”. JAMIA

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328>）

（Wikipedia entry for natural language processing:

[https://en.wikipedia.org/wiki/Natural-language\\_processing](https://en.wikipedia.org/wiki/Natural-language_processing)）现在，最新的工作是从头开始在大型语料库上端到端训练网络，让这些规则从数据中浮现出来。在过去的几年中，互联网上作为服务使用的最常用的自动翻译系统是基于深度学习的。

在本章中，您的目标是将文本转换为神经网络可以处理的内容，就像前面的情况一样，它是数字的张量。如果可以这样做，以后再为您的文本处理工作选择正确的体系结构，则可以使用 PyTorch 进行 NLP。您马上就会看到此功能的强大之处：如果以正确的形式描述问题，则可以使用相同的 PyTorch 工具在不同域中的任务上实现最新性能。这项工作的第一部分是重构数据。

网络在两个级别上对文本进行操作：在字符级别上，一次处理一个字符，在单词级别上，单个单词是网络中最细粒度的实体。无论是在字符级别还是在单词级别进行操作，用于将文本信息编码为张量形式的技术都是相同的。这种技术绝非魔术。您之前偶然碰到过它。它就是独热编码。

从字符级示例开始。首先，获取一些文本进行处理。Gutenberg 项目 (<http://www.gutenberg.org>) 是一个了不起的资源，它是一项志愿者工作，可以对文化作品进行数字化和存档，并使其以开放格式免费提供，包括纯文本文件。如果您的目标是更大型的语料库，那么 Wikipedia 语料库就非常出色：它是 Wikipedia 文章的完整集合，其中包含 19

亿个单词和超过 440 万条文章。您可以在英语语料库网站上找到其他几种语料库。  
(<https://www.english-corpora.org>)

从 Gutenberg 项目网站上 (<http://www.gutenberg.org/files/1342/1342-0.txt>) 加载 Jane Austen 的《傲慢与偏见》。保存文件并读入文件，如下面的清单所示。

### Listing 3.3 code/p1ch4/3\_text\_jane\_austin.ipynb

```
# In[2]:  
with open('../data/p1ch4/jane-austin/1342-0.txt', encoding='utf8') as f:  
    text = f.read()
```

在继续之前，您需要先处理一个细节：编码。编码是一个广泛的主题，因此我们现在要做的就是接触它。每个书面字符都由一个代码表示，该代码是一段适当长度的位，可以唯一地标识每个字符。最简单的这种编码是 ASCII（美国信息交换标准代码），其历史可以追溯到 1960 年代。ASCII 使用 128 个整数对 128 个字符进行编码。例如，字母 a 对应于二进制 1100001 或十进制 97；字母 b 对应于二进制 1100010 或十进制 98，依此类推。编码适合 8 位，这在 1965 年是一个很大的收获。

注意：显然，128 个字符不足以说明正确表示除英语以外的其他书面文字所需的所有字形、重音、连字和其他功能。为此，已经开发了其他编码，使用更多的位作为用于更广泛的字符的代码。更大范围的字符被标准化为 Unicode，它将所有已知字符映射为数字，这些数字的位表示由特定的编码提供。流行的编码包括 UTF-8、UTF-16 和 UTF-32，其中数字是 8 位、16 位或 32 位整数的序列。Python 3.x 中的字符串是 Unicode 字符串。

您将对字符进行独热编码，以将独热编码限制为对分析的文本有用的字符集。在这种情况下，因为您以英语加载了文本，所以使用 ASCII 并处理少量编码非常安全。您还可以将所有字符都小写以减少编码中的字符数。同样，您可以筛选出与预期文本类型无关的标点符号、数字和其他字符，这可能或可能不会对神经网络产生实际影响，具体取决于手头的任务。

此时，您需要解析文本中的字符，并为每个字符提供独热编码。每个字符将由一个长度等于编码中字符数的向量表示。除与编码中字符位置相对应的索引处的为 1 以外，该矢量将包含全零。

首先，将您的文本 text 分成几行，然后关注于任意选择的一行：

```
# In[3]:  
lines = text.split('\n')  
line = lines[200]  
line  
  
# Out[3]:  
'"Impossible, Mr. Bennet, impossible, when I am not acquainted with him'
```

创建一个张量，该张量可以包含整行字符总数的独热编码：

```
# In[4]:  
letter_tensor = torch.zeros(len(line), 128) ← I28 hardcoded due to  
letter_tensor.shape  
  
# Out[4]:  
torch.Size([70, 128])
```

请注意，letter\_tensor 每行包含一个独热编码的字符。现在在每一行上的正确位置置 1，以便每一行代表正确的字符。设置 1 的索引对应于字符在编码中的索引：

```
# In[5]:  
for i, letter in enumerate(line.lower().strip()):  
    letter_index = ord(letter) if ord(letter) < 128 else 0 ← The text uses directional double quotes,  
    letter_tensor[i][letter_index] = 1  
    which aren't valid ASCII, so screen them out here.
```

您已经将句子独热编码成神经网络可以消化的表示形式。您可以通过沿张量的行建立词汇表和独热编码的句子（单词序列）来以相同的方式进行单词级编码。由于词汇表包含许多单词，因此此方法会产生可能不实用的宽编码向量。在本章的后面，您将看到一种通过使用嵌入的方式来表示单词级别文本的更有效方法。现在，坚持使用独热编码以查看会发生什么。

定义 clean\_words 函数，它接受文本并将其返回其小写字母形式并去除标点符号。在线“Impossible, Mr. Bennet”上调用它时，会得到以下信息：

```
# In[6]:  
def clean_words(input_str):  
    punctuation = '.', ',', ';', ':', '!', '?', '-'  
    word_list = input_str.lower().replace('\n', ' ').split()  
    word_list = [word.strip(punctuation) for word in word_list]  
    return word_list  
  
words_in_line = clean_words(line)  
line, words_in_line  
  
# Out[6]:  
('impossible, Mr. Bennet, impossible, when I am not acquainted with him',  
['impossible',  
'mr',  
'bennet',  
'impossible',  
'when',  
'i',  
'am',  
'not',  
'acquainted',  
'with',  
'him'])
```

接下来，在编码中建立单词到索引的映射：

```
# In[7]:  
word_list = sorted(set(clean_words(text)))  
word2index_dict = {word: i for (i, word) in enumerate(word_list)}  
  
len(word2index_dict), word2index_dict['impossible']  
  
# Out[7]:  
(7261, 3394)
```

请注意，`all_words` 现在是一个字典，其中单词作为键，而整数作为值。独热编码时，您将使用该词典有效地找到单词的索引。

现在专注于您的句子。将其分解为单词并对其进行独热编码（即，对每个单词使用一个独热编码矢量填充张量）。创建一个空向量，并为句子中的单词分配一个独热编码值：

```
# In[8] :
word_tensor = torch.zeros(len(words_in_line), len(word2index_dict))
for i, word in enumerate(words_in_line):
```

```
    word_index = word2index_dict[word]
    word_tensor[i][word_index] = 1
    print('{:2} {:4} {}'.format(i, word_index, word))

print(word_tensor.shape)

# Out[8] :
0 3394 impossible
1 4305 mr
2 813 bennet
3 3394 impossible
4 7078 when
5 3315 i
6 415 am
7 4436 not
8 239 acquainted
9 7148 with
10 3215 him
torch.Size([11, 7261])
```

此时，张量 `tensor` 表示长度为 11 的一个句子，其长度为 7261，即字典中的单词数。

### 3.3.1 文字嵌入

独热编码是一种用于在张量中表示的分类数据的有用技术。就像您可能预料到的那样，当要编码的项目数实际上不受限制时，单语编码开始崩溃，例如语料库中的单词。在一本书中，您将会有超过 7,000 个条目！

当然，您可以做一些工作来对单词进行重复数据删除、压缩替代拼写、将过去和将来时态合并为单个标记，等等。尽管如此，通用的英语编码仍将是巨大的。更糟糕的是，每次遇到一个新单词时，都必须在向量中添加一个新列，这意味着要在模型中添加一组新的权重以针对该新词汇的输入，从训练的角度来看这将是痛苦的。

如何将编码压缩到更易于管理的大小，并限制大小增长？嗯，可以使用浮点数向量，而不是使用多个零和一个 1 的向量。举例来说，一个 100 个浮点数的向量确实可以表示大量的单词。诀窍是找到一种有效的方法，以一种有助于下游学习的方式将单个单词映射到此 100 维空间。此技术称为嵌入。

原则上，您可以遍历词汇表，并为每个单词生成一个 100 个随机浮点数的集合。这种方法行之有效，因为您可以将大量词汇塞入 100 个数字中，但是它会放弃基于含义或上下文的单词之间距离的任何概念。使用这个单词嵌入的模型将必须处理其输入向量中的少量结构。

理想的解决方案是以这种方式生成嵌入，即在相似语境中使用的单词映射到嵌入的附近区域。

如果要手工设计解决此问题的方法，则可以决定通过沿轴映射基本名词和形容词来构建嵌入空间。您可以生成一个二维空间，在该空间中，轴映射名词“水果”到（0.0-0.33），“花”到（0.33-0.66）和“狗”到（0.66-1.0）以及形容词“红色的”到（0.0-0.2），“橙色的”到（0.2-0.4），“黄色的”到（0.4-0.6），“白色的”到（0.6-0.8）和“棕色的”到（0.8-1.0）。您现在的目标是取下实际的水果、花朵和狗，并在嵌入中布置它们。

开始嵌入单词时，可以将“苹果”映射到“水果”和“红色的”象限中的数字。同样，您可以轻松地映射“橘子”，“柠檬”，“荔枝”和“猕猴桃”（依次列出五颜六色的水果）。然后，您可以从花开始，分配“玫瑰”，“罂粟”，“水仙花”，“百合”等等。好吧，没有很多棕色花。好吧，“向日葵”可以获得“花”，“黄色”和“棕色”，而“雏菊”可以获得“花”，“白色”和“黄色”。也许您应该更新“猕猴桃”以将其映射到“水果”，“棕色”和“绿色”附近。对于狗和颜色，您可以将“美洲赤狗”、“狐狸”嵌入到“橙色”，“金毛”、“贵宾犬”到“白色”等等。然而大多数种类的狗都是“棕色的”。

尽管对于大型语料库而言，手动进行此映射并不可行，但您应注意，尽管嵌入大小为2，但您描述了除基数8之外的15个不同的单词，如果您花一些时间，可能还会塞入更多有创造力的单词。

您可能已经猜到了，这种工作可以自动化。通过处理有机文本的语料库，您可以生成与此类似的嵌入。主要区别在于嵌入向量具有100到1,000个元素，并且轴并不直接映射到概念，但是概念上相似的单词映射到嵌入空间的相邻区域，该区域的轴是任意浮点维度。

尽管使用的确切算法（其中一个例子是<https://en.wikipedia.org/wiki/Word2vec>）稍稍超出了我们在此要关注的范围，但我们想提一提的是，嵌入通常是通过使用神经网络生成的，试图从邻近词（上下文）中预测一个词。在这种情况下，您可以从独热编码的单词开始，然后使用一个（通常相当浅的）神经网络来生成嵌入。当嵌入可用时，您可以将其用于下游任务。

所产生的嵌入的一个有趣方面是，相似的词不仅会聚在一起，而且与其他词的空间关系也会一致。如果要为“apple”使用嵌入向量，并开始为其他词向量做加法和做减法，则可以开始进行类比，例如apple-red-sweet+yellow+sour，最后得到一个类似于一个代表“柠檬”向量。

这里我们不会使用文本嵌入，但是当必须用数值向量表示集合中的大量条目时，它们是必不可少的工具。

## 3.4 图像

卷积神经网络的引入彻底改变了计算机视觉（

[https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network#History](https://en.wikipedia.org/wiki/Convolutional_neural_network#History)），基于图像的系统自此获

得了新的功能。通过对端到端网络进行成对的输入和期望输出示例的训练，需要高度优化的算法构建块的复杂流水线问题可以以前所未有的性能水平解决。为了参与这一革命，您需要能够从常见的图像格式加载图像，然后将数据转换为张量表示，该张量表示以 PyTorch 期望的方式排列了图像的各个部分。

图像表示为标量的集合，这些标量以规则的网格排列，并具有高度和宽度（以像素为单位）。每个网格点（像素）可能只有一个标量，可以表示为灰度图像，每个网格点可能有多个标量，它们通常代表不同的颜色或不同的特征，例如深度相机的深度。

例如，在消费类相机中，代表单个像素值的标量通常使用 8 位整数编码。在医学、科学和工业应用中，您常常会找到数值精度更高的像素，例如 12 位和 16 位。在像素编码有关物理特性（例如骨密度、温度或深度）的信息的情况下，此精度可提供更大的范围或更高的灵敏度。

您有几种将数字编码为颜色的方法（[https://en.wikipedia.org/wiki/Color\\_model](https://en.wikipedia.org/wiki/Color_model)）。最常见的是 RGB，它定义了一种具有三个数字的颜色，分别代表红色、绿色和蓝色的强度。您可以将颜色通道视为仅是所讨论颜色的灰度强度图，类似于您根据讨论的场景，通过一副纯红色太阳镜观察所看到的样子。图 3.3 显示了一个彩虹，其中每个 RGB 通道都捕获了光谱的特定部分。（该图简化了，因为它做了一些省略。例如，橙色和黄色的带表示为红色和绿色的组合。）

图像有几种文件格式，但是幸运的是，您有很多方法可以在 Python 中加载图像。首先使用 `imageio` 模块加载 PNG 图像。在本章中，您将使用 `imageio`，因为它使用统一的 API 处理不同的数据类型。现在加载图像，如下面的清单所示。

#### Listing 3.4 code/p1ch4/5\_image\_dog.ipynb

```
# In[2]:  
import imageio  
  
img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')  
img_arr.shape  
  
# Out[2]:  
(720, 1280, 3)
```

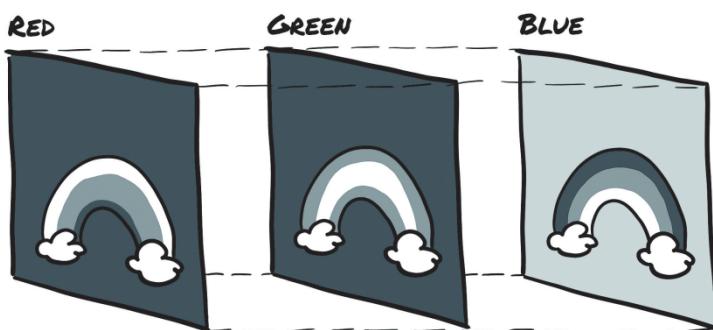


图 3.3 分解成红色、绿色和蓝色通道的彩虹

在这一点上，`img` 是一个具有 NumPy 数组的对象，具有三个维度：两个空间维度（宽度和高度）以及第三个维度，分别对应于颜色通道红色、绿色和蓝色。任何输出 NumPy 数组的库都这样做以获得 PyTorch 张量。唯一需要注意的是维度的布局。处理图像数据的 PyTorch 模块要求将张量布置为  $C \times H \times W$ （分别为通道、高度和宽度）。

您可以使用转置函数来获得适当的布局。给定输入张量  $W \times H \times C$ ，您可以通过交换第一个和最后一个通道来获得正确的布局：

```
# In[3]:  
img = torch.from_numpy(img_arr)  
out = torch.transpose(img, 0, 2)
```

您之前已经看过此示例，但是请注意，此操作不会复制张量数据。相反，`out` 使用与 `img` 相同的底层存储，并在张量级别上使用大小和步长信息。这种布置很方便，因为操作代价很小，但是（小心）更改 `img` 中的像素会导致 `out` 的变化。

还要注意，其他深度学习框架使用不同的布局。最初，TensorFlow 将通道维度保持在最后，导致  $H \times W \times C$  布局。（现在，它支持多种布局。）从低级性能的角度来看，此策略是有利有弊，但只要适当地重塑张量，它就不会对您有影响。

到目前为止，您已经描述了一张图片。遵循与以前的数据类型相同的策略，创建包含多个图像的数据集以用作神经网络的输入，然后沿第一维将这些图像成批存储，以获得  $N \times C \times H \times W$  维的张量。

作为使用堆栈构建张量的更有效的替代方法，您可以预分配适当大小的张量，并用从目录中加载的图像来填充它，

```
# In[4]:  
batch_size = 100  
batch = torch.zeros(100, 3, 256, 256, dtype=torch.uint8)
```

这表示您的批次将包含 100 个 RGB 图像，高度分别为 256 像素，宽度为 256 像素。请注意张量的类型：您期望每种颜色都以 8 位整数表示，就像大多数标准消费类相机的摄影格式一样。现在您可以从输入目录加载所有 png 图像并将其存储在张量中：

```
# In[5]:  
import os  
  
data_dir = '../data/plch4/image-cats/'  
filenames = [name for name in os.listdir(data_dir) if os.path.splitext(name)  
            == '.png']  
for i, filename in enumerate(filenames):  
    img_arr = imageio.imread(filename)  
    batch[i] = torch.transpose(torch.from_numpy(img_arr), 0, 2)
```

如前所述，神经网络通常使用浮点数的张量作为输入。正如您将在接下来的章节中看到的那样，当输入数据的范围大约为 0 到 1 或 -1 到 1 时，神经网络表现出最佳的训练性能（这是

对其构造块的定义方式产生的影响）。

您要做的典型的事情是将张量转换为浮点并标准化像素值。强制转换为浮点数很容易，但是归一化比较棘手，因为它取决于应介于 0 到 1（或-1 到 1）之间的您决定的输入的范围。一种可能性是将像素值除以 255（8 位无符号的最大可表示数字）：

```
# In[6]:  
batch = batch.float()  
batch /= 255.0
```

另一种可能性是计算输入数据的均值和标准差并对其进行缩放，以使输出在每个通道上的均值为 0 和标准差为 1：

```
# In[7]:  
n_channels = batch.shape[1]  
for c in range(n_channels):  
    mean = torch.mean(batch[:, c])  
    std = torch.std(batch[:, c])  
    batch[:, c] = (batch[:, c] - mean) / std
```

您可以对输入执行其他几种操作，包括旋转、缩放和裁切之类的几何变换。这些操作可能有助于训练，也可能是必要的。任意输入都能符合网络的输入要求，例如图像大小。您会偶然发现其中许多策略。现在，请记住您有可用的图像处理选项。

## 3.5 体积数据

您已经了解了如何加载和表示 2D 图像，比如使用相机拍摄得到的图像。在涉及 CT（计算机断层扫描）扫描等医学成像应用程序的场景下，通常需要处理从头到脚方向堆叠的图像序列，每个序列对应于整个身体上的一个切片。在 CT 扫描中，强度代表身体不同部位的密度：肺、脂肪、水、肌肉、骨骼，以密度依次递增，在临床工作站上显示 CT 扫描时，从暗到亮来进行映射。根据穿过人体后到达检测器的 X 射线量计算每个点的密度，并使用一些复杂的数学运算将原始传感器数据解卷积为完整体积。

CT 具有单个强度通道，类似于灰度图像。通常，在本机数据格式中，通道维度被忽略了，因此原始数据通常具有三个维度。通过将单个 2D 切片堆叠到 3D 张量中，您可以构建表示对象的 3D 解剖结构的体积数据。与图 3.3 不同，图 3.4 中的额外的维度表示物理空间中的偏移，而不是可见光谱的特定频带。

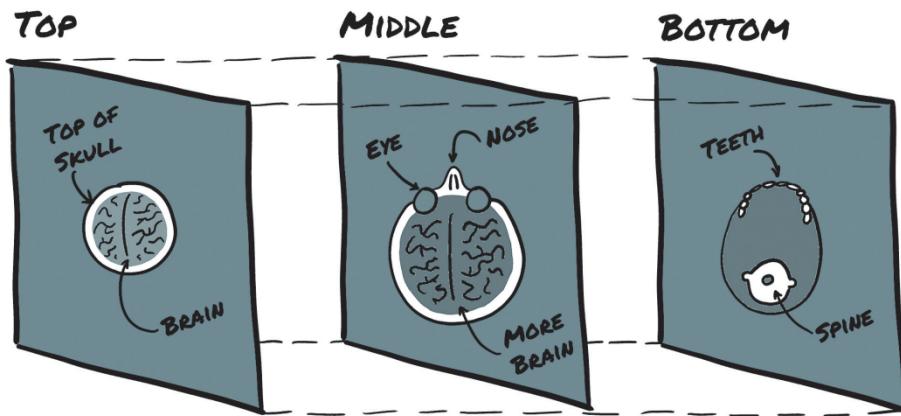


图 3.4 从头部顶部到颌骨的 CT 扫描切片

我们将不在此处详细介绍医学成像数据格式。就目前而言，足以说在存储体积数据的张量和存储图像数据的张量之间不存在根本差异。在通道维度之后，您有一个额外的维度，深度，最终得到 5D 张量，其形状为  $N \times C \times D \times H \times W$ 。

使用 `imageio` 模块中的 `volread` 函数加载一个样本 CT 扫描，该函数以目录作为参数，并将所有 DICOM（数字影像通信和存储）文件（

<https://wiki.cancerimagingarchive.net/display/Public/CPTAC-LSCC#dd4a08a246524596add33b9f8f00f288>）组合成一个 NumPy 3D 数组，如下表所示。

#### Listing 3.5 code/p1ch4/6\_volumetric\_ct.ipynb

```
# In [2]:
import imageio

dir_path = "../data/p1ch4/volumetric-dicom/2-LUNG 3.0_B70f-04083"
vol_arr = imageio.volread(dir_path, 'DICOM')
vol_arr.shape

# Out [2]:
Reading DICOM (examining files): 1/99 files (1.0% 99/99 files (100.0%)
  Found 1 correct series.
Reading DICOM (loading data): 87/99 (87.999/99 (100.0%)

# Out [2]:
(99, 512, 512)
```

同样在这种情况下，由于缺少通道信息，布局与 PyTorch 期望的布局不同。您必须使用 `unsqueeze` 函数为通道维度腾出空间：

```
# In[3]:  
vol = torch.from_numpy(vol_arr).float()  
vol = torch.transpose(vol, 0, 2)  
vol = torch.unsqueeze(vol, 0)  
  
vol.shape  
  
# Out[3]:  
torch.Size([1, 512, 512, 99])
```

此时，可以像本章前面所述那样，通过沿批处理方向堆叠多个体积来组装 5D 数据集。

## 结论

本章涵盖了很多基础知识。您学习了如何加载最常见的数据类型并对其进行整形以供神经网络使用。当然，野外数据格式比我们希望的在单个体积中描述的还要多。有些内容，例如医学史，过于复杂而无法涵盖。但是，对于感兴趣的读者，我们确实在代码存储库中提供了额外的 Jupyter notebooks 中创建音频和视频张量的简短示例（<https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch4>）。

## 练习

- 用手机或其他数码相机拍摄红色、蓝色和绿色项目的几张照片。
  - 加载每个图像，并将其转换为张量。
  - 对于每个图像张量，请使用`.mean()`方法来了解图像的亮度。
  - 现在，取图像每个通道的平均值。您能否仅从通道平均值中识别出红色、绿色和蓝色项目？
- 选择一个包含 Python 源代码的相对较大的文件。
  - 建立源文件中所有单词的索引。（请随自己的意愿使您的标记简单或复杂；我们建议您先将`r“[^ a-zA-Z0-9 _] +”`替换为空格。）
  - 将你的索引与为“傲慢与偏见”创建的索引进行比较。哪个更大？
  - 为源代码文件创建独热编码。
  - 这种编码会丢失什么信息？该信息与“傲慢与偏见”编码中丢失的信息相比如何？

## 总结

- 神经网络要求将数据表示为多维数值张量，通常为 32 位浮点数。
- 由于 PyTorch 库与 Python 标准库和周围的生态系统进行交互方式，因此可以方便地加载最常见的数据类型并将其转换为 PyTorch 张量。
- 通常，PyTorch 期望根据模型架构（例如卷积与递归）按照特定的维度对数据进行布局。

使用 PyTorch 张量 API 可以有效地实现数据重塑。

- 电子表格可以很容易地转换为张量。分类值和序数值列的处理方式应不同于间隔值列。
- 通过使用字典，可以将文本或分类数据编码为独热编码表示。
- 图像可以具有一个或多个通道。最常见的是典型数码照片的红色、绿色和蓝色通道。
- 单通道数据格式有时会省略明确的通道维度。
- 体积数据类似于 2D 图像数据，但增加了第三个维度：深度。
- 许多图像的每个通道的位深度为 8，尽管每通道 12 位和 16 位并不罕见。这些位深度可以存储为 32 位浮点数，而不损失精度。

## 4 学习的机理

本章覆盖内容有：

- 了解算法如何从数据中学习
- 利用微分和梯度下降以参数估计的形式来重新构造学习
- 从头开始学习简单的学习算法

随着近十年来机器学习的蓬勃发展，从经验中学习的机器的概念已成为技术界和新闻界的主流主题。现在，机器到底是如何学习的？它的机制或背后的算法是什么？从外部观察者的角度来看，学习算法将输入数据与所需输出配对。当发生学习时，该算法在输入与训练时所输入的数据足够相似的新数据时，便能够产生正确的输出。借助深度学习，即使输入数据和所需的输出彼此相差很大时（即使它们来自不同的域，例如图像和描述它的句子），该过程也可以工作。

实际上，帮助您解释输入/输出关系的模型可以追溯到几个世纪之前。约翰·开普勒（Johannes Kepler）是一位生活在 1571 年至 1630 年之间的德国数学天文学家，他基于他的导师 Tycho Brahe 在肉眼观察期间收集的数据（是的，肉眼和一张纸），他在 1600 年代初提出了他的三个行星运动定律时。他没有牛顿的万有引力定律可供使用（实际上，牛顿利用开普勒的著作来解决问题），他推断出可能适合数据的最简单的几何模型。顺便说一下，他花了 6 年的时间盯着那些对他没有意义的数据并逐步认识并公式化这些定律（<http://galileoandinstein.physics.virginia.edu/1995/lectures/morekepl.html>）。您可以在图 4.1 中看到这一过程。

第一定律是：“每个行星的轨道都是椭圆形，太阳位于两个焦点之一。”他不知道是什么使轨道变成椭圆形，但给出了对行星（或大行星的卫星，如木星）的一组观测结果，他此时可以估计形状（离心率）和椭圆的大小（半通径）。利用从数据中计算出的这两个参数，他可以判断出行星在天空中可能位于何处。当他提出第二定律时：“一条连接行星和太阳的直线在相等的时间间隔内扫过的面积相等”，他还可以根据给出的观测结果判断出行星何时到达空间中的特定点（[https://en.wikipedia.org/wiki/Kepler%27s\\_laws\\_of\\_planetary\\_motion](https://en.wikipedia.org/wiki/Kepler%27s_laws_of_planetary_motion)）。

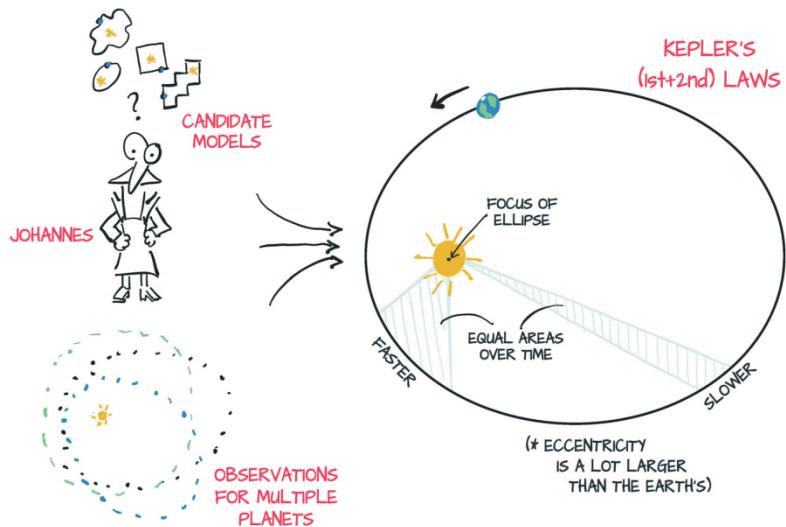


图 4.1 约翰尼斯·开普勒 (Johannes Kepler) 考虑了可能适合手头数据的多个候选模型，并最终确立了椭圆模型。

开普勒如何在没有计算机、袖珍计算器甚至微积分（而这在当时还没有被发明）的情况下估计椭圆的偏心率和椭圆尺寸？您可以从开普勒自己在《新天文学》一书中的回忆，或者从 J.V. Field 的《证明的起源》系列中找到答案，来了解答案：

本质上，开普勒必须尝试不同的形状，使用一定数量的观测值来找到曲线，然后使用曲线找到更多的位置（在他有观测值的时候），然后检查这些计算出的位置是否与观测到的位置一致。

总结一下，开普勒在过去的六年中

- 1 从他的朋友布拉 (Brahe) 得到了很多很好的数据（并非没有任何努力）。
- 2 试图将这些数据可视化，因为他感觉到有些麻烦。
- 3 选择有可能拟合数据的最简单模型（椭圆形）。
- 4 拆分数据，以便他可以处理部分数据，并保留独立的数据集合进行验证。
- 5 从试探性的偏心率和大小开始，然后进行迭代，直到模型适合观察结果为止。
- 6 在独立观察中验证了他的模型。
- 7 怀疑地往回看。

从 1609 年到现在，您都可以找到一本数据科学手册。

科学的历史是建立在这七个步骤之上的，正如科学家们在几个世纪以来所学到的那样，背离它们是灾难的根源。

这些步骤正是您从数据中学到的东西。在这里，说要拟合数据和说要使算法从数据中学习几乎没有区别。该过程始终涉及具有未知参数的函数，其参数是根据数据估算的，简称模型。

您可以辩解说，从数据中学习得出的结论是，该基础模型并非旨在解决特定问题（例如开普勒工作中的椭圆形问题），并且能够近似更广泛的函数族。一个神经网络可以预测第谷·布拉赫（Tycho Brahe）的轨迹，而无需开普勒的独到见解来尝试将数据拟合为椭圆形。但是，艾萨克·牛顿爵士从原模型中得出他的万有引力定律将要困难得多。

您对后一种模型感兴趣：这些模型不是为解决特定的狭窄任务而设计的，而是可以使用输入和输出对自动进行调整以专门解决许多类似的任务，换句话说，就是对通用模型在手头任务相关的数据上进行了训练。尤其是，PyTorch 旨在简化创建模型的过程，对于该模型，可以解析地表达相对于参数的拟合误差的导数。如果最后一句话没有道理，请不要担心；第 1.1 节应为您讲清楚。

本章介绍如何自动执行通用函数拟合，这是您使用深度学习所做的全部工作，而深度神经网络是通用函数，PyTorch 使此过程尽可能简单和透明。为了确保正确理解关键概念并让您从第一原理中了解学习算法的原理，我们将从比深度神经网络简单得多的模型开始。

## 4.1 学习就是参数估计

在本节中，您将学习如何获取数据，选择模型并估算模型的参数，以便对新数据给出良好的预测。为此，您将摆出行星运动的复杂性，将注意力转移到物理学中第二难的问题上：校准仪器。

图 4.2 简要概述了到本章末尾您将要实现的内容。给定输入数据和相应的期望输出（真值）以及权重的初始值，向模型提供输入数据（正向传递），并通过将结果输出与真值情况进行比较来评估误差的度量。为了优化模型的参数，其权重-权重单位变化后误差的变化（误差相对于参数的梯度）-通过使用链式法则对复合函数的导数进行计算（向后传递）。然后，权重的值沿导致误差减小的方向更新。重复该过程，直到对不可见数据的评估错误降至可接受的水平以下。

如果听起来有点晦涩难懂，我们用整整一章来进行说明。待我们完成后，所有内容都将落到位，并且之前的段落也能让您理解。

接下来，您要处理噪声数据集，建立模型并为其实现学习算法。您将首先手动完成所有操作，但是到本章结束时，您将让 PyTorch 完成所有繁重的工作。在本章结束时，我们将介绍训练深度神经网络所依据的许多基本概念，即使激动人心的示例很简单且模型不是神经网络（尚未！）。

### 4.1.1 热门问题

假设您去了一些不起眼的地方，然后带回了一个花哨的壁挂式模拟温度计。看起来很棒，非常适合您的客厅。唯一的缺点是它不显示单位。不要担心；你有一个计划。您将以自己喜欢的单位建立一个读数和相应温度值的数据集，选择一个模型，并迭代调整其权重，直到误差

的测量值足够低为止，您最后终于可以在其中解释新的读数并用您理解的单位。

首先记录好旧摄氏的温度数据，并记录新温度计的测量值。

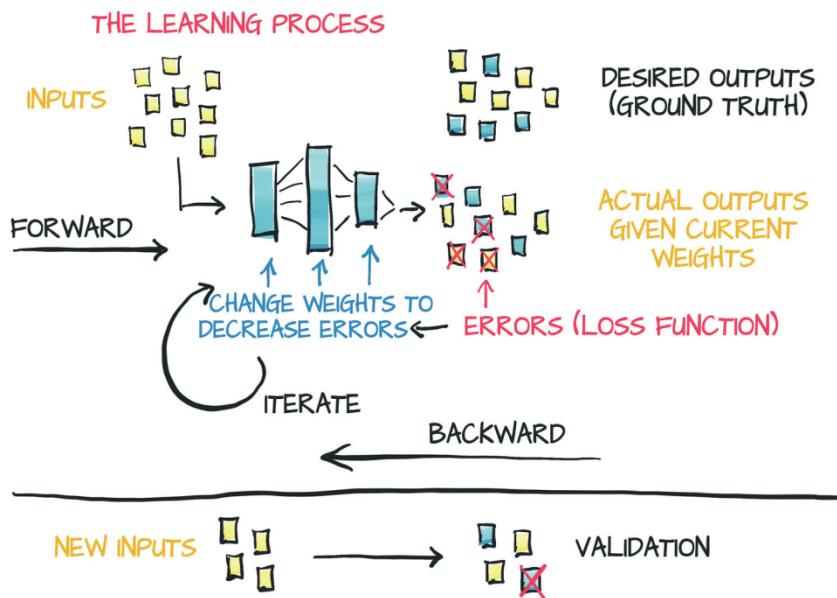


图 4.2 学习过程的思想模型

几周后，数据如下：

#### Listing 4.1 code/p1ch5/1\_parameter\_estimation.ipynb

```
# In[2] :  
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]  
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]  
t_c = torch.tensor(t_c)  
t_u = torch.tensor(t_u)
```

$t_c$  是摄氏温度， $t_u$  是未知单位。您可以预期设备本身和您的近似读数都会产生噪声。为了方便起见，数据已经在张量中，您将很快使用它。

### 4.1.2 选择线性模型作为首次尝试

在没有进一步知识的情况下，正如开普勒所做的工作，假设最简单的模型来对两组测量之间进行转换。两组数据可能是线性相关。也就是说，将  $t_u$  乘以一个因子并加上一个常数，您可以得到摄氏温度：

$$t_c = w * t_u + b$$

这个假设合理吗？很有可能；您会看到最终模型的效果会有多好。（您选择了  $w$  和  $b$  作为权重和偏差的名称，这是两个常用的线性缩放项和附加常数项，您会一直遇到。）

**注意** 剧透警告：我们知道线性模型是正确的，因为问题和数据是伪造的了，但是请耐

心等待；该模型是一个有用的激励示例，有助于您了解 PyTorch 的内部操作。

现在，您需要根据已有的数据估算模型中的参数  $w$  和  $b$ 。您必须执行此操作，以使通过运行模型的未知温度  $t_u$  获得的温度接近以摄氏度为单位的温度。如果这个过程听起来像是通过一组测量值拟合一条直线，那正是您在做的事情。当您使用 PyTorch 遍历此简单示例时，应意识到训练神经网络本质上涉及将模型更改为具有更多（或一公吨）更多参数的复杂模型。

为了再次充实示例，您有一个带有一些未知参数的模型，并且需要估计这些参数，以使预测输出和测量值之间的误差尽可能小。您注意到您仍然需要定义这种错误的度量。如果误差很大，那么这种度量（我们称为损失函数）应该很高，并且理想情况下，对于完美匹配，应该尽可能低。因此，您的优化过程应以找到  $w$  和  $b$  为目标，以使损失函数处于最低水平。

### 4.1.3 减少损失是您想要的

损失函数（或代价函数）是计算学习过程试图最小化的单个数值的函数。损耗的计算通常包括获取一些训练样本的期望输出与模型在嵌入这些样本时所产生的输出之间的差异，在这种情况下，模型输出的预测温度  $t_p$  与实际测量值之间的差异即  $t_p - t_c$ 。

您需要确保在  $t_p$  高于或低于真实  $t_c$  时，损失函数都会使损失为正，因为目标是使该值最小。（将损失无限地推向负数是没有用的。）您有几种选择，最直接的是  $|t_p - t_c|$  和  $(t_p - t_c)^2$ 。根据您选择的数学表达式，您可以强调或消除某些错误。从概念上讲，损失函数是一种从训练样本中优先确定要修复哪些错误的方法，这样，参数更新将导致对高权重样本的输出进行调整，而不是对损失较小的其他样本的输出进行更改。

这两个示例损失函数都具有明显的最小值零，并且随着预测值在任一方向上远离真实值而单调增长。因此，两个函数都被认为是凸函数。因为您的模型是线性的，所以作为  $w$  和  $b$  的函数的损失也是凸的。损失是模型参数的凸函数的情况通常很容易处理，因为您可以使用专门的算法以有效的方式找到最小值。深度神经网络不会出现凸损失，因此，这些方法通常对您没有用。

对于两个损失函数  $|t_p - t_c|$  和  $(t_p - t_c)^2$ ，如图 4.3 所示，请注意，误差平方损失函数在最小值附近表现得更好：当  $t_p$  等于  $t_c$  时，误差平方损耗相对于  $t_p$  的导数为零。相反，绝对值函数在您想要收敛的位置不具有导数。这个问题并不像实际看起来那么重要，但是暂时坚持选择误差平方函数。

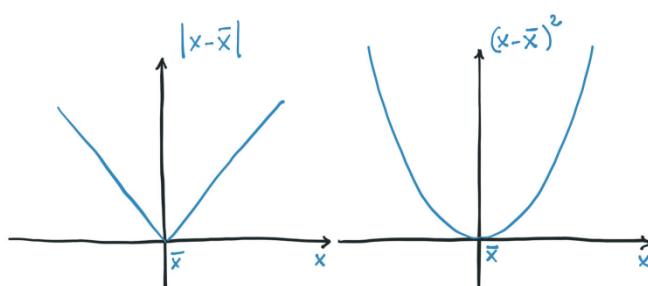


图 4.3 误差绝对值与误差平方

值得注意的是，平方差也比绝对差更严重地惩罚了严重错误的结果。通常，稍微出错的结果要好于一些严重错误的结果，并且平方差有助于按需要对这些结果进行优先级排序。

#### 4.1.4 从问题到 PyTorch

您已经弄清楚了模型和损失函数，因此，您已经了解了高层次的情况。现在，您需要启动学习过程并向其提供实际数据。另外，已经有了足够的数学符号，因此现在切换到 PyTorch。毕竟，您是来这里玩的。

您已经创建了数据张量，因此将模型写为 Python 函数

```
# In[3]:  
def model(t_u, w, b):  
    return w * t_u + b
```

其中您期望  $t_u$ 、 $w$  和  $b$  分别为输入张量、权重参数和 bias 参数。在您的模型中，参数将是 PyTorch 标量（也称为零维张量），并且乘积运算将使用广播来产生返回的张量。现在定义您的损失：

```
# In[4]:  
def loss_fn(t_p, t_c):  
    squared_diffs = (t_p - t_c)**2  
    return squared_diffs.mean()
```

请注意，您正在构建一个张量差，将它们的平方取平方，最后通过对所得张量中的所有元素求平均来产生标量损失函数。损失是均方损失。

现在您可以初始化参数，调用模型，

```
# In[5]:  
w = torch.ones(1)  
b = torch.zeros(1)  
  
t_p = model(t_u, w, b)  
t_p  
  
# Out[5]:  
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000, 21.8000,  
       48.4000, 60.4000, 68.4000])
```

并检查损失值：

```
# In[6]:  
loss = loss_fn(t_p, t_c)  
loss  
  
# Out[6]:  
tensor(1763.8846)
```

在本节中，您实现了模型和损失。该部分的重点是如何估计  $w$  和  $b$ ，以使损失达到最小值。首先，您需要手工解决问题；然后您将学习如何利用 PyTorch 超级功能以更通用的现成方式解决相同的问题。

### 4.1.5 沿梯度下降

在本节中，您将使用所谓的梯度下降算法来针对参数优化损失函数，并根据第一原理建立关于梯度下降工作原理的直觉，这将在将来为您带来很多帮助。有多种方法可以更有效地解决此特定示例，但这些方法不适用于大多数深度学习任务。梯度下降是一个简单的想法，可以令人惊讶地很好地扩展到具有数百万个参数的大型神经网络模型。

首先从图 4.4 中方便地勾勒出的思想图像开始。假设您在带有两个标有  $w$  和  $b$  的旋钮的机器前。您可以在屏幕上看到损失值，并被告知将损失值最小化。不知道旋钮对损耗的影响，您可能会开始摆弄它们，并为每个旋钮确定使损耗减小的方向。您可能决定将两个旋钮都朝着减少损耗的方向旋转。如果您距离最佳值还很远，那么您可能会看到损失迅速减小，然后随着接近最小值而逐渐放慢减小的速度。您会注意到，损耗有时会再次上升，因此您需要反转一个或两个旋钮的旋转方向。您还将了解到，当损耗变化缓慢时，最好更精细地调节旋钮，以避免到达损耗上升的位置。一段时间之后，最终您会收敛到最低限度。



图 4.4 是优化过程的卡通图，其中带有 w 和 b 旋钮的人搜索使损耗减小的方向

梯度下降没有太大不同。其思想是计算相对于每个参数的损耗变化率，并沿损耗减小的方向对每个参数应用变化。当您摆弄旋钮时，通过对 w 和 b 进行小的更改来估计该变化率，以查看该邻域中的损耗变化了多少：

```
# In[7]:  
delta = 0.1  
  
loss_rate_of_change_w = \  
    (loss_fn(model(t_u, w + delta, b), t_c) -  
     loss_fn(model(t_u, w - delta, b), t_c)) / (2.0 * delta)
```

这段代码说的是，在 w 和 b 的当前值的很小范围内，w 的单位增加会导致损耗的某些变化。如果变化为负，则需要增加 w 以使损失最小，而如果变化为正，则需要减小 w。减多少？对 w 施加与损失的变化率成比例的变化是个好主意，尤其是在损失具有多个参数的情况下：将变化应用于对损失有重大变化的参数。通常，缓慢地更改参数也是明智的做法，因为更改速度可能会在与当前 w 值的距离较远的地方相差很大。因此，您应该以较小的比例缩放变化率。这个比例因子有很多名称。机器学习中使用的名称是 learning\_rate。

```
# In[8]:  
learning_rate = 1e-2  
  
w = w - learning_rate * loss_rate_of_change_w
```

您可以使用 b 进行相同操作：

```
# In[9]:  
loss_rate_of_change_b = \  
    (loss_fn(model(t_u, w, b + delta), t_c) -  
     loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)  
  
b = b - learning_rate * loss_rate_of_change_b
```

此代码表示梯度下降的基本参数更新步骤。通过重复这些评估（假设您选择的学习率足够低），您可以收敛到参数的最佳值，对于该参数，根据给定数据计算出的损失最小。我们将很快向您展示完整的迭代过程，但是这种计算变化率的方法相当粗糙，需要升级。在下一节中，您将了解原因和方式。

## 4.1.6 获得分析

通过重复评估模型和损失以探测 w 和 b 附近的损失函数的行为来计算变化率，并不能很好地适应具有许多参数的模型。此外，并不总是很清楚该邻域应该有多大。您之前选择的 delta 等于 0.1，但一切都取决于损失函数的形状，该函数是 w 和 b 的函数。如果损失与增量相比变化得太快，那么您将无法感知下坡的位置。

如果您可以使邻域无限小，如图 4.5 所示，该出现什么情况？这时发生的情况就是您分析得出参数的损失函数的偏导数。在具有两个或多个参数的模型中，您将计算针对每个参数的损失函数的各自偏导数，并将它们放在导数向量中就得到：梯度。

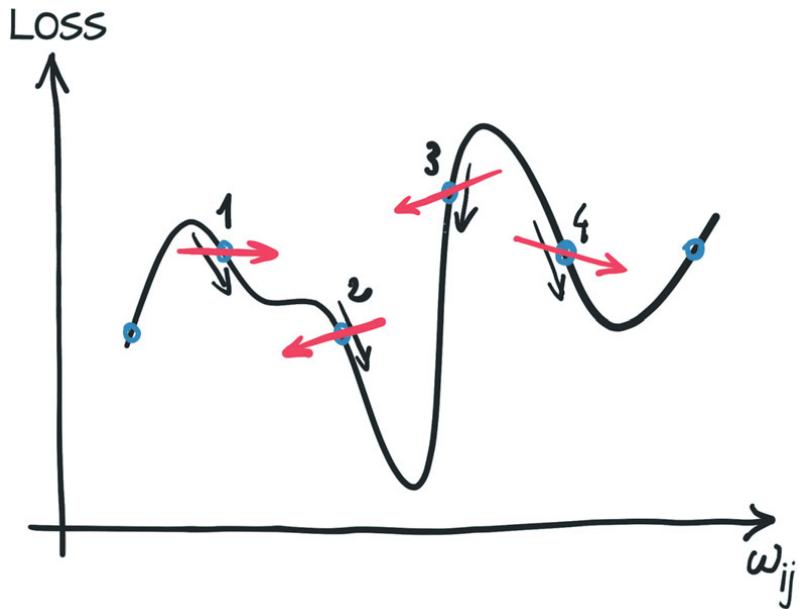


图 4.5 在离散位置和分析位置评估下坡方向时的估计差异

要计算损失函数相对于参数的导数，您可以应用链式规则，并计算损失相对于其输入（即模型的输出）的导数乘以模型相对于参数的导数：

$$\frac{d \text{ loss\_fn}}{d w} = (\frac{d \text{ loss\_fn}}{d t_p}) * (\frac{d t_p}{d w})$$

回想一下，该模型是线性函数，损耗是平方和。现在找出导数的表达式。回顾损失的表达

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

并记住  $d x^2 / d x = 2x$ ，

```
# In[10]:
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c)
    return dsq_diffs
```

至于模型，回想一下该模型是

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

你得到的导数：

```
# In[11]:
def dmodel_dw(t_u, w, b):
    return t_u

# In[12]:
def dmodel_db(t_u, w, b):
    return 1.0
```

将所有这些放在一起，返回损耗相对于  $w$  和  $b$  的梯度的函数为

```
# In[13]:
def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dw = dloss_fn(t_p, t_c) * dmodel_dw(t_u, w, b)
    dloss_db = dloss_fn(t_p, t_c) * dmodel_db(t_u, w, b)
    return torch.stack([dloss_dw.mean(), dloss_db.mean()])
```

用数学符号表示的相同想法如图 4.6 所示。

同样，您要对所有数据点求平均值（求和并除以一个常数），以得到损失的每个偏导数的单个标量。

$$\nabla_{w,b} L = \left( \frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = \left( \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b} \right)$$

↑ gradient     
 ↑ partial derivatives     
 ↑ model  $m_{w,b}(x)$      
 ↑ parameters

图 4.6 损耗函数相对于权重的导数

### 4.1.7 训练循环

现在，您已准备就绪，可以优化参数。从参数的试探值开始，您可以迭代地对其进行更新以进行固定数量的迭代，或者直到  $w$  和  $b$  停止更改为止。您可以使用多个停止条件，但现在选择固定的迭代次数作为迭代停止条件。

在讨论过程中，我们将向您介绍另一种术语。您在其中更新所有训练样本的参数的训练迭代称为一个时期（epoch）。

完整的训练循环如下所示：

```
# In[14]:  
def training_loop(n_epochs, learning_rate, params, t_u, t_c):  
    for epoch in range(1, n_epochs + 1):  
        w, b = params  
        t_p = model(t_u, w, b) ← This is the forward pass.  
        loss = loss_fn(t_p, t_c)  
        grad = grad_fn(t_u, t_c, t_p, w, b) ← And this is the backward pass.  
  
        This logging  
        line can be  
        verbose. ↓  
        params = params - learning_rate * grad  
        print('Epoch %d, Loss %f' % (epoch, float(loss)))  
  
    return params
```

本文中用于输出的实际日志记录逻辑更为复杂（请参见同一 notebook 文件中的单元格 15），（[https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/p1ch5/1\\_parameter\\_estimation.ipynb](https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/p1ch5/1_parameter_estimation.ipynb)）但区别对于理解本章的核心概念并不重要。

现在调用您的训练循环：

```
# In[16]:  
training_loop(  
    n_epochs = 100,  
    learning_rate = 1e-2,  
    params = torch.tensor([1.0, 0.0]),  
    t_u = t_u,  
    t_c = t_c)  
  
# Out[16]:  
Epoch 1, Loss 1763.884644  
    Params: tensor([-44.1730, -0.8260])  
    Grad:  tensor([4517.2964,  82.6000])  
Epoch 2, Loss 5802484.500000  
    Params: tensor([2568.4011,   45.1637])  
    Grad:  tensor([-261257.4062, -4598.9707])  
Epoch 3, Loss 19408031744.000000  
    Params: tensor([-148527.7344, -2616.3933])  
    Grad:  tensor([15109615.0000,  266155.7188])  
...  
.
```

```
Epoch 10, Loss 90901075478458130961171361977860096.000000  
    Params: tensor([3.2144e+17, 5.6621e+15])  
    Grad:  tensor([-3.2700e+19, -5.7600e+17])  
Epoch 11, Loss inf  
    Params: tensor([-1.8590e+19, -3.2746e+17])  
    Grad:  tensor([1.8912e+21, 3.3313e+19])  
  
tensor([-1.8590e+19, -3.2746e+17])
```

等等——发生了什么事？您的训练过程爆炸，导致损失趋于无穷。这个结果清楚地表明，参数接收的更新太大。它们的值在每次更新在修正过头后开始来回摆动，而下一个修正则更过头。优化过程不稳定。它发散而不是收敛到最小值。您希望看到对参数的更新越来越小，而不是较大，如图 4.7 所示。

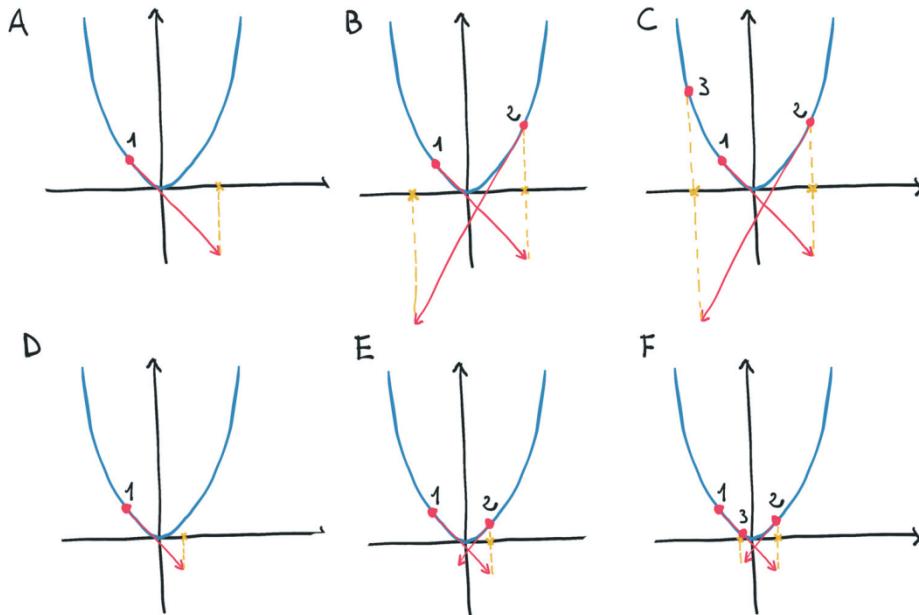


图 4.7 上图：由于步长较大，导致凸函数（抛物线形）的发散优化。底部：小步长的收敛优化。

您如何限制 `learning_rate * grad` 的大小？好吧，这个过程看起来很简单。您可以选择一个较小的 `learning_rate`。通常，您会更改学习率的数量级，因此您可以尝试 `1e-3` 或 `1e-4`，这将使更新量降低几个数量级。用 `1e-4` 来查看其工作原理：

```
# In[17]:  
training_loop(  
    n_epochs = 100,  
    learning_rate = 1e-4,
```

```
params = torch.tensor([1.0, 0.0]),
t_u = t_u,
t_c = t_c)

# Out[17]:
Epoch 1, Loss 1763.884644
    Params: tensor([ 0.5483, -0.0083])
    Grad:   tensor([4517.2964,    82.6000])
Epoch 2, Loss 323.090546
    Params: tensor([ 0.3623, -0.0118])
    Grad:   tensor([1859.5493,   35.7843])
Epoch 3, Loss 78.929634
    Params: tensor([ 0.2858, -0.0135])
    Grad:   tensor([765.4666,  16.5122])
...
Epoch 10, Loss 29.105242
    Params: tensor([ 0.2324, -0.0166])
    Grad:   tensor([1.4803, 3.0544])
Epoch 11, Loss 29.104168
    Params: tensor([ 0.2323, -0.0169])
    Grad:   tensor([0.5780, 3.0384])
...
Epoch 99, Loss 29.023582
    Params: tensor([ 0.2327, -0.0435])
    Grad:   tensor([-0.0533, 3.0226])
Epoch 100, Loss 29.022669
    Params: tensor([ 0.2327, -0.0438])
    Grad:   tensor([-0.0532, 3.0226])

tensor([ 0.2327, -0.0438])
```

很好，现在收敛行为变得稳定。但是还有另一个问题：参数的更新很小，因此损耗会缓慢下降并最终停滞。您可以通过使用自适应性的 `learning_rate` 来解决此问题，即根据更新的大小进行更改 `learning_rate`。为此，可以使用几种优化方案。您会在本章末尾的“*Optimizers a-la Carte*”部分看到一个方案。

更新术语中存在另一个潜在的麻烦制造者：梯度本身。回到最优化阶段 1 的 `grad` 变量。您会看到，权重的第一个时期梯度大约是偏置的梯度的 50 倍，因此权重和偏差存在于不同尺度的空间中。在这种情况下，足够大的学习率足以有意义地更新一个参数，对于另一个参数的更新则是不稳定的，或者适合第二个参数的学习率不足以有意义地改变第一个。除非您更改问题的公式，否则您将无法更新参数。您可以为每个参数设置单独的学习率，但是对于具有多个参数的模型，此方法将非常麻烦。这会非常繁琐。

您可以采用一种更简单的方法来检查所有内容：更改输入，以使梯度没有太大不同。粗略地说，您可以确保输入范围与 -1.0 到 1.0 的范围相差不大。在这种情况下，您可以通过将 `t_u` 乘以 0.1 来达到与该示例足够接近的效果：

```
# In[18]:
t_un = 0.1 * t_u
```

在这里，通过将 `n` 附加到变量名来表示 `t_u` 的标准化版本。此时，您可以在标准化输入上运行训练循环：

```
# In[19]:
def training_loop():
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,      # You've updated t_u to
    t_c = t_c)      # your new, rescaled t_un.

# Out[19] :
Epoch 1, Loss 80.364342
    Params: tensor([1.7761, 0.1064])
    Grad:   tensor([-77.6140, -10.6400])
Epoch 2, Loss 37.574917
    Params: tensor([2.0848, 0.1303])
    Grad:   tensor([-30.8623, -2.3864])
Epoch 3, Loss 30.871077
    Params: tensor([2.2094, 0.1217])
    Grad:   tensor([-12.4631,  0.8587])
...
Epoch 10, Loss 29.030487
    Params: tensor([ 2.3232, -0.0710])
    Grad:   tensor([-0.5355,  2.9295])
Epoch 11, Loss 28.941875
    Params: tensor([ 2.3284, -0.1003])
    Grad:   tensor([-0.5240,  2.9264])
...
Epoch 99, Loss 22.214186
    Params: tensor([ 2.7508, -2.4910])
    Grad:   tensor([-0.4453,  2.5208])
Epoch 100, Loss 22.148710
    Params: tensor([ 2.7553, -2.5162])
    Grad:   tensor([-0.4445,  2.5165])

    tensor([ 2.7553, -2.5162])
```

即使您将学习率设置回 1e-2，在迭代更新过程中参数也不会爆炸。现在看一下梯度。它们的数量级相似，因此对两个参数使用单个 `learning_rate` 效果很好。您可能要比将缩放比例扩大十倍做得更好，但是由于这样做足以满足您的需求，因此请坚持下去。

**注意:** 此处的规范化可以帮助您训练网络，但您可以提出这样的观点，即正则化对于优化此问题的参数不是必要的。这绝对是真的！这个问题很小，您有很多解决方法。但是，对于更大，更复杂的问题，归一化是用于改善模型收敛性的简便有效（如果不是至关重要的话）的工具。

接下来，运行循环进行足够的迭代，可以看到参数的变化变小。将 `n_epochs` 更改为 5000：

```
# In[20]:  
params = training_loop(  
    n_epochs = 5000,  
    learning_rate = 1e-2,  
    params = torch.tensor([1.0, 0.0]),  
    t_u = t_un,  
    t_c = t_c,  
    print_params = False)  
  
params  
  
# Out[20]:  
Epoch 1, Loss 80.364342  
Epoch 2, Loss 37.574917  
Epoch 3, Loss 30.871077  
...  
Epoch 10, Loss 29.030487  
Epoch 11, Loss 28.941875  
...  
Epoch 99, Loss 22.214186  
Epoch 100, Loss 22.148710  
...  
Epoch 4000, Loss 2.927680  
Epoch 5000, Loss 2.927648  
  
tensor([ 5.3671, -17.3012])
```

很好。在沿梯度下降方向更改参数时，您看到损耗减小。损失并没有为零，这可能意味着迭代不足以收敛为零，或者数据点不在一条线上。如预期的那样，您的测量结果并非十分准确，或者读数中包含噪音。

但是请注意:  $w$  和  $b$  的值看起来非常糟糕，就像将摄氏度转换为华氏度所需要的数字一样（在考虑了将输入乘以 0.1 的较早归一化之后）。确切值是  $w = 5.5556$  和  $b = -17.7778$ 。您的温度计一直在显示华氏温度，这并不是什么大发现，但可以证明您的梯度下降优化过程非常有效。

接下来，从一开始就应该做的事情: 绘制数据。为了刺激性，我们直到现在才介绍这个主题（令人惊奇的效果）。但是，认真的说，任何从事数据科学工作的人都应该做的第一件事就是绘制数据。

```
# In[21]:  
%matplotlib inline  
from matplotlib import pyplot as plt  
t_p = model(t_un, *params)
```

Remember that you're training on  
the normalized unknown units.

```

fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o')

```

But you're plotting the raw unknown values.

此代码产生图 4.8。

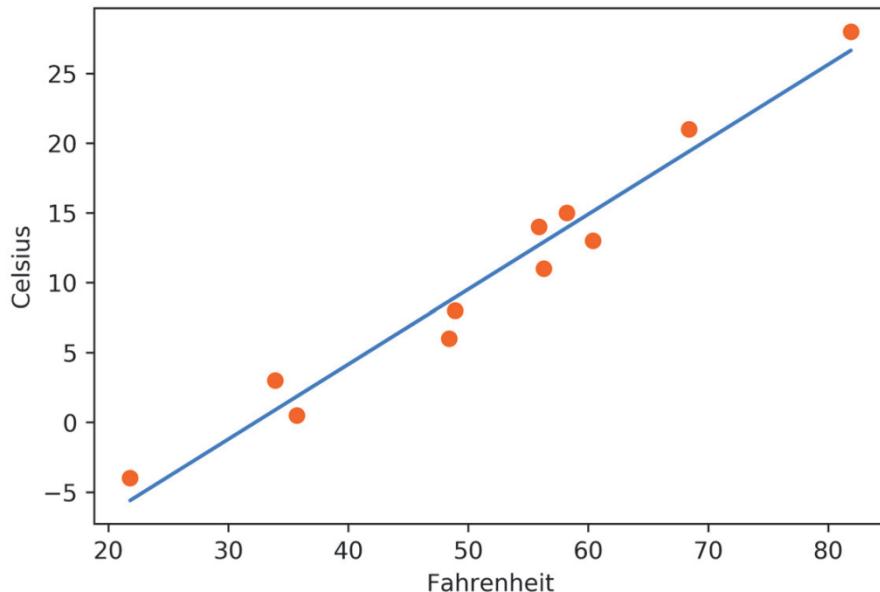


图 4.8 线性拟合模型（实线）与输入数据（圆圈）的关系图

线性模型似乎是数据的良好模型。您的测量结果似乎也有些不稳定。您应该打电话给验光师换一副新眼镜，或者考虑退还您的高档温度计。

## 4.2 PyTorch 的 autograd: 将一切都后向传播

到目前为止，在您的小冒险中，您看到了一个反向传播的简单示例。通过链式法则向后传播导数，可以计算出函数组成（模型和损失）相对于它们的最内层参数  $w$  和  $b$  的梯度。基本要求是您要处理的所有函数在数学分析上都是可微分的。在这种情况下，您可以一次扫描相对于参数的梯度（我们之前称其为“损耗变化率”）。

如果您有一个具有数百万个参数的复杂模型，只要模型是可微的，计算梯度相对于参数的损失就等于编写导数的解析表达式并对其进行一次评估。当然，为线性和非线性函数的较深组成部分的导数编写分析表达式并不是一件很有趣的事情。它也不是特别快。

在这种情况下，PyTorch 张量就可以通过一个名为 `autograd` 的 PyTorch 组件来解决。PyTorch 张量可以根据操作和源自它们的父级张量记住它们的来源，并且它们可以根据其输入自动提供此类操作的导数链。您无需手动对模型求导；给定一个正向表达式，无论嵌套如

何，PyTorch 都会自动提供该表达式相对于其输入参数的梯度。

此时，最好的方法是重写温度计校准代码，这一次使用 `autograd`，看看会发生什么。首先，调用您的模型和损失函数，如下面的清单所示。

### Listing 4.2 code/p1ch5/2\_autograd.ipynb

```
# In[3]:  
def model(t_u, w, b):  
    return w * t_u + b  
  
# In[4]:  
def loss_fn(t_p, t_c):  
    squared_diffs = (t_p - t_c)**2  
    return squared_diffs.mean()
```

再次初始化参数张量：

```
# In[5]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

注意张量构造函数的 `require_grad = True` 参数吗？这个参数告诉 PyTorch 跟踪由参数操作产生的整个张量族谱。换句话说，任何具有该参数作为祖先的张量都可以访问从它到该张量所调用的函数链。如果这些函数是可微的（大多数 PyTorch 张量运算是可微的），则导数的值将自动保存在参数张量的 `grad` 属性。

通常，所有 PyTorch 张量都有一个名为 `grad` 的属性，通常为 `None`：

```
# In[6]:  
params.grad is None  
  
# Out[6]:  
True
```

您要做的就是从将 `require_grad` 设置为 `True` 的张量开始，调用模型、计算损失，然后向后调用损失张量：

```
# In[7]:  
loss = loss_fn(model(t_u, *params), t_c)  
loss.backward()
```

```

params.grad

# Out [7] :
tensor([4517.2969,      82.6000])

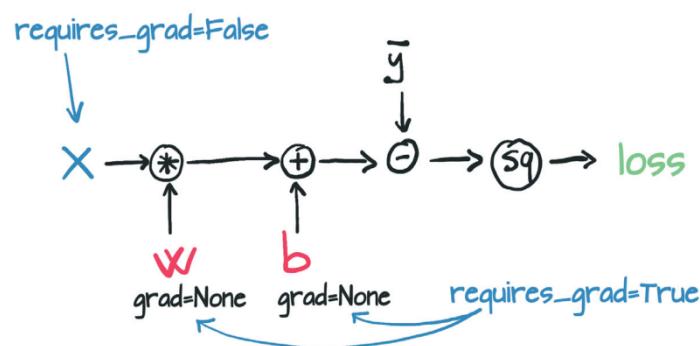
```

在这一点上，`params` 的 `grad` 属性包含关于参数的每个元素的损失的偏导数（图 4.9）。

您可以使用任意数量的张量（将 `require_grad` 设置为 `True`）和使用任何函数组成。在这种情况下，PyTorch 会在整个函数链（计算图）中计算损失的导数，并在这些张量的 `grad` 属性（图的叶节点）中累积其值。

**警报：**大陷阱。这是 PyTorch 的新手（以及很多经验丰富的人）经常犯的错误。我们这里写的是积累，而不是存储。

**警告** 调用 `backward` 函数会导致导数在叶节点处累积。将其用于参数更新后，需要将其显式清零。



`loss.backward()`

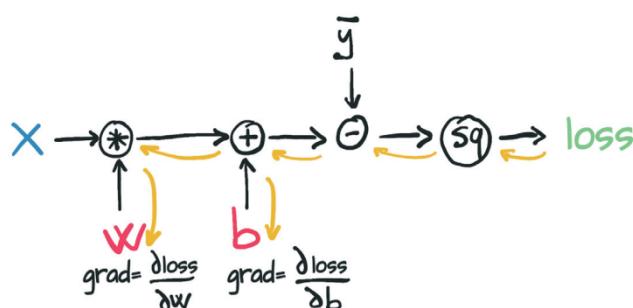


图 4.9 使用 autograd 计算的模型的正向图和反向图

重复一遍，调用 `backward` 函数会导致导数在叶节点处累积。因此，如果之前调用了 `backward` 函数，并再次求得损失，然后再次调用了 `backward` 函数（如在任何训练循环中一样），每个叶节点上的梯度被累加（求和）在前一次迭代所计算的叶节点上，这导致梯度值不正确。

为防止这种情况发生，您需要在每次迭代时将梯度显式归零。您可以使用原地函数 `zero_` 方法轻松地做到这一点：

```
# In[8]:  
if params.grad is not None:  
    params.grad.zero_()
```

**注意** 您可能很好奇，为什么在每次调用 `backwards` 函数后将梯度归零是必需的步骤，而不是自动进行的步骤。原因是为复杂模型中的梯度提供更大的灵活性和控制力。

让此提醒深入您的脑海，现在开始看看启用了 `autograd` 的训练代码的样子，从头到尾如下所示：

```
# In[9]:  
def training_loop(n_epochs, learning_rate, params, t_u, t_c):  
    for epoch in range(1, n_epochs + 1):  
        if params.grad is not None:    ←  
            params.grad.zero_()          This could be done at any point in the loop  
                                         prior to calling loss.backward()  
        t_p = model(t_u, *params)  
        loss = loss_fn(t_p, t_c)  
        loss.backward()  
  
        params = (params - learning_rate *  
                  params.grad).detach().requires_grad_()  
                                         It's somewhat cumbersome, but as  
                                         you'll see in "Optimizers a-la Carte,"  
                                         it's not an issue in practice.  
        if epoch % 500 == 0:  
            print('Epoch %d, Loss %f' % (epoch, float(loss))) ←  
  
    return params
```

请注意，更新参数时，您还执行了奇怪的`.detach().requires_grad_()`。要了解原因，请考虑一下您构建的计算图。重新格式化参数更新行，以免重复使用变量名：`p1 = (p0 * lr * p0.grad)` 在这里，`p0` 是用于初始化模型的随机权重。`p0.grad` 是通过损失函数根据 `p0` 和您的训练数据的组合来计算的。

到目前为止，一切都很好。现在，您需要查看循环的第二次迭代：`p2 = (p1 * lr * p1.grad)`。如您所见，`p1` 的计算图回到 `p0`，这是有问题的，因为（a）您必须将 `p0` 保留在内存中（直到您完成训练），并且（b）混淆了哪里的问题 您应该通过反向传播分配误差。

而是通过调用`.detach()` 将新的参数张量从与其更新表达式关联的计算图中分离出来。这样，`params` 实际上会丢失生成它的操作的内存。然后，您可以通过调用`.requires_grad_()` 重新启用跟踪，该操作是 `in_place` 操作（请参见 `trailing_`），可重新激活张量的自动梯度。现在，您可以释放旧版本参数所拥有的内存，并且只需通过当前权重进行反向传播。

查看此代码是否有效：

```
# In [10] :
def training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,      ←
    t_c = t_c)

# Out [10] :
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([ 5.3671, -17.3012], requires_grad=True)
```

Note that again, you're using the normalized `t_un` instead of `t_u`.

Adding this `requires_grad=True` is key.

您将获得与以前相同的结果。对你有好处！尽管您可以手动计算导数，但您不再需要。

### 4.2.1 优化器菜单

此代码使用香草梯度下降进行优化，在这种简单情况下效果很好。不用说，几种优化策略和技巧可以帮助收敛，尤其是当模型变得复杂时。

现在是时候介绍 PyTorch 从用户代码中为优化策略的方式写摘要了，例如训练循环，使您免于必须自己更新模型中每个参数的繁琐工作。`torch` 模块有一个 `optim` 子模块，您可以在其中找到实现不同优化算法的类。以下是一个简短的清单：

#### Listing 4.3 code/p1ch5/3\_optimizers.ipynb

```
# In [5] :
import torch.optim as optim

dir(optim)

# Out [5] :
```

```

['ASGD',
'Adadelta',
'Adagrad',
'Adam',
'Adamax',
'L-BFGS',
'Optimizer',
'RMSprop',
'Rprop',
'SGD',
'SparseAdam',
...
]

```

每个优化器构造函数都将参数列表（也称为 PyTorch 张量，通常将 `require_grad` 设置为 `True`）作为第一个输入。传递给优化器的所有参数都保留在优化器对象内，以便优化器可以更新其值并访问其 `grad` 属性，如图 4.10 所示。

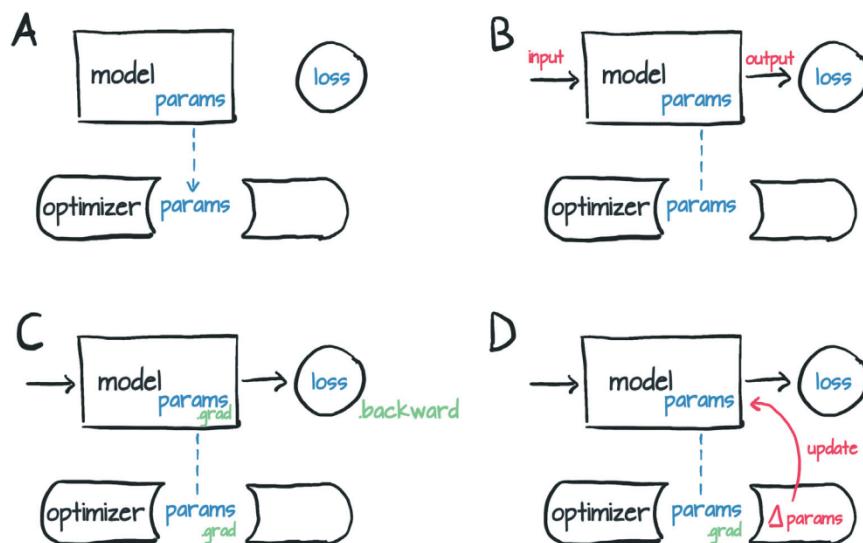


图 4.10 是优化器如何保留对参数 (A) 的引用的概念表示，在根据输入 (B) 计算损失之后，对`.backward` 的调用导致在参数 (C) 上产生`.grad` 属性。此时，优化器可以访问`.grad` 并计算参数更新 (D)。

每个优化器都公开两种方法：`zero_grad` 和 `step`。前者将构造时传递给优化器的所有参数的 `grad` 属性归零。后者根据特定优化器实施的优化策略更新这些参数的值。

现在创建参数并实例化梯度下降优化器：

```
# In[6]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)  
learning_rate = 1e-5  
optimizer = optim.SGD([params], lr=learning_rate)
```

在这里，SGD 代表随机梯度下降。优化器本身是原始的梯度下降（只要 momentum 参数设置为 0.0，这是默认值）。术语“随机”来自以下事实：通常通过对所有输入样本的随机子集（称为最小批量）求平均值来获得梯度。但是，优化器本身并不知道是对所有样本（朴素）还是对其随机子集（随机）进行了损失计算，因此两种情况下的算法相同。

无论如何，请使用您喜欢的新型优化程序：

```
# In[7]:  
t_p = model(t_u, *params)  
loss = loss_fn(t_p, t_c)  
loss.backward()  
  
optimizer.step()  
  
params  
  
# Out[7]:  
tensor([ 9.5483e-01, -8.2600e-04], requires_grad=True)
```

调用 step 时，params 的值已更新，您无需亲自操作它！发生的事情是，优化器通过从中减去 learning\_rate 和 grad 的乘积来查看 params.grad 和更新的 params，与您以前的手动编写代码完全相同。

您准备好将此代码放在训练循环中了吗？不！大陷阱几乎吸引了您 您忘了将梯度归零。如果您在循环中调用了前面的代码，则在每次调用 backwards 时，梯度都会在叶子节点中累积，并且梯度下降遍布到各处！

这是循环就绪的代码，在正确的位置（在调用 backwards 之前）有额外的 zero\_grad：

```
# In[8]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)  
learning_rate = 1e-2  
optimizer = optim.SGD([params], lr=learning_rate)  
  
t_p = model(t_un, *params)  
loss = loss_fn(t_p, t_c)  
optimizer.zero_grad() ←———— As before, the placement of this call is somewhat arbitrary. It could be earlier in the loop as well.  
loss.backward()  
optimizer.step()  
  
params  
  
# Out[8]:  
tensor([1.7761, 0.1064], requires_grad=True)
```

完美！看看 optim 模块如何帮助您抽象出特定的优化方案？您所要做的就是为其提供一个参数列表（该列表可能非常长，这对于深度神经网络模型而言是必需的），然后不再需要关注细节。

相应地更新您的训练循环：

```
# In[9]:
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params

# In[10]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate) ←
    training_loop(
        n_epochs = 5000,
        optimizer = optimizer,
        params = params,
        t_u = t_u,
        t_c = t_c)

# Out[10]:
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([-5.3671, -17.3012], requires_grad=True)
```

It's important that both  
params here are the same  
object; otherwise, the  
optimizer won't know what  
parameters the model used.

同样，您将获得与以前相同的结果。太棒了！您进一步确认您知道如何手动降低梯度！要测试更多优化器，您要做的就是实例化一个不同的优化器，例如 Adam，而不是 SGD。其余代码保持不变。这些东西很方便。

我们不会详细介绍 Adam，但是可以说，它是一种更复杂的优化器，其中自适应地设置了学习率。此外，它对参数缩放的敏感度要低得多，因为它不敏感，您可以返回使用原始（非标准化）输入  $t_u$  甚至将学习率提高到  $1e-1$ 。Adam 算法甚至都不会变化：

```

# In[11]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate) ← New optimizer class here.

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u, ← Note that you're back to the
    t_c = t_c) original t_u as input.

# Out[11]:
Epoch 500, Loss 7.612901
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928578
Epoch 2000, Loss 2.927646

tensor([ 0.5367, -17.3021], requires_grad=True)

```

优化器并不是您训练循环中唯一灵活的部分。将注意力转移到模型上。要针对相同的数据和相同的损失训练神经网络，您只需更改 `model` 函数即可。在这种情况下，这样做是没有意义的，因为您知道将摄氏温度转换为华氏温度就是线性转换。神经网络使您可以去除关于应近似函数的形状的任意假设。即使这样，即使基本过程是高度非线性的（例如，用句子描述图像的情况下），神经网络也要能够成功进行训练。

我们触及了许多基本概念，使您可以在训练复杂的深度学习模型的同时了解内部情况：反向传播以估计梯度，自动梯度，并通过使用梯度下降或其他优化器来优化模型的权重。我们没有太多要讲的了。其余的大部分填补了空白，无论它们有多广泛。

接下来，我们讨论如何分割样本，从而为学习更好地控制自动梯度。

## 4.2.2 训练、验证和过拟合

约翰内斯·开普勒 (Johannes Kepler) 将部分数据保留，以便他可以在独立的观察结果上验证他的模型，这是一件至关重要的事情，特别是当您采用的模型可以近似任何形状的函数时，例如在神经网络中。换句话说，适应性强的模型往往会使用其许多参数来确保在数据点处的损失最小，但是您无法保证该模型在远离数据点或在数据点之间表现良好。毕竟，这就是您要优化器要做的一切：将数据点的损失降至最低。可以肯定的是，如果您有独立的数据点，而不是用来评估损失或沿其负梯度下降，那么您很快就会发现，在那些独立的数据点上评估损失会产生高于预期的损失。我们已经提到过这种现象，这称为过拟合。

您可以采取的打击过拟合的第一步是认识到可能会发生。为此，正如开普勒 (Kepler) 在 1600 年所想，您必须从数据集中取出几个数据点（验证集），并将模型拟合到其余数据点（训练集），如图 4.11 所示。然后，在拟合模型时，您可以在训练集上一次评估损失，在验证集上一次评估损失。当您尝试确定是否已将模型适合数据时，您必须查看每个数据集！

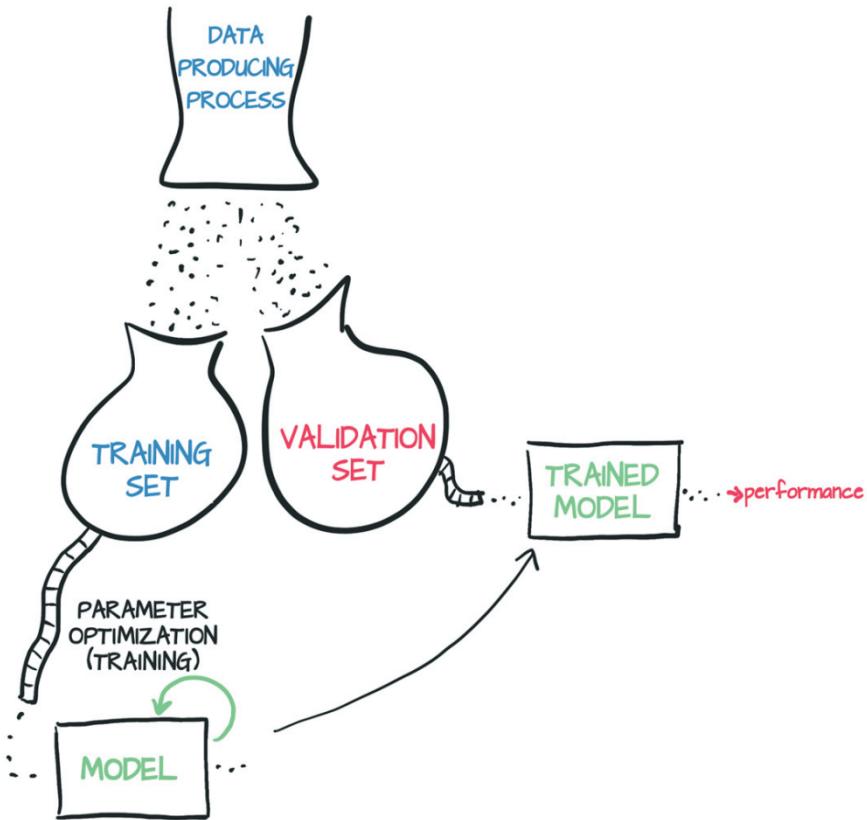


图 4.11 数据生成过程的概念表示以及训练数据和独立验证数据的收集和使用

第一个数字是训练损失，它告诉您模型是否完全拟合训练集，换句话说，模型是否具有足够的能力来处理数据中的相关信息。如果您神秘的温度计设法通过使用对数刻度来测量温度，那么糟糕的线性模型将没有机会拟合这些测量值，并且无法正确地转换为摄氏度。在这种情况下，您的训练损失（您在训练循环中打印的损失）将在接近零之前就停止下降。

深度神经网络可以潜在地近似复杂的函数，前提是神经元（即参数）的数量足够高。参数越少，网络将能够近似的函数形状越简单。因此，这是一条规则：如果训练损失没有减少，则该模型对于数据来说太简单了。另一种可能性是您的数据不包含有意义的信息以用于解释输出。如果商店里的家伙卖给您的是气压计而不是温度计，那么即使您使用魁北克省的最新神经网络架构（<https://www.umontreal.ca/en/artificialintelligence>），也无法仅凭压力就预测摄氏温度。

验证集如何？好吧，如果在验证集中评估的损失没有随训练集一起减少，则您的模型正在改善其在训练过程中看到的样本的拟合度，但并不能将其泛化到该精确训练集之外的样本。在新的、以前不可见的点上评估模型时，损失函数的值很差。这是第二条规则：如果训练损失和验证损失相差很大，则说明您过拟合了。

在这里，我们将以温度计为例来深入研究这种现象。您可能已经决定使用更复杂的函数来拟合数据，例如分段多项式或大型神经网络。如图 4.12 所示，此函数可以生成一个蜿蜒穿过数据点的模型，因为它将损失逼近于零。由于功能远离数据点的行为不会增加损失，因此没有任何方法可以使模型远离训练数据点。

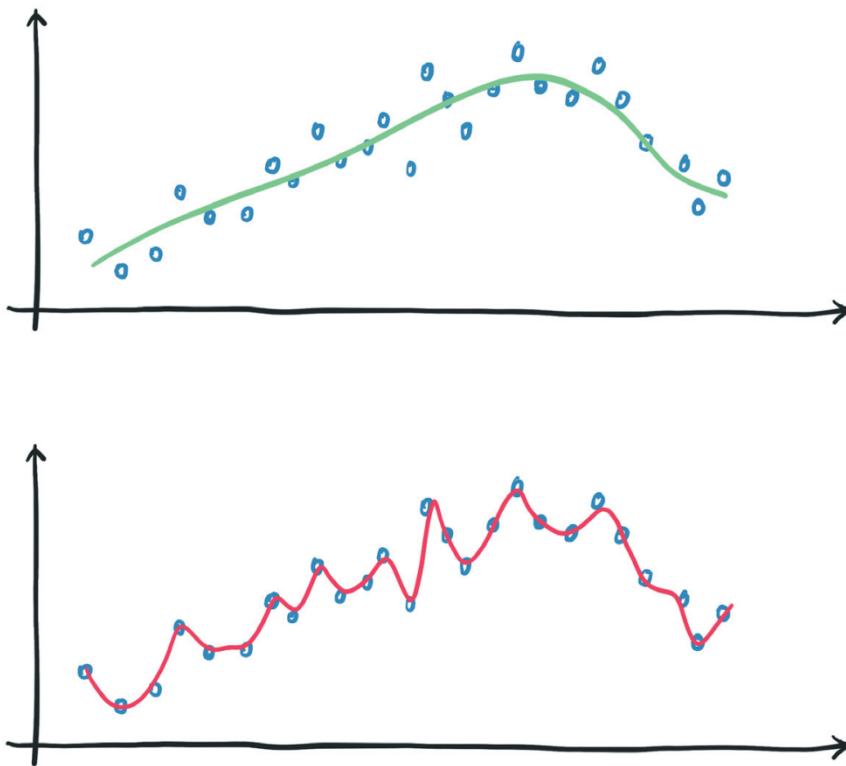


图 4.12 过度拟合的极端示例

不过，有什么解决办法？问得好。过度拟合似乎是一个问题，要确保模型在数据点之间的行为对于您尝试近似的过程是明智的。首先，您应确保为该过程获取足够的数据。如果您通过定期以低频率采样从正弦过程中收集数据，则很难为它拟合模型。

假设您有足够的数据点，则应确保能够拟合训练数据的模型在数据点之间尽可能规则。您有几种方法可以实现此目标。一种方法是在损失函数中添加所谓的惩罚项，以使模型的行为更平稳，变化更慢（在一些点上）。另一种方法是在输入样本中添加噪声，在训练数据样本之间人为地创建新的数据点，并迫使模型也尝试拟合它们。其他几种方式与前面的方式有些相关。但是，至少作为第一步，您可以为自己做的最大努力就是简化模型。从直观的角度来看，较简单的模型可能无法像较复杂的模型那样完美地拟合训练数据，但在数据点之间的行为可能会更规则。

您在这里做了一些不错的权衡。一方面，您需要进行建模以具有足够的能力来拟合训练集。另一方面，模型需要避免过拟合。因此，就参数而言，选择正确大小的神经网络模型的过程基于两个步骤：增大直到拟合为止，然后缩小直到不再过拟合。

您的生活将是拟合与过拟合之间的平衡。您可以通过以相同的方式对  $t_u$  和  $t_c$  进行打乱，然后将生成的张量随机分为两部分，将数据分为训练集和验证集。

对张量的元素进行打乱等于找到其索引的排列。randperm 函数执行以下操作：

```
# In[12]:  
n_samples = t_u.shape[0]  
n_val = int(0.2 * n_samples)  
  
shuffled_indices = torch.randperm(n_samples)  
  
train_indices = shuffled_indices[:-n_val]  
val_indices = shuffled_indices[-n_val:]  
  
train_indices, val_indices ← Because these values are random, don't be surprised  
# Out[12]: if your values end up being different from here on.  
(tensor([ 8,  0,  3,  6,  4,  1,  2,  5, 10]), tensor([9, 7]))
```

您将获得索引张量，可用于从数据张量开始构建训练集和验证集：

```
# In[13]:  
train_t_u = t_u[train_indices]  
train_t_c = t_c[train_indices]  
  
val_t_u = t_u[val_indices]  
  
val_t_c = t_c[val_indices]  
  
train_t_un = 0.1 * train_t_u  
val_t_un = 0.1 * val_t_u
```

您的训练循环不会改变。您想评估每个 epoch 的验证损失，以便有机会识别模型是否过拟合。

```

# In[14]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c,
                 val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)           ←
        train_loss = loss_fn(train_t_p, train_t_c)       ← These two pairs of lines are
                                                        the same except for the
        val_t_p = model(val_t_u, *params)               ←
        val_loss = loss_fn(val_t_p, val_t_c)             ← train_* vs. val_*
                                                        inputs.

        optimizer.zero_grad()                         ← Note that you have no val_loss.backward()
        train_loss.backward()                         ← here because you don't want to train the model on
        optimizer.step()                            ← the validation data.

    if epoch <= 3 or epoch % 500 == 0:
        print('Epoch {}, Training loss {}, Validation loss {}'.format(
            epoch, float(train_loss), float(val_loss)))

    return params

# In[15]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    params = params,
    train_t_u = train_t_un,          | Because you're using SGD again, you're
    val_t_u = val_t_un,              | back to using normalized inputs.
    train_t_c = train_t_c,
    val_t_c = val_t_c)

# Out[15]:
Epoch 1, Training loss 88.59708404541016, Validation loss 43.31699752807617
Epoch 2, Training loss 34.42190933227539, Validation loss 35.03486633300781
Epoch 3, Training loss 27.57990264892578, Validation loss 40.214229583740234
Epoch 500, Training loss 9.516923904418945, Validation loss 9.02982234954834
Epoch 1000, Training loss 4.543173789978027, Validation loss
2.596876621246338
Epoch 1500, Training loss 3.1108808517456055, Validation loss
2.9066450595855713
Epoch 2000, Training loss 2.6984243392944336, Validation loss
4.1561737060546875
Epoch 2500, Training loss 2.579646348953247, Validation loss
5.138668537139893
Epoch 3000, Training loss 2.5454416275024414, Validation loss
5.755766868591309

tensor([-5.6473, -18.7334], requires_grad=True)

```

在这里，我们对模型并不完全公平。验证集很小，因此验证损失仅在一定程度上有意义。无论如何，请注意，验证损失比您的训练损失要高，尽管相差不大。由于模型参数是由训练集定型的，因此可以预期模型在训练集上表现更好的事实。您的主要目标是同时减少训练损失和验证损失。尽管在理想情况下，只要验证损失与训练损失保持合理的距离，这两个损失将大致相同，但是您知道您的模型正在继续学习有关数据的泛化知识。在图 4.13 中，情况 C 是理想的，而情况 D 是可接受的。在情况 A 中，模型根本没有学习，在情况 B 中，您看到过拟合。

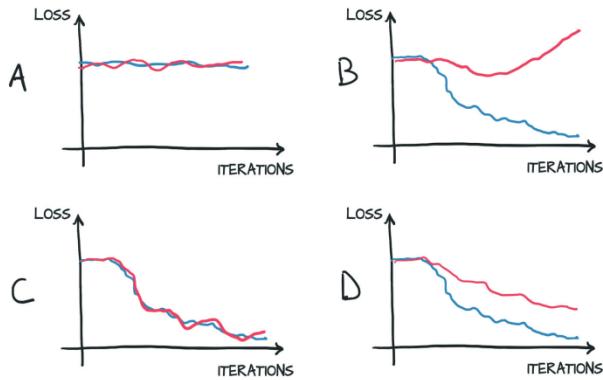


图 4.13 训练（蓝色）和验证（红色）损失的过拟合场景。（A）训练和验证损失不会减少；由于数据中没有信息或模型能力不足，因此该模型无法学习。（B）训练损失减少而验证损失增加（过度拟合）。（C）训练损失和验证损失同时减少；由于该模型并非处于过拟合的极限，因此性能可能会进一步提高。（D）训练和验证损失具有不同的绝对数值，但趋势相似；过拟合在控制之下。

### 4.2.3 自动微分中的小问题并将其关闭

从训练循环中，您可以了解到，您只能在 `train_loss` 上调用 `backward` 函数。因此，误差只会根据训练集反向传播。验证集用于根据未用于训练的数据对模型输出的准确性进行独立评估。

好奇的读者此时将有一个问题的雏形。对模型进行两次评估（一次在 `train_t_u` 上，然后在 `val_t_u` 上），然后调用 `backward`。这不会使 autograd 变得混乱吗？验证集传递过程中生成的值会不会影响 `backward` 函数？

幸运的是，事实并非如此。训练循环中的第一行在 `train_t_u` 上计算 `model` 以产生 `train_t_p`。然后从 `train_t_p` 评估 `train_loss`，创建一个将 `train_t_u` 到 `train_t_p` 链接到 `train_loss` 的计算图。在 `val_t_u` 上再次评估模型时，将生成 `val_t_p` 和 `val_loss`。在这种情况下，将创建一个单独的计算图，该图将 `val_t_u` 到 `val_t_p` 链接到 `val_loss`。单独的张量已通过相同的函数 `model` 和 `loss_fn` 运行，生成了单独的计算图，如图 4.14 所示。

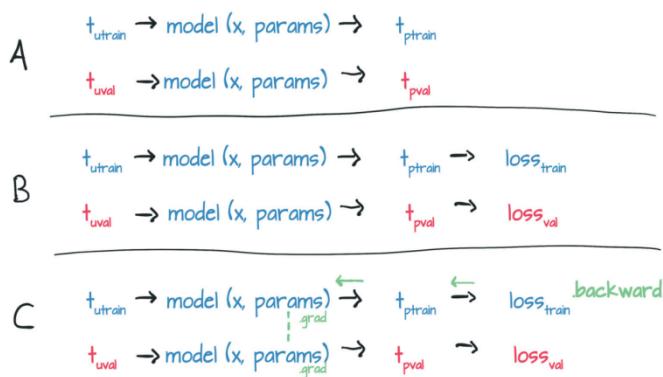


图 4.14 该图显示了当在其中一个上调用`.backward` 时，梯度如何通过具有两个损耗的图传播

这两个图唯一的共同点是参数。当在 `train_loss` 上调用 `backward` 函数时，将在第一个图形上运行 `backward` 函数。换句话说，您可以根据 `train_t_u` 生成的计算来对 `train_loss` 相对于参数的求得的导数进行累加。

如果您还（错误地）在 `val_loss` 上调用了 `backward` 函数，那么您将已经积累了 `val_loss` 相对于同一叶节点上的参数的导数。还记得 `zero_grad` 的事情吗，除非您明确地将梯度归零，否则每次您向后调用时，梯度都会在此之上累积？好吧，这里会发生类似的事情：在 `val_loss` 上调用 `backward` 函数会导致在 `train_loss.backward()` 调用期间生成的梯度之上，在参数 `params` 张量中累积的梯度。在这种情况下，您可以在整个数据集（训练和验证）上有效地训练模型，因为坡度将取决于两者。非常有趣。

这里是另一个讨论的元素：因为您永远不会在 `val_loss` 上调用 `backward` 函数，所以为什么要首先构建图形？实际上，您可以将 `model` 和 `loss_fn` 作为普通函数来调用，而无需跟踪历史记录。跟踪历史记录无论经过优化如何，都会带来额外的代价，您可以在验证过程中放弃这些代价，尤其是当模型具有数百万个参数时。

为了解决这种情况，PyTorch 允许您在不需要时使用 `torch.no_grad` 上下文管理器关闭自动梯度。就小问题而言，在速度或内存消耗方面，您将看不到任何明显的优势。但是对于较大的型号，差异可能会加起来。您可以通过检查 `val_loss` 张量上的 `require_grad` 属性的值来确保此上下文管理器正常工作：

```
# In[16]:  
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c,  
                  val_t_c):  
    for epoch in range(1, n_epochs + 1):  
        train_t_p = model(train_t_u, *params)  
        train_loss = loss_fn(train_t_p, train_t_c)  
  
        with torch.no_grad():      ←  
            val_t_p = model(val_t_u, *params)           | Context manager here.  
            val_loss = loss_fn(val_t_p, val_t_c)  
            assert val_loss.requires_grad == False       ←  
                | All requires_grad args  
                | are forced to False  
                | inside this block.  
        optimizer.zero_grad()  
        train_loss.backward()  
        optimizer.step()
```

使用相关的 `set_grad_enabled` 上下文，您还可以根据布尔表达式（通常表示您是在训练还是在推理），将代码设置为在启用或禁用 `autograd` 的情况下运行。您可以定义一个 `calc_forward` 函数，该函数接受输入中的数据，并根据布尔参数 `train_is` 运行带有或不带有 `autograd` 的 `model` 函数和 `loss_fn` 函数：

```
# In[17]:  
def calc_forward(t_u, t_c, is_train):  
    with torch.set_grad_enabled(is_train):  
        t_p = model(t_u, *params)  
        loss = loss_fn(t_p, t_c)  
    return loss
```

## 结论

本章从一个大问题开始：机器如何从示例中学习？我们在本章的其余部分中描述了可以优化模型以拟合数据的机制。我们选择坚持使用简单的模型来显示所有运动部件而没有不必要的复杂性。

## 练习题

将模型重新定义为  $w2 * t\_u ** 2 + w1 * t\_u + b$ 。

—必须更改训练循环的哪些部分等以适应此重新定义？

·哪些部分与替换模型无关？

—训练后产生的损失变得更高还是更低？

·结果是更好还是更坏？

## 总结

·线性模型是用于拟合数据的最简单的合理模型。

·凸优化技术可以用于线性模型，但它们不能推广到神经网络，因此本章重点介绍参数估计。

·深度学习可用于并非为解决特定任务而设计的通用模型，但可以自动调整以专门解决当前的问题。

·学习算法等于根据观察结果优化模型参数。损失函数是对执行任务中的错误的一种度量，例如预测输出和测量值之间的错误。优化目标是使损失函数尽可能低。

·`loss` 损失函数相对于模型参数的变化率可用于在减少损失的方向上更新相应的参数。

·PyTorch 中的 `optim` 模块提供了一组现成的优化器，用于更新参数和最小化损失函数。

·优化器使用 PyTorch 的 `autograd` 功能来计算每个参数的梯度，具体取决于该参数对最终输出的贡献程度。此功能允许用户在复杂的前向传播过程中依赖动态计算图。

·上下文管理器（例如 `torch.no_grad()`）：可用于控制 `autograd` 行为。

·数据通常分为训练样本和验证样本的独立集合，从而模型可以对未经训练的数据进行评估。

·当模型的性能在训练集上持续提高但在验证集上下降时，就发生了模型过拟合。这种情况通常发生在模型无法泛化时，而是只是记忆训练集所需的输出。

## 5 使用神经网络来拟合你的数据

本章覆盖内容有：

- 使用非线性激活函数作为与线性模型的主要区别
- 常用的多种激活函数

PyTorch 的 nn 模块 书中所有神经网络模块都来自这个模块

您已仔细研究了线性模型如何学习以及如何在 PyTorch 中实现它，重点研究了一个简单的回归问题，该问题需要具有一个输入一个输出的线性模型。这个简单的示例使您可以剖析模型学习的机制，而不会因模型本身的实现而过度分散注意力。不管基础模型是什么，向参数反向传播误差，然后通过采用相对于损耗的梯度来更新这些参数，这些部分都是相同的（图 5.1）。

在本章中，您将对模型架构进行更改。您将实现一个完整的人工神经网络来解决您的问题。

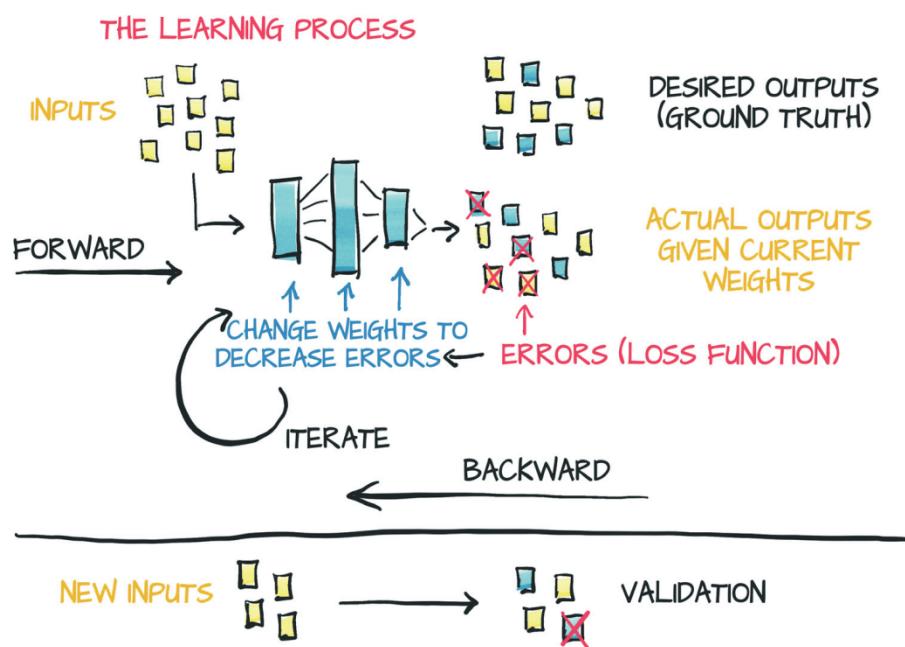


图 5.1 学习过程的思想模型

您的温度计转换训练循环保持不变，华氏度到摄氏温度的样本分为训练集和验证集也保持不变。您可以开始使用二次模型，将模型重写为其输入的二次函数（例如  $y = a * x^{**} 2 + b * x + c$ ）。因为这个模型是可微的，所以 PyTorch 将负责计算梯度，并且训练循环将照常工作。不过，这对您来说并不太有趣，因为您仍在调整函数的形状。

在本章中，您将开始了解使用 PyTorch 功能进行的基础工作，这些功能将在您处理项目

时日复一日地使用。您将了解 PyTorch API 内部的情况，而不仅仅把它当做黑魔法来使用。

不过，在开始实现新模型之前，我们将解释人工神经网络的含义。

## 5.1 人工神经元

深度学习的核心是神经网络，即能够通过简单函数的组合来表示复杂函数的数学实体。神经网络这个词显然暗含了它与人类大脑工作方式的联系。实际上，尽管最初的模型是受神经科学 (<http://psycnet.apa.org/doiLanding?doi=10.1037%2Fh0042519>) 启发的，但现代人工神经网络仅与大脑中神经元的机制有模糊的相似之处。人工和生理神经网络似乎可以使用模糊近似的数学策略来近似复杂的函数，因为该策略族有效地起作用。

**注意** 我们将从这里开始丢弃“人工”，并将这些结构称为神经网络。

这些复杂函数的基本组成部分是神经元，如图 5.2 所示。从本质上讲，神经元不过是输入的线性变换（例如，输入乘以数字[权重]，再加上常数[偏置]），然后应用固定的非线性函数（称为激活函数）。

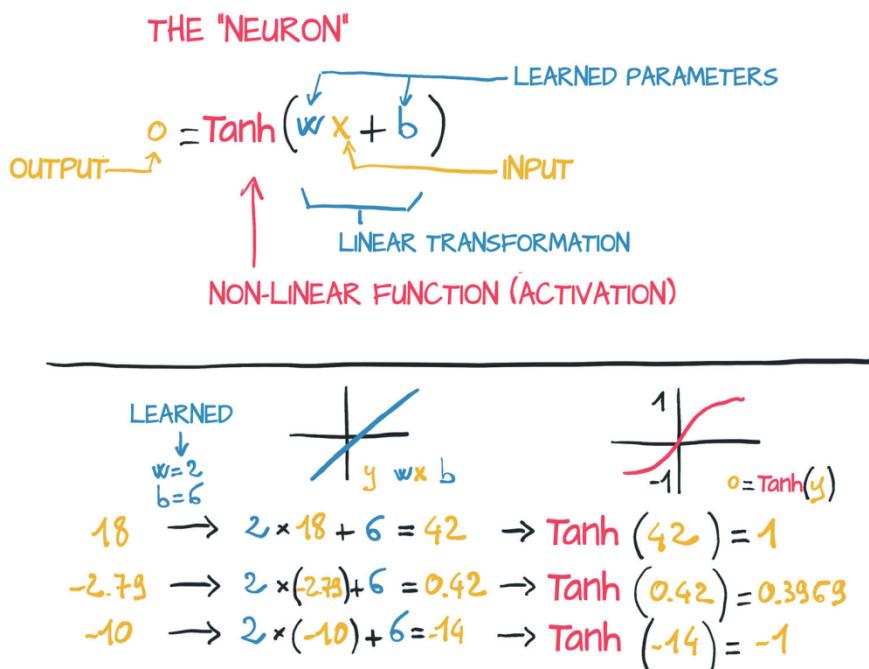


图 5.2 人工神经元：包含在非线性函数中的线性变换

在数学上，您可以将其写为  $o = f(w * x + b)$ ，其中  $x$  为输入， $w$  为权重或缩放因子， $b$  为偏置或偏移。 $f$  是激活函数，在此处设置为双曲正切或“tanh”函数。通常， $x$  以及由此得到的  $o$  可以是简单的标量，也可以是矢量值（含有许多标量值）。类似地， $w$  可以是单个标量或矩阵，而  $b$  是标量或向量（但是输入和权重的维数必须匹配）。在后一种情况下，该表达式被称为神经元层，因为它通过用多维权重和多维偏置表示许多神经元组。

如图 5.3 所示，多层神经网络由上述功能组成：

```

x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
y = f(w_n * x_n + b_n)

```

其中神经元层的输出用作下一层的输入。请记住，这里的  $w_0$  是矩阵，而  $x$  是向量！在此使用向量可使  $w_0$  表示整个神经元层，而不仅仅是使用单个权重。

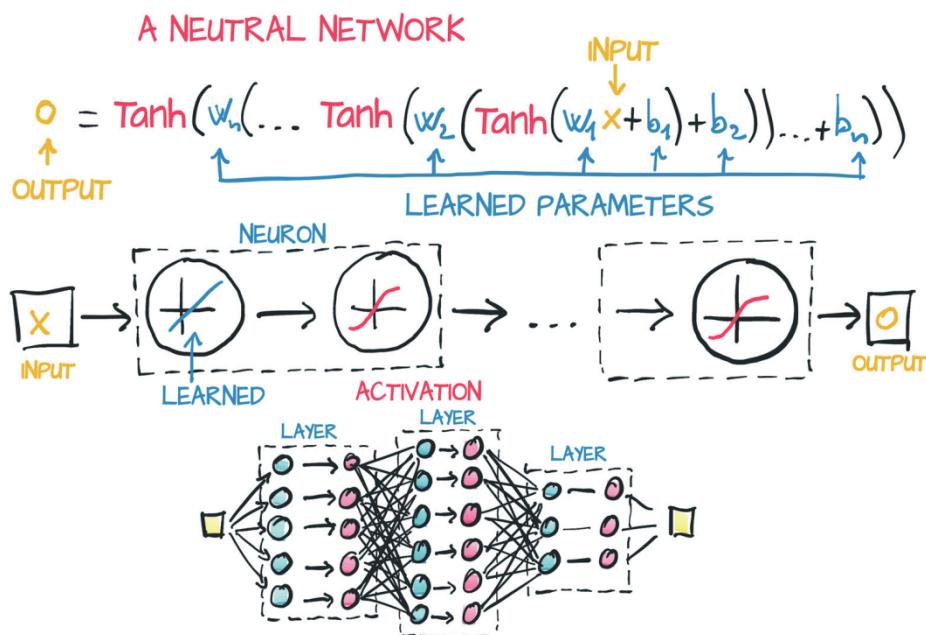


图 5.3 具有三层的神经网络

较早的线性模型与您将用于深度学习的模型之间的重要区别是误差函数的形状。线性模型和误差平方损失函数具有凸的误差曲线，具有单一的、明确定义的最小值。如果要使用其他方法，则可以自动确定地解决它。您的参数更新尝试尽其所能估计单个正确的答案。

即使使用相同的误差平方损失函数，神经网络也不具有凸误差表面的相同属性。您尝试近似的每个参数都没有一个正确的答案。相反，您尝试获取所有参数来协同工作以产生有用的输出。由于有用的输出只会逼近真值，因此会有一定程度的不完善。这些缺陷在何处以及如何表现是有些任意的，并且通过暗示，控制输出的参数（以及缺陷）也有些任意的。从机械角度来看，此输出结果使神经网络训练看起来很像参数估计，但是您必须记住，理论基础是完全不同的。

神经网络之所以具有非凸误差表面的很大一部分原因是由于激活函数。一组神经元逼近各种有用函数的能力取决于每个神经元固有的线性和非线性的行为组合。

### 5.1.1 您需要的只需激活函数

(深度) 神经网络中最简单的单位是线性运算（缩放+偏移），然后是激活函数。您在最新

模型中进行了线性运算；线性操作是整个模型。激活函数的作用是将先前线性操作的输出集中到给定范围内。

假设您为图像分配了“好狗狗”分数。金毛猎犬和西班牙猎狗的照片应该评分很高；飞机和垃圾车的图像应得分较低。熊图片也应具有较低的分数，尽管比垃圾车的分数高一点。

问题是您必须定义高分。因为您已经可以使用 float32 的全部范围，所以可以提高很多。即使您说“这是 10 分制”，有时您的模型也会超过 10 分而得到 11 分。请记住，在幕后，这都是  $w * x + b$  矩阵乘法的总和，这自然不会将自己限制在特定的输出范围内。

您想要做的就是将线性操作的输出严格地限制在特定范围内，这样该输出的使用者就不必处理幼犬的数字输入为 12/10，熊的数字输入为 -10，垃圾车的数字输入为 -1000。

一种可能性是限制输出值。零以下的任何数都将设置为零，而十以上的任何数都将被设置为 10。您可以使用一个简单的激活函数

`torch.nn.Hardtanh` (<https://pytorch.org/docs/stable/nn.html#hardtanh>)。

另一个运作良好的函数族是 `torch.nn.Sigmoid`，它是  $1 / (1 + e^{-x})$ ，`torch.tanh` 以及您稍后会看到的其他函数。这些函数的曲线随着  $x$  趋于负无穷大而渐近地接近零或负 1，随着  $x$  的增加而靠近 1，并且在  $x = 0$  时具有大致恒定的斜率。从概念上讲，这种形状的函数可以很好地工作，因为这意味着您的神经元（即线性函数，然后是激活函数）将对线性函数输出中间的区域敏感；其他所有东西都集中在边界值旁边。如图 5.4 所示，垃圾车的得分为 -0.97，而熊，狐狸和狼的得分可能在 -0.3 到 0.3 之间。

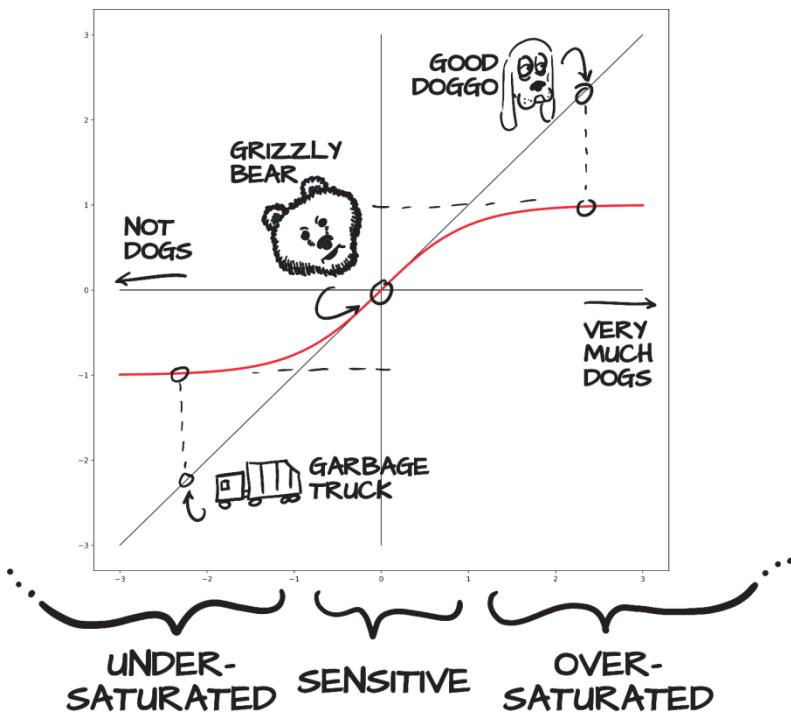


图 5.4 通过 `tanh` 激活函数将狗、熊和垃圾车映射为“狗的相像度”

垃圾车被标记为“不是狗”，狗被映射“显然是狗”的标签上，而熊最终停在中间。在代码中，

您将看到确切的值：

```
>>> import math  
>>> math.tanh(-2.2)      ← Garbage truck  
-0.9757431300314515  
>>> math.tanh(0.1)        ← Bear  
0.09966799462495582  
>>> math.tanh(2.5)        ← Good doggo  
0.9866142981514303
```

当熊处于敏感范围内时，熊的微小变化会导致结果发生明显变化。您可以从灰熊换成北极熊（北极熊的脸庞似乎更像是传统的犬科），并且随着向图的“非常像狗”的一端移动时，你可以看到 Y 轴上的跳跃。相反，考拉熊长得不太像狗，并且你会看到激活后的输出发生下降。不过，您可以对垃圾车做很多事情，使其像狗一样登记。即使发生了巨大的变化，您也可能只会看到从 -0.97 到 -0.8 左右的变化。

有许多激活函数可用，其中一些如图 5.5 所示。在第一列中，您将看到连续函数 Tanh 和 Softplus。第二列左侧是 Hardtanh 和 ReLU 激活功能的“硬 hard”版本。ReLU（修正线性单元）特别值得注意，因为它被认为是性能最好的常用激活函数之一，因为许多最新的技术都使用了它。Sigmoid 激活函数（也称为 logistic 函数）已在早期深度学习工作中广泛使用，但已不常用。最后，LeakyReLU 函数将标准 ReLU 修改为具有小的正斜率，而不是对于负输入斜率严格为零。（该斜率通常为 0.01，但为清楚起见，此处显示的斜率为 0.1。）

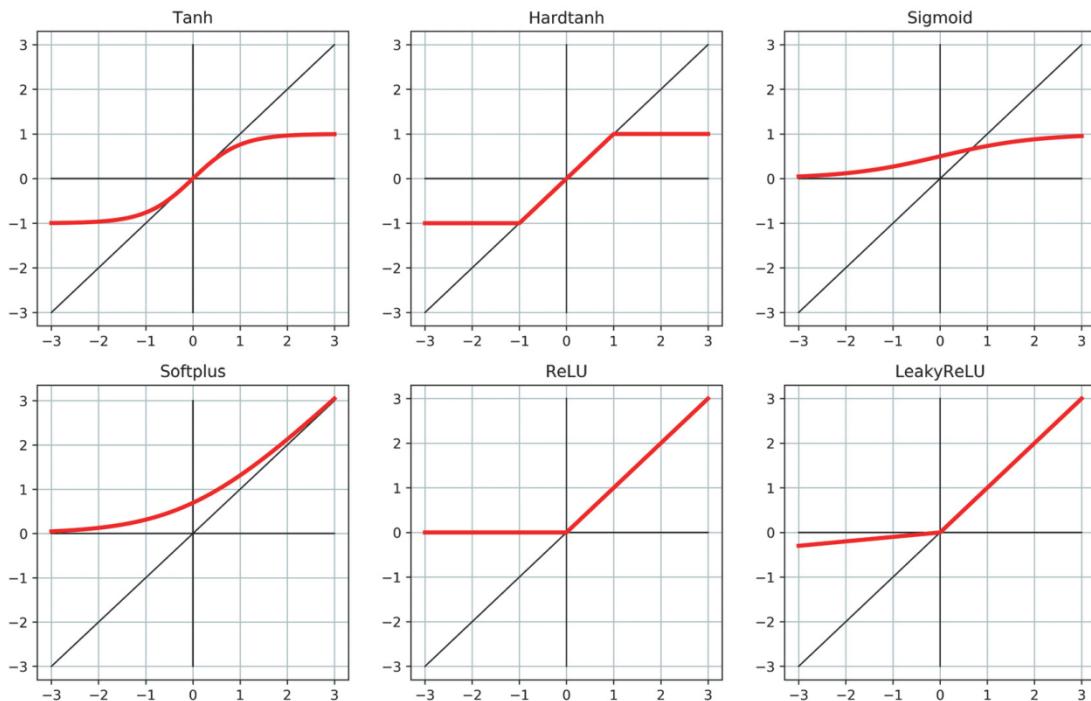


图 5.5 常用和不太常用的激活函数的集合

激活函数很奇特，因为有如此众多被证明成功的函数（很多函数都没有在图 5.5 中列举

出），很明显几乎没有严格的要求。因此，我们将讨论有关激活函数的一些一般性，这些一般性可能在特定的领域会被证明是错的。也就是说，根据定义

(<https://openai.com/blog/nonlinear-computation-in-linear-networks/>)，激活函数是：

- 是非线性的-重复使用  $w \cdot x + b$  而没有多项式的激活函数。非线性可以让整个网络近似更复杂的函数。
- 具有可微性-它们具有可微性，因此可以通过它们计算梯度。正如您在 Hardtanh 或 ReLU 中看到的那样，点不连续性很好。

没有这些函数，网络要么会退回到复杂的多项式，要么变得难以训练。

函数通常也具有如下特点（尽管并非全部如此）：

- 至少要有一个敏感范围，其中输入的微小变化会导致输出相应的微小变化。
- 至少具有一个不敏感（或饱和）范围，其中输入的变化导致的输出变化很小甚至没有变化。

举例来说，由于结合了具有不同权重和输入偏置的敏感范围，因此 Hardtanh 函数可以轻松地用于函数的分段线性近似。

激活功能通常（但并非普遍如此）具有以下至少一项：

- 当输入变为负无穷大时接近（或达到）其下限
- 对于正无穷大的输入产生相似但相反的上限

考虑一下您对反向传播的工作原理的了解，可以发现，当输入处于响应范围内时，误差将通过激活函数更有效地向后传播，而误差不会严重影响输入饱和的神经元（因为由于输出在平坦区域，梯度将接近零）。

综上所述，此机制非常强大。我们的意思是，在由线性+激活单元构成的网络中，当向网络提供不同的输入时，(a) 不同的单元对相同的输入做出不同的响应，并且 (b) 与这些输入相关的误差将主要影响在敏感范围内运行的神经元，而其他单元或多或少不受学习过程的影响。此外，由于激活相对于其输入的导数通常在敏感范围内接近 1，因此，对于在该范围内工作的设备，通过梯度下降来估计线性变换的参数看起来很像线性拟合中的情形。

您开始对如何将多个线性+激活单元并行连接并一个接一个地堆叠到一个能够近似复杂函数的数学对象上有了更深入的了解。单元的不同组合对不同范围内的输入做出响应，并且对于这些参数，通过梯度下降相对容易优化，因为学习的行为将非常类似于线性函数，直到输出饱和为止。

### 5.1.2 学习对于神经网络意味着什么

从线性变换堆栈中进行模型构建，然后进行可微分的激活，可以生成可以拟合高度非线性过程的模型，并且您可以通过梯度下降很好地估计其参数。即使您要处理具有数百万个参数的

模型，这一事实仍然适用。使用深度神经网络之所以如此吸引人，是因为它使您不必担心表示数据的确切函数（无论是二次多项式、分段多项式还是其他函数）。使用深度神经网络模型，您将获得通用逼近器和估算其参数的方法。通过构建简单的搭建模块，可以根据模型容量及其对复杂的输入/输出关系进行建模的能力来定制该近似器。

图 5.6 显示了一些示例。

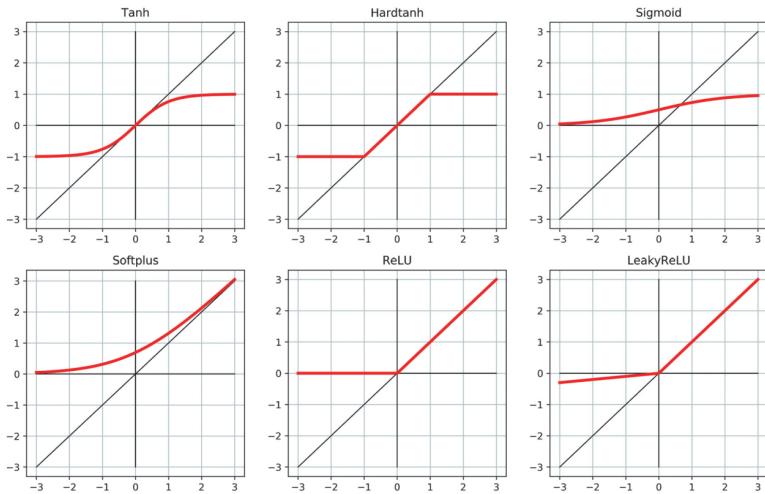


图 5.6 组合多个线性单元和 tanh 激活函数以产生非线性输出

左上方的四个图显示了四个神经元 A、B、C 和 D，每个神经元都有自己的（任意选择的）权重和偏置。每个神经元都使用 Tanh 激活函数，最小值为-1，最大值为 1。不同的权重和偏差会移动中心点，并极大地改变从最小值到最大值的过渡，但显然它们都是相同的一般形状。右边的列显示了加在一起的两对神经元（A+B，然后是 C+D）。在这里，您开始看到一些有趣的属性，它们模仿了单层神经元。A + B 显示出一条轻微的 S 曲线，其极值趋近于零，但正负都在中间。相反，C+D 仅具有较大的正凸点，其峰值比单神经元最大值高 1。

在第三行中，您开始组成神经元，就像它们在两层网络中一样。C (A + B) 和 D (A + B) 具有与 A + B 相同的正负颠簸，但正峰值更微妙。C (A + B) + D (A + B) 的成分显示出新的属性：两个清晰的负凸点，以及在主要关注区域左侧可能还有一个细微的第二正峰。所有这一切只发生在只有四个神经元的两层网络情况下！

同样，选择这些神经元的参数只是为了产生视觉上有趣的结果。训练包括找到这些权重和偏差的可接受值，以使所得网络正确执行一项任务，例如根据给定的地理坐标和一年中的时间预测可能的温度。通过成功完成一项任务，我们的意思是在与训练数据相同的数据生成过程所产生的不可见的数据上获得正确的输出。一个成功训练的网络，通过权重和偏置的值，以有意义的数值表示形式捕获了数据的固有结构，这些数字表示形式对于以前不可见的数据可以正常工作。

这是您实现学习机制的又一步：深度神经网络可让您近似高度非线性的现象，而无需为它们显式地建立模型。相反，从通用的未经训练的模型开始，您可以通过为它提供一组输入和输出以及一个从中进行反向传播的损失函数来将其专门用于特定任务。通过实例，将通用模型专门用于任务是我们所谓的学习，因为模型并不是在考虑特定任务的情况下构建的：模型中没有编码来描述该任务如何工作的规则。

根据您的温度计经验，您假设两个温度计都是线性测量温度的。这个假设就是我们隐式地为任务编码的地方：我们对输入/输出函数的形状进行了硬编码；除了对散布在一条线上的数据点之外，我们无法拟合。随着问题的维数的增长（许多输出和许多输入）和输入/输出关系变得复杂，假设输入/输出函数的形状不太可能办得到。物理学家或应用数学家的工作通常是根据第一原理来对现象进行功能描述，以便可以通过测量来估计未知参数并获得准确的世界模型。另一方面，深度神经网络是一系列函数，可以近似各种输入/输出关系，而不必要求提供一种现象的解释模型。在某种程度上，您要放弃解释，以换取解决日益复杂的问题的可能性。换句话说，有时您缺乏能力、信息或计算资源来为您所呈现的内容建立显式模型，因此数据驱动方法是您解决问题的唯一方法。

## 5.2 PyTorch 的 nn 模块

关于神经网络的所有这些讨论可能会让您对使用 PyTorch 从头开始构建一个神经网络感到好奇。第一步是将神经网络单元替换为线性模型。从正确性的角度来看，此步骤是一个毫无用处的步骤，因为您已经验证了校准只需要线性函数，但是对于开始一个足够简单的问题并在以后进行扩展仍然有用。

PyTorch 有一个专门用于神经网络的完整子模块 `torch.nn`。该子模块包含创建各种神经网络体系结构所需的构建块。这些构建块在 PyTorch 术语中称为模块（在其他框架中称为层）。

PyTorch 的模块是从 `nn.Module` 基类派生的 Python 类。一个模块可以具有一个或多个参数实例作为属性，这些实例是张量，其值在训练过程中进行了优化。（在线性模型中考虑 `w` 和 `b`。）一个模块还可以具有一个或多个子模块（`nn.Module` 的子类）作为属性，并且它也可以跟踪其参数。

**注意** 子模块必须是顶级属性，而不能装在 `list` 或 `dict` 实例中！否则，优化器将无法找到子模块（及其参数）。对于您的模型需要子模块列表或字典的情况，PyTorch 提供 `nn.ModuleList` 和 `nn.ModuleDict`。

毫不奇怪，您可以找到一个名为 `nn.Linear` 的 `nn.Module` 子类，该子类对其输入进行仿射变换（通过参数属性 `weight` 和 `bias`）；它相当于您之前在温度计实验中实施的方法。现在，从您上次中断的地方开始，然后将之前的代码转换为使用 `nn` 的形式。

所有 PyTorch 提供的 `nn.Module` 子类都定义了其 `call` 方法，使您可以实例化 `nn.Linear` 并将其像一个函数一样调用，如下面的清单所示。

### Listing 5.1 code/p1ch6/1\_neural\_networks.ipynb

```
# In[5]:  
import torch.nn as nn  
  
linear_model = nn.Linear(1, 1)  
linear_model(t_un_val)    ← You look into the constructor  
                          arguments in a moment.  
  
# Out[5]:  
tensor ([-0.9852],  
       [-2.6876]), grad_fn=<AddmmBackward>
```

使用一组参数调用 `nn.Module` 的实例最终将调用具有相同参数的名为 `forward` 的方法。`forward` 方法执行前向计算； `call` 方法在进行调用 `forward` 之前和之后还会执行其他相当重要的杂务。因此，从技术上讲，可以直接调用 `forward` 函数，并且产生与 `call` 相同的输出，但是不应从用户代码中完成：

```
>>> y = model(x)      ← Correct!  
>>> y = model.forward(x) ← Silent error.  
                           Don't do it!
```

以下清单显示了 `Module.call` 的实现（为清楚起见进行了一些简化）。

### Listing 5.2 torch/nn/modules/module.py, line 483, class: Module

```
def __call__(self, *input, **kwargs):  
    for hook in self._forward_pre_hooks.values():  
        hook(self, input)  
  
    result = self.forward(*input, **kwargs)  
  
    for hook in self._forward_hooks.values():  
        hook_result = hook(self, input, result)  
        # ...  
  
    for hook in self._backward_hooks.values():  
        # ...  
  
    return result
```

如您所见，如果直接使用 `.forward(...)`，很多钩连点将无法正确调用。

现在回到线性模型。`nn.Linear` 的构造函数接受三个参数：输入特征的数量，输出特征的数量以及线性模型是否包含偏置（此处默认为 `True`）。

```
# In[5]:  
import torch.nn as nn  
  
linear_model = nn.Linear(1, 1) ← The arguments are input size, output size,  
linear_model(t_un_val) and bias defaulting to True.  
  
# Out[5]:  
tensor([-0.9852], [-2.6876]), grad_fn=<AddmmBackward>
```

在这种情况下，特征的数量是指模块的输入和输出张量的大小，因此为 1 和 1。例如，如果在输入中同时使用了温度和气压，则在输入中将具有两个特征，在输出中具有一个特征。如您所见，对于具有多个中间模块的更复杂的模型，特征的数量与模型的能力有关。

您有一个具有一个输入和一个输出特征的 `nn.Linear` 实例，它需要一个权重和一个偏置

```
# In[6]:  
linear_model.weight  
  
# Out[6]:  
Parameter containing:  
tensor([-0.4992], requires_grad=True)  
  
# In[7]:  
linear_model.bias  
  
# Out[7]:  
Parameter containing:  
tensor([0.1031], requires_grad=True)
```

您可以使用一些输入来调用模块：

```
# In[8]:  
x = torch.ones(1)  
linear_model(x)  
  
# Out[8]:  
tensor([-0.3961], grad_fn=<AddBackward0>)
```

尽管 PyTorch 可以让您摆脱它，但是您没有提供正确维度的输入。您有一个可以接受一个输入并产生一个输出的模型，但是 PyTorch `nn.Module` 及其子类被设计为可以同时对多个样本进行处理。为了容纳多个样本，模块希望输入的第零维为批次中的样本数。

`nn` 中的任何模块都被写成是对于一批多个输入产生输出。因此，假设您需要对 10 个样本运行 `nn.Linear`，则可以创建大小为  $B \times N_{in}$  的输入张量，其中  $B$  是批处理的大小，而  $N_{in}$  是输入特征的数量，然后在模型中运行一次：

```
# In[9]:
x = torch.ones(10, 1)
linear_model(x)

# Out[9]:
tensor([[-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961],
       [-0.3961]], grad_fn=<AddmmBackward>)
```

图 5.7 显示了批图像数据的类似情况。输入为  $B \times C \times H \times W$ ，批处理大小为 3（例如，狗、鸟然后是汽车的图像），三个通道维度（红色、绿色和蓝色），高度和宽度的像素数未指定。

如您所见，输出是大小为  $B \times N_{out}$  的张量，其中  $N_{out}$  是输出特征的数量——在这种情况下为四。

我们要进行此批处理的原因是多方面的。一种主要的动机是确保我们要进行的计算足够大，以使我们用来执行计算的计算资源饱和。特别是 GPU 是高度并行化的，因此在小型模型上的单个输入将使大多数计算单元处于空闲状态。通过提供批次的输入，可以将计算分散到多个空闲单元中，这意味着批次的结果将像单个结果一样快地返回。另一个好处是，某些高级模型将使用整个批次中的统计信息，而随着批次大小的增加，这些统计信息会变得更好。

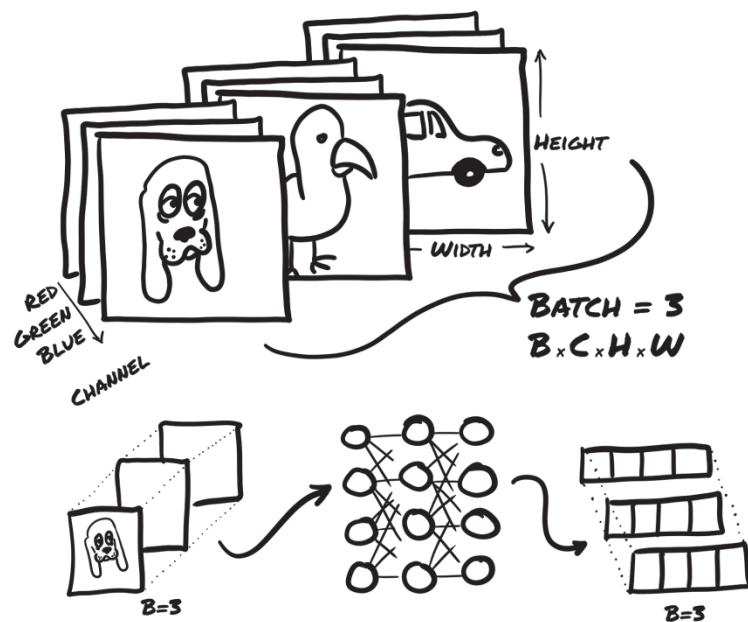


图 5.7 将三张 RGB 图像一起组成批并馈送入神经网络。输出的是大小为 4 的三个向量的批。

现在回到温度计数据。您的  $t_u$  和  $t_c$  是大小为  $B$  的两个一维张量。由于广播，您可以将线性模型写为  $w * x + b$ ，其中  $w$  和  $b$  是两个标量参数。该模型之所以有效，是因为您具有一个输入特征。如果您有两个，则需要添加额外的维度，以将该 1D 张量转变为矩阵，在行中表示不同的样本，在列中表示不同的特征。

这就是您要切换为使用 `nn.Linear` 所要做的。您可以将  $B$  输入重塑为  $B \times N_{in}$  维，其中  $N_{in}$  为 1。您可以使用 `unsqueeze` 函数轻松地做到这一点：

```
# In[2]:  
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]  
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]  
t_c = torch.tensor(t_c).unsqueeze(1) | Here, you add the extra dimension at axis 1.  
t_u = torch.tensor(t_u).unsqueeze(1)  
  
t_u.shape  
  
# Out[2]:  
torch.Size([11, 1])
```

完成了！现在更新您的训练代码。首先，将您的手工模型替换为 `nn.Linear(1,1)`；然后将线性模型参数传递给优化器：

```
# In[10]:  
linear_model = nn.Linear(1, 1) ← A redefinition from above.  
optimizer = optim.SGD(← You replace [params]  
    linear_model.parameters(), ← with this method call.  
    lr=1e-2)
```

早先，您负责创建参数并将其作为第一个参数传递给 `optim.SGD`。现在，您可以使用 `parameters` 方法，向任何 `nn.Module` 模块其或其任何子模块访问其拥有的参数列表：

```
# In[11]:  
linear_model.parameters()  
  
# Out[11]:  
<generator object Module.parameters at 0x0000020A2B022D58>  
  
# In[12]:  
list(linear_model.parameters())  
  
# Out[12]:  
[Parameter containing:  
 tensor([[0.3791]], requires_grad=True), Parameter containing:  
 tensor([-0.5349], requires_grad=True)]
```

该调用递归到模块的 `init` 构造函数中定义的子模块中，并返回遇到的所有参数的平面列表，因此您可以像之前一样方便地将其传递给优化器构造函数。

您已经可以弄清楚训练循环中会发生什么。向优化器提供了使用 `require_grad = True` 定义的张量列表。按照定义，所有参数都是通过这种方式定义的，因为它们需要通过梯度下降进行优化。调用 `training_loss.backward()` 时，在图的叶节点上梯度 `grad` 被累加，这些叶子节点正是传递给优化器的参数。

此时，SGD 优化器具有所需的一切。调用 `optimizer.step()` 时，它将迭代每个参数，并按与其 `grad` 属性中存储的内容成比例的数量对其进行更改，这是干净的设计。

现在看一下训练循环：

```
# In[13]:  
def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train, t_u_val,  
                 t_c_train, t_c_val):  
    for epoch in range(1, n_epochs + 1):  
        t_p_train = model(t_u_train)           ←  
        loss_train = loss_fn(t_p_train, t_c_train)  
        t_p_val = model(t_u_val)             ← Now the model is  
        loss_val = loss_fn(t_p_val, t_c_val)   passed in instead of  
                                              the individual params.  
        optimizer.zero_grad()               ← The loss function is also passed in.  
        loss_train.backward()              ← You'll use it in a moment.  
        optimizer.step()  
  
        if epoch == 1 or epoch % 1000 == 0:  
            print('Epoch {}, Training loss {}, Validation loss {}'.format(  
                  epoch, float(loss_train), float(loss_val)))
```

训练循环几乎没有改变，除了现在您不需要显式地将参数 `params` 传递给模型 `model`，因为模型内部本身保存了其参数。

您可以使用 `torch.nn` 的最后一些部分：损失。实际上，`nn` 带有几个常见的损失函数，其中一个是 `nn.MSELoss`（MSE 代表均方误差），这正是您先前定义的 `loss_fn`。`nn` 中的损失函数仍然是 `nn.Module` 的子类，因此创建一个实例并将其作为函数调用。在这种情况下，您将不再需要手写 `loss_fn`，并用该子类替换它：

```

# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train, ←
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print()
print(linear_model.weight)
print(linear_model.bias)

# Out[15]:
Epoch 1, Training loss 92.3511962890625, Validation loss 57.714385986328125
Epoch 1000, Training loss 4.910993576049805, Validation loss
1.173926591873169
Epoch 2000, Training loss 3.014694929122925, Validation loss
2.8020541667938232
Epoch 3000, Training loss 2.857640504837036, Validation loss
4.464878559112549
Parameter containing:
tensor([[5.5647]], requires_grad=True)
Parameter containing:
tensor([-18.6750], requires_grad=True)

```

You're no longer using your handwritten loss function from earlier.

输入到我们训练循环中的所有其他内容都保持不变。甚至我们的结果也和以前一样。当然，可以期望会获得相同的结果，因为若两者有差异将暗示这两种实现之一有 bug。

这是一段漫长的旅程，要探索这些二十多行的代码，需要进行很多探索。我们希望，到现在，不再变得魔幻，留下的是逻辑原理。在本章中学习的内容将使您可以拥有自己编写的代码，而不仅仅是在事情变得更加复杂时使用黑箱子。

您还需要采取最后一步：用神经网络代替线性模型作为近似函数。正如我们之前所说，使用神经网络不会产生更高质量的模型，因为校准问题背后的过程基本上是线性的。但是，在可控的环境中从线性网络跳到神经网络是很好的选择，这样您以后就不会感到迷惑了。

本节将固定所有其他内容，包括损失函数，并仅重新定义模型。您将构建最简单的可能的神经网络：一个线性模块，然后将一个激活函数输入另一个线性模块。由于历史原因，第一个线性+激活层通常称为隐藏层，因为它的输出不会直接观察到，而是会馈送到输出层。尽管模型的输入和输出大小均为 1（它们具有一个输入和一个输出特征），但是第一个线性模块的输出大小通常大于 1。回顾前面关于激活作用的解释，这种情况可能导致不同的单位对输入的不同范围做出响应，从而增加了模型的容量。最后一个线性层获取激活的输出，并将它们线性组合以产生输出值。

`nn` 提供了一种通过 `nn.Sequential` 容器连接模块的简单方法：

```
# In[16]:
seq_model = nn.Sequential(
    nn.Linear(1, 13),      ←
    nn.Tanh(),
    nn.Linear(13, 1))     ←
seq_model           This 13 must match
                     the first size, however.

# Out[16]:
Sequential(
    (0): Linear(in_features=1, out_features=13, bias=True)
    (1): Tanh()
    (2): Linear(in_features=13, out_features=1, bias=True)
)
```

I3 was chosen arbitrarily. We wanted to pick a number that was a different size from the other various tensor shapes floating around.

结果是一个模型，该模型采用指定为 `nn.Sequential` 的第一个模块期望的输入，将中间输出传递给后续模块，并产生最后一个模块返回的输出。该模型从 1 个输入特征中散开到 13 个隐藏特征，将通过 `tanh` 激活函数传递它们，并将得到的 13 个数字线性组合为 1 个输出特征。

调用 `model.parameters()` 可以从第一线性模块和第二线性模块中收集权重和偏置。在这种情况下，通过打印形状可以检查参数：

```
# In[17]:
[param.shape for param in seq_model.parameters()]

# Out[17]:
[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torch.Size([1])]
```

这些是优化器将获得的张量。同样，在调用 `model.backward()` 之后，所有参数都将保存在 `grad` 属性中，然后优化器会在调用 `optimizer.step()` 时相应地更新其值，这与之前的线性模型没有太大不同。毕竟，这两个模型都是可微分的模型，可以通过梯度下降进行训练。

关于 `nn.Modules` 中的参数的一些注意事项：当您检查由几个子模块组成的模型的参数时，可以方便地通过它们的名称来识别参数。有一种方法叫做 `named_parameters`：

```
# In[18]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[18]:
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

实际上，`Sequential` 中每个模块的名称都是该模块在参数中出现的顺序。有趣的是，`Sequential` 还接受一个有序字典 `OrderedDict`，您可以在其中为传递给 `Sequential` 的每个模块命名：

```
# In[19]:  
from collections import OrderedDict  
  
seq_model = nn.Sequential(OrderedDict([  
    ('hidden_linear', nn.Linear(1, 8)),  
    ('hidden_activation', nn.Tanh()),  
    ('output_linear', nn.Linear(8, 1))  
]))  
  
seq_model  
  
# Out[19]:  

```

此代码使您可以获得子模块的更多说明性名称：

```
# In[20]:  
for name, param in seq_model.named_parameters():  
    print(name, param.shape)  
  
# Out[20]:  
hidden_linear.weight torch.Size([8, 1])  
hidden_linear.bias torch.Size([8])  
output_linear.weight torch.Size([1, 8])  
output_linear.bias torch.Size([1])
```

您也可以通过访问子模块来访问特定的参数，就像它们是属性一样：

```
# In[21]:  
seq_model.output_linear.bias  
  
# Out[21]:  
Parameter containing:  
tensor([-0.2194], requires_grad=True)
```

与您在本章开始时一样，该代码对于检查参数或其梯度非常有用，例如在训练期间监视梯度。假设您要打印出隐藏层线性部分的权重的梯度。您可以为新的神经网络模型运行训练循环，然后在最后一个 epoch 之后查看梯度结果：

```

# In[22]:
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3) ←
    Note that the learning
    rate has dropped a bit
    to help with stability.

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    model = seq_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print('output', seq_model(t_un_val))
print('answer', t_c_val)
print('hidden', seq_model.hidden_linear.weight.grad)

# Out[22]:
Epoch 1, Training loss 207.2268524169922, Validation loss 106.6062240600586
Epoch 1000, Training loss 6.121204376220703, Validation loss
    2.213937759399414
Epoch 2000, Training loss 5.273784637451172, Validation loss
    0.0025627268478274345
Epoch 3000, Training loss 2.4436306953430176, Validation loss
    1.9463319778442383
Epoch 4000, Training loss 1.6909029483795166, Validation loss
    4.027190685272217
Epoch 5000, Training loss 1.4900192022323608, Validation loss
    5.368413925170898
output tensor([[[-1.8966],
    [11.1774]], grad_fn=<AddmmBackward>])
answer tensor([[-4.],
    [14.]])
hidden tensor([[-0.0073],
    [ 4.0584],
    [-4.5080],
    [-4.4498],
    [ 0.0127],
    [-0.0073],
    [-4.1530],
    [-0.6572]])

```

您还可以在整个数据上评估模型，以查看它与一条线有何不同：

```

# In[23]:
from matplotlib import pyplot as plt

t_range = torch.arange(20., 90.).unsqueeze(1)

fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
plt.plot(t_range.numpy(), seq_model(0.1 * t_range).detach().numpy(), 'c-')
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx')

```

此代码产生图 5.8。

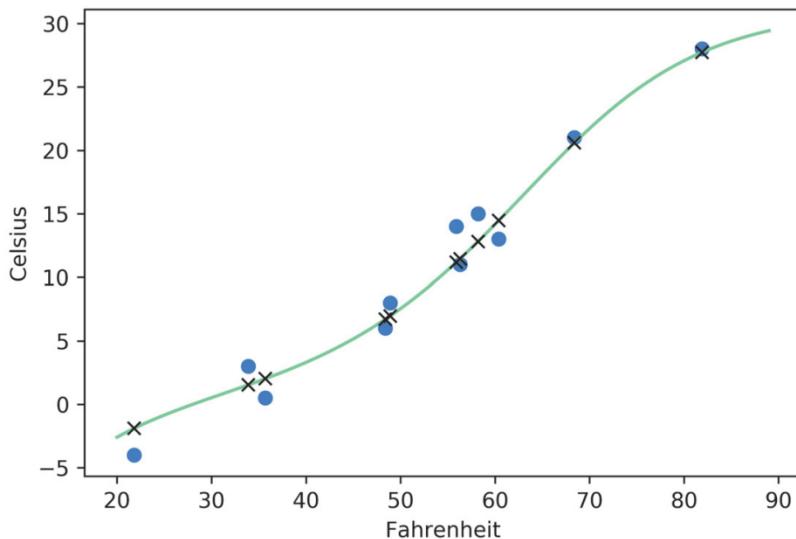


图 5.8 神经网络模型的图，其中包含输入数据（用圆圈表示），期望输出（用叉表示）和显示样品之间行为的实线

您会意识到神经网络有过拟合的趋势，因为它试图追赶包括噪声在内的测量点。不过，它总体上的效果还不错。

### 5.3 继承 nn.Module 类

对于更大、更复杂的项目，您需要将 `nn.Sequential` 舍弃，而应使用赋予您更大灵活性的方法将 `nn.Module` 子类化。要为 `nn.Module` 子类化，至少需要定义 `.forward(...)` 函数，该函数将输入数据输入到模块并返回输出结果。如果您使用标准 `torch` 操作，则 `autograd` 会自动处理反向传递。

**注意** 通常，您的整个模型都是作为 `nn.Module` 的子类实现的，而子类又可以包含也是 `nn.Module` 的子类的子模块。

我们将向您展示三种方法来实现相同的网络结构，使用越来越复杂的 PyTorch 功能来实现这一点，并改变隐藏层中神经元的数量，以使它们更易于区分。

第一种方法是 `nn.Sequential` 的简单实例，如下面的清单所示。

### Listing 5.3 code/p1ch6/3\_nn\_module\_subclassing.ipynb

```
# In[2]:  
seq_model = nn.Sequential(  
    nn.Linear(1, 11),      ←  
    nn.Tanh(),  
    nn.Linear(11, 1))  
  
seq_model  
  
# Out[2]:  
Sequential(  
    (0): Linear(in_features=1, out_features=11, bias=True)  
    (1): Tanh()  
    (2): Linear(in_features=11, out_features=1, bias=True)  
)
```

The choice of 11 is somewhat arbitrary,  
but the sizes of the two layers must match.

尽管此代码有效，但是您没有关于各层使用目的的语义信息。您可以通过给每个层一个标签，使用有序词典而不是列表作为输入来纠正这种情况：

```
# In[3]:  
from collections import OrderedDict  
  
namedseq_model = nn.Sequential(OrderedDict([  
    ('hidden_linear', nn.Linear(1, 12)),  
    ('hidden_activation', nn.Tanh()),  
    ('output_linear', nn.Linear(12, 1))  
]))  
  
namedseq_model  
  
# Out[3]:  
Sequential(  
    (hidden_linear): Linear(in_features=1, out_features=12, bias=True)  
    (hidden_activation): Tanh()  
    (output_linear): Linear(in_features=12, out_features=1, bias=True)  
)
```

好多了。除了 `nn.Sequential` 类（恰当地命名！）提供的纯顺序传递之外，您没有任何能力控制通过网络的数据流。您可以自己子类化 `nn.Module` 来完全控制输入数据的处理：

```
# In[4]:  
class SubclassModel(nn.Module):  
    def __init__(self):  
        super().__init__()
```

```

    self.hidden_linear = nn.Linear(1, 13)
    self.hidden_activation = nn.Tanh()
    self.output_linear = nn.Linear(13, 1)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = self.hidden_activation(hidden_t)
        output_t = self.output_linear(activated_t)

        return output_t

subclass_model = SubclassModel()
subclass_model

# Out[4]:
SubclassModel(
  (hidden_linear): Linear(in_features=1, out_features=13, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=13, out_features=1, bias=True)
)

```

这段代码最终变得更加冗长，因为您必须定义要拥有的层，然后定义在 `forward` 函数中应如何以及以什么顺序处理它们。但是这样的重复工作为您在 `sequential` 模型中提供了难以置信的灵活性，因为您现在可以自由地在 `Forward` 函数中执行各种有趣的事情。尽管以下这个示例不太有意义，但是可以这样做

If `random.random()>0.5`, `Activated_t = self.hidden_activation(hidden_t)`,`else hidden_t` 以此只在一半的情况下使用激活函数！因为 PyTorch 使用基于动态图的自动梯度，所以无论返回什么 `random.random()`，梯度都可以在有时出现的激活中正确流动！

通常，您希望使用模块的构造函数来定义我们在 `forward` 函数中调用的子模块，以便它们可以在模块的整个生命周期中保存其参数。例如，您可以在构造函数中实例化 `nn.Linear` 的两个实例，并在 `forward` 函数中使用它们。有趣的是，将 `nn.Module` 的实例分配给 `nn.Module` 中的属性，就像您在此处的构造函数中所做的那样，会自动将模块注册为子模块，这使模块可以访问其子模块的参数，而无需用户进一步操作。

回到非随机的 `SubclassModel`，您会看到该类的打印输出类似于具有命名参数的 `sequential` 模型的输出。这是有道理的，因为您使用了相同的名称并打算实现相同的体系结构。如果查看所有三个模型的参数，也会在此处看到相似之处（除了隐藏神经元数量的差异之外）：

```

# In[5]:
for type_str, model in [('seq', seq_model), ('namedseq', namedseq_model),
                       ('subclass', subclass_model)]:
    print(type_str)
    for name_str, param in model.named_parameters():
        print("{:21} {:19} {}".format(name_str, str(param.shape), param.numel()))

    print()

```

```
# Out[5]:
seq
0.weight      torch.Size([11, 1]) 11
0.bias        torch.Size([11])    11
2.weight      torch.Size([1, 11]) 11
2.bias        torch.Size([1])    1

namedseq
hidden_linear.weight  torch.Size([12, 1]) 12
hidden_linear.bias    torch.Size([12])    12
output_linear.weight  torch.Size([1, 12]) 12
output_linear.bias    torch.Size([1])    1

subclass
hidden_linear.weight  torch.Size([13, 1]) 13
hidden_linear.bias    torch.Size([13])    13
output_linear.weight  torch.Size([1, 13]) 13
output_linear.bias    torch.Size([1])    1
```

此处发生的是，`named_parameters()` 调用会深入研究构造函数中为属性分配的所有子模块，然后递归调用这些子模块上的 `named_parameters()`。无论子模块有多嵌套，任何 `nn.Module` 都可以访问所有子参数的列表。通过访问将由 `autograd` 保存的 `grad` 属性，优化器知道如何更改参数以最大程度地减少损失。

**注意** Python 列表或字典实例中包含的子模块不会自动注册！子类可以使用 `nn.Module`([https://pytorch.org/docs/stable/nn.html#torch.nn.Module.add\\_module](https://pytorch.org/docs/stable/nn.html#torch.nn.Module.add_module)) 的 `add_module(name, module)` 方法手动注册这些模块，也可以使用提供的 `nn.ModuleList` 和 `nn.ModuleDict` 类（它们为包含的实例提供自动注册）。

回顾 `SubclassModel` 类的实现，并考虑在构造函数中注册子模块的实用程序，以便您可以访问其参数，同时注册没有参数的子模块（如 `nn.Tanh`）似乎有点浪费。直接在 `forward` 函数中调用它们难道不是很容易吗？当然可以。

PyTorch 具有每个 `nn` 模块的函数的对应物。从功能上讲，我们的意思是“没有内部状态”或“其输出值完全由输入参数值确定”。确实，`torch.nn.functional` 提供了许多与 `nn` 中相同的模块，但是所有最终参数都作为参数移到了函数调用中。例如，`nn.Linear` 的功能对应项是 `nn.functional.linear`。其是具有签名 `linear(input, weight,bias=None)` 的函数。`weight` 和 `bias` 参数是传入该函数的参数。

回到模型，为 `nn.Linear` 继续使用 `nn` 模块会很有意义，以便 `SubclassModel` 可以在训练期间管理其所有 `Parameter` 实例。但是，您可以安全地切换到 `Tanh` 的函数对应物，因为它没有参数：

```

# In[6]:
class SubclassFunctionalModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden_linear = nn.Linear(1, 14)
        self.output_linear = nn.Linear(14, 1)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = torch.tanh(hidden_t)      ↪
        output_t = self.output_linear(activated_t)

        return output_t

func_model = SubclassFunctionalModel()
func_model

# Out[6]:
SubclassFunctionalModel(
  (hidden_linear): Linear(in_features=1, out_features=14, bias=True)
  (output_linear): Linear(in_features=14, out_features=1, bias=True)
)

```

功能版本更加简洁，与非功能版本完全等效（随着模型变得越来越复杂，保存的代码行开始累加！）请注意，实例化需要参数的模块仍然有意义。它们在构造函数中的初始化。例如，HardTanh 采用可选的 `min_val` 和 `max_val` 参数，而不是在 `forward` 主体中重复声明这些参数，您可以创建 HardTanh 实例并重用它。

**提示** 尽管 `tanh` 之类的通用科学函数在 1.0 版的 `torch.nn.function` 中仍然存在，但不建议使用这些入口点，这些在将来会被弃用，而应使用顶级 Torch 命名空间中的入口点。更多有用的函数保留在 `torch.nn.functional` 中。

## 结论

尽管我们处理了一个简单的问题，但在本章中我们讨论了很多内容。我们剖析了构建可微模型，并通过使用梯度下降，先使用原始 `autograd` 然后依赖 `nn` 对其进行训练。到现在为止，您应该对幕后发生的事情充满信心的理解。

我们希望 PyTorch 的这种体验能给您带来更多的欲望！

## 更多资源

大量书籍和其他资源可用来帮助教授深度学习。我们推荐以下内容：

- 官方 PyTorch 网站：<https://pytorch.org>
- 安德鲁·W·特拉斯卡 (Andrew W. Traska) 撰写的《 Grokking 深度学习》是建立深刻的思想模型和对深度神经网络基础机制的直观理解的重要资源。
- 有关该领域的详尽介绍和参考，我们将引导您进入 Ian Goodfellow, Yoshua Bengio

和 Aaron Courville 的《深度学习》。

- 最后但并非最不重要的是，本书的完整版本现已在 Early Access 中提供，预计在 2019 年末将会印刷出版：<https://www.manning.com/books/deeplearningwith-pytorch>。

## 练习

- 在简单的神经网络模型中尝试不同隐藏神经元数量以及学习率的实验。
  - 哪些变化导致模型的输出更加线性？
  - 您能否使模型明显过拟合数据？
- 物理学中最难解决的问题是找到一种合适的葡萄酒来庆祝发现。加载第 3 章中的葡萄酒数据，并使用适当数量的输入参数创建一个新模型。
  - 与您使用的温度数据相比，训练需要多长时间？
  - 您能解释一下哪些因素会影响训练时间吗？
  - 在训练该数据集时，您能减少损失吗？
  - 如何绘制该数据集的图形？

## 总结

- 神经网络可以自动调整以专门解决眼前的问题。
- 神经网络可以轻松访问模型中损失相对于任何参数的导数，从而使参数的演化变得高效。得益于其自动微分引擎，PyTorch 可以轻松提供此类导数。
- 围绕线性变换的激活函数使神经网络能够逼近高度非线性的函数，同时使它们变得足够简单以进行优化。
- nn 模块与张量标准库一起提供了用于创建神经网络的所有构建块。
- 要识别出过拟合，必须将训练数据点集与验证集分开。没有解决过拟合的方法，除了获取更多数据（或增加数据中的可变性），并且采用更简单的模型是一个好的开始。
- 任何从事数据科学的人都应该时刻绘制数据。