Chapter 1

从张量开始

1.1 实际数据转为浮点数

1.2 张量：多维数组

1.2.1 张量的本质

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 Figure 1.1 左侧所示。另外，PyTorch 张量或 NumPy 数组通常是连续内存块的视图，这些内存块包含未装箱的 C 数字类型，而不是 Python 对象。在本例中，每个元素都是 32 位（4 字节）的浮点数，如 Figure 1.1 右侧所示。这意味着存储 1,000,000 个浮点数的一维张量将恰好需要 4,000,000 个连续字节，再加上元数据的小开销，如维度和数字类型。

1.3 命名张量

张量的维度或坐标轴通常用来表示诸如像素位置或颜色通道的信息，这意味着当我们要把一个张量作为索引时，我们需要记住维度的顺序并按此顺序编写索引。在通过多个张量转换数据时，跟

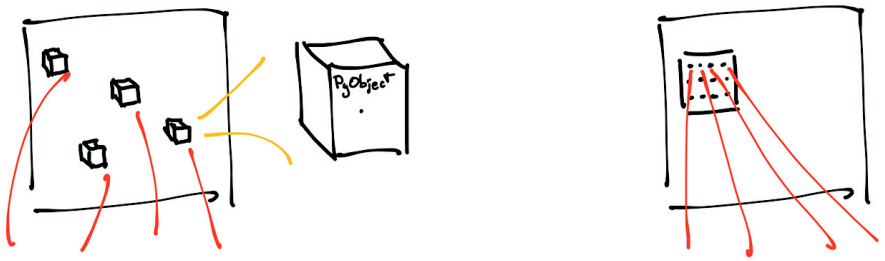

Figure 1.1: Python 列表（装箱）数值与张量或数组（非装箱）数值的对比

踪每个维度包含哪些数据可能容易出错。

PyTorch 1.3 将命名张量作为试验性的特性。张量工厂函数（诸如 tensor()和 rand()函数）有一个 names 参数，该参数是一个字符串序列。当我们已经有一个张量并且想要为其添加名称但不改变现有的名称时，我们可以对其调用 `refine_names()` 方法。与索引类似，省略号（...）允许你省略任意数量的维度。使用 `rename()` 兄弟方法，还可以覆盖或删除（通过传入 None）现有名称。

对于有 2 个输入的操作，除了常规维度检查，即检查张量维度是否相同，以及是否一个张量维度为 1 并且可以广播给另一个张量，PyTorch 还将检查张量名称。到目前为止，它还没有提供自动维度对齐功能，因此我们需要显式地进行此操作。`align_as()` 方法返回一个张量，其中添加了缺失的维度。

如果我们想在对命名的张量进行操作的函数之外使用张量，需要通过将这些张量重命名为 None 来删除它们的名称。

1.4 张量的元素类型

使用标准 Python 数字类型可能不是最优的，原因如下：

- Python 中的数字是对象。例如，一个浮点数在计算机上可能只需要 32 位来表示，而 Python会通过引用计数将它转换成一个完整的 Python 对象，等等。如果我们需要存储少量数值，采用装箱操作并不是问题，但是如果我们需要存储数百万的数据，采用装箱操作会非常低效。
- Python 中的列表属于对象的顺序集合，没有为有效地获取两个向量的点积或将向量求和而定义的操作。另外，Python 列表无法优化其内容在内存中的排列，因为它们是指向Python 对象的指针的可索引集合，这些对象可能是任何数据类型而不仅仅是数字。最后，Python 列表是一维的，尽管我们可创建元素为列表的列表，但这同样是非常低效的。
- Python 解释器与优化后的已编译的代码相比速度很慢。在大型数字类型的数据集合上执行数学运算，使用用编译过的更低级语言（如 C 语言）编写的优化代码可以快得多。

1.4.1 适合任何场合的 dtype

在神经网络中发生的计算通常是用 32 位浮点精度执行的。采用更高的精度，如 64 位，并不会提高模型精度，反而需要更多的内存和计算时间。16 位半精度浮点数的数据类型在标准 CPU 中并不存在，而是由现代 GPU 提供的。如果需要的话，可以切换到半精度来减少神经网络占用的空间，这样做对精度的影响也很小。

张量可以作为其他张量的索引，在这种情况下，PyTorch 期望索引张量为 64 位的整数。创建一个将整数作为参数的张量，例如使用 torch.tensor([2,2])，默认会创建一个 64 位的整数张量。因此，我们将把大部分时间用于处理 32 位浮点数和 64 位有符号整数。

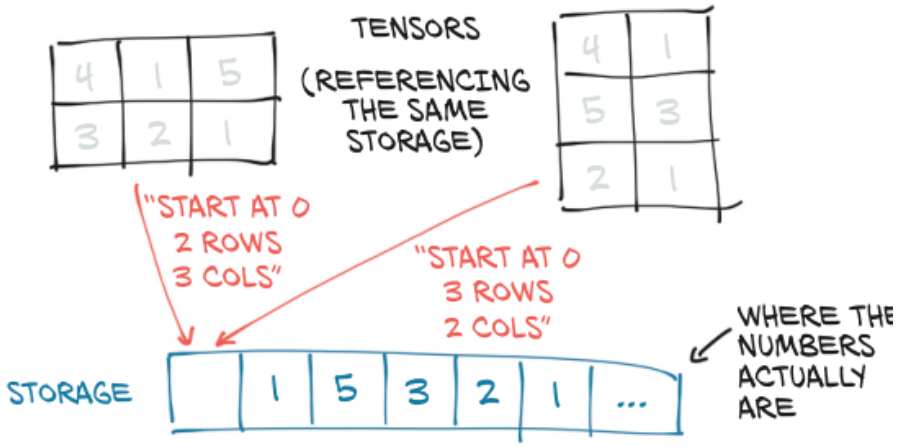

Figure 1.2: 张量是 Storage 实例的视图

1.5 张量的存储视图

张量中的值被分配到由 torch.Storage 实例所管理的连续内存块中。存储区是由数字数据组成的一维数组，即包含给定类型的数字的连续内存块，例如 float（代表 32 位浮点数）或 int64（代表 64 位整数）。一个 PyTorch 的 Tensor 实例就是这样一个 Storage 实例的视图，该实例能够使用偏移量和每个维度的步长对该存储区进行索引。多个张量可以索引同一存储区，即使它们索引到的数据不同。

我们不能用 2 个索引来索引二维张量的存储区。不管和存储区关联的任何其他张量的维度是多少，它的布局始终是一维的。从这点来说，改变一个存储区的值导致与其关联的张量的内容发生变化就不足为怪了

1.6 张量元数据：大小、偏移量和步长

为了在存储区中建立索引，张量依赖于一些明确定义它们的信息：大小、偏移量和步长。Figure 1.3 显示了它们如何相互作用。大小（在 NumPy 中称之为形状）是一个元组，表示张量在每个维度上有多少个元素。偏移量是指存储区中某元素相对张量中的第 1 个元素的索引。步长是指存储区中为了获得下一个元素需要跳过的元素数量。

1.6.1 另一个张量的存储视图

步长是一个元组，指示当索引在每个维度中增加 1 时在存储区中必须跳过的元素数量。

访问一个二维张量中的位置 (i,j) 的元素会导致访问存储中的第 `storage_offset+stride[0]*i+stride[1]*j` 个元素。偏移量通常为 0，如果这个张量是为容纳更大的张量而创建的存储视图，那么偏移量可以为正值。

这种张量和存储区之间的间接关系使得一些操作开销并不大，如转置一个张量或者提取一个子

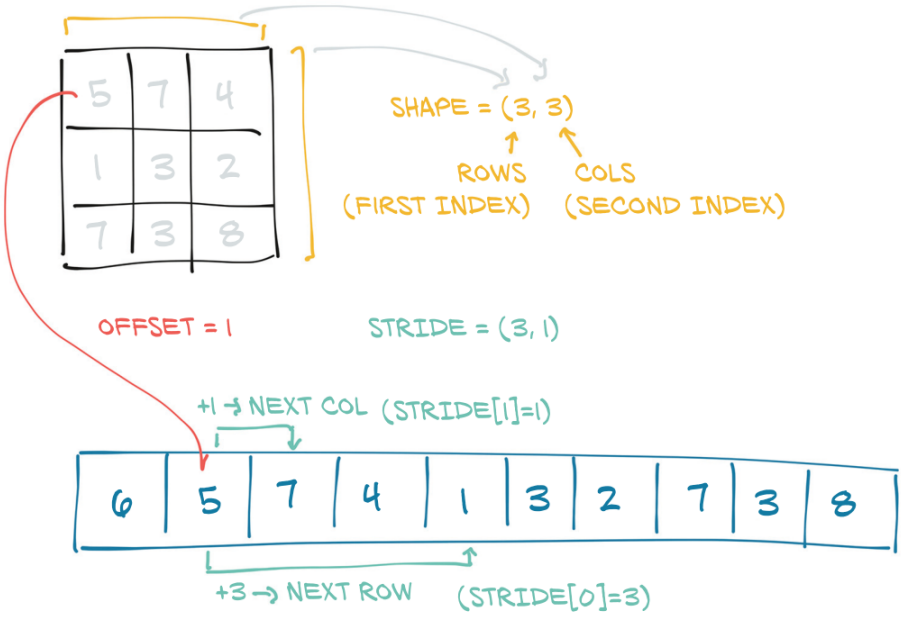

Figure 1.3: 张量的偏移量、大小和步长之间的关系。这里的张量实例是一个具有更大存储区的视图，就像创建一个更大的张量时可能被分配的存储区一样

张量，因为它们不会导致内存重新分配，而是创建一个新的张量对象，该张量具有不同的大小、偏移量和步长。

1.6.2 无复制转置

转置不会分配新的内存，只是创建一个新 Tensor 实例，该实例具有与原始张量不同的步长顺序。

将张量 points 的第 1 个索引增加 1，例如，从 points[0,0]到 points[1,0]，将在存储区中跳过 2 个元素，而将第 2 个索引增加 1。例如，从 points[0,0]到点 points[0,1]，将在存储区中跳过 1 个元素。换句话说，存储区按顺序逐行保存张量中的元素。

我们将张量 points 转置为 `points_t`，如 Figure 1.4 所示。我们在步长中改变元素顺序后，增加的行（张量的第 1 个索引）将沿着存储区跳跃 1 个单位，就像我们沿着 points 的列移动一样。这就是转置的定义。转置不会分配新的内存，只是创建一个新 Tensor 实例，该实例具有与原始张量不同的步长顺序。

1.6.3 高维转置

一个张量的值在存储区中从最右的维度开始向前排列被定义为连续张量。连续张量很方便，因为我们可以有效地按顺序访问它们，而不必在存储中到处跳转。虽然由于现代 CPU 上的 RAM 访问方式，通过改进数据局部性可以提高性能，但这个优势当然取决于算法访问的方式。

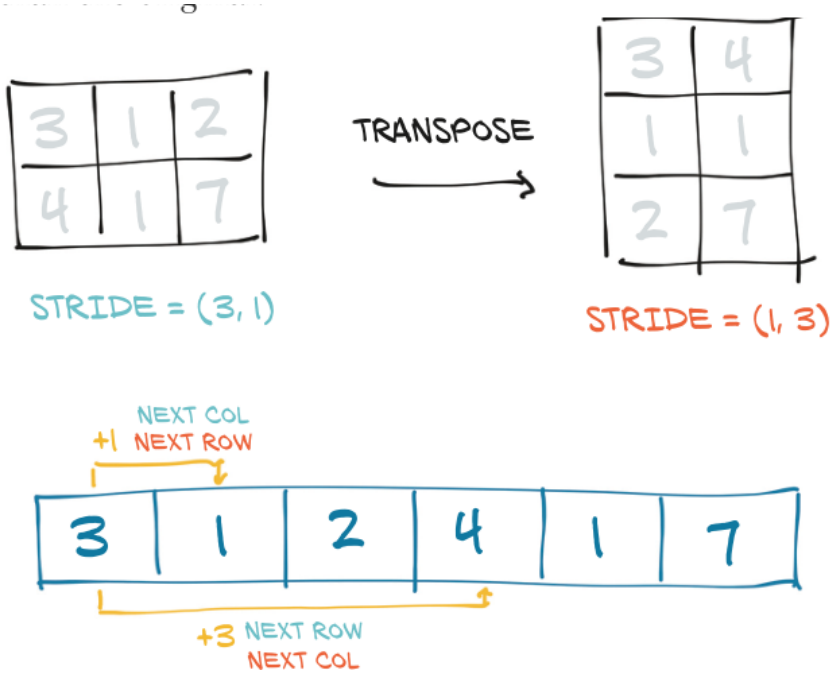

Figure 1.4: 对一个张量进行转置操作

1.6.4 连续张量

在 PyTorch 中一些张量操作只对连续张量起作用，如我们在第 4 章中要遇到的 view() 方法。在这种情况下，PyTorch 将抛出一个提供有用信息的异常，并要求我们显式地调用 contiguous() 方法。值得注意的是，如果张量已经是连续的，那么调用 contiguous() 方法不会产生任何操作，也不会影响性能。

1.7 将张量存储到 GPU

1.7.1 管理张量的设备属性

除了 dtype，PyTorch 张量还有设备（device）的概念，即张量数据在计算机上的位置。

1.8 NumPy 互操作性

为了从张量 points 得到一个 NumPy 数组，我们只需要调用 `numpy()`方法。它将返回一个大小、形状和数字类型都与代码对应的 NumPy 多维数组。有趣的是，返回的数组与张量存储共享相同的底层缓冲区。这意味着，只要数据位于 CPU 上的 RAM 中，就可以有效地执行 numpy()方法，而且基本上不需要任何开销，**还意味着修改 NumPy 数组将导致原始张量的变化**。如果张量是在 GPU 上存储的，PyTorch 将把张量的内容复制到 CPU 上分配的 NumPy 数组中。我们可以用 `from_numpy` 方法从一个 NumPy 数组中获得一个 PyTorch 张量。