3 It starts with a tensor

This chapter covers

- Understanding tensors, the basic data structure in PyTorch
- Indexing and operating on tensors
- Interoperating with NumPy multidimensional arrays
- Moving computations to the GPU for speed

In the previous chapter, we took a tour of some of the many applications that deep learning enables. They invariably consisted of taking data in some form, like images or text, and producing data in another form, like labels, numbers, or more images or text. Viewed from this angle, deep learning really consists of building a system that can transform data from one representation to another. This transformation is driven by extracting commonalities from a series of examples that demonstrate the desired mapping. For example, the system might note the general shape of a dog and the typical colors of a golden retriever. By combining the two image properties, the system can correctly map images with a given shape and color to the golden retriever label, instead of a black lab (or a tawny tomcat, for that matter). The resulting system can consume broad swaths of similar inputs and produce meaningful output for those inputs.

The process begins by converting our input into floating-point numbers. We will cover converting image pixels to numbers, as we see in the first step of figure 3.1, in chapter 4 (along with many other types of data). But before we can get to that, in this chapter, we learn how to deal with all the floating-point numbers in PyTorch by using tensors.

3.1 The world as floating-point numbers

Since floating-point numbers are the way a network deals with information, we need a way to encode real-world data of the kind we want to process into something digestible by a network and then decode the output back to something we can understand and use for our purpose.

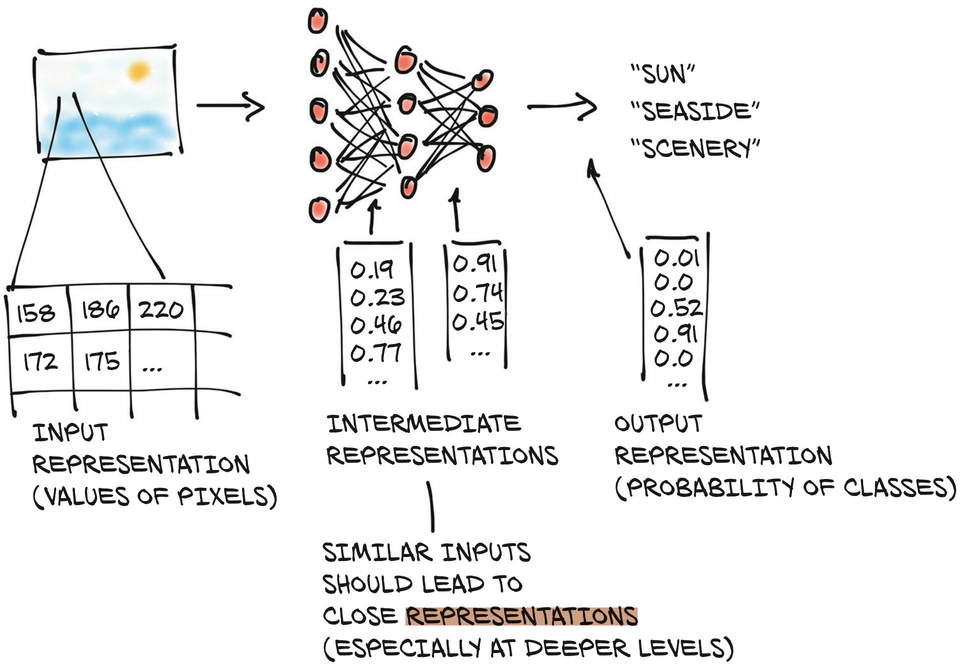

"SUN"
"SEASIDE"
"SCENERY"
158 186 220
172 175 ...
0.19 0.23 0.46 0.77 ...
0.91 0.74 0.45 ...
0.01 0.0 0.52 0.91 0.0 ...
INPUT REPRESENTATION (VALUES OF PIXELS)
INTERMEDIATE REPRESENTATIONS
OUTPUT REPRESENTATION (PROBABILITY OF CLASSES)
SIMILAR INPUTS SHOULD LEAD TO CLOSE REPRESENTATIONS (ESPECIALLY AT DEEPER LEVELS)

Figure 3.1 A deep neural network learns how to transform an input representation to an output representation. (Note: The numbers of neurons and outputs are not to scale.)

A deep neural network typically learns the transformation from one form of data to another in stages, which means the partially transformed data between each stage can be thought of as a sequence of intermediate representations. For image recognition, early representations can be things such as edge detection or certain textures like fur. Deeper representations can capture more complex structures like ears, noses, or eyes.

In general, such intermediate representations are collections of floating-point numbers that characterize the input and capture the data's structure in a way that is instrumental for describing how inputs are mapped to the outputs of the neural network. Such characterization is specific to the task at hand and is learned from relevant examples. These collections of floating-point numbers and their manipulation are at the heart of modern AI—we will see several examples of this throughout the book.

It's important to keep in mind that these intermediate representations (like those shown in the second step of figure 3.1) are the results of combining the input with the weights of the previous layer of neurons. Each intermediate representation is unique to the inputs that preceeded it.

Before we can begin the process of converting our data to floating-point input, we must first have a solid understanding of how PyTorch handles and stores data—as input, as intermediate representations, and as output. This chapter will be devoted to precisely that.

To this end, PyTorch introduces a fundamental data structure: the *tensor*. We already bumped into tensors in chapter 2, when we ran inference on pretrained networks. For those who come from mathematics, physics, or engineering, the term *tensor* comes bundled with the notion of spaces, reference systems, and transformations between them. For better or worse, those notions do not apply here. In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions, as we can see in figure 3.2. Another name for the same concept is *multidimensional array*. The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor.

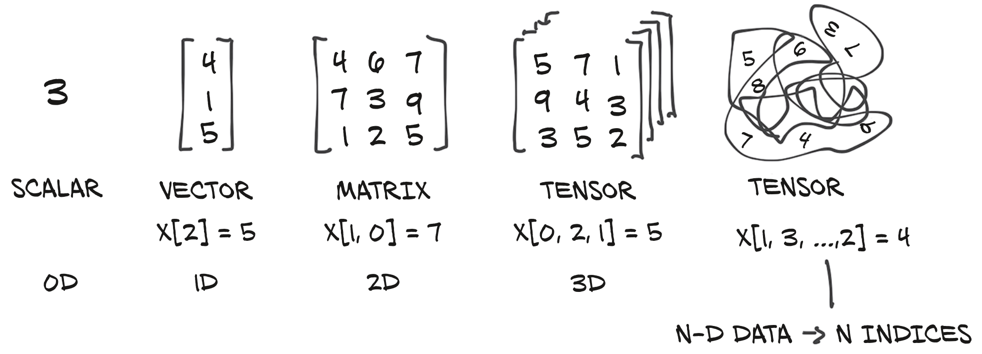


Figure 3.2 **Tensors are the building blocks for representing data in PyTorch.**

PyTorch is not the only library that deals with multidimensional arrays. NumPy is by far the most popular multidimensional array library, to the point that it has now arguably become the *lingua franca* of data science. PyTorch features seamless interoperability with NumPy, which brings with it first-class integration with the rest of the scientific libraries in Python, such as SciPy (www.scipy.org), Scikit-learn (https://scikit-learn.org), and Pandas (https://pandas.pydata.org).

Compared to NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform very fast operations on graphical processing units (GPUs), distribute operations on multiple devices or machines, and keep track of the graph of computations that created them. These are all important features when implementing a modern deep learning library.

We'll start this chapter by introducing PyTorch tensors, covering the basics in order to set things in motion for our work in the rest of the book. First and foremost, we'll learn how to manipulate tensors using the PyTorch tensor library. This includes things like how the data is stored in memory, how certain operations can be performed on arbitrarily large tensors in constant time, and the aforementioned NumPy interoperability and GPU acceleration. Understanding the capabilities and API of tensors is important if they're to become go-to tools in our programming toolbox. In the next chapter, we'll put this knowledge to good use and learn how to represent several different kinds of data in a way that enables learning with neural networks.

3.2 Tensors: Multidimensional arrays

We have already learned that tensors are the fundamental data structure in PyTorch. A tensor is an array: that is, a data structure that stores a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices.

3.2.1 From Python lists to PyTorch tensors

Let's see `list` indexing in action so we can compare it to tensor indexing. Take a list of three numbers in Python (.code/p1ch3/1_tensors.ipynb):

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

We can access the first element of the list using the corresponding zero-based index:

```
# In[2]:
a[0]

# Out[2]:
1.0

# In[3]:
a[2] = 3.0
a

# Out[3]:
[1.0, 2.0, 3.0]
```

It is not unusual for simple Python programs dealing with vectors of numbers, such as the coordinates of a 2D line, to use Python lists to store the vectors. As we will see in the following chapter, using the more efficient tensor data structure, many types of data—from images to time series, and even sentences—can be represented. By defining operations over tensors, some of which we'll explore in this chapter, we can slice and manipulate data expressively and efficiently at the same time, even from a high-level (and not particularly fast) language such as Python.

3.2.2 Constructing our first tensors

Let's construct our first PyTorch tensor and see what it looks like. It won't be a particularly meaningful tensor for now, just three ones in a column:

```
# In[4]:
import torch          <-- Imports the torch module
a = torch.ones(3)     <-- Creates a one-dimensional
a                         tensor of size 3 filled with 1s

# 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.])
```

After importing the `torch` module, we call a function that creates a (one-dimensional) tensor of size 3 filled with the value `1.0`. We can access an element using its zero-based index or assign a new value to it. Although on the surface this example doesn't differ much from a list of number objects, under the hood things are completely different.

3.2.3 The essence of tensors

Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory, as shown on the left in figure 3.3. PyTorch tensors or NumPy arrays, on the other hand, are views over (typically) contiguous memory blocks containing *unboxed* C numeric types rather than Python objects. Each element is a 32-bit (4-byte) `float` in this case, as we can see on the right side of figure 3.3. This means storing a 1D tensor of 1,000,000 float numbers will require exactly 4,000,000 contiguous bytes, plus a small overhead for the metadata (such as dimensions and numeric type).

Say we have a list of coordinates we'd like to use to represent a geometrical object: perhaps a 2D triangle with vertices at coordinates (4, 1), (5, 3), and (2, 1). The example is not particularly pertinent to deep learning, but it's easy to follow. Instead of having coordinates as numbers in a Python list, as we did earlier, we can use a

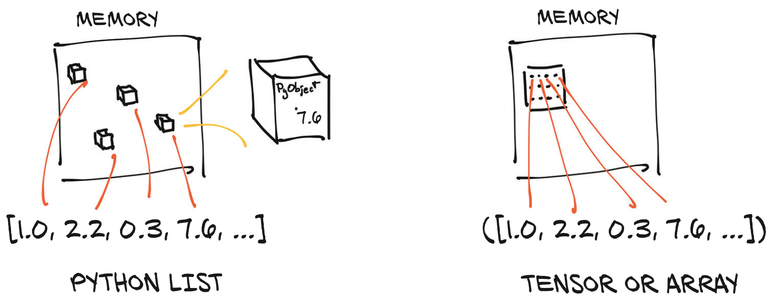


Figure 3.3 Python object (boxed) numeric values versus tensor (unboxed array) numeric values

one-dimensional tensor by storing *X*s in the even indices and *Y*s in the odd indices, like this:

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

Using .zeros is just a way to get an appropriately sized array.

We overwrite those zeros with the values we actually want.

We can also pass a Python list to the constructor, to the same effect:

```
# In[9]:
points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
points

# Out[9]:
tensor([4., 1., 5., 3., 2., 1.])
```

To get the coordinates of the first point, we do the following:

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

# Out[10]:
(4.0, 1.0)
```

This is OK, although it would be practical to have the first index refer to individual 2D points rather than point coordinates. For this, we can use a 2D tensor:

```
# In[11]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points
```

```
# Out[11]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

Here, we pass a list of lists to the constructor. We can ask the tensor about its shape:

```
# In[12]:
points.shape

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

This informs us about the size of the tensor along each dimension. We could also use `zeros` or `ones` to initialize the tensor, providing the size as a tuple:

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

# Out[13]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

Now we can access an individual element in the tensor using two indices:

```
# In[14]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[14]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])

# In[15]:
points[0, 1]

# Out[15]:
tensor(1.)
```

This returns the *Y*-coordinate of the zeroth point in our dataset. We can also access the first element in the tensor as we did before to get the 2D coordinates of the first point:

```
# In[16]:
points[0]

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

The output is another tensor that presents a different *view* of the same underlying data. The new tensor is a 1D tensor of size 2, referencing the values of the first row in the `points` tensor. Does this mean a new chunk of memory was allocated, values were copied into it, and the new memory was returned wrapped in a new tensor object? No, because that would be very inefficient, especially if we had millions of points. We'll revisit how tensors are stored later in this chapter when we cover views of tensors in section 3.7.

3.3 Indexing tensors

What if we need to obtain a tensor containing all points but the first? That's easy using range indexing notation, which also applies to standard Python lists. Here's a reminder:

```
# In[53]:
some_list = list(range(6))
some_list[:]
some_list[1:4]
some_list[1:]
some_list[:4]
some_list[:-1]
some_list[1:4:2]
```

All elements in the list

From element 1 inclusive to element 4 exclusive

From element 1 inclusive to the end of the list

From the start of the list to element 4 exclusive

From the start of the list to one before the last element

From element 1 inclusive to element 4 exclusive, in steps of 2

To achieve our goal, we can use the same notation for PyTorch tensors, with the added benefit that, just as in NumPy and other Python scientific libraries, we can use range indexing for each of the tensor's dimensions:

```
# In[54]:
points[1:]
points[1:, :]
points[1:, 0]
points[None]
```

All rows after the first; implicitly all columns

All rows after the first; all columns

All rows after the first; first column

Adds a dimension of size 1, just like unsqueeze

In addition to using ranges, PyTorch features a powerful form of indexing, called *advanced indexing*, which we will look at in the next chapter.

3.4 Named tensors

The dimensions (or axes) of our tensors usually index something like pixel locations or color channels. This means when we want to index into a tensor, we need to remember the ordering of the dimensions and write our indexing accordingly. As data is transformed through multiple tensors, keeping track of which dimension contains what data can be error-prone.

To make things concrete, imagine that we have a 3D tensor like `img_t` from section 2.1.4 (we will use dummy data for simplicity here), and we want to convert it to grayscale. We looked up typical weights for the colors to derive a single brightness value:[1]

```
# In[2]:
img_t = torch.randn(3, 5, 5) # shape [channels, rows, columns]
weights = torch.tensor([0.2126, 0.7152, 0.0722])
```

We also often want our code to generalize—for example, from grayscale images represented as 2D tensors with height and width dimensions to color images adding a third channel dimension (as in RGB), or from a single image to a batch of images. In section 2.1.4, we introduced an additional batch dimension in `batch_t`; here we pretend to have a batch of 2:

```
# In[3]:
batch_t = torch.randn(2, 3, 5, 5) # shape [batch, channels, rows, columns]
```

So sometimes the RGB channels are in dimension 0, and sometimes they are in dimension 1. But we can generalize by counting from the end: they are always in dimension –3, the third from the end. The lazy, unweighted mean can thus be written as follows:

```
# In[4]:
img_gray_naive = img_t.mean(-3)
batch_gray_naive = batch_t.mean(-3)
img_gray_naive.shape, batch_gray_naive.shape

# Out[4]:
(torch.Size([5, 5]), torch.Size([2, 5, 5]))
```

But now we have the weight, too. PyTorch will allow us to multiply things that are the same shape, as well as shapes where one operand is of size 1 in a given dimension. It also appends leading dimensions of size 1 automatically. This is a feature called *broadcasting*. `batch_t` of shape (2, 3, 5, 5) is multiplied by `unsqueezed_weights` of shape (3, 1, 1), resulting in a tensor of shape (2, 3, 5, 5), from which we can then sum the third dimension from the end (the three channels):

```
# In[5]:
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)
img_weights = (img_t * unsqueezed_weights)
batch_weights = (batch_t * unsqueezed_weights)
img_gray_weighted = img_weights.sum(-3)
batch_gray_weighted = batch_weights.sum(-3)
batch_weights.shape, batch_t.shape, unsqueezed_weights.shape

# Out[5]:
(torch.Size([2, 3, 5, 5]), torch.Size([2, 3, 5, 5]), torch.Size([3, 1, 1]))
```

[1] As perception is not trivial to norm, people have come up with many weights. For example, see https://en.wikipedia.org/wiki/Luma_(video).

Because this gets messy quickly—and for the sake of efficiency—the PyTorch function `einsum` (adapted from NumPy) specifies an indexing mini-language[2] giving index names to dimensions for sums of such products. As often in Python, broadcasting—a form of summarizing unnamed things—is done using three dots `'…'`; but don't worry too much about `einsum`, because we will not use it in the following:

```
# In[6]:
img_gray_weighted_fancy = torch.einsum('...chw,c->...hw', img_t, weights)
batch_gray_weighted_fancy = torch.einsum('...chw,c->...hw', batch_t, weights)
batch_gray_weighted_fancy.shape

# Out[6]:
torch.Size([2, 5, 5])
```

As we can see, there is quite a lot of bookkeeping involved. This is error-prone, especially when the locations where tensors are created and used are far apart in our code. This has caught the eye of practitioners, and so it has been suggested[3] that the dimension be given a name instead.

PyTorch 1.3 added *named tensors* as an experimental feature (see https://pytorch.org/tutorials/intermediate/named_tensor_tutorial.html and https://pytorch.org/docs/stable/named_tensor.html). Tensor factory functions such as `tensor` and `rand` take a `names` argument. The names should be a sequence of strings:

```
# In[7]:
weights_named = torch.tensor([0.2126, 0.7152, 0.0722], names=['channels'])
weights_named

# Out[7]:
tensor([0.2126, 0.7152, 0.0722], names=('channels',))
```

When we already have a tensor and want to add names (but not change existing ones), we can call the method `refine_names` on it. Similar to indexing, the ellipsis (…) allows you to leave out any number of dimensions. With the `rename` sibling method, you can also overwrite or drop (by passing in `None`) existing names:

```
# In[8]:
img_named =  img_t.refine_names(..., 'channels', 'rows', 'columns')
batch_named = batch_t.refine_names(..., 'channels', 'rows', 'columns')
print("img named:", img_named.shape, img_named.names)
print("batch named:", batch_named.shape, batch_named.names)

# Out[8]:
img named: torch.Size([3, 5, 5]) ('channels', 'rows', 'columns')
batch named: torch.Size([2, 3, 5, 5]) (None, 'channels', 'rows', 'columns')
```

[2] Tim Rocktäschel's blog post "Einsum is All You Need—Einstein Summation in Deep Learning" (https://rockt.github.io/2018/04/30/einsum) gives a good overview.

[3] See Sasha Rush, "Tensor Considered Harmful," Harvardnlp, http://nlp.seas.harvard.edu/NamedTensor.

For operations with two inputs, in addition to the usual dimension checks—whether sizes are the same, or if one is 1 and can be broadcast to the other—PyTorch will now check the names for us. So far, it does not automatically align dimensions, so we need to do this explicitly. The method `align_as` returns a tensor with missing dimensions added and existing ones permuted to the right order:

```
# In[9]:
weights_aligned = weights_named.align_as(img_named)
weights_aligned.shape, weights_aligned.names

# Out[9]:
(torch.Size([3, 1, 1]), ('channels', 'rows', 'columns'))
```

Functions accepting dimension arguments, like `sum`, also take named dimensions:

```
# In[10]:
gray_named = (img_named * weights_aligned).sum('channels')
gray_named.shape, gray_named.names

# Out[10]:
(torch.Size([5, 5]), ('rows', 'columns'))
```

If we try to combine dimensions with different names, we get an error:

```
gray_named = (img_named[..., :3] * weights_named).sum('channels')

RuntimeError: Error when
 attempting to broadcast dims ['channels', 'rows',
  'columns'] and dims ['channels']: dim 'columns' and dim 'channels'
  are at the same position from the right but do not match.
```

If we want to use tensors outside functions that operate on named tensors, we need to drop the names by renaming them to `None`. The following gets us back into the world of unnamed dimensions:

```
# In[12]:
gray_plain = gray_named.rename(None)
gray_plain.shape, gray_plain.names

# Out[12]:
(torch.Size([5, 5]), (None, None))
```

Given the experimental nature of this feature at the time of writing, and to avoid mucking around with indexing and alignment, we will stick to unnamed in the remainder of the book. Named tensors have the potential to eliminate many sources of alignment errors, which—if the PyTorch forum is any indication—can be a source of headaches. It will be interesting to see how widely they will be adopted.

3.5 Tensor element types

So far, we have covered the basics of how tensors work, but we have not yet touched on what kinds of numeric types we can store in a `Tensor`. As we hinted at in section 3.2, using the standard Python numeric types can be suboptimal for several reasons:

- *Numbers in Python are objects.* Whereas a floating-point number might require only, for instance, 32 bits to be represented on a computer, Python will convert it into a full-fledged Python object with reference counting, and so on. This operation, called *boxing*, is not a problem if we need to store a small number of numbers, but allocating millions gets very inefficient.
- *Lists in Python are meant for sequential collections of objects.* There are no operations defined for, say, efficiently taking the dot product of two vectors, or summing vectors together. Also, Python lists have no way of optimizing the layout of their contents in memory, as they are indexable collections of pointers to Python objects (of any kind, not just numbers). Finally, Python lists are one-dimensional, and although we can create lists of lists, this is again very inefficient.
- *The Python interpreter is slow compared to optimized, compiled code.* Performing mathematical operations on large collections of numerical data can be much faster using optimized code written in a compiled, low-level language like C.

For these reasons, data science libraries rely on NumPy or introduce dedicated data structures like PyTorch tensors, which provide efficient low-level implementations of numerical data structures and related operations on them, wrapped in a convenient high-level API. To enable this, the objects within a tensor must all be numbers of the same type, and PyTorch must keep track of this numeric type.

3.5.1 Specifying the numeric type with dtype

The `dtype` argument to tensor constructors (that is, functions like `tensor`, `zeros`, and `ones`) specifies the numerical data (d) type that will be contained in the tensor. The data type specifies the possible values the tensor can hold (integers versus floating-point numbers) and the number of bytes per value.[4] The `dtype` argument is deliberately similar to the standard NumPy argument of the same name. Here's a list of the possible values for the `dtype` argument:

- `torch.float32` or `torch.float`: 32-bit floating-point
- `torch.float64` or `torch.double`: 64-bit, double-precision floating-point
- `torch.float16` or `torch.half`: 16-bit, half-precision floating-point
- `torch.int8`: signed 8-bit integers
- `torch.uint8`: unsigned 8-bit integers
- `torch.int16` or `torch.short`: signed 16-bit integers
- `torch.int32` or `torch.int`: signed 32-bit integers
- `torch.int64` or `torch.long`: signed 64-bit integers
- `torch.bool`: Boolean

[4] And signed-ness, in the case of `uint8`.

The default data type for tensors is 32-bit floating-point.

3.5.2 A dtype for every occasion

As we will see in future chapters, computations happening in neural networks are typically executed with 32-bit floating-point precision. Higher precision, like 64-bit, will not buy improvements in the accuracy of a model and will require more memory and computing time. The 16-bit floating-point, half-precision data type is not present natively in standard CPUs, but it is offered on modern GPUs. It is possible to switch to half-precision to decrease the footprint of a neural network model if needed, with a minor impact on accuracy.

Tensors can be used as indexes in other tensors. In this case, PyTorch expects indexing tensors to have a 64-bit integer data type. Creating a tensor with integers as arguments, such as using `torch.tensor([2, 2])`, will create a 64-bit integer tensor by default. As such, we'll spend most of our time dealing with `float32` and `int64`.

Finally, predicates on tensors, such as `points > 1.0`, produce `bool` tensors indicating whether each individual element satisfies the condition. These are the numeric types in a nutshell.

3.5.3 Managing a tensor's dtype attribute

In order to allocate a tensor of the right numeric type, we can specify the proper `dtype` as an argument to the constructor. For example:

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

We can find out about the `dtype` for a tensor by accessing the corresponding attribute:

```
# In[48]:
short_points.dtype

# Out[48]:
torch.int16
```

We can also cast the output of a tensor creation function to the right type using the corresponding casting method, such as

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

or the more convenient `to` method:

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

Under the hood, `to` checks whether the conversion is necessary and, if so, does it. The `dtype`-named casting methods like `float` are shorthands for `to`, but the `to` method can take additional arguments that we'll discuss in section 3.9.

When mixing input types in operations, the inputs are converted to the larger type automatically. Thus, if we want 32-bit computation, we need to make sure all our inputs are (at most) 32-bit:

```
# In[51]:
points_64 = torch.rand(5, dtype=torch.double)
points_short = points_64.to(torch.short)
points_64 * points_short  # works from PyTorch 1.3 onwards

# Out[51]:
tensor([0., 0., 0., 0., 0.], dtype=torch.float64)
```

rand initializes the tensor elements to random numbers between 0 and 1.

3.6 The tensor API

At this point, we know what PyTorch tensors are and how they work under the hood. Before we wrap up, it is worth taking a look at the tensor operations that PyTorch offers. It would be of little use to list them all here. Instead, we're going to get a general feel for the API and establish a few directions on where to find things in the online documentation at http://pytorch.org/docs.

First, the vast majority of operations on and between tensors are available in the `torch` module and can also be called as methods of a tensor object. For instance, the `transpose` function we encountered earlier can be used from the `torch` module

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

a.shape, a_t.shape

# Out[71]:
(torch.Size([3, 2]), torch.Size([2, 3]))
```

or as a method of the `a` tensor:

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

a.shape, a_t.shape

# Out[72]:
(torch.Size([3, 2]), torch.Size([2, 3]))
```

There is no difference between the two forms; they can be used interchangeably.

We mentioned the online docs earlier (http://pytorch.org/docs). They are exhaustive and well organized, with the tensor operations divided into groups:

- *Creation ops*—Functions for constructing a tensor, like `ones` and `from_numpy`
- *Indexing, slicing, joining, mutating ops*—Functions for changing the shape, stride, or content of a tensor, like `transpose`
- *Math ops*—Functions for manipulating the content of the tensor through computations
  – *Pointwise ops*—Functions for obtaining a new tensor by applying a function to each element independently, like `abs` and `cos`
  – *Reduction ops*—Functions for computing aggregate values by iterating through tensors, like `mean`, `std`, and `norm`
  – *Comparison ops*—Functions for evaluating numerical predicates over tensors, like `equal` and `max`
  – *Spectral ops*—Functions for transforming in and operating in the frequency domain, like `stft` and `hamming_window`
  – *Other operations*—Special functions operating on vectors, like `cross`, or matrices, like `trace`
  – *BLAS and LAPACK operations*—Functions following the Basic Linear Algebra Subprograms (BLAS) specification for scalar, vector-vector, matrix-vector, and matrix-matrix operations
- *Random sampling*—Functions for generating values by drawing randomly from probability distributions, like `randn` and `normal`
- *Serialization*—Functions for saving and loading tensors, like `load` and `save`
- *Parallelism*—Functions for controlling the number of threads for parallel CPU execution, like `set_num_threads`

Take some time to play with the general tensor API. This chapter has provided all the prerequisites to enable this kind of interactive exploration. We will also encounter several of the tensor operations as we proceed with the book, starting in the next chapter.

3.7 Tensors: Scenic views of storage

It is time for us to look a bit closer at the implementation under the hood. Values in tensors are allocated in contiguous chunks of memory managed by `torch.Storage` instances. A storage is a one-dimensional array of numerical data: that is, a contiguous block of memory containing numbers of a given type, such as `float` (32 bits representing a floating-point number) or `int64` (64 bits representing an integer). A PyTorch `Tensor` instance is a view of such a `Storage` instance that is capable of indexing into that storage using an offset and per-dimension strides.[5]

Multiple tensors can index the same storage even if they index into the data differently. We can see an example of this in figure 3.4. In fact, when we requested `points[0]` in section 3.2, what we got back is another tensor that indexes the same

[5] `Storage` may not be directly accessible in future PyTorch releases, but what we show here still provides a good mental picture of how tensors work under the hood.

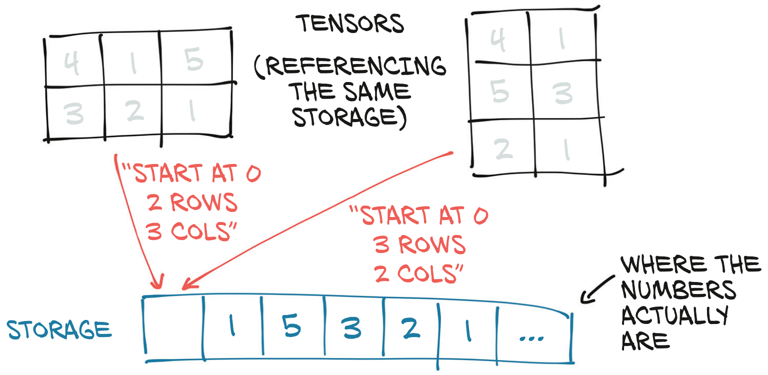


Figure 3.4 **Tensors are views of a `Storage` instance.**

storage as the `points` tensor—just not all of it, and with different dimensionality (1D versus 2D). The underlying memory is allocated only once, however, so creating alternate tensor-views of the data can be done quickly regardless of the size of the data managed by the `Storage` instance.

3.7.1 *Indexing into storage*

Let's see how indexing into the storage works in practice with our 2D points. The storage for a given tensor is accessible using the `.storage` property:

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

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

Even though the tensor reports itself as having three rows and two columns, the storage under the hood is a contiguous array of size 6. In this sense, the tensor just knows how to translate a pair of indices into a location in the storage.

We can also index into a storage manually. For instance:

```
# In[18]:
points_storage = points.storage()
points_storage[0]

# Out[18]:
4.0
```

```
# In[19]:
points.storage()[1]

# Out[19]:
1.0
```

We can't index a storage of a 2D tensor using two indices. The layout of a storage is always one-dimensional, regardless of the dimensionality of any and all tensors that might refer to it.

At this point, it shouldn't come as a surprise that changing the value of a storage leads to changing the content of its referring tensor:

```
# In[20]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_storage = points.storage()
points_storage[0] = 2.0
points

# Out[20]:
tensor([[2., 1.],
        [5., 3.],
        [2., 1.]])
```

3.7.2 Modifying stored values: In-place operations

In addition to the operations on tensors introduced in the previous section, a small number of operations exist only as methods of the `Tensor` object. They are recognizable from a trailing underscore in their name, like `zero_`, which indicates that the method operates *in place* by modifying the input instead of creating a new output tensor and returning it. For instance, the `zero_` method zeros out all the elements of the input. Any method *without* the trailing underscore leaves the source tensor unchanged and instead returns a new tensor:

```
# In[73]:
a = torch.ones(3, 2)

# In[74]:
a.zero_()
a

# Out[74]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

3.8 Tensor metadata: Size, offset, and stride

In order to index into a storage, tensors rely on a few pieces of information that, together with their storage, unequivocally define them: size, offset, and stride. How these interact is shown in figure 3.5. The size (or shape, in NumPy parlance) is a tuple

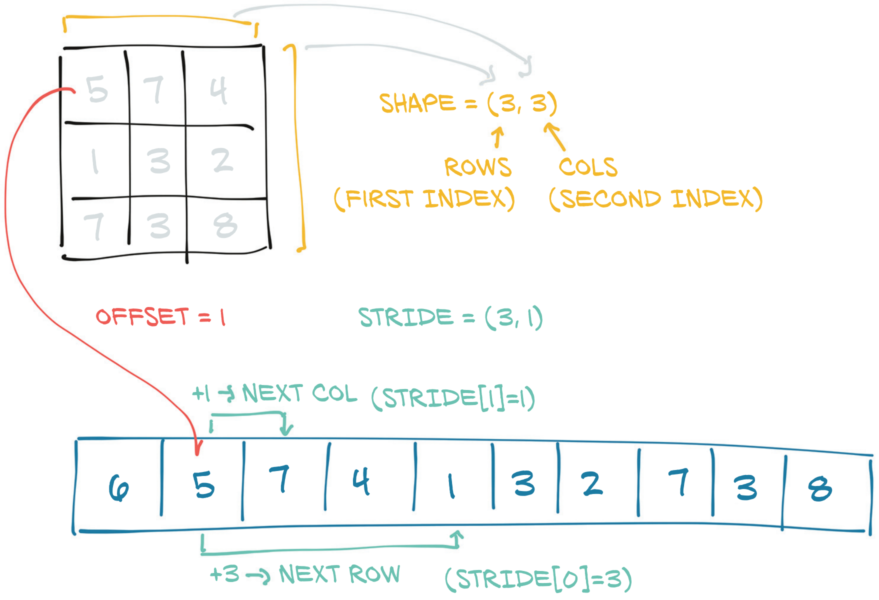


Figure 3.5 Relationship between a tensor's offset, size, and stride. Here the tensor is a view of a larger storage, like one that might have been allocated when creating a larger tensor.

indicating how many elements across each dimension the tensor represents. The storage offset is the index in the storage corresponding to the first element in the tensor. The stride is the number of elements in the storage that need to be skipped over to obtain the next element along each dimension.

3.8.1 *Views of another tensor's storage*

We can get the second point in the tensor by providing the corresponding index:

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

# Out[21]:
2

# In[22]:
second_point.size()

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

The resulting tensor has offset 2 in the storage (since we need to skip the first point, which has two items), and the size is an instance of the `Size` class containing one element, since the tensor is one-dimensional. It’s important to note that this is the same information contained in the `shape` property of tensor objects:

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

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

The stride is a tuple indicating the number of elements in the storage that have to be skipped when the index is increased by 1 in each dimension. For instance, our `points` tensor has a stride of `(2, 1)`:

```
# In[24]:
points.stride()

# Out[24]:
(2, 1)
```

Accessing an element `i, j` in a 2D tensor results in accessing the `storage_offset + stride[0] * i + stride[1] * j` element in the storage. The offset will usually be zero; if this tensor is a view of a storage created to hold a larger tensor, the offset might be a positive value.

This indirection between `Tensor` and `Storage` makes some operations inexpensive, like transposing a tensor or extracting a subtensor, because they do not lead to memory reallocations. Instead, they consist of allocating a new `Tensor` object with a different value for size, storage offset, or stride.

We already extracted a subtensor when we indexed a specific point and saw the storage offset increasing. Let’s see what happens to the size and stride as well:

```
# In[25]:
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,)
```

The bottom line is that the subtensor has one less dimension, as we would expect, while still indexing the same storage as the original `points` tensor. This also means changing the subtensor will have a side effect on the original tensor:

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

# Out[28]:
tensor([[ 4.,  1.],
        [10.,  3.],
        [ 2.,  1.]])
```

This might not always be desirable, so we can eventually clone the subtensor into a new tensor:

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

# Out[29]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

3.8.2 Transposing without copying

Let's try transposing now. Let's take our `points` tensor, which has individual points in the rows and *X* and *Y* coordinates in the columns, and turn it around so that individual points are in the columns. We take this opportunity to introduce the `t` function, a shorthand alternative to `transpose` for two-dimensional tensors:

```
# In[30]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[30]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])

# In[31]:
points_t = points.t()
points_t

# Out[31]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])
```

TIP To help build a solid understanding of the mechanics of tensors, it may be a good idea to grab a pencil and a piece of paper and scribble diagrams like the one in figure 3.5 as we step through the code in this section.

We can easily verify that the two tensors share the same storage

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

# Out[32]:
True
```

and that they differ only in shape and stride:

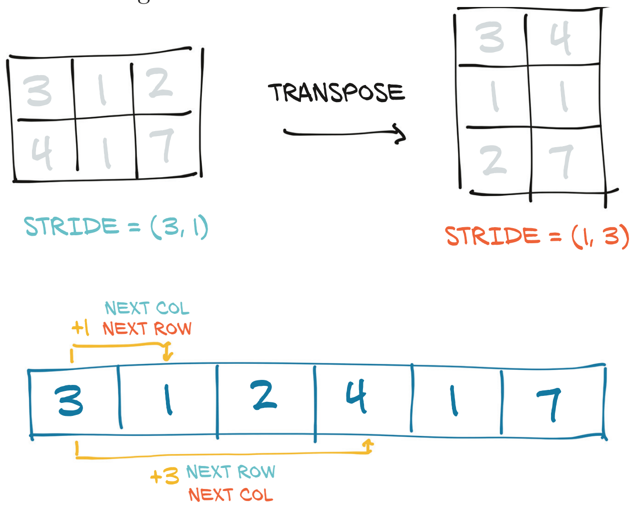
```
# In[33]:
points.stride()

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

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

This tells us that increasing the first index by one in points—for example, going from points[0,0] to points[1,0]—will skip along the storage by two elements, while increasing the second index—from points[0,0] to points[0,1]—will skip along the storage by one. In other words, the storage holds the elements in the tensor sequentially row by row.

We can transpose points into points_t, as shown in figure 3.6. We change the order of the elements in the stride. After that, increasing the row (the first index of the tensor) will skip along the storage by one, just like when we were moving along columns in points. This is the very definition of transposing. No new memory is allocated: transposing is obtained only by creating a new Tensor instance with different stride ordering than the original.



gure 3.6 Transpose
eration applied to a tensor

3.8.3 Transposing in higher dimensions

Transposing in PyTorch is not limited to matrices. We can transpose a multidimensional array by specifying the two dimensions along which transposing (flipping shape and stride) should occur:

```
# In[35]:
some_t = torch.ones(3, 4, 5)
transpose_t = some_t.transpose(0, 2)
some_t.shape

# Out[35]:
torch.Size([3, 4, 5])

# In[36]:
transpose_t.shape

# Out[36]:
torch.Size([5, 4, 3])

# In[37]:
some_t.stride()

# Out[37]:
(20, 5, 1)

# In[38]:
transpose_t.stride()

# Out[38]:
(1, 5, 20)
```

A tensor whose values are laid out in the storage starting from the rightmost dimension onward (that is, moving along rows for a 2D tensor) is defined as `contiguous`. Contiguous tensors are convenient because we can visit them efficiently in order without jumping around in the storage (improving data locality improves performance because of the way memory access works on modern CPUs). This advantage of course depends on the way algorithms visit.

3.8.4 Contiguous tensors

Some tensor operations in PyTorch only work on contiguous tensors, such as `view`, which we'll encounter in the next chapter. In that case, PyTorch will throw an informative exception and require us to call `contiguous` explicitly. It's worth noting that calling `contiguous` will do nothing (and will not hurt performance) if the tensor is already contiguous.

In our case, `points` is contiguous, while its transpose is not:

```
# In[39]:
points.is_contiguous()
```

```
# Out[39]:
True

# In[40]:
points_t.is_contiguous()

# Out[40]:
False
```

We can obtain a new contiguous tensor from a non-contiguous one using the `contiguous` method. The content of the tensor will be the same, but the stride will change, as will the storage:

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

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

# In[42]:
points_t.storage()

# Out[42]:
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage 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([[4., 5., 2.],
        [1., 3., 1.]])

# In[45]:
points_t_cont.stride()

# Out[45]:
(3, 1)
```

```
# In[46]:
points_t_cont.storage()

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

Notice that the storage has been reshuffled in order for elements to be laid out row-by-row in the new storage. The stride has been changed to reflect the new layout.

As a refresher, figure 3.7 shows our diagram again. Hopefully it will all make sense now that we've taken a good look at how tensors are built.

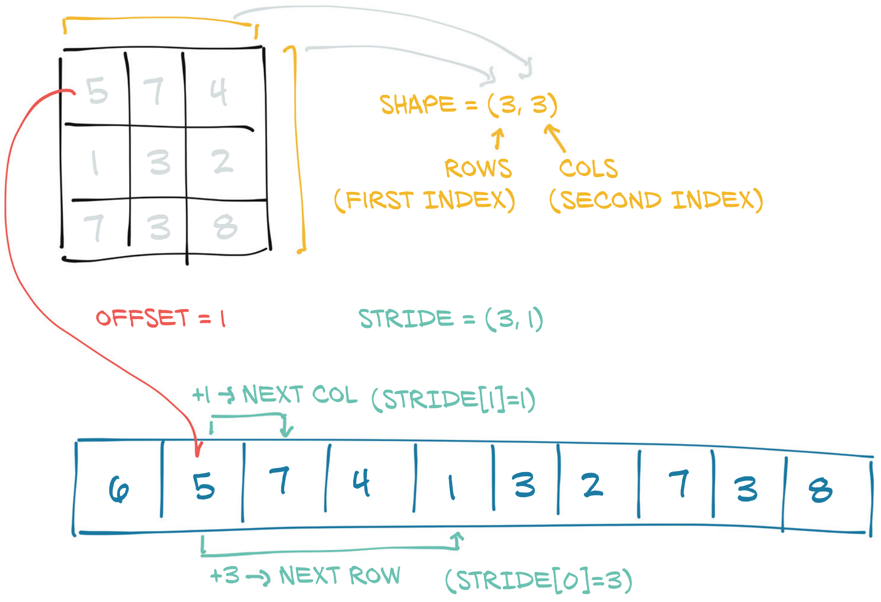


Figure 3.7 Relationship between a tensor's offset, size, and stride. Here the tensor is a view of a larger storage, like one that might have been allocated when creating a larger tensor.

3.9 *Moving tensors to the GPU*

So far in this chapter, when we've talked about storage, we've meant memory on the CPU. PyTorch tensors also can be stored on a different kind of processor: a graphics processing unit (GPU). Every PyTorch tensor can be transferred to (one of) the GPU(s) in order to perform massively parallel, fast computations. All operations that will be performed on the tensor will be carried out using GPU-specific routines that come with PyTorch.

PyTorch support for various GPUs

As of mid-2019, the main PyTorch releases only have acceleration on GPUs that have support for CUDA. PyTorch can run on AMD's ROCm (https://rocm.github.io), and the master repository provides support, but so far, you need to compile it yourself. (Before the regular build process, you need to run `tools/amd_build/build_amd.py` to translate the GPU code.) Support for Google's tensor processing units (TPUs) is a work in progress (https://github.com/pytorch/xla), with the current proof of concept available to the public in Google Colab: https://colab.research.google.com. Implementation of data structures and kernels on other GPU technologies, such as OpenCL, are not planned at the time of this writing.

3.9.1 *Managing a tensor's device attribute*

In addition to `dtype`, a PyTorch `Tensor` also has the notion of `device`, which is where on the computer the tensor data is placed. Here is how we can create a tensor on the GPU by specifying the corresponding argument to the constructor:

```
# In[64]:
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
```

We could instead copy a tensor created on the CPU onto the GPU using the `to` method:

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

Doing so returns a new tensor that has the same numerical data, but stored in the RAM of the GPU, rather than in regular system RAM. Now that the data is stored locally on the GPU, we'll start to see the speedups mentioned earlier when performing mathematical operations on the tensor. In almost all cases, CPU- and GPU-based tensors expose the same user-facing API, making it much easier to write code that is agnostic to where, exactly, the heavy number crunching is running.

If our machine has more than one GPU, we can also decide on which GPU we allocate the tensor by passing a zero-based integer identifying the GPU on the machine, such as

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

At this point, any operation performed on the tensor, such as multiplying all elements by a constant, is carried out on the GPU:

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

Note that the points_gpu tensor is not brought back to the CPU once the result has been computed. Here's what happened in this line:

1. The points tensor is copied to the GPU.
2. A new tensor is allocated on the GPU and used to store the result of the multiplication.
3. A handle to that GPU tensor is returned.

Therefore, if we also add a constant to the result

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

the addition is still performed on the GPU, and no information flows to the CPU (unless we print or access the resulting tensor). In order to move the tensor back to the CPU, we need to provide a cpu argument to the to method, such as

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

We can also use the shorthand methods cpu and cuda instead of the to method to achieve the same goal:

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

It's also worth mentioning that by using the to method, we can change the placement and the data type simultaneously by providing both device and dtype as arguments.

3.10 NumPy interoperability

We've mentioned NumPy here and there. While we do not consider NumPy a prerequisite for reading this book, we strongly encourage you to become familiar with NumPy due to its ubiquity in the Python data science ecosystem. PyTorch tensors can be converted to NumPy arrays and vice versa very efficiently. By doing so, we can take advantage of the huge swath of functionality in the wider Python ecosystem that has built up around the NumPy array type. This zero-copy interoperability with NumPy arrays is due to the storage system working with the Python buffer protocol (https://docs.python.org/3/c-api/buffer.html).

To get a NumPy array out of our points tensor, we just call

```
# 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)
```

which will return a NumPy multidimensional array of the right size, shape, and numerical type. Interestingly, the returned array shares the same underlying buffer with the tensor storage. This means the `numpy` method can be effectively executed at basically no cost, as long as the data sits in CPU RAM. It also means modifying the NumPy array will lead to a change in the originating tensor. If the tensor is allocated on the GPU, PyTorch will make a copy of the content of the tensor into a NumPy array allocated on the CPU.

Conversely, we can obtain a PyTorch tensor from a NumPy array this way

```
# In[56]:
points = torch.from_numpy(points_np)
```

which will use the same buffer-sharing strategy we just described.

NOTE While the default numeric type in PyTorch is 32-bit floating-point, for NumPy it is 64-bit. As discussed in section 3.5.2, we usually want to use 32-bit floating-points, so we need to make sure we have tensors of `dtype torch.float` after converting.

3.11 Generalized tensors are tensors, too

For the purposes of this book, and for the vast majority of applications in general, tensors are multidimensional arrays, just as we've seen in this chapter. If we risk a peek under the hood of PyTorch, there is a twist: how the data is stored under the hood is separate from the tensor API we discussed in section 3.6. Any implementation that meets the contract of that API can be considered a tensor!

PyTorch will cause the right computation functions to be called regardless of whether our tensor is on the CPU or the GPU. This is accomplished through a *dispatching* mechanism, and that mechanism can cater to other tensor types by hooking up the user-facing API to the right backend functions. Sure enough, there are other kinds of tensors: some are specific to certain classes of hardware devices (like Google TPUs), and others have data-representation strategies that differ from the dense array style we've seen so far. For example, sparse tensors store only nonzero entries, along with index information. The PyTorch dispatcher on the left in figure 3.8 is designed to be extensible; the subsequent switching done to accommodate the various numeric types of figure 3.8 shown on the right is a fixed aspect of the implementation coded into each backend.

We will meet *quantized* tensors in chapter 15, which are implemented as another type of tensor with a specialized computational backend. Sometimes the usual tensors we use are called *dense* or *strided* to differentiate them from tensors using other memory layouts.

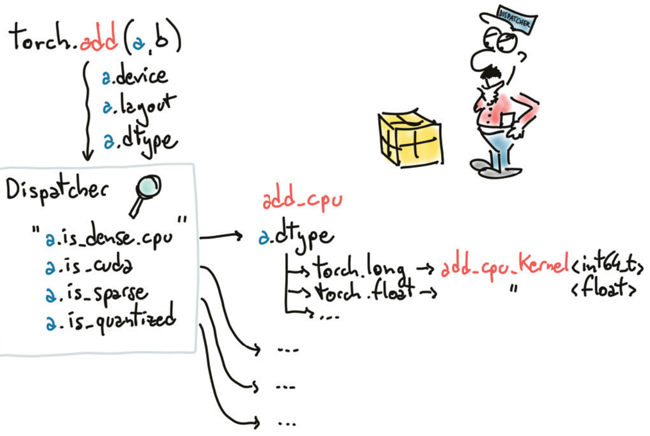


Figure 3.8 The dispatcher in PyTorch is one of its key infrastructure bits.

As with many things, the number of kinds of tensors has grown as PyTorch supports a broader range of hardware and applications. We can expect new kinds to continue to arise as people explore new ways to express and perform computations with PyTorch.

3.12 Serializing tensors

Creating a tensor on the fly is all well and good, but if the data inside is valuable, we will want to save it to a file and load it back at some point. After all, we don't want to have to retrain a model from scratch every time we start running our program! PyTorch uses `pickle` under the hood to serialize the tensor object, plus dedicated serialization code for the storage. Here's how we can save our `points` tensor to an ourpoints.t file:

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

As an alternative, we can pass a file descriptor in lieu of the filename:

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

Loading our points back is similarly a one-liner

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

or, equivalently,

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

While we can quickly save tensors this way if we only want to load them with PyTorch, the file format itself is not interoperable: we can't read the tensor with software other than PyTorch. Depending on the use case, this may or may not be a limitation, but we should learn how to save tensors interoperably for those times when it is. We'll look next at how to do so.

3.12.1 Serializing to HDF5 with h5py

Every use case is unique, but we suspect needing to save tensors interoperably will be more common when introducing PyTorch into existing systems that already rely on different libraries. New projects probably won't need to do this as often.

For those cases when you need to, however, you can use the HDF5 format and library (www.hdfgroup.org/solutions/hdf5). HDF5 is a portable, widely supported format for representing serialized multidimensional arrays, organized in a nested key-value dictionary. Python supports HDF5 through the `h5py` library (www.h5py.org), which accepts and returns data in the form of NumPy arrays.

We can install `h5py` using

```
$ conda install h5py
```

At this point, we can save our `points` tensor by converting it to a NumPy array (at no cost, as we noted earlier) and passing it to the `create_dataset` function:

```
# In[61]:
import h5py

f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'w')
dset = f.create_dataset('coords', data=points.numpy())
f.close()
```

Here `'coords'` is a key into the HDF5 file. We can have other keys—even nested ones. One of the interesting things in HDF5 is that we can index the dataset while on disk and access only the elements we're interested in. Let's suppose we want to load just the last two points in our dataset:

```
# In[62]:
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'r')
dset = f['coords']
last_points = dset[-2:]
```

The data is not loaded when the file is opened or the dataset is required. Rather, the data stays on disk until we request the second and last rows in the dataset. At that point, `h5py` accesses those two columns and returns a NumPy array-like object encapsulating that region in that dataset that behaves like a NumPy array and has the same API.

Owing to this fact, we can pass the returned object to the `torch.from_numpy` function to obtain a tensor directly. Note that in this case, the data is copied over to the tensor's storage:

```
# In[63]:
last_points = torch.from_numpy(dset[-2:])
f.close()
```

Once we're finished loading data, we close the file. Closing the HDFS file invalidates the datasets, and trying to access `dset` afterward will give an exception. As long as we stick to the order shown here, we are fine and can now work with the `last_points` tensor.

3.13 Conclusion

Now we have covered everything we need to get started with representing everything in floats. We'll cover other aspects of tensors—such as creating views of tensors; indexing tensors with other tensors; and broadcasting, which simplifies performing element-wise operations between tensors of different sizes or shapes—as needed along the way.

In chapter 4, we will learn how to represent real-world data in PyTorch. We will start with simple tabular data and move on to something more elaborate. In the process, we will get to know more about tensors.

3.14 Exercises

1. Create a tensor `a` from `list(range(9))`. Predict and then check the size, offset, and stride.
   - a. Create a new tensor using `b = a.view(3, 3)`. What does `view` do? Check that `a` and `b` share the same storage.
   - b. Create a tensor `c = b[1:,1:]`. Predict and then check the size, offset, and stride.
2. Pick a mathematical operation like cosine or square root. Can you find a corresponding function in the `torch` library?
   - a. Apply the function element-wise to `a`. Why does it return an error?
   - b. What operation is required to make the function work?
   - c. Is there a version of your function that operates in place?

3.15 Summary

- Neural networks transform floating-point representations into other floating-point representations. The starting and ending representations are typically human interpretable, but the intermediate representations are less so.
- These floating-point representations are stored in tensors.
- Tensors are multidimensional arrays; they are the basic data structure in PyTorch.

- PyTorch has a comprehensive standard library for tensor creation, manipulation, and mathematical operations.
- Tensors can be serialized to disk and loaded back.
- All tensor operations in PyTorch can execute on the CPU as well as on the GPU, with no change in the code.
- PyTorch uses a trailing underscore to indicate that a function operates in place on a tensor (for example, `Tensor.sqrt_`).