COSE474-2024F-Deep Learning

2.3. Linear Algebra

```
import torch
```

2.3.1. Scalars

We denote scalars by ordinary lower-cased letters (e.g., x, y, and z) and the space of all (continuous) *real-valued* scalars by $\mathbb{R}$. For expedience, we will skip past rigorous definitions of *spaces*: just remember that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ (pronounced "in") denotes membership in a set. For example, $x, y \in \{0, 1\}$ indicates that x and y are variables that can only take values 0 or 1.

(**Scalars are implemented as tensors that contain only one element.**) Below, we assign two scalars and perform the familiar addition, multiplication, division, and exponentiation operations.

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)
```

```
x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

2.3.2. Vectors

- As with their code counterparts, we call these scalars the elements of the vector (synonyms include entries and components).
- For example, if we were training a model to predict the risk of a loan defaulting, we might associate each applicant with a vector whose components correspond to quantities like their income, length of employment, or number of previous defaults. If we were studying the risk of heart attack, each vector might represent a patient and its components might correspond to their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. We denote vectors by bold lowercase letters, (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).
- Vectors are implemented as $1^{\textrm{st}}$-order tensors.
- In general, such tensors can have arbitrary lengths, subject to memory limitations.
- Caution: in Python, as in most programming languages, vector indices start at 0, also known as *zero-based indexing*, whereas in linear algebra subscripts begin at 1 (one-based indexing).

```
x = torch.arange(3)
x
```

```
tensor([0, 1, 2])
```

We can refer to an element of a vector by using a subscript. For example, x_2 denotes the second element of $\mathbf{x}$. Since x_2 is a scalar, we do not bold it. By default, we visualize vectors by stacking their elements vertically.

$$\mathbf{x} = \begin{bmatrix}x_1 \\ \vdots \\ x_n\end{bmatrix},$$

Here $x_1, \ldots, x_n$ are elements of the vector. Later on, we will distinguish between such *column vectors* and *row vectors* whose elements are stacked horizontally. Recall that [**we access a tensor's elements via indexing.**]

```
x[2]
```

```
tensor(2)
```

To indicate that a vector contains n elements, we write $\mathbf{x} \in \mathbb{R}^n$. Formally, we call n the *dimensionality* of the vector. [**In code, this corresponds to the tensor's length**], accessible via Python's built-in `len` function.

```
len(x)
```

```
3
```

The shape is a tuple that indicates a tensor's length along each axis. Tensors with just one axis have shapes with just one element.

```
x.shape
```

```
torch.Size([3])
```

2.3.3. Matrices

Just as scalars are 0^{th}-order tensors and vectors are 1^{st}-order tensors, matrices are 2^{nd}-order tensors. We denote matrices by bold capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), and represent them in code by tensors with two axes. The expression $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates that a matrix $\mathbf{A}$ contains $m \times n$ real-valued scalars, arranged as m rows and n columns. When $m = n$, we say that a matrix is *square*. Visually, we can illustrate any matrix as a table. To refer to an individual element, we subscript both the row and column indices, e.g., a_{ij} is the value that belongs to $\mathbf{A}$'s i^{th} row and j^{th} column:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \\ \end{bmatrix}.$$

:eqlabel: `eq_matrix_def`

In code, we represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ by a 2^{nd}-order tensor with shape (m, n). [**We can convert any appropriately sized $m \times n$ tensor into an $m \times n$ matrix**] by passing the desired shape to `reshape`:

```
A = torch.arange(6).reshape(3, 2)
A
```

```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

Sometimes we want to flip the axes. When we exchange a matrix's rows and columns, the result is called its *transpose*. Formally, we signify a matrix $\mathbf{A}$'s transpose by $\mathbf{A}^\top$ and if $\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for all i and j. Thus, the transpose of an $m \times n$ matrix is an $n \times m$ matrix:

$$\mathbf{A}^\top =\begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}.$$

In code, we can access any (**matrix's transpose**) as follows:

```
A.T
```

```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

[**Symmetric matrices are the subset of square matrices that are equal to their own transposes: $\mathbf{A} = \mathbf{A}^\top$.**] The following matrix is symmetric:

```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

2.3.4. Tensors

- Tensors (**give us a generic way of describing extensions to $n^{\textrm{th}}$$n^{\textrm{th}}$-order arrays.**) We call software objects of the *tensor class* "tensors" precisely because they too can have arbitrary numbers of axes. While it may be confusing to use the word *tensor* for both the mathematical object and its realization in code, our meaning should usually be clear from context. We denote general tensors by capital letters with a special font face (e.g., $\mathsf{X}$$\mathsf{X}$, $\mathsf{Y}$$\mathsf{Y}$, and $\mathsf{Z}$$\mathsf{Z}$) and their indexing mechanism (e.g., $x_{ijk}$$x_{ijk}$ and $[\mathsf{X}]_{1, 2i-1, 3}$$[\mathsf{X}]_{1, 2i-1, 3}$) follows naturally from that of matrices.
- Tensors will become more important when we start working with images. Each image arrives as a $3^{\textrm{rd}}$$3^{\textrm{rd}}$-order tensor with axes corresponding to the height, width, and *channel*. At each spatial location, the intensities of each color (red, green, and blue) are stacked along the channel. Furthermore, a collection of images is represented in code by a $4^{\textrm{th}}$$4^{\textrm{th}}$-order tensor, where distinct images are indexed along the first axis. Higher-order tensors are constructed, as were vectors and matrices, by growing the number of shape components.

```
torch.arange(24).reshape(2, 3, 4)
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

2.3.5. Basic Properties of Tensor Arithmetic

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

The [**elementwise product of two matrices is called their *Hadamard product***] (denoted $\odot$$\odot$). We can spell out the entries of the Hadamard product of two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$$\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$:

$$ \mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11} b_{11} & a_{12} b_{12} & \dots & a_{1n} b_{1n} \\ a_{21} b_{21} & a_{22} b_{22} & \dots & a_{2n} b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} b_{m1} & a_{m2} b_{m2} & \dots & a_{mn} b_{mn} \end{bmatrix}. $$$$ \mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11} b_{11} & a_{12} b_{12} & \dots & a_{1n} b_{1n} \\ a_{21} b_{21} & a_{22} b_{22} & \dots & a_{2n} b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} b_{m1} & a_{m2} b_{m2} & \dots & a_{mn} b_{mn} \end{bmatrix}. $$

```
A * B
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

2.3.6. Reduction

Often, we wish to calculate [**the sum of a tensor's elements.**] To express the sum of the elements in a vector $\mathbf{x}$$\mathbf{x}$ of length $n$$n$, we write $\sum_{i=1}^n x_i$$\sum_{i=1}^n x_i$. There is a simple function for it:

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2.]), tensor(3.))
```

To express [**sums over the elements of tensors of arbitrary shape**], we simply sum over all its axes. For example, the sum of the elements of an $m \times n$$m \times n$ matrix $\mathbf{A}$$\mathbf{A}$ could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum(axis=0).shape
```

```
(torch.Size([2, 3]), torch.Size([3]))
```

Specifying axis=1 will reduce the column dimension (axis 1) by summing up elements of all the columns.

```
A.shape, A.sum(axis=1).shape
```

```
(torch.Size([2, 3]), torch.Size([2]))
```

```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

```
tensor(True)
```

A related quantity is the mean, also called the *average*. We calculate the mean by dividing the sum by the total number of elements. Because computing the mean is so common, it gets a dedicated library function that works analogously to sum.

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(2.5000), tensor(2.5000))
```

The function for calculating the mean can also reduce a tensor along specific axes.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

2.3.7. Non-Reduction Sum

Sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean. This matters when we want to use the broadcast mechanism.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

For instance, since `sum_A` keeps its two axes after summing each row, we can (**divide `A` by `sum_A` with broadcasting**) to create a matrix where each row sums up to $1$$1$.

```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

If we want to calculate the cumulative sum of elements of A along some axis, say axis=0 (row by row), we can call the cumsum function. By design, this function does not reduce the input tensor along any axis.

```
A.cumsum(axis=0)
```

```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

2.3.8. Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra would not deserve its own section. Fortunately, this is where things get more interesting. One of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$$\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$$\mathbf{x}^\top \mathbf{y}$ (also known as *inner product*, $\langle \mathbf{x}, \mathbf{y} \rangle$$\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

Equivalently, (**we can calculate the dot product of two vectors by performing an elementwise multiplication followed by a sum:**)

```
torch.sum(x * y)
```

```
tensor(3.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector $\mathbf{x} \in \mathbb{R}^n$$\mathbf{x} \in \mathbb{R}^n$, and a set of weights, denoted by $\mathbf{w} \in \mathbb{R}^n$$\mathbf{w} \in \mathbb{R}^n$, the weighted sum of the values in $\mathbf{x}$$\mathbf{x}$ according to the weights $\mathbf{w}$$\mathbf{w}$ could be expressed as the dot product $\mathbf{x}^\top \mathbf{w}$$\mathbf{x}^\top \mathbf{w}$. When the weights are nonnegative and sum to $1$$1$, i.e., $\left(\sum_{i=1}^{n} {w_i} = 1\right)$$\left(\sum_{i=1}^{n} {w_i} = 1\right)$, the dot product expresses a *weighted average*. After normalizing two vectors to have unit length, the dot products express the cosine of the angle between them. Later in this section, we will formally introduce this notion of *length*.

2.3.9. Matrix–Vector Products

Now that we know how to calculate dot products, we can begin to understand the *product* between an $m \times n$$m \times n$ matrix $\mathbf{A}$$\mathbf{A}$ and an $n$$n$-dimensional vector $\mathbf{x}$$\mathbf{x}$. To start off, we visualize our matrix in terms of its row vectors

$$\mathbf{A}= \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \\ \end{bmatrix},$$$$\mathbf{A}= \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \\ \end{bmatrix},$$

where each $\mathbf{a}^\top_{i} \in \mathbb{R}^n$$\mathbf{a}^\top_{i} \in \mathbb{R}^n$ is a row vector representing the $i^\textrm{th}$$i^\textrm{th}$ row of the matrix $\mathbf{A}$$\mathbf{A}$.

[**The matrix--vector product $\mathbf{A}\mathbf{x}$$\mathbf{A}\mathbf{x}$ is simply a column vector of length $m$$m$, whose $i^\textrm{th}$$i^\textrm{th}$ element is the dot product $\mathbf{a}^\top_i \mathbf{x}$$\mathbf{a}^\top_i \mathbf{x}$:**]

$$ \mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \\ \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{x} \\ \mathbf{a}^\top_{2} \mathbf{x} \\ \vdots\\ \mathbf{a}^\top_{m} \mathbf{x}\\ \end{bmatrix}. $$$$ \mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \\ \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{x} \\ \mathbf{a}^\top_{2} \mathbf{x} \\ \vdots\\ \mathbf{a}^\top_{m} \mathbf{x}\\ \end{bmatrix}. $$

We can think of multiplication with a matrix $\mathbf{A}\in \mathbb{R}^{m \times n}$$\mathbf{A}\in \mathbb{R}^{m \times n}$ as a transformation that projects vectors from $\mathbb{R}^{n}$$\mathbb{R}^{n}$ to $\mathbb{R}^{m}$$\mathbb{R}^{m}$. These transformations are remarkably useful. For example, we can represent rotations as multiplications by certain square matrices. Matrix--vector products also describe the key calculation involved in computing the outputs of each layer in a neural network given the outputs from the previous layer.

To express a matrix--vector product in code, we use the `mv` function. Note that the column dimension of `A` (its length along axis 1) must be the same as the dimension of `x` (its length). Python has a convenience operator `@` that can execute both matrix--vector and matrix--matrix products (depending on its arguments). Thus we can write `A@x`.

```
A.shape, x.shape, torch.mv(A, x), A@x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

2.3.10. Matrix–Matrix Multiplication

Once you have gotten the hang of dot products and matrix--vector products, then *matrix--matrix multiplication* should be straightforward.

Say that we have two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$$\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$$\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \\ \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \\ \end{bmatrix}.$$$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \\ \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \\ \end{bmatrix}.$$

Let $\mathbf{a}^\top_{i} \in \mathbb{R}^k$$\mathbf{a}^\top_{i} \in \mathbb{R}^k$ denote the row vector representing the $i^\textrm{th}$$i^\textrm{th}$ row of the matrix $\mathbf{A}$$\mathbf{A}$ and let $\mathbf{b}_{j} \in \mathbb{R}^k$$\mathbf{b}_{j} \in \mathbb{R}^k$ denote the column vector from the $j^\textrm{th}$$j^\textrm{th}$ column of the matrix $\mathbf{B}$$\mathbf{B}$:

$$\mathbf{A}= \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix}, \quad \mathbf{B}=\begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix}. $$$$\mathbf{A}= \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix}, \quad \mathbf{B}=\begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix}. $$

To form the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$$\mathbf{C} \in \mathbb{R}^{n \times m}$, we simply compute each element $c_{ij}$$c_{ij}$ as the dot product between the $i^{\textrm{th}}$$i^{\textrm{th}}$ row of $\mathbf{A}$$\mathbf{A}$ and the $j^{\textrm{th}}$$j^{\textrm{th}}$ column of $\mathbf{B}$$\mathbf{B}$, i.e., $\mathbf{a}^\top_i \mathbf{b}_j$$\mathbf{a}^\top_i \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix} \begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{b}_1 & \mathbf{a}^\top_{1}\mathbf{b}_2& \cdots & \mathbf{a}^\top_{1} \mathbf{b}_m \\ \mathbf{a}^\top_{2}\mathbf{b}_1 & \mathbf{a}^\top_{2} \mathbf{b}_2 & \cdots & \mathbf{a}^\top_{2} \mathbf{b}_m \\ \vdots & \vdots & \ddots &\vdots\\ \mathbf{a}^\top_{n} \mathbf{b}_1 & \mathbf{a}^\top_{n}\mathbf{b}_2& \cdots& \mathbf{a}^\top_{n} \mathbf{b}_m \end{bmatrix}. $$$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix} \begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{b}_1 & \mathbf{a}^\top_{1}\mathbf{b}_2& \cdots & \mathbf{a}^\top_{1} \mathbf{b}_m \\ \mathbf{a}^\top_{2}\mathbf{b}_1 & \mathbf{a}^\top_{2} \mathbf{b}_2 & \cdots & \mathbf{a}^\top_{2} \mathbf{b}_m \\ \vdots & \vdots & \ddots &\vdots\\ \mathbf{a}^\top_{n} \mathbf{b}_1 & \mathbf{a}^\top_{n}\mathbf{b}_2& \cdots& \mathbf{a}^\top_{n} \mathbf{b}_m \end{bmatrix}. $$

[**We can think of the matrix--matrix multiplication $\mathbf{AB}$$\mathbf{AB}$ as performing $m$$m$ matrix--vector products or $m \times n$$m \times n$ dot products and stitching the results together to form an $n \times m$$n \times m$ matrix.**] In the following snippet, we perform matrix multiplication on `A` and `B`. Here, `A` is a matrix with two rows and three columns, and `B` is a matrix with three rows and four columns. After multiplication, we obtain a matrix with two rows and four columns.

```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

The term *matrix--matrix multiplication* is often simplified to *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.11. Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* it is. For instance, the $\ell_2$$\ell_2$ norm measures the (Euclidean) length of a vector. Here, we are employing a notion of *size* that concerns the magnitude of a vector's components (not its dimensionality).

A norm is a function $\| \cdot \|$$\| \cdot \|$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector $\mathbf{x}$$\mathbf{x}$, if we scale (all elements of) the vector by a scalar $\alpha \in \mathbb{R}$$\alpha \in \mathbb{R}$, its norm scales accordingly: $$\|\alpha \mathbf{x}\| = |\alpha| \|\mathbf{x}\|.$$$$\|\alpha \mathbf{x}\| = |\alpha| \|\mathbf{x}\|.$$
2. For any vectors $\mathbf{x}$$\mathbf{x}$ and $\mathbf{y}$$\mathbf{y}$: norms satisfy the triangle inequality: $$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|.$$$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|.$$
3. The norm of a vector is nonnegative and it only vanishes if the vector is zero: $$\|\mathbf{x}\| > 0 \textrm{ for all } \mathbf{x} \neq 0.$$$$\|\mathbf{x}\| > 0 \textrm{ for all } \mathbf{x} \neq 0.$$

Many functions are valid norms and different norms encode different notions of size. The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of a right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called [**the $\ell_2$$\ell_2$ *norm***] and expressed as

(**$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}.$$$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}.$$**)

The method `norm` calculates the $\ell_2$$\ell_2$ norm.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

[**The $\ell_1$$\ell_1$ norm**] is also common and the associated measure is called the Manhattan distance. By definition, the $\ell_1$$\ell_1$ norm sums the absolute values of a vector's elements:

(**$$\|\mathbf{x}\|_1 = \sum_{i=1}^n \left|x_i \right|.$$$$\|\mathbf{x}\|_1 = \sum_{i=1}^n \left|x_i \right|.$$**)

Compared to the $\ell_2$$\ell_2$ norm, it is less sensitive to outliers. To compute the $\ell_1$$\ell_1$ norm, we compose the absolute value with the sum operation.

```
torch.abs(u).sum()
```

```
tensor(7.)
```

Both the $\ell_2$$\ell_2$ and $\ell_1$$\ell_1$ norms are special cases of the more general $\ell_p$$\ell_p$ *norms*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n \left|x_i \right|^p \right)^{1/p}.$$$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n \left|x_i \right|^p \right)^{1/p}.$$

In the case of matrices, matters are more complicated. After all, matrices can be viewed both as collections of individual entries *and* as objects that operate on vectors and transform them into other vectors. For instance, we can ask by how much longer the matrix--vector product $\mathbf{X} \mathbf{v}$$\mathbf{X} \mathbf{v}$ could be relative to $\mathbf{v}$$\mathbf{v}$. This line of thought leads to what is called the *spectral* norm. For now, we introduce [**the *Frobenius norm*, which is much easier to compute**] and defined as the square root of the sum of the squares of a matrix's elements:

[**$$\|\mathbf{X}\|_\textrm{F} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}.$$$$\|\mathbf{X}\|_\textrm{F} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}.$$**]

The Frobenius norm behaves as if it were an $\ell_2$$\ell_2$ norm of a matrix-shaped vector. Invoking the following function will calculate the Frobenius norm of a matrix.

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

While we do not want to get too far ahead of ourselves, we already can plant some intuition about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *maximize* the revenue associated with a recommender model; *minimize* the distance between predictions and the ground truth observations; *minimize* the distance between representations of photos of the same person while *maximizing* the distance between representations of photos of different people. These distances, which constitute the objectives of deep learning algorithms, are often expressed as norms.

2.3.12. Discussion

In this section, we have reviewed all the linear algebra that you will need to understand a significant chunk of modern deep learning. There is a lot more to linear algebra, though, and much of it is useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you will be more inclined to learn more mathematics once you have gotten your hands dirty applying machine learning to real datasets. So while we reserve the right to introduce more mathematics later on, we wrap up this section here.

If you are eager to learn more linear algebra, there are many excellent books and online resources. For a more advanced crash course, consider checking out :citet: `Strang.1993`, :citet: `Kolter.2008`, and :citet: `Petersen.Pedersen.ea.2008`.

To recap:

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as `sum` and `mean`, respectively.
- Elementwise products are called Hadamard products. By contrast, dot products, matrix--vector products, and matrix--matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix--matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the $\ell_1$$\ell_1$ and $\ell_2$$\ell_2$ norms, and common matrix norms include the *spectral* and *Frobenius* norms.

```
Start coding or generate with AI.
```