17 | Appendix: Mathematics for Deep Learning

Brent Werness (*Amazon*), **Rachel Hu** (*Amazon*), and authors of this book

One of the wonderful parts of modern deep learning is the fact that much of it can be understood and used without a full understanding of the mathematics below it. This is a sign that the field is maturing. Just as most software developers no longer need to worry about the theory of computable functions, neither should deep learning practitioners need to worry about the theoretical foundations of maximum likelihood learning.

But, we are not quite there yet.

In practice, you will sometimes need to understand how architectural choices influence gradient flow, or the implicit assumptions you make by training with a certain loss function. You might need to know what in the world entropy measures, and how it can help you understand exactly what bits-per-character means in your model. These all require deeper mathematical understanding.

This appendix aims to provide you the mathematical background you need to understand the core theory of modern deep learning, but it is not exhaustive. We will begin with examining linear algebra in greater depth. We develop a geometric understanding of all the common linear algebraic objects and operations that will enable us to visualize the effects of various transformations on our data. A key element is the development of the basics of eigen-decompositions.

We next develop the theory of differential calculus to the point that we can fully understand why the gradient is the direction of steepest descent, and why back-propagation takes the form it does. Integral calculus is then discussed to the degree needed to support our next topic, probability theory.

Problems encountered in practice frequently are not certain, and thus we need a language to speak about uncertain things. We review the theory of random variables and the most commonly encountered distributions so we may discuss models probabilistically. This provides the foundation for the naive Bayes classifier, a probabilistic classification technique.

Closely related to probability theory is the study of statistics. While statistics is far too large a field to do justice in a short section, we will introduce fundamental concepts that all machine learning practitioners should be aware of, in particular: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals.

Last, we turn to the topic of information theory, which is the mathematical study of information storage and transmission. This provides the core language by which we may discuss quantitatively how much information a model holds on a domain of discourse.

Taken together, these form the core of the mathematical concepts needed to begin down the path towards a deep understanding of deep learning.

17.1 Geometry and Linear Algebraic Operations

In Section 2.3, we encountered the basics of linear algebra and saw how it could be used to express common operations for transforming our data. Linear algebra is one of the key mathematical pillars underlying much of the work that we do deep learning and in machine learning more broadly. While Section 2.3 contained enough machinery to communicate the mechanics of modern deep learning models, there is a lot more to the subject. In this section, we will go deeper, highlighting some geometric interpretations of linear algebra operations, and introducing a few fundamental concepts, including of eigenvalues and eigenvectors.

17.1.1 Geometry of Vectors

First, we need to discuss the two common geometric interpretations of vectors, as either points or directions in space. Fundamentally, a vector is a list of numbers such as the Python list below.

```
v = [1, 7, 0, 1]
```

Mathematicians most often write this as either a *column* or *row* vector, which is to say either as

$$\mathbf{x} = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 1 \end{bmatrix}, \tag{17.1.1}$$

or

$$\mathbf{x}^\top = \begin{bmatrix} 1 & 7 & 0 & 1 \end{bmatrix}. \tag{17.1.2}$$

These often have different interpretations, where data points are column vectors and weights used to form weighted sums are row vectors. However, it can be beneficial to be flexible. Matrices are useful data structures: they allow us to organize data that have different modalities of variation. For example, rows in our matrix might correspond to different houses (data points), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 2.2. Thus, although the default orientation of a single vector is a column vector, in a matrix that represents a tabular dataset, it is more conventional to treat each data point as a row vector in the matrix. And, as we will see in later chapters, this convention will enable common deep learning practices. For example, along the outermost axis of an `ndarray`, we can access or enumerate minibatches of data points, or just data points if no minibatch exists.

Given a vector, the first interpretation that we should give it is as a point in space. In two or three dimensions, we can visualize these points by using the components of the vectors to define the location of the points in space compared to a fixed reference called the *origin*. This can be seen in Fig. 17.1.1.

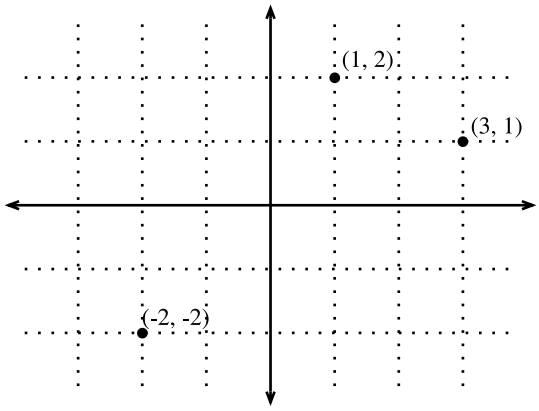


Fig. 17.1.1: An illustration of visualizing vectors as points in the plane. The first component of the vector gives the x-coordinate, the second component gives the y-coordinate. Higher dimensions are analogous, although much harder to visualize.

This geometric point of view allows us to consider the problem on a more abstract level. No longer faced with some insurmountable seeming problem like classifying pictures as either cats or dogs, we can start considering tasks abstractly as collections of points in space and picturing the task as discovering how to separate two distinct clusters of points.

In parallel, there is a second point of view that people often take of vectors: as directions in space. Not only can we think of the vector $\mathbf{v} = [2, 3]^\top$ as the location 2 units to the right and 3 units up from the origin, we can also think of it as the direction itself to take 2 steps to the right and 3 steps up. In this way, we consider all the vectors in figure Fig. 17.1.2 the same.

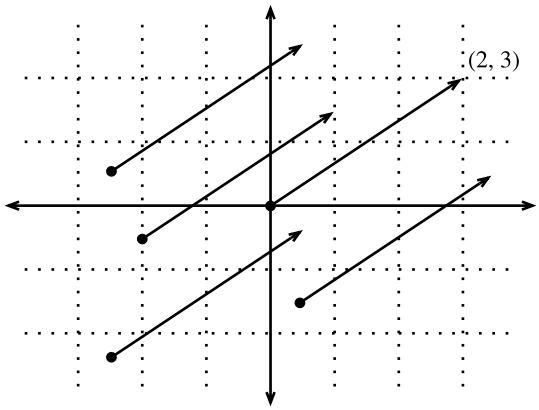


Fig. 17.1.2: Any vector can be visualized as an arrow in the plane. In this case, every vector drawn is a representation of the vector $(2, 3)$.

One of the benefits of this shift is that we can make visual sense of the act of vector addition. In particular, we follow the directions given by one vector, and then follow the directions given by the other, as is seen in Fig. 17.1.3.

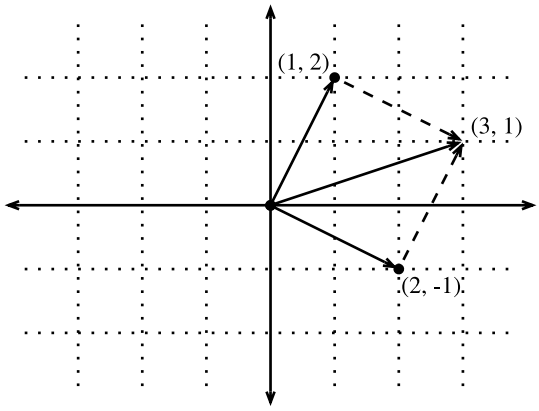


Fig. 17.1.3: We can visualize vector addition by first following one vector, and then another.

Vector subtraction has a similar interpretation. By considering the identity that $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, we see that the vector $\mathbf{u} - \mathbf{v}$ is the direction that takes us from the point $\mathbf{u}$ to the point $\mathbf{v}$.

17.1.2 Dot Products and Angles

As we saw in Section 2.3, if we take two column vectors say $\mathbf{u}$ and $\mathbf{v}$, we can form their dot product by computing:

$$\mathbf{u}^\top \mathbf{v} = \sum_i u_i \cdot v_i. \tag{17.1.3}$$

Because (17.1.3) is symmetric, we will mirror the notation of classical multiplication and write

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}, \tag{17.1.4}$$

to highlight the fact that exchanging the order of the vectors will yield the same answer.

The dot product (17.1.3) also admits a geometric interpretation: it is closely related to the angle between two vectors. Consider the angle shown in Fig. 17.1.4.

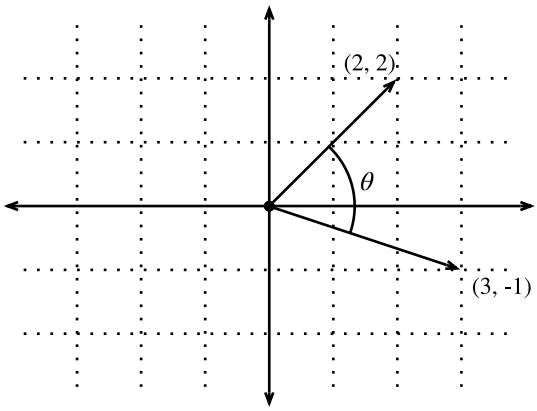


Fig. 17.1.4: Between any two vectors in the plane there is a well defined angle θ. We will see this angle is intimately tied to the dot product.

To start, let's consider two specific vectors:

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{17.1.5}$$

The vector $\mathbf{v}$ is length r and runs parallel to the x-axis, and the vector $\mathbf{w}$ is of length s and at angle θ with the x-axis.

If we compute the dot product of these two vectors, we see that

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{17.1.6}$$

With some simple algebraic manipulation, we can rearrange terms to obtain

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{17.1.7}$$

In short, for these two specific vectors, the dot product combined with the norms tell us the angle between the two vectors. This same fact is true in general. We will not derive the expression here, however, if we consider writing $\|\mathbf{v} - \mathbf{w}\|^2$ in two ways: one with the dot product, and the other geometrically using the law of cosines, we can obtain the full relationship. Indeed, for any two vectors $\mathbf{v}$ and $\mathbf{w}$, the angle between the two vectors is

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{17.1.8}$$

This is a nice result since nothing in the computation references two-dimensions. Indeed, we can use this in three or three million dimensions without issue.

As a simple example, let's see how to compute the angle between a pair of vectors:

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import gluon, np, npx
npx.set_np()

def angle(v, w):
    return np.arccos(v.dot(w) / (np.linalg.norm(v) * np.linalg.norm(w)))

angle(np.array([0, 1, 2]), np.array([2, 3, 4]))
```

```
array(0.41899002)
```

We will not use it right now, but it is useful to know that we will refer to vectors for which the angle is $\pi/2$ (or equivalently $90°$) as being *orthogonal*. By examining the equation above, we see that this happens when $\theta = \pi/2$, which is the same thing as $\cos(\theta) = 0$. The only way this can happen is if the dot product itself is zero, and two vectors are orthogonal if and only if $\mathbf{v} \cdot \mathbf{w} = 0$. This will prove to be a helpful formula when understanding objects geometrically.

It is reasonable to ask: why is computing the angle useful? The answer comes in the kind of invariance we expect data to have. Consider an image, and a duplicate image, where every pixel value is the same but 10% the brightness. The values of the individual pixels are in general far from the original values. Thus, if one computed the distance between the original image and the darker one, the distance can be large.

However, for most ML applications, the *content* is the same—it is still an image of a cat as far as a cat/dog classifier is concerned. However, if we consider the angle, it is not hard to see that for

any vector $\mathbf{v}$, the angle between $\mathbf{v}$ and $0.1 \cdot \mathbf{v}$ is zero. This corresponds to the fact that scaling vectors keeps the same direction and just changes the length. The angle considers the darker image identical.

Examples like this are everywhere. In text, we might want the topic being discussed to not change if we write twice as long of document that says the same thing. For some encoding (such as counting the number of occurrences of words in some vocabulary), this corresponds to a doubling of the vector encoding the document, so again we can use the angle.

Cosine Similarity

In ML contexts where the angle is employed to measure the closeness of two vectors, practitioners adopt the term *cosine similarity* to refer to the portion

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{17.1.9}$$

The cosine takes a maximum value of 1 when the two vectors point in the same direction, a minimum value of -1 when they point in opposite directions, and a value of 0 when the two vectors are orthogonal. Note that if the components of high-dimensional vectors are sampled randomly with mean 0, their cosine will nearly always be close to 0.

17.1.3 Hyperplanes

In addition to working with vectors, another key object that you must understand to go far in linear algebra is the *hyperplane*, a generalization to higher dimensions of a line (two dimensions) or of a plane (three dimensions). In an d-dimensional vector space, a hyperplane has $d - 1$ dimensions and divides the space into two half-spaces.

Let's start with an example. Suppose that we have a column vector $\mathbf{w} = [2, 1]^\top$. We want to know, "what are the points $\mathbf{v}$ with $\mathbf{w} \cdot \mathbf{v} = 1$?" By recalling the connection between dot products and angles above (17.1.8), we can see that this is equivalent to

$$\|\mathbf{v}\|\|\mathbf{w}\| \cos(\theta) = 1 \iff \|\mathbf{v}\| \cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{17.1.10}$$

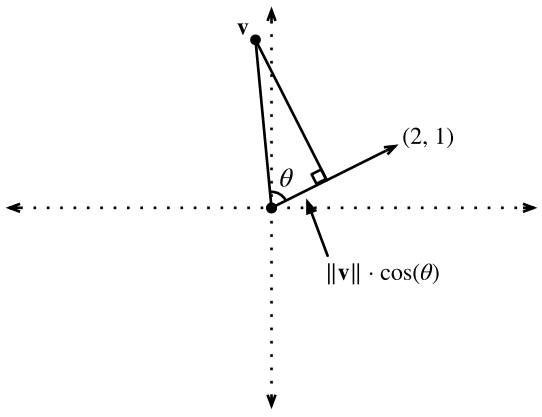


Fig. 17.1.5: Recalling trigonometry, we see the formula $\|\mathbf{v}\| \cos(\theta)$ is the length of the projection of the vector $\mathbf{v}$ onto the direction of $\mathbf{w}$

If we consider the geometric meaning of this expression, we see that this is equivalent to saying that the length of the projection of $\mathbf{v}$ onto the direction of $\mathbf{w}$ is exactly $1/\|\mathbf{w}\|$, as is shown in Fig. 17.1.5. The set of all points where this is true is a line at right angles to the vector $\mathbf{w}$. If we wanted, we could find the equation for this line and see that it is $2x + y = 1$ or equivalently $y = 1 - 2x$.

If we now look at what happens when we ask about the set of points with $\mathbf{w} \cdot \mathbf{v} > 1$ or $\mathbf{w} \cdot \mathbf{v} < 1$, we can see that these are cases where the projections are longer or shorter than $1/\|\mathbf{w}\|$, respectively. Thus, those two inequalities define either side of the line. In this way, we have found a way to cut our space into two halves, where all the points on one side have dot product below a threshold, and the other side above as we see in Fig. 17.1.6.

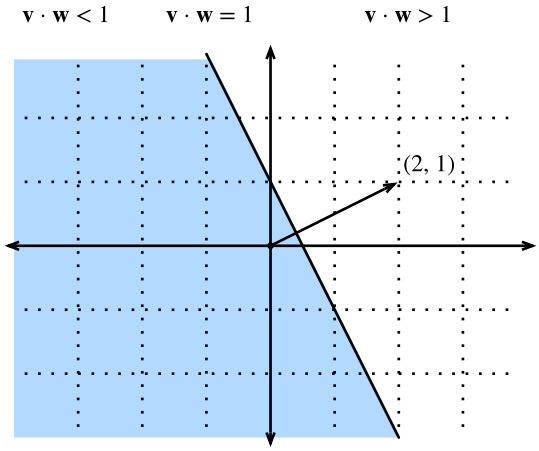


Fig. 17.1.6: If we now consider the inequality version of the expression, we see that our hyperplane (in this case: just a line) separates the space into two halves.

The story in higher dimension is much the same. If we now take $\mathbf{w} = [1, 2, 3]^\top$ and ask about the points in three dimensions with $\mathbf{w} \cdot \mathbf{v} = 1$, we obtain a plane at right angles to the given vector $\mathbf{w}$. The two inequalities again define the two sides of the plane as is shown in Fig. 17.1.7.

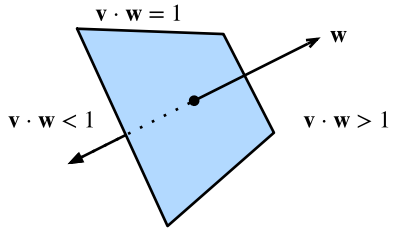


Fig. 17.1.7: Hyperplanes in any dimension separate the space into two halves.

While our ability to visualize runs out at this point, nothing stops us from doing this in tens, hundreds, or billions of dimensions. This occurs often when thinking about machine learned models. For instance, we can understand linear classification models like those from Section 3.4, as methods to find hyperplanes that separate the different target classes. In this context, such hyperplanes are often referred to as *decision planes*. The majority of deep learned classification models end with a linear layer fed into a softmax, so one can interpret the role of the deep neural network to be to find a non-linear embedding such that the target classes can be separated cleanly by hyperplanes.

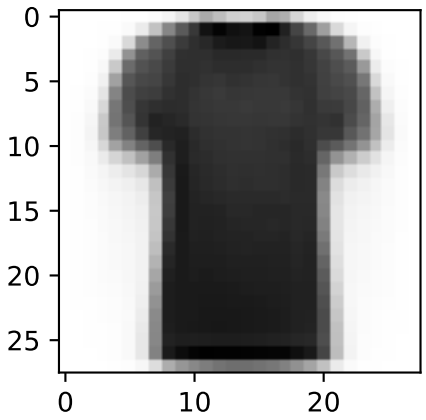
To give a hand-built example, notice that we can produce a reasonable model to classify tiny images of t-shirts and trousers from the Fashion MNIST dataset (seen in Section 3.5) by just taking

the vector between their means to define the decision plane and eyeball a crude threshold. First we will load the data and compute the averages.

```
# Load in the dataset
train = gluon.data.vision.FashionMNIST(train=True)
test = gluon.data.vision.FashionMNIST(train=False)

X_train_0 = np.stack([x[0] for x in train if x[1] == 0]).astype(float)
X_train_1 = np.stack([x[0] for x in train if x[1] == 1]).astype(float)
X_test = np.stack(
    [x[0] for x in test if x[1] == 0 or x[1] == 1]).astype(float)
y_test = np.stack(
    [x[1] for x in test if x[1] == 0 or x[1] == 1]).astype(float)

# Compute averages
ave_0 = np.mean(X_train_0, axis=0)
ave_1 = np.mean(X_train_1, axis=0)
```

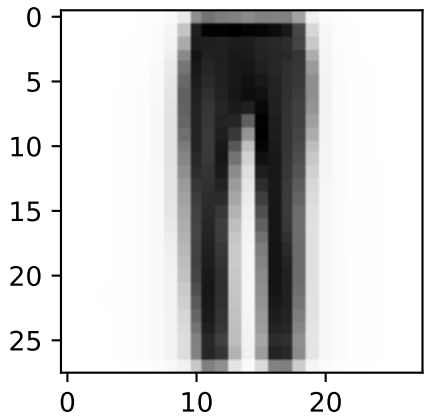
It can be informative to examine these averages in detail, so let's plot what they look like. In this case, we see that the average indeed resembles a blurry image of a t-shirt.

```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



In the second case, we again see that the average resembles a blurry image of trousers.

```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



In a fully machine learned solution, we would learn the threshold from the dataset. In this case, I simply eyeballed a threshold that looked good on the training data by hand.

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
predictions = X_test.reshape(2000, -1).dot(w.flatten()) > -1500000

# Accuracy
np.mean(predictions.astype(y_test.dtype) == y_test, dtype=np.float64)
```

```
array(0.801, dtype=float64)
```

17.1.4 Geometry of Linear Transformations

Through Section 2.3 and the above discussions, we have a solid understanding of the geometry of vectors, lengths, and angles. However, there is one important object we have omitted discussing, and that is a geometric understanding of linear transformations represented by matrices. Fully internalizing what matrices can do to transform data between two potentially different high dimensional spaces takes significant practice, and is beyond the scope of this appendix. However, we can start building up intuition in two dimensions.

Suppose that we have some matrix:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{17.1.11}$$

If we want to apply this to an arbitrary vector $\mathbf{v} = [x, y]^\top$, we multiply and see that

$$\begin{aligned} \mathbf{A}\mathbf{v} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \\ &= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} \\ &= x \left\{ \mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} + y \left\{ \mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}. \end{aligned} \tag{17.1.12}$$

This may seem like an odd computation, where something clear became somewhat impenetrable. However, it tells us that we can write the way that a matrix transforms *any* vector in terms of how it transforms *two specific vectors*: $[1, 0]^\top$ and $[0, 1]^\top$. This is worth considering for a moment. We have essentially reduced an infinite problem (what happens to any pair of real numbers) to a finite one (what happens to these specific vectors). These vectors are an example a *basis*, where we can write any vector in our space as a weighted sum of these *basis vectors*.

Let's draw what happens when we use the specific matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \tag{17.1.13}$$

If we look at the specific vector $\mathbf{v} = [2, -1]^\top$, we see this is $2 \cdot [1, 0]^\top + -1 \cdot [0, 1]^\top$, and thus we know that the matrix A will send this to $2(\mathbf{A}[1, 0]^\top) + -1(\mathbf{A}[0, 1])^\top = 2[1, -1]^\top - [2, 3]^\top = [0, -5]^\top$. If we follow this logic through carefully, say by considering the grid of all integer pairs of points, we see that what happens is that the matrix multiplication can skew, rotate, and scale the grid, but the grid structure must remain as you see in Fig. 17.1.8.

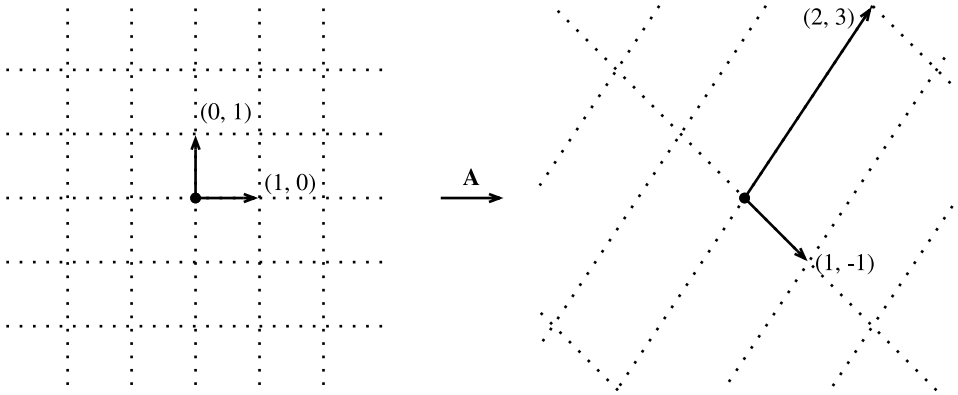


Fig. 17.1.8: The matrix **A** acting on the given basis vectors. Notice how the entire grid is transported along with it.

This is the most important intuitive point to internalize about linear transformations represented by matrices. Matrices are incapable of distorting some parts of space differently than others. All they can do is take the original coordinates on our space and skew, rotate, and scale them.

Some distortions can be severe. For instance the matrix

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \tag{17.1.14}$$

compresses the entire two-dimensional plane down to a single line. Identifying and working with such transformations are the topic of a later section, but geometrically we can see that this is fundamentally different from the types of transformations we saw above. For instance, the result from matrix **A** can be "bent back" to the original grid. The results from matrix **B** cannot because we will never know where the vector $[1, 2]^\top$ came from—was it $[1, 1]^\top$ or $[0, -1]^\top$?

While this picture was for a 2×2 matrix, nothing prevents us from taking the lessons learned into higher dimensions. If we take similar basis vectors like $[1, 0, \ldots, 0]$ and see where our matrix sends them, we can start to get a feeling for how the matrix multiplication distorts the entire space in whatever dimension space we are dealing with.

17.1.5 Linear Dependence

Consider again the matrix

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \tag{17.1.15}$$

This compresses the entire plane down to live on the single line $y = 2x$. The question now arises: is there some way we can detect this just looking at the matrix itself? The answer is that indeed we can. Let's take $\mathbf{b}_1 = [2, 4]^\top$ and $\mathbf{b}_2 = [-1, -2]^\top$ be the two columns of $\mathbf{B}$. Remember that we can write everything transformed by the matrix $\mathbf{B}$ as a weighted sum of the columns of the matrix: like $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$. We call this a *linear combination*. The fact that $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ means that we can write any linear combination of those two columns entirely in terms of say $\mathbf{b}_2$ since

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{17.1.16}$$

This means that one of the columns is, in a sense, redundant because it does not define a unique direction in space. This should not surprise us too much since we already saw that this matrix collapses the entire plane down into a single line. Moreover, we see that the linear dependence $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ captures this. To make this more symmetrical between the two vectors, we will write this as

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{17.1.17}$$

In general, we will say that a collection of vectors $\mathbf{v}_1, \ldots \mathbf{v}_k$ are *linearly dependent* if there exist coefficients $a_1, \ldots, a_k$ *not all equal to zero* so that

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{17.1.18}$$

In this case, we can solve for one of the vectors in terms of some combination of the others, and effectively render it redundant. Thus, a linear dependence in the columns of a matrix is a witness to the fact that our matrix is compressing the space down to some lower dimension. If there is no linear dependence we say the vectors are *linearly independent*. If the columns of a matrix are linearly independent, no compression occurs and the operation can be undone.

17.1.6 Rank

If we have a general $n \times m$ matrix, it is reasonable to ask what dimension space the matrix maps into. A concept known as the *rank* will be our answer. In the previous section, we noted that a linear dependence bears witness to compression of space into a lower dimension and so we will be able to use this to define the notion of rank. In particular, the rank of a matrix $\mathbf{A}$ is the largest number of linearly independent columns amongst all subsets of columns. For example, the matrix

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \tag{17.1.19}$$

has rank$(B) = 1$, since the two columns are linearly dependent, but either column by itself is not linearly dependent. For a more challenging example, we can consider

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \tag{17.1.20}$$

and show that **C** has rank two since, for instance, the first two columns are linearly independent, however any of the four collections of three columns are dependent.

This procedure, as described, is very inefficient. It requires looking at every subset of the columns of our given matrix, and thus is potentially exponential in the number of columns. Later we will see a more computationally efficient way to compute the rank of a matrix, but for now, this is sufficient to see that the concept is well defined and understand the meaning.

17.1.7 Invertibility

We have seen above that multiplication by a matrix with linearly dependent columns cannot be undone, i.e., there is no inverse operation that can always recover the input. However, multiplication by a full-rank matrix (i.e., some **A** that is $n \times n$ matrix with rank n), we should always be able to undo it. Consider the matrix

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{17.1.21}$$

which is the matrix with ones along the diagonal, and zeros elsewhere. We call this the *identity* matrix. It is the matrix which leaves our data unchanged when applied. To find a matrix which undoes what our matrix **A** has done, we want to find a matrix $\mathbf{A}^{-1}$ such that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{17.1.22}$$

If we look at this as a system, we have $n \times n$ unknowns (the entries of $\mathbf{A}^{-1}$) and $n \times n$ equations (the equality that needs to hold between every entry of the product $\mathbf{A}^{-1}\mathbf{A}$ and every entry of $\mathbf{I}$) so we should generically expect a solution to exist. Indeed, in the next section we will see a quantity called the *determinant*, which has the property that as long as the determinant is not zero, we can find a solution. We call such a matrix $\mathbf{A}^{-1}$ the *inverse* matrix. As an example, if **A** is the general 2×2 matrix

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{17.1.23}$$

then we can see that the inverse is

$$\frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{17.1.24}$$

We can test to see this by seeing that multiplying by the inverse given by the formula above works in practice.

```
M = np.array([[1, 2], [1, 4]])
M_inv = np.array([[2, -1], [-0.5, 0.5]])
M_inv.dot(M)
```

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

Numerical Issues

While the inverse of a matrix is useful in theory, we must say that most of the time we do not wish to *use* the matrix inverse to solve a problem in practice. In general, there are far more numerically stable algorithms for solving linear equations like

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{17.1.25}$$

than computing the inverse and multiplying to get

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{17.1.26}$$

Just as division by a small number can lead to numerical instability, so can inversion of a matrix which is close to having low rank.

Moreover, it is common that the matrix $\mathbf{A}$ is *sparse*, which is to say that it contains only a small number of non-zero values. If we were to explore examples, we would see that this does not mean the inverse is sparse. Even if $\mathbf{A}$ was a 1 million by 1 million matrix with only 5 million non-zero entries (and thus we need only store those 5 million), the inverse will typically have almost every entry non-negative, requiring us to store all 1M^2 entries—that is 1 trillion entries!

While we do not have time to dive all the way into the thorny numerical issues frequently encountered when working with linear algebra, we want to provide you with some intuition about when to proceed with caution, and generally avoiding inversion in practice is a good rule of thumb.

17.1.8 Determinant

The geometric view of linear algebra gives an intuitive way to interpret a a fundamental quantity known as the *determinant*. Consider the grid image from before, but now with a highlighted region (Fig. 17.1.9).

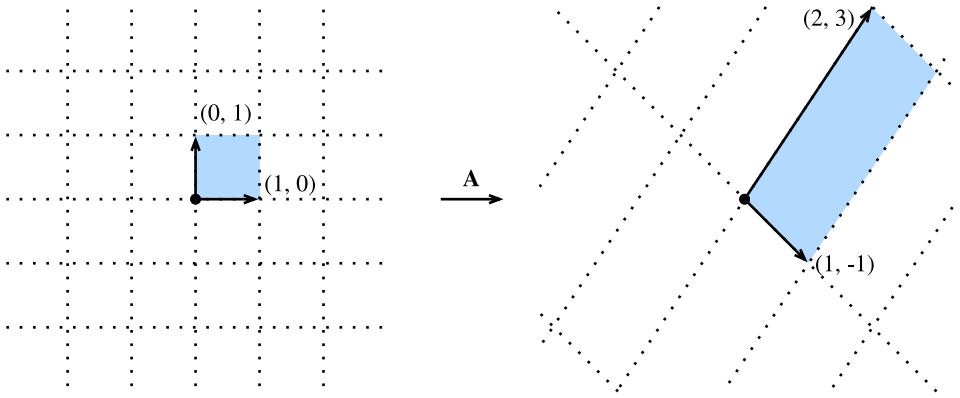


Fig. 17.1.9: The matrix $\mathbf{A}$ again distorting the grid. This time, I want to draw particular attention to what happens to the highlighted square.

Look at the highlighted square. This is a square with edges given by $(0, 1)$ and $(1, 0)$ and thus it has area one. After $\mathbf{A}$ transforms this square, we see that it becomes a parallelogram. There is no reason this parallelogram should have the same area that we started with, and indeed in the specific case shown here of

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix}, \tag{17.1.27}$$

it is an exercise in coordinate geometry to compute the area of this parallelogram and obtain that the area is 5.

In general, if we have a matrix

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{17.1.28}$$

we can see with some computation that the area of the resulting parallelogram is $ad - bc$. This area is referred to as the *determinant*.

Let's check this quickly with some example code.

```
import numpy as np
np.linalg.det(np.array([[1, -1], [2, 3]]))
```

```
5.000000000000001
```

The eagle-eyed amongst us will notice that this expression can be zero or even negative. For the negative term, this is a matter of convention taken generally in mathematics: if the matrix flips the figure, we say the area is negated. Let's see now that when the determinant is zero, we learn more.

Let's consider

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{17.1.29}$$

If we compute the determinant of this matrix, we get $2 \cdot (-2) - 4 \cdot (-1) = 0$. Given our understanding above, this makes sense. $\mathbf{B}$ compresses the square from the original image down to a line segment, which has zero area. And indeed, being compressed into a lower dimensional space is the only way to have zero area after the transformation. Thus we see the following result is true: a matrix A is invertible if and only if the determinant is not equal to zero.

As a final comment, imagine that we have any figure drawn on the plane. Thinking like computer scientists, we can decompose that figure into a collection of little squares so that the area of the figure is in essence just the number of squares in the decomposition. If we now transform that figure by a matrix, we send each of these squares to parallelograms, each one of which has area given by the determinant. We see that for any figure, the determinant gives the (signed) number that a matrix scales the area of any figure.

Computing determinants for larger matrices can be laborious, but the intuition is the same. The determinant remains the factor that $n \times n$ matrices scale n-dimensional volumes.

17.1.9 Tensors and Common Linear Algebra Operations

In Section 2.3 the concept of tensors was introduced. In this section, we will dive more deeply into tensor contractions (the tensor equivalent of matrix multiplication), and see how it can provide a unified view on a number of matrix and vector operations.

With matrices and vectors we knew how to multiply them to transform data. We need to have a similar definition for tensors if they are to be useful to us. Think about matrix multiplication:

$$\mathbf{C} = \mathbf{AB}, \tag{17.1.30}$$

or equivalently

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{17.1.31}$$

This pattern is one we can repeat for tensors. For tensors, there is no one case of what to sum over that can be universally chosen, so we need specify exactly which indices we want to sum over. For instance we could consider

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \tag{17.1.32}$$

Such a transformation is called a *tensor contraction*. It can represent a far more flexible family of transformations that matrix multiplication alone.

As a often-used notational simplification, we can notice that the sum is over exactly those indices that occur more than once in the expression, thus people often work with *Einstein notation*, where the summation is implicitly taken over all repeated indices. This gives the compact expression:

$$y_{il} = x_{ijkl} a_{jk}. \tag{17.1.33}$$

Common Examples from Linear Algebra

Let's see how many of the linear algebraic definitions we have seen before can be expressed in this compressed tensor notation:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$
- $(\mathbf{A}\mathbf{v})_i = \sum_j a_{ij} v_j$
- $(\mathbf{A}\mathbf{B})_{ik} = \sum_j a_{ij} b_{jk}$
- $\text{tr}(\mathbf{A}) = \sum_i a_{ii}$

In this way, we can replace a myriad of specialized notations with short tensor expressions.

Expressing in Code

Tensors may flexibly be operated on in code as well. As seen in Section 2.3, we can create tensors as is shown below.

```
# Define tensors
B = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = np.array([[1, 2], [3, 4]])
v = np.array([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
((2, 2), (2, 2, 3), (2,))
```

Einstein summation has been implemented directly via `np.einsum`. The indices that occurs in the Einstein summation can be passed as a string, followed by the tensors that are being acted upon. For instance, to implement matrix multiplication, we can consider the Einstein summation seen above ($\mathbf{Av} = a_{ij}v_j$) and strip out the indices themselves to get the implementation:

```
# Reimplement matrix multiplication
np.einsum("ij, j -> i", A, v), A.dot(v)
```

```
(array([ 5, 11]), array([ 5, 11]))
```

This is a highly flexible notation. For instance if we want to compute what would be traditionally written as

$$c_{kl} = \sum_{ij} \mathbf{B}_{ijk}\mathbf{A}_{il}v_j. \tag{17.1.34}$$

it can be implemented via Einstein summation as:

```
np.einsum("ijk, il, j -> kl", B, A, v)
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

This notation is readable and efficient for humans, however bulky if for whatever reason we need to generate a tensor contraction programmatically. For this reason, `einsum` provides an alternative notation by providing integer indices for each tensor. For example, the same tensor contraction can also be written as:

```
np.einsum(B, [0, 1, 2], A, [0, 3], v, [1], [2, 3])
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

Either notation allows for concise and efficient representation of tensor contractions in code.

Summary

- Vectors can be interpreted geometrically as either points or directions in space.
- Dot products define the notion of angle to arbitrarily high-dimensional spaces.
- Hyperplanes are high-dimensional generalizations of lines and planes. They can be used to define decision planes that are often used as the last step in a classification task.
- Matrix multiplication can be geometrically interpreted as uniform distortions of the underlying coordinates. They represent a very restricted, but mathematically clean, way to transform vectors.
- Linear dependence is a way to tell when a collection of vectors are in a lower dimensional space than we would expect (say you have 3 vectors living in a 2-dimensional space). The rank of a matrix is the size of the largest subset of its columns that are linearly independent.

- When a matrix's inverse is defined, matrix inversion allows us to find another matrix that undoes the action of the first. Matrix inversion is useful in theory, but requires care in practice owing to numerical instability.
- Determinants allow us to measure how much a matrix expands or contracts a space. A nonzero determinant implies an invertible (non-singular) matrix and a zero-valued determinant means that the matrix is non-invertible (singular).
- Tensor contractions and Einstein summation provide for a neat and clean notation for expressing many of the computations that are seen in machine learning.

Exercises

1. What is the angle between
$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix}? \tag{17.1.35}$$
2. True or false: $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ are inverses of one another?
3. Suppose that we draw a shape in the plane with area 100m^2. What is the area after transforming the figure by the matrix
$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{17.1.36}$$
4. Which of the following sets of vectors are linearly independent?
   - $\left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix} \right\}$
   - $\left\{ \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$
   - $\left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\}$
5. Suppose that you have a matrix written as $A = \begin{bmatrix} c \\ d \end{bmatrix} \cdot \begin{bmatrix} a & b \end{bmatrix}$ for some choice of values a, b, c, and d. True or false: the determinant of such a matrix is always 0?
6. The vectors $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are orthogonal. What is the condition on a matrix A so that Ae_1 and Ae_2 are orthogonal?
7. How can you write $\text{tr}(\mathbf{A}^4)$ in Einstein notation for an arbitrary matrix A?

17.2 Eigendecompositions

Eigenvalues are often one of the most useful notions we will encounter when studying linear algebra, however, as a beginner, it is easy to overlook their importance. Below, we introduce eigendecomposition and try to convey some sense of just why it is so important.

Suppose that we have a matrix A with the following entries:

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}. \tag{17.2.1}$$

If we apply A to any vector $\mathbf{v} = [x, y]^\top$, we obtain a vector $\mathbf{v}A = [2x, -y]^\top$. This has an intuitive interpretation: stretch the vector to be twice as wide in the x-direction, and then flip it in the y-direction.

However, there are *some* vectors for which something remains unchanged. Namely $[1, 0]^\top$ gets sent to $[2, 0]^\top$ and $[0, 1]^\top$ gets sent to $[0, -1]^\top$. These vectors are still in the same line, and the only modification is that the matrix stretches them by a factor of 2 and -1 respectively. We call such vectors *eigenvectors* and the factor they are stretched by *eigenvalues*.

In general, if we can find a number λ and a vector $\mathbf{v}$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{17.2.2}$$

We say that $\mathbf{v}$ is an eigenvector for A and λ is an eigenvalue.

17.2.1 Finding Eigenvalues

Let's figure out how to find them.
By subtracting off the $\lambda\vec{v}$ from both sides, and then factoring out the vector, we see the above is equivalent to:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \tag{17.2.3}$$

For (17.2.3) to happen, we see that $(\mathbf{A} - \lambda\mathbf{I})$ must compress some direction down to zero, hence it is not invertible, and thus the determinant is zero. Thus, we can find the *eigenvalues* by finding for what λ is $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$. Once we find the eigenvalues, we can solve $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ to find the associated *eigenvector(s)*.

An Example

Let's see this with a more challenging matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{17.2.4}$$

If we consider $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$, we see this is equivalent to the polynomial equation $0 = (2-\lambda)(3-\lambda) - 2 = (4-\lambda)(1-\lambda)$. Thus, two eigenvalues are 4 and 1. To find the associated vectors, we then need to solve

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ and } \begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \tag{17.2.5}$$

We can solve this with the vectors $[1, -1]^\top$ and $[1, 2]^\top$ respectively.

We can check this in code using the built-in `numpy.linalg.eig` routine.

```
%matplotlib inline
import d2l
from IPython import display
import numpy as np

np.linalg.eig(np.array([[2, 1], [2, 3]]))
```

```
(array([1., 4.]), array([[-0.70710678, -0.4472136 ],
        [ 0.70710678, -0.89442719]]))
```

Note that `numpy` normalizes the eigenvectors to be of length one, whereas we took ours to be of arbitrary length. Additionally, the choice of sign is arbitrary. However, the vectors computed are parallel to the ones we found by hand with the same eigenvalues.

17.2.2 Decomposing Matrices

Let's continue the previous example one step further. Let

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{17.2.6}$$

be the matrix where the columns are the eigenvectors of the matrix $\mathbf{A}$. Let

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{17.2.7}$$

be the matrix with the associated eigenvalues on the diagonal. Then the definition of eigenvalues and eigenvectors tells us that

$$\mathbf{A}\mathbf{W} = \mathbf{W}\mathbf{\Sigma}. \tag{17.2.8}$$

The matrix W is invertible, so we may multiply both sides by W^{-1} on the right, we see that we may write

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1}. \tag{17.2.9}$$

In the next section we will see some nice consequences of this, but for now we need only know that such a decomposition will exist as long as we can find a full collection of linearly independent eigenvectors (so that W is invertible).

17.2.3 Operations on Eigendecompositions

One nice thing about eigendecompositions (17.2.9) is that we can write many operations we usually encounter cleanly in terms of the eigendecomposition.
As a first example, consider:

$$\mathbf{A}^n = \overbrace{\mathbf{A}\cdots\mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1})\cdots(\mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1})}^{n \text{ times}} = \mathbf{W}\overbrace{\mathbf{\Sigma}\cdots\mathbf{\Sigma}}^{n \text{ times}}\mathbf{W}^{-1} = \mathbf{W}\mathbf{\Sigma}^n\mathbf{W}^{-1}. \tag{17.2.10}$$

This tells us that for any positive power of a matrix, the eigendecomposition is obtained by just raising the eigenvalues to the same power. The same can be shown for negative powers, so if we want to invert a matrix we need only consider

$$\mathbf{A}^{-1} = \mathbf{W}\mathbf{\Sigma}^{-1}\mathbf{W}^{-1}, \tag{17.2.11}$$

or in other words, just invert each eigenvalue. This will work as long as each eigenvalue is non-zero, so we see that invertible is the same as having no zero eigenvalues.

Indeed, additional work can show that if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of a matrix, then the determinant of that matrix is

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{17.2.12}$$

or the product of all the eigenvalues. This makes sense intuitively because whatever stretching $\mathbf{W}$ does, W^{-1} undoes it, so in the end the only stretching that happens is by multiplication by the diagonal matrix $\mathbf{\Sigma}$, which stretches volumes by the product of the diagonal elements.

Finally, recall that the rank was the maximum number of linearly independent columns of your matrix. By examining the eigendecomposition closely, we can see that the rank is the same as the number of non-zero eigenvalues of $\mathbf{A}$.

The examples could continue, but hopefully the point is clear: eigendecompositions can simplify many linear-algebraic computations and are a fundamental operation underlying many numerical algorithms and much of the analysis that we do in linear algebra.

17.2.4 Eigendecompositions of Symmetric Matrices

It is not always possible to find enough linearly independent eigenvectors for the above process to work. For instance the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{17.2.13}$$

has only a single eigenvector, namely $(0, 1)$. To handle such matrices, we require more advanced techniques than we can cover (such as the Jordan Normal Form, or Singular Value Decomposition). We will often need to restrict our attention to those matrices where we can guarantee the existence of a full set of eigenvectors.

The most commonly encountered family are the *symmetric matrices*, which are those matrices where $\mathbf{A} = \mathbf{A}^\top$. In this case, we may take W to be an *orthogonal matrix*—a matrix whose columns are all length one vectors that are at right angles to one another, where $\mathbf{W}^\top = \mathbf{W}^{-1}$—and all the eigenvalues will be real.

Thus, in this special case, we can write (17.2.9) as

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^\top. \tag{17.2.14}$$

17.2.5 Gershgorin Circle Theorem

Eigenvalues are often difficult to reason with intuitively. If presented an arbitrary matrix, there is little that can be said about what the eigenvalues are without computing them. There is, however, one theorem that can make it easy to approximate well if the largest values are on the diagonal.

Let $\mathbf{A} = (a_{ij})$ be any square matrix ($n \times n$). We will define $r_i = \sum_{j \neq i} |a_{ij}|$. Let $\mathcal{D}_i$ represent the disc in the complex plane with center a_{ii} radius r_i. Then, every eigenvalue of $\mathbf{A}$ is contained in one of the $\mathcal{D}_i$.

This can be a bit to unpack, so let's look at an example.
Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{17.2.15}$$

We have $r_1 = 0.3$, $r_2 = 0.6$, $r_3 = 0.8$ and $r_4 = 0.9$. The matrix is symmetric, so all eigenvalues are real. This means that all of our eigenvalues will be in one of the ranges of

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{17.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{17.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{17.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{17.2.19}$$

Performing the numerical computation shows that the eigenvalues are approximately 0.99, 2.97, 4.95, 9.08, all comfortably inside the ranges provided.

```
A = np.array([[1.0, 0.1, 0.1, 0.1],
              [0.1, 3.0, 0.2, 0.3],
              [0.1, 0.2, 5.0, 0.5],
              [0.1, 0.3, 0.5, 9.0]])

v, _ = np.linalg.eig(A)
v
```

```
array([9.08033648, 0.99228545, 4.95394089, 2.97343718])
```

In this way, eigenvalues can be approximated, and the approximations will be fairly accurate in the case that the diagonal is significantly larger than all the other elements.

It is a small thing, but with a complex and subtle topic like eigendecomposition, it is good to get any intuitive grasp we can.

17.2.6 A Useful Application: The Growth of Iterated Maps

Now that we understand what eigenvectors are in principle, let's see how they can be used to provide a deep understanding of a problem central to neural network behavior: proper weight initialization.

Eigenvectors as Long Term Behavior

The full mathematical investigation of the initialization of deep neural networks is beyond the scope of the text, but we can see a toy version here to understand how eigenvalues can help us see how these models work. As we know, neural networks operate by interspersing layers of linear transformations with non-linear operations. For simplicity here, we will assume that there is no non-linearity, and that the transformation is a single repeated matrix operation A, so that the output of our model is

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A}\mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \tag{17.2.20}$$

When these models are initialized, A is taken to be a random matrix with Gaussian entries, so let's make one of those. To be concrete, we start with a mean zero, variance one Gaussian distributed 5×5 matrix.

```
np.random.seed(8675309)

k = 5
A = np.random.randn(k, k)
A
```

```
array([[ 0.58902366,  0.73311856, -1.1621888 , -0.55681601, -0.77248843],
       [-0.16822143, -0.41650391, -1.37843129,  0.74925588,  0.17888446],
       [ 0.69401121, -1.9780535 , -0.83381434,  0.56437344,  0.31201299],
       [-0.87334496,  0.15601291, -0.38710108, -0.23920821,  0.88850104],
       [ 1.29385371, -0.76774106,  0.20131613,  0.91800842,  0.38974115]])
```

Behavior on Random Data

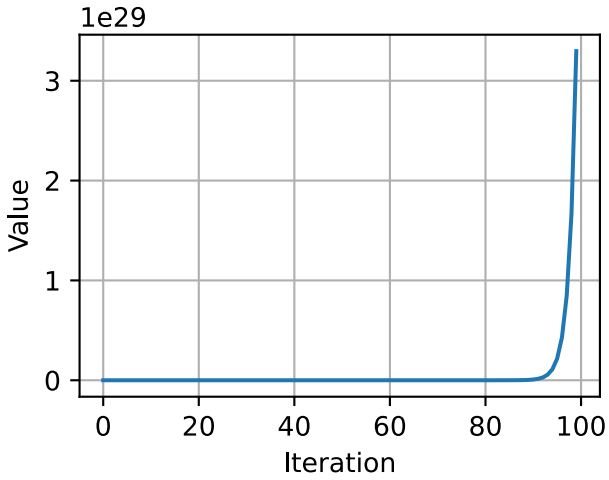
For simplicity in our toy model, we will assume that the data vector we feed in $\mathbf{v}_{in}$ is a random five dimensional Gaussian vector. Let's think about what we want to have happen. For context, lets think of a generic ML problem, where we are trying to turn input data, like an image, into a prediction, like the probability the image is a picture of a cat. If repeated application of $\mathbf{A}$ stretches a random vector out to be very long, then small changes in input will be amplified into large changes in output—tiny modifications of the input image would lead to vastly different predictions. This does not seem right!

On the flip side, if $\mathbf{A}$ shrinks random vectors to be shorter, then after running through many layers, the vector will essentially shrink to nothing, and the output will not depend on the input. This is also clearly not right either!

We need to walk the narrow line between growth and decay to make sure that our output changes depending on our input, but not much!

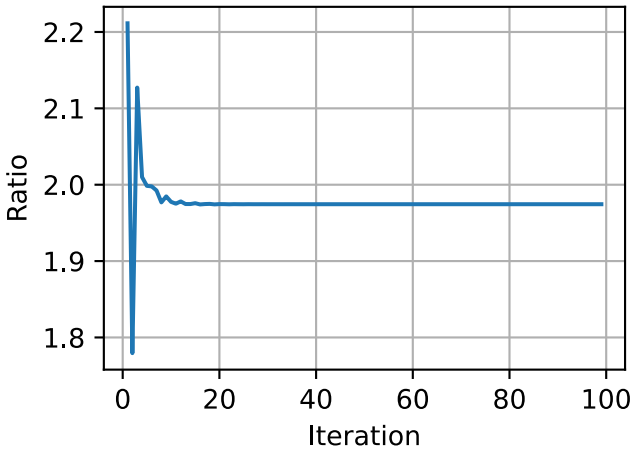
Let's see what happens when we repeatedly multiply our matrix $\mathbf{A}$ against a random input vector, and keep track of the norm.

```
# Calculate the sequence of norms after repeatedly applying A
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```



The norm is growing uncontrollably! Indeed if we take the list of quotients, we will see a pattern.

```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



If we look at the last portion of the above computation, we see that the random vector is stretched by a factor of `1.974459321485[...]`, where the portion at the end shifts a little, but the stretching

factor is stable.

Relating Back to Eigenvectors

We have seen that eigenvectors and eigenvalues correspond to the amount something is stretched, but that was for specific vectors, and specific stretches. Let's take a look at what they are for **A**. A bit of a caveat here: it turns out that to see them all, we will need to go to complex numbers. You can think of these as stretches and rotations. By taking the norm of the complex number (square root of the sums of squares of real and imaginary parts) we can measure that stretching factor. Let's also sort them.

```
# Compute the eigenvalues
eigs = np.linalg.eigvals(A).tolist()
norm_eigs = [np.absolute(x) for x in eigs]
norm_eigs.sort()
"Norms of eigenvalues: {}".format(norm_eigs)
```

```
'Norms of eigenvalues: [0.8786205280381857, 1.2757952665062624, 1.4983381517710659, 1.
↪4983381517710659, 1.974459321485074]'
```

An Observation

We see something a bit unexpected happening here: that number we identified before for the long term stretching of our matrix **A** applied to a random vector is *exactly* (accurate to thirteen decimal places!) the largest eigenvalue of **A**. This is clearly not a coincidence!

But, if we now think about what is happening geometrically, this starts to make sense. Consider a random vector. This random vector points a little in every direction, so in particular, it points at least a little bit in the same direction as the eigenvector of **A** associated with the largest eigenvalue. This is so important that it is called the *principle eigenvalue* and *principle eigenvector*. After applying **A**, our random vector gets stretched in every possible direction, as is associated with every possible eigenvector, but it is stretched most of all in the direction associated with this principle eigenvector. What this means is that after apply in A, our random vector is longer, and points in a direction closer to being aligned with the principle eigenvector. After applying the matrix many times, the alignment with the principle eigenvector becomes closer and closer until, for all practical purposes, our random vector has been transformed into the principle eigenvector! Indeed this algorithm is the basis for what is known as the *power iteration* for finding the largest eigenvalue and eigenvector of a matrix. For details see, for example, (VanLoan & Golub, 1983).

Fixing the Normalization

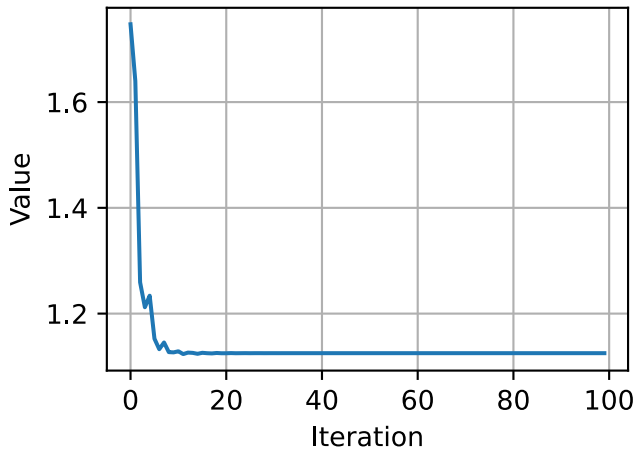
Now, from above discussions, we concluded that we do not want a random vector to be stretched or squished at all, we would like random vectors to stay about the same size throughout the entire process. To do so, we now rescale our matrix by this principle eigenvalue so that the largest eigenvalue is instead now just one. Let's see what happens in this case.

```
# Rescale the matrix A
A /= norm_eigs[-1]
```

(continues on next page)

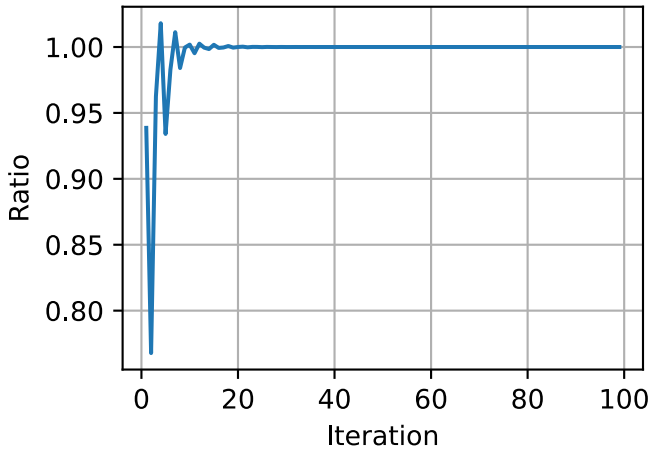
(continued from previous page)

```
# Do the same experiment again
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```



We can also plot the ration between consecutive norms as before and see that indeed it stabilizes.

```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



17.2.7 Conclusions

We now see exactly what we hoped for! After normalizing the matrices by the principle eigenvalue, we see that the random data does not explode as before, but rather eventually equilibrates to a specific value. It would be nice to be able to do these things from first principles, and it turns out that if we look deeply at the mathematics of it, we can see that the largest eigenvalue of a large random matrix with independent mean zero, variance one Gaussian entries is on average about $\sqrt{n}$, or in our case $\sqrt{5} \approx 2.2$, due to a fascinating fact known as the *circular law* (Ginibre, 1965). The relationship between the eigenvalues (and a related object called singular values) of random matrices has been shown to have deep connections to proper initialization of neural networks as was discussed in (Pennington et al., 2017) and subsequent works.

Summary

- Eigenvectors are vectors which are stretched by a matrix without changing direction.
- Eigenvalues are the amount that the eigenvectors are stretched by the application of the matrix.
- The eigendecomposition of a matrix can allow for many operations to be reduced to operations on the eigenvalues.
- The Gershgorin Circle Theorem can provide approximate values for the eigenvalues of a matrix.
- The behavior of iterated matrix powers depends primarily on the size of the largest eigenvalue. This understanding has many applications in the theory of neural network initialization.

Exercises

1. What are the eigenvalues and eigenvectors of
$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}? \tag{17.2.21}$$
2. What are the eigenvalues and eigenvectors of the following matrix, and what is strange about this example compared to the previous one?
$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}. \tag{17.2.22}$$
3. Without computing the eigenvalues, is it possible that the smallest eigenvalue of the following matrix is less that 0.5? *Note*: this problem can be done in your head.
$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{17.2.23}$$

17.3 Single Variable Calculus

In Section 2.4, we saw the basic elements of differential calculus. This section takes a deeper dive into the fundamentals of calculus and how we can understand and apply it in the context of machine learning.

17.3.1 Differential Calculus

Differential calculus is fundamentally the study of how functions behave under small changes. To see why this is so core to deep learning, let's consider an example.

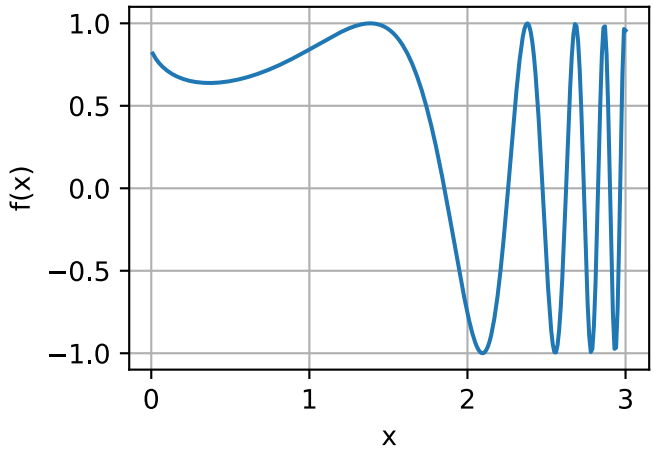
Suppose that we have a deep neural network where the weights are, for convenience, concatenated into a single vector $\mathbf{w} = (w_1, \ldots, w_n)$. Given a training dataset, we consider the loss of our neural network on this dataset, which we will write as $\mathcal{L}(\mathbf{w})$.

This function is extraordinarily complex, encoding the performance of all possible models of the given architecture on this dataset, so it is nearly impossible to tell what set of weights $\mathbf{w}$ will minimize the loss. Thus, in practice, we often start by initializing our weights *randomly*, and then iteratively take small steps in the direction which makes the loss decrease as rapidly as possible.

The question then becomes something that on the surface is no easier: how do we find the direction which makes the weights decrease as quickly as possible? To dig into this, let's first examine the case with only a single weight: $L(\mathbf{w}) = L(x)$ for a single real value x.

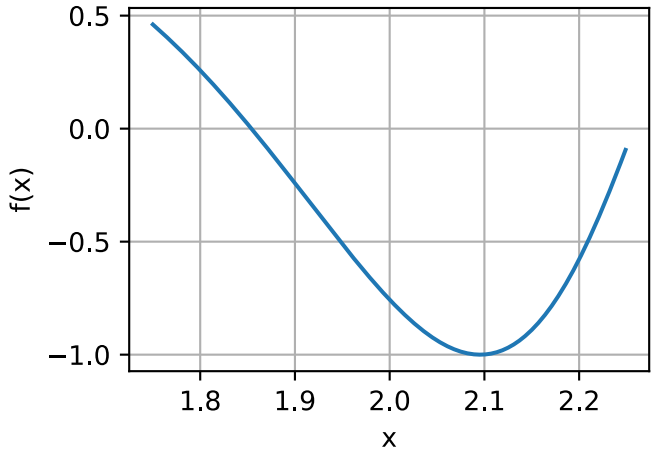
Let's take x and try to understand what happens when we change it by a small amount to $x + \epsilon$. If you wish to be concrete, think a number like $\epsilon = 0.0000001$. To help us visualize what happens, let's graph an example function, $f(x) = \sin(x^x)$, over the $[0, 3]$.

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot a function in a normal range
x_big = np.arange(0.01, 3.01, 0.01)
ys = np.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```



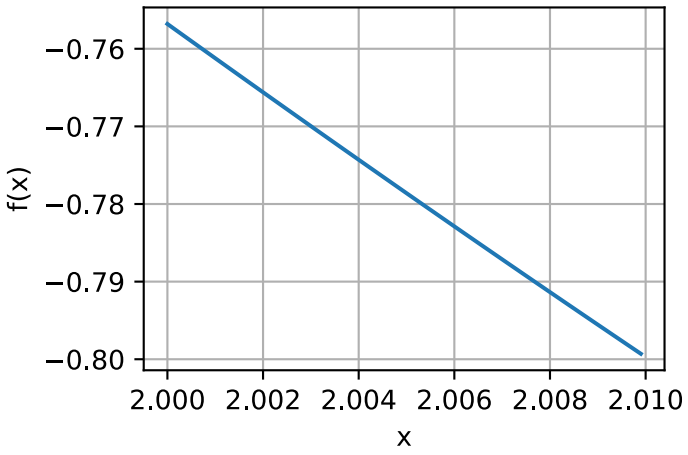
At this large scale, the function's behavior is not simple. However, if we reduce our range to something smaller like $[1.75, 2.25]$, we see that the graph becomes much simpler.

```
# Plot a the same function in a tiny range
x_med = np.arange(1.75, 2.25, 0.001)
ys = np.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



Taking this to an extreme, if we zoom into a tiny segment, the behavior becomes far simpler: it is just a straight line.

```
# Plot a the same function in a tiny range
x_small = np.arange(2.0, 2.01, 0.0001)
ys = np.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



This is the key observation of single variable calculus: the behavior of familiar functions can be modeled by a line in a small enough range. This means that for most functions, it is reasonable to expect that as we shift the x value of the function by a little bit, the output $f(x)$ will also be shifted by a little bit. The only question we need to answer is, "How large is the change in the output compared to the change in the input? Is it half as large? Twice as large?"

Thus, we can consider the ratio of the change in the output of a function for a small change in the input of the function. We can write this formally as

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{17.3.1}$$

This is already enough to start to play around with in code. For instance, suppose that we know that $L(x) = x^2 + 1701(x-4)^3$, then we can see how large this value is at the point $x = 4$ as follows.

```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3

# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print("epsilon = {:.5f} -> {:.5f}".format(
        epsilon, (L(4+epsilon) - L(4)) / epsilon))
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

Now, if we are observant, we will notice that the output of this number is suspiciously close to 8. Indeed, if we decrease ϵ, we will see value becomes progressively closer to 8. Thus we may conclude, correctly, that the value we seek (the degree a change in the input changes the output) should be 8 at the point $x = 4$. The way that a mathematician encodes this fact is

$$\lim_{\epsilon \to 0} \frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{17.3.2}$$

As a bit of a historical digression: in the first few decades of neural network research, scientists used this algorithm (the *method of finite differences*) to evaluate how a loss function changed under small perturbation: just change the weights and see how the loss changed. This is computationally inefficient, requiring two evaluations of the loss function to see how a single change of one variable influenced the loss. If we tried to do this with even a paltry few thousand parameters, it would require several thousand evaluations of the network over the entire dataset! It was not solved until 1986 that the *backpropagation algorithm* introduced in (Rumelhart et al., 1988) provided a way to calculate how *any* change of the weights together would change the loss in the same computation time as a single prediction of the network over the dataset.

Back in our example, this value 8 is different for different values of x, so it makes sense to define it as a function of x. More formally, this value dependent rate of change is referred to as the *derivative* which is written as

$$\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{17.3.3}$$

Different texts will use different notations for the derivative. For instance, all of the below notations indicate the same thing:

$$\frac{df}{dx} = \frac{d}{dx} f = f' = \nabla_x f = D_x f = f_x. \tag{17.3.4}$$

Most authors will pick a single notation and stick with it, however even that is not guaranteed. It is best to be familiar with all of these. We will use the notation $\frac{df}{dx}$ throughout this text, unless

we want to take the derivative of a complex expression, in which case we will use $\frac{d}{dx}f$ to write expressions like

$$\frac{d}{dx}\left[x^4 + \cos\left(\frac{x^2+1}{2x-1}\right)\right]. \tag{17.3.5}$$

Often times, it is intuitively useful to unravel the definition of derivative (17.3.3) again to see how a function changes when we make a small change of x:

$$\begin{aligned}\frac{df}{dx}(x) = \lim_{\epsilon\to 0}\frac{f(x+\epsilon)-f(x)}{\epsilon} &\implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon)-f(x)}{\epsilon}\\ &\implies \epsilon\frac{df}{dx}(x) \approx f(x+\epsilon) - f(x)\\ &\implies f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x).\end{aligned} \tag{17.3.6}$$

The last equation is worth explicitly calling out. It tells us that if you take any function and change the input by a small amount, the output would change by that small amount scaled by the derivative.

In this way, we can understand the derivative as the scaling factor that tells us how large of change we get in the output from a change in the input.

17.3.2 Rules of Calculus

We now turn to the task of understanding how to compute the derivative of an explicit function. A full formal treatment of calculus would derive everything from first principles. We will not indulge in this temptation here, but rather provide an understanding of the common rules encountered.

Common Derivatives

As was seen in Section 2.4, when computing derivatives one can often times use a series of rules to reduce the computation to a few core functions. We repeat them here for ease of reference.

- **Derivative of constants.** $\frac{d}{dx}c = 0$.
- **Derivative of linear functions.** $\frac{d}{dx}(ax) = a$.
- **Power rule.** $\frac{d}{dx}x^n = nx^{n-1}$.
- **Derivative of exponentials.** $\frac{d}{dx}e^x = e^x$.
- **Derivative of the logarithm.** $\frac{d}{dx}\log(x) = \frac{1}{x}$.

Derivative Rules

If every derivative needed to be separately computed and stored in a table, differential calculus would be near impossible. It is a gift of mathematics that we can generalize the above derivatives and compute more complex derivatives like finding the derivative of $f(x) = \log\left(1 + (x-1)^{10}\right)$. As was mentioned in Section 2.4, the key to doing so is to codify what happens when we take functions and combine them in various ways, most importantly: sums, products, and compositions.

- **Sum rule.** $\frac{d}{dx}\left(g(x) + h(x)\right) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$.

- **Product rule.** $\frac{d}{dx}\left(g(x)\cdot h(x)\right) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$.
- **Chain rule.** $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x))\cdot\frac{dh}{dx}(x)$.

Let's see how we may use (17.3.6) to understand these rules. For the sum rule, consider following chain of reasoning:

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) + h(x+\epsilon) \\ &\approx g(x) + \epsilon\frac{dg}{dx}(x) + h(x) + \epsilon\frac{dh}{dx}(x) \\ &= g(x) + h(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right) \\ &= f(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right). \end{aligned} \tag{17.3.7}$$

By comparing this result with the fact that $f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x)$, we see that $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ as desired. The intuition here is: when we change the input x, g and h jointly contribute to the change of the output by $\frac{dg}{dx}(x)$ and $\frac{dh}{dx}(x)$.

The product is more subtle, and will require a new observation about how to work with these expressions. We will begin as before using (17.3.6):

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon)\cdot h(x+\epsilon) \\ &\approx \left(g(x) + \epsilon\frac{dg}{dx}(x)\right)\cdot\left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &= g(x)\cdot h(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x) \\ &= f(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x). \end{aligned} \tag{17.3.8}$$

This resembles the computation done above, and indeed we see our answer ($\frac{df}{dx}(x) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$) sitting next to ϵ, but there is the issue of that term of size ϵ^2. We will refer to this as a *higher-order term*, since the power of ϵ^2 is higher than the power of ϵ^1. We will see in a later section that we will sometimes want to keep track of these, however for now observe that if $\epsilon = 0.0000001$, then $\epsilon^2 = 0.0000000000001$, which is vastly smaller. As we send $\epsilon \to 0$, we may safely ignore the higher order terms. As a general convention in this appendix, we will use "$\approx$" to denote that the two terms are equal up to higher order terms. However, if we wish to be more formal we may examine the difference quotient

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon\frac{dg}{dx}(x)\frac{dh}{dx}(x), \tag{17.3.9}$$

and see that as we send $\epsilon \to 0$, the right hand term goes to zero as well.

Finally, with the chain rule, we can again progress as before using (17.3.6) and see that

$$\begin{aligned} f(x+\epsilon) &= g(h(x+\epsilon)) \\ &\approx g\left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &\approx g(h(x)) + \epsilon\frac{dh}{dx}(x)\frac{dg}{dh}(h(x)) \\ &= f(x) + \epsilon\frac{dg}{dh}(h(x))\frac{dh}{dx}(x), \end{aligned} \tag{17.3.10}$$

where in the second line we view the function g as having its input ($h(x)$) shifted by the tiny quantity $\epsilon \frac{dh}{dx}(x)$.

These rule provide us with a flexible set of tools to compute essentially any expression desired. For instance,

$$\begin{aligned}
\frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] &= \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right] \\
&= \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1]+\frac{d}{dx}[(x-1)^{10}]\right) \\
&= \left(1+(x-1)^{10}\right)^{-1}\left(0+10(x-1)^{9}\frac{d}{dx}[x-1]\right) \\
&= 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^{9} \\
&= \frac{10(x-1)^{9}}{1+(x-1)^{10}}.
\end{aligned} \tag{17.3.11}$$

Where each line has used the following rules:

1. The chain rule and derivative of logarithm.
2. The sum rule.
3. The derivative of constants, chain rule, and power rule.
4. The sum rule, derivative of linear functions, derivative of constants.

Two things should be clear after doing this example:

1. Any function we can write down using sums, products, constants, powers, exponentials, and logarithms can have its derivate computed mechanically by following these rules.
2. Having a human follow these rules can be tedious and error prone!

Thankfully, these two facts together hint towards a way forward: this is a perfect candidate for mechanization! Indeed backpropagation, which we will revisit later in this section, is exactly that.

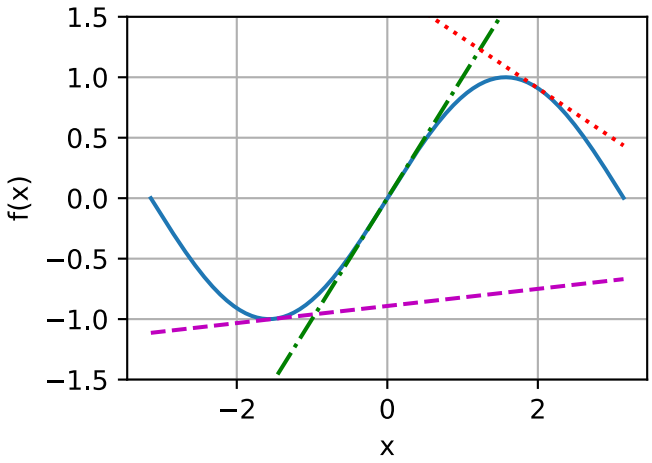
Linear Approximation

When working with derivatives, it is often useful to geometrically interpret the approximation used above. In particular, note that the equation

$$f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \tag{17.3.12}$$

approximates the value of f by a line which passes through the point $(x, f(x))$ and has slope $\frac{df}{dx}(x)$. In this way we say that the derivative gives a linear approximation to the function f, as illustrated below:

```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Higher Order Derivatives

Let's now do something that may on the surface seem strange. Take a function f and compute the derivative $\frac{df}{dx}$. This gives us the rate of change of f at any point.

However, the derivative, $\frac{df}{dx}$, can be viewed as a function itself, so nothing stops us from computing the derivative of $\frac{df}{dx}$ to get $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$. We will call this the second derivative of f. This function is the rate of change of the rate of change of f, or in other words, how the rate of change is changing. We may apply the derivative any number of times to obtain what is called the n-th derivative. To keep the notation clean, we will denote the n-th derivative as

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{17.3.13}$$

Let's try to understand *why* this is a useful notion. Below, we visualize $f^{(2)}(x)$, $f^{(1)}(x)$, and $f(x)$.

First, consider the case that the second derivative $f^{(2)}(x)$ is a positive constant. This means that the slope of the first derivative is positive. As a result, the first derivative $f^{(1)}(x)$ may start out negative, becomes zero at a point, and then becomes positive in the end. This tells us the slope of our original function f and therefore, the function f itself decreases, flattens out, then increases. In other words, the function f curves up, and has a single minimum as is shown in Fig. 17.3.1.

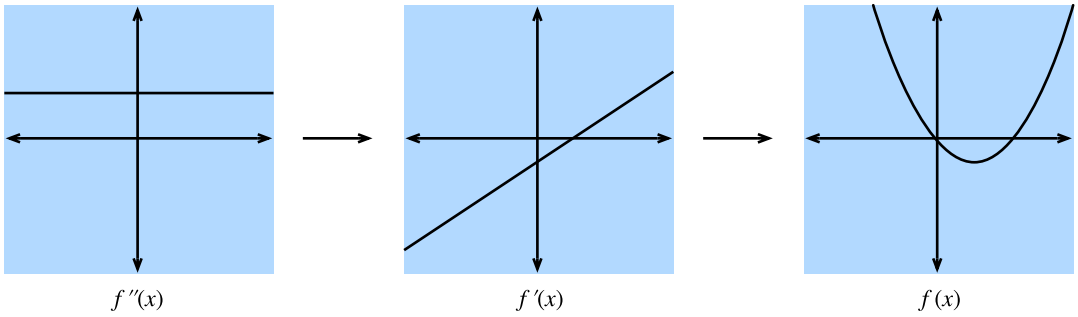


Fig. 17.3.1: If we assume the second derivative is a positive constant, then the fist derivative in increasing, which implies the function itself has a minimum.

Second, if the second derivative is a negative constant, that means that the first derivative is decreasing. This implies the first derivative may start out positive, becomes zero at a point, and then

becomes negative. Hence, the function f itself increases, flattens out, then decreases. In other words, the function f curves down, and has a single maximum as is shown in Fig. 17.3.2.

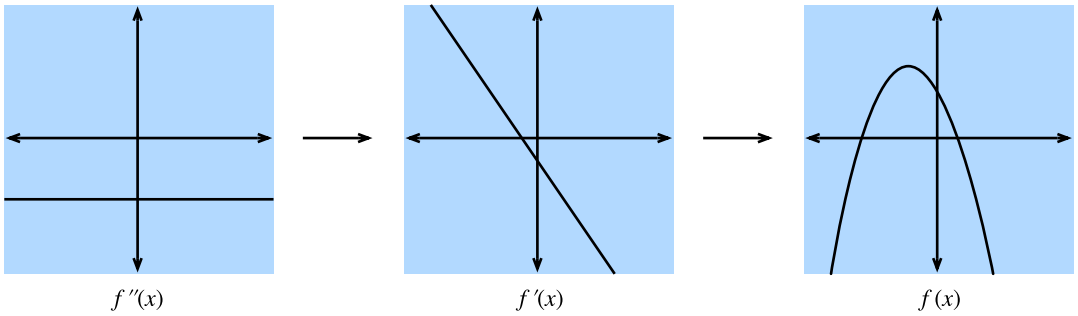


Fig. 17.3.2: If we assume the second derivative is a negative constant, then the fist derivative in decreasing, which implies the function itself has a maximum.

Third, if the second derivative is a always zero, then the first derivative will never change—it is constant! This means that f increases (or decreases) at a fixed rate, and f is itself a straight line as is shown in Fig. 17.3.3.

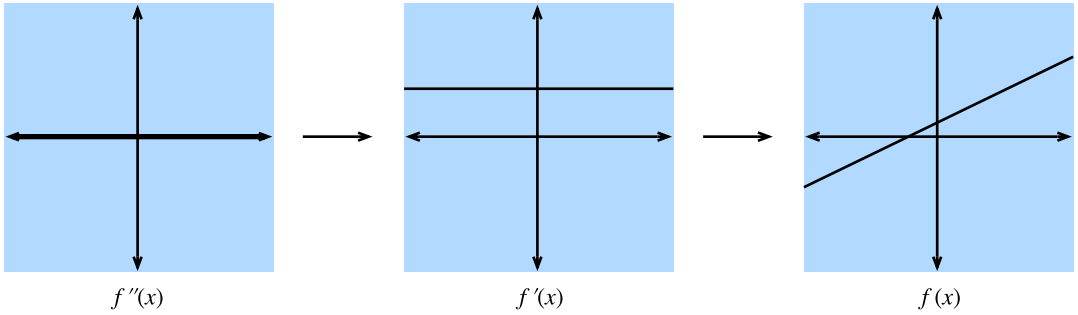


Fig. 17.3.3: If we assume the second derivative is zero, then the fist derivative is constant, which implies the function itself is a straight line.

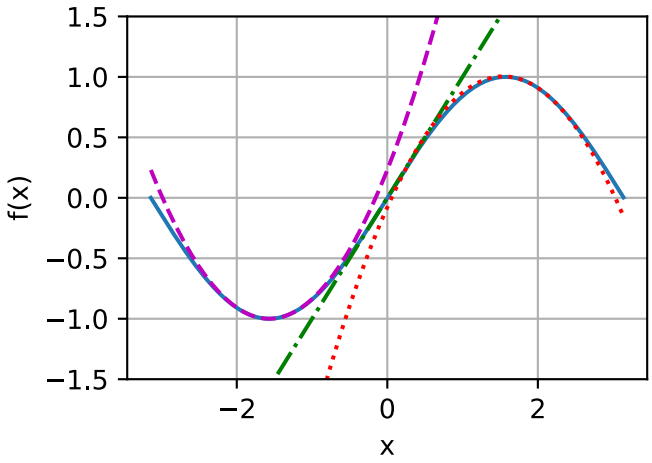
To summarize, the second derivative can be interpreted as describing the way that the function f curves. A positive second derivative leads to a upwards curve, while a negative second derivative means that f curves downwards, and a zero second derivative means that f does not curve at all.

Let's take this one step further. Consider the function $g(x) = ax^2 + bx + c$. We can then compute that

$$\begin{aligned} \frac{dg}{dx}(x) &= 2ax + b \\ \frac{d^2g}{dx^2}(x) &= 2a. \end{aligned} \tag{17.3.14}$$

If we have some original function $f(x)$ in mind, we may compute the first two derivatives and find the values for a, b, and c that make them match this computation. Similarly to the previous section where we saw that the first derivative gave the best approximation with a straight line, this construction provides the best approximation by a quadratic. Let's visualize this for $f(x) = \sin(x)$.

```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0) -
                              (xs - x0)**2 * np.sin(x0) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



We will extend this idea to the idea of a *Taylor series* in the next section.

Taylor Series

The *Taylor series* provides a method to approximate the function $f(x)$ if we are given values for the first n derivatives at a point x_0, i.e., $\{f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0)\}$. The idea will be to find a degree n polynomial that matches all the given derivatives at x_0.

We saw the case of $n = 2$ in the previous section and a little algebra shows this is

$$f(x) \approx \frac{1}{2}\frac{d^2 f}{dx^2}(x_0)(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{17.3.15}$$

As we can see above, the denominator of 2 is there to cancel out the 2 we get when we take two derivatives of x^2, while the other terms are all zero. Same logic applies for the first derivative and the value itself.

If we push the logic further to $n = 3$, we will conclude that

$$f(x) \approx \frac{\frac{d^3 f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2 f}{dx^2}(x_0)}{2}(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{17.3.16}$$

where the $6 = 3 \times 2 = 3!$ comes from the constant we get in front if we take three derivatives of x^3.

Furthermore, we can get a degree n polynomial by

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i. \tag{17.3.17}$$

where the notation

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{17.3.18}$$

Indeed, $P_n(x)$ can be viewed as the best n-th degree polynomial approximation to our function $f(x)$.

While we are not going to dive all the way into the error of the above approximations, it is worth mentioning the the infinite limit. In this case, for well behaved functions (known as real analytic functions) like $\cos(x)$ or e^x, we can write out the infinite number of terms and approximate the exactly same function

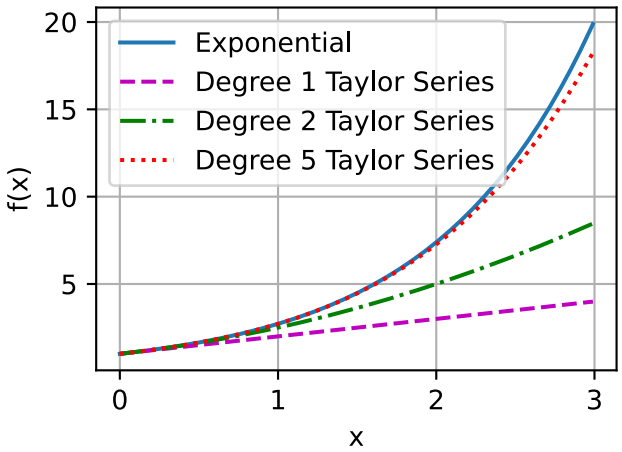
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{17.3.19}$$

Take $f(x) = e^x$ as am example. Since e^x is its own derivative, we know that $f^{(n)}(x) = e^x$. Therefore, e^x can be reconstructed by taking the Taylor series at $x_0 = 0$, i.e.,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots. \tag{17.3.20}$$

Let's see how this works in code and observe how increasing the degree of the Taylor approximation brings us closer to the desired function e^x.

```
# Compute the exponential function
xs = np.arange(0, 3, 0.01)
ys = np.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```



Taylor series have two primary applications:

1. *Theoretical applications*: Often when we try to understand a too complex function, using Taylor series enables we turn it into a polynomial that we can work with directly.
2. *Numerical applications*: Some functions like e^x or $\cos(x)$ are difficult for machines to compute. They can store tables of values at a fixed precision (and this is often done), but it still leaves open questions like "What is the 1000-th digit of $\cos(1)$?" Taylor series are often helpful to answer such questions.

Summary

- Derivatives can be used to express how functions change when we change the input by a small amount.
- Elementary derivatives can be combined using derivative rules to create arbitrarily complex derivatives.
- Derivatives can be iterated to get second or higher order derivatives. Each increase in order provides more fine grained information on the behavior of the function.
- Using information in the derivatives of a single data point, we can approximate well behaved functions by polynomials obtained from the Taylor series.

Exercises

1. What is the derivative of $x^3 - 4x + 1$?
2. What is the derivative of $\log(\frac{1}{x})$?
3. True or False: If $f'(x) = 0$ then f has a maximum or minimum at x?
4. Where is the minimum of $f(x) = x\log(x)$ for $x \geq 0$ (where we assume that f takes the limiting value of 0 at $f(0)$)?

17.4 Multivariable Calculus

Now that we have a fairly strong understanding of derivatives of a function of a single variable, let's return to our original question where we were considering a loss function of potentially billions of weights.

17.4.1 Higher-Dimensional Differentiation

What Section 17.3 tells us is that if we change a single one of these billions of weights leaving every other one fixed, we know what will happen! This is nothing more than a function of a single variable, so we can write

$$L(w_1 + \epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \tag{17.4.1}$$

We will call the derivative in one variable while fixing the other the *partial derivative*, and we will use the notation $\frac{\partial}{\partial w_1}$ for the derivative in (17.4.1).

Now, let's take this and change w_2 a little bit to $w_2 + \epsilon_2$:

$$\begin{aligned}
L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N) &\approx L(w_1, w_2 + \epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2 + \epsilon_2, \ldots, w_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \epsilon_2 \frac{\partial}{\partial w_2} \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N).
\end{aligned} \tag{17.4.2}$$

We have again used the idea that $\epsilon_1 \epsilon_2$ is a higher order term that we can discard in the same way we could discard ϵ^2 in the previous section, along with what we saw in (17.4.1). By continuing in this manner, we may write that

$$L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \tag{17.4.3}$$

This may look like a mess, but we can make this more familiar by noting that the sum on the right looks exactly like a dot product, so if we let

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}\right]^\top, \tag{17.4.4}$$

then

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{17.4.5}$$

We will call the vector $\nabla_{\mathbf{w}} L$ the *gradient* of L.

Equation (17.4.5) is worth pondering for a moment. It has exactly the format that we encountered in one dimension, just we have converted everything to vectors and dot products. It allows us to tell approximately how the function L will change given any perturbation to the input. As we will see in the next section, this will provide us with an important tool in understanding geometrically how we can learn using information contained in the gradient.

But first, let's see this approximation at work with an example. Suppose that we are working with the function

$$f(x, y) = \log(e^x + e^y) \text{ with gradient } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{17.4.6}$$

If we look at a point like $(0, \log(2))$, we see that

$$f(x, y) = \log(3) \text{ with gradient } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{17.4.7}$$

Thus, if we want to approximate f at $(\epsilon_1, \log(2) + \epsilon_2)$, we see that we should have the specific instance of (17.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{17.4.8}$$

We can test this in code to see how good the approximation is.

```
%matplotlib inline
import d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import autograd, np, npx
npx.set_np()

def f(x, y):
    return np.log(np.exp(x) + np.exp(y))
def grad_f(x, y):
    return np.array([np.exp(x) / (np.exp(x) + np.exp(y)),
                     np.exp(y) / (np.exp(x) + np.exp(y))])

epsilon = np.array([0.01, -0.03])
grad_approx = f(0, np.log(2)) + epsilon.dot(grad_f(0, np.log(2)))
true_value = f(0 + epsilon[0], np.log(2) + epsilon[1])
"Approximation: {}, True Value: {}".format(grad_approx, true_value)
```

```
'Approximation: 1.0819457, True Value: 1.0821242'
```

17.4.2 Geometry of Gradients and Gradient Descent

Consider the again (17.4.5):

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{17.4.9}$$

Let's suppose that I want to use this to help minimize our loss L. Let's understand geometrically the algorithm of gradient descent first described in Section 2.5. What we will do is the following:

1. Start with a random choice for the initial parameters $\mathbf{w}$.
2. Find the direction $\mathbf{v}$ that makes L decrease the most rapidly at $\mathbf{w}$.
3. Take a small step in that direction: $\mathbf{w} \to \mathbf{w} + \epsilon\mathbf{v}$.
4. Repeat.

The only thing we do not know exactly how to do is to compute the vector $\mathbf{v}$ in the second step. We will call such a direction the *direction of steepest descent*. Using the geometric understanding of dot products from Section 17.1, we see that we can rewrite (17.4.5) as

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{17.4.10}$$

Note that we have taken our direction to have length one for convenience, and used θ for the angle between $\mathbf{v}$ and $\nabla_{\mathbf{w}} L(\mathbf{w})$. If we want to find the direction that decreases L as rapidly as possible, we want to make this as expression as negative as possible. The only way the direction we pick enters into this equation is through $\cos(\theta)$, and thus we wish to make this cosine as negative as possible. Now, recalling the shape of cosine, we can make this as negative as possible by making $\cos(\theta) = -1$ or equivalently making the angle between the gradient and our chosen direction to be π radians, or equivalently 180 degrees. The only way to achieve this is to head in the exact opposite direction: pick $\mathbf{v}$ to point in the exact opposite direction to $\nabla_{\mathbf{w}} L(\mathbf{w})$!

This brings us to one of the most important mathematical concepts in machine learning: the direction of steepest decent points in the direction of $-\nabla_{\mathbf{w}} L(\mathbf{w})$. Thus our informal algorithm can be rewritten as follows.

1. Start with a random choice for the initial parameters $\mathbf{w}$.
2. Compute $\nabla_{\mathbf{w}} L(\mathbf{w})$.
3. Take a small step in the opposite of that direction: $\mathbf{w} \to \mathbf{w} - \epsilon \nabla_{\mathbf{w}} L(\mathbf{w})$.
4. Repeat.

This basic algorithm has been modified and adapted many ways by many researchers, but the core concept remains the same in all of them. Use the gradient to find the direction that decreases the loss as rapidly as possible, and update the parameters to take a step in that direction.

17.4.3 A Note on Mathematical Optimization

Throughout this book, we focus squarely on numerical optimization techniques for the practical reason that all functions we encounter in the deep learning setting are too complex to minimize explicitly.

However, it is a useful exercise to consider what the geometric understanding we obtained above tells us about optimizing functions directly.

Suppose that we wish to find the value of $\mathbf{x}_0$ which minimizes some function $L(\mathbf{x})$. Let's suppose that moreover someone gives us a value and tells us that it is the value that minimizes L. Is there anything we can check to see if their answer is even plausible?

Again consider (17.4.5):

$$L(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx L(\mathbf{x}_0) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \tag{17.4.11}$$

If the gradient is not zero, we know that we can take a step in the direction $-\epsilon \nabla_{\mathbf{x}} L(\mathbf{x}_0)$ to find a value of L that is smaller. Thus, if we truly are at a minimum, this cannot be the case! We can conclude that if $\mathbf{x}_0$ is a minimum, then $\nabla_{\mathbf{x}} L(\mathbf{x}_0 = 0$. We call points with $\nabla_{\mathbf{x}} L(\mathbf{x}_0 = 0$ *critical points*.

This is nice, because in some rare settings, we *can* explicitly find all the points where the gradient is zero, and find the one with the smallest value.

For a concrete example, consider the function

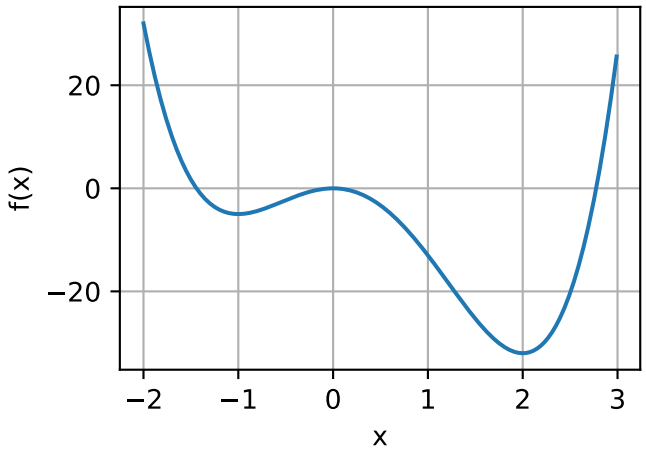
$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{17.4.12}$$

This function has derivative

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{17.4.13}$$

The only possible location of minima are at $x = -1, 0, 2$, where the function takes the values $-5, 0, -32$ respectively, and thus we can conclude that we minimize our function when $x = 2$. A quick plot confirms this.

```
x = np.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



This highlights an important fact to know when working either theoretically or numerically: the only possible points where we can minimize (or maximize) a function will have gradient equal to zero, however, not every point with gradient zero is the true *global* minimum (or maximum).

17.4.4 Multivariate Chain Rule

Let's suppose that we have a function of four variables (w, x, y, and z) which we can make by composing many terms:

$$\begin{aligned} f(u,v) &= (u+v)^2 \\ u(a,b) &= (a+b)^2, \qquad v(a,b) = (a-b)^2, \\ a(w,x,y,z) &= (w+x+y+z)^2, \qquad b(w,x,y,z) = (w+x-y-z)^2. \end{aligned} \tag{17.4.14}$$

Such chains of equations are common when working with neural networks, so trying to understand how to compute gradients of such functions is key. We can start to see visual hints of this connection in Fig. 17.4.1 if we take a look at what variables directly relate to one another.

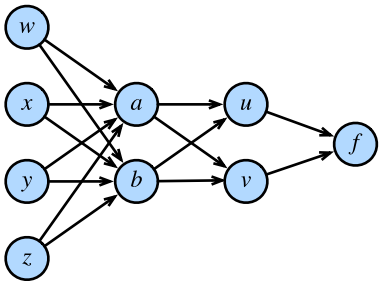


Fig. 17.4.1: The function relations above where nodes represent values and edges show functional dependence.

Nothing stops us from just composing everything from (17.4.14) and writing out that

$$f(w,x,y,z) = \left(\left((w+x+y+z)^2 + (w+x-y-z)^2\right)^2 + \left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2\right)^2. \tag{17.4.15}$$

We may then take the derivative by just using single variable derivatives, but if we did that we would quickly find ourself swamped with terms, many of which are repeats! Indeed, one can see that, for instance:

$$\begin{aligned}\frac{\partial f}{\partial w} = 2\Big(2\left(2(w+x+y+z) - 2(w+x-y-z)\right)\left((w+x+y+z)^2 - (w+x-y-z)^2\right) + \\ 2\left(2(w+x-y-z) + 2(w+x+y+z)\right)\left((w+x-y-z)^2 + (w+x+y+z)^2\right)\Big) \times \\ \left(\left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2 + \left((w+x-y-z)^2 + (w+x+y+z)^2\right)^2\right).\end{aligned} \tag{17.4.16}$$

If we then also wanted to compute $\frac{\partial f}{\partial x}$, we would end up with a similar equation again with many repeated terms, and many *shared* repeated terms between the two derivatives. This represents a massive quantity of wasted work, and if we needed to compute derivatives this way, the whole deep learning revolution would have stalled out before it began!

Let's break up the problem. We will start by trying to understand how f changes when we change a, essentially assuming that w, x, y, and z all do not exist. We will reason as we did back when we worked with the gradient for the first time. Let's take a and add a small amount ϵ to it.

$$\begin{aligned} & f(u(a+\epsilon,b), v(a+\epsilon,b)) \\ \approx & f\left(u(a,b) + \epsilon\frac{\partial u}{\partial a}(a,b), v(a,b) + \epsilon\frac{\partial v}{\partial a}(a,b)\right) \\ \approx & f(u(a,b), v(a,b)) + \epsilon\left[\frac{\partial f}{\partial u}(u(a,b),v(a,b))\frac{\partial u}{\partial a}(a,b) + \frac{\partial f}{\partial v}(u(a,b),v(a,b))\frac{\partial v}{\partial a}(a,b)\right]. \end{aligned} \tag{17.4.17}$$

The first line follows from the definition of partial derivative, and the second follows from the definition of gradient. It is notationally burdensome to track exactly where we evaluate every derivative, as in the expression $\frac{\partial f}{\partial u}(u(a,b), v(a,b))$, so we often abbreviate this to the much more memorable

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{17.4.18}$$

It is useful to think about the meaning of the process. We are trying to understand how a function of the form $f(u(a,b), v(a,b))$ changes its value with a change in a. There are two pathways this can

occur: there is the pathway where $a \to u \to f$ and where $a \to v \to f$. We can compute both of these contributions via the chain rule: $\frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial x}$ and $\frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x}$ respectively, and added up.

Imagine we have a different network of functions where the functions on the right depend on those they are connected to on the left as is shown in Fig. 17.4.2.

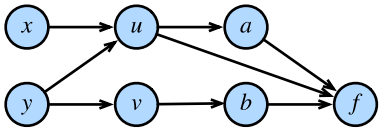


Fig. 17.4.2: Another more subtle example of the chain rule.

To compute something like $\frac{\partial f}{\partial y}$, we need to sum over all (in this case 3) paths from y to f giving

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{17.4.19}$$

Understanding the chain rule in this way will pay great dividends when trying to understand how gradients flow through networks, and why various architectural choices like those in LSTMs (Section 9.2) or residual layers (Section 7.6) can help shape the learning process by controlling gradient flow.

17.4.5 The Backpropagation Algorithm

Let's return to the example of (17.4.14) the previous section where

$$\begin{aligned} f(u,v) &= (u+v)^2 \\ u(a,b) &= (a+b)^2, \qquad v(a,b) = (a-b)^2, \\ a(w,x,y,z) &= (w+x+y+z)^2, \qquad b(w,x,y,z) = (w+x-y-z)^2. \end{aligned} \tag{17.4.20}$$

If we want to compute say $\frac{\partial f}{\partial w}$ we may apply the multi-variate chain rule to see:

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}. \end{aligned} \tag{17.4.21}$$

Let's try using this decomposition to compute $\frac{\partial f}{\partial w}$. Notice that all we need here are the various single step partials:

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u+v), \quad \frac{\partial f}{\partial v} = 2(u+v), \\ \frac{\partial u}{\partial a} &= 2(a+b), \quad \frac{\partial u}{\partial b} = 2(a+b), \\ \frac{\partial v}{\partial a} &= 2(a-b), \quad \frac{\partial v}{\partial b} = -2(a-b), \\ \frac{\partial a}{\partial w} &= 2(w+x+y+z), \quad \frac{\partial b}{\partial w} = 2(w+x-y-z). \end{aligned} \tag{17.4.22}$$

If we write this out into code this becomes a fairly manageable expression.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print("    f at {}, {}, {}, {} is {}".format(w, x, y, z, f))

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, df_dw))
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

However, note that this still does not make it easy to compute something like $\frac{\partial f}{\partial x}$. The reason for that is the *way* we chose to apply the chain rule. If we look at what we did above, we always kept ∂w in the denominator when we could. In this way, we chose to apply the chain rule seeing how w changed every other variable. If that is what we wanted, this would be a good idea. However, think back to our motivation from deep learning: we want to see how every parameter changes the *loss*. In essence, we want to apply the chain rule keeping ∂f in the numerator whenever we can!

To be more explicit, note that we can write

$$\begin{aligned}\frac{\partial f}{\partial w} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w},\\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a},\\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial b} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}.\end{aligned} \tag{17.4.23}$$

Note that this application of the chain rule has us explicitly compute $\frac{\partial f}{\partial u}, \frac{\partial f}{\partial u}, \frac{\partial f}{\partial u}, \frac{\partial f}{\partial u},$ and $\frac{\partial f}{\partial u}$. Nothing stops us from also including the equations:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x},\\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y},\\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial z}.\end{aligned} \tag{17.4.24}$$

and then keeping track of how f changes when we change *any* node in the entire network. Let's implement it.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
```

(continues on next page)

(continued from previous page)

```
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print("    f at {}, {}, {}, {} is {}".format(w, x, y, z, f))

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz
print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, df_dw))
print("df/dx at {}, {}, {}, {} is {}".format(w, x, y, z, df_dx))
print("df/dy at {}, {}, {}, {} is {}".format(w, x, y, z, df_dy))
print("df/dz at {}, {}, {}, {} is {}".format(w, x, y, z, df_dz))
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

The fact that we compute derivatives from f back towards the inputs rather than from the inputs forward to the outputs (as we did in the first code snippet above) is what gives this algorithm its name: *backpropagation*. Note that there are two steps: 1. Compute the value of the function, and the single step partials from front to back. While not done above, this can be combined into a single *forward pass*. 2. Compute the gradient of f from back to front. We call this the *backwards pass*.

This is precisely what every deep learning algorithm implements to allow the computation of the gradient of the loss with respect to every weight in the network at one pass. It is an astonishing fact that we have such a decomposition.

To see how MXNet has encapsulated this, let's take a quick look at this example.

```
# Initialize as ndarrays, then attach gradients
w, x, y, z = np.array(-1), np.array(0), np.array(-2), np.array(1)

w.attach_grad()
x.attach_grad()
y.attach_grad()
z.attach_grad()

# Do the computation like usual, tracking gradients
with autograd.record():
    a, b = (w + x + y + z)**2, (w + x - y - z)**2
    u, v = (a + b)**2, (a - b)**2
```

(continues on next page)

(continued from previous page)

```
    f = (u + v)**2

# Execute backward pass
f.backward()

print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, w.grad))
print("df/dx at {}, {}, {}, {} is {}".format(w, x, y, z, x.grad))
print("df/dy at {}, {}, {}, {} is {}".format(w, x, y, z, y.grad))
print("df/dz at {}, {}, {}, {} is {}".format(w, x, y, z, z.grad))
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

All of what we did above can be done automatically by calling `f.backwards()`.

17.4.6 Hessians

As with single variable calculus, it is useful to consider higher-order derivatives in order to get a handle on how we can obtain a better approximation to a function than using the gradient alone.

There is one immediate problem one encounters when working with higher order derivatives of functions of several variables, and that is there are a large number of them. If we have a function $f(x_1, \ldots, x_n)$ of n variables, then we can take n^2 many second derivatives, namely for any choice of i and j:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j} f\right). \tag{17.4.25}$$

This is traditionally assembled into a matrix called the *Hessian*:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \tag{17.4.26}$$

Not every entry of this matrix is independent. Indeed, we can show that as long as both *mixed partials* (partial derivatives with respect to more than one variable) exist and are continuous, we can say that for any i, and j,

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{17.4.27}$$

This follows by considering first perturbing a function in the direction of x_i, and then perturbing it in x_j and then comparing the result of that with what happens if we perturb first x_j and then x_i, with the knowledge that both of these orders lead to the same final change in the output of f.

As with single variables, we can use these derivatives to get a far better idea of how the function behaves near a point. In particular, we can use it to find the best fitting quadratic near a point $\mathbf{x}_0$, as we saw in a single variable.

Let's see an example. Suppose that $f(x_1, x_2) = a + b_1 x_1 + b_2 x_2 + c_{11} x_1^2 + c_{12} x_1 x_2 + c_{22} x_2^2$. This is the general form for a quadratic in two variables. If we look at the value of the function, its gradient, and its Hessian (17.4.26), all at the point zero:

$$\begin{aligned} f(0,0) &= a, \\ \nabla f(0,0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H} f(0,0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}. \end{aligned} \tag{17.4.28}$$

If we from this, we see we can get our original polynomial back by saying

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2} \mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{17.4.29}$$

In general, if we computed this expansion any point $\mathbf{x}_0$, we see that

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0). \tag{17.4.30}$$

This works for any dimensional input, and provides the best approximating quadratic to any function at a point. To give an example, let's plot the function

$$f(x, y) = x e^{-x^2 - y^2}. \tag{17.4.31}$$

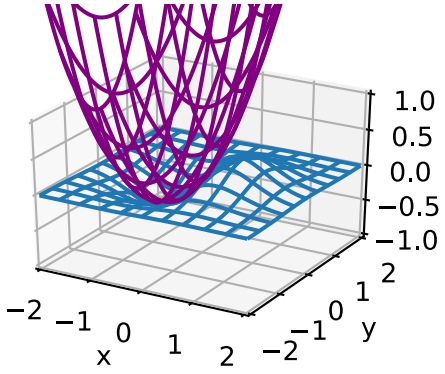
One can compute that the gradient and Hessian are

$$\nabla f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 1 - 2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H} f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 4x^3 - 6x & 4x^2 y - 2y \\ 4x^2 y - 2y & 4xy^2 - 2x \end{pmatrix}. \tag{17.4.32}$$

And thus, with a little algebra, see that the approximating quadratic at $[-1, 0]^\top$ is

$$f(x, y) \approx e^{-1} \left(-1 - (x + 1) + 2(x + 1)^2 + 2y^2 \right). \tag{17.4.33}$$

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101),
                   np.linspace(-2, 2, 101), indexing='ij')
z = x*np.exp(- x**2 - y**2)

# Compute gradient and Hessian at (1, 0)
w = np.exp(-1)*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x, y, w, **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```



This forms the basis for Newton's Algorithm discussed in Section 11.3, where we perform numerical optimization iteratively finding the best fitting quadratic, and then exactly minimizing that quadratic.

17.4.7 A Little Matrix Calculus

Derivatives of functions involving matrices turn out to be particularly nice. This section can become notationally heavy, so may be skipped in a first reading, but it is useful to know how derivatives of functions involving common matrix operations are often much cleaner than one might initially anticipate, particularly given how central matrix operations are to deep learning applications.

Let's begin with an example. Suppose that we have some fixed row vector $\boldsymbol{\beta}$, and we want to take the product function $f(\mathbf{x}) = \boldsymbol{\beta}\mathbf{x}$, and understand how the dot product changes when we change $\mathbf{x}$.

A bit of notation that will be useful when working with matrix derivatives in ML is called the *denominator layout matrix derivative* where we assemble our partial derivatives into the shape of whatever vector, matrix, or tensor is in the denominator of the differential. In this case, we will write

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}. \tag{17.4.34}$$

where we matched the shape of the column vector $\mathbf{x}$.

If we write out our function into components this is

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{17.4.35}$$

If we now take the partial derivative with respect to say β_1, note that everything is zero but the first term, which is just x_1 multiplied by β_1, so the we obtain that

$$\frac{df}{dx_1} = \beta_1, \tag{17.4.36}$$

or more generally that

$$\frac{df}{dx_i} = \beta_i. \tag{17.4.37}$$

We can now reassemble this into a matrix to see

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}^\top. \tag{17.4.38}$$

This illustrates a few factors about matrix calculus that we will often counter throughout this section:

- First, The computations will get rather involved.
- Second, The final results are much cleaner than the intermediate process, and will always look similar to the single variable case. In this case, note that $\frac{d}{dx}(bx) = b$ and $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}\mathbf{x}) = \boldsymbol{\beta}^\top$ are both similar.
- Third, transposes can often appear seemingly from nowhere. The core reason for this is the convention that we match the shape of the denominator, thus when we multiply matrices, we will need to take transposes to match back to the shape of the original term.

To keep building intuition, let's try a computation that is a little harder. Suppose that we have a column vector $\mathbf{x}$, and a square matrix A and we want to compute

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}). \tag{17.4.39}$$

To drive towards easier to manipulate notation, let's consider this problem using Einstein notation. In this case we can write the function as

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \tag{17.4.40}$$

To compute our derivative, we need to understand for every k, what the value of

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{17.4.41}$$

By the product rule, this is

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{17.4.42}$$

For a term like $\frac{dx_i}{dx_k}$, it is not hard to see that this is one when $i = k$ and zero otherwise. This means that every term where i and k are different vanish from this sum, so the only terms that remain in that first sum are the ones where $i = k$. The same reasoning holds for the second term where we need $j = k$. This gives

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{17.4.43}$$

Now, the names of the indices in Einstein notation are arbitrary—the fact that i and j are different is immaterial to this computation at this point, so we can re-index so that they both use i to see that

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{17.4.44}$$

Now, here is where we start to need some practice to go further. Let's try and identify this outcome in terms of matrix operations. $a_{ki} + a_{ik}$ is the k, i-th component of $\mathbf{A} + \mathbf{A}^\top$. This gives

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{17.4.45}$$

Similarly, this term is now the product of the matrix $\mathbf{A} + \mathbf{A}^\top$ by the vector $\mathbf{x}$, so we see that

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x})\right]_k = \frac{d}{dx_k}x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{17.4.46}$$

Thus, we see that the k-th entry of the desired derivative from (17.4.39) is just the k-th entry of the vector on the right, and thus the two are the same. Thus yields

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{17.4.47}$$

This required significantly more work than our last one, but the final result is small. More than that, consider the following computation for traditional single variable derivatives:

$$\frac{d}{dx}(xax) = \frac{dx}{dx}ax + xa\frac{dx}{dx} = (a + a)x. \tag{17.4.48}$$

Equivalently $\frac{d}{dx}(ax^2) = 2ax = (a + a)x$. Again, we get a result that looks rather like the single variable result but with a transpose tossed in.

At this point, the pattern should be looking rather suspicious, so let's try to figure out why. When we take matrix derivatives like this, let's first assume that the expression we get will be another matrix expression: an expression we can write it in terms of products and sums of matrices and their transposes. If such an expression exists, it will need to be true for all matrices. In particular, it will need to be true of 1×1 matrices, in which case the matrix product is just the product of the numbers, the matrix sum is just the sum, and the transpose does nothing at all! In other words, whatever expression we get *must* match the single variable expression. This means that, with some practice, one can often guess matrix derivatives just by knowing what the associated single variable expression must look like!

Let's try this out. Suppose that $\mathbf{X}$ is a $n \times m$ matrix, $\mathbf{U}$ is an $n \times r$ and $\mathbf{V}$ is an $r \times m$. Let's try to compute

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = ? \tag{17.4.49}$$

This computation is important in an area called matrix factorization. For us, however, it is just a derivative to compute. Let's try to imaging what this would be for 1×1 matrices. In that case, we get the expression

$$\frac{d}{dv}(x - uv)^2 = 2(x - uv)u, \tag{17.4.50}$$

where, the derivative is rather standard. If we try to convert this back into a matrix expression we get

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = 2(\mathbf{X} - \mathbf{U}\mathbf{V})\mathbf{U}. \tag{17.4.51}$$

However, if we look at this it does not quite work. Recall that $\mathbf{X}$ is $n \times m$, as is $\mathbf{U}\mathbf{V}$, so the matrix $2(\mathbf{X} - \mathbf{U}\mathbf{V})$ is $n \times m$. On the other hand $\mathbf{U}$ is $n \times r$, and we cannot multiply a $n \times m$ and a $n \times r$ matrix since the dimensions do not match!

We want to get $\frac{d}{d\mathbf{V}}$, which is the same shape of $\mathbf{V}$, which is $r \times m$. So somehow we need to take a $n \times m$ matrix and a $n \times r$ matrix, multiply them together (perhaps with some transposes) to get a $r \times m$. We can do this by multiplying $U^\top$ by $(\mathbf{X} - \mathbf{U}\mathbf{V})$. Thus, we can guess the solution to (17.4.49) is

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = 2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{17.4.52}$$

To show we that this works, we would be remiss to not provide a detailed computation. If we already believe that this rule-of-thumb works, feel free to skip past this derivation. To compute

$$\frac{d}{d\mathbf{V}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2, \tag{17.4.53}$$

we must find for every a, and b

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)^2. \tag{17.4.54}$$

Recalling that all entries of $\mathbf{X}$ and $\mathbf{U}$ are constants as far as $\frac{d}{dv_{ab}}$ is concerned, we may push the derivative inside the sum, and apply the chain rule to the square to get

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \sum_{i,j} 2\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)\left(\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{17.4.55}$$

As in the previous derivation, we may note that $\frac{dv_{kj}}{dv_{ab}}$ is only non-zero if the $k = a$ and $j = b$. If either of those conditions do not hold, the term in the sum is zero, and we may freely discard it. We see that

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \sum_i 2\left(x_{ib}-\sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{17.4.56}$$

An important subtlety here is that the requirement that $k = a$ does not occur inside the inner sum since that k is a dummy variable which we are summing over inside the inner term. For a notationally cleaner example, consider why

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right). \tag{17.4.57}$$

From this point, we may start identifying components of the sum. First,

$$\sum_k u_{ik}v_{kb} = [\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.58}$$

So the entire expression in the inside of the sum is

$$x_{ib}-\sum_k u_{ik}v_{kb} = [\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.59}$$

This means we may now write our derivative as

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = 2\sum_i [\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}u_{ia}. \tag{17.4.60}$$

We want this to look like the a, b element of a matrix so we can use the technique as in the previous example to arrive at a matrix expression, which means that we need to exchange the order of the indices on u_{ia}. If we notice that $u_{ia} = [\mathbf{U}^\top]_{ai}$, we can then write

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = 2\sum_i [\mathbf{U}^\top]_{ai}[\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.61}$$

This is a matrix product, and thus we can conclude that

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = [2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V})]_{ab}. \tag{17.4.62}$$

and thus we may write the solution to (17.4.49)

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = 2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{17.4.63}$$

This matches the solution we guessed above!

It is reasonable to ask at this point, "Why can I not just write down matrix versions of all the calculus rules I have learned? It is clear this is still mechanical. Why do we not just get it over with!" And indeed there are such rules and (Petersen et al., 2008) provides an excellent summary. However, due to the plethora of ways matrix operations can be combined compared to single values, there are many more matrix derivative rules than single variable ones. It is often the case that it is best to work with the indices, or leave it up to automatic differentiation when appropriate.

Summary

- In higher dimensions, we can define gradients which serve the same purpose as derivatives in one dimension. These allow us to see how a multi-variable function changes when we make an arbitrary small change to the inputs.
- The backpropagation algorithm can be seen to be a method of organizing the multi-variable chain rule to allow for the efficient computation of many partial derivatives.
- Matrix calculus allows us to write the derivatives of matrix expressions in concise ways.

Exercises

1. Given a row vector $\boldsymbol{\beta}$, compute the derivatives of both $f(\mathbf{x}) = \boldsymbol{\beta}\mathbf{x}$ and $g(\mathbf{x}) = \mathbf{x}^\top\boldsymbol{\beta}^\top$. Why do you get the same answer?
2. Let $\mathbf{v}$ be an n dimension vector. What is $\frac{\partial}{\partial \mathbf{v}}\|\mathbf{v}\|_2$?
3. Let $L(x, y) = \log(e^x + e^y)$. Compute the gradient. What is the sum of the components of the gradient?
4. Let $f(x, y) = x^2y + xy^2$. Show that the only critical point is $(0, 0)$. By considering $f(x, x)$, determine if $(0, 0)$ is a maximum, minimum, or neither.
5. Suppose that we are minimizing a function $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$. How can we geometrically interpret the condition of $\nabla f = 0$ in terms of g and h?

17.5 Integral Calculus

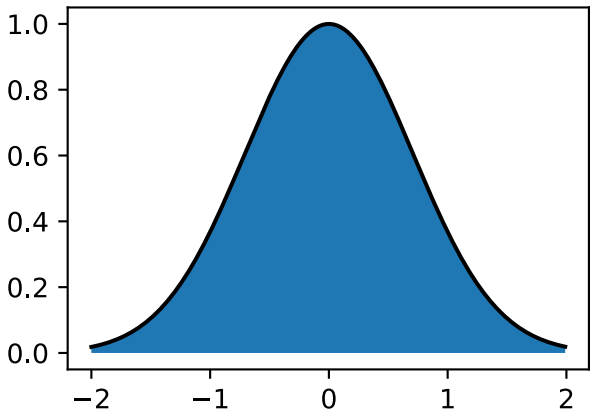
Differentiation only makes up half of the content of a traditional calculus education. The other pillar, integration, starts out seeming a rather disjoint question, "What is the area underneath this curve?" While seemingly unrelated, integration is tightly intertwined with the differentiation via what is known as the *fundamental theorem of calculus*.

At the level of machine learning we discuss in this book, we will not need a deep understanding of integration. However, we will provide a brief introduction to lay the groundwork for any further applications we will encounter later on.

17.5.1 Geometric Interpretation

Suppose that we have a function $f(x)$. For simplicity, let's assume that $f(x)$ is non-negative (never takes a value less than zero). What we want to try and understand is: what is the area contained between $f(x)$ and the x-axis?

```
%matplotlib inline
import d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```



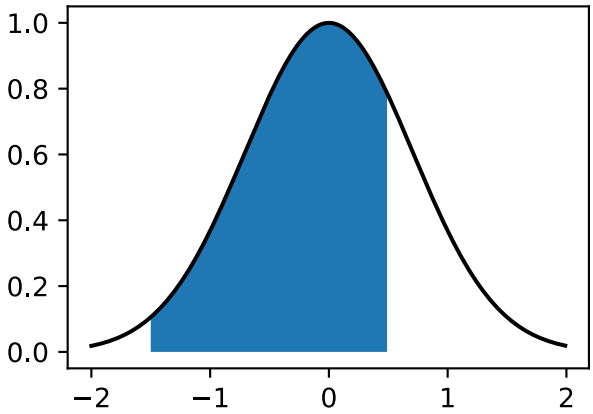
In most cases, this area will be infinite or undefined (consider the area under $f(x) = x^2$), so people will often talk about the area between a pair of ends, say a and b.

```
x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)
```

(continues on next page)

(continued from previous page)

```
d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



We will denote this area by the integral symbol below:

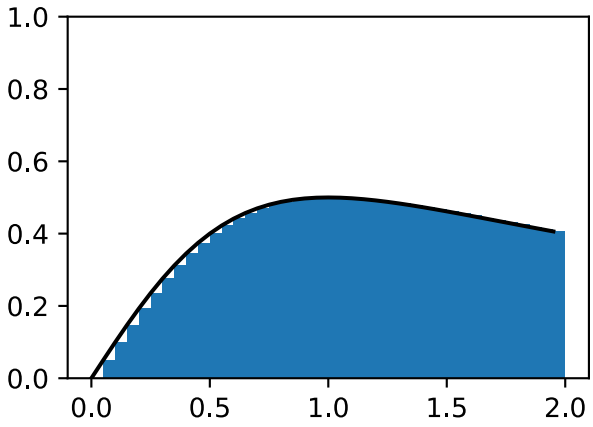
$$\text{Area}(\mathcal{A}) = \int_a^b f(x)\,dx. \tag{17.5.1}$$

The inner variable is a dummy variable, much like the index of a sum in a $\sum$, and so this can be equivalently written with any inner value we like:

$$\int_a^b f(x)\,dx = \int_a^b f(z)\,dz. \tag{17.5.2}$$

There is a traditional way to try and understand how we might try to approximate such integrals: we can imaging taking the region in-between a and b and chopping it into N vertical slices. If N is large, we can approximate the area of each slice by a rectangle, and then add up the areas to get the total area under the curve. Let's take a look at an example doing this in code. We will see how to get the true value in a later section.

```
epsilon = 0.05
a = 0
b = 2

x = np.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = np.sum(epsilon*f)
true = np.log(2) / 2

d2l.set_figsize()
d2l.plt.bar(x.asnumpy(), f.asnumpy(), width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

"Approximation: {}, Truth: {}".format(approx, true)
```




```
'Approximation: 0.79448557, Truth: 0.34657359027997264'
```

The issue is that while it can be done numerically, we can do this approach analytically for only the simplest functions like

$$\int_a^b x \, dx. \tag{17.5.3}$$

Anything somewhat more complex like our example from the code above

$$\int_a^b \frac{x}{1+x^2} \, dx. \tag{17.5.4}$$

is beyond what we can solve with such a direct method.

We will instead take a different approach. We will work intuitively with the notion of the area, and learn the main computational tool used to find integrals: the *fundamental theorem of calculus*. This will be the basis for our study of integration.

17.5.2 The Fundamental Theorem of Calculus

To dive deeper into the theory of integration, let's introduce a function

$$F(x) = \int_0^x f(y)dy. \tag{17.5.5}$$

This function measures the area between 0 and x depending on how we change x. Notice that this is everything we need since

$$\int_a^b f(x) \, dx = F(b) - F(a). \tag{17.5.6}$$

This is a mathematical encoding of the fact that we can measure the area out to the far end-point and then subtract off the area to the near end point as indicated in Fig. 17.5.1.

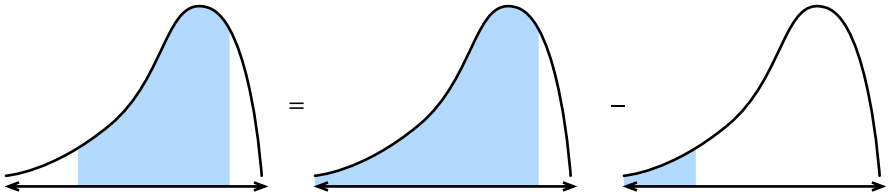


Fig. 17.5.1: Visualizing why we may reduce the problem of computing the area under a curve between two points to computing the area to the left of a point.

Thus, if we can figure out what the integral over any interval is by figuring out what $F(x)$ is.

To do so, let's consider an experiment. As we often do in calculus, let's imaging what happens when we shift the value by a tiny bit. From the comment above, we know that

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y)\,dy. \tag{17.5.7}$$

This tells us that the function changes by the area under a tiny sliver of a function.

This is the point at which we make an approximation. If we look at a tiny sliver of area like this, it looks like this area is close to the rectangular area with height the value of $f(x)$ and the base width ϵ. Indeed, one can show that as $\epsilon \to 0$ this approximation becomes better and better. Thus we can conclude:

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{17.5.8}$$

However, we can now notice: this is exactly the pattern we expect if we were computing the derivative of F! Thus we see the following rather surprising fact:

$$\frac{dF}{dx}(x) = f(x). \tag{17.5.9}$$

This is the *fundamental theorem of calculus*. We may write it in expanded form as

$$\frac{d}{dx}\int_{-\infty}^{x} f(y)\,dy = f(x). \tag{17.5.10}$$

It takes the concept of finding areas (*a priori* rather hard), and reduces it to a statement derivatives (something much more completely understood). One last comment that we must make is that this does not tell we exactly what $F(x)$. Indeed $F(x)+C$ for any C has the same derivative. This is a fact-of-life in the theory of integration. Thankfully, notice that when working with definite integrals, the constants drop out, and thus are irrelevant to the outcome.

$$\int_a^b f(x)\,dx = (F(b)+C) - (F(a)+C) = F(b) - F(a). \tag{17.5.11}$$

This may seem like abstract non-sense, but let's take a moment to appreciate that it has given us a whole new perspective on computing integrals. Our goal is no-longer to do some sort of chop-and-sum process to try and recover the area, rather we need only find a function whose derivative is the function we have! This is incredible since we can now list many rather difficult integrals by just reversing the table from Section 17.3.2. For instance, we know that the derivative of x^n is nx^{n-1}. Thus, we can say using the fundamental theorem (17.5.10) that

$$\int_0^x ny^{n-1}\,dy = x^n - 0^n = x^n. \tag{17.5.12}$$

Similarly, we know that the derivative of e^x is itself, so that means

$$\int_0^x e^x \, dx = e^x - e^0 = e^x - 1. \tag{17.5.13}$$

In this way, we can develop the entire theory of integration leveraging ideas from differential calculus freely. Every integration rule derives from this one fact.

17.5.3 Change of Variables

Just as with differentiation, there are a number of rules which make the computation of integrals more tractable. In fact, every rule of differential calculus (like the product rule, sum rule, and chain rule) has a corresponding rule for integral calculus (integration by parts, linearity of integration, and the change of variables formula respectively). In this section, we will dive into what is arguably the most important from the list: the change of variables formula.

First, suppose that we have a function which is itself an integral:

$$F(x) = \int_0^x f(y) \, dy. \tag{17.5.14}$$

Let's suppose that we want to know how this function looks when we compose it with another to obtain $F(u(x))$. By the chain rule, we know

$$\frac{d}{dx} F(u(x)) = \frac{dF}{dx}(u(x)) \cdot \frac{du}{dx}. \tag{17.5.15}$$

We can turn this into a statement about integration by using the fundamental theorem (17.5.10) as above. This gives

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{dx}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{17.5.16}$$

Recalling that F is itself an integral gives that the left hand side may be rewritten to be

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x \frac{dF}{dx}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{17.5.17}$$

Similarly, recalling that F is an integral allows us to recognize that $\frac{dF}{dx} = f$ using the fundamental theorem (17.5.10), and thus we may conclude

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x f(u(y)) \cdot \frac{du}{dy} \, dy. \tag{17.5.18}$$

This is the *change of variables* formula.

For a more intuitive derivation, consider what happens when we take an integral of $f(u(x))$ between x and $x + \epsilon$. For a small ϵ, this integral is approximately $\epsilon f(u(x))$, the area of the associated rectangle. Now, let's compare this with the integral of $f(y)$ from $u(x)$ to $u(x + \epsilon)$. We know that $u(x + \epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$, so the area of this rectangle is approximately $\epsilon \frac{du}{dx}(x) f(u(x))$. Thus, to make the area of these two rectangles to agree, we need to multiply the first one by $\frac{du}{dx}(x)$ as is illustrated in Fig. 17.5.2.

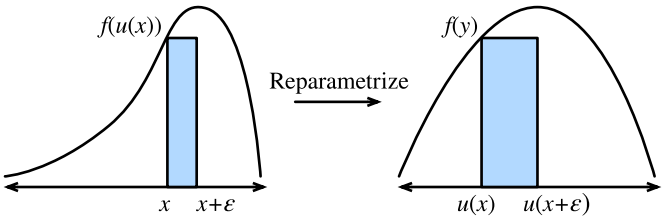


Fig. 17.5.2: Visualizing the transformation of a single thin rectangle under the change of variables.

This tells us that

$$\int_{x}^{x+\epsilon} f(u(y))\frac{du}{dy}(y)\ dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\ dy. \tag{17.5.19}$$

This is the change of variables formula expressed for a single small rectangle.

If $u(x)$ and $f(x)$ are properly chosen, this can allow for the computation of incredibly complex integrals. For instance, if we even chose $f(y) = 1$ and $u(x) = e^{-x^2}$ (which means $\frac{du}{dx}(x) = -2xe^{-x^2}$, this can show for instance that

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\ dy = -2\int_{0}^{1} ye^{-y^2}\ dy, \tag{17.5.20}$$

and thus by rearranging that

$$\int_{0}^{1} ye^{-y^2}\ dy = \frac{1 - e^{-1}}{2}. \tag{17.5.21}$$

17.5.4 A Comment on Sign Conventions

Keen-eyed readers will observe something strange about the computations above. Namely, computations like

$$\int_{e^{-0}}^{e^{-1}} 1\ dy = e^{-1} - 1 < 0, \tag{17.5.22}$$

can produce negative numbers. When thinking about areas, it can be strange to see a negative value, and so it is worth digging into what the convention is.

Mathematicians take the notion of signed areas. This manifests itself in two ways. First, if we consider a function $f(x)$ which is sometimes less than zero, then the area will also be negative. So for instance

$$\int_{0}^{1} (-1)\ dx = -1. \tag{17.5.23}$$

Similarly, integrals which progress from right to left, rather than left to right are also taken to be negative areas

$$\int_{0}^{-1} 1\ dx = -1. \tag{17.5.24}$$

The standard area (from left to right of a positive function) is always positive. Anything obtained by flipping it (say flipping over the x-axis to get the integral of a negative number, or flipping over

the y-axis to get an integral in the wrong order) will produce a negative area. And indeed, flipping twice will give a pair of negative signs that cancel out to have positive area

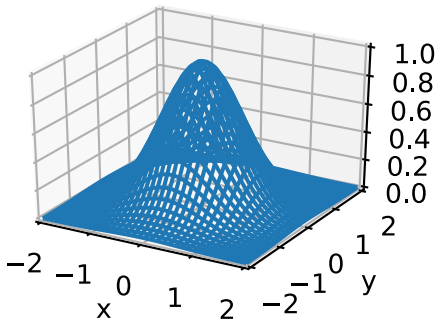
$$\int_0^{-1} (-1)\, dx = 1. \tag{17.5.25}$$

If this discussion sounds familiar, it is! In Section 17.1 we discussed how the determinant represented the signed area in much the same way.

17.5.5 Multiple Integrals

In some cases, we will need to work in higher dimensions. For instance, suppose that we have a function of two variables, like $f(x, y)$ and we want to know the volume under f when x ranges over $[a, b]$ and y ranges over $[c, d]$.

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101), np.linspace(-2, 2, 101),
                   indexing='ij')
z = np.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```



We write this as

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy. \tag{17.5.26}$$

Suppose that we wish to compute this integral. My claim is that we can do this by iteratively computing first the integral in say x and then shifting to the integral in y, that is to say

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy = \int_c^d \left(\int_a^b f(x, y)\, dx \right) dy. \tag{17.5.27}$$

Let's see why this is.

Consider the figure above where we have split the function into $\epsilon \times \epsilon$ squares which we will index with integer coordinates i, j. In this case, our integral is approximately

$$\sum_{i,j} \epsilon^2 f(\epsilon i, \epsilon j). \tag{17.5.28}$$

Once we discretize the problem, we may add up the values on these squares in whatever order we like, and not worry about changing the values. This is illustrated in Fig. 17.5.3. In particular, we can say that

$$\sum_j \epsilon \left(\sum_i \epsilon f(\epsilon i, \epsilon j) \right). \tag{17.5.29}$$

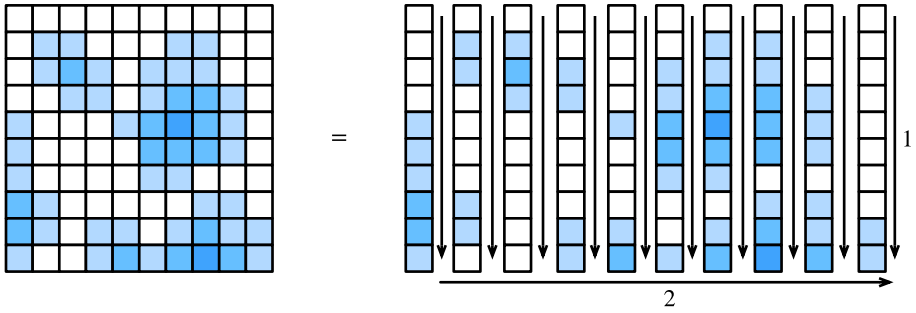


Fig. 17.5.3: Illustrating how to decompose a sum over many squares as a sum over first the columns (1), then adding the column sums together (2).

The sum on the inside is precisely the discretization of the integral

$$G(\epsilon j) = \int_a^b f(x, \epsilon j)\, dx. \tag{17.5.30}$$

Finally, notice that if we combine these two expressions we get

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y)\, dy = \int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy. \tag{17.5.31}$$

Thus putting it all together, we have that

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy = \int_c^d \left(\int_a^b f(x, y)\, dx \right) dy. \tag{17.5.32}$$

Notice that, once discretized, all we did was rearrange the order in which we added a list of numbers. This may make it seem like it is nothing, however this result (called *Fubini's Theorem*) is not always true! For the type of mathematics encountered when doing machine learning (continuous functions), there is no concern, however it is possible to create examples where it fails (for example the function $f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3$ over the rectangle $[0, 2] \times [0, 1]$).

Note that the choice to do the integral in x first, and then the integral in y was arbitrary. We could have equally well chosen to do y first and then x to see

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy = \int_a^b \left(\int_c^d f(x, y)\, dy \right) dx. \tag{17.5.33}$$

Often times, we will condense down to vector notation, and say that for $U = [a, b] \times [c, d]$ this is

$$\int_U f(\mathbf{x})\ d\mathbf{x}. \tag{17.5.34}$$

17.5.6 Change of Variables in Multiple Integrals

As we with single variables in (17.5.18), the ability to change variables inside a higher dimensional integral is a key tool. Let's summarize the result without derivation.

We need a function that reparametrizes our domain of integration. We can take this to be $\phi : \mathbb{R}^n \to \mathbb{R}^n$, that is any function which takes in n real variables and returns another n. To keep the expressions clean, we will assume that ϕ is *injective* which is to say it never folds over itself ($\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$).

In this case, we can say that

$$\int_{\phi(U)} f(\mathbf{x})\ d\mathbf{x} = \int_U f(\phi(\mathbf{x}))\ |\det(D\phi(\mathbf{x}))|\ d\mathbf{x}. \tag{17.5.35}$$

where $D\phi$ is the *Jacobian* of ϕ, which is the matrix of partial derivatives of $\boldsymbol{\phi} = (\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n))$,

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{17.5.36}$$

Looking closely, we see that this is similar to the single variable chain rule (17.5.18), except we have replaced the term $\frac{du}{dx}(x)$ with $|\det(D\phi(\mathbf{x}))|$. Let's see how we can to interpret this term. Recall that the $\frac{du}{dx}(x)$ term existed to say how much we stretched our x-axis by applying u. The same process in higher dimensions is to determine how much we stretch the area (or volume, or hyper-volume) of a little square (or little *hyper-cube*) by applying $\boldsymbol{\phi}$. If $\boldsymbol{\phi}$ was the multiplication by a matrix, then we know how the determinant already gives the answer.

With some work, one can show that the *Jacobian* provides the best approximation to a multivariable function $\boldsymbol{\phi}$ at a point by a matrix in the same way we could approximate by lines or planes with derivatives and gradients. Thus the determinant of the Jacobian exactly mirrors the scaling factor we identified in one dimension.

It takes some work to fill in the details to this, so do not worry if they are not clear now. Let's see at least one example we will make use of later on. Consider the integral

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2 - y^2}\ dx\ dy. \tag{17.5.37}$$

Playing with this integral directly will get us no-where, but if we change variables, we can make significant progress. If we let $\boldsymbol{\phi}(r, \theta) = (r\cos(\theta), r\sin(\theta))$ (which is to say that $x = r\cos(\theta)$, $y = r\sin(\theta)$), then we can apply the change of variable formula to see that this is the same thing as

$$\int_0^{\infty} \int_0^{2\pi} e^{-r^2}\ |\det(D\phi(\mathbf{x}))|\ d\theta\ dr, \tag{17.5.38}$$

where

$$|\det(D\phi(\mathbf{x}))| = \left|\det \begin{bmatrix} \cos(\theta) & -r\sin(\theta) \\ \sin(\theta) & r\cos(\theta) \end{bmatrix}\right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \tag{17.5.39}$$

Thus, the integral is

$$\int_0^\infty \int_0^{2\pi} re^{-r^2} \, d\theta \, dr = 2\pi \int_0^\infty re^{-r^2} \, dr = \pi, \tag{17.5.40}$$

where the final equality follows by the same computation that we used in section Section 17.5.3.

We will meet this integral again when we study continuous random variables in Section 17.6.

Summary

- The theory of integration allows us to answer questions about areas or volumes.
- The fundamental theorem of calculus allows us to leverage knowledge about derivatives to compute areas via the observation that the derivative of the area up to some point is given by the value of the function being integrated.
- Integrals in higher dimensions can be computed by iterating single variable integrals.

Exercises

1. What is $\int_1^2 \frac{1}{x} \, dx$?
2. Use the change of variables formula to integrate $\int_0^{\sqrt{\pi}} x \sin(x^2) \, dx$.
3. What is $\int_{[0,1]^2} xy \, dx \, dy$?
4. Use the change of variables formula to compute $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 \, dy \, dx$ and $\int_0^1 \int_0^2 f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3 \, dx \, dy$ to see they are different.

17.6 Random Variables

In Section 2.6 we saw the basics of how to work with discrete random variables, which in our case refer to those random variables which take either a finite set of possible values, or the integers. In this section, we develop the theory of *continuous random variables*, which are random variables which can take on any real value.

17.6.1 Continuous Random Variables

Continuous random variables are a significantly more subtle topic than discrete random variables. A fair analogy to make is that the technical jump is comparable to the jump between adding lists of numbers and integrating functions. As such, we will need to take some time to develop the theory.

From Discrete to Continuous

To understand the additional technical challenges encountered when working with continuous random variables, let's perform a thought experiment. Suppose that we are throwing a dart at the dart board, and we want to know the probability that it hits exactly 2cm from the center of the board.

To start with, we imagine measuring to a single digit of accuracy, that is to say with bins for 0cm, 1cm, 2cm, and so on. We throw say 100 darts at the dart board, and if 20 of them fall into the bin for 2cm we conclude that 20% of the darts we throw hit the board 2cm away from the center.

However, when we look closer, this does not match our question! We wanted exact equality, whereas these bins hold all that fell between say 1.5cm and 2.5cm.

Undeterred, we continue further. We measure even more precisely, say 1.9cm, 2.0cm, 2.1cm, and now see that perhaps 3 of the 100 darts hit the board in the 2.0cm bucket. Thus we conclude the probability is 3%.

However, this does not solve anything! We have just pushed the issue down one digit further. Let's abstract a bit. Imagine we know the probability that the first k digits match with $2.00000\ldots$ and we want to know the probability it matches for the first $k+1$ digits. It is fairly reasonable to assume that the $k+1^{\text{th}}$ digit is essentially a random choice from the set $\{0, 1, 2, \ldots, 9\}$. At least, we cannot conceive of a physically meaningful process which would force the number of micrometers away form the center to prefer to end in a 7 vs a 3.

What this means is that in essence each additional digit of accuracy we require should decrease probability of matching by a factor of 10. Or put another way, we would expect that

$$P(\text{distance is } 2.00\ldots \text{ to } k \text{ digits}) \approx p \cdot 10^{-k}. \tag{17.6.1}$$

The value p essentially encodes what happens with the first few digits, and the 10^{-k} handles the rest.

Notice that if we know the position accurate to $k = 4$ digits after the decimal. that means we know the value falls within the interval say $[(1.99995, 2.00005]$ which is an interval of length $2.00005 - 1.99995 = 10^{-4}$. Thus, if we call the length of this interval ϵ, we can say

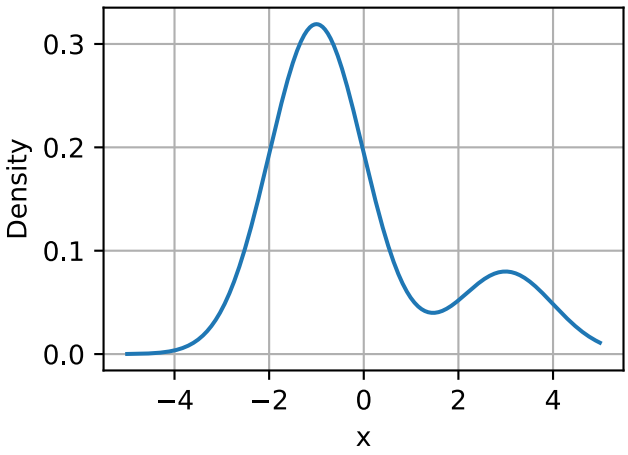
$$P(\text{distance is in an } \epsilon\text{-sized interval around } 2) \approx \epsilon \cdot p. \tag{17.6.2}$$

Let's take this one final step further. We have been thinking about the point 2 the entire time, but never thinking about other points. Nothing is different there fundamentally, but it is the case that the value p will likely be different. We would at least hope that a dart thrower was more likely to hit a point near the center, like 2cm rather than 20cm. Thus, the value p is not fixed, but rather should depend on the point x. This tells us that we should expect

$$P(\text{distance is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{17.6.3}$$

Indeed, (17.6.3) precisely defines the *probability density function*. It is a function $p(x)$ which encodes the relative probability of hitting near one point versus another. Let's visualize what such a function might look like.

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot the probability density function for some random variable
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2)/np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2)/np.sqrt(2 * np.pi)

d2l.plot(x, p, 'x', 'Density')
```



The locations where the function value is large indicates regions where we are more likely to find the random value. The low portions are areas where we are unlikely to find the random value.

Probability Density Functions

Let's now investigate this further. We have already seen what a probability density function is intuitively for a random variable X, namely the density function is a function $p(x)$ so that

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{17.6.4}$$

But what does this imply for the properties of $p(x)$?

First, probabilities are never negative, thus we should expect that $p(x) \geq 0$ as well.

Second, let's imagine that we slice up the $\mathbb{R}$ into an infinite number of slices which are ϵ wide, say with slices $(\epsilon \cdot i, \epsilon \cdot (i+1)]$. For each of these, we know from (17.6.4) the probability is approximately

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{17.6.5}$$

so summed over all of them it should be

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{17.6.6}$$

This is nothing more than the approximation of an integral discussed in Section 17.5, thus we can say that

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x)\,dx. \tag{17.6.7}$$

We know that $P(X \in \mathbb{R}) = 1$, since the random variable must take on *some* number, we can conclude that for any density

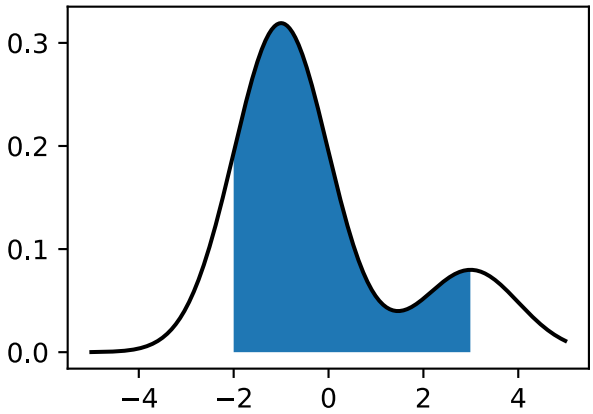
$$\int_{-\infty}^{\infty} p(x)\,dx = 1. \tag{17.6.8}$$

Indeed, digging into this further shows that for any a, and b, we see that

$$P(X \in (a, b]) = \int_{a}^{b} p(x)\,dx. \tag{17.6.9}$$

We may approximate this is code by using the same discrete approximation methods as before. In this case we can approximate the probability of falling in the blue region.

```
# Approximate probability using numerical integration
epsilon = 0.01
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2) / np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2) / np.sqrt(2 * np.pi)

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

"Approximate Probability: {}".format(np.sum(epsilon*p[300:800]))
```




```
'Approximate Probability: 0.7736172'
```

It turns out that these two properties describe exactly the space of possible probability density functions (or *p.d.f.*'s for the commonly encountered abbreviation). They are non-negative functions $p(x) \geq 0$ such that

$$\int_{-\infty}^{\infty} p(x)\,dx = 1. \tag{17.6.10}$$

We interpret this function by using integration to obtain the probability our random variable is in a specific interval:

$$P(X \in (a, b]) = \int_a^b p(x)\, dx. \tag{17.6.11}$$

In `sec_distributions` we will see a number of common distributions, but let's continue working in the abstract.

Cumulative Distribution Functions

In the previous section, we saw the notion of the p.d.f. In practice, this is a commonly encountered method to discuss continuous random variables, but it has one significant pitfall: that the values of the p.d.f. are not themselves probabilities, but rather a function that we must integrate to yield probabilities. There is nothing wrong with a density being larger than 10, as long as it is not larger than 10 for more than an interval of length $1/10$. This can be counter-intuitive, so people often also think in terms of the *cumulative distribution function*, or c.d.f., which *is* a probability.

In particular, by using (17.6.11), we define the c.d.f. for a random variable X with density $p(x)$ by

$$F(x) = \int_{-\infty}^x p(x)\, dx = P(X \leq x). \tag{17.6.12}$$

Let's observe a few properties.

- $F(x) \to 0$ as $x \to -\infty$.
- $F(x) \to 1$ as $x \to \infty$.
- $F(x)$ is non-decreasing ($y > x \implies F(y) \geq F(x)$).
- $F(x)$ is continuous (has no jumps) if X is a continuous random variable.

With the fourth bullet point, note that this would not be true if X were discrete, say taking the values 0 and 1 both with probability $1/2$. In that case

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \geq 1. \end{cases} \tag{17.6.13}$$

In this example, we see one of the benefits of working with the c.d.f., the ability to deal with continuous or discrete random variables in the same framework, or indeed mixtures of the two (flip a coin: if heads return the roll of a die, if tails return the distance of a dart throw from the center of a dart board).

Means

Suppose that we are dealing with a random variables X. The distribution itself can be hard to interpret. It is often useful to be able to summarize the behavior of a random variable concisely. Numbers that help us capture the behavior of a random variable are called *summary statistics*. The most commonly encountered ones are the *mean*, the *variance*, and the *standard deviation*.

The *mean* encodes the average value of a random variable. If we have a discrete random variable X, which takes the values x_i with probabilities p_i, then the mean is given by the weighted average: sum the values times the probability that the random variable takes on that value:

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{17.6.14}$$

The way we should interpret the mean (albeit with caution) is that it tells us essentially where the random variable tends to be located.

As a minimalistic example that we will examine throughout this section, let's take X to be the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. We can compute using (17.6.14) that, for any possible choice of a and p, the mean is

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{17.6.15}$$

Thus we see that the mean is a. This matches the intuition since a is the location around which we centered our random variable.

Because they are helpful, let's summarize a few properties.

- For any random variable X and numbers a and b, we have that $\mu_{aX+b} = a\mu_X + b$.
- If we have two random variables X and Y, we have $\mu_{X+Y} = \mu_X + \mu_Y$.

Means are useful for understanding the average behavior of a random variable, however the mean is not sufficient to even have a full intuitive understanding. Making a profit of $\$10 \pm \$1$ per sale is very different from making $\$10 \pm \$15$ per sale despite having the same average value. The second one has a much larger degree of fluctuation, and thus represents a much larger risk. Thus, to understand the behavior of a random variable, we will need at minimum one more measure: some measure of how widely a random variable fluctuates.

Variances

This leads us to consider the *variance* of a random variable. This is a quantitative measure of how far a random variable deviates from the mean. Consider the expression $X - \mu_X$. This is the deviation of the random variable from its mean. This value can be positive or negative, so we need to do something to make it positive so that we are measuring the magnitude of the deviation.

A reasonable thing to try is to look at $|X - \mu_X|$, and indeed this leads to a useful quantity called the *mean absolute deviation*, however due to connections with other areas of mathematics and statistics, people often use a different solution.

In particular, they look at $(X - \mu_X)^2$. If we look at the typical size of this quantity by taking the mean, we arrive at the variance

$$\sigma_X^2 = \text{Var}(X) = E\left[(X - \mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{17.6.16}$$

The last equality in (17.6.16) holds by expanding out the definition in the middle, and applying the properties of expectation.

Let's look at our example where X is the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. In this case $\mu_X = a$, so all we need to compute is $E\left[X^2\right]$. This can readily be done:

$$E\left[X^2\right] = (a-2)^2 p + a^2(1-2p) + (a+2)p = a^2 + 8p. \tag{17.6.17}$$

Thus, we see that by (17.6.16) our variance is

$$\sigma_X^2 = \text{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{17.6.18}$$

This result again makes sense. The largest p can be is $1/2$ which corresponds to picking $a-2$ or $a+2$ with a coin flip. The variance of this being 4 corresponds to the fact that both $a-2$ and $a+2$ are 2 units away from the mean, and $2^2 = 4$. On the other end of the spectrum, if $p = 0$, this random variable always takes the value 0 and so it has no variance at all.

We will list a few properties of variance below:

- For any random variable X, $\text{Var}(X) \geq 0$, with $\text{Var}(X) = 0$ if and only if X is a constant.
- For any random variable X and numbers a and b, we have that $\text{Var}(aX + b) = a^2\text{Var}(X)$.
- If we have two *independent* random variables X and Y, we have $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$.

When interpreting these values, there can be a bit of a hiccup. In particular, let's try imagining what happens if we keep track of units through this computation. Suppose that we are working with the star rating assigned to a product on the web page. Then a, $a-2$, and $a+2$ are all measured in units of stars. Similarly, the mean μ_X is then also measured in stars (being a weighted average). However, if we get to the variance, we immediately encounter an issue, which is we want to look at $(X - \mu_X)^2$, which is in units of *squared stars*. This means that the variance itself is not comparable to the original measurements. To make it interpretable, we will need to return to our original units.

Standard Deviations

This summary statistics can always be deduced from the variance by taking the square root! Thus we define the *standard deviation* to be

$$\sigma_X = \sqrt{\text{Var}(X)}. \tag{17.6.19}$$

In our example, this means we now have the standard deviation is $\sigma_X = 2\sqrt{2p}$. If we are dealing with units of stars for our review example, σ_X is again in units of stars.

The properties we had for the variance can be restated for the standard deviation.

- For any random variable X, $\sigma_X \geq 0$.
- For any random variable X and numbers a and b, we have that $\sigma_{aX+b} = |a|\sigma_X$
- If we have two *independent* random variables X and Y, we have $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$.

It is natural at this moment to ask, "If the standard deviation is in the units of our original random variable, does it represent something we can draw with regards to that random variable?" The answer is a resounding yes! Indeed much like the mean told we the typical location of our random variable, the standard deviation gives the typical range of variation of that random variable. We can make this rigorous with what is known as Chebychev's inequality:

$$P\left(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{17.6.20}$$

Or to state it verbally in the case of $\alpha = 10$, 99% of the samples from any random variable fall within 10 standard deviations of the mean. This gives an immediate interpretation to our standard summary statistics.

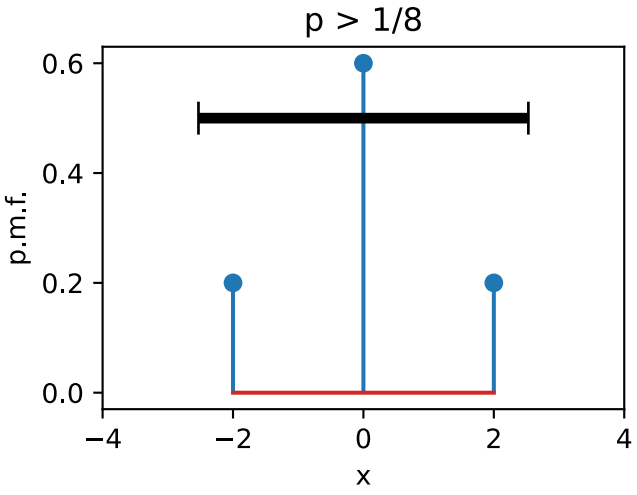
To see how this statement is rather subtle, let's take a look at our running example again where X is the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. We saw that the mean was a and the standard deviation was $2\sqrt{2p}$. This means, if we take Chebychev's inequality (17.6.20) with $\alpha = 2$, we see that the expression is

$$P\left(X \notin [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \leq \frac{1}{4}. \tag{17.6.21}$$

This means that 75% of the time, this random variable will fall within this interval for any value of p. Now, notice that as $p \to 0$, this interval also converges to the single point a. But we know that our random variable takes the values $a-2, a$, and $a+2$ only so eventually we can be certain $a-2$ and $a+2$ will fall outside the interval! The question is, at what p does that happen. So we want to solve: for what p does $a + 4\sqrt{2p} = a + 2$, which is solved when $p = 1/8$, which is *exactly* the first p where it could possibly happen without violating our claim that no more than $1/4$ of samples from the distribution would fall outside the interval ($1/8$ to the left, and $1/8$ to the right).

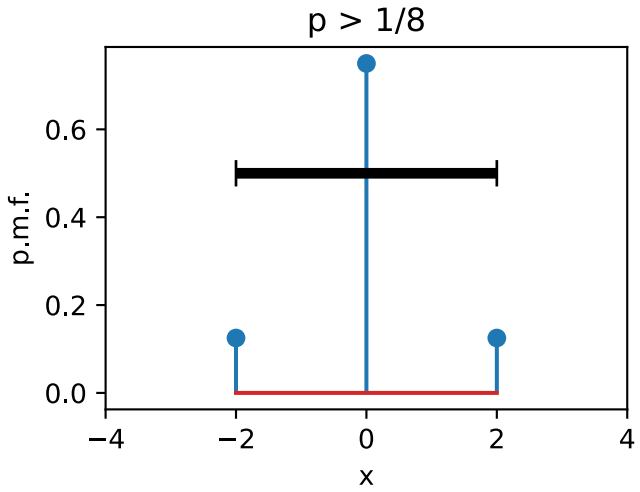
Let's visualize this. We will show the probability of getting the three values as three vertical bars with height proportional to the probability. The interval will be drawn as a horizontal line in the middle. The first plot shows what happens for $p > 1/8$ where the interval safely contains all points.

```
# Define a helper to plot these figures
def plot_chebychev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * np.sqrt(2 * p),
                   a + 4 * np.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title("p > 1/8")

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebychev(0.0, 0.2)
```



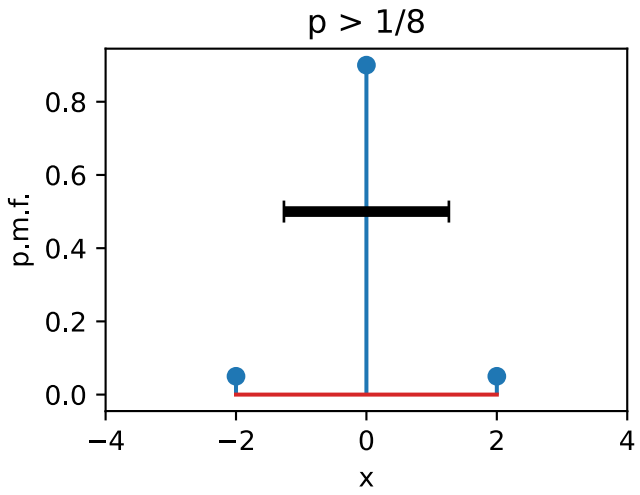
The second shows that at $p = 1/8$, the interval exactly touches the two points. This shows that the inequality is *sharp*, since no smaller interval could be taken while keeping the inequality true.

```
# Plot interval when p = 1/8
plot_chebychev(0.0, 0.125)
```



The third shows that for $p < 1/8$ the interval only contains the center. This does not invalidate the inequality since we only needed to ensure that no more than $1/4$ of the probability falls outside the interval, which means that once $p < 1/8$, the two points at $a - 2$ and $a + 2$ can be discarded.

```
# Plot interval when p < 1/8
plot_chebychev(0.0, 0.05)
```



Means and Variances in the Continuum

This has all been in terms of discrete random variables, but the case of continuous random variables is similar. To intuitively understand how this works, imagine that we split the real number line into intervals of length ϵ given by $(\epsilon i, \epsilon(i+1)]$. Once we do this, our continuous random variable has been made discrete and we can use (17.6.14) say that

$$\begin{aligned}\mu_X &\approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ &\approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon,\end{aligned} \tag{17.6.22}$$

where p_X is the density of X. This is an approximation to the integral of $xp_X(x)$, so we can conclude that

$$\mu_X = \int_{-\infty}^{\infty} x p_X(x) \, dx. \tag{17.6.23}$$

Similarly, using (17.6.16) the variance can be written as

$$\sigma_X^2 = E[X^2] - \mu_X^2 = \int_{-\infty}^{\infty} x^2 p_X(x) \, dx - \left(\int_{-\infty}^{\infty} x p_X(x) \, dx \right)^2. \tag{17.6.24}$$

Everything stated above about the mean, the variance, and the standard deviation above still apply in this case. For instance, if we consider the random variable with density

$$p(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{otherwise.} \end{cases} \tag{17.6.25}$$

we can compute

$$\mu_X = \int_{-\infty}^{\infty} x p(x) \, dx = \int_0^1 x \, dx = \frac{1}{2}. \tag{17.6.26}$$

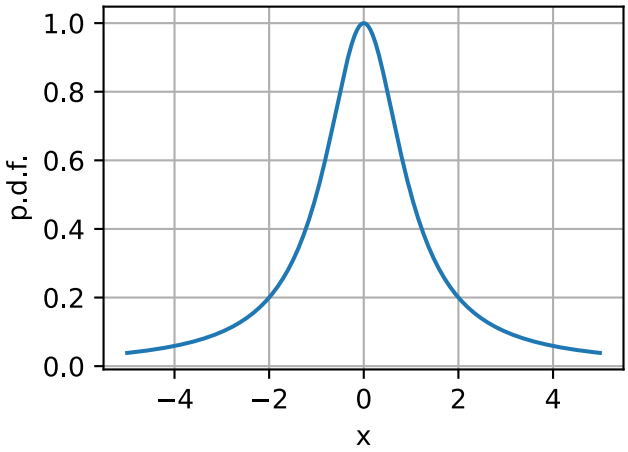
and

$$\sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x) \, dx - \left(\frac{1}{2} \right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{17.6.27}$$

As a warning, let's examine one more example, known as the *Cauchy distribution*. This is the distribution with p.d.f. given by

$$p(x) = \frac{1}{1 + x^2}. \tag{17.6.28}$$

```
# Plot the Cauchy distribution p.d.f.
x = np.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```



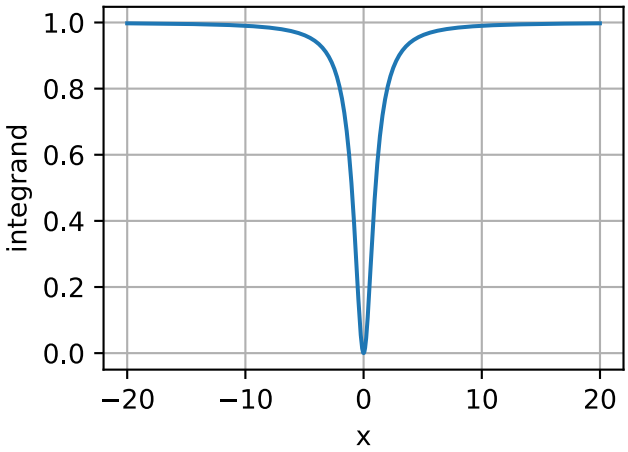
This function looks innocent, and indeed consulting a table of integrals will show it has area one under it, and thus it defines a continuous random variable.

To see what goes astray, let's try to compute the variance of this. This would involve using (17.6.16) computing

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2} \, dx. \tag{17.6.29}$$

The function on the inside looks like this:

```
# Plot the integrand needed to compute the variance
x = np.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



This function clearly has infinite area under it since it is essentially the constant one with a small dip near zero, and indeed we could show that

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2} \, dx = \infty. \tag{17.6.30}$$

This means it does not have a well-defined finite variance.

However, looking deeper shows an even more disturbing result. Let's try to compute the mean using (17.6.14). Using the change of variables formula, we see

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2}\, dx = \frac{1}{2}\int_1^{\infty} \frac{1}{u}\, du. \tag{17.6.31}$$

The integral inside is the definition of the logarithm, so this is in essence $\log(\infty) = \infty$, so there is no well-defined average value either!

Machine learning scientists define their models so that we most often do not need to deal with these issues, and will in the vast majority of cases deal with random variables with well-defined means and variances. However, every so often random variables with *heavy tails* (that is those random variables where the probabilities of getting large values are large enough to make things like the mean or variance undefined) are helpful in modeling physical systems, thus it is worth knowing that they exist.

Joint Density Functions

The above work all assumes we are working with a single real valued random variable. But what if we are dealing with two or more potentially highly correlated random variables? This circumstance is the norm in machine learning: imagine random variables like $R_{i,j}$ which encode the red value of the pixel at the (i, j) coordinate in an image, or P_t which is a random variable given by a stock price at time t. Nearby pixels tend to have similar color, and nearby times tend to have similar prices. We cannot treat them as separate random variables, and expect to create a successful model (we will see in Section 17.8 a model that under-performs due to such an assumption). We need to develop the mathematical language to handle these correlated continuous random variables.

Thankfully, with the multiple integrals in Section 17.5 we can develop such a language. Suppose that we have, for simplicity, two random variables X, Y which can be correlated. Then, similar to the case of a single variable, we can ask the question:

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y). \tag{17.6.32}$$

Similar reasoning to the single variable case shows that this should be approximately

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y) \approx \epsilon^2 p(x, y), \tag{17.6.33}$$

for some function $p(x, y)$. This is referred to as the joint density of X and Y. Similar properties are true for this as we saw in the single variable case. Namely:

- $p(x, y) \geq 0$;
- $\int_{\mathbb{R}^2} p(x, y)\, dx\, dy = 1$;
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y)\, dx\, dy$.

In this way, we can deal with multiple, potentially correlated random variables. If we wish to work with more than two random variables, we can extend the multivariate density to as many coordinates as desired by considering $p(\mathbf{x}) = p(x_1, \ldots, x_n)$. The same properties of being non-negative, and having total integral of one still hold.

Marginal Distributions

When dealing with multiple variables, we often times want to be able to ignore the relationships and ask, "how is this one variable distributed?" Such a distribution is called a *marginal distribution*.

To be concrete, let's suppose that we have two random variables X, Y with joint density given by $p_{X,Y}(x, y)$. We will be using the subscript to indicate what random variables the density is for. The question of finding the marginal distribution is taking this function, and using it to find $p_X(x)$.

As with most things, it is best to return to the intuitive picture to figure out what should be true. Recall that the density is the function p_X so that

$$P(X \in [x, x + \epsilon]) \approx \epsilon \cdot p_X(x). \tag{17.6.34}$$

There is no mention of Y, but if all we are given is $p_{X,Y}$, we need to include Y somehow. We can first observe that this is the same as

$$P(X \in [x, x + \epsilon], \text{ and } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{17.6.35}$$

Our density does not directly tell us about what happens in this case, we need to split into small intervals in y as well, so we can write this as

$$\begin{aligned} \epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x + \epsilon], \text{ and } Y \in [\epsilon \cdot i, \epsilon \cdot (i + 1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i). \end{aligned} \tag{17.6.36}$$

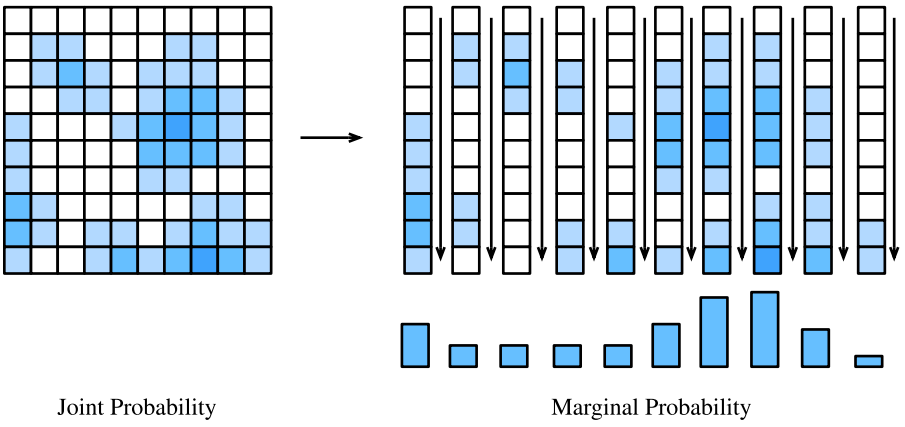


Fig. 17.6.1: By summing along the columns of our array of probabilities, we are able to obtain the marginal distribution for just the random variable represented along the x-axis.

This tells us to add up the value of the density along a series of squares in a line as is show in in Fig. 17.6.1. Indeed, after canceling one factor of epsilon from both sides, and recognizing the sum on the right is the integral over y, we can conclude that

$$\begin{aligned} p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y)\, dy. \end{aligned} \tag{17.6.37}$$

Thus we see

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x,y)\, dy. \tag{17.6.38}$$

This tells us that to get a marginal distribution, we integrate over the variables we do not care about. This process is often referred to as *integrating out* or *marginalized out* the unneeded variables.

Covariance

When dealing with multiple random variables, there is one additional summary statistic which is helpful to know: the *covariance*. This measures the degree that two random variable fluctuate together.

Suppose that we have two random variables X and Y, to begin with, let's suppose they are discrete, taking on values (x_i, y_j) with probability p_{ij}. In this case, the covariance is defined as

$$\sigma_{XY} = \text{Cov}(X,Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij}. = E[XY] - E[X]E[Y]. \tag{17.6.39}$$

To think about this intuitively: consider the following pair of random variables. Suppose that X takes the values 1 and 3, and Y takes the values -1 and 3. Suppose that we have the following probabilities

$$\begin{aligned} P(X = 1 \text{ and } Y = -1) &= \frac{p}{2}, \\ P(X = 1 \text{ and } Y = 3) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = -1) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = 3) &= \frac{p}{2}, \end{aligned} \tag{17.6.40}$$

where p is a parameter in $[0,1]$ we get to pick. Notice that if $p = 1$ then they are both always their minimum or maximum values simultaneously, and if $p = 0$ they are guaranteed to take their flipped values simultaneously (one is large when the other is small and vice versa). If $p = 1/2$, then the four possibilities are all equally likely, and neither should be related. Let's compute the covariance. First, note $\mu_X = 2$ and $\mu_Y = 1$, so we may compute using (17.6.39):

$$\begin{aligned} \text{Cov}(X,Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij} \\ &= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\ &= 4p - 2. \end{aligned} \tag{17.6.41}$$

When $p = 1$ (the case where the are both maximally positive or negative at the same time) has a covariance of 2. When $p = 0$ (the case where they are flipped) the covariance is -2. Finally, when $p = 1/2$ (the case where they are unrelated), the covariance is 0. Thus we see that the covariance measures how these two random variables are related.

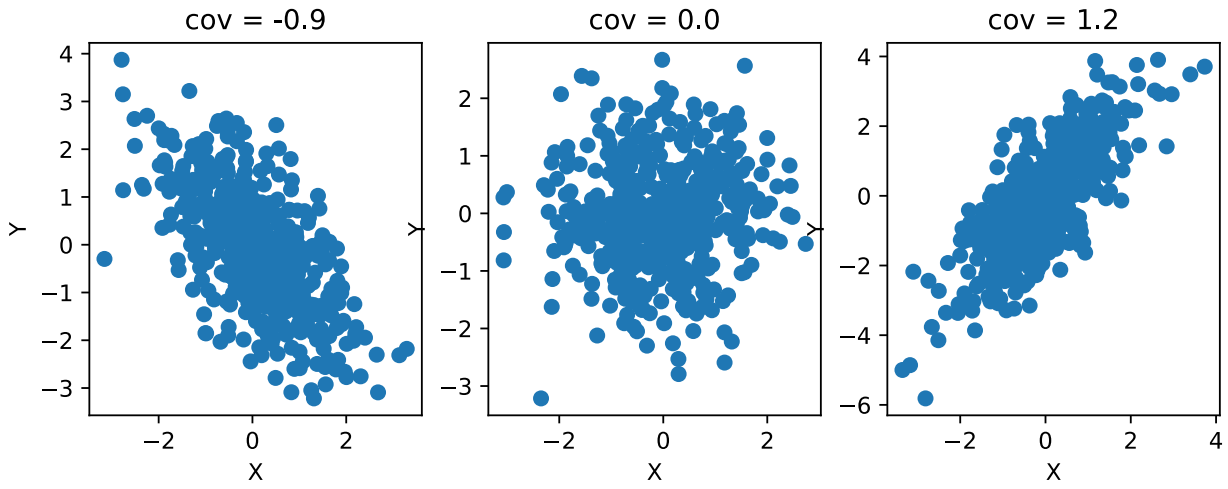
A quick note on the covariance is that it only measures these linear relationships. More complex relationships like $X = Y^2$ where Y is randomly chosen from $\{-2, -1, 0, 1, 2\}$ with equal probability can be missed. Indeed a quick computation shows that these random variables have covariance zero, despite one being a deterministic function of the other.

For continuous random variables, much the same story holds. At this point, we are pretty comfortable with doing the transition between discrete and continuous, so we will provide the continuous analogue of (17.6.39) without any derivation.

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y)p(x, y) \, dx \, dy. \tag{17.6.42}$$

For visualization, let's take a look at a collection of random variables with tunable covariance.

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = covs[i]*X + np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title("cov = {}".format(covs[i]))
d2l.plt.show()
```



Let's see some properties of covariances:

- For any random variable X, $\text{Cov}(X, X) = \text{Var}(X)$.
- For any random variables X, Y and numbers a and b, $\text{Cov}(aX + b, Y) = \text{Cov}(X, aY + b) = a\text{Cov}(X, Y)$.
- If X and Y are independent then $\text{Cov}(X, Y) = 0$.

In addition, we can use the covariance to expand a relationship we saw before. Recall that is X and Y are two independent random variables then

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y). \tag{17.6.43}$$

With knowledge of covariances, we can expand this relationship. Indeed, some algebra can show that in general,

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y). \tag{17.6.44}$$

This allows us to generalize the variance summation rule for correlated random variables.

Correlation

As we did in the case of means and variances, let's now consider units. If X is measured in one unit (say inches), and Y is measured in another (say dollars), the covariance is measured in the product of these two units inches $\times$ dollars. These units can be hard to interpret. What we will often want in this case is a unit-less measurement of relatedness. Indeed, often we do not care about exact quantitative correlation, but rather ask if the correlation is in the same direction, and how strong the relationship is.

To see what makes sense, let's perform a thought experiment. Suppose that we convert our random variables in inches and dollars to be in inches and cents. In this case the random variable Y is multiplied by 100. If we work through the definition, this means that $\text{Cov}(X, Y)$ will be multiplied by 100. Thus we see that in this case a change of units change the covariance by a factor of 100. Thus, to find our unit-invariant measure of correlation, we will need to divide by something else that also gets scaled by 100. Indeed we have a clear candidate, the standard deviation! Indeed if we define the *correlation coefficient* to be

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}, \tag{17.6.45}$$

we see that this is a unit-less value. A little mathematics can show that this number is between -1 and 1 with 1 meaning maximally positively correlated, whereas -1 means maximally negatively correlated.

Returning to our explicit discrete example above, we can see that $\sigma_X = 1$ and $\sigma_Y = 2$, so we can compute the correlation between the two random variables using (17.6.45) to see that

$$\rho(X, Y) = \frac{4p - 2}{1 \cdot 2} = 2p - 1. \tag{17.6.46}$$

This now ranges between -1 and 1 with the expected behavior of 1 meaning most correlated, and -1 meaning minimally correlated.

As another example, consider X as any random variable, and $Y = aX + b$ as any linear deterministic function of X. Then, one can compute that

$$\sigma_Y = \sigma_{aX+b} = |a|\sigma_X, \tag{17.6.47}$$

$$\text{Cov}(X, Y) = \text{Cov}(X, aX + b) = a\text{Cov}(X, X) = a\text{Var}(X), \tag{17.6.48}$$

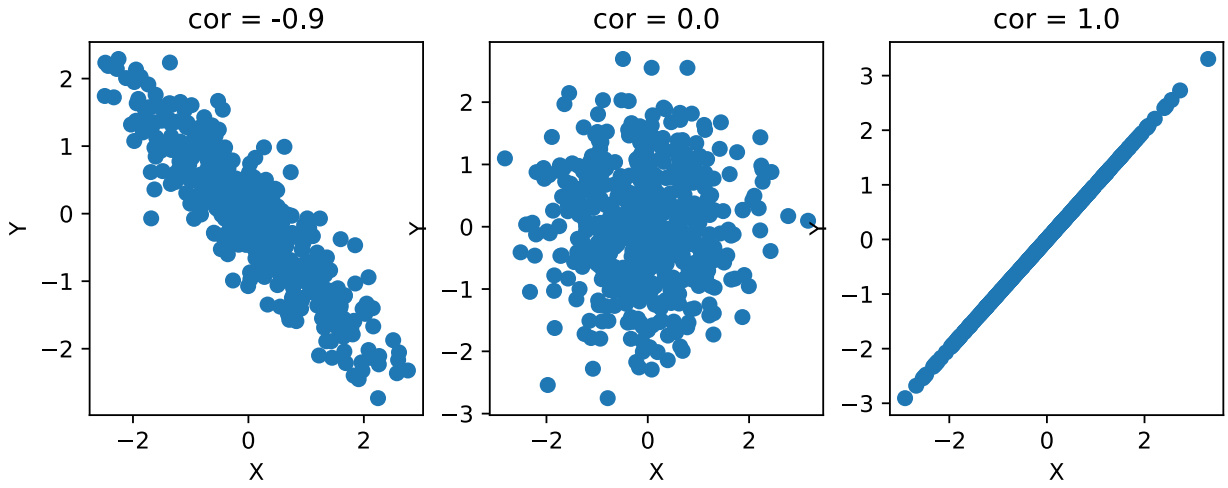
and thus by (17.6.45) that

$$\rho(X, Y) = \frac{a\text{Var}(X)}{|a|\sigma_X^2} = \frac{a}{|a|} = \text{sign}(a). \tag{17.6.49}$$

Thus we see that the correlation is $+1$ for any $a > 0$, and -1 for any $a < 0$ illustrating that correlation measures the degree and directionality the two random variables are related, not the scale that the variation takes.

Let's again plot a collection of random variables with tunable correlation.

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = cors[i] * X + np.sqrt(1 - cors[i]**2) * np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title("cor = {}".format(cors[i]))
d2l.plt.show()
```



Let's list a few properties of the correlation below.

- For any random variable X, $\rho(X, X) = 1$.
- For any random variables X, Y and numbers a and b, $\rho(aX+b, Y) = \rho(X, aY+b) = \rho(X, Y)$.
- If X and Y are independent with non-zero variance then $\rho(X, Y) = 0$.

As a final note, you may feel like some of these formulae are familiar. Indeed, if we expand everything out assuming that $\mu_X = \mu_Y = 0$, we see that this is

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{17.6.50}$$

This looks like a sum of a product of terms divided by the square root of sums of terms. This is exactly the formula for the cosine of the angle between two vectors $\mathbf{v}, \mathbf{w}$ with the different coordinates weighted by p_{ij}:

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2} \sqrt{\sum_i w_i^2}}. \tag{17.6.51}$$

Indeed if we think of norms as being related to standard deviations, and correlations as being cosines of angles, much of the intuition we have from geometry can be applied to thinking about random variables.

Summary

- Continuous random variables are random variables that can take on a continuum of values. They have some technical difficulties that make them more challenging to work with compared to discrete random variables.
- The probability density function allows us to work with continuous random variables by giving a function where the area under the curve on some interval gives the probability of finding a sample point in that interval.
- The cumulative distribution function is the probability of observing the random variable to be less than a given threshold. It can provide a useful alternate viewpoint which unifies discrete and continuous variables.
- The mean is the average value of a random variable.
- The variance is the expected square of the difference between the random variable and its mean.
- The standard deviation is the square root of the variance. It can be thought of as measuring the range of values the random variable may take.
- Chebychev's inequality allows us to make this intuition rigorous by giving an explicit interval that contains the random variable most of the time.
- Joint densities allow us to work with correlated random variables. We may marginalize joint densities by integrating over unwanted random variables to get the distribution of the desired random variable.
- The covariance and correlation coefficient provide a way to measure any linear relationship between two correlated random variables.

Exercises

1. Suppose that we have the random variable with density given by $p(x) = \frac{1}{x^2}$ for $x \geq 1$ and $p(x) = 0$ otherwise. What is $P(X > 2)$?
2. The Laplace distribution is a random variable whose density is given by $p(x = \frac{1}{2}e^{-|x|}$. What is the mean and the standard deviation of this function? As a hint, $\int_0^\infty xe^{-x}\ dx = 1$ and $\int_0^\infty x^2 e^{-x}\ dx = 2$.
3. I walk up to you on the street and say "I have a random variable with mean 1, standard deviation 2, and I observed 25% of my samples taking a value larger than 9." Do you believe me? Why or why not?
4. Suppose that you have two random variables X, Y, with joint density given by $p_{XY}(x, y) = 4xy$ for $x, y \in [0, 1]$ and $p_{XY}(x, y) = 0$ otherwise. What is the covariance of X and Y?

17.7 Maximum Likelihood

One of the most commonly encountered way of thinking in machine learning is the maximum likelihood point of view. This is the concept that when working with a probabilistic model with unknown parameters, the parameters which make the data have the highest probability are the most likely ones.

17.7.1 The Maximum Likelihood Principle

This has a Bayesian interpretation which can be helpful to think about. Suppose that we have a model with parameters $\boldsymbol{\theta}$ and a collection of data points X. For concreteness, we can imagine that $\boldsymbol{\theta}$ is a single value representing the probability that a coin comes up heads when flipped, and X is a sequence of independent coin flips. We will look at this example in depth later.

If we want to find the most likely value for the parameters of our model, that means we want to find

$$\operatorname{argmax} P(\boldsymbol{\theta} \mid X). \tag{17.7.1}$$

By Bayes' rule, this is the same thing as

$$\operatorname{argmax} \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{17.7.2}$$

The expression $P(X)$, a parameter agnostic probability of generating the data, does not depend on $\boldsymbol{\theta}$ at all, and so can be dropped without changing the best choice of $\boldsymbol{\theta}$. Similarly, we may now posit that we have no prior assumption on which set of parameters are better than any others, so we may declare that $P(\boldsymbol{\theta})$ does not depend on theta either! This, for instance, makes sense in our coin flipping example where the probability it comes up heads could be any value in $[0, 1]$ without any prior belief it is fair or not (often referred to as an *uninformative prior*). Thus we see that our application of Bayes' rule shows that our best choice of $\boldsymbol{\theta}$ is the maximum likelihood estimate for $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}). \tag{17.7.3}$$

As a matter of common terminology, the probability of the data given the parameters ($P(X \mid \boldsymbol{\theta})$) is referred to as the *likelihood*.

A Concrete Example

Let's see how this works in a concrete example. Suppose that we have a single parameter θ representing the probability that a coin flip is heads. Then the probability of getting a tails is $1 - \theta$, and so if our observed data X is a sequence with n_H heads and n_T tails, we can use the fact that independent probabilities multiply to see that

$$P(X \mid \theta) = \theta^{n_H}(1 - \theta)^{n_T}. \tag{17.7.4}$$

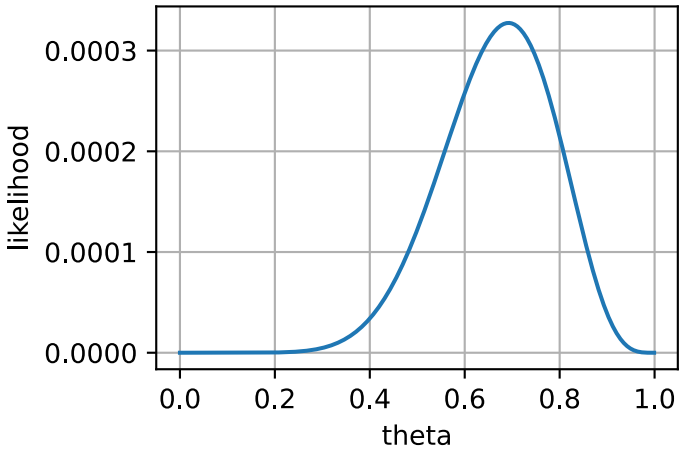
If we flip 13 coins and get the sequence "HHHTHTTHHHHHT", which has $n_H = 9$ and $n_T = 4$, we see that this is

$$P(X \mid \theta) = \theta^9(1 - \theta)^4. \tag{17.7.5}$$

One nice thing about this example will be that we know the answer going in. Indeed, if we said verbally, "I flipped 13 coins, and 9 came up heads, what is our best guess for the probability that the coin comes us heads?," everyone would correctly guess $9/13$. What this maximum likelihood method will give us is a way to get that number from first principals in a way that will generalize to vastly more complex situations.

For our example, the plot of $P(X \mid \theta)$ is as follows:

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()

theta = np.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



This has its maximum value somewhere near our expected $9/13 \approx 0.7\ldots$. To see if it is exactly there, we can turn to calculus. Notice that at the maximum, the function is flat. Thus, we could find the maximum likelihood estimate (17.7.1) by finding the values of θ where the derivative is zero, and finding the one that gives the highest probability. We compute:

$$\begin{aligned} 0 &= \frac{d}{d\theta} P(X \mid \theta) \\ &= \frac{d}{d\theta} \theta^9 (1-\theta)^4 \\ &= 9\theta^8(1-\theta)^4 - 4\theta^9(1-\theta)^3 \\ &= \theta^8(1-\theta)^3(9-13\theta). \end{aligned} \tag{17.7.6}$$

This has three solutions: 0, 1 and $9/13$. The first two are clearly minima, not maxima as they assign probability 0 to our sequence. The final value does *not* assign zero probability to our sequence, and thus must be the maximum likelihood estimate $\hat{\theta} = 9/13$.

17.7.2 Numerical Optimization and the Negative Log-Likelihood

The previous example is nice, but what if we have billions of parameters and data points.

First notice that, if we make the assumption that all the data points are independent, we can no longer practically consider the likelihood itself as it is a product of many probabilities. Indeed, each probability is in $[0, 1]$, say typically of size about $1/2$, and the product of $(1/2)^{1000000000}$ is far below machine precision. We cannot work with that directly.

However, recall that the logarithm turns products to sums, in which case

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6\ldots \tag{17.7.7}$$

This number fits perfectly within even a single precision 32-bit float. Thus, we should consider the *log-likelihood*, which is

$$\log(P(X \mid \boldsymbol{\theta})). \tag{17.7.8}$$

Since the function $x \mapsto \log(x)$ is increasing, maximizing the likelihood is the same thing as maximizing the log-likelihood. Indeed in Section 17.8 we will see this reasoning applied when working with the specific example of the naive Bayes classifier.

We often work with loss functions, where we wish to minimize the loss. We may turn maximum likelihood into the minimization of a loss by taking $-\log(P(X \mid \boldsymbol{\theta}))$, which is the *negative log-likelihood*.

To illustrate this, consider the coin flipping problem from before, and pretend that we do not know the closed form solution. The we may compute that

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1-\theta)). \tag{17.7.9}$$

This can be written into code, and freely optimized even for billions of coin flips.

```
# Set up our data
n_H = 8675309
n_T = 25624

# Initialize our paramteres
theta = np.array(0.5)
theta.attach_grad()

# Perform gradient descent
lr = 0.00000000001
for iter in range(10):
    with autograd.record():
        loss = -(n_H * np.log(theta) + n_T * np.log(1 - theta))
    loss.backward()
    theta -= lr * theta.grad

# Check output
theta, n_H / (n_H + n_T)
```

```
(array(0.50172704), 0.9970550284664874)
```

Numerical convenience is only one reason people like to use negative log-likelihoods. Indeed, there are a several reasons that it can be preferable.

The second reason we consider the log-likelihood is the simplified application of calculus rules. As discussed above, due to independence assumptions, most probabilities we encounter in machine learning are products of individual probabilities.

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{17.7.10}$$

This means that if we directly apply the product rule to compute a derivative we get

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) = &\left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_2 \mid \boldsymbol{\theta})\right) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &\vdots \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right). \end{aligned} \tag{17.7.11}$$

This requires $n(n-1)$ multiplications, along with $(n-1)$ additions, so it is total of quadratic time in the inputs! Sufficient cleverness in grouping terms will reduce this to linear time, but it requires some thought. For the negative log-likelihood we have instead

$$-\log\left(P(X \mid \boldsymbol{\theta})\right) = -\log(P(x_1 \mid \boldsymbol{\theta})) - \log(P(x_2 \mid \boldsymbol{\theta})) \cdots - \log(P(x_n \mid \boldsymbol{\theta})), \tag{17.7.12}$$

which then gives

$$-\frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X \mid \boldsymbol{\theta})\right) = \frac{1}{P(x_1 \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right). \tag{17.7.13}$$

This requires only n divides and $n-1$ sums, and thus is linear time in the inputs.

The third and final reason to consider the negative log-likelihood is the relationship to information theory, which we will discuss in detail in Section 17.10. This is a rigorous mathematical theory which gives a way to measure the degree of information or randomness in a random variable. The key object of study in that field is the entropy which is

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{17.7.14}$$

which measures the randomness of a source. Notice that this is nothing more than the average $-\log$ probability, and thus if we take our negative log-likelihood and divide by the number of data points, we get a relative of entropy known as cross-entropy. This theoretical interpretation alone would be sufficiently compelling to motivate reporting the average negative log-likelihood over the dataset as a way of measuring model performance.

17.7.3 Maximum Likelihood for Continuous Variables

Everything that we have done so far assumes we are working with discrete random variables, but what if we want to work with continuous ones?

The short summary is that nothing at all changes, except we replace all the instances of the probability with the probability density. Recalling that we write densities with lower case p, this means that for example we now say

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{17.7.15}$$

The question becomes, "Why is this OK?" After all, the reason we introduced densities was because probabilities of getting specific outcomes themselves was zero, and thus is not the probability of generating our data for any set of parameters zero?

Indeed, this is the case, and understanding why we can shift to densities is an exercise in tracing what happens to the epsilons.

Let's first re-define our goal. Suppose that for continuous random variables we no longer want to compute the probability of getting exactly the right value, but instead matching to within some range ϵ. For simplicity, we assume our data is repeated observations $x_1, \ldots, x_N$ of identically distributed random variables $X_1, \ldots, X_N$. As we have seen previously, this can be written as

$$\begin{aligned} &P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta}) \\ \approx &\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \end{aligned} \tag{17.7.16}$$

Thus, if we take negative logarithms of this we obtain

$$\begin{aligned} &-\log(P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta})) \\ \approx &-N\log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})). \end{aligned} \tag{17.7.17}$$

If we examine this expression, the only place that the ϵ occurs is in the additive constant $-N\log(\epsilon)$. This does not depend on the parameters $\boldsymbol{\theta}$ at all, so the optimal choice of $\boldsymbol{\theta}$ does not depend on our choice of ϵ! If we demand four digits or four-hundred, the best choice of $\boldsymbol{\theta}$ remains the same, thus we may freely drop the epsilon to see that what we want to optimize is

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{17.7.18}$$

Thus, we see that the maximum likelihood point of view can operate with continuous random variables as easily as with discrete ones by replacing the probabilities with probability densities.

Summary

- The maximum likelihood principle tells us that the best fit model for a given dataset is the one that generates the data with the highest probability.
- Often people work with the negative log-likelihood instead for a variety of reasons: numerical stability, conversion of products to sums (and the resulting simplification of gradient computations), and theoretical ties to information theory.
- While simplest to motivate in the discrete setting, it may be freely generalized to the continuous setting as well by maximizing the probability density assigned to the datapoints.

Exercises

1. Suppose that you know that a random variable has density $\frac{1}{\alpha}e^{-\alpha x}$ for some value α. You obtain a single observation from the random variable which is the number 3. What is the maximum likelihood estimate for α?
2. Suppose that you have a dataset of samples $\{x_i\}_{i=1}^N$ drawn from a Gaussian with unknown mean, but variance 1. What is the maximum likelihood estimate for the mean?

17.8 Naive Bayes

Throughout the previous sections, we learned about the theory of probability and random variables. To put this theory to work, let's introduce the *naive Bayes* classifier. This uses nothing but probabilistic fundamentals to allow us to perform classification of digits.

Learning is all about making assumptions. If we want to classify a new data point that we have never seen before we have to make some assumptions about which data points are similar to each other. The naive Bayes classifier, a popular and remarkably clear algorithm, assumes all features are independent from each other to simplify the computation. In this section, we will apply this model to recognize characters in images.

```
%matplotlib inline
import d2l
import math
from mxnet import gluon, np, npx
npx.set_np()
d2l.use_svg_display()
```

17.8.1 Optical Character Recognition

MNIST (LeCun et al., 1998) is one of widely used datasets. It contains 60,000 images for training and 10,000 images for validation. Each image contains a handwritten digit from 0 to 9. The task is classifying each image into the corresponding digit.

Gluon provides a `MNIST` class in the `data.vision` module to automatically retrieve the dataset from the internet. Subsequently, Gluon will use the already-downloaded local copy. We specify whether we are requesting the training set or the test set by setting the value of the parameter `train` to `True` or `False`, respectively. Each image is a grayscale image with both width and height of 28 with shape $(28,28,1)$. We use a customized transformation to remove the last channel dimension. In addition, the dataset represents each pixel by a unsigned 8-bit integer. We quantize them into binary features to simplify the problem.

```
def transform(data, label):
    return np.floor(data.astype('float32') / 128).squeeze(axis=-1), label
```

(continues on next page)

(continued from previous page)

```
mnist_train = gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = gluon.data.vision.MNIST(train=False, transform=transform)
```

We can access a particular example, which contains the image and the corresponding label.

```
image, label = mnist_train[2]
image.shape, label
```

```
((28, 28), array(4, dtype=int32))
```

Our example, stored here in the variable `image`, corresponds to an image with a height and width of 28 pixels.

```
image.shape, image.dtype
```

```
((28, 28), dtype('float32'))
```

Our code stores the label of each image as a scalar. Its type is a 32-bit integer.

```
label, type(label), label.dtype
```

```
(array(4, dtype=int32), mxnet.numpy.ndarray, dtype('int32'))
```

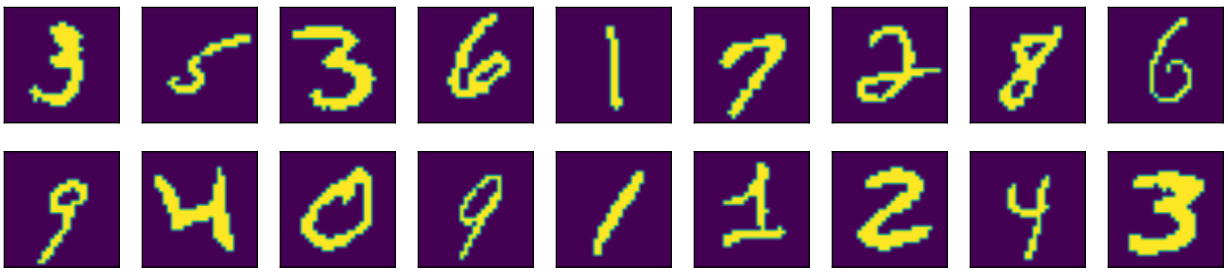
We can also access multiple examples at the same time.

```
images, labels = mnist_train[10:38]
images.shape, labels.shape
```

```
((28, 28, 28), (28,))
```

Let's visualize these examples.

```
d2l.show_images(images, 2, 9);
```



17.8.2 The Probabilistic Model for Classification

In a classification task, we map an example into a category. Here an example is a grayscale 28×28 image, and a category is a digit. (Refer to Section 3.4 for a more detailed explanation.) One natural way to express the classification task is via the probabilistic question: what is the most likely label given the features (i.e., image pixels)? Denote by $\mathbf{x} \in \mathbb{R}^d$ the features of the example and $y \in \mathbb{R}$ the label. Here features are image pixels, where we can reshape a 2-dimensional image to a vector so that $d = 28^2 = 784$, and labels are digits. The probability of the label given the features is $p(y \mid \mathbf{x})$. If we are able to compute these probabilities, which are $p(y \mid \mathbf{x})$ for $y = 0, \ldots, 9$ in our example, then the classifier will output the prediction $\hat{y}$ given by the expression:

$$\hat{y} = \operatorname{argmax} p(y \mid \mathbf{x}). \tag{17.8.1}$$

Unfortunately, this requires that we estimate $p(y \mid \mathbf{x})$ for every value of $\mathbf{x} = x_1, ..., x_d$. Imagine that each feature could take one of 2 values. For example, the feature $x_1 = 1$ might signify that the word apple appears in a given document and $x_1 = 0$ would signify that it does not. If we had 30 such binary features, that would mean that we need to be prepared to classify any of 2^{30} (over 1 billion!) possible values of the input vector $\mathbf{x}$.

Moreover, where is the learning? If we need to see every single possible example in order to predict the corresponding label then we are not really learning a pattern but just memorizing the dataset.

17.8.3 The Naive Bayes Classifier

Fortunately, by making some assumptions about conditional independence, we can introduce some inductive bias and build a model capable of generalizing from a comparatively modest selection of training examples. To begin, let's use Bayes theorem, to express the classifier as

$$\hat{y} = \operatorname{argmax}_y p(y \mid \mathbf{x}) = \operatorname{argmax}_y \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})}. \tag{17.8.2}$$

Note that the denominator is the normalizing term $p(\mathbf{x})$ which does not depend on the value of the label y. As a result, we only need to worry about comparing the numerator across different values of y. Even if calculating the denominator turned out to be intractable, we could get away with ignoring it, so long as we could evaluate the numerator. Fortunately, even if we wanted to recover the normalizing constant, we could. We can always recover the normalization term since $\sum_y p(y \mid \mathbf{x}) = 1$.

Now, let's focus on $p(\mathbf{x} \mid y)$. Using the chain rule of probability, we can express the term $p(\mathbf{x} \mid y)$ as

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{17.8.3}$$

By itself, this expression does not get us any further. We still must estimate roughly 2^d parameters. However, if we assume that *the features are conditionally independent of each other, given the label*, then suddenly we are in much better shape, as this term simplifies to $\prod_i p(x_i \mid y)$, giving us the predictor

$$\hat{y} = \operatorname{argmax}_y \prod_{i=1}^{d} p(x_i \mid y)p(y). \tag{17.8.4}$$

If we can estimate $\prod_i p(x_i = 1 \mid y)$ for every i and y, and save its value in $P_{xy}[i, y]$, here P_{xy} is a $d \times n$ matrix with n being the number of classes and $y \in \{1, \ldots, n\}$. In addition, we estimate $p(y)$

for every y and save it in $P_y[y]$, with P_y a n-length vector. Then for any new example $\mathbf{x}$, we could compute

$$\hat{y} = \text{argmax}_y \prod_{i=1}^{d} P_{xy}[x_i, y]P_y[y], \tag{17.8.5}$$

for any y. So our assumption of conditional independence has taken the complexity of our model from an exponential dependence on the number of features $\mathcal{O}(2^d n)$ to a linear dependence, which is $\mathcal{O}(dn)$.

17.8.4 Training

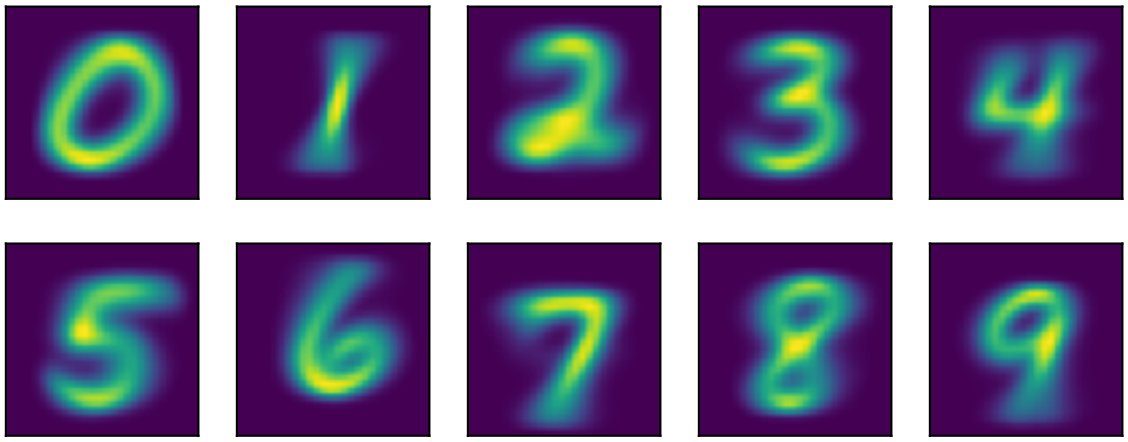
The problem now is that we do not know P_{xy} and P_y. So we need to estimate their values given some training data first. This is *training* the model. Estimating P_y is not too hard. Since we are only dealing with 10 classes, we may count the number of occurrences n_y for each of the digits and divide it by the total amount of data n. For instance, if digit 8 occurs $n_8 = 5,800$ times and we have a total of $n = 60,000$ images, the probability estimate is $p(y = 8) = 0.0967$.

```
X, Y = mnist_train[:]  # All training examples

n_y = np.zeros((10))
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
array([0.09871667, 0.11236667, 0.0993    , 0.10218333, 0.09736667,
       0.09035   , 0.09863333, 0.10441667, 0.09751666, 0.09915   ])
```

Now on to slightly more difficult things P_{xy}. Since we picked black and white images, $p(x_i \mid y)$ denotes the probability that pixel i is switched on for class y. Just like before we can go and count the number of times n_{iy} such that an event occurs and divide it by the total number of occurrences of y, i.e., n_y. But there is something slightly troubling: certain pixels may never be black (e.g., for well cropped images the corner pixels might always be white). A convenient way for statisticians to deal with this problem is to add pseudo counts to all occurrences. Hence, rather than n_{iy} we use $n_{iy} + 1$ and instead of n_y we use $n_y + 1$. This is also called *Laplace Smoothing*. It may seem ad-hoc, however it may be well motivated from a Bayesian point-of-view.

```
n_x = np.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = np.array(X.asnumpy()[Y.asnumpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



By visualizing these $10 \times 28 \times 28$ probabilities (for each pixel for each class) we could get some mean looking digits.

Now we can use (17.8.5) to predict a new image. Given $\mathbf{x}$, the following functions computes $p(\mathbf{x} \mid y)p(y)$ for every y.

```
def bayes_pred(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(axis=1)  # p(x|y)
    return np.array(p_xy) * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

This went horribly wrong! To find out why, let's look at the per pixel probabilities. They are typically numbers between 0.001 and 1. We are multiplying 784 of them. At this point it is worth mentioning that we are calculating these numbers on a computer, hence with a fixed range for the exponent. What happens is that we experience *numerical underflow*, i.e., multiplying all the small numbers leads to something even smaller until it is rounded down to zero. We discussed this as a theoretical issue in Section 17.7, but we see the phenomena clearly here in practice.

As discussed in that section, we fix this by use the fact that $\log ab = \log a + \log b$, i.e., we switch to summing logarithms. Even if both a and b are small numbers, the logarithm values should be in a proper range.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Since the logarithm is an increasing function, we can rewrite (17.8.5) as

$$\hat{y} = \operatorname{argmax}_y \sum_{i=1}^{d} \log P_{xy}[x_i, y] + \log P_y[y]. \tag{17.8.6}$$

We can implement the following stable version:

```
log_P_xy = np.log(P_xy)
log_P_xy_neg = np.log(1 - P_xy)
log_P_y = np.log(P_y)

def bayes_pred_stable(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
array([-269.00424, -301.73447, -245.21458, -218.8941 , -193.46907,
       -206.10315, -292.54315, -114.62834, -220.35619, -163.18881])
```

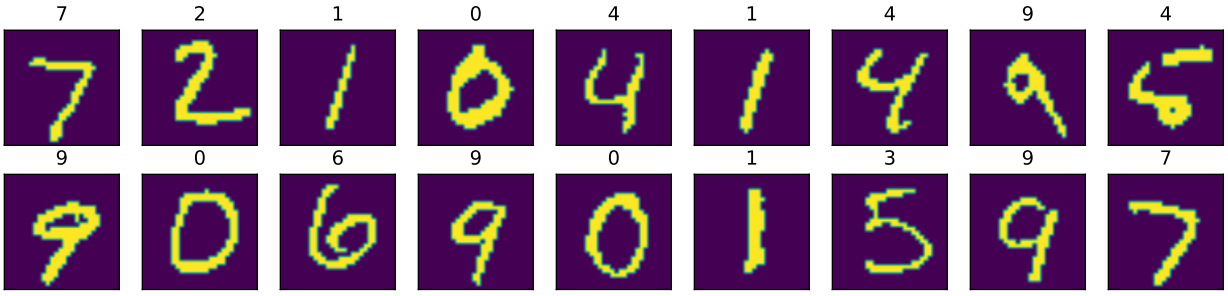
We may now check if the prediction is correct.

```
# Convert label which is a scalar tensor of int32 dtype
# to a Python scalar integer for comparison
py.argmax(axis=0) == int(label)
```

```
array(True)
```

If we now predict a few validation examples, we can see the Bayes classifier works pretty well.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(axis=0).astype(np.int32) for x in X]

X, y = mnist_test[:18]
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Finally, let's compute the overall accuracy of the classifier.

```
X, y = mnist_test[:]
preds = np.array(predict(X), dtype=np.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8426
```

Modern deep networks achieve error rates of less than 0.01. The relatively poor performance is due to the incorrect statistical assumptions that we made in our model: we assumed that each and every pixel are *independently* generated, depending only on the label. This is clearly not how humans write digits, and this wrong assumption led to the downfall of our overly naive (Bayes) classifier.

Summary

- Using Bayes' rule, a classifier can be made by assuming all observed features are independent.
- This classifier can be trained on a dataset by counting the number of occurrences of combinations of labels and pixel values.
- This classifier was the gold standard for decades for tasks such as spam detection.

Exercises

1. Consider the dataset $[[0, 0], [0, 1], [1, 0], [1, 1]]$ with labels given by the XOR of the two elements $[0, 1, 1, 0]$. What are the probabilities for a Naive Bayes classifier built on this dataset. Does it successfully classify our points? If not, what assumptions are violated?
2. Suppose that we did not use Laplace smoothing when estimating probabilities and a data point arrived at testing time which contained a value never observed in training. What would the model output?
3. The naive Bayes classifier is a specific example of a Bayesian network, where the dependence of random variables are encoded with a graph structure. While the full theory is beyond the scope of this section (see (Koller & Friedman, 2009) for full details), explain why allowing explicit dependence between the two input variables in the XOR model allows for the creation of a successful classifier.

17.9 Statistics

Undoubtedly, to be a top deep learning practitioner, the ability to train the state-of-the-art and high accurate models is crucial. However, it is often unclear when improvements are significant, or only the result of random fluctuations in the training process. To be able to discuss uncertainty in estimated values, we must learn some statistics.

The earliest reference of *statistics* can be traced back to an Arab scholar Al-Kindi in the 9^{th}-century, who gave a detailed description of how to use statistics and frequency analysis to decipher encrypted messages. After 800 years, the modern statistics arose from Germany in 1700s, when the researchers focused on the demographic and economic data collection and analysis. Today, statistics is the science subject that concerns the collection, processing, analysis, interpretation and visualization of data. What is more, the core theory of statistics has been widely used in the research within academia, industry, and government.

More specifically, statistics can be divided to *descriptive statistics* and *statistical inference*. The former focus on summarizing and illustrating the features of a collection of observed data, which is referred to as a *sample*. The sample is drawn from a *population*, denotes the total set of similar individuals, items, or events of our experiment interests. Contrary to descriptive statistics, *statistical inference* further deduces the characteristics of a population from the given *samples*, based on the assumptions that the sample distribution can replicate the population distribution at some degree.

You may wonder: "What is the essential difference between machine learning and statistics?" Fundamentally speaking, statistics focuses on the inference problem. This type of problems includes modeling the relationship between the variables, such as causal inference, and testing the statistically significance of model parameters, such as A/B testing. In contrast, machine learning emphasizes on making accurate predictions, without explicitly programming and understanding each parameter's functionality.

In this section, we will introduce three types of statistics inference methods: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals. These methods can help us infer the characteristics of a given population, i.e., the true parameter θ. For brevity, we assume that the true parameter θ of a given population is a scalar value. It is straightforward to extend to the case where θ is a vector or a tensor, thus we omit it in our discussion.

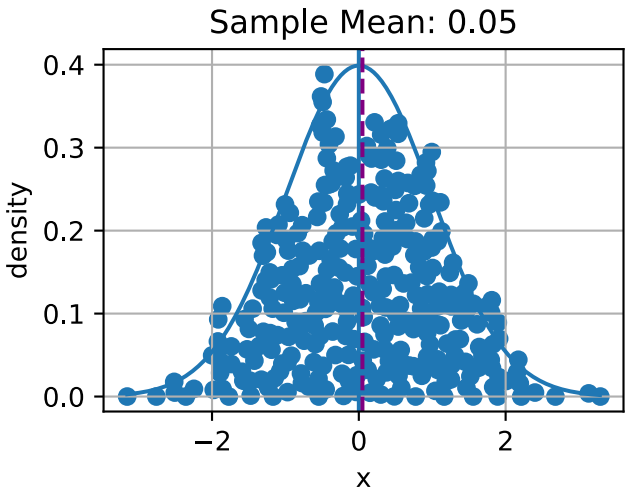
17.9.1 Evaluating and Comparing Estimators

In statistics, an *estimator* is a function of given samples used to estimate the true parameter θ. We will write $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ for the estimate of θ after observing the samples $\{x_1, x_2, \ldots, x_n\}$.

We've seen simple examples of estimators before in section Section 17.7. If you have a number of samples from a Bernoulli random variable, then the maximum likelihood estimate for the probability the random variable is one can be obtained by counting the number of ones observed and dividing by the total number of samples. Similarly, an exercise asked you to show that the maximum likelihood estimate of the mean of a Gaussian given a number of samples is given by the average value of all the samples. These estimators will almost never give the true value of the parameter, but ideally for a large number of samples the estimate will be close.

As an example, we show below the true density of a Gaussian random variable with mean zero and variance one, along with a collection samples from that Gaussian. We constructed the y coordinate so every point is visible and the relationship to the original density is clearer.

```
import d2l
from mxnet import np, npx
import random
npx.set_np()

# Sample datapoints and create y coordinate
epsilon = 0.1
random.seed(8675309)
xs = np.random.normal(loc=0, scale=1, size=(300,))

ys = [np.sum(np.exp(-(xs[0:i] - xs[i])**2 / (2 * epsilon**2))
             / np.sqrt(2*np.pi*epsilon**2)) / len(xs) for i in range(len(xs))]

# Compute true density
xd = np.arange(np.min(xs), np.max(xs), 0.01)
yd = np.exp(-xd**2/2) / np.sqrt(2 * np.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=np.mean(xs), linestyle='--', color='purple')
d2l.plt.title("Sample Mean: {:.2f}".format(float(np.mean(xs))))
d2l.plt.show()
```



There can be many ways to compute an estimator of a parameter $\hat{\theta}_n$. In this section, we introduce three common methods to evaluate and compare estimators: the mean squared error, the standard deviation, and statistical bias.

Mean Squared Error

Perhaps the simplest metric used to evaluate estimators is the *mean squared error (MSE)* (or *:math:`l_2` loss*) of an estimator can be defined as

$$\text{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{17.9.1}$$

This allows us to quantify the average squared deviation from the true value. MSE is always non-negative. If you have read Section 3.1, you will recognize it as the most commonly used regression loss function. As a measure to evaluate an estimator, the closer its value to zero, the closer the estimator is close to the true parameter θ.

Statistical Bias

The MSE provides a natural metric, but we can easily imagine multiple different phenomena that might make it large. Two that we will see are fundamentally important are the fluctuation in the estimator due to randomness in the dataset, and systematic error in the estimator due to the estimation procedure.

First, let's measure the systematic error. For an estimator $\hat{\theta}_n$, the mathematical illustration of *statistical bias* can be defined as

$$\text{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{17.9.2}$$

Note that when $\text{bias}(\hat{\theta}_n) = 0$, the expectation of the estimator $\hat{\theta}_n$ is equal to the true value of parameter. In this case, we say $\hat{\theta}_n$ is an unbiased estimator. In general, an unbiased estimator is better than a biased estimator since its expectation is the same as the true parameter.

It is worth being aware, however, that biased estimators are frequently used in practice. There are cases where unbiased estimators do not exist without further assumptions, or are intractable to compute. This may seem like a significant flaw in an estimator, however the majority of estimators encountered in practice are at least asymptotically unbiased in the sense that the bias tends to zero as the number of available samples tends to infinity: $\lim_{n \to \infty} \text{bias}(\hat{\theta}_n) = 0$.

Variance and Standard Deviation

Second, let's measure the randomness in the estimator. Recall from Section 17.6, the *standard deviation* (or *standard error*) is defined as the squared root of the variance. We may measure the degree of fluctuation of an estimator by measuring the standard deviation or variance of that estimator.

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{17.9.3}$$

It is important to compare (17.9.3) to (17.9.1). In this equation we do not compare to the true population value θ, but instead to $E(\hat{\theta}_n)$, the expected sample mean. Thus we are not measuring how far the estimator tends to be from the true value, but instead we measuring the fluctuation of the estimator itself.

The Bias-Variance Trade-off

It is intuitively clear that these two components contribute to the mean squared error. What is somewhat shocking is that we can show that this is actually a *decomposition* of the mean squared error into two contributions. That is to say that we can write the mean squared error as the sum of the variance and the square or the bias.

$$\begin{aligned}\mathrm{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - E(\hat{\theta}_n) + E(\hat{\theta}_n) - \theta)^2] \\ &= E[(\hat{\theta}_n - E(\hat{\theta}_n))^2] + E[(E(\hat{\theta}_n) - \theta)^2] \\ &= \mathrm{Var}(\hat{\theta}_n) + [\mathrm{bias}(\hat{\theta}_n)]^2.\end{aligned} \tag{17.9.4}$$

We refer the above formula as *bias-variance trade-off*. The mean squared error can be divided into precisely two sources of error: the error from high bias and the error from high variance. On the one hand, the bias error is commonly seen in a simple model (such as a linear regression model), which cannot extract high dimensional relations between the features and the outputs. If a model suffers from high bias error, we often say it is *underfitting* or lack of *generalization* as introduced in (Section 4.4). On the flip side, the other error source—high variance usually results from a too complex model, which overfits the training data. As a result, an *overfitting* model is sensitive to small fluctuations in the data. If a model suffers from high variance, we often say it is *overfitting* and lack of *flexibility* as introduced in (Section 4.4).

Evaluating Estimators in Code

Since the standard deviation of an estimator has been implementing in MXNet by simply calling `a.std()` for a `ndarray` "a", we will skip it but implement the statistical bias and the mean squared error in MXNet.

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(np.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(np.mean(np.square(data - true_theta)))
```

To illustrate the equation of the bias-variance trade-off, let's simulate of normal distribution $\mathcal{N}(\theta, \sigma^2)$ with $10,000$ samples. Here, we use a $\theta = 1$ and $\sigma = 4$. As the estimator is a function of the given samples, here we use the mean of the samples as an estimator for true θ in this normal distribution $\mathcal{N}(\theta, \sigma^2)$.

```
theta_true = 1
sigma = 4
sample_length = 10000
samples = np.random.normal(theta_true, sigma, sample_length)
theta_est = np.mean(samples)
theta_est
```

```
array(0.9503336)
```

Let's validate the trade-off equation by calculating the summation of the squared bias and the variance of our estimator. First, calculate the MSE of our estimator.

```
mse(samples, theta_true)
```

```
array(15.781996)
```

Next, we calculate $\text{Var}(\hat{\theta}_n)+[\text{bias}(\hat{\theta}_n)]^2$ as below. As you can see, the two values agree to numerical precision.

```
bias = stat_bias(theta_true, theta_est)
np.square(samples.std()) + np.square(bias)
```

```
array(15.781995)
```

17.9.2 Conducting Hypothesis Tests

The most commonly encountered topic in statistical inference is hypothesis testing. While hypothesis testing was popularized in the early 20th century, the first use can be traced back to John Arbuthnot in the 1700s. John tracked 80-year birth records in London and concluded that more men were born than women each year. Following that, the modern significance testing is the intelligence heritage by Karl Pearson who invented p-value and Pearson's chi-squared test), William Gosset who is the father of Student's t-distribution, and Ronald Fisher who initialed the null hypothesis and the significance test.

A *hypothesis test* is a way of evaluating some evidence against the default statement about a population. We refer the default statement as the *null hypothesis* H_0, which we try to reject using the observed data. Here, we use H_0 as a starting point for the statistical significance testing. The *alternative hypothesis* H_A (or H_1) is a statement that is contrary to the null hypothesis. A null hypothesis is often stated in a declarative form which posits a relationship between variables. It should reflect the brief as explicit as possible, and be testable by statistics theory.

Imagine you are a chemist. After spending thousands of hours in the lab, you develop a new medicine which can dramatically improve one's ability to understand math. To show its magic power, you need to test it. Naturally, you may need some volunteers to take the medicine and see whether it can help them learn math better. How do you get started?

First, you will need carefully random selected two groups of volunteers, so that there is no difference between their math understanding ability measured by some metrics. The two groups are commonly referred to as the test group and the control group. The *test group* (or *treatment group*) is a group of individuals who will experience the medicine, while the *control group* represents the group of users who are set aside as a benchmark, i.e., identical environment setups except taking this medicine. In this way, the influence of all the variables are minimized, except the impact of the independent variable in the treatment.

Second, after a period of taking the medicine, you will need to measure the two groups' math understanding by the same metrics, such as letting the volunteers do the same tests after learning a new math formula. Then, you can collect their performance and compare the results. In this case, our null hypothesis will be that there is no difference between the two groups, and our alternate will be that there is.

This is still not fully formal. There are many details you have to think of carefully. For example, what is the suitable metrics to test their math understanding ability? How many volunteers for your test so you can be confident to claim the effectiveness of your medicine? How long should

you run the test? How do you decided if there is a difference between the two groups? Do you care about the average performance only, or do you also the range of variation of the scores. And so on.

In this way, hypothesis testing provides framework for experimental design and reasoning about certainty in observed results. If we can now show that the null hypothesis is very unlikely to be true, we may reject it with confidence.

To complete the story of how to work with hypothesis testing, we need to now introduce some additional terminology and make some of our concepts above formal.

Statistical Significance

The *statistical significance* measures the probability of erroneously reject the null hypothesis, H_0, when it should not be rejected, i.e.,

$$\text{statistical significance } = 1 - \alpha = P(\text{reject } H_0 \mid H_0 \text{ is true}). \tag{17.9.5}$$

It is also referred to as the *type I error* or *false positive*. The α, is called as the *significance level* and its commonly used value is 5%, i.e., $1 - \alpha = 95\%$. The level of statistical significance level can be explained as the level of risk that we are willing to take, when we reject a true null hypothesis.

Fig. 17.9.1 shows the the observations' values and probability of a given normal distribution in a two-sample hypothesis test. If the observation data point is located outsides the 95% threshold, it will be a very unlikely observation under the null hypothesis assumption. Hence, there might be something wrong with the null hypothesis and we will reject it.

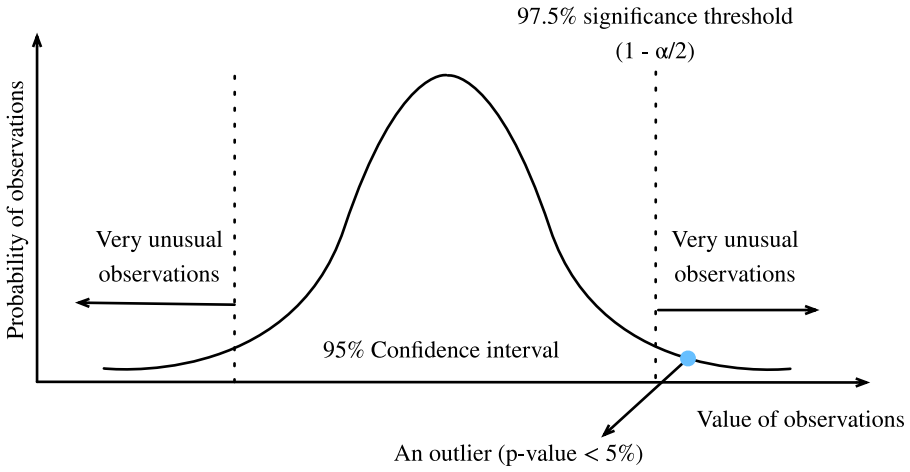


Fig. 17.9.1: Statistical significance.

Statistical Power

The *statistical power* (or *sensitivity*) measures the probability of reject the null hypothesis, H_0, when it should be rejected, i.e.,

$$\text{statistical power } = P(\text{reject } H_0 \mid H_0 \text{ is false}). \tag{17.9.6}$$

Recall that a *type I error* is error caused by rejecting the null hypothesis when it is true, whereas a *type II error* is resulted from failing to reject the null hypothesis when it is false. A type II error is usually denoted as β, and hence the corresponding statistical power is $1 - \beta$.

Intuitively, statistical power can be interpreted as how likely our test will detect a real discrepancy of some minimum magnitude at a desired statistical significance level. 80% is a commonly used statistical power threshold. The higher the statistical power, the more likely we are to detect true differences.

One of the most common uses of statistical power is in determining the number of samples needed. The probability you reject the null hypothesis when it is false depends on the degree to which it is false (known as the *effect size*) and the number of samples you have. As you might expect, small effect sizes will require a very large number of samples to be detectable with high probability. While beyond the scope of this brief appendix to derive in detail, as an example, want to be able to reject a null hypothesis that our sample came from a mean zero variance one Gaussian, and we believe that our sample's mean is actually close to one, we can do so with acceptable error rates with a sample size of only 8. However, if we think our sample population true mean is close to 0.01, then we'd need a sample size of nearly 80000 to detect the difference.

We can imagine the power as a water filter. In this analogy, a high power hypothesis test is like a high quality water filtration system that will reduce harmful substances in the water as much as possible. On the other hand, a smaller discrepancy is like a low quality water filter, where some relative small substances may easily escape from the gaps. Similarly, if the statistical power is not of enough high power, then the test may not catch the smaller discrepancy.

Test Statistic

A *test statistic* $T(x)$ is a scalar which summarizes some characteristic of the sample data. The goal of defining such a statistic is that it should allow us to distinguish between different distributions and conduct our hypothesis test. Thinking back to our chemist example, if we wish to show that one population performs better than the other, it could be reasonable to take the mean as the test statistic. Different choices of test statistic can lead to statistical test with drastically different statistical power.

Often, $T(X)$ (the distribution of the test statistic under our null hypothesis) will follow, at least approximately, a common probability distribution such as a normal distribution when considered under the null hypothesis. If we can derive explicitly such a distribution, and then measure our test statistic on our dataset, we can safely reject the null hypothesis if our statistic is far outside the range that we would expect. Making this quantitative leads us to the notion of p-values.

p-value

The *:math:`p`-value* (or the *probability value*) is the probability that $T(X)$ is at least as extreme as the observed test statistic $T(x)$ assuming that the null hypothesis is *true*, i.e.,

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \tag{17.9.7}$$

If the p-value is smaller than or equal to a pre-defined and fixed statistical significance level α, we may reject the null hypothesis. Otherwise, we will conclude that we are lack of evidence to reject the null hypothesis. For a given population distribution, the *region of rejection* will be the interval contained of all the points which has a p-value smaller than the statistical significance level α.

One-side Test and Two-sided Test

Normally there are two kinds of significance test: the one-sided test and the two-sided test. The *one-sided test* (or *one-tailed test*) is applicable when the null hypothesis and the alternative hypothesis only have one direction. For example, the null hypothesis may state that the true parameter θ is less than or equal to a value c. The alternative hypothesis would be that θ is greater than c. That is, the region of rejection is on only one side of the sampling distribution. Contrary to the one-sided test, the *two-sided test* (or *two-tailed test*) is applicable when the region of rejection is on both sides of the sampling distribution. An example in this case may have a null hypothesis state that the true parameter θ is equal to a value c. The alternative hypothesis would be that θ is not equal to c.

General Steps of Hypothesis Testing

After getting familiar with the above concepts, let's go through the general steps of hypothesis testing.

1. State the question and establish a null hypotheses H_0.
2. Set the statistical significance level α and a statistical power $(1 - \beta)$.
3. Obtain samples through experiments. The number of samples needed will depend on the statistical power, and the expected effect size.
4. Calculate the test statistic and the p-value.
5. Make the decision to keep or reject the null hypothesis based on the p-value and the statistical significance level α.

To conduct a hypothesis test, we start by defining a null hypothesis and a level of risk that we are willing to take. Then we calculate the test statistic of the sample, taking an extreme value of the test statistic as evidence against the null hypothesis. If the test statistic falls within the reject region, we may reject the null hypothesis in favor of the alternative.

Hypothesis testing is applicable in a variety of scenarios such as the clinical trails and A/B testing.

17.9.3 Constructing Confidence Intervals

When estimating the value of a parameter θ, point estimators like $\hat{\theta}$ are of limited utility since they contain no notion of uncertainty. Rather, it would be far better if we could produce an interval that would contain the true parameter θ with high probability. If you were interested in such ideas a century ago, then you would have been excited to read "Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability" by Jerzy Neyman (Neyman, 1937), who first introduced the concept of confidence interval in 1937.

To be useful, a confidence interval should be as small as possible for a given degree of certainty. Let's see how to derive it.

Definition

Mathematically, a *confidence interval* for the true parameter θ is an interval C_n that computed from the sample data such that

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{17.9.8}$$

Here $\alpha \in (0, 1)$, and $1 - \alpha$ is called the *confidence level* or *coverage* of the interval. This is the same α as the significance level as we discussed about above.

Note that (17.9.8) is about variable C_n, not about the fixed θ. To emphasize this, we write $P_\theta(C_n \ni \theta)$ rather than $P_\theta(\theta \in C_n)$.

Interpretation

It is very tempting to interpret a 95% confidence interval as an interval where you can be 95% sure the true parameter lies, however this is sadly not true. The true parameter is fixed, and it is the interval that is random. Thus a better interpretation would be to say that if you generated a large number of confidence intervals by this procedure, 95% of the generated intervals would contain the true parameter.

This may seem pedantic, but it can have real implications for the interpretation of the results. In particular, we may satisfy (17.9.8) by constructing intervals that we are *almost certain* do not contain the true value, as long as we only do so rarely enough. We close this section by providing three tempting but false statements. An in-depth discussion of these points can be found in (Morey et al., 2016).

- **Fallacy 1**. Narrow confidence intervals mean we can estimate the parameter precisely.
- **Fallacy 2**. The values inside the confidence interval are more likely to be the true value than those outside the interval.
- **Fallacy 3**. The probability) that a particular observed 95% confidence interval contains the true value is 95%.

Sufficed to say, confidence intervals are subtle objects. However, if you keep the interpretation clear, they can be powerful tools.

A Gaussian Example

Let's discuss the most classical example, the confidence interval for the mean of a Gaussian of unknown mean and variance. Suppose we collect n samples $\{x_i\}_{i=1}^n$ from our Gaussian $\mathcal{N}(\mu, \sigma^2)$. We can compute estimators for the mean and standard deviation by taking

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^n x_i \text{ and } \hat{\sigma}_n^2 = \frac{1}{n-1}\sum_{i=1}^n (x_i - \hat{\mu})^2. \tag{17.9.9}$$

If we now consider the random variable

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}}, \tag{17.9.10}$$

we obtain a random variable following a well-known distribution called the *Student's t-distribution on $n-1$ degrees of freedom*.

This distribution is very well studied, and it is known, for instance, that as $n \to \infty$, it is approximately a standard Gaussian, and thus by looking up values of the Gaussian c.d.f. in a table, we may conclude that the value of T is in the interval $[-1.96, 1.96]$ at least 95% of the time. For finite values of n, the interval needs to be somewhat larger, but are well known and precomputed in tables.

Thus, we may conclude that for large n,

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \tag{17.9.11}$$

Rearranging this by multiplying both sides by $\hat{\sigma}_n/\sqrt{n}$ and then adding $\hat{\mu}_n$, we obtain

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \tag{17.9.12}$$

Thus we know that we have found our 95% confidence interval:

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{17.9.13}$$

It is safe to say that (17.9.13) is one of the most used formula in statistics. Let's close our discussion of statistics by implementing it. For simplicity, we assume we are in the asymptotic regime. Small values of N should include the correct value of `t_star` obtained either programmatically or from a t-table.

```
# Number of samples
N = 1000

# Sample dataset
samples = np.random.normal(loc=0, scale=1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = np.mean(samples)
sigma_hat = samples.std(ddof=1)
(mu_hat - t_star*sigma_hat/np.sqrt(N), mu_hat + t_star*sigma_hat/np.sqrt(N))
```

```
(array(-0.07853346), array(0.04412608))
```

Summary

- Statistics focuses on inference problems, whereas deep learning emphasizes on making accurate predictions without explicitly programming and understanding.
- There are three common statistics inference methods: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals.
- There are three most common estimators: statistical bias, standard deviation, and mean square error.
- A confidence interval is an estimated range of a true population parameter that we can construct by given the samples.
- Hypothesis testing is a way of evaluating some evidence against the default statement about a population.

Exercises

1. Let $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Unif}(0, \theta)$, where "iid" stands for *independent and identically distributed*. Consider the following estimators of θ:

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{17.9.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{17.9.15}$$

   - Find the statistical bias, standard deviation, and mean square error of $\hat{\theta}$.
   - Find the statistical bias, standard deviation, and mean square error of $\tilde{\theta}$.
   - Which estimator is better?

2. For our chemist example in introduction, can you derive the 5 steps to conduct a two-sided hypothesis testing? Given the statistical significance level $\alpha = 0.05$ and the statistical power $1 - \beta = 0.8$.

3. Run the confidence interval code with $N = 2$ and $\alpha = 0.5$ for 100 independently generated dataset, and plot the resulting intervals (in this case `t_star = 1.0`). You will see several very short intervals which are very far from containing the true mean 0. Does this contradict the interpretation of the confidence interval? Do you feel comfortable using short intervals to indicate high precision estimates?

17.10 Information Theory

The universe is overflowing with information. Information provides a common language across disciplinary rifts: from Shakespeare's Sonnet to researchers' paper on Cornell ArXiv, from Van Gogh's printing Starry Night to Beethoven's music Symphony No. 5, from the first programming language Plankalkül to the state-of-the-art machine learning algorithms. Everything must follow the rules of information theory, no matter the format. With information theory, we can measure and compare how much information is present in different signals. In this section, we will investigate the fundamental concepts of information theory and applications of information theory in machine learning.

Before we get started, let's outline the relationship between machine learning and information theory. Machine learning aims to extract interesting signals from data and make critical predictions. On the other hand, information theory studies encoding, decoding, transmitting, and manipulating information. As a result, information theory provides fundamental language for discussing the information processing in machine learned systems. For example, many machine learning applications use the cross entropy loss as described in Section 3.4. This loss can be directly derived from information theoretic considerations.

17.10.1 Information

Let's start with the "soul" of information theory: information. *Information* can be encoded in anything with a particular sequence of one or more encoding formats. Suppose that we task ourselves with trying to define a notion of information. What could be are starting point?

Consider the following thought experiment. We have a friend with a deck of cards. They will shuffle the deck, flip over some cards, and tell us statements about the cards. We will try to assess the information content of each statement.

First, they flip over a card and tell us, "I see a card." This provides us with no information at all. We were already certain that this was the case so we hope the information should be zero.

Next, they flip over a card and say, "I see a heart." This provides us some information, but in reality there are only 4 different suits that were possible, each equally likely, so we are not surprised by this outcome. We hope that whatever the measure of information, this event should have low information content.

Next, they flip over a card and say, "This is the 3 of spades." This is more information. Indeed there were 52 equally likely possible outcomes, and our friend told us which one it was. This should be a medium amount of information.

Let's take this to the logical extreme. Suppose that finally they flip over every card from the deck and read off the entire sequence of the shuffled deck. There are $52!$ different orders to the deck, again all equally likely, so we need a lot of information to know which one it is.

Any notion of information we develop must conform to this intuition. Indeed, in the next sections we will learn how to compute that these events have 0 bits, 2 bits, 5.7 bits, and 225.6 bits of information respectively.

If we read through these thought experiments, we see a natural idea. As a starting point, rather than caring about the knowledge, we may build off the idea that information represents the degree of surprise or the abstract possibility of the event. For example, if we want to describe an unusual event, we need a lot information. For a common event, we may not need much information.

In 1948, Claude E. Shannon published *A Mathematical Theory of Communication* (Shannon, 1948) establishing the theory of information. In his book, Shannon introduced the concept of information entropy for the first time. We will begin our journey here.

Self-information

Since information embodies the abstract possibility of an event, how do we map the possibility to the number of bits? Shannon introduced the terminology *bit* as the unit of information, which was originally created by John Tukey. So what is a "bit" and why do we use it to measure information? Historically, an antique transmitter can only send or receive two types of code: 0 and 1. Indeed, binary encoding is still in common use on all modern digital computers. In this way, any information is encoded by a series of 0 and 1. And hence, a series of binary digits of length n contains n bits of information.

Now, suppose that for any series of codes, each 0 or 1 occurs with a probability of $\frac{1}{2}$. Hence, an event X with a series of codes of length n, occurs with a probability of $\frac{1}{2^n}$. At the same time, as we mentioned before, this series contains n bits of information. So, can we generalize to a math function which can transfer the probability p to the number of bits? Shannon gave the answer by defining *self-information*

$$I(X) = -\log_2(p), \tag{17.10.1}$$

as the *bits* of information we have received for this event X. Note that we will always use base-2 logarithms in this section. For the sake of simplicity, the rest of this section will omit the subscript 2 in the logarithm notation, i.e., $\log(.)$ always refers to $\log_2(.)$. For example, the code "0010" has a self-information

$$I(``0010") = -\log(p(``0010")) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits}. \tag{17.10.2}$$

We can calculate self information in MXNet as shown below. Before that, let's first import all the necessary packages in this section.

```
from mxnet import np
from mxnet.metric import NegativeLogLikelihood
from mxnet.ndarray import nansum
import random

def self_information(p):
    return -np.log2(p)

self_information(1/64)
```

```
6.0
```

17.10.2 Entropy

As self-information only measures the information of a single discrete event, we need a more generalized measure for any random variable of either discrete or continuous distribution.

Motivating Entropy

Let's try to get specific about what we want. This will be an informal statement of what are known as the *axioms of Shannon entropy*. It will turn out that the following collection of common-sense statements force us to a unique definition of information. A formal version of these axioms, along with several others may be found in (Csiszar, 2008).

1. The information we gain by observing a random variable does not depend on what we call the elements, or the presence of additional elements which have probability zero.
2. The information we gain by observing two random variables is no more than the sum of the information we gain by observing them separately. If they are independent, then it is exactly the sum.
3. The information gained when observing (nearly) certain events is (nearly) zero.

While proving this fact is beyond the scope of our text, it is important to know that this uniquely determines the form that entropy must take. The only ambiguity that these allow is in the choice of fundamental units, which is most often normalized by making the choice we saw before that the information provided by a single fair coin flip is one bit.

Definition

For any random variable X that follows a probability distribution P with a probability density function (p.d.f.) or a probability mass function (p.m.f.) $p(x)$, we measure the expected amount of information through *entropy* (or *Shannon entropy*)

$$H(X) = -E_{x \sim P}[\log p(x)]. \tag{17.10.3}$$

To be specific, if X is discrete,

$$H(X) = -\sum_i p_i \log p_i, \text{ where } p_i = P(X_i). \tag{17.10.4}$$

Otherwise, if X is continuous, we also refer entropy as *differential entropy*

$$H(X) = -\int_x p(x) \log p(x) \, dx. \tag{17.10.5}$$

In MXNet, we can define entropy as below.

```
def entropy(p):
    entropy = - p * np.log2(p)
    # nansum will sum up the non-nan number
    out = nansum(entropy.as_nd_ndarray())
    return out

entropy(np.array([0.1, 0.5, 0.1, 0.3]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

Interpretations

You may be curious: in the entropy definition (17.10.3), why do we use an expectation of a negative logarithm? Here are some intuitions.

First, why do we use a *logarithm* function log? Suppose that $p(x) = f_1(x)f_2(x)\ldots, f_n(x)$, where each component function $f_i(x)$ is independent from each other. This means that each $f_i(x)$ contributes independently to the total information obtained from $p(x)$. As discussed above, we want the entropy formula to be additive over independent random variables. Luckily, log can naturally turn a product of probability distributions to a summation of the individual terms.

Next, why do we use a *negative* log? Intuitively, more frequent events should contain less information than less common events, since we often gain more information from an unusual case than from an ordinary one. However, log is monotonically increasing with the probabilities, and indeed negative for all values in $[0, 1]$. We need to construct a monotonically decreasing relationship between the probability of events and their entropy, which will ideally be always positive (for nothing we observe should force us to forget what we have known). Hence, we add a negative sign in front of log function.

Last, where does the *expectation* function come from? Consider a random variable X. We can interpret the self-information ($-\log(p)$) as the amount of *surprise* we have at seeing a particular outcome. Indeed, as the probability approaches zero, the surprise becomes infinite. Similarly, we can interpret The entropy as the average amount of surprise from observing X. For example, imagine that a slot machine system emits statistical independently symbols $s_1, \ldots, s_k$ with probabilities $p_1, \ldots, p_k$ respectively. Then the entropy of this system equals to the average self-information from observing each output, i.e.,

$$H(S) = \sum_i p_i \cdot I(s_i) = -\sum_i p_i \cdot \log p_i. \tag{17.10.6}$$

Properties of Entropy

By the above examples and interpretations, we can derive the following properties of entropy (17.10.3). Here, we refer to X as an event and P as the probability distribution of X.

- Entropy is non-negative, i.e., $H(X) \geq 0, \forall X$.
- If $X \sim P$ with a p.d.f. or a p.m.f. $p(x)$, and we try to estimate P by a new probability distribution Q with a p.d.f. or a p.m.f. $q(x)$, then

  $$H(X) = -E_{x\sim P}[\log p(x)] \leq -E_{x\sim P}[\log q(x)], \text{ with equality if and only if } P = Q. \tag{17.10.7}$$

  Alternatively, $H(X)$ gives a lower bound of the average number of bits needed to encode symbols drawn from P.
- If $X \sim P$, then x conveys the maximum amount of information if it spreads evenly among all possible outcomes. Specifically, if the probability distribution P is discrete with k-class $\{p_1, \ldots, p_k\}$, then

  $$H(X) \leq \log(k), \text{ with equality if and only if } p_i = \frac{1}{k}, \forall x_i. \tag{17.10.8}$$

If P is a continuous random variable, then the story becomes much more complicated. However, if we additionally impose that P is supported on a finite interval (with all values between 0 and 1), then P has the highest entropy if it is the uniform distribution on that interval.

17.10.3 Mutual Information

Previously we defined entropy of a single random variable X, how about the entropy of a pair random variables (X, Y)? We can think of these techniques as trying to answer the following type of question, "What information is contained in X and Y together compared to each separately? Is there redundant information, or is it all unique?"

For the following discussion, we always use (X, Y) as a pair of random variables that follows a joint probability distribution P with a p.d.f. or a p.m.f. $p_{X,Y}(x, y)$, while X and Y follow probability distribution $p_X(x)$ and $p_Y(y)$, respectively.

Joint Entropy

Similar to entropy of a single random variable (17.10.3), we define the *joint entropy* $H(X, Y)$ of a pair random variables (X, Y) as

$$H(X, Y) = -E_{(x,y)\sim P}[\log p_{X,Y}(x, y)]. \tag{17.10.9}$$

Precisely, on the one hand, if (X, Y) is a pair of discrete random variables, then

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \tag{17.10.10}$$

On the other hand, if (X, Y) is a pair of continuous random variables, then we define the *differential joint entropy* as

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y)\ \log p_{X,Y}(x, y)\ dx\ dy. \tag{17.10.11}$$

We can think of (17.10.9) as telling us the total randomness in the pair of random variables. As a pair of extremes, if $X = Y$ are two identical random variables, then the information in the pair is exactly the information in one and we have $H(X, Y) = H(X) = H(Y)$. On the other extreme, if X and Y are independent then $H(X, Y) = H(X) + H(Y)$. Indeed we will always have that the information contained in a pair of random variables is no smaller than the entropy of either random variable and no more than the sum of both.

$$H(X), H(Y) \leq H(X, Y) \leq H(X) + H(Y). \tag{17.10.12}$$

Let's implement joint entropy from scratch in MXNet.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * np.log2(p_xy)
    # nansum will sum up the non-nan number
    out = nansum(joint_ent.as_nd_ndarray())
    return out

joint_entropy(np.array([[0.1, 0.5], [0.1, 0.3]]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

Notice that this is the same *code* as before, but now we interpret it differently as working on the joint distribution of the two random variables.

Conditional Entropy

The joint entropy defined above the amount of information contained in a pair of random variables. This is useful, but often times it is not what we care about. Consider the setting of machine learning. Let's take X to be the random variable (or vector of random variables) that describes the pixel values of an image, and Y to be the random variable which is the class label. X should contain substantial information—a natural image is a complex thing. However, the information contained in Y once the image has been show should be low. Indeed, the image of a digit should already contain the information about what digit it is unless the digit is illegible. Thus, to continue to extend our vocabulary of information theory, we need to be able to reason about the information content in a random variable conditional on another.

In the probability theory, we saw the definition of the *conditional probability* to measure the relationship between variables. We now want to analogously define the *conditional entropy* $H(Y \mid X)$. We can write this as

$$H(Y \mid X) = -E_{(x,y)\sim P}[\log p(y \mid x)], \tag{17.10.13}$$

where $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ is the conditional probability. Specifically, if (X, Y) is a pair of discrete random variables, then

$$H(Y \mid X) = -\sum_x \sum_y p(x,y) \log p(y \mid x). \tag{17.10.14}$$

If (X, Y) is a pair of continuous random variables, then the *differential joint entropy* is similarly defined as

$$H(Y \mid X) = -\int_x \int_y p(x,y)\ \log p(y \mid x)\ dx\ dy. \tag{17.10.15}$$

It is now natural to ask, how does the *conditional entropy* $H(Y \mid X)$ relate to the entropy $H(X)$ and the joint entropy $H(X, Y)$? Using the definitions above, we can express this cleanly:

$$H(Y \mid X) = H(X,Y) - H(X). \tag{17.10.16}$$

This has an intuitive interpretation: the information in Y given X ($H(Y \mid X)$) is the same as the information in both X and Y together ($H(X, Y)$) minus the information already contained in X. This gives us the information in Y which is not also represented in X.

Now, let's implement conditional entropy (17.10.13) from scratch in MXNet.

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * np.log2(p_y_given_x)
    # nansum will sum up the non-nan number
    out = nansum(cond_ent.as_nd_ndarray())
    return out

conditional_entropy(np.array([[0.1, 0.5], [0.2, 0.3]]), np.array([0.2, 0.8]))
```

```
[0.8635472]
<NDArray 1 @cpu(0)>
```

Mutual Information

Given the previous setting of random variables (X, Y), you may wonder: "Now that we know how much information is contained in Y but not in X, can we similarly ask how much information is shared between X and Y?" The answer will be the *mutual information* of (X, Y), which we will write as $I(X, Y)$.

Rather than diving straight into the formal definition, let's practice our intuition by first trying to derive an expression for the mutual information entirely based on terms we have constructed before. We wish to find the information shared between two random variables. One way we could try to do this is to start with all the information contained in both X and Y together, and then we take off the parts that are not shared. The information contained in both X and Y together is written as $H(X, Y)$. We want to subtract from this the information contained in X but not in Y, and the information contained in Y but not in X. As we saw in the previous section, this is given by $H(X \mid Y)$ and $H(Y \mid X)$ respectively. Thus, we have that the mutual information should be

$$I(X, Y) = H(X, Y) - H(Y \mid X) - H(X \mid Y). \tag{17.10.17}$$

Indeed, this is a valid definition for the mutual information. If we expand out the definitions of these terms and combine them, a little algebra shows that this is the same as

$$I(X, Y) = -E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)} \right\}. \tag{17.10.18}$$

We can summarize all of these relationships in image Fig. 17.10.1. It is an excellent test of intuition to see why the following statements are all also equivalent to $I(X, Y)$.

- $H(X) - H(X \mid Y)$
- $H(Y) - H(Y \mid X)$
- $H(X) + H(Y) - H(X, Y)$

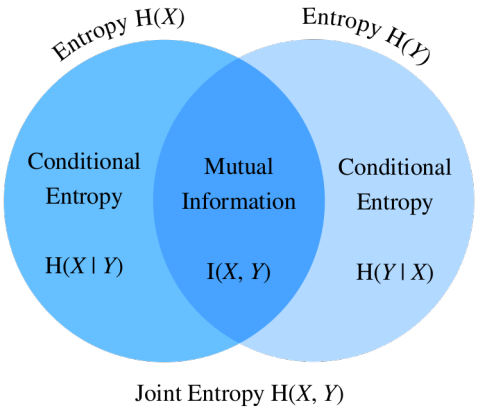


Fig. 17.10.1: Mutual information's relationship with joint entropy and conditional entropy.

In many ways we can think of the mutual information (17.10.18) as principled extension of correlation coefficient we saw in Section 17.6. This allows us to ask not only for linear relationships

between variables, but for the maximum information shared between the two random variables of any kind.

Now, let's implement mutual information from scratch.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = -p_xy * np.log2(p)
    # nansum will sum up the non-nan number
    out = nansum(mutual.as_nd_ndarray())
    return out

mutual_information(np.array([[0.1, 0.5], [0.1, 0.3]]),
                   np.array([0.2, 0.8]),
                   np.array([[0.75, 0.25]]))
```

```
[-0.71946025]
<NDArray 1 @cpu(0)>
```

Properties of Mutual Information

Rather than memorizing the definition of mutual information (17.10.18), you only need to keep in mind its notable properties:

- Mutual information is symmetric, i.e., $I(X, Y) = I(Y, X)$.
- Mutual information is non-negative, i.e., $I(X, Y) \geq 0$.
- $I(X, Y) = 0$ if and only if X and Y are independent. For example, if X and Y are independent, then knowing Y does not give any information about X and vice versa, so their mutual information is zero.
- Alternatively, if X is an invertible function of Y, then Y and X share all information and

$$I(X, Y) = H(Y) = H(X). \tag{17.10.19}$$

Pointwise Mutual Information

When we worked with entropy at the beginning of this chapter, we were able to provide an interpretation of $-\log(p_X(x))$ as how *surprised* we were with the particular outcome. We may give a similar interpretation to the logarithmic term in the mutual information, which is often referred to as the *pointwise mutual information*:

$$\text{pmi}(x, y) = \log \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}. \tag{17.10.20}$$

We can think of (17.10.20) as measuring how much more or less likely the specific combination of outcomes x and y are compared to what we would expect for independent random outcomes. If it is large and positive, then these two specific outcomes occur much more frequently than they would compared to random chance (*note*: the denominator is $p_X(x)p_Y(y)$ which is the probability of the two outcomes were independent), whereas if it is large and negative it represents the two outcomes happening far less than we would expect by random chance.

This allows us to interpret the mutual information (17.10.18) as the average amount that we were surprised to see two outcomes occurring together compared to what we would expect if they were independent.

Applications of Mutual Information

Mutual information may be a little abstract in it pure definition, so how does it related to machine learning? In natural language processing, one of the most difficult problems is the *ambiguity resolution*, or the issue of the meaning of a word being unclear from context. For example, recently a headline in the news reported that "Amazon is on fire". You may wonder whether the company Amazon has a building on fire, or the Amazon rain forest is on fire.

In this case, mutual information can help us resolve this ambiguity. We first find the group of words that each has a relatively large mutual information with the company Amazon, such as e-commerce, technology, and online. Second, we find another group of words that each has a relatively large mutual information with the Amazon rain forest, such as rain, forest, and tropical. When we need to disambiguate "Amazon", we can compare which group has more occurrence in the context of the word Amazon. In this case the article would go on to describe the forest, and make the context clear.

17.10.4 Kullback–Leibler Divergence

As what we have discussed in Section 2.3, we can use norms to measure distance between two points in space of any dimensionality. We would like to be able to do a similar task with probability distributions. There are many ways to go about this, but information theory provides one of the nicest. We now explore the *Kullback–Leibler (KL) divergence*, which provides a way to measure if two distributions are close together or not.

Definition

Given a random variable X that follows the probability distribution P with a p.d.f. or a p.m.f. $p(x)$, and we estimate P by another probability distribution Q with a p.d.f. or a p.m.f. $q(x)$. Then the *Kullback–Leibler (KL) divergence* (or *relative entropy*) between P and Q is

$$D_{\mathrm{KL}}(P\|Q) = E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]. \tag{17.10.21}$$

As with the pointwise mutual information (17.10.20), we can again provide an interpretation of the logarithmic term: $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$ will be large and positive if we see x far more often under P than we would expect for Q, and large and negative if we see the outcome far less than expected. In this way, we can interpret it as our *relative* surprise at observing the outcome compared to how surprised we would be observing it from our reference distribution.

In MXNet, let's implement the KL divergence from Scratch.

```
def kl_divergence(p, q):
    kl = p * np.log2(p / q)
    out = nansum(kl.as_nd_ndarray())
    return out.abs().asscalar()
```

KL Divergence Properties

Let's take a look at some properties of the KL divergence (17.10.21).

- KL divergence is non-symmetric, i.e.,

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P), \text{ if } P \neq Q. \tag{17.10.22}$$

- KL divergence is non-negative, i.e.,

$$D_{\mathrm{KL}}(P\|Q) \geq 0. \tag{17.10.23}$$

  Note that the equality holds only when $P = Q$.

- If there exists an x such that $p(x) > 0$ and $q(x) = 0$, then $D_{\mathrm{KL}}(P\|Q) = \infty$.
- There is a close relationship between KL divergence and mutual information. Besides the relationship shown in Fig. 17.10.1, $I(X, Y)$ is also numerically equivalent with the following terms:

  1. $D_{\mathrm{KL}}(P(X, Y) \parallel P(X)P(Y))$;
  2. $E_Y\{D_{\mathrm{KL}}(P(X \mid Y) \parallel P(X))\}$;
  3. $E_X\{D_{\mathrm{KL}}(P(Y \mid X) \parallel P(Y))\}$.

  For the first term, we interpret mutual information as the KL divergence between $P(X, Y)$ and the product of $P(X)$ and $P(Y)$, and thus is a measure of how different the joint distribution is from the distribution if they were independent. For the second term, mutual information tells us the average reduction in uncertainty about Y that results from learning the value of the X's distribution. Similarly to the third term.

Example

Let's go through a toy example to see the non-symmetry explicitly.

First, let's generate and sort three `ndarrays` of length $10,000$: an objective `ndarray` p which follows a normal distribution $N(0, 1)$, and two candidate `ndarrays` q_1 and q_2 which follow normal distributions $N(-1, 1)$ and $N(1, 1)$ respectively.

```
random.seed(1)

nd_length = 10000
p = np.random.normal(loc=0, scale=1, size=(nd_length, ))
q1 = np.random.normal(loc=-1, scale=1, size=(nd_length, ))
q2 = np.random.normal(loc=1, scale=1, size=(nd_length, ))

p = np.array(sorted(p.asnumpy()))
q1 = np.array(sorted(q1.asnumpy()))
q2 = np.array(sorted(q2.asnumpy()))
```

Since q_1 and q_2 are symmetric with respect to the y-axis (i.e., $x = 0$), we expect a similar value of KL divergence between $D_{\mathrm{KL}}(p\|q_1)$ and $D_{\mathrm{KL}}(p\|q_2)$. As you can see below, there is only a 1% off between $D_{\mathrm{KL}}(p\|q_1)$ and $D_{\mathrm{KL}}(p\|q_2)$.

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
(8470.638, 8664.999, 2.268504302642314)
```

In contrast, you may find that $D_{\mathrm{KL}}(q_2 \| p)$ and $D_{\mathrm{KL}}(p \| q_2)$ are off a lot, with around 40% off as shown below.

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(13536.835, 43.88678828000115)
```

17.10.5 Cross Entropy

If you are curious about applications of information theory in deep learning, here is a quick example. We define the true distribution P with probability distribution $p(x)$, and the estimated distribution Q with probability distribution $q(x)$, and we will use them in the rest of this section.

Say we need to solve a binary classification problem based on given n data points $\{x_1, \ldots, x_n\}$. Assume that we encode 1 and 0 as the positive and negative class label y_i respectively, and our neural network is parameterized by θ. If we aim to find a best θ so that $\hat{y}_i = p_\theta(y_i \mid x_i)$, it is natural to apply the maximum log-likelihood approach as was seen in Section 17.7. To be specific, for true labels y_i and predictions $\hat{y}_i = p_\theta(y_i \mid x_i)$, the probability to be classified as positive is $\pi_i = p_\theta(y_i = 1 \mid x_i)$. Hence, the log-likelihood function would be

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\ &= \sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{17.10.24}$$

Maximizing the log-likelihood function $l(\theta)$ is identical to minimizing $-l(\theta)$, and hence we can find the best θ from here. To generalize the above loss to any distributions, we also called $-l(\theta)$ the *cross entropy loss* $\mathrm{CE}(y, \hat{y})$, where y follows the true distribution P and $\hat{y}$ follows the estimated distribution Q.

This was all derived by working from the maximum likelihood point of view. However, if we look closely we can see that terms like $\log(\pi_i)$ have entered into our computation which is a solid indication that we can understand the expression from an information theoretic point of view.

Formal Definition

Like KL divergence, for a random variable X, we can also measure the divergence between the estimating distribution Q and the true distribution P via *cross entropy*,

$$\mathrm{CE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{17.10.25}$$

By using properties of entropy discussed above, we can also interpret it as the summation of the entropy $H(P)$ and the KL divergence between P and Q, i.e.,

$$\mathrm{CE}(P, Q) = H(P) + D_{\mathrm{KL}}(P \| Q). \tag{17.10.26}$$

In MXNet, we can implement the cross entropy loss as below.

```
def cross_entropy(y_hat, y):
    ce = -np.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Now define two `ndarrays` for the labels and predictions, and calculate the cross entropy loss of them.

```
labels = np.array([0, 2])
preds = np.array([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
array(0.94856)
```

Properties

As alluded in the beginning of this section, cross entropy (17.10.25) can be used to define a loss function in the optimization problem. It turns out that the following are equivalent:

1. Maximizing predictive probability of Q for distribution P, (i.e., $E_{x \sim P}[\log(q(x))]$);
2. Minimizing cross entropy $\mathrm{CE}(P, Q)$;
3. Minimizing the KL divergence $D_{\mathrm{KL}}(P \| Q)$.

The definition of cross entropy indirectly proves the equivalent relationship between objective 2 and objective 3, as long as the entropy of true data $H(P)$ is constant.

Cross Entropy as An Objective Function of Multi-class Classification

If we dive deep into the classification objective function with cross entropy loss CE, we will find minimizing CE is equivalent to maximizing the log-likelihood function L.

To begin with, suppose that we are given a dataset with n samples, and it can be classified into k-classes. For each data point i, we represent any k-class label $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ by *one-hot encoding*. To be specific, if the data point i belongs to class j, then we set the j-th entry to 1, and all other components to 0, i.e.,

$$y_{ij} = \begin{cases} 1 & j \in J; \\ 0 & \text{otherwise.} \end{cases} \tag{17.10.27}$$

For instance, if a multi-class classification problem contains three classes A, B, and C, then the labels $\mathbf{y}_i$ can be encoded in $\{A : (1, 0, 0); B : (0, 1, 0); C : (0, 0, 1)\}$.

Assume that our neural network is parameterized by θ. For true label vectors $\mathbf{y}_i$ and predictions

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^{k} y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{17.10.28}$$

Hence, the *cross entropy loss* would be

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{17.10.29}$$

On the other side, we can also approach the problem through maximum likelihood estimation. To begin with, let's quickly introduce a k-class multinoulli distribution. It is an extension of the Bernoulli distribution from binary class to multi-class. If a random variable $\mathbf{z} = (z_1, \ldots, z_k)$ follows a k-class *multinoulli distribution* with probabilities $\mathbf{p} = (p_1, \ldots, p_k)$, i.e.,

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \text{Multi}(p_1, \ldots, p_k), \text{ where } \sum_{i=1}^{k} p_i = 1, \tag{17.10.30}$$

then the joint probability mass function(p.m.f.) of $\mathbf{z}$ is

$$\mathbf{p}^{\mathbf{z}} = \prod_{j=1}^{k} p_j^{z_j}. \tag{17.10.31}$$

It can be seen that each data point, $\mathbf{y}_i$, is following a k-class multinoulli distribution with probabilities $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$. Therefore, the joint p.m.f. of each data point $\mathbf{y}_i$ is $\pi^{\mathbf{y}_i} = \prod_{j=1}^{k} \pi_j^{y_{ij}}$. Hence, the log-likelihood function would be

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^{n} \prod_{j=1}^{k} \pi_j^{y_{ij}} = \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log \pi_j. \tag{17.10.32}$$

Since in maximum likelihood estimation, we maximizing the objective function $l(\theta)$ by having $\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$. Therefore, for any multi-class classification, maximizing the above log-likelihood function $l(\theta)$ is equivalent to minimizing the CE loss $\text{CE}(y, \hat{y})$.

To test the above proof, let's apply the built-in measure `NegativeLogLikelihood` in MXNet. Using the same `labels` and `preds` as in the earlier example, we will get the same numerical loss as the previous example up to the 5 decimal place.

```
nll_loss = NegativeLogLikelihood()
nll_loss.update(labels.as_nd_ndarray(), preds.as_nd_ndarray())
nll_loss.get()
```

```
('nll-loss', 0.9485599994659424)
```

Summary

- Information theory is a field of study about encoding, decoding, transmitting, and manipulating information.
- Entropy is the unit to measure how much information is presented in different signals.
- KL divergence can also measure the divergence between two distributions.
- Cross Entropy can be viewed as an objective function of multi-class classification. Minimizing cross entropy loss is equivalent to maximizing the log-likelihood function.

Exercises

1. Verify that the card examples from the first section indeed have the claimed entropy.
2. Let's compute the entropy from a few data sources:
   - Assume that you are watching the output generated by a monkey at a typewriter. The monkey presses any of the 44 keys of the typewriter at random (you can assume that it has not discovered any special keys or the shift key yet). How many bits of randomness per character do you observe?
   - Being unhappy with the monkey, you replaced it by a drunk typesetter. It is able to generate words, albeit not coherently. Instead, it picks a random word out of a vocabulary of $2,000$ words. Moreover, assume that the average length of a word is 4.5 letters in English. How many bits of randomness do you observe now?
   - Still being unhappy with the result, you replace the typesetter by a high quality language model. These can currently obtain perplexity numbers as low as 15 points per character. The perplexity is defined as a length normalized probability, i.e.,

     $$PPL(x) = [p(x)]^{1/\text{length(x)}}. \tag{17.10.33}$$

     How many bits of randomness do you observe now?
3. Explain intuitively why $I(X,Y) = H(X) - H(X|Y)$. Then, show this is true by expressing both sides as an expectation with respect to the joint distribution.
4. What is the KL Divergence between the two Gaussian distributions $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$?