

Double-click (or enter) to edit

COSE474-2024F: Deep Learning

✓ 2.1. Data Manipulation

✓ 2.1.1. Getting Started

```
import torch
```

```
x = torch.arange(12, dtype=torch.float32)
x
```

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

```
x.numel()
```

```
→ 12
```

```
x.shape
```

```
→ torch.Size([12])
```

```
X = x.reshape(3,4)
X
```

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

```
torch.zeros((2, 3, 4))
```

```
→ tensor([[[[0., 0., 0., 0.],
            [0., 0., 0., 0.],
            [0., 0., 0., 0.]],

           [[0., 0., 0., 0.],
            [0., 0., 0., 0.],
            [0., 0., 0., 0.]]]])
```

```
torch.ones((2, 3, 4))
```

```
→ tensor([[[[1., 1., 1., 1.],
            [1., 1., 1., 1.],
            [1., 1., 1., 1.]],

           [[1., 1., 1., 1.],
            [1., 1., 1., 1.],
            [1., 1., 1., 1.]]]])
```

```
torch.randn(3, 4)
```

```
→ tensor([[ 0.9408, -0.7046, -0.9199,  0.8976],
          [-0.0096,  0.0765,  0.7844, -1.0034],
          [-0.9448,  2.1987, -0.5243,  0.1473]])
```

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
→ tensor([[2, 1, 4, 3],
          [1, 2, 3, 4],
          [4, 3, 2, 1]])
```

Double-click (or enter) to edit

✓ 2.1.2. Indexing and Slicing

```
X[-1], X[1:3]
```

```
↩ tensor([ 8.,  9., 10., 11.]),
   tensor([[ 4.,  5.,  6.,  7.],
           [ 8.,  9., 10., 11.]])
```

```
X[1, 2] = 17
X
```

```
↩ tensor([[ 0.,  1.,  2.,  3.],
           [ 4.,  5., 17.,  7.],
           [ 8.,  9., 10., 11.]])
```

```
X[:, 2, :] = 12
X
```

```
↩ tensor([[12., 12., 12., 12.],
           [12., 12., 12., 12.],
           [ 8.,  9., 10., 11.]])
```

✓ 2.1.3 Operations

```
torch.exp(x)
```

```
↩ tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
          162754.7969, 162754.7969, 162754.7969, 2980.9580, 8103.0840,
          22026.4648, 59874.1406])
```

Likewise, we denote *binary* scalar operators, which map pairs of real numbers to a (single) real number via the signature $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Given any two vectors \mathbf{u} and \mathbf{v} of the same shape, and a binary operator f , we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i , where c_i , u_i , and v_i are the i^{th} elements of vectors \mathbf{c} , \mathbf{u} , and \mathbf{v} . Here, we produced the vector-valued $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ by *lifting* the scalar function to an elementwise vector operation. The common standard arithmetic operators for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**) have all been *lifted* to elementwise operations for identically-shaped tensors of arbitrary shape.

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
↩ (tensor([ 3.,  4.,  6., 10.]),
   tensor([-1.,  0.,  2.,  6.]),
   tensor([ 2.,  4.,  8., 16.]),
   tensor([0.5000, 1.0000, 2.0000, 4.0000]),
   tensor([ 1.,  4., 16., 64.]])
```

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

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

```
X == Y
```

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

```
X.sum()
```

```
↳ tensor(66.)
```

2.1.4 Broadcasting

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

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

Since a and b are 3×1 and 1×2 matrices, respectively, their shapes do not match up. Broadcasting produces a larger 3×2 matrix by replicating matrix a along the columns and matrix b along the rows before adding them elementwise.

```
a + b
```

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

2.1.5 Saving Memory

Running operations can cause new memory to be allocated to host results. For example, if we write $Y = X + Y$, we dereference the tensor that Y used to point to and instead point Y at the newly allocated memory. We can demonstrate this issue with Python's `id()` function, which gives us the exact address of the referenced object in memory. Note that after we run $Y = Y + X$, `id(Y)` points to a different location. That is because Python first evaluates $Y + X$, allocating new memory for the result and then points Y to this new location in memory.

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
↳ False
```

This might be undesirable for two reasons.

- First, we do not want to run around allocating memory unnecessarily all the time. In machine learning, we often have hundreds of megabytes of parameters and update all of them multiple times per second. Whenever possible, we want to perform these updates in place.
- Second, we might point at the same parameters from multiple variables. If we do not update in place, we must be careful to update all of these references, lest we spring a memory leak or inadvertently refer to stale parameters.

Fortunately, performing in-place operations is easy. We can assign the result of an operation to a previously allocated array Y by using slice notation: $Y[:] =$. To illustrate this concept, we overwrite the values of tensor Z , after initializing it, using `zeros_like`, to have the same shape as Y .

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
↳ id(Z): 134456531408656
   id(Z): 134456531408656
```

If the value of X is not reused in subsequent computations, we can also use $X[:] = X + Y$ or $X += Y$ to reduce the memory overhead of the operation.

```
before = id(X)
X += Y
id(X) == before
```

```
↳ True
```

✓ 2.1.6. Conversion to Other Python Objects

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

✓ 2.1.7. Summary

The tensor class is the main interface for storing and manipulating data in deep learning libraries. Tensors provide a variety of functionalities including construction routines; indexing and slicing; basic mathematics operations; broadcasting; memory-efficient assignment; and conversion to and from other Python objects.

✓ 2.1.8. Exercises

Run the code in this section. Change the conditional statement $X == Y$ to $X < Y$ or $X > Y$, and then see what kind of tensor you can get.

Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
result_lt = X < Y
result_gt = X > Y
```

```
print("X < Y:\n", result_lt)
print("X > Y:\n", result_gt)
```

```
X < Y:
tensor([[ True, False,  True, False],
        [False, False, False, False],
        [False, False, False, False]])
X > Y:
tensor([[False, False, False, False],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]])
```

```
a = torch.arange(6).reshape((2, 3, 1))
b = torch.arange(2).reshape((1, 1, 2))
```

```
result = a + b
```

```
print("Tensor a:\n", a)
print("Tensor b:\n", b)
print("Result of a + b:\n", result)
```

```
Tensor a:
tensor([[[[0],
          [1],
          [2]],

         [[3],
          [4],
          [5]]]])
Tensor b:
tensor([[[[0, 1]]]])
Result of a + b:
tensor([[[[0, 1],
          [1, 2],
          [2, 3]],

         [[3, 4],
```

```
[4, 5],  
[5, 6]])
```

Start coding or [generate](#) with AI.

COSE474-2024F: Deep Learning

2.2 Data Preprocessing

2.2.1. Reading the Dataset


Comma-separated values (CSV) files are ubiquitous for the storing of tabular (spreadsheet-like) data. In them, each line corresponds to one record and consists of several (comma-separated) fields, e.g., "Albert Einstein,March 14 1879,Ulm,Federal polytechnic school,field of gravitational physics". To demonstrate how to load CSV files with pandas, we create a CSV file below [../data/house_tiny.csv](#). This file represents a dataset of homes, where each row corresponds to a distinct home and the columns correspond to the number of rooms (NumRooms), the roof type (RoofType), and the price (Price).

```
import os

os.makedirs(os.path.join('.', 'data'), exist_ok=True)
data_file = os.path.join('.', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write(''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000'')
```

```
import pandas as pd


data = pd.read_csv(data_file)
print(data)
```



	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000


2.2.2. Data Preparation

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```



	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```



	NumRooms	RoofType_Slate	RoofType_nan
0	3.0	False	True
1	2.0	False	True
2	4.0	True	False
3	3.0	False	True

2.2.3. Conversion to the Tensor Format

```
import torch

X = torch.tensor(inputs.to_numpy(dtype=float))
```

```

y = torch.tensor(targets.to_numpy(dtype=float))
X, y

↳ (tensor([[3., 0., 1.],
           [2., 0., 1.],
           [4., 1., 0.],
           [3., 0., 1.]], dtype=torch.float64),
  tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))

```

✓ 2.2.4. Discussion and Takeaways

- The use of mean imputation is straightforward and useful when you have missing numerical values, but this method assumes that missing values are random.

✓ 2.2.4.1. My own exercise

```

import numpy as np

data = {
    'Name': ['John', 'Sarah', 'Tom', 'Lucy', 'David'],
    'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
    'Math': [85, 78, None, 95, 70],
    'Physics': [90, 85, 80, None, 75],
    'Passed': ['Yes', 'No', 'Yes', 'Yes', 'No']
}

df = pd.DataFrame(data)

df['Math'].fillna(df['Math'].mean(), inplace=True)
df['Physics'].fillna(df['Physics'].mean(), inplace=True)

df_encoded = pd.get_dummies(df, columns=['Gender', 'Passed'])
df_encoded.drop(columns=['Name'], inplace=True)

X = torch.tensor(df_encoded.drop(columns=['Passed_No', 'Passed_Yes']).values.astype(np.float32), dtype=torch.float32)
y = torch.tensor(df_encoded['Passed_Yes'].values.astype(np.float32), dtype=torch.float32)

X

↳ tensor([[85.0000, 90.0000, 0.0000, 1.0000],
          [78.0000, 85.0000, 1.0000, 0.0000],
          [82.0000, 80.0000, 0.0000, 1.0000],
          [95.0000, 82.5000, 1.0000, 0.0000],
          [70.0000, 75.0000, 0.0000, 1.0000]])

y

↳ tensor([1., 0., 1., 1., 0.])

```

- In this exercise, I used mean imputation to fill in the missing scores for each subject. This approach is simple but effective for small datasets.
- I used one-hot encoding to convert the categorical Gender and Passed columns into numerical data.
- Passed_Yes is treated as the label for a binary classification task.

COSE474-2024F-Deep Learning

2.3. Linear Algebra

```
import torch
```

2.3.1. Scalars

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

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

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

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

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

2.3.2. Vectors

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

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

→ tensor([0, 1, 2])

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

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

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

```
x[2]
```

→ tensor(2)

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

```
len(x)
```

```
3
```

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

```
x.shape
```

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

2.3.3. Matrices

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

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

```
:eqlabel: eq_matrix_def
```

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

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

```
A
```

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

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

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

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

```
A.T
```

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

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

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

```
A == A.T
```

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

✓ 2.3.4. Tensors

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

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

```
→ tensor([[[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]],
          [[12, 13, 14, 15],
           [16, 17, 18, 19],
           [20, 21, 22, 23]]])
```

✓ 2.3.5. Basic Properties of Tensor Arithmetic

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

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

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

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

```
A * B
```

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

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

```
→ (tensor([[[ 2,  3,  4,  5],
           [ 6,  7,  8,  9],
           [10, 11, 12, 13]],
          [[14, 15, 16, 17],
           [18, 19, 20, 21],
           [22, 23, 24, 25]]]),
  torch.Size([2, 3, 4]))
```

✓ 2.3.6. Reduction

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

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

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

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

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

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

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

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

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

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

Out: tensor(True)
```

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

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

Out: (tensor(2.5000), tensor(2.5000))
```

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

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

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

2.3.7. Non-Reduction Sum

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

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

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

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

```
A / sum_A

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

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

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

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

2.3.8. Dot Products

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

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

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

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

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

```
tensor(3.)
```

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

2.3.9. Matrix–Vector Products

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

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}^{(1)} \\ \mathbf{a}^{(2)} \\ \vdots \\ \mathbf{a}^{(m)} \end{bmatrix}, \mathbf{A} = \begin{bmatrix} \mathbf{a}^{(1)} \\ \mathbf{a}^{(2)} \\ \vdots \\ \mathbf{a}^{(m)} \end{bmatrix},$$

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

[The matrix–vector product $\mathbf{A}\mathbf{x}$ is simply a column vector of length m , whose i^{th} element is the dot product $\mathbf{a}^{(i)} \cdot \mathbf{x}$.]

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}^{(1)} \cdot \mathbf{x} \\ \mathbf{a}^{(2)} \cdot \mathbf{x} \\ \vdots \\ \mathbf{a}^{(m)} \cdot \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{a}^{(1)} \cdot \mathbf{x} \\ \mathbf{a}^{(2)} \cdot \mathbf{x} \\ \vdots \\ \mathbf{a}^{(m)} \cdot \mathbf{x} \end{bmatrix}.$$

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

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

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

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

2.3.10. Matrix–Matrix Multiplication

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

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

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

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

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

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

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}^{(1)} \cdot \mathbf{b}_{(1)} & \mathbf{a}^{(1)} \cdot \mathbf{b}_{(2)} & \cdots & \mathbf{a}^{(1)} \cdot \mathbf{b}_{(m)} \\ \mathbf{a}^{(2)} \cdot \mathbf{b}_{(1)} & \mathbf{a}^{(2)} \cdot \mathbf{b}_{(2)} & \cdots & \mathbf{a}^{(2)} \cdot \mathbf{b}_{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}^{(n)} \cdot \mathbf{b}_{(1)} & \mathbf{a}^{(n)} \cdot \mathbf{b}_{(2)} & \cdots & \mathbf{a}^{(n)} \cdot \mathbf{b}_{(m)} \end{bmatrix}.$$

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

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

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

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

✓ 2.3.11. Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* it is. For instance, the ℓ_2 norm measures the (Euclidean) length of a vector. Here, we are employing a notion of *size* that concerns the magnitude of a vector's components (not its dimensionality).

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

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

Many functions are valid norms and different norms encode different notions of size. The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of a right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called **[the ℓ_2 norm]** and expressed as

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

The method `norm` calculates the ℓ_2 norm.

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

```
tensor(5.)
```

[The ℓ_1 norm] is also common and the associated measure is called the Manhattan distance. By definition, the ℓ_1 norm sums the absolute values of a vector's elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

Compared to the ℓ_2 norm, it is less sensitive to outliers. To compute the ℓ_1 norm, we compose the absolute value with the sum operation.

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

```
tensor(7.)
```

Both the ℓ_2 and ℓ_1 norms are special cases of the more general ℓ_p norms:

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

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

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}$$

The Frobenius norm behaves as if it were an ℓ_2 norm of a matrix-shaped vector. Invoking the following function will calculate the Frobenius norm of a matrix.

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

```
tensor(6.)
```

While we do not want to get too far ahead of ourselves, we already can plant some intuition about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *maximize* the revenue associated with a recommender model; *minimize* the distance between predictions and the ground truth observations; *minimize* the distance

between representations of photos of the same person while *maximizing* the distance between representations of photos of different people. These distances, which constitute the objectives of deep learning algorithms, are often expressed as norms.

✓ 2.3.12. Discussion

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

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

To recap:

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as `sum` and `mean` , respectively.
- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the ℓ_1 and ℓ_2 norms, and common matrix norms include the *spectral* and *Frobenius* norms.

Start coding or [generate](#) with AI.

COSE474-Deep Learning

2.5. Automatic Differentiation

Fortunately all modern deep learning frameworks take this work off our plates by offering automatic differentiation (often shortened to autograd). As we pass data through each successive function, the framework builds a computational graph that tracks how each value depends on others. To calculate derivatives, automatic differentiation works backwards through this graph applying the chain rule. The computational algorithm for applying the chain rule in this fashion is called backpropagation.

While autograd libraries have become a hot concern over the past decade, they have a long history. In fact the earliest references to autograd date back over half of a century :cite:Wengert.1964. The core ideas behind modern backpropagation date to a PhD thesis from 1980 :cite:Speelpenning.1980 and were further developed in the late 1980s :cite:Griewank.1989. While backpropagation has become the default method for computing gradients, it is not the only option. For instance, the Julia programming language employs forward propagation :cite:Revels.Lubin.Papamarkou.2016. Before exploring methods, let's first master the autograd package.

```
import torch
```

2.5.1. A Simple Function

Let's assume that we are interested in (**differentiating the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to the column vector \mathbf{x} .**) To start, we assign \mathbf{x} an initial value.

```
x = torch.arange(4.0)
x
→ tensor([0., 1., 2., 3.]
```

[Before we calculate the gradient of y with respect to \mathbf{x} , we need a place to store it.] In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory. Note that the gradient of a scalar-valued function with respect to a vector \mathbf{x} is vector-valued with the same shape as \mathbf{x} .

```
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad # The gradient is None by default
```

We now calculate our function of \mathbf{x} and assign the result to y .

```
y = 2 * torch.dot(x, x)
y
→ tensor(28., grad_fn=<MulBackward0>)
```

We can now take the gradient of y with respect to \mathbf{x} by calling its backward method. Next, we can access the gradient via \mathbf{x} 's grad attribute.

```
y.backward()
x.grad
→ tensor([ 0.,  4.,  8., 12.]
```

(We already know that the gradient of the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to \mathbf{x} should be $4\mathbf{x}$.) We can now verify that the automatic gradient computation and the expected result are identical.

```
x.grad == 4 * x
→ tensor([ True,  True,  True,  True])
```

Now let's calculate another function of x and take its gradient. Note that PyTorch does not automatically reset the gradient buffer when we record a new gradient. Instead, the new gradient is added to the already-stored gradient. This behavior comes in handy when we want to optimize the sum of multiple objective functions. To reset the gradient buffer, we can call `x.grad.zero_()` as follows:

```
x.grad.zero_() # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

→ tensor([1., 1., 1., 1.])

Start coding or [generate](#) with AI.

✓ 2.5.2. Backward for Non-Scalar Variables

When y is a vector, the most natural representation of the derivative of y with respect to a vector x is a matrix called the Jacobian that contains the partial derivatives of each component of y with respect to each component of x . Likewise, for higher-order y and x , the result of differentiation could be an even higher-order tensor.

While Jacobians do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of y with respect to the full vector x , yielding a vector of the same shape as x . For example, we often have a vector representing the value of our loss function calculated separately for each example among a batch of training examples. Here, we just want to sum up the gradients computed individually for each example.

Because deep learning frameworks vary in how they interpret gradients of non-scalar tensors, PyTorch takes some steps to avoid confusion. Invoking `backward` on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar. More formally, we need to provide some vector \mathbf{v} such that `backward` will compute $\mathbf{v}^\top \partial_x \mathbf{y}$ rather than $\partial_x \mathbf{y}$. This next part may be confusing, but for reasons that will become clear later, this argument (representing \mathbf{v}) is named `gradient`. For a more detailed description, see Yang Zhang's [Medium post](#).

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y))) # Faster: y.sum().backward()
x.grad
```

→ tensor([0., 2., 4., 6.])

✓ 2.5.3. Detaching Computation

Sometimes, we wish to move some calculations outside of the recorded computational graph. For example, say that we use the input to create some auxiliary intermediate terms for which we do not want to compute a gradient. In this case, we need to detach the respective computational graph from the final result. The following toy example makes this clearer: suppose we have $z = x * y$ and $y = x * x$ but we want to focus on the direct influence of x on z rather than the influence conveyed via y . In this case, we can create a new variable u that takes the same value as y but whose provenance (how it was created) has been wiped out. Thus u has no ancestors in the graph and gradients do not flow through u to x . For example, taking the gradient of $z = x * u$ will yield the result u , (not $3 * x * x$ as you might have expected since $z = x * x * x$).

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x
```

```
z.sum().backward()
x.grad == u
```

→ tensor([True, True, True, True])

Note that while this procedure detaches y 's ancestors from the graph leading to z , the computational graph leading to y persists and thus we can calculate the gradient of y with respect to x .

```
x.grad.zero_()
y.sum().backward()
```

```
x.grad == 2 * x
```

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

✓ 2.5.4. Gradients and Python Control Flow

So far we reviewed cases where the path from input to output was well defined via a function such as $z = x * x * x$. Programming offers us a lot more freedom in how we compute results. For instance, we can make them depend on auxiliary variables or condition choices on intermediate results. One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable. To illustrate this, consider the following code snippet where the number of iterations of the while loop and the evaluation of the if statement both depend on the value of the input a .

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Below, we call this function, passing in a random value, as input. Since the input is a random variable, we do not know what form the computational graph will take. However, whenever we execute $f(a)$ on a specific input, we realize a specific computational graph and can subsequently run backward.

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Even though our function f is, for demonstration purposes, a bit contrived, its dependence on the input is quite simple: it is a linear function of a with piecewise defined scale. As such, $f(a) / a$ is a vector of constant entries and, moreover, $f(a) / a$ needs to match the gradient of $f(a)$ with respect to a .

```
a.grad == d / a
```

```
→ tensor(True)
```

Dynamic control flow is very common in deep learning. For instance, when processing text, the computational graph depends on the length of the input. In these cases, automatic differentiation becomes vital for statistical modeling since it is impossible to compute the gradient a priori.

✓ 2.5.5. Discussion

You have now gotten a taste of the power of automatic differentiation. The development of libraries for calculating derivatives both automatically and efficiently has been a massive productivity booster for deep learning practitioners, liberating them so they can focus on less menial. Moreover, autograd lets us design massive models for which pen and paper gradient computations would be prohibitively time consuming. Interestingly, while we use autograd to optimize models (in a statistical sense) the optimization of autograd libraries themselves (in a computational sense) is a rich subject of vital interest to framework designers. Here, tools from compilers and graph manipulation are leveraged to compute results in the most expedient and memory-efficient manner.

For now, try to remember these basics: (i) attach gradients to those variables with respect to which we desire derivatives; (ii) record the computation of the target value; (iii) execute the backpropagation function; and (iv) access the resulting gradient.

Start coding or [generate](#) with AI.

COSE474-2024F Deep Learning

3.1 Linear Regression

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting the length of stay (for patients in the hospital), forecasting demand (for retail sales), among numerous others. Not every prediction problem is one of classical regression. Later on, we will introduce classification problems, where the goal is to predict membership among a set of categories.

As a running example, suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years). To develop a model for predicting house prices, we need to get our hands on data, including the sales price, area, and age for each home. In the terminology of machine learning, the dataset is called a training dataset or training set, and each row (containing the data corresponding to one sale) is called an example (or data point, instance, sample). The thing we are trying to predict (price) is called a label (or target). The variables (age and area) upon which the predictions are based are called features (or covariates).

```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

3.1.1. Basics

Linear regression is both the simplest and most popular among the standard tools for tackling regression problems. Dating back to the dawn of the 19th century :cite: Legendre.1805, Gauss.1809, linear regression flows from a few simple assumptions. First, we assume that the relationship between features \mathbf{x} and target y is approximately linear, i.e., that the conditional mean $E[Y | X = \mathbf{x}]$ can be expressed as a weighted sum of the features \mathbf{x} . This setup allows that the target value may still deviate from its expected value on account of observation noise. Next, we can impose the assumption that any such noise is well behaved, following a Gaussian distribution. Typically, we will use n to denote the number of examples in our dataset. We use superscripts to enumerate samples and targets, and subscripts to index coordinates. More concretely, $\mathbf{x}^{(i)}$ denotes the i^{th} sample and $x_j^{(i)}$ denotes its j^{th} coordinate.

3.1.1.1. Model

At the heart of every solution is a model that describes how features can be transformed into an estimate of the target. The assumption of linearity means that the expected value of the target (price) can be expressed as a weighted sum of the features (area and age):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b.$$

:eqlabel: eq_price-area

Here w_{area} and w_{age} are called *weights*, and b is called a *bias* (or *offset* or *intercept*). The weights determine the influence of each feature on our prediction. The bias determines the value of the estimate when all features are zero. Even though we will never see any newly-built homes with precisely zero area, we still need the bias because it allows us to express all linear functions of our features (rather than restricting us to lines that pass through the origin). Strictly speaking, :eqref: eq_price-area is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via a weighted sum, combined with a *translation* via the added bias. Given a dataset, our goal is to choose the weights \mathbf{w} and the bias b that, on average, make our model's predictions fit the true prices observed in the data as closely as possible.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form, as in :eqref: eq_price-area, is common. In machine learning, we usually work with high-dimensional datasets, where it is more convenient to employ compact linear algebra notation. When our inputs consist of d features, we can assign each an index (between 1 and d) and express our prediction \hat{y} (in general the "hat" symbol denotes an estimate) as

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b.$$

Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^d$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^d$, we can express our model compactly via the dot product between \mathbf{w} and \mathbf{x} :

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

:eqlabel: eq_linreg-y

In :eqref: eq_linreg-y, the vector \mathbf{x} corresponds to the features of a single example. We will often find it convenient to refer to features of our entire dataset of n examples via the *design matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$. Here, \mathbf{X} contains one row for every example and one column for every feature. For a collection of features \mathbf{X} , the predictions $\hat{\mathbf{y}} \in \mathbb{R}^n$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b,$$

:eqlabel: eq_linreg-y-vec

where broadcasting (:numref: subsec_broadcasting) is applied during the summation. Given features of a training dataset \mathbf{X} and corresponding (known) labels \mathbf{y} , the goal of linear regression is to find the weight vector \mathbf{w} and the bias term b such that, given features of a new data example sampled from the same distribution as \mathbf{X} , the new example's label will (in expectation) be predicted with the smallest error.

Even if we believe that the best model for predicting y given \mathbf{x} is linear, we would not expect to find a real-world dataset of n examples where $y^{(i)}$ exactly equals $\mathbf{w}^\top \mathbf{x}^{(i)} + b$ for all $1 \leq i \leq n$. For example, whatever instruments we use to observe the features \mathbf{X} and labels \mathbf{y} , there might be a small amount of measurement error. Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

Before we can go about searching for the best *parameters* (or *model parameters*) \mathbf{w} and b , we will need two more things: (i) a measure of the quality of some given model; and (ii) a procedure for updating the model to improve its quality.


✓ 3.1.1.2. Loss Function

Naturally, fitting our model to the data requires that we agree on some measure of *fitness* (or, equivalently, of *unfitness*). *Loss functions* quantify the distance between the *real* and *predicted* values of the target. The loss will usually be a nonnegative number where smaller values are better and perfect predictions incur a loss of 0. For regression problems, the most common loss function is the squared error. When our prediction for an example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the *squared error* is given by:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)} \right)^2.$$

:eqlabel: eq_mse

The constant $\frac{1}{2}$ makes no real difference but proves to be notationally convenient, since it cancels out when we take the derivative of the loss. Because the training dataset is given to us, and thus is out of our control, the empirical error is only a function of the model parameters. In :numref: fig_fit_linreg, we visualize the fit of a linear regression model in a problem with one-dimensional inputs.

 Fitting a linear regression model to one-dimensional data. :label: fig_fit_linreg

Note that large differences between estimates $\hat{y}^{(i)}$ and targets $y^{(i)}$ lead to even larger contributions to the loss, due to its quadratic form (this quadraticity can be a double-edge sword; while it encourages the model to avoid large errors it can also lead to excessive sensitivity to anomalous data). To measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2.$$

When training the model, we seek parameters (\mathbf{w}^*, b^*) that minimize the total loss across all training examples:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} L(\mathbf{w}, b).$$

✓ 3.1.1.3. Analytic Solution

Unlike most of the models that we will cover, linear regression presents us with a surprisingly easy optimization problem. In particular, we can find the optimal parameters (as assessed on the training data) analytically by applying a simple formula as follows. First, we can subsume the bias b into the parameter \mathbf{w} by appending a column to the design matrix consisting of all 1s. Then our prediction problem is to minimize $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. As long as the design matrix \mathbf{X} has full rank (no feature is linearly dependent on the others), then there will be just one critical point on the loss surface and it corresponds to the minimum of the loss over the entire domain. Taking the derivative of the loss with respect to \mathbf{w} and setting it equal to zero yields:

$$\partial_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top \mathbf{X}\mathbf{w}.$$

Solving for \mathbf{w} provides us with the optimal solution for the optimization problem. Note that this solution

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

will only be unique when the matrix $\mathbf{X}^\top \mathbf{X}$ is invertible, i.e., when the columns of the design matrix are linearly independent :cite: Golub.Van-Loan.1996 .

While simple problems like linear regression may admit analytic solutions, you should not get used to such good fortune. Although analytic solutions allow for nice mathematical analysis, the requirement of an analytic solution is so restrictive that it would exclude almost all exciting aspects of deep learning.

✓ 3.1.1.4. Minibatch Stochastic Gradient Descent

Fortunately, even in cases where we cannot solve the models analytically, we can still often train models effectively in practice. Moreover, for many tasks, those hard-to-optimize models turn out to be so much better that figuring out how to train them ends up being well worth the trouble.

The key technique for optimizing nearly every deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent*.

The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset. In practice, this can be extremely slow: we must pass over the entire dataset before making a single update, even if the update steps might be very powerful :cite: Liu.Nocedal.1989 . Even worse, if there is a lot of redundancy in the training data, the benefit of a full update is limited.

The other extreme is to consider only a single example at a time and to take update steps based on one observation at a time. The resulting algorithm, *stochastic gradient descent* (SGD) can be an effective strategy :cite: Bottou.2010 , even for large datasets. Unfortunately, SGD has drawbacks, both computational and statistical. One problem arises from the fact that processors are a lot faster multiplying and adding numbers than they are at moving data from main memory to processor cache. It is up to an order of magnitude more efficient to perform a matrix–vector multiplication than a corresponding number of vector–vector operations. This means that it can take a lot longer to process one sample at a time compared to a full batch. A second problem is that some of the layers, such as batch normalization (to be described in :numref: sec_batch_norm), only work well when we have access to more than one observation at a time.

The solution to both problems is to pick an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a *minibatch* of observations :cite: Li.Zhang.Chen.ea.2014 . The specific choice of the size of the said minibatch depends on many factors, such as the amount of memory, the number of accelerators, the choice of layers, and the total dataset size. Despite all that, a number between 32 and 256, preferably a multiple of a large power of 2, is a good start. This leads us to *minibatch stochastic gradient descent*.

In its most basic form, in each iteration t , we first randomly sample a minibatch \mathcal{B}_t consisting of a fixed number $|\mathcal{B}|$ of training examples. We then compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters. Finally, we multiply the gradient by a predetermined small positive value η , called the *learning rate*, and subtract the resulting term from the current parameter values. We can express the update as follows:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b).$$

In summary, minibatch SGD proceeds as follows: (i) initialize the values of the model parameters, typically at random; (ii) iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient. For quadratic losses and affine transformations, this has a closed-form expansion:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &= \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right) \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) &= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned}$$

:eqlabel: eq_linreg_batch_update

Since we pick a minibatch \mathcal{B} we need to normalize by its size $|\mathcal{B}|$. Frequently minibatch size and learning rate are user-defined. Such tunable parameters that are not updated in the training loop are called *hyperparameters*. They can be tuned automatically by a number of techniques, such as Bayesian optimization :cite: Frazier.2018 . In the end, the quality of the solution is typically assessed on a separate *validation dataset* (or *validation set*).

After training for some predetermined number of iterations (or until some other stopping criterion is met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$. Note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss, nor even deterministic. Although the algorithm converges slowly towards the minimizers it typically will not find them exactly in a finite number of steps. Moreover, the minibatches \mathcal{B} used for updating the parameters are chosen at random. This breaks determinism.

Linear regression happens to be a learning problem with a global minimum (whenever \mathbf{X} is full rank, or equivalently, whenever $\mathbf{X}^T \mathbf{X}$ is invertible). However, the loss surfaces for deep networks contain many saddle points and minima. Fortunately, we typically do not care about finding an exact set of parameters but merely any set of parameters that leads to accurate predictions (and thus low loss). In practice, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training sets* :cite:Izmailov.Podoprikin.Garipov.ea.2018, Frankle.Carbin.2018. The more formidable task is to find parameters that lead to accurate predictions on previously unseen data, a challenge called *generalization*. We return to these topics throughout the book.

3.1.1.5. Predictions

Given the model $\hat{\mathbf{w}}^T \mathbf{x} + \hat{b}$, we can now make *predictions* for a new example, e.g., predicting the sales price of a previously unseen house given its area x_1 and age x_2 . Deep learning practitioners have taken to calling the prediction phase *inference* but this is a bit of a misnomer—*inference* refers broadly to any conclusion reached on the basis of evidence, including both the values of the parameters and the likely label for an unseen instance. If anything, in the statistics literature *inference* more often denotes parameter inference and this overloading of terminology creates unnecessary confusion when deep learning practitioners talk to statisticians. In the following we will stick to *prediction* whenever possible.

3.1.2. Vectorization for Speed

When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

To see why this matters so much, let's consider two methods for adding vectors. To start, we instantiate two 10,000-dimensional vectors containing all 1s. In the first method, we loop over the vectors with a Python for-loop. In the second, we rely on a single call to `+`.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

Now we can benchmark the workloads. First, we add them, one coordinate at a time, using a for-loop.

```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

```
0.14105 sec
```

Alternatively, we rely on the reloaded `+` operator to compute the elementwise sum.

```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
0.00106 sec
```

The second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the mathematics to the library so we do not have to write as many calculations ourselves, reducing the potential for errors and increasing portability of the code.

3.1.3. The Normal Distribution and Squared Loss

So far we have given a fairly functional motivation of the squared loss objective: the optimal parameters return the conditional expectation $E[Y | X]$ whenever the underlying pattern is truly linear, and the loss assigns large penalties for outliers. We can also provide a more formal motivation for the squared loss objective by making probabilistic assumptions about the distribution of noise.

Linear regression was invented at the turn of the 19th century. While it has long been debated whether Gauss or Legendre first thought up the idea, it was Gauss who also discovered the normal distribution (also called the *Gaussian*). It turns out that the normal distribution and linear

regression with squared loss share a deeper connection than common parentage.

To begin, recall that a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

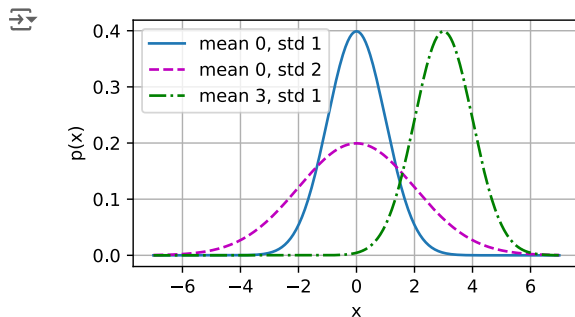
Below [we define a function to compute the normal distribution].

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

We can now visualize the normal distributions.

```
# Use NumPy again for visualization
x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



Note that changing the mean corresponds to a shift along the x -axis, and increasing the variance spreads the distribution out, lowering its peak.

One way to motivate linear regression with squared loss is to assume that observations arise from noisy measurements, where the noise ϵ follows the normal distribution $\mathcal{N}(0, \sigma^2)$:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2).$$

Thus, we can now write out the *likelihood* of seeing a particular y for a given \mathbf{x} via

$$P(y | \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right).$$

As such, the likelihood factorizes. According to *the principle of maximum likelihood*, the best values of parameters \mathbf{w} and b are those that maximize the *likelihood* of the entire dataset:

$$P(\mathbf{y} | \mathbf{X}) = \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}).$$

The equality follows since all pairs $(\mathbf{x}^{(i)}, y^{(i)})$ were drawn independently of each other. Estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*. While, maximizing the product of many exponential functions, might look difficult, we can simplify things significantly, without changing the objective, by maximizing the logarithm of the likelihood instead. For historical reasons, optimizations are more often expressed as minimization rather than maximization. So, without changing anything, we can *minimize* the *negative log-likelihood*, which we can express as follows:

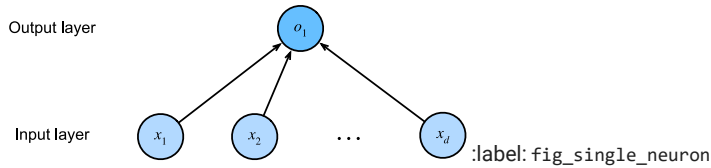
$$-\log P(\mathbf{y} | \mathbf{X}) = \sum_{i=1}^n \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b)^2.$$

If we assume that σ is fixed, we can ignore the first term, because it does not depend on \mathbf{w} or b . The second term is identical to the squared error loss introduced earlier, except for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ either. It follows that minimizing the mean squared error is equivalent to the maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

✓ 3.1.4. Linear Regression as a Neural Network

While linear models are not sufficiently rich to express the many complicated networks that we will introduce in this book, (artificial) neural networks are rich enough to subsume linear models as networks in which every feature is represented by an input neuron, all of which are connected directly to the output.

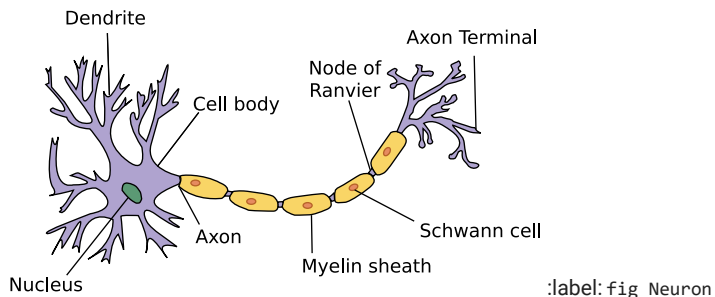
:numref: fig_single_neuron depicts linear regression as a neural network. The diagram highlights the connectivity pattern, such as how each input is connected to the output, but not the specific values taken by the weights or biases.



The inputs are x_1, \dots, x_d . We refer to d as the *number of inputs* or the *feature dimensionality* in the input layer. The output of the network is o_1 . Because we are just trying to predict a single numerical value, we have only one output neuron. Note that the input values are all *given*. There is just a single *computed* neuron. In summary, we can think of linear regression as a single-layer fully connected neural network. We will encounter networks with far more layers in later chapters.

✓ 3.1.4.1 Biology

Because linear regression predates computational neuroscience, it might seem anachronistic to describe linear regression in terms of neural networks. Nonetheless, they were a natural place to start when the cyberneticists and neurophysiologists Warren McCulloch and Walter Pitts began to develop models of artificial neurons. Consider the cartoonish picture of a biological neuron in :numref: fig_Neuron, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*.



Information x_i arriving from other neurons (or environmental sensors) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i , determining the effect of the inputs, e.g., activation or inhibition via the product $x_i w_i$. The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, possibly subject to some nonlinear postprocessing via a function $\sigma(y)$. This information is then sent via the axon to the axon terminals, where it reaches its destination (e.g., an actuator such as a muscle) or it is fed into another neuron via its dendrites.

Certainly, the high-level idea that many such units could be combined, provided they have the correct connectivity and learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express arises from our study of real biological neural systems. At the same time, most research in deep learning today draws inspiration from a much wider source. We invoke


:cite: Russell.Norvig.2016 who pointed out that although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics innovation for some centuries. Likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, linguistics, psychology, statistics, computer science, and many other fields.

✓ 3.1.5. Summary

In this section, we introduced traditional linear regression, where the parameters of a linear function are chosen to minimize squared loss on the training set. We also motivated this choice of objective both via some practical considerations and through an interpretation of linear regression as maximum likelihood estimation under an assumption of linearity and Gaussian noise. After discussing both computational considerations and connections to statistics, we showed how such linear models could be expressed as simple neural networks where the inputs are directly wired to the output(s). While we will soon move past linear models altogether, they are sufficient to introduce most of the

components that all of our models require: parametric forms, differentiable objectives, optimization via minibatch stochastic gradient descent, and ultimately, evaluation on previously unseen data.

```
!pip install d2l==1.0.3
```

 Show hidden output

COSE474-2024F: Deep Learning

✓ 3.2. Object-Oriented Design for Implementation

In our introduction to linear regression, we walked through various components including the data, the model, the loss function, and the optimization algorithm. Indeed, linear regression is one of the simplest machine learning models. Training it, however, uses many of the same components that other models in this book require. Therefore, before diving into the implementation details it is worth designing some of the APIs that we use throughout. Treating components in deep learning as objects, we can start by defining classes for these objects and their interactions. This object-oriented design for implementation will greatly streamline the presentation and you might even want to use it in your projects.

Inspired by open-source libraries such as PyTorch Lightning, at a high level we wish to have three classes: (i) Module contains models, losses, and optimization methods; (ii) DataModule provides data loaders for training and validation; (iii) both classes are combined using the Trainer class, which allows us to train models on a variety of hardware platforms. Most code in this book adapts Module and DataModule. We will touch upon the Trainer class only when we discuss GPUs, CPUs, parallel training, and optimization algorithms.

```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

✓ 3.2.1. Utilities

We need a few utilities to simplify object-oriented programming in Jupyter notebooks. One of the challenges is that class definitions tend to be fairly long blocks of code. Notebook readability demands short code fragments, interspersed with explanations, a requirement incompatible with the style of programming common for Python libraries. The first utility function allows us to register functions as methods in a class after the class has been created. In fact, we can do so even after we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

Let's have a quick look at how to use it. We plan to implement a class A with a method do. Instead of having code for both A and do in the same code block, we can first declare the class A and create an instance a.

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

Next we define the method do as we normally would, but not in class A's scope. Instead, we decorate this method by add_to_class with class A as its argument. In doing so, the method is able to access the member variables of A just as we would expect had it been included as part of A's definition. Let's see what happens when we invoke it for the instance a.

```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)
```

```
a.do()
```

```
↪ Class attribute "b" is 1
```

The second one is a utility class that saves all arguments in a class's **init** method as class attributes. This allows us to extend constructor call signatures implicitly without additional code.

```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

To use it, we define our class that inherits from HyperParameters and calls save_hyperparameters in the **init** method.

```
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))
```

```
b = B(a=1, b=2, c=3)
```

```
↪ self.a = 1 self.b = 2
   There is no self.c = True
```

The final utility allows us to plot experiment progress interactively while it is going on. In deference to the much more powerful (and complex) [TensorBoard](#) we name it ProgressBoard. The implementation is deferred to :numref:sec_utils. For now, let's simply see it in action.

The draw method plots a point (x, y) in the figure, with label specified in the legend. The optional every_n smooths the line by only showing 1/n points in the figure. Their values are averaged from the n neighbor points in the original figure.

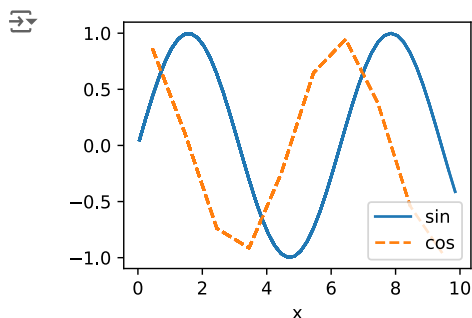
```
class ProgressBoard(d2l.HyperParameters): #@save
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

In the following example, we draw sin and cos with a different smoothness. If you run this code block, you will see the lines grow in animation

```
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



✓ 3.2.2. Models

The `Module` class is the base class of all models we will implement. At the very least we need three methods. The first, `__init__`, stores the learnable parameters, the `training_step` method accepts a data batch to return the loss value, and finally, `configure_optimizers` returns the optimization method, or a list of them, that is used to update the learnable parameters. Optionally we can define `validation_step` to report the evaluation measures. Sometimes we put the code for computing the output into a separate `forward` method to make it more reusable.

```
class Module(nn.Module, d2l.HyperParameters): #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not initied'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

You may notice that `Module` is a subclass of `nn.Module`, the base class of neural networks in PyTorch. It provides convenient features for handling neural networks. For example, if we define a forward method, such as `forward(self, X)`, then for an instance `a` we can invoke this method by `a(X)`. This works since it calls the forward method in the built-in `call` method. You can find more details and examples about `nn`.

✓ 3.2.3. Data

The `DataModule` class is the base class for data. Quite frequently the `init` method is used to prepare the data. This includes downloading and preprocessing if needed. The `train_dataloader` returns the data loader for the training dataset. A data loader is a (Python) generator that yields a data batch each time it is used. This batch is then fed into the `training_step` method of `Module` to compute the loss. There is an optional `val_dataloader` to return the validation dataset loader. It behaves in the same manner, except that it yields data batches for the `validation_step` method in `Module`.

```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError
```

```
def train_dataloader(self):
    return self.get_dataloader(train=True)

def val_dataloader(self):
    return self.get_dataloader(train=False)
```

✓ 3.2.4. Training

The Trainer class trains the learnable parameters in the Module class with data specified in DataModule. The key method is fit, which accepts two arguments: model, an instance of Module, and data, an instance of DataModule. It then iterates over the entire dataset max_epochs times to train the model. As before, we will defer the implementation of this method to later chapters.

```
class Trainer(d2l.HyperParameters):
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.epoch = 0
        self.train_batch_idx = 0
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            self.fit_epoch()


    def fit_epoch(self):
        raise NotImplementedError
```

✓ 3.2.5. Summary

To highlight the object-oriented design for our future deep learning implementation, the above classes simply show how their objects store data and interact with each other. We will keep enriching implementations of these classes, such as via @add_to_class, in the rest of the book. Moreover, these fully implemented classes are saved in the D2L library, a lightweight toolkit that makes structured modeling for deep learning easy. In particular, it facilitates reusing many components between projects without changing much at all. For instance, we can replace just the optimizer, just the model, just the dataset, etc.; this degree of modularity pays dividends throughout the book in terms of conciseness and simplicity (this is why we added it) and it can do the same for your own projects.

Start coding or [generate](#) with AI.


```
!pip install d2l==1.0.3
```

 Show hidden output

COSE474-2024F: Deep Learning

✓ 3.4. Linear Regression Implementation from Scratch

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

✓ 3.4.1. Defining the Model

Before we can begin optimizing our model's parameters by minibatch SGD, we need to have some parameters in the first place. In the following we initialize weights by drawing random numbers from a normal distribution with mean 0 and a standard deviation of 0.01. The magic number 0.01 often works well in practice, but you can specify a different value through the argument `sigma`. Moreover we set the bias to 0. Note that for object-oriented design we add the code to the `init` method of a subclass of `d2l.Module`

```
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

Next we must **[define our model, relating its input and parameters to its output.]** We simply take the matrix–vector product of the input features \mathbf{X} and the model weights \mathbf{w} , and add the offset b to each example. The product $\mathbf{X}\mathbf{w}$ is a vector and b is a scalar. Because of the broadcasting mechanism, when we add a vector and a scalar, the scalar is added to each component of the vector. The resulting `forward` method is registered in the `LinearRegressionScratch` class via `add_to_class`.

```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

✓ 3.4.2. Defining the Loss Function

Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first. In the implementation, we need to transform the true value y into the predicted value's shape y_{hat} . The result returned by the following method will also have the same shape as y_{hat} . We also return the averaged loss value among all examples in the minibatch.

```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

✓ 3.4.3. Defining the Optimization Algorithm

Our goal here is to illustrate how to train more general neural networks, and that requires that we teach you how to use minibatch SGD. Hence we will take this opportunity to introduce your first working example of SGD. At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters. Next, we update the parameters in the direction that may reduce the loss.

The following code applies the update, given a set of parameters, a learning rate `lr`. Since our loss is computed as an average over the minibatch, we do not need to adjust the learning rate against the batch size. In later chapters we will investigate how learning rates should be

adjusted for very large minibatches as they arise in distributed large-scale learning. For now, we can ignore this dependency.

We define our `SGD` class, a subclass of `d2l.HyperParameters` (introduced in :numref:oo-design-utilities), to have a similar API as the built-in SGD optimizer. We update the parameters in the `step` method. The `zero_grad` method sets all gradients to 0, which must be run before a backpropagation step.

```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

We next define the `configure_optimizers` method, which returns an instance of the `SGD` class.

```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

✓ 3.4.4. Training

Now that we have all of the parts in place (parameters, loss function, model, and optimizer), we are ready to **[implement the main training loop.]**

It is crucial that you understand this code fully since you will employ similar training loops for every other deep learning model covered in this book. In each *epoch*, we iterate through the entire training dataset, passing once through every example (assuming that the number of examples is divisible by the batch size). In each *iteration*, we grab a minibatch of training examples, and compute its loss through the model's `training_step` method. Then we compute the gradients with respect to each parameter. Finally, we will call the optimization algorithm to update the model parameters. In summary, we will execute the following loop:

- Initialize parameters (\mathbf{w}, b)
- Repeat until done
 - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w}, b)} \frac{1}{|B|} \sum_{i \in B} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
 - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Recall that the synthetic regression dataset that we generated in :numref:sec_synthetic-regression-data does not provide a validation dataset. In most cases, however, we will want a validation dataset to measure our model quality. Here we pass the validation dataloader once in each epoch to measure the model performance. Following our object-oriented design, the `prepare_batch` and `fit_epoch` methods are registered in the `d2l.Trainer` class (introduced in :numref:oo-design-training).

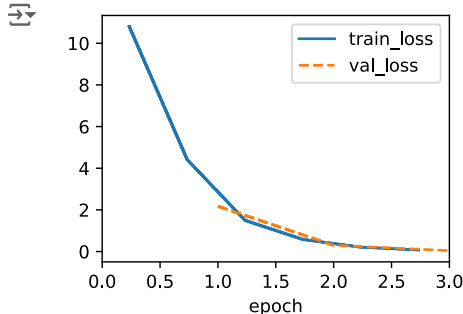
```
@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0: # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
        self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
```

```
with torch.no_grad():
    self.model.validation_step(self.prepare_batch(batch))
self.val_batch_idx += 1
```

We are almost ready to train the model, but first we need some training data. Here we use the `SyntheticRegressionData` class and pass in some ground truth parameters. Then we train our model with the learning rate `lr=0.03` and set `max_epochs=3`. Note that in general, both the number of epochs and the learning rate are hyperparameters. In general, setting hyperparameters is tricky and we will usually want to use a three-way split, one set for training, a second for hyperparameter selection, and the third reserved for the final evaluation. We elide these details for now but will revise them later.

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



[+ Code](#) [+ Text](#)

Because we synthesized the dataset ourselves, we know precisely what the true parameters are. Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop. Indeed they turn out to be very close to each other.

```
with torch.no_grad():
    print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
    print(f'error in estimating b: {data.b - model.b}')

```

```
error in estimating w: tensor([ 0.1053, -0.1708])
error in estimating b: tensor([0.2220])
```


We should not take the ability to exactly recover the ground truth parameters for granted. In general, for deep models unique solutions for the parameters do not exist, and even for linear models, exactly recovering the parameters is only possible when no feature is linearly dependent on the others. However, in machine learning, we are often less concerned with recovering true underlying parameters, but rather with parameters that lead to highly accurate prediction (Vapnik, 1992). Fortunately, even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions, owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to highly accurate prediction.

✓ 3.4.5. Summary

In this section, we took a significant step towards designing deep learning systems by implementing a fully functional neural network model and training loop. In this process, we built a data loader, a model, a loss function, an optimization procedure, and a visualization and monitoring tool. We did this by composing a Python object that contains all relevant components for training a model. While this is not yet a professional-grade implementation it is perfectly functional and code like this could already help you to solve small problems quickly. In the coming sections, we will see how to do this both more concisely (avoiding boilerplate code) and more efficiently (using our GPUs to their full potential).

[Start coding](#) or [generate](#) with AI.


```
!pip install d2l==1.0.3
```

 Show hidden output

✓ COSE474-2024F: Deep Learning

4.1. Softmax Regression

Regression is the hammer we reach for when we want to answer how much? or how many? questions. If you want to predict the number of dollars (price) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model. However, even within regression models, there are important distinctions. For instance, the price of a house will never be negative and changes might often be relative to its baseline price. As such, it might be more effective to regress on the logarithm of the price. Likewise, the number of days a patient spends in hospital is a discrete nonnegative random variable. As such, least mean squares might not be an ideal approach either. This sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called survival modeling.

The point here is not to overwhelm you but just to let you know that there is a lot more to estimation than simply minimizing squared errors. And more broadly, there is a lot more to supervised learning than regression. In this section, we focus on classification problems where we put aside how much? questions and instead focus on which category? questions.

Does this email belong in the spam folder or the inbox?

Is this customer more likely to sign up or not to sign up for a subscription service?

Does this image depict a donkey, a dog, a cat, or a rooster?

Which movie is Aston most likely to watch next?

Which section of the book are you going to read next?

Colloquially, machine learning practitioners overload the word classification to describe two subtly different problems: (i) those where we are interested only in hard assignments of examples to categories (classes); and (ii) those where we wish to make soft assignments, i.e., to assess the probability that each category applies. The distinction tends to get blurred, in part, because often, even when we only care about hard assignments, we still use models that make soft assignments.

Even more, there are cases where more than one label might be true. For instance, a news article might simultaneously cover the topics of entertainment, business, and space flight, but not the topics of medicine or sports. Thus, categorizing it into one of the above categories on their own would not be very useful. This problem is commonly known as multi-label classification. See Tsoumakas and Katakis (2007) for an overview and Huang et al. (2015) for an effective algorithm when tagging images.

✓ 4.1.1. Classification

To get our feet wet, let's start with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4 . Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict {baby, toddler, adolescent, young adult, adult, geriatric}, then it might even make sense to cast this as an [ordinal regression](#) problem and keep the labels in this format. See :citet: Moon.Smo1a.Chang.ea.2010 for an overview of different types of ranking loss functions and :citet: Beutel1.Murray.Faloutsos.ea.2014 for a Bayesian approach that addresses responses with more than one mode.

In general, classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0. In our case, a label y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

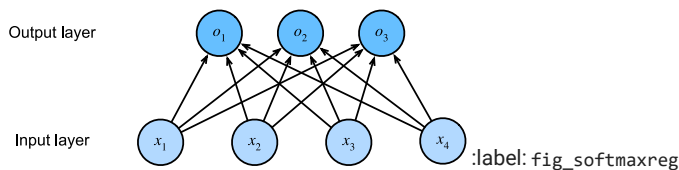
$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

4.1.1.1. Linear Model

In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization. Each output corresponds to its own affine function. In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). This yields:

$$\begin{aligned}o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.\end{aligned}$$

The corresponding neural network diagram is shown in :numref:fig_softmaxreg. Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, o_1 , o_2 , and o_3 , depends on every input, x_1 , x_2 , x_3 , and x_4 , the output layer can also be described as a *fully connected layer*.



For a more concise notation we use vectors and matrices: $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$ is much better suited for mathematics and code. Note that we have gathered all of our weights into a 3×4 matrix and all biases $\mathbf{b} \in \mathbb{R}^3$ in a vector.

4.1.1.2. The Softmax

Assuming a suitable loss function, we could try, directly, to minimize the difference between \mathbf{o} and the labels \mathbf{y} . While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Both aspects render the estimation problem difficult to solve and the solution very brittle to outliers. For instance, if we assume that there is a positive linear dependency between the number of bedrooms and the likelihood that someone will buy a house, the probability might exceed 1 when it comes to buying a mansion! As such, we need a mechanism to "squish" the outputs.

There are many ways we might accomplish this goal. For instance, we could assume that the outputs \mathbf{o} are corrupted versions of \mathbf{y} , where the corruption occurs by means of adding noise ϵ drawn from a normal distribution. In other words, $\mathbf{y} = \mathbf{o} + \epsilon$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. This is the so-called *probit model*, first introduced by :citel: Fechner. 1860. While appealing, it does not work quite as well nor lead to a particularly nice optimization problem, when compared to the softmax.

Another way to accomplish this goal (and to ensure nonnegativity) is to use an exponential function $P(y = i) \propto \exp o_i$. This does indeed satisfy the requirement that the conditional class probability increases with increasing o_i , it is monotonic, and all probabilities are nonnegative. We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called *normalization*. Putting these two pieces together gives us the *softmax* function:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}.$$

:eqlabel: eq_softmax_y_and_o

Note that the largest coordinate of \mathbf{o} corresponds to the most likely class according to $\hat{\mathbf{y}}$. Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability. Thus,

$$\operatorname{argmax}_j \hat{y}_j = \operatorname{argmax}_j o_j.$$

The idea of a softmax dates back to :citel: Gibbs. 1902, who adapted ideas from physics. Dating even further back, Boltzmann, the father of modern statistical physics, used this trick to model a distribution over energy states in gas molecules. In particular, he discovered that the prevalence of a state of energy in a thermodynamic ensemble, such as the molecules in a gas, is proportional to $\exp(-E/kT)$. Here, E is the energy of a state, T is the temperature, and k is the Boltzmann constant. When statisticians talk about increasing or decreasing the "temperature" of a statistical system, they refer to changing T in order to favor lower or higher energy states. Following Gibbs' idea, energy equates to error. Energy-based models :cite: Ranzato. Boureau. Chopra. et al. 2007 use this point of view when describing problems in deep learning.

4.1.1.3. Vectorization

To improve computational efficiency, we vectorize calculations in minibatches of data. Assume that we are given a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with dimensionality (number of inputs) d . Moreover, assume that we have q categories in the output. Then the weights satisfy $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned}\mathbf{O} &= \mathbf{XW} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}).\end{aligned}$$

:eqlabel: eq_minibatch_softmax_reg

This accelerates the dominant operation into a matrix–matrix product \mathbf{XW} . Moreover, since each row in \mathbf{X} represents a data example, the softmax operation itself can be computed *rowwise*: for each row of \mathbf{O} , exponentiate all entries and then normalize them by the sum. Note, though, that care must be taken to avoid exponentiating and taking logarithms of large numbers, since this can cause numerical overflow or underflow. Deep learning frameworks take care of this automatically.

✓ 4.1.2. Loss Function

Now that we have a mapping from features to probabilities, we need a way to optimize the accuracy of this mapping. We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss

4.1.2.1. Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as the (estimated) conditional probabilities of each class, given any input \mathbf{x} , such as $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$. In the following we assume that for a dataset with features \mathbf{X} the labels \mathbf{Y} are represented using a one-hot encoding label vector. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^n P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution $P(\mathbf{y} \mid \mathbf{x}^{(i)})$. Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^n -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^n l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

where for any pair of label \mathbf{y} and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^q y_j \log \hat{y}_j.$$

:eqlabel: eq_1_cross_entropy

For reasons explained later on, the loss function in :eqref: eq_1_cross_entropy is commonly called the *cross-entropy loss*. Since \mathbf{y} is a one-hot vector of length q , the sum over all its coordinates j vanishes for all but one term. Note that the loss $l(\mathbf{y}, \hat{\mathbf{y}})$ is bounded from below by 0 whenever $\hat{\mathbf{y}}$ is a probability vector: no single entry is larger than 1, hence their negative logarithm cannot be lower than 0; $l(\mathbf{y}, \hat{\mathbf{y}}) = 0$ only if we predict the actual label with *certainty*. This can never happen for any finite setting of the weights because taking a softmax output towards 1 requires taking the corresponding input o_i to infinity (or all other outputs o_j for $j \neq i$ to negative infinity). Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss ($-\log 0 = \infty$).

4.1.2.2. Softmax and Cross-Entropy Loss

Since the softmax function and the corresponding cross-entropy loss are so common, it is worth understanding a bit better how they are computed. Plugging :eqref: eq_softmax_y_and_o into the definition of the loss in :eqref: eq_1_cross_entropy and using the definition of the softmax we obtain

$$\begin{aligned}l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^q y_j \log \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} \\ &= \sum_{j=1}^q y_j \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j \\ &= \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j.\end{aligned}$$

To understand a bit better what is going on, consider the derivative with respect to any logit o_j . We get

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j.$$

In other words, the derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate \hat{y} . This is not a coincidence. In any exponential family model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for the label \mathbf{y} . The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l in `eqref: eq_1_cross_entropy` still works well, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels. This loss is called the *cross-entropy loss* and it is one of the most commonly used losses for classification problems. We can demystify the name by introducing just the basics of information theory. In a nutshell, it measures the number of bits needed to encode what we see, \mathbf{y} , relative to what we predict that should happen, $\hat{\mathbf{y}}$. We provide a very basic explanation in the following. For further details on information theory see `:cite: Cover.1999` or `:cite: mackay2003information`.

✓ 4.1.3. Information Theory Basics

Many deep learning papers use intuition and terms from information theory. To make sense of them, we need some common language. This is a survival guide. Information theory deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

4.1.3.1. Entropy

The central idea in information theory is to quantify the amount of information contained in data. This places a limit on our ability to compress data. For a distribution P its *entropy*, $H[P]$, is defined as:

$$H[P] = \sum_j -P(j) \log P(j).$$

`:eqlabel: eq_softmax_reg_entropy`

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution P , we need at least $H[P]$ "nats" to encode it `:cite: Shannon.1948`. If you wonder what a "nat" is, it is the equivalent of bit but when using a code with base e rather than one with base 2. Thus, one nat is $\frac{1}{\log(2)} \approx 1.44$ bit.

4.1.3.2. Surprisal

You might be wondering what compression has to do with prediction. Imagine that we have a stream of data that we want to compress. If it is always easy for us to predict the next token, then this data is easy to compress. Take the extreme example where every token in the stream always takes the same value. That is a very boring data stream! And not only it is boring, but it is also easy to predict. Because the tokens are always the same, we do not have to transmit any information to communicate the contents of the stream. Easy to predict, easy to compress.

However if we cannot perfectly predict every event, then we might sometimes be surprised. Our surprise is greater when an event is assigned lower probability. Claude Shannon settled on $\log \frac{1}{P(j)} = -\log P(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $P(j)$. The entropy defined in `eqref: eq_softmax_reg_entropy` is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

4.1.3.3. Cross-Entropy Revisited

So if entropy is the level of surprise experienced by someone who knows the true probability, then you might be wondering, what is cross-entropy? The cross-entropy from P to Q , denoted $H(P, Q)$, is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P . This is given by $H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$. The lowest possible cross-entropy is achieved when $P = Q$. In this case, the cross-entropy from P to Q is $H(P, P) = H(P)$.

In short, we can think of the cross-entropy classification objective in two ways: (i) as maximizing the likelihood of the observed data; and (ii) as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

✓ 4.1.4. Summary and Discussion


In this section, we encountered the first nontrivial loss function, allowing us to optimize over *discrete* output spaces. Key in its design was that we took a probabilistic approach, treating discrete categories as instances of draws from a probability distribution. As a side effect, we encountered the softmax, a convenient activation function that transforms outputs of an ordinary neural network layer into valid discrete probability distributions. We saw that the derivative of the cross-entropy loss when combined with softmax behaves very similarly to the derivative of squared error; namely by taking the difference between the expected behavior and its prediction. And, while we were only able to scratch the very surface of it, we encountered exciting connections to statistical physics and information theory.

While this is enough to get you on your way, and hopefully enough to whet your appetite, we hardly dived deep here. Among other things, we skipped over computational considerations. Specifically, for any fully connected layer with d inputs and q outputs, the parametrization and computational cost is $\mathcal{O}(dq)$, which can be prohibitively high in practice. Fortunately, this cost of transforming d inputs into q outputs can be reduced through approximation and compression. For instance Deep Fried Convnets :cite: Yang.Moczulski.Denil.ea.2015 uses a combination of permutations, Fourier transforms, and scaling to reduce the cost from quadratic to log-linear. Similar techniques work for more advanced structural matrix approximations :cite: sindhwani2015structured. Lastly, we can use quaternion-like decompositions to reduce the cost to $\mathcal{O}(\frac{dq}{n})$, again if we are willing to trade off a small amount of accuracy for computational and storage cost :cite: Zhang.Tay.Zhang.ea.2021 based on a compression factor n . This is an active area of research. What makes it challenging is that we do not necessarily strive for the most compact representation or the smallest number of floating point operations but rather for the solution that can be executed most efficiently on modern GPUs.

+ Code + Text

Start coding or [generate](#) with AI.

```
!pip install d2l==1.0.3
```

 Show hidden output

✓ COSE474-2024F: Deep Learning

4.2. The Image Classification Dataset

One widely used dataset for image classification is the MNIST dataset (LeCun et al., 1998) of handwritten digits. At the time of its release in the 1990s it posed a formidable challenge to most machine learning algorithms, consisting of 60,000 images of pixels resolution (plus a test dataset of 10,000 images). To put things into perspective, back in 1995, a Sun SPARCStation 5 with a whopping 64MB of RAM and a blistering 5 MFLOPs was considered state of the art equipment for machine learning at AT&T Bell Laboratories. Achieving high accuracy on digit recognition was a key component in automating letter sorting for the USPS in the 1990s. Deep networks such as LeNet-5 (LeCun et al., 1995), support vector machines with invariances (Schölkopf et al., 1996), and tangent distance classifiers (Simard et al., 1998) all could reach error rates below 1%.

For over a decade, MNIST served as the point of reference for comparing machine learning algorithms. While it had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%, making it unsuitable for distinguishing between strong models and weaker ones. Even more, the dataset allows for very high levels of accuracy, not typically seen in many classification problems. This skewed algorithmic development towards specific families of algorithms that can take advantage of clean datasets, such as active set methods and boundary-seeking active set algorithms. Today, MNIST serves as more of a sanity check than as a benchmark. ImageNet (Deng et al., 2009) poses a much more relevant challenge. Unfortunately, ImageNet is too large for many of the examples and illustrations in this book, as it would take too long to train to make the examples interactive. As a substitute we will focus our discussion in the coming sections on the qualitatively similar, but much smaller Fashion-MNIST dataset (Xiao et al., 2017) which was released in 2017. It contains images of 10 categories of clothing at 28×28 pixels resolution.

```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```


✓ 4.2.1. Loading the Dataset

Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it. We can download and read it into memory using built-in framework utilities.

```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

Fashion-MNIST consists of images from 10 categories, each represented by 6000 images in the training dataset and by 1000 in the test dataset. A test dataset is used for evaluating model performance (it must not be used for training). Consequently the training set and the test set contain 60,000 and 10,000 images, respectively.

```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

 (60000, 10000)

The images are grayscale and upsampled to 32×32 pixels in resolution above. This is similar to the original MNIST dataset which consisted of (binary) black and white images. Note, though, that most modern image data has three channels (red, green, blue) and that hyperspectral images can have in excess of 100 channels (the HyMap sensor has 126 channels). By convention we store an image as a $c \times h \times w$ tensor, where c is the number of color channels, h is the height and w is the width.

```
data.train[0][0].shape
```

```
→ torch.Size([1, 32, 32])
```

The categories of Fashion-MNIST have human-understandable names. The following convenience method converts between numeric labels and their names.

✓ 4.2.2. Reading a Minibatch

To make our life easier when reading from the training and test sets, we use the built-in data iterator rather than creating one from scratch. Recall that at each iteration, a data iterator reads a minibatch of data with size **batch_size**. We also randomly shuffle the examples for the training data iterator.

```
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

To see how this works, let's load a minibatch of images by invoking the **train_dataloader** method. It contains 64 images.

```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
→ /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes
  warnings.warn(_create_warning_msg(
  torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

Let's look at the time it takes to read the images. Even though it is a built-in loader, it is not blazingly fast. Nonetheless, this is sufficient since processing images with a deep network takes quite a bit longer. Hence it is good enough that training a network will not be I/O constrained.

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
→ '12.37 sec'
```

✓ 4.2.3. Visualization

We will often be using the Fashion-MNIST dataset. A convenience function `show_images` can be used to visualize the images and the associated labels. Skipping implementation details, we just show the interface below: we only need to know how to invoke `d2l.show_images` rather than how it works for such utility functions.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5): # "@save" is not an allowed annotation - allowed values include [@param,
    """Plot a list of images."""                                     @title, @markdown].
    raise NotImplementedError
```

Let's put it to good use. In general, it is a good idea to visualize and inspect data that you are training on. Humans are very good at spotting oddities and because of that, visualization serves as an additional safeguard against mistakes and errors in the design of experiments. Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-10-d872326f59fa> in <cell line: 8>()
      6     d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
      7     batch = next(iter(data.val_dataloader()))
----> 8     data.visualize(batch)

<ipython-input-10-d872326f59fa> in visualize(self, batch, nrows, ncols, labels)
      3     X, y = batch
      4     if not labels:
----> 5         labels = self.text_labels(y)
      6     d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
      7     batch = next(iter(data.val_dataloader()))

AttributeError: 'FashionMNIST' object has no attribute 'text_labels'
```


✓ 4.2.4. Summary

We now have a slightly more realistic dataset to use for classification. Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories. We will use this dataset in subsequent sections and chapters to evaluate various network designs, from a simple linear model to advanced residual networks. As we commonly do with images, we read them as a tensor of shape (batch size, number of channels, height, width). For now, we only have one channel as the images are grayscale (the visualization above uses a false color palette for improved visibility).

Lastly, data iterators are a key component for efficient performance. For instance, we might use GPUs for efficient image decompression, video transcoding, or other preprocessing. Whenever possible, you should rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

Start coding or [generate](#) with AI.

```
!pip install d2l==1.0.3
```

 Show hidden output

✓ COSE474-2024F: Deep Learning

4.3. The Base Classification Model

```
import torch
from d2l import torch as d2l
```

✓ 4.3.1. The Classifier Class

We define the `Classifier` class below. In the `validation_step` we report both the loss value and the classification accuracy on a validation batch. We draw an update for every `num_val_batches` batches. This has the benefit of generating the averaged loss and accuracy on the whole validation data. These average numbers are not exactly correct if the final batch contains fewer examples, but we ignore this minor difference to keep the code simple.

```
class Classifier(d2l.Module):
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)

@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

✓ 4.3.2. Accuracy

Given the predicted probability distribution y_{hat} , we typically choose the class with the highest predicted probability whenever we must output a hard prediction. Indeed, many applications require that we make a choice. For instance, Gmail must categorize an email into "Primary", "Social", "Updates", "Forums", or "Spam". It might estimate probabilities internally, but at the end of the day it has to choose one among the classes.

When predictions are consistent with the label class y , they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult to optimize accuracy directly (it is not differentiable), it is often the performance measure that we care about the most. It is often *the* relevant quantity in benchmarks. As such, we will nearly always report it when training classifiers.

Accuracy is computed as follows. First, if y_{hat} is a matrix, we assume that the second dimension stores prediction scores for each class. We use `argmax` to obtain the predicted class by the index for the largest entry in each row. Then we **[compare the predicted class with the ground truth y elementwise.]** Since the equality operator `==` is sensitive to data types, we convert y_{hat} 's data type to match that of y . The result is a tensor containing entries of 0 (false) and 1 (true). Taking the sum yields the number of correct predictions.

```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predictions."""
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

✓ 4.3.3. Summary

Classification is a sufficiently common problem that it warrants its own convenience functions. Of central importance in classification is the accuracy of the classifier. Note that while we often care primarily about accuracy, we train classifiers to optimize a variety of other objectives

for statistical and computational reasons. However, regardless of which loss function was minimized during training, it is useful to have a convenience method for assessing the accuracy of our classifier empirically.

Start coding or [generate](#) with AI.

```
!pip install d2l==1.0.3
```

 Show hidden output


✓ COSE474-2024F: Deep Learning

4.4. Softmax Regression Implementation from Scratch

```
import torch
from d2l import torch as d2l
```

✓ 4.4.1. The Softmax

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

 (tensor([[5., 7., 9.]]),
tensor([[6.],
[15.]])

Computing the softmax requires three steps: (i) exponentiation of each term; (ii) a sum over each row to compute the normalization constant for each example; (iii) division of each row by its normalization constant, ensuring that the result sums to 1:


$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}.$$

The (logarithm of the) denominator is called the (log) *partition function*. It was introduced in [statistical physics](#) to sum over all possible states in a thermodynamic ensemble. The implementation is straightforward:

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition # The broadcasting mechanism is applied here
```

For any input X , we turn each element into a nonnegative number. Each row sums up to 1, as is required for a probability. Caution: the code above is not robust against very large or very small arguments. While it is sufficient to illustrate what is happening, you should not use this code verbatim for any serious purpose. Deep learning frameworks have such protections built in and we will be using the built-in softmax going forward.

```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

 (tensor([[0.2247, 0.1639, 0.1748, 0.1561, 0.2804],
[0.1595, 0.1795, 0.2647, 0.1535, 0.2428]]),
tensor([1., 1.]))

✓ 4.4.2. The Model

We now have everything that we need to implement **[the softmax regression model.]** As in our linear regression example, each instance will be represented by a fixed-length vector. Since the raw data here consists of 28×28 pixel images, **[we flatten each image, treating them as vectors of length 784.]** In later chapters, we will introduce convolutional neural networks, which exploit the spatial structure in a more satisfying way.

In softmax regression, the number of outputs from our network should be equal to the number of classes. **(Since our dataset has 10 classes, our network has an output dimension of 10.)** Consequently, our weights constitute a 784×10 matrix plus a 1×10 row vector for the biases. As with linear regression, we initialize the weights w with Gaussian noise. The biases are initialized as zeros.

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
                                requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

Note that we flatten each 28×28 pixel image in the batch into a vector using `reshape` before passing the data through our model.

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

✓ 4.4.3. The Cross-Entropy Loss

This may be the most common loss function in all of deep learning. At the moment, applications of deep learning easily cast as classification problems far outnumber those better treated as regression problems.

Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label. For efficiency we avoid Python for-loops and use indexing instead. In particular, the one-hot encoding in \mathbf{y} allows us to select the matching terms in $\hat{\mathbf{y}}$.

To see this in action we **[create sample data $\mathbf{y_hat}$ with 2 examples of predicted probabilities over 3 classes and their corresponding labels \mathbf{y}]** The correct labels are 0 and 2 respectively (i.e., the first and third class). **[Using \mathbf{y} as the indices of the probabilities in $\mathbf{y_hat}$,]** we can pick out terms efficiently.

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
# tensor([0.1000, 0.5000])

def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
# tensor(1.4979)

@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

✓ 4.4.4. Training

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```


4.4.5. Prediction

```

| 1 | train loss |
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape

```

```

torch.Size([256])

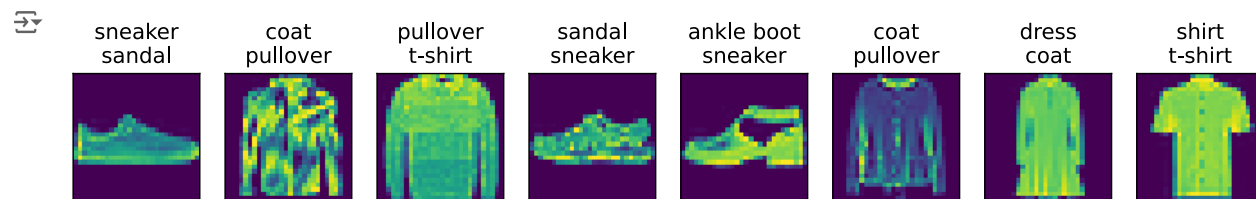
```

We are more interested in the images we label incorrectly. We visualize them by comparing their actual labels (first line of text output) with the predictions from the model (second line of text output).

```

wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)

```




4.4.6. Summary

By now we are starting to get some experience with solving linear regression and classification problems. With it, we have reached what would arguably be the state of the art of 1960–1970s of statistical modeling. In the next section, we will show you how to leverage deep learning frameworks to implement this model much more efficiently.

Start coding or [generate](#) with AI.

```
!pip install d2l==1.0.3
```

 Show hidden output

✓ COSE474-2024F: Deep Learning

5.1. Multilayer Perceptrons

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

✓ 5.1.1. Hidden Layers

5.1.1.1. Limitations of Linear Models

For example, linearity implies the weaker assumption of monotonicity, i.e., that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative). Sometimes that makes sense. For example, if we were trying to predict whether an individual will repay a loan, we might reasonably assume that all other things being equal, an applicant with a higher income would always be more likely to repay than one with a lower income. While monotonic, this relationship likely is not linearly associated with the probability of repayment. An increase in income from 0 to 50,000 likely corresponds to a bigger increase in likelihood of repayment than an increase from 1 million to 1.05 million. One way to handle this might be to postprocess our outcome such that linearity becomes more plausible, by using the logistic map (and thus the logarithm of the probability of outcome).

Note that we can easily come up with examples that violate monotonicity. Say for example that we want to predict health as a function of body temperature. For individuals with a normal body temperature above 37°C (98.6°F), higher temperatures indicate greater risk. However, if the body temperatures drops below 37°C, lower temperatures indicate greater risk! Again, we might resolve the problem with some clever preprocessing, such as using the distance from 37°C as a feature.

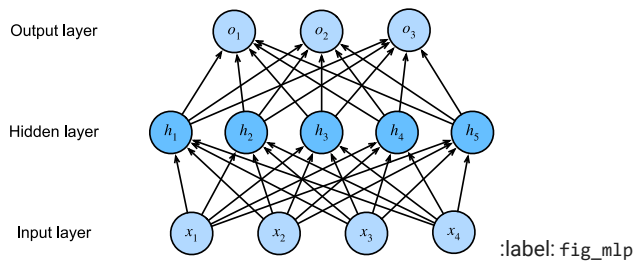
But what about classifying images of cats and dogs? Should increasing the intensity of the pixel at location (13, 17) always increase (or always decrease) the likelihood that the image depicts a dog? Reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats and dogs is to assess the brightness of individual pixels. This approach is doomed to fail in a world where inverting an image preserves the category.

And yet despite the apparent absurdity of linearity here, as compared with our previous examples, it is less obvious that we could address the problem with a simple preprocessing fix. That is, because the significance of any pixel depends in complex ways on its context (the values of the surrounding pixels). While there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable, we simply do not know how to calculate it by hand. With deep neural networks, we used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation.

This problem of nonlinearity has been studied for at least a century (Fisher, 1925). For instance, decision trees in their most basic form use a sequence of binary decisions to decide upon class membership (Quinlan, 1993). Likewise, kernel methods have been used for many decades to model nonlinear dependencies (Aronszajn, 1950). This has found its way into nonparametric spline models (Wahba, 1990) and kernel methods (Schölkopf and Smola, 2002). It is also something that the brain solves quite naturally. After all, neurons feed into other neurons which, in turn, feed into other neurons again (Ramón y Cajal and Azoulay, 1894). Consequently we have a sequence of relatively simple transformations.

5.1.1.2. Incorporating Hidden Layers

We can overcome the limitations of linear models by incorporating one or more hidden layers. The easiest way to do this is to stack many fully connected layers on top of one another. Each layer feeds into the layer above it, until we generate outputs. We can think of the first $L - 1$ layers as our representation and the final layer as our linear predictor. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP* (numref:fig_mlp).



This MLP has four inputs, three outputs, and its hidden layer contains five hidden units. Since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is two. Note that both layers are fully connected. Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer. Alas, we are not quite done yet.

5.1.1.3. From Linear to Nonlinear

As before, we denote by the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ a minibatch of n examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has h hidden units, we denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*. Since the hidden and output layers are both fully connected, we have hidden-layer weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. This allows us to calculate the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\begin{aligned}\mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned}$$

Note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters. So what have we gained in exchange? You might be surprised to find out that—in the model defined above—we *gain nothing for our troubles!* The reason is plain. The hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units. An affine function of an affine function is itself an affine function. Moreover, our linear model was already capable of representing any affine function.

To see this formally we can just collapse out the hidden layer in the above definition, yielding an equivalent single-layer model with parameters $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}.$$

In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function* σ to be applied to each hidden unit following the affine transformation. For instance, a popular choice is the ReLU (rectified linear unit) activation function (Nair.Hinton.2010) $\sigma(x) = \max(0, x)$ operating on its arguments elementwise. The outputs of activation functions $\sigma(\cdot)$ are called *activations*. In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned}$$

Since each row in \mathbf{X} corresponds to an example in the minibatch, with some abuse of notation, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time. Note that we used the same notation for softmax when we denoted a rowwise operation in :numref:subsec_softmax_vectorization. Quite frequently the activation functions we use apply not merely rowwise but elementwise. That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one atop another, yielding ever more expressive models.

5.1.1.4. Universal Approximators

We know that the brain is capable of very sophisticated statistical analysis. As such, it is worth asking, just how powerful a deep network could be. This question has been answered multiple times, e.g., in Cybenko (1989) in the context of MLPs, and in Micchelli (1984) in the context of reproducing kernel Hilbert spaces in a way that could be seen as radial basis function (RBF) networks with a single hidden layer. These (and related results) suggest that even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function. Actually learning that function is the hard part, though. You might think of your neural network as being a bit like the C programming language. The language, like any other modern language, is capable of expressing any computable program. But actually coming up with a program that meets your specifications is the hard part.

Moreover, just because a single-hidden-layer network can learn any function does not mean that you should try to solve all of your problems with one. In fact, in this case kernel methods are way more effective, since they are capable of solving the problem exactly even in infinite dimensional spaces (Kimeldorf and Wahba, 1971, Schölkopf et al., 2001). In fact, we can approximate many functions much more compactly by

using deeper (rather than wider) networks (Simonyan and Zisserman, 2014). We will touch upon more rigorous arguments in subsequent chapters.

5.1.2. Activation Functions

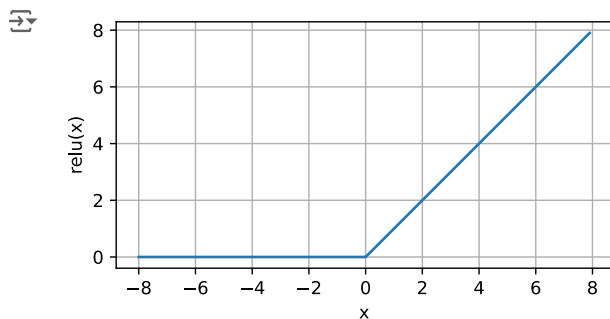
5.1.2.1. ReLU Function

The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the rectified linear unit (ReLU) (Nair and Hinton, 2010). ReLU provides a very simple nonlinear transformation. Given an element x , the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max(x, 0).$$

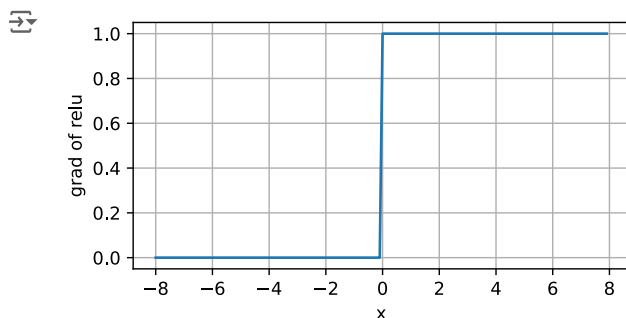
Informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0. To gain some intuition, we can plot the function. As you can see, the activation function is piecewise linear.

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0. We can get away with this because the input may never actually be zero (mathematicians would say that it is nondifferentiable on a set of measure zero). There is an old adage that if subtle boundary conditions matter, we are probably doing (real) mathematics, not engineering. That conventional wisdom may apply here, or at least, the fact that we are not performing constrained optimization (Mangasarian, 1965, Rockafellar, 1970). We plot the derivative of the ReLU function below.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks (more on this later).

Note that there are many variants to the ReLU function, including the parametrized ReLU (pReLU) function (He et al., 2015). This variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

✓ 5.1.2.2. Sigmoid Function

[The **sigmoid function transforms those inputs**] whose values lie in the domain \mathbb{R} , (to **outputs that lie on the interval (0, 1).**) For that reason, the sigmoid is often called a *squashing function*: it squashes any input in the range $(-\infty, \infty)$ to some value in the range $(0, 1)$:

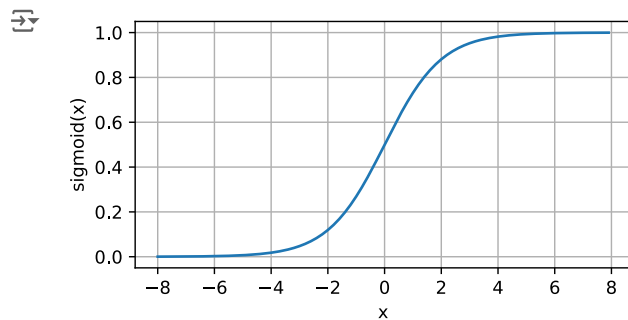
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

In the earliest neural networks, scientists were interested in modeling biological neurons that either *fire* or *do not fire*. Thus the pioneers of this field, going all the way back to McCulloch and Pitts, the inventors of the artificial neuron, focused on thresholding units (McCulloch and Pitts, 1943). A thresholding activation takes value 0 when its input is below some threshold and value 1 when the input exceeds the threshold.

When attention shifted to gradient-based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still widely used as activation functions on the output units when we want to interpret the outputs as probabilities for binary classification problems: you can think of the sigmoid as a special case of the softmax. However, the sigmoid has largely been replaced by the simpler and more easily trainable ReLU for most use in hidden layers. Much of this has to do with the fact that the sigmoid poses challenges for optimization (LeCun et al., 1998) since its gradient vanishes for large positive *and* negative arguments. This can lead to plateaus that are difficult to escape from. Nonetheless sigmoids are important.

Below, we plot the sigmoid function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation.

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

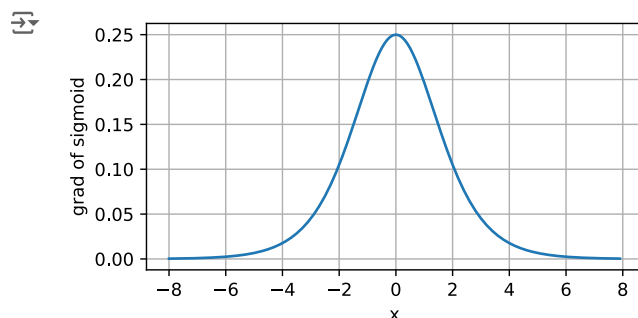


The derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x) (1 - \text{sigmoid}(x)).$$

The derivative of the sigmoid function is plotted below. Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25. As the input diverges from 0 in either direction, the derivative approaches 0.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



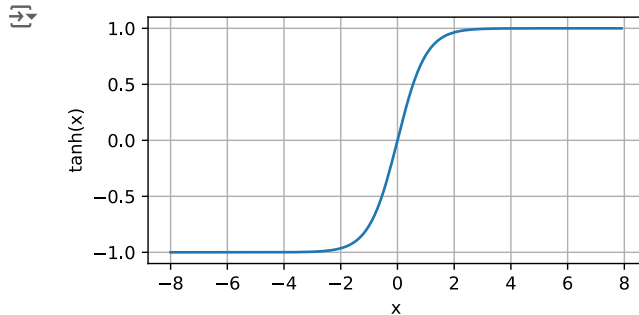
✓ 5.1.2.3. Tanh Function

Like the sigmoid function, [the **tanh (hyperbolic tangent) function also squashes its inputs**], transforming them into elements on the interval (between -1 and 1):

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

We plot the tanh function below. Note that as input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system (Kalman.Kwasny,1992).

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

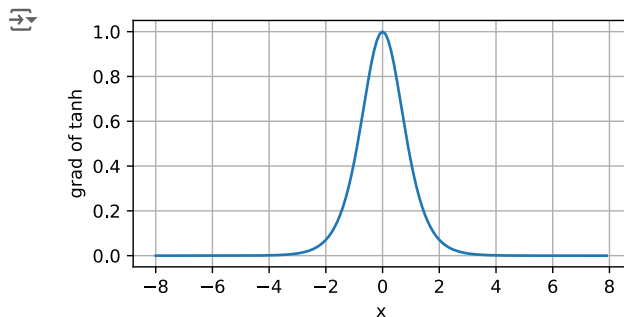


The derivative of the tanh function is:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

It is plotted below. As the input nears 0, the derivative of the tanh function approaches a maximum of 1. And as we saw with the sigmoid function, as input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```




5.1.3. Discussion

- Linear models assume a direct proportionality between inputs and outputs. This limitation is especially problematic for complex problems like image classification. The increasing the intensity of a single pixel does not reliably indicate whether an image depicts a cat or a dog. It fails because it assumes that feature importance remains consistent across all input examples.
- Incorporate with hidden layer can allow the model to capture more complex patterns and interactions between features besides enable the model to build hierarchical representations, learning simple patterns at lower layers and more abstract features at higher layers.
- ReLU has become the default activation function due to its computational efficiency and ability to avoid vanishing gradients.

Start coding or [generate](#) with AI.

Double-click (or enter) to edit

```
!pip install d2l==1.0.3
```

 Show hidden output

✓ COSE474-2024F: Deep Learning

5.2. Implementation of Multilayer Perceptrons

```
import torch
from torch import nn
from d2l import torch as d2l
```

✓ 5.2.1. Implementation from Scratch

✓ 5.2.1.1. Initializing Model Parameters

Recall that Fashion-MNIST contains 10 classes, and that each image consists of a $28 \times 28 = 784$ grid of grayscale pixel values. As before we will disregard the spatial structure among the pixels for now, so we can think of this as a classification dataset with 784 input features and 10 classes. To begin, we will **[implement an MLP with one hidden layer and 256 hidden units.]** Both the number of layers and their width are adjustable (they are considered hyperparameters). Typically, we choose the layer widths to be divisible by larger powers of 2. This is computationally efficient due to the way memory is allocated and addressed in hardware.

Again, we will represent our parameters with several tensors. Note that *for every layer*, we must keep track of one weight matrix and one bias vector. As always, we allocate memory for the gradients of the loss with respect to these parameters.

In the code below we use `nn.Parameter` to automatically register a class attribute as a parameter to be tracked by `autograd` (`numref: sec_autograd`).

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

✓ 5.2.1.2. Model

To make sure we know how everything works, we will implement the ReLU activation ourselves rather than invoking the built-in `relu` function directly.

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

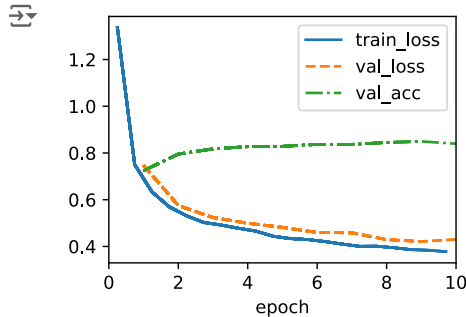
Since we are disregarding spatial structure, we `reshape` each two-dimensional image into a flat vector of length `num_inputs`. Finally, we **(implement our model)** with just a few lines of code. Since we use the framework built-in `autograd` this is all that it takes.

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```


✓ 5.2.1.3. Training

Fortunately, the training loop for MLPs is exactly the same as for softmax regression. We define the model, data, and trainer, then finally invoke the fit method on model and data.

```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



✓ 5.2.2. Concise Implementation

As you might expect, by relying on the high-level APIs, we can implement MLPs even more concisely.

✓ 5.2.2.1. Model

Compared with our concise implementation of softmax regression implementation (Section 4.5), the only difference is that we add *two* fully connected layers where we previously added only one. The first is the hidden layer, the second is the output layer.

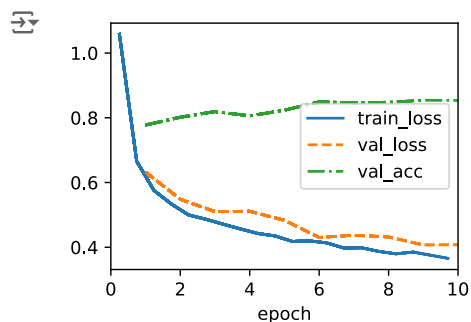
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                  nn.ReLU(), nn.LazyLinear(num_outputs))
```

Previously, we defined forward methods for models to transform input using the model parameters. These operations are essentially a pipeline: you take an input and apply a transformation (e.g., matrix multiplication with weights followed by bias addition), then repetitively use the output of the current transformation as input to the next transformation. However, you may have noticed that no forward method is defined here. In fact, MLP inherits the forward method from the Module class (Section 3.2.2) to simply invoke `self.net(X)` (`X` is input), which is now defined as a sequence of transformations via the Sequential class. The Sequential class abstracts the forward process enabling us to focus on the transformations. We will further discuss how the Sequential class works in Section 6.1.2.

✓ 5.2.2.2. Training

The training loop is exactly the same as when we implemented softmax regression. This modularity enables us to separate matters concerning the model architecture from orthogonal considerations.

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



5.2.3. Summary

Now that we have more practice in designing deep networks, the step from a single to multiple layers of deep networks does not pose such a significant challenge any longer. In particular, we can reuse the training algorithm and data loader. Note, though, that implementing MLPs from scratch is nonetheless messy: naming and keeping track of the model parameters makes it difficult to extend models. For instance, imagine wanting to insert another layer between layers 42 and 43. This might now be layer 42b, unless we are willing to perform sequential renaming. Moreover, if we implement the network from scratch, it is much more difficult for the framework to perform meaningful performance optimizations.

Nonetheless, you have now reached the state of the art of the late 1980s when fully connected deep networks were the method of choice for neural network modeling. Our next conceptual step will be to consider images. Before we do so, we need to review a number of statistical basics and details on how to compute models efficiently.

Start coding or [generate](#) with AI.

✓ COSE474-2024F: Deep Learning

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the deep learning framework.

The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. Surprisingly often, academic papers had to allocate numerous pages to deriving update rules. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

In this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*). To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs. To start, we focus our exposition on a one-hidden-layer MLP with weight decay (ℓ_2 regularization, to be described in subsequent chapters).

5.3.1. Forward Propagation

Forward propagation (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must "pay the cost to be the boss".

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)} \mathbf{x},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h :

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden layer output \mathbf{h} is also an intermediate variable. Assuming that the parameters of the output layer possess only a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector of length q :

$$\mathbf{o} = \mathbf{W}^{(2)} \mathbf{h}.$$

Assuming that the loss function is l and the example label is y , we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y).$$

As we will see the definition of ℓ_2 regularization to be introduced later, given the hyperparameter λ , the regularization term is

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_{\text{F}}^2 + \|\mathbf{W}^{(2)}\|_{\text{F}}^2 \right),$$

:eqlabel: eq_forward-s

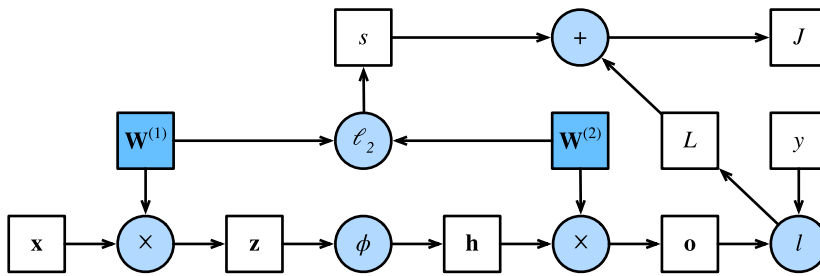
where the Frobenius norm of the matrix is simply the ℓ_2 norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

We refer to J as the *objective function* in the following discussion.

✓ 5.3.2. Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. Fig. 5.3.1 contains the graph associated with the simple network described above, where squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward.



✓ 5.3.3. Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions $Y = f(X)$ and $Z = g(Y)$, in which the input and the output X, Y, Z are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z with respect to X via

$$\frac{\partial Z}{\partial X} = \text{prod} \left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right).$$

Here we use the `prod` operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator `prod` hides all the notational overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in :numref:fig_forward, are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J / \partial \mathbf{W}^{(1)}$ and $\partial J / \partial \mathbf{W}^{(2)}$. To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s :

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1.$$

Next, we compute the gradient of the objective function with respect to variable of the output layer \mathbf{o} according to the chain rule:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q.$$

Next, we calculate the gradients of the regularization term with respect to both parameters:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

Now we are able to calculate the gradient $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}.$$

:eqlabel: eq_backprop-J-h

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer output $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}.$$

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable \mathbf{z} requires that we use the elementwise multiplication operator, which we denote by \odot :

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}).$$

Finally, we can obtain the gradient $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

✓ 5.3.4. Training Neural Networks

When training neural networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed.

Take the aforementioned simple network as an illustrative example. On the one hand, computing the regularization term (5.3.5) during forward propagation depends on the current values of model parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. They are given by the optimization algorithm according to

backpropagation in the most recent iteration. On the other hand, the gradient calculation for the parameter (5.3.11) during backpropagation depends on the current value of the hidden layer output \mathbf{h} , which is given by forward propagation.

Therefore when training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to out-of-memory errors.

✓ 5.3.5. Summary

Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. It proceeds from the input to the output layer. Backpropagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order. When training deep learning models, forward propagation and backpropagation are interdependent, and training requires significantly more memory than prediction.

Start coding or [generate](#) with AI.