In [6]:
```
pip install torch
```

```
Collecting torch
  Downloading torch-2.2.2-cp311-none-macosx_10_9_x86_64.whl.metadata (25 kB)
Collecting filelock (from torch)
  Downloading filelock-3.16.0-py3-none-any.whl.metadata (3.0 kB)
Requirement already satisfied: typing-extensions>=4.8.0 in /Library/Frameworks/Python.framework/Versions/3.11/li
b/python3.11/site-packages (from torch) (4.12.2)
Collecting sympy (from torch)
  Downloading sympy-1.13.2-py3-none-any.whl.metadata (12 kB)
Collecting networkx (from torch)
  Downloading networkx-3.3-py3-none-any.whl.metadata (5.1 kB)
Requirement already satisfied: jinja2 in /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages (from torch) (3.1.4)
Collecting fsspec (from torch)
  Downloading fsspec-2024.9.0-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: MarkupSafe>=2.0 in /Library/Frameworks/Python.framework/Versions/3.11/lib/python3
.11/site-packages (from jinja2->torch) (2.1.5)
Collecting mpmath<1.4,>=1.1.0 (from sympy->torch)
  Downloading mpmath-1.3.0-py3-none-any.whl.metadata (8.6 kB)
Downloading torch-2.2.2-cp311-none-macosx_10_9_x86_64.whl (150.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 150.8/150.8 MB 719.0 kB/s eta 0:00:0000:0100:06
Downloading filelock-3.16.0-py3-none-any.whl (16 kB)
Downloading fsspec-2024.9.0-py3-none-any.whl (179 kB)
Downloading networkx-3.3-py3-none-any.whl (1.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 576.0 kB/s eta 0:00:00a 0:00:01
Downloading sympy-1.13.2-py3-none-any.whl (6.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.2/6.2 MB 675.4 kB/s eta 0:00:00a 0:00:01
Downloading mpmath-1.3.0-py3-none-any.whl (536 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 536.2/536.2 kB 802.7 kB/s eta 0:00:00:--:--
Installing collected packages: mpmath, sympy, networkx, fsspec, filelock, torch
Successfully installed filelock-3.16.0 fsspec-2024.9.0 mpmath-1.3.0 networkx-3.3 sympy-1.13.2 torch-2.2.2
Note: you may need to restart the kernel to use updated packages.
```

In [4]:
```python
import torch
x = torch.arange(12, dtype=torch.float32)
x
```

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

In [5]:
```python
x.numel()
```

Out[5]:
```
12
```

In [7]:
```python
x.shape
```

Out[7]:
```
torch.Size([12])
```

In [8]:
```python
X = x.reshape(3, 4) #어차피 나눠지니까 둘 중 하나만 알면 나머지에 -1넣으면됨
X
```

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

In [9]:
```python
torch.zeros((2, 3, 4))
```

Out[9]:
```
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.]]])
```

In [10]:
```python
torch.ones((2, 3, 4))
```

Out[10]:
```
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.]]])
```

In [11]:
```python
torch.randn(3, 4)
```

Out[11]:
```
tensor([[ 0.0169,  0.0614,  0.1916, -2.0591],
        [-0.5592, -0.2677,  0.8810,  0.0313],
        [-1.1016, -1.3585, -0.9030,  0.2280]])
```

In [12]:
```python
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

Out[12]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

In [13]:
```python
X[-1], X[1:3]
```

Out[13]:
```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

In [14]:
```python
X[1, 2] = 17
X
```

Out[14]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [15]:
```python
X[:2, :] = 12
X
```

Out[15]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

In [16]:
```python
torch.exp(x)
```

Out[16]:
```
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])
```

In [17]:
```python
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
```

Out[17]:
```
(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.]))
```

In [18]:
```python
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)
```

Out[18]:
```
(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.]]))
```

In [19]:
```python
X == Y
```

Out[19]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

In [20]:
```python
X.sum()
```

Out[20]:
```
tensor(66.)
```

In [21]:
```python
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

Out[21]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [22]:
```python
a + b
```

Out[22]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

In [23]:
```python
before = id(Y)
Y = Y + X
```

```
id(Y) == before
```

Out[23]: False

In [24]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 4698319152
id(Z): 4698319152
```

In [25]:
```
before = id(X)
X += Y
id(X) == before
```

Out[25]: True

In [26]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[26]: (numpy.ndarray, torch.Tensor)

In [27]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[27]: (tensor([3.5000]), 3.5, 3.5, 3)

Discussion and Takeaway message: In this section, we learned about basic grammar and functions that deal with arrangements with pytorch. What I learned newly was that when rescuing, even if only one of the two numbers was written and the rest of the numbers were filled in with -1, the code still performed well. In addition, I learned a new function 'torch.cat ()' that connects two or more tensors according to a given dimension. It was a unit that also provided tips for managing memory efficiently.

In [2]:
```python
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''')
```

In [4]:
```python
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
```

In [5]:
```python
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
```

In [6]:
```python
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
```

In [7]:
```python
import torch

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

Out[7]:
```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

Discussion: First, I created a data file, and I was surprised to find that, despite using what I felt was a somewhat forced method, the CSV file came out correctly when printed. (I used a multi-line string for the input and separated the fields with newlines, which is why I described it as a forced method.) Then, I split the 'Rooftype' column into 'Slate' and 'NaN', dividing the table into True/False values and converted it into a tensor. Through this process, I realized that human-friendly data and machine-friendly data have different goals.

In [2]:
```python
import torch
```

In [3]:
```python
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[3]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

In [4]:
```python
x = torch.arange(3)
x
```

Out[4]:
```
tensor([0, 1, 2])
```

In [5]:
```python
x[2]
```

Out[5]:
```
tensor(2)
```

In [6]:
```python
len(x)
```

Out[6]:
```
3
```

In [7]:
```python
x.shape
```

Out[7]:
```
torch.Size([3])
```

In [8]:
```python
A = torch.arange(6).reshape(3,2)
A
```

Out[8]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [9]:
```python
A.T
```

Out[9]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

In [10]:
```python
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

Out[10]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

In [11]:
```python
torch.arange(24).reshape(2,3,4)
```

Out[11]:
```
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]]])
```

In [12]:
```python
A = torch.arange(6, dtype=torch.float32).reshape(2,3)
B = A.clone()
A, A+B
```

Out[12]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [13]:
```python
A*B
```

Out[13]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [14]:
```python
a = 2
X = torch.arange(24).reshape(2,3,4)
a+X, (a*x).shape
```

Out[14]:
```
(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([3]))
```

In [15]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

Out[15]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

In [16]:
```
A.shape, A.sum()
```

Out[16]:
```
(torch.Size([2, 3]), tensor(15.))
```

In [17]:
```
A.shape, A.sum(axis=0).shape
```

Out[17]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [18]:
```
A.shape, A.sum(axis=1).shape
```

Out[18]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [20]:
```
A.sum(axis=[0,1]) == A.sum()
```

Out[20]:
```
tensor(True)
```

In [21]:
```
A.mean(), A.sum() / A.numel()
```

Out[21]:
```
(tensor(2.5000), tensor(2.5000))
```

In [22]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[22]:
```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

In [23]:
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[23]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [24]:
```
A/sum_A
```

Out[24]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [25]:
```
A.cumsum(axis=0)
```

Out[25]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

In [26]:
```
y = torch.ones(3, dtype=torch.float32)
x,y,torch.dot(x,y)
```

Out[26]:
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

In [27]:
```
torch.sum(x*y)
```

Out[27]:
```
tensor(3.)
```

In [29]:
```
A.shape, x.shape, torch.mv(A,x), A@x
```

Out[29]:
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

In [30]:
```
B = torch.ones(3,4)
torch.mm(A,B), A@B
```

Out[30]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

In [31]:
```
u = torch.tensor([3.0,-4.0])
torch.norm(u)
```

Out[31]:
```
tensor(5.)
```

In [32]:
```
torch.abs(u).sum()
```

Out[32]:
```
tensor(7.)
```

In [34]:
```
torch.norm(torch.ones((4,9)))
```

Out[34]:
```
tensor(6.)
```

Discussion: Linear algebra was one of the first subjects I studied in my first semester, but unfortunately, I wasn't very focused on my studies back then, so there are many gaps in my understanding of the concepts. Nevertheless, while working on this unit's exercises, I was able to revisit parts that I had forgotten, such as matrix transposition. The most useful part of this unit was realizing how easily matrix-vector and matrix-matrix multiplication can be done using the "@" operator.

In [1]:
```python
import torch
```

In [2]:
```python
x = torch.arange(4.0)
x
```

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

In [3]:
```python
x.requires_grad_(True)
x.grad
```

In [4]:
```python
y = 2 * torch.dot(x,x)
y
```

Out[4]: `tensor(28., grad_fn=<MulBackward0>)`

In [5]:
```python
y.backward()
x.grad
```

Out[5]: `tensor([ 0.,  4.,  8., 12.])`

In [6]:
```python
x.grad == 4*x
```

Out[6]: `tensor([True, True, True, True])`

In [7]:
```python
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

Out[7]: `tensor([1., 1., 1., 1.])`

In [8]:
```python
x.grad.zero_()
y = x*x
y.backward(gradient=torch.ones(len(y)))
x.grad
```

Out[8]: `tensor([0., 2., 4., 6.])`

In [9]:
```python
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

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

Out[9]: `tensor([True, True, True, True])`

In [10]:
```python
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

Out[10]: `tensor([True, True, True, True])`

In [11]:
```python
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
```

In [12]:
```python
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [13]:
```python
a.grad == d / a
```

Out[13]: `tensor(True)`

Discussion: This unit covers basic concepts related to automatic differentiation, but compared to the previous sections, I found it a bit challenging to understand the code at first. Perhaps it's because I skipped section 2_4, and going through that might have made this one easier to grasp. Nevertheless, I found it fascinating how we can create y by taking the dot product of a tensor with itself and multiplying it by 2 (in this case, y is a scalar value), then use the .backward() function to compute the gradients and check the results. Instead of trying to fully grasp everything in this unit, my goal is to understand what .backward() and .grad mean, as well as remember that .grad.zero_()

serves to reset the gradients.

In [1]:
```python
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

In [2]:
```python
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [3]:
```python
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

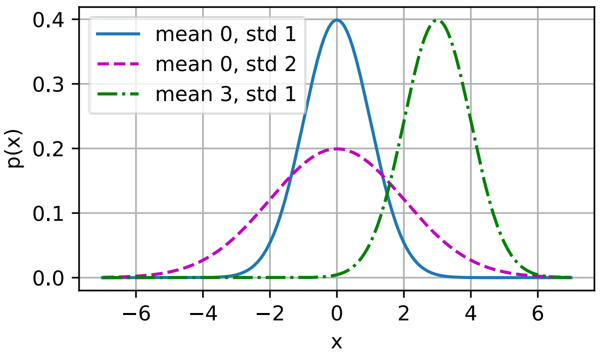
Out[3]: '0.08976 sec'

In [4]:
```python
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[4]: '0.00038 sec'

In [5]:
```python
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)
```

In [6]:
```python
x = np.arange(-7, 7, 0.01)

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])
```



Discussion: The content covered the basic concepts of linear regression, what a loss function is, and how to find the optimal weights through stochastic gradient descent. I came to understand that the ultimate goal is to adjust each parameter in such a way as to minimize the value of the loss function. In the coding part, what surprised me the most was that performing addition element-wise versus adding an entire vector at once with the + operator showed a significant difference in time, with the latter being much faster. Honestly, since the result is also a vector, I still don't quite understand why adding it all at once is so much faster, considering that each element needs to be summed anyway. Additionally, we covered topics on normal distribution and squared loss, and thankfully, I was already somewhat familiar with these concepts.

In [1]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

In [2]:
```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [3]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [4]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

```
Class attribute "b" is 1
```

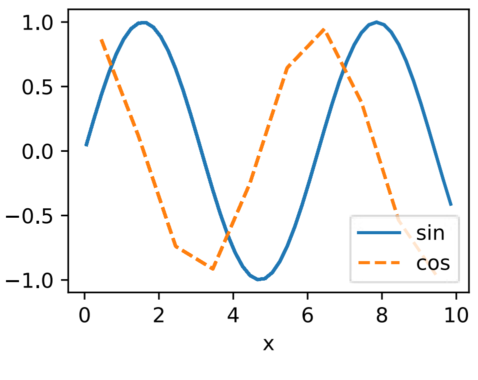
In [5]:
```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

In [6]:
```
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
```

In [7]:
```
class ProgressBoard(d2l.HyperParameters):
    """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', 'C3'],
                 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
```

In [8]:
```
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)
```



In [9]:
```
class Module(nn.Module, d2l.HyperParameters):
    """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):
```

```python
        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 inited'
        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
```

In [10]:

```python
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)
```

In [11]:

```python
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
```

Discussion: This section introduces an object-oriented approach to deep learning, focusing on the modular design of components like 'Module', 'DataModule', and 'Trainer'. By defining reusable classes, the implementation becomes cleaner and more adaptable to various projects. However, since I am not very familiar with object-oriented programming, I found it a bit challenging to fully understand the code structure and interactions between the classes. Despite the initial difficulty, I can see how these approaches promotes better scalability and maintainability in real projects. And it was personally fascinating that the graph was drawn as if dancing in the sine and cosine functions.

In [1]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

In [3]:
```
class LinearRegressionScratch(d2l.Module):  #@save
    """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)
```

In [4]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

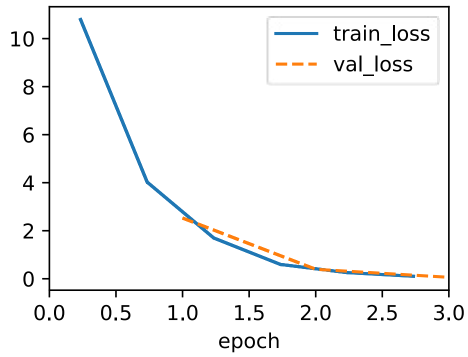
In [5]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

In [6]:
```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch SGD."""
    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_()
```

In [7]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In [8]:
```
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
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: #나중
                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
```

In [9]:
```
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)
```



In [10]:
```
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([ 0.1526, -0.2056])
error in estimating b: tensor([0.2406])
```

Discussion: Before diving into the main content, I had maintained the stance of not typing out comments, which is why I didn't include #@save. However, since it kept appearing repeatedly, I decided to look it up and learned that it indicates sections of code that can be reused multiple times. Thus, unlike other comments, I decided to include it. This section focused on implementing a linear regression model and training the model's weights and bias using SGD. I'm curious how the learning rate (lr) in the SGD class will affect the final result, as this hasn't been covered yet. The loss function used is MSE, which I was happy to see because it's a function I learned about in computational mathematics before. During the training process, I came across the new concept of an "epoch." After the data was trained, the estimated parameters and the actual parameters were compared by printing out the error between them.

In [1]:
```
# no code
```

Discussion: In this chapter, I learned about softmax regression, a method for estimating the probability that a given input belongs to a specific class. I had previously read a popular science book on deep learning, where I vaguely understood that the softmax function outputs values between 0 and 1, sums them to 1, and that these values represent probabilities. However, I didn't fully understand the detailed process of how the loss function is calculated using one-hot encoding. This was something new I learned. I also found it interesting that Cross-Entropy Error (CEE) is used as the loss function.

In [1]:
```python
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

In [2]:
```python
class FashionMNIST(d2l.DataModule):  #@save
    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)
```

In [3]:
```python
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../data/Fas
hionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████████████████████| 26421880/26421880 [00:05<00:00, 4731535.78it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ../data/Fas
hionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████████████████████████████| 29515/29515 [00:00<00:00, 99545.58it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ../data/Fash
ionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|████████████████████████████| 4422102/4422102 [00:01<00:00, 3521825.30it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ../data/Fash
ionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████████████████████████████| 5148/5148 [00:00<00:00, 3375375.49it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

Out[3]:
```
(60000, 10000)
```

In [4]:
```python
data.train[0][0].shape
```

Out[4]:
```
torch.Size([1, 32, 32])
```

In [5]:
```python
@d2l.add_to_class(FashionMNIST)  #@save
def text_labels(self, indices):
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

In [6]:
```python
@d2l.add_to_class(FashionMNIST)  #@save
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)
```

In [7]:
```python
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

In [8]:
```python
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

Out[8]:
```
'5.13 sec'
```

In [9]:
```python
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
```

```
    raise NotImplementedError
```

In [10]:
```
@d2l.add_to_class(FashionMNIST)  #@save
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)
```

Discussion: In this chapter, we worked on loading and visualizing images using an actual dataset. Usually, examples tend to use the handwritten digits dataset, so this was a bit different and interesting. The code for resizing the data to a specified size and loading it in batches using 'get_dataloader' felt somewhat unfamiliar but fascinating. The performance measurement showed a result of 5.13 seconds, which made me realize how important time is in these tasks.

In [1]:

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

In [2]:

```python
class Classifier(d2l.Module):  #@save
    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)
```

In [3]:

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

In [4]:

```python
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    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
```

Discussion: In this chapter, I implemented a classification model and created a 'classifier class' for it. The class has three methods, each playing its role by incorporating elements from what we've previously covered. However, the @d2l.add_to_class syntax still feels quite unfamiliar.

In [1]:
```
import torch
from d2l import torch as d2l
```

In [2]:
```
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)
```

Out[2]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

In [3]:
```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition
```

In [4]:
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

Out[4]:
```
(tensor([[0.2128, 0.1564, 0.1422, 0.1940, 0.2945],
         [0.1785, 0.2052, 0.1561, 0.2170, 0.2432]]),
 tensor([1., 1.]))
```

In [5]:
```
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]
```

In [6]:
```
@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)
```

In [7]:
```
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]
```

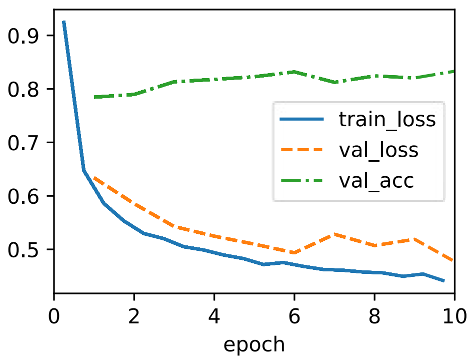
Out[7]:
```
tensor([0.1000, 0.5000])
```

In [8]:
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[8]:
```
tensor(1.4979)
```

In [9]:
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

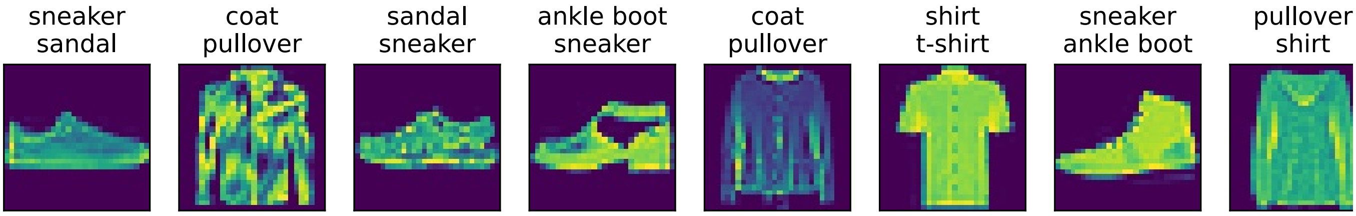
In [10]:
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



In [11]:
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[11]: torch.Size([256])

In [12]:
```
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)
```



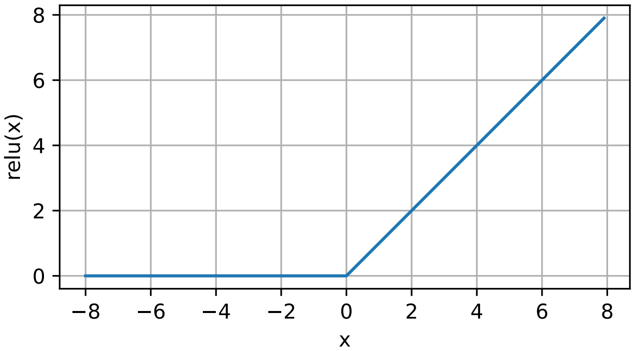
Discussion: In this section, I implemented the softmax function and went through the process of actually training a model using the FashionMNIST dataset. Much of it involved reapplying concepts we had previously covered, with only slight changes in format, so it wasn't too difficult to understand. Watching the graph as the loss values steadily decreased during training was interesting. The part where incorrect predictions were visualized was something I hadn't seen before, but it was fascinating to note that there weren't any completely absurd mistakes, like confusing shoes for a top. The errors made were more in line with mistakes a tired person might make, which made me pay close attention to this aspect.

In [ ]:

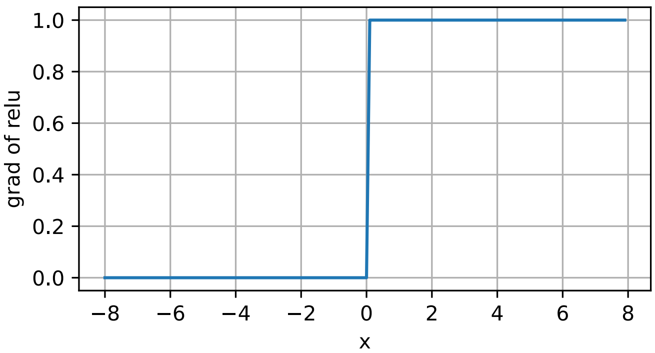
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [1]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

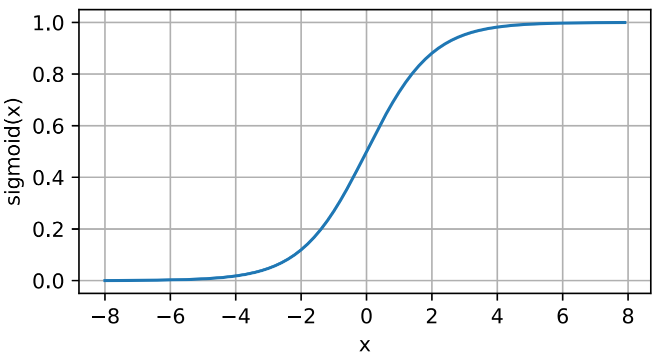
In [2]:
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



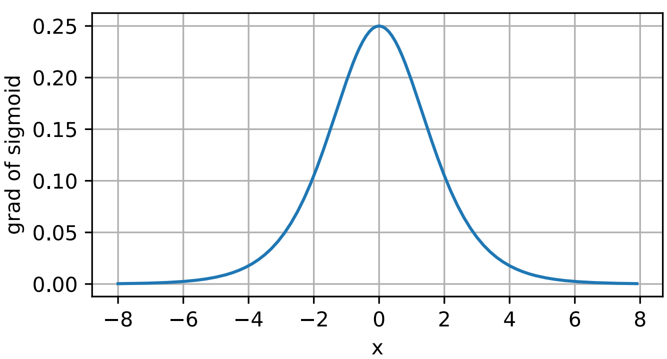
In [3]:
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



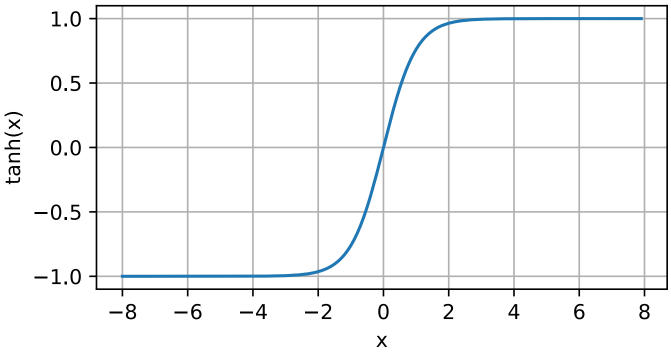
In [4]:
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



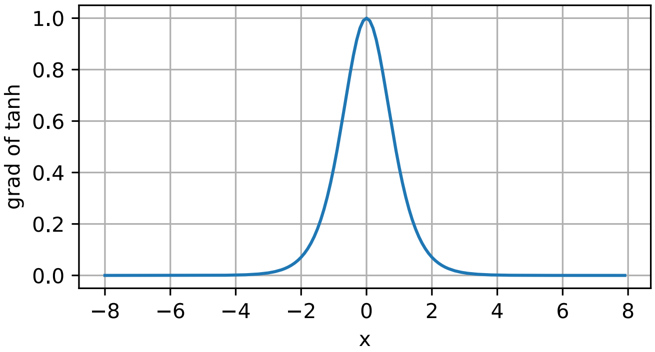
In [6]:
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



In [8]:
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



In [9]:
```
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))
```



Discussion: I learned about multilayer networks in this chapter, and at first, I wondered why we needed to use such complex functions. However, I came to understand that no matter how many linear functions are combined, they cannot break free from linearity, so it's important to use activation functions to introduce non-linearity. Among the various activation functions, I personally like the sigmoid function the most.:

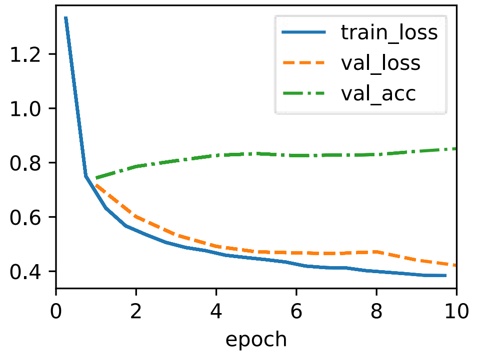
In [1]:
```python
import torch
from torch import nn
from d2l import torch as d2l
```

In [2]:
```python
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))
```

In [3]:
```python
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

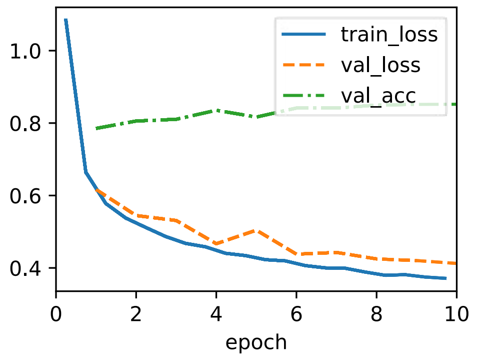
In [4]:
```python
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

In [5]:
```python
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)
```



In [6]:
```python
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))
```

In [7]:
```python
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



Discussion: In this chapter, I had the opportunity to actually implement a Multilayer Perceptron and conduct experiments. In the training process before the Concise Implementation, the example on this website showed fluctuations that gradually reduced, but in my case, both the val_loss and val_acc steadily approached the target without any noticeable dips. This made me realize that the intermediate process can vary greatly from person to person and with each run. Once again, I was reminded that this isn't about finding a definitive answer, but rather an artificial intelligence subject.

In [1]:

```
#No code
```

Discussion: When I first encountered the concept of a computational graph, I thought, "Is this really necessary?" However, I came to realize that it's a very important concept because it allows us to intuitively and clearly understand how a single operation or value affects the final result, even when the calculations become very complex. I once implemented a simple version of a computational graph, and although it was difficult to implement, I remember that it was faster in execution compared to the general method of calculating gradients.