2.1

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

In [2]:
```
x = torch.arange(12, dtype=torch.float32)
x
```

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

In [3]:
```
x.numel()
```

Out[3]:
```
12
```

In [4]:
```
x.shape
```

Out[4]:
```
torch.Size([12])
```

In [5]:
```
X = x.reshape(3, 4)
X
```

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

In [6]:
```
torch.zeros((2, 3, 4))
```

Out[6]:
```
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 [7]:
```
torch.ones((2, 3, 4))
```

Out[7]:
```
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 [8]:
```
torch.randn(3, 4)
```

Out[8]:
```
tensor([[ 2.9372, -0.2109, -0.7924, -0.3183],
        [ 0.5128,  0.2158, -0.4106, -2.1554],
        [-0.0734,  2.0788, -0.4792,  0.1684]])
```

In [9]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

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

In [10]:
```
X[-1], X[1:3]
```

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

In [11]:
```
X[1, 2] = 17
X
```

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

In [12]:
```
torch.exp(x)
```

Out[12]:
```
tensor([1.0000e+00, 2.7183e+00, 7.3891e+00, 2.0086e+01, 5.4598e+01, 1.4841e+02,
        2.4155e+07, 1.0966e+03, 2.9810e+03, 8.1031e+03, 2.2026e+04, 5.9874e+0
4])
```

In [13]:
```
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[13]:
```
(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 [14]:
```
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[14]:
```
(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 [15]:
```
X == Y
```

Out[15]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

In [16]:
```
X.sum()
```

Out[16]:
```
tensor(66.)
```

In [17]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

Out[17]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [18]:
```
a + b
```

Out[18]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

In [19]:
```
before = id(Y)
Y = Y + X
id(Y) == before
```

Out[19]:
```
False
```

In [20]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 2763793282208
id(Z): 2763793282208
```

In [21]:
```
before = id(X)
X += Y
id(X) == before
```

Out[21]:
```
True
```

In [22]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[22]:
```
(numpy.ndarray, torch.Tensor)
```

In [23]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[23]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

In [24]:
```
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]])
```

In [25]:
```
result_comparison = (X < Y) | (X > Y)
print('2D comparison result:\n', result_comparison)
```

```
2D comparison result:
 tensor([[ True, False,  True, False],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]])
```

In [26]:
```
X_3d = X.reshape((1, 3, 4))
Y_3d = Y.reshape((1, 3, 4))
```

In [27]:
```
result_comparison_3d = (X_3d < Y_3d) | (X_3d > Y_3d)
print('3D comparison result:\n', result_comparison_3d)
```

```
3D comparison result:
 tensor([[[ True, False,  True, False],
         [ True,  True,  True,  True],
         [ True,  True,  True,  True]]])
```

Discussion and Takeaway Messages for 2.1:

- Element-wise Comparisons: The operation (X < Y) | (X > Y) is an element-wise comparison between two tensors. This means that each element in tensor X is compared with the corresponding element in tensor Y, and the result is a tensor of boolean values (True or False). This kind of operation is essential in situations where you need to perform conditional logic across multiple values of the tensors.

- Broadcasting Mechanism: The transition from 2D to 3D tensors demonstrates PyTorch's broadcasting mechanism, which allows operations on tensors of different shapes. In this case, reshaping the tensors to add an extra dimension (from 2D to 3D) does not affect the outcome of the comparison because broadcasting aligns the shapes, making the comparison possible without additional effort.

- Shape Flexibility: The broadcasting mechanism adds flexibility in tensor operations, allowing you to perform operations between tensors that don't necessarily have the same shape, as long as certain alignment rules are met. This feature is particularly useful for neural network operations where tensors with different dimensions often need to interact.

- Performance Implications: Element-wise operations on tensors, especially when leveraging broadcasting, are highly efficient. PyTorch is optimized for such operations, making it an excellent choice for handling large datasets, performing matrix operations, or working with multi-dimensional data. However, it's important to ensure that the shapes are compatible to avoid unnecessary errors.

2.2

In [28]:
```
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 [29]:
```
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 [30]:
```
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 [31]:
```
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 [32]:
```
import torch

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

Out[32]:
```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

1.Try loading datasets, e.g., Abalone from the UCI Machine Learning Repository and inspect their properties. What fraction of them has missing values? What fraction of the variables is numerical, categorical, or text?

can't download the dataset, because it is not available.

2.Try out indexing and selecting data columns by name rather than by column number. The pandas documentation on indexing has further details on how to do this.

don't have dataset.

3.How large a dataset do you think you could load this way? What might be the limitations? Hint: consider the time to read the data, representation, processing, and memory footprint. Try this out on your laptop. What changes if you try it out on a server?

- The size of the dataset you can load depends on available RAM, disk I/O speed, and processing power (CPU/GPU). On laptops with limited RAM (e.g., 8GB–16GB), large datasets might exceed memory or slow down due to disk read times. Servers with more memory, faster storage (NVMe SSDs), and powerful GPUs can handle larger datasets faster. Techniques like lazy loading and batch processing help mitigate these limitations.

4. How would you deal with data that has a very large number of categories? What if the category labels are all unique? Should you include the latter?

- For data with a large number of categories, you can apply techniques like embedding layers (for categorical data) or one-hot encoding, depending on the model. If the category labels are all unique (e.g., IDs or unique identifiers), it's often best not to include them directly, as they don't provide meaningful patterns for the model. Instead, consider excluding or transforming them, such as by grouping categories, hashing, or using frequency-based approaches to reduce dimensionality and avoid overfitting.

5. What alternatives to pandas can you think of? How about loading NumPy tensors from a file? Check out Pillow, the Python Imaging Library.

- Alternatives to Pandas include Dask, Vaex, and PySpark for handling larger datasets and parallel processing. For loading NumPy tensors from a file, you can use HDF5 (via h5py), Zarr, or Numpy's own load() and save() functions. For image data, Pillow (PIL) is widely used for opening, manipulating, and saving images. It can be combined with NumPy to convert images into arrays/tensors for deep learning tasks.

2.3

In [33]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[33]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

In [34]:
```
x = torch.arange(4)
x
```

Out[34]:
```
tensor([0, 1, 2, 3])
```

In [35]:
```
x[3]
```

Out[35]:
```
tensor(3)
```

In [36]:
```
len(x)
```

Out[36]:
```
4
```

In [37]:
```
x.shape
```

Out[37]:
```
torch.Size([4])
```

In [38]:
```
A = torch.arange(20).reshape(5, 4)
A
```

Out[38]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
```

In [39]:
```
A.T
```

Out[39]:
```
tensor([[ 0,  4,  8, 12, 16],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19]])
```

In [40]:
```
B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

Out[40]:
```
tensor([[1, 2, 3],
        [2, 0, 4],
        [3, 4, 5]])
```

In [41]:
```
B == B.T
```

Out[41]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

In [42]:
```
X = torch.arange(24).reshape(2, 3, 4)
X
```

Out[42]:
```
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 [43]:
```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

Out[43]:
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]),
 tensor([[ 0.,  2.,  4.,  6.],
         [ 8., 10., 12., 14.],
         [16., 18., 20., 22.],
         [24., 26., 28., 30.],
         [32., 34., 36., 38.]]))
```

In [44]:
```
A * B
```

Out[44]:
```
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
```

In [45]:
```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[45]:
```
(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]))
```

In [46]:
```
x = torch.arange(4, dtype=torch.float32)
x, x.sum()
```

Out[46]:
```
(tensor([0., 1., 2., 3.]), tensor(6.))
```

In [47]:
```
A.shape, A.sum()
```

Out[47]:
```
(torch.Size([5, 4]), tensor(190.))
```

In [48]:
```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

Out[48]:
```
(tensor([40., 45., 50., 55.]), torch.Size([4]))
```

In [49]:
```
A.sum(axis=[0, 1])  # Same as `A.sum()`
```

Out[49]:
```
tensor(190.)
```

In [50]:
```
A.mean(), A.sum() / A.numel()
```

Out[50]:
```
(tensor(9.5000), tensor(9.5000))
```

In [51]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[51]:
```
(tensor([ 8.,  9., 10., 11.]), tensor([ 8.,  9., 10., 11.]))
```

In [52]:
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

Out[52]:
```
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
```

In [53]:
```
A / sum_A
```

Out[53]:
```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

In [54]:
```
A.cumsum(axis=0)
```

Out[54]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
```

In [55]:
```
y = torch.ones(4, dtype=torch.float32)
x, y, torch.dot(x, y)
```

Out[55]:
```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

In [56]:
```
torch.sum(x * y)
```

Out[56]:
```
tensor(6.)
```

In [57]:
```
A.shape, x.shape, torch.mv(A, x)
```

Out[57]:
```
(torch.Size([5, 4]), torch.Size([4]), tensor([ 14.,  38.,  62.,  86., 110.]))
```

In [58]:
```
B = torch.ones(4, 3)
torch.mm(A, B)
```

Out[58]:
```
tensor([[ 6.,  6.,  6.],
        [22., 22., 22.],
        [38., 38., 38.],
        [54., 54., 54.],
        [70., 70., 70.]])
```

In [59]:
```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

Out[59]:
```
tensor(5.)
```

In [60]:
```
torch.abs(u).sum()
```

Out[60]:
```
tensor(7.)
```

In [61]:
```
torch.norm(torch.ones((4, 9)))
```

Out[61]:
```
tensor(6.)
```

Exercises

We defined the tensor X of shape (2, 3, 4) in this section. What is the output of len(X)? Write your answer without implementing any code, then check your answer using code.

In [62]:
```
x = torch.arange(24).reshape(2, 3, 4)
x
```

Out[62]:
```
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 [63]:
```
len(x)
```

Out[63]:
```
2
```

For a tensor X of arbitrary shape, does len(X) always correspond to the length of a certain axis of X? What is that axis?

In [64]:
```
x = torch.arange(24).reshape(3, 2, 4)
x
```

Out[64]:
```
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 [65]:
```
len(x)
```

Out[65]:
```
3
```

In [66]:
```
x = torch.arange(24).reshape(4, 3, 2)
x
```

Out[66]:
```
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 [67]:
```
len(x)
```

Out[67]:
```
4
```

Run A / A.sum(axis=1) and see what happens. Can you analyze the reason?

In [68]:
```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
```

In [69]:
```
## A / A.sum(axis=1)
```

In [70]:
```
A
```

Out[70]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
```

In [71]:
```
A.sum(axis=1)
```

Out[71]:
```
tensor([ 6., 22., 38., 54., 70.])
```

- The reason is that the matrix A has 4 columns, while A has 5.

Consider a tensor with shape (2, 3, 4). What are the shapes of the summation outputs along axis 0, 1, and 2?

In [72]:
```
X = torch.arange(24).reshape(4, 3, 2)
X
```

Out[72]:
```
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 [73]:
```
X.sum(axis=0)
```

Out[73]:
```
tensor([[36, 40],
        [44, 48],
        [52, 56]])
```

In [74]:
```
X.sum(axis=1)
```

Out[74]:
```
tensor([[ 6,  9],
        [24, 27],
        [42, 45],
        [60, 63]])
```

In [75]:
```
X.sum(axis=2)
```

Out[75]:
```
tensor([[ 1,  5,  9],
        [13, 17, 21],
        [25, 29, 33],
        [37, 41, 45]])
```

Feed a tensor with 3 or more axes to the linalg.norm function and observe its output. What does this function compute for tensors of arbitrary shape?

In [76]:
```
X = torch.arange(24).reshape(4, 3, 2)
X
```

Out[76]:
```
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 [77]:
```
import numpy as np

np.linalg.norm(X)
```

Out[77]:
```
65.75712889109438
```

In [78]:
```
X = torch.arange(48).reshape(4, 3, 2, 2)
X
```

Out[78]:
```
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]]],


        [[[24, 25],
          [26, 27]],

         [[28, 29],
          [30, 31]],

         [[32, 33],
          [34, 35]]],


        [[[36, 37],
          [38, 39]],

         [[40, 41],
          [42, 43]],

         [[44, 45],
          [46, 47]]]])
```

In [79]:
```
np.linalg.norm(X)
```

Out[79]:
```
188.9973544788392
```

- the norm function computes the norm of any given tensor

2.5

In [80]:
```
x = torch.arange(4.0)
x
```

Out[80]:
```
tensor([0., 1., 2., 3.])
```

In [81]:
```
x.requires_grad_(True)
x.grad
```

In [82]:
```
y = 2 * torch.dot(x, x)
y
```

Out[82]:
```
tensor(28., grad_fn=<MulBackward0>)
```

In [83]:
```
y.backward()
x.grad
```

Out[83]:
```
tensor([ 0.,  4.,  8., 12.])
```

In [84]:
```
x.grad == 4 * x
```

Out[84]:
```
tensor([True, True, True, True])
```

In [85]:
```
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

Out[85]:
```
tensor([1., 1., 1., 1.])
```

In [86]:
```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))
x.grad
```

Out[86]:
```
tensor([0., 2., 4., 6.])
```

In [87]:
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

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

Out[87]:
```
tensor([True, True, True, True])
```

In [88]:
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

Out[88]:
```
tensor([True, True, True, True])
```

In [89]:
```
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 [90]:
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [91]:
```
a.grad == d / a
```

Out[91]:
```
tensor(True)
```

After running the function for backpropagation, immediately run it again and see what happens. Investigate.

In [92]:
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In the control flow example where we calculate the derivative of d with respect to a, what would happen if we changed the variable a to a random vector or matrix. At this point, the result of the calculation f(a) is no longer a scalar. What happens to the result? How do we analyze this?

In [93]:
```
a = torch.randn(size=(2, 2), requires_grad=True)
d = f(a)
```

In [94]:
```
print(d.sum().backward())
```

```
None
```

3.1

In [95]:
```
%matplotlib inline
import math
import time
import numpy as np
from torch import nn
from d2l import torch as d2l
```

In [96]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [97]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

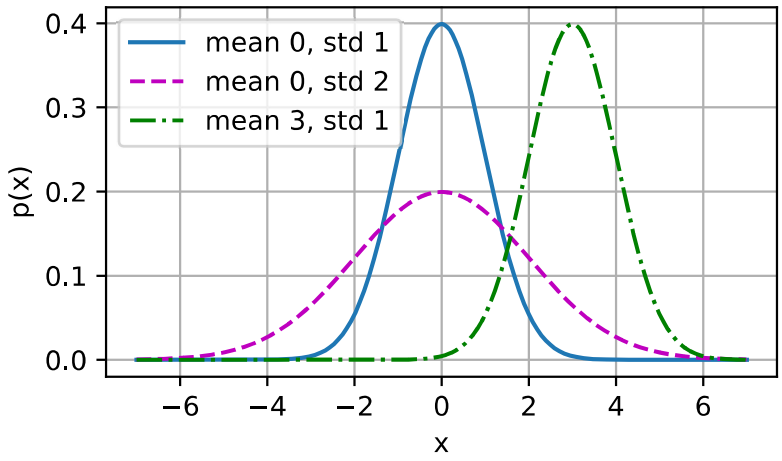
Out[97]: '0.14156 sec'

In [98]:
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[98]: '0.00100 sec'

In [99]:
```
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 [100…
```
# 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])
```



This was more interesting than I thought initially

With the likelihood image and the prediction image the negative log-likelihood should be image. This is kind of a closed-form solution that should be easily vectorizable.

I could not find a closed-form analytical solution to the gradient in respect to the weights to solve image though. The farthest I got was

image image 761×96 3.31 KB

where image are the residuals. The gradient is therefore not a smooth function and has jumps where the residual changes its sign (i.e. where the prediction exactly hits a data point to be predicted). The problem with stochastic gradient descent (and probably to some extent also to the case where all the data is used) could be that near the stationary point the gradient would jump by flipping the sign of a single vector image and never converge. I don't know what is done in this case, but I could imagine that increasing the batch size when the optimization comes closer to a stationary point could be useful, or reduction of the learning rate over time, but this would be certainly a parameter to tune.

I would try also to average the gradient between minibatches, e.g. so that the gradient used to update the parameters consists of some small constant image times the gradient of the current batch plus image times the gradient of the previous batch. Unfortunately, this would make parallelization over batches difficult.

3.2

In [101…
```
def add_to_class(Class):  #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [102…
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [103…
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

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

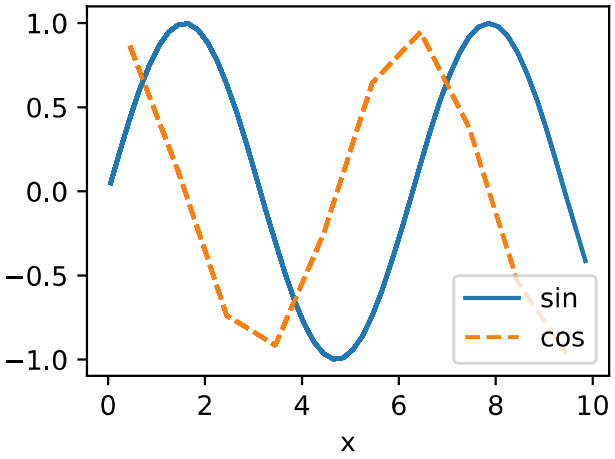
In [104…
```
class HyperParameters:  #@save
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

In [105…
```
# 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
```

In [106…
```
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', '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 [107…
```
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 [108…
```
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 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 [109…

```
class DataModule(d2l.HyperParameters):  #@save
    """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 [110…

```
class Trainer(d2l.HyperParameters):  #@save
    """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
```

exercises:

- In the D2L library, the save_hyperparameters method simplifies the process of saving constructor arguments by automatically assigning them as instance attributes. By removing this method, parameters like a and b are no longer automatically accessible via self.a or self.b. Without manual assignment, trying to print these values would result in an AttributeError. The reason is that save_hyperparameters

internally captures all arguments passed during initialization and saves them to the instance. Without this automation, explicit assignment is necessary to access them later. Thus, save_hyperparameters ensures code conciseness and prevents potential errors.

3.4

In [111…
```
%matplotlib inline
```

In [112…
```
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 [113…
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

In [114…
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

In [115…
```
class SGD(d2l.HyperParameters):  #@save
    """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_()
```

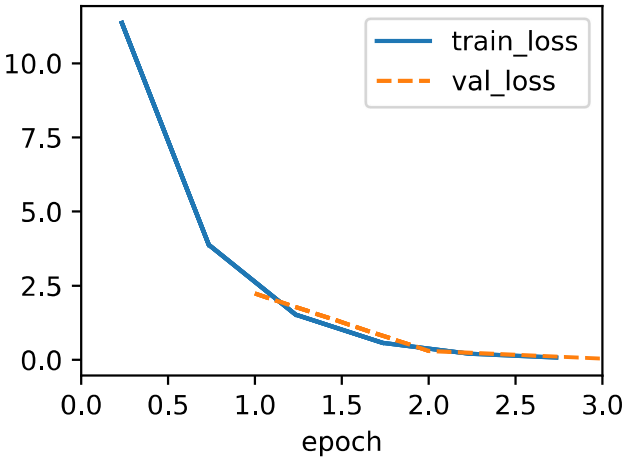
In [116…
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In [117…
```
@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:  # 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
```

In [118…

```
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 [119…

```
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([ 0.1188, -0.1669])
error in estimating b: tensor([0.2075])
```

1000 0.01

w = torch.zeros(size=(2,1), requires_grad=True) w nn([[0.], [0.]], requires_grad=True)

lr = 0.03 # Learning rate num_epochs = 3 # Number of iterations net = linreg # Our fancy linear model loss = squared_loss # 0.5 (y-y')^2

for epoch in range(num_epochs): # Assuming the number of examples can be divided by the batch size, all # the examples in the training data set are used once in one epoch # iteration. The features and tags of mini-batch examples are given by X # and y respectively for X, y in data_iter(batch_size, features, labels): l = loss(net(X, w, b), y) # Minibatch loss in X and y l.mean().backward() # Compute gradient on l with respect to [w,b] sgd([w, b], lr, batch_size) # Update parameters using their gradient with torch.no_grad(): train_l = loss(net(features, w, b), labels) print(f'epoch {epoch+1}, loss {float(train_l.mean())}')

Assume that you are Georg Simon Ohm trying to come up with a model for resistance that relates voltage and current. Can you use automatic differentiation to learn the parameters of your model?

- set y as voltage and set x as current.

What are the problems you might encounter if you wanted to compute the second derivatives of the loss? How would you fix them?

y.backward(retain_graph=True)

Why is the reshape method needed in the loss function?

- true_w has one row and len(w) columns, but w has len(w) rows and one column.

Experiment using different learning rates to find out how quickly the loss function value drops. Can you reduce the error by increasing the number of epochs of training?

- set lr = your_num

If the number of examples cannot be divided by the batch size, what happens to data_iter at the end of an epoch?

- In the last loop of for i in range(0, num_examples, batch_size): : j = torch.tensor(indices[i: num_examples)

4.2

In [120…

```
import torchvision
from torchvision import transforms

d2l.use_svg_display()
```

In [121…
```
class FashionMNIST(d2l.DataModule):  #@save
    """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)
```

In [122…
```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

Out[122… (60000, 10000)

In [123…
```
data.train[0][0].shape
```

Out[123… torch.Size([1, 32, 32])

In [124…
```
@d2l.add_to_class(FashionMNIST)  #@save
def text_labels(self, indices):
    """Return text labels."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

In [125…
```
@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 [126…
```
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 [127…
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

Out[127… '6.06 sec'

In [128…
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    raise NotImplementedError
```

In [129…
```
@d2l.add_to_class(FashionMNIST)  #@save
def visualize(self, batch, nrows=1, ncols=15, 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)
```

The question is about the efficiency of loading data, so it's better to focus on the underlying computer system rather than methods like mini-batch SGD or SGD, which are primarily used to speed up training on GPUs. In fact, mini-batch SGD is typically employed to parallelize computations on GPUs, but if you have enough TPUs, you could even use a batch size of 1 across the entire dataset. Here's a simplified explanation, based on a few assumptions since I'm not entirely familiar with Nvidia GPU or TPU architecture: Let's assume that the I/O process in the system involves techniques such as interrupts, Direct Memory Access (DMA), and channels, and that some buffers are used for temporary data storage. When the buffer reaches its capacity, any of these methods—interrupts, DMA, or channels—must handle the "interrupt processing." This shifts control from the DMA or channel to the CPU/GPU/TPU, causing the I/O operation to pause. Reducing the batch size increases the frequency of these pauses, leading to more frequent interruptions and ultimately decreasing I/O efficiency.timer = d2l.Timer() train_iter, test_iter = load_data_fashion_mnist(32, resize=64) for X, y in train_iter: print(X.shape, X.dtype, y.shape, y.dtype) break f'{timer.stop():.2f} sec'timer = d2l.Timer() train_iter, test_iter = load_data_fashion_mnist(1, resize=64) for X, y in train_iter: print(X.shape, X.dtype, y.shape, y.dtype) break f'{timer.stop():.2f} sec'

4.3

In [142…

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

In [143…

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

@d2l.add_to_class(Classifier) #@save 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

1-

In [146…

```
N = 1000
M = 100
num_minibatches = N // M

minibatch_losses = np.random.rand(num_minibatches)

L_val = np.mean(minibatch_losses)
L_minibatch = minibatch_losses[-1]

L_val, L_minibatch
```

Out[146… (0.5802451893134991, 0.763006688776522)

2-

In [147…
```
def compute_validation_loss(true_loss, num_minibatches):
    minibatch_losses = np.random.normal(true_loss, 0.1, num_minibatches)
    return np.mean(minibatch_losses)

true_val_loss = 0.5
estimates = [compute_validation_loss(true_val_loss, num_minibatches) for _ in ra

expected_val_loss = np.mean(estimates)

expected_val_loss, true_val_loss
```

Out[147… (0.4998161595093477, 0.5)

3-

In [148…
```
p_c_given_x = np.array([0.2, 0.5, 0.3])

loss_matrix = np.array([[0, 1, 2],
                        [1, 0, 1],
                        [2, 1, 0]])

expected_loss = np.dot(p_c_given_x, loss_matrix)

optimal_class = np.argmin(expected_loss)

expected_loss, optimal_class
```

Out[148… (array([1.1, 0.5, 0.9]), 1)

4.4

In [149…
```
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[149…
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

In [150…
```
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
```

In [151…
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

Out[151…
```
(tensor([[0.2168, 0.2008, 0.1432, 0.2360, 0.2033],
         [0.2191, 0.1577, 0.1266, 0.2683, 0.2283]]),
 tensor([1., 1.]))
```

In [152…
```
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 [153…
```
@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 [154…
```
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]
```

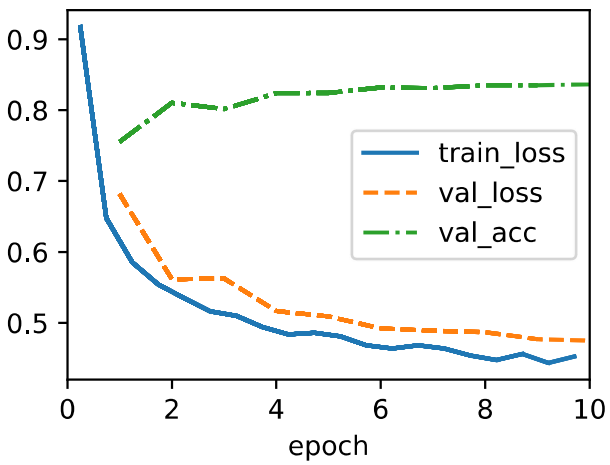
Out[154…
```
tensor([0.1000, 0.5000])
```

In [155…
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[155…
```
tensor(1.4979)
```

In [156…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

In [157…
```
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 [158…
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[158…
```
torch.Size([256])
```

In [159…
```
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)
```

Discussions

In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause? Hint: try to calculate the size of exp(50).

In [160…
```
X = torch.tensor([50])

torch.exp(X)
```

Out[160…
```
tensor([5.1847e+21])
```

5.1847e+21 is a large number (not near the float limit, but still not so far away), so when calculating things with this number it is possible for the result to overflow.

The function cross_entropy in this section was implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation? Hint: consider the domain of the logarithm. The domain of the logarithm are all numbers greater than 0. The problem would arise if some probability would be 0.

What solutions you can think of to fix the two problems above? For question 1, I would always check what the function returns to see if the numerical overflow occured. Maybe there's a better solution, but this one seems good. Or I'd check for numerical overflow in the function itself, just prior to returning the result.

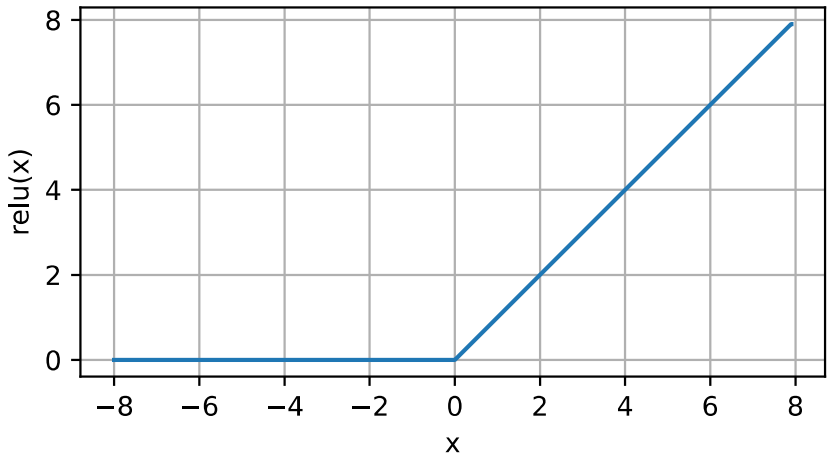
For question 2, I could add a tiny constant epsilon (such as 0.001 or something) to every probability, so that even if probability of some class is indeed 0, it would then be equal to epsilon.

Is it always a good idea to return the most likely label? For example, would you do this for medical diagnosis? For a medical diagnosis, I would probably not return the most likely label by itself, but rather the entire probability distribution over the outcomes and have a medical professional check it.

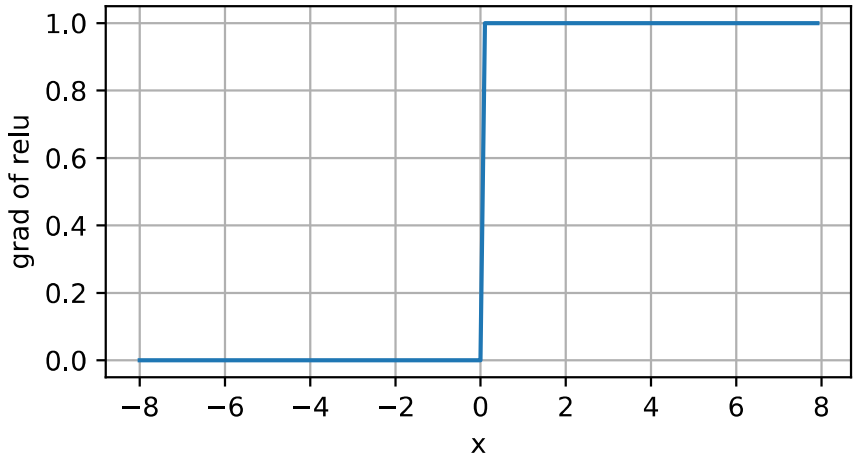
Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary? If I have a large vocabulary, then I might have a situation where there's a lot of possible next words and since there's a lot of them, the total probability of 1 would have to be distributed among them all. That would be worse than me narrowing down the probability to a select fewer words.

5.1

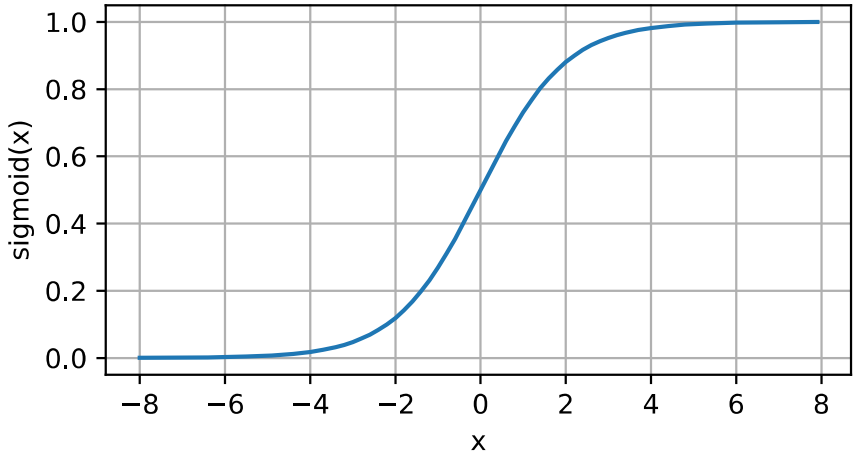
In [161…
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



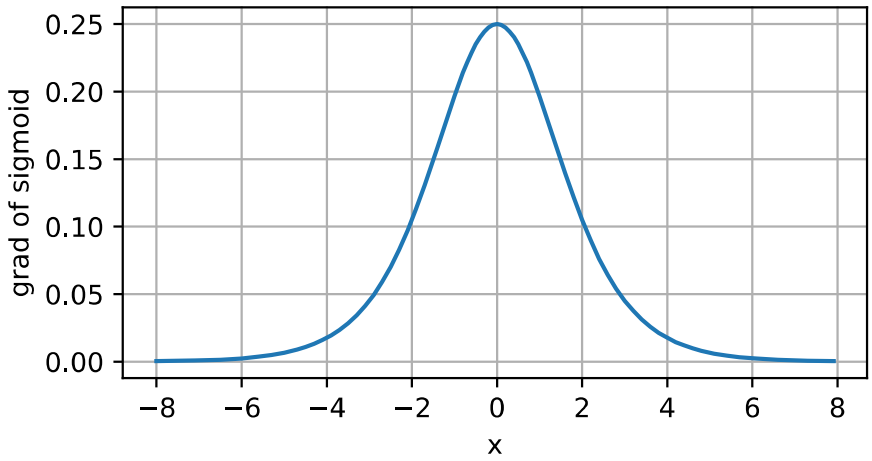
In [162…
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



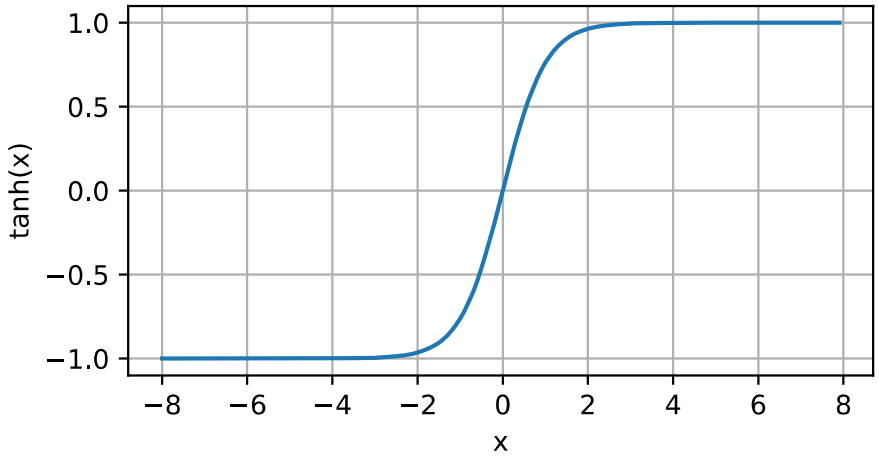
In [163…
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



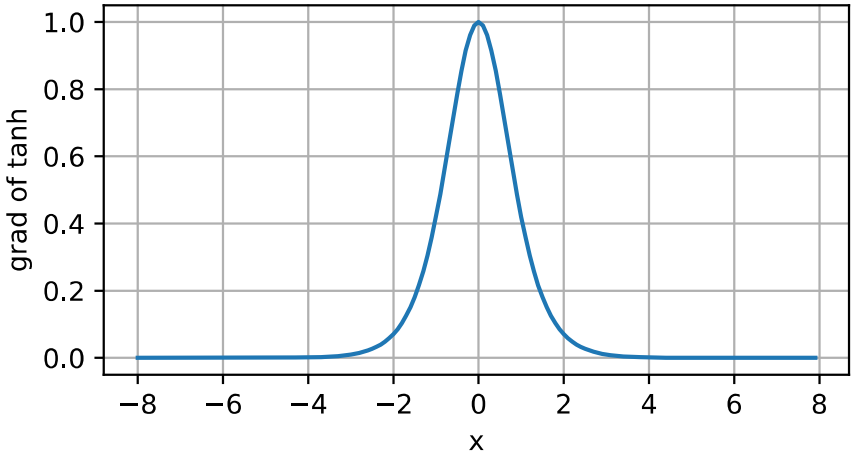
In [165…
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



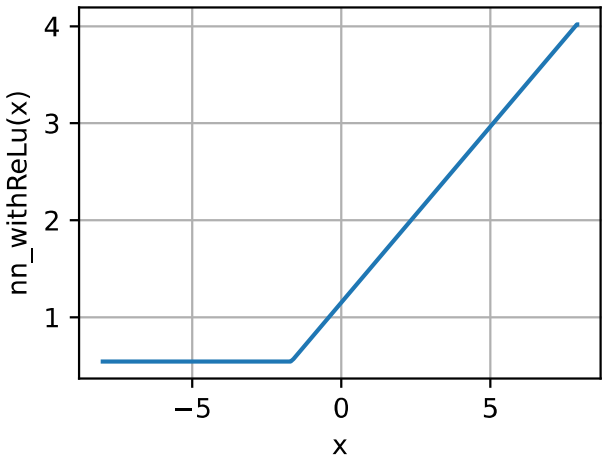
In [166…
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



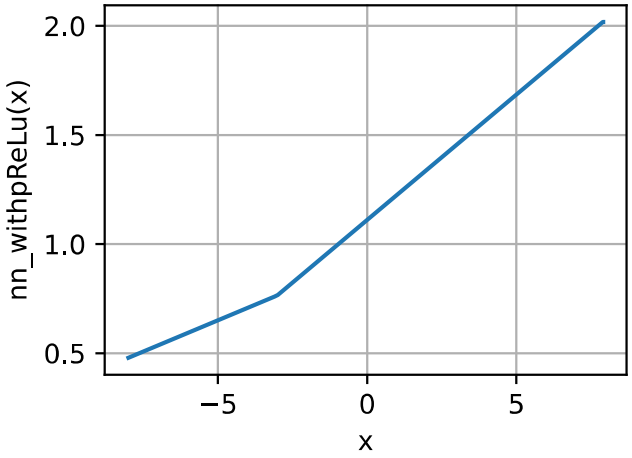
In [167…
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



In [169…
```
x_1 = torch.arange(-8.0, 8.0, 0.1)
w_1 = torch.rand(1, 1)
b_1 = torch.rand(1, 1)
y_1 = x_1*w_1+b_1
x_2 = torch.relu(y_1)
w_2 = torch.rand(1, 1)
b_2 = torch.rand(1, 1)
y_2 = x_2*w_2+b_2
d2l.plot(x.detach(), y_2.detach(), 'x', 'nn_withReLu(x)')
```



In [170…
```
x_1 = torch.arange(-8.0, 8.0, 0.1)
w_1 = torch.rand(1, 1)
b_1 = torch.rand(1, 1)
y_1 = x_1*w_1+b_1
x_2 = torch.prelu(y_1,torch.tensor(0.5))
w_2 = torch.rand(1, 1)
b_2 = torch.rand(1, 1)
y_2 = x_2*w_2+b_2
d2l.plot(x.detach(), y_2.detach(), 'x', 'nn_withpReLu(x)')
```



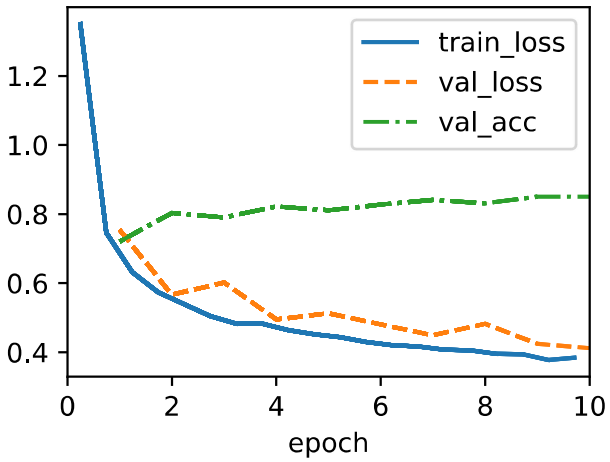
5.2

In [171…
```
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 [172…
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

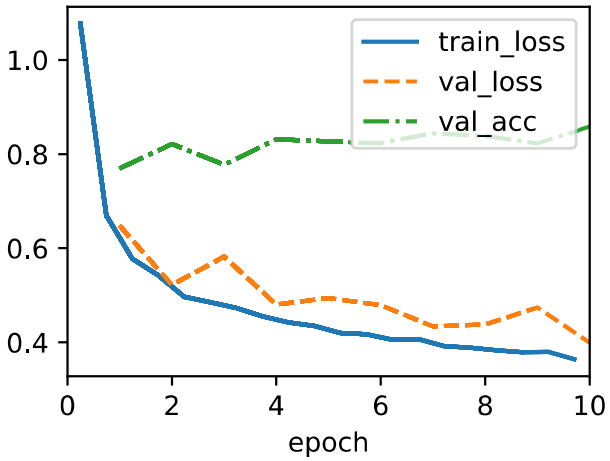
In [173…
```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

In [174…
```
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 [175…
```
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 [176…
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



In this section, we explored how to implement a simple Multilayer Perceptron (MLP) for the Fashion-MNIST dataset both from scratch and using high-level deep learning frameworks. The MLP consists of 784 input features, a hidden layer of 256 units, and 10 output classes. Implementing from scratch involves manually initializing weights and biases, defining the forward pass, and handling the ReLU activation function. While this approach gives detailed control, it can be tedious and error-prone, especially when scaling models.

The concise implementation using high-level libraries like PyTorch or TensorFlow abstracts much of the complexity. By using the Sequential class, layers can be stacked easily, and the model's forward pass is handled automatically. This makes model design simpler and more modular, allowing for faster experimentation and greater focus on architecture rather than low-level details. Furthermore, built-in optimizations provided by these libraries ensure efficient memory usage and

computational performance, particularly when working with GPUs.

Overall, while building MLPs from scratch offers valuable insight into neural networks, high-level frameworks significantly ease the process of model development and training. They make it easier to scale and experiment with larger models and more complex architectures, which is crucial as neural networks grow in size and complexity. Understanding both approaches helps deepen knowledge of model mechanics and optimization techniques in modern deep learning.

5.3

Gradient Dimensionality

If the inputs to a scalar function f f are $n \times m$ n×m matrices, the gradient of f f with respect to the input matrix X X is an $n \times m$ n×m matrix. The gradient's dimensionality matches that of the input matrix since each element in the input influences the scalar output.

Add Bias to the Hidden Layer

In the MLP model described earlier, adding a bias term to the hidden layer means we add a vector b 1 b 1of size equal to the number of hidden units (256 in this case). The forward propagation equation now becomes:

$$H$$

ReLU (X W 1 + b 1) H=ReLU(XW 1+b 1) where X X is the input, W 1 W 1is the weight matrix, and b 1 b 1is the bias vector. The bias ensures that even if all input features are zero, there can still be some activation in the hidden layer.

Computational Graph (with Bias)

The computational graph for this model consists of:

Input X X, Weight matrices W 1 W 1and W 2 W 2, Bias vectors b 1 b 1and b 2 b 2, The activation function (ReLU) applied to the hidden layer, Output layer computing class scores using W 2 W 2and b 2 b 2. The graph flows from inputs, through linear transformations (matrix multiplication + bias), non-linear activation (ReLU), and finally the output.

Memory Footprint for Training and Prediction

Training: Memory is used for storing:

Inputs and outputs for each layer, Weights and biases, Gradients of the loss with respect to the weights, Activations (needed for backpropagation). For a batch size B B, input size $n \times m$ n×m, hidden layer size 256 256, and output size 10 10, the memory footprint for training is proportional to $B \times (n \times m + 256 + 10)$ B×(n×m+256+10).

Prediction: Only the forward activations and model parameters are needed, so memory usage is lower compared to training (no gradient storage).

Computing Second Derivatives

When computing second derivatives, the computational graph becomes more complex, as each partial derivative needs to be differentiated again. This dramatically increases both the memory and time complexity. Specifically, second-order derivatives (like Hessians) require storing and computing gradients with respect to gradients, leading to quadratic growth in both time and space complexity.

Partitioning the Computational Graph Over Multiple GPUs

Yes, it is possible to partition the computational graph over multiple GPUs. This can be done either by:

Model Parallelism: Different parts of the model are distributed across GPUs (e.g., hidden layers on different GPUs). Data Parallelism: The same model is replicated on multiple GPUs, and the data is split across them. Each GPU computes gradients for its data shard, and these gradients are then averaged. Advantages:

Distributes memory load across GPUs, allowing larger models or batches. Potential for faster training by parallelizing computation. Disadvantages:

Communication overhead between GPUs, which may slow down training. More complex implementation, especially when splitting models rather than just data.

Training with Smaller Minibatches

Training with smaller minibatches reduces the memory footprint, allowing for training on limited GPU memory. However, it can introduce greater variance in gradient estimates, potentially slowing down convergence. Smaller batches may also underutilize the computational power of the GPU, leading to inefficiency.

Advantages:

Enables training on devices with less memory. Can provide better generalization due to noisy gradients.

Disadvantages:

Slower convergence due to higher gradient variance. Inefficiency on high-performance hardware due to underutilization.

In [ ]: